Francisco Hao

[公司名称][公司地址]

c#基础篇

目录

[第 1 章 Unity脚本开发环境介绍 3](#_Toc419015103)

[1.1 使用Microsoft Visual Studio 2012 编辑器来编写脚本 4](#_Toc419015104)

[1.2 创建第一个脚本 4](#_Toc419015105)

[1.2.1 \*print与log函数 6](#_Toc419015106)

[1.3 Console视图 7](#_Toc419015107)

[1.4 Unity 脚本的组成部分 8](#_Toc419015108)

[第 2 章 编程语言基础 9](#_Toc419015109)

[2.1 什么是编程 9](#_Toc419015110)

[2.2 程序的特征 9](#_Toc419015111)

[第 3 章 变量和常量 10](#_Toc419015112)

[3.1 变量与内存 10](#_Toc419015113)

[3.2 变量的声明和使用 10](#_Toc419015114)

[3.2.1 声明变量的语法 10](#_Toc419015115)

[3.2.2 多重变量声明 10](#_Toc419015116)

[3.2.3 变量初始化 11](#_Toc419015117)

[3.2.4 变量的命名规则 11](#_Toc419015118)

[3.3 变量种类 11](#_Toc419015119)

[3.4 常量 11](#_Toc419015120)

[3.5 转义字符 12](#_Toc419015121)

[3.6 变量作用域 14](#_Toc419015122)

[第 4 章 数据类型 14](#_Toc419015123)

[4.1 预定义类型 15](#_Toc419015124)

[4.2 用户自定义类型 16](#_Toc419015125)

[4.3 值类型和引用类型 16](#_Toc419015126)

[4.4 数据类型的分配与回收 20](#_Toc419015127)

[4.4.1 类型的内存分配 20](#_Toc419015128)

[4.4.2 托管类型与非托管类型的区别 21](#_Toc419015129)

[4.5 类型的大小 22](#_Toc419015130)

[4.6 类型转换 23](#_Toc419015131)

[4.6.1 隐式转换 25](#_Toc419015132)

[4.6.2 显示转换 27](#_Toc419015133)

[4.6.3 引用转换 30](#_Toc419015134)

[4.6.4 装箱与拆箱 32](#_Toc419015135)

[第 5 章 运算符与表达式 33](#_Toc419015136)

[5.1 什么是运算符、表达式？ 33](#_Toc419015137)

[5.2 运算符的本质 34](#_Toc419015138)

[5.3 赋值运算符 34](#_Toc419015139)

[5.4 算术运算符 35](#_Toc419015140)

[5.5 关系运算符 35](#_Toc419015141)

[5.6 逻辑运算符 36](#_Toc419015142)

[5.7 条件运算符 37](#_Toc419015143)

[5.8 递增和递减运算符 38](#_Toc419015144)

[5.9 typeof 运算符 39](#_Toc419015145)

[5.10 is 运算符 39](#_Toc419015146)

[5.11 as 运算符 40](#_Toc419015147)

[5.12 优先级顺序 41](#_Toc419015148)

[第 6 章 流程控制语句 42](#_Toc419015149)

[6.1 什么是语句 42](#_Toc419015150)

[6.2 控制流语句 43](#_Toc419015151)

[6.3 条件语句 43](#_Toc419015152)

[6.3.1 if 语句 43](#_Toc419015153)

[6.3.2 if…else语句 44](#_Toc419015154)

[6.3.3 if…else if…else语句 45](#_Toc419015155)

[6.3.4 switch语句 46](#_Toc419015156)

[6.4 循环语句 47](#_Toc419015157)

[6.4.1 while循环 48](#_Toc419015158)

[6.4.2 do-while循环 49](#_Toc419015159)

[6.4.3 for循环 49](#_Toc419015160)

[6.4.4 foreach-in循环 51](#_Toc419015161)

[6.5 跳转语句(循环中断) 52](#_Toc419015162)

[6.5.1 break语句 52](#_Toc419015163)

[6.5.2 continue语句 53](#_Toc419015164)

[6.5.3 Return语句 53](#_Toc419015165)

[6.5.4 标签和goto语句 53](#_Toc419015166)

[第 7 章 字段 54](#_Toc419015167)

[7.1 字段的声明 54](#_Toc419015168)

[7.2 字段的初始化 54](#_Toc419015169)

[7.3 Unity中公有字段的特殊性 55](#_Toc419015170)

[7.4 字段的作用 55](#_Toc419015171)

[7.5 字段的数据类型 56](#_Toc419015172)

[第 8 章 方法 56](#_Toc419015173)

[什么是方法 56](#_Toc419015174)

[8.1 局部变量及其有效范围 57](#_Toc419015175)

[8.2 方法的调用 58](#_Toc419015176)

[8.3 返回值 58](#_Toc419015177)

[8.4 参数 59](#_Toc419015178)

[8.4.1 形参 59](#_Toc419015179)

[8.4.2 实参 59](#_Toc419015180)

[8.5 值参数和引用参数 60](#_Toc419015181)

[8.5.1 值参数 60](#_Toc419015182)

[8.5.2 引用参数 61](#_Toc419015183)

[8.5.3 总结 61](#_Toc419015184)

[8.6 可变参数（params） 61](#_Toc419015185)

[8.7 out参数 63](#_Toc419015186)

[8.8 参数数组 65](#_Toc419015187)

[8.9 递归 66](#_Toc419015188)

[8.10 方法重载 66](#_Toc419015189)

[第 9 章 数组 67](#_Toc419015190)

[9.1 数组的声明 69](#_Toc419015191)

[9.2 数组实例化 69](#_Toc419015192)

[9.3 数组初始化 69](#_Toc419015193)

[9.4 数组元素的总数 70](#_Toc419015194)

[9.5 访问数组元素 70](#_Toc419015195)

[9.5.1 数组的遍历 70](#_Toc419015196)

[9.6 修改数组元素的值 71](#_Toc419015197)

[第 10 章 字符串 71](#_Toc419015198)

[10.1 修改字符串内容 71](#_Toc419015199)

[10.2 修改某一个字符的值 72](#_Toc419015200)

[10.3 替换字符串 72](#_Toc419015201)

[10.4 子字符串 73](#_Toc419015202)

[10.5 Contains、StartWith与EndsWith 73](#_Toc419015203)

[10.6 获取子字符串的位置 74](#_Toc419015204)

[10.7 Split分割字符串 75](#_Toc419015205)

[10.8 Trim去空格 76](#_Toc419015206)

[10.9 综合案例 76](#_Toc419015207)

[10.10 常用属性与方法 77](#_Toc419015208)

[10.10.1 Length 77](#_Toc419015209)

[10.10.2 IsNullOrEmpty 77](#_Toc419015210)

[10.10.3 ToUpper 78](#_Toc419015211)

[10.10.4 Equals 78](#_Toc419015212)

[10.10.5 IndexOf 79](#_Toc419015213)

[10.10.6 Substring 79](#_Toc419015214)

[10.10.7 Split 79](#_Toc419015215)

[10.10.8 Join 80](#_Toc419015216)

[10.10.9 Format 80](#_Toc419015217)

[10.10.10 replace 80](#_Toc419015218)

# Unity脚本开发环境介绍

Unity的脚本是整个项目的开发的关键要素，对象之间的任何逻辑判断都需要通过脚本来完成。如果说贴图、模型等资源的好坏决定一个项目的视觉品味，那么脚本将直接决定这个项目的内在质量，决定了这个项目的流畅性。

Unity一共支持3种语言来编写脚本，分别是JavaScript、C#和Boo，这三种语言不分好坏，用哪一种编写都可以达到同样的目的。本书主要讲解的是如何使用C#来编写脚本，因为C#语言在编程思想上更符合Unity引擎的原理。

## 使用Microsoft Visual Studio 2012 编辑器来编写脚本

Unity可以部署在windows与Mac OS X 两种操作系统下，它自带了一个跨平台的脚本编辑器MonoDevelop。MonoDevelop脚本编辑器使用起来非常方便，可以同时使用Javascript、C#和Boo三种语言来编写脚本。

而Microsoft Visual Studio 2012是微软公司开发的编译器，它的功能非常强大。在windows平台下，开发基于C#语言的脚本时，Microsoft Visual Studio 2012用气来非常的方便、专业。

本书的主要学习环境是在windows平台下，使用Microsoft Visual Studio 2012来进行脚本编程的学习的。在Mac OS X平台下学习的读者，可以采用MonoDevelop来进行脚本的编写。尽管编辑器有不同，但是C#的语法在使用上没有任何不同。

安装Unity时，会默认安装MonoDevelop编辑器，不需要再进行安装。而Microsoft Visual Studio 2012需要下载并安装到计算机上，才能使用。本书的光盘中有Microsoft Visual Studio 2012免费版。安装方法请自行查找相关的教程。

安装好Microsoft Visual Studio 2012后，想要使用它来编写脚本，还需要修改下默认的脚本打开方式。具体操作如下：

* 在Unity编辑器界面上方的菜单栏中点击“Edit”→“Preferences…”菜单项，打开“Unity Preference”界面。
* 选择左侧的“External Tools”菜单项，然后在“External Script Editor”下拉列表中选择“Browse…”选项。
* 在打开的“Browse for application”界面中，找到Microsoft Visual Studio 2012根目录的“Common7/IDE”文件夹，然后选择可执行文件“devenv”。

## 创建第一个脚本

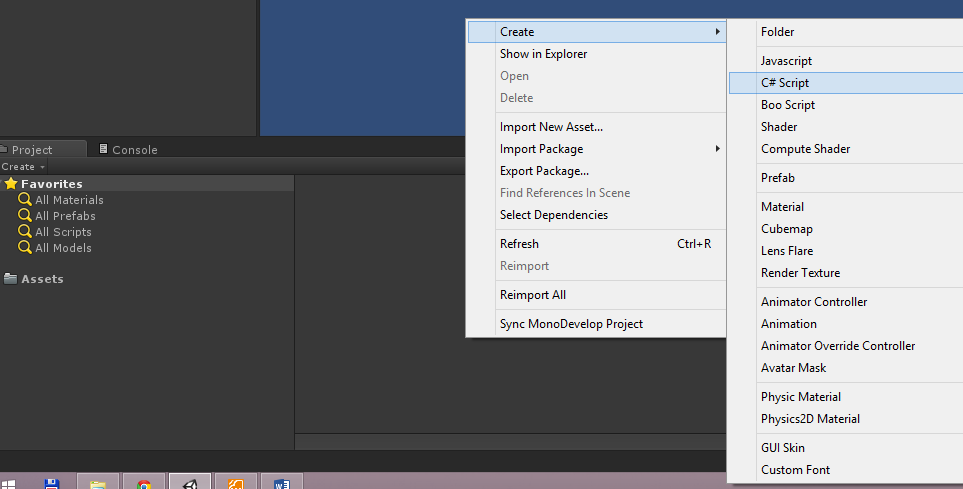
在一般的开发环境中，代码写完后，直接就可以进行编译，但是Unity的脚本不能执直接编译，需要把脚本文件绑定到组件上才能进行编译。

Unity脚本开发的步骤如下：

* 创建脚本文件。
* 编写脚本。
* 绑定脚本文件到组件上。
* 点击“play”按钮，运行脚本。

现在，我们通过一个简单的例子来学习如何使用Unity进行脚本的开发。

首先，在“project”视图右侧的面板上右击，选择“create”→“C# Script”创建一个脚本。如图：



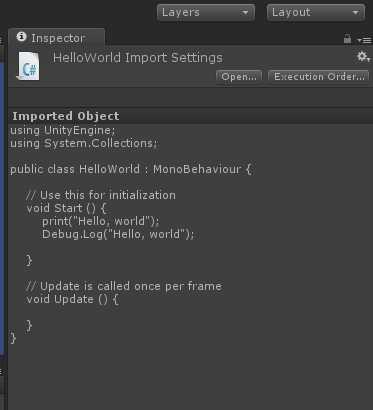
然后，修改脚本文件的名称为“HelloWorld”。如图：



双击脚本文件，会在Microsoft Visual Studio 2012中打开该脚本文件。我们可以在脚本文件中进行编写了。修改完成后的代码如下：

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class HelloWorld : MonoBehaviour {  // Use this for initialization  void Start () {  print("Hello, world");  Debug.Log("Hello, world");    }    // Update is called once per frame  void Update () {    }  } |

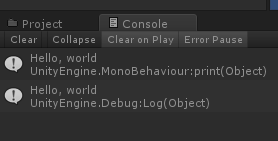
代码编写完成后，在Microsoft Visual Studio 2012中保存一下该文件。然后切换到Unity编辑器，在右侧的“Inspector”视图中，可以看到脚本文件被修改后的效果。如图：



之后，鼠标拖动“HelloWorld.cs”到“Hierarchy”视图的“Camera”对象上，进行绑定。脚本文件作为Camera对象的组件存在。



最后，点击“play”按钮，在“console”视图中，可以看到脚本运行的结果。



### \*print与log函数

Print与log函数用来在console中输出信息。

Print函数继承自MonoBehaviour，它只有一个参数，这个参数可以是对象（object）的名称或者字符串，如果是字符串则可以使用富文本的形式。

Console函数继承自UnityEngine，除了print中的使用形式外，它还能有2个参数。

下面为print与log函数的使用方法：

|  |  |
| --- | --- |
| using UnityEngine;  using System.Collections;  public class zyf : MonoBehaviour {  // Use this for initialization  void Start () {  print(gameObject);//可以是对象的信息  print("gameObject");  print("<color=red>gameObject</color>");//字符串可以使用富文本  Debug.Log("hello",gameObject);//log函数可以有两个参数，第一个参数为显示的信息，第二个参数为第一个参数的对象，即第一个参数是第二个参数要显示的信息    }  } |  |

## Console视图

在上面的例子中，我们知道Print的方法以及Debug.Log方法[[1]](#footnote-1)会在console视图中输出打印结果。除了输出相关信息外，console视图还能对脚本中错误、不安全或冗余的代码提出警告或错误提示。

例如，在新建的脚本文件的Start方法上方添加语句“int i = 2”。

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class HelloWorld : MonoBehaviour {  int i = 2;  void Start ()  {  }  void Update ()  {  }  } |

把该脚本保存后返回Unity，在console视图中可以看到类似如下的以黄色三角形开始的信息。如果我们忽略这个警告，该脚本仍然能运行成功。



接着我们在“int i = 2”语句下添加一行代码：“int b = i；”。保存脚本后返回Unity，看看发生了什么？我们会发现在console视图中多出了类似以下的消息：



以红色六边形开始的信息是一个错误提示。当出现错误提示时，脚本就不能运行成功了，必须改正错误才能运行脚本。我们在错误提示上双击，就会切换到Microsoft visual studio 2012并跳到出现错误的语句上。

## Unity 脚本的组成部分

本书主要研究的是Unity脚本，会广泛的使用代码示例，所以我们先来看看Unity脚本的样子，还有它的不同部分代表什么意思。

让我们从观察下面的简单的脚本开始，逐个解释它的各个组成部分。这个脚本包含在一个名为“HelloWorld.cs”的文本文件里。

|  |
| --- |
| **引用命名空间UnityEngine和System.Collections。之后可以直接使用这两个命名空间中的类库**  using UnityEngine;  using System.Collections;  **声明一个命名空间**  namespace My  {  **声明一个继承自MonoBehavior 的类**  public class HelloWorld : MonoBehaviour  {  **声明一个字段**  public string str = "HelloWorld";  /// <summary>  /// Use this for initialization  **块注释**  /// </summary>  void Start()  {  print("Hello, world");  **声明一个方法**  Debug.Log("Hello, world");  }  **行注释**  // Update is called once per frame  void Update()  {  **声明一个方法**  }  }  } |

C#脚本是一种类型声明。完整的脚本是由一个或多个类型组成的。比如上面的例子中，有一个名为HelloWorld的类型声明，它包含了一个字段和两个方法。

此外，Unity4.3支持命名空间。通过命名空间，可以把多个类组织在同一起，方便第三方使用[[2]](#footnote-2)。

# 编程语言基础

## 什么是编程

编程就是告诉计算机要做什么。计算机没有意识，它不知道自己应该做什么，需要你给它下达指令，才能做事。

|  |
| --- |
| 术语 |
| * 指令（instrction）: 就是下达给计算机的一个基本命令，通常要求计算机做某件特定的事情 | |

多个相关的指令组成程序，多个程序集合到一起构成了软件，软件就是运行在计算机上的程序。例如，一个简单的计算器软件，它由加减乘除等指令构成，可以进行计算的操作。

## 程序的特征

我们知道程序就是下达给计算机的一系列指令。它可以分为输入（input）、处理（process）输入以及输出（output）。

还以上面的计算器为例，它就具备了这3个基本要素。

输入：键入的数字与计算方式

处理：程序检查键入的数字与计算方式，并计算出结果

输出：程序最后输出处理的结果。

下面再看一个例子，这个程序也具备所有这三个基本要素：在一个赛车游戏中，输入是来自手柄、键盘或触摸屏手势传来的信号，处理是判断赛车是加速、减速、变向或者其他的活动。输出是在屏幕上显示的图像和扬声器或耳机传出的声音。

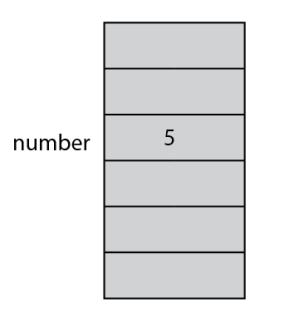
# 变量和常量

为了处理输入，计算机把输入的数据以及程序本身保存在它的内存（memory）中。但是我们怎么能知道输入的数据是放在内存中哪里呢？另外，由怎么能把它找出来呢？而这个是通过变量来实现的。

## 变量与内存

变量是一个名称，它好像内存的一个标签。实质上是内存中的一段存储空间(常被称为数据的容器)。

例如： int number = 5； 会在空闲的内存中存入数字“5”，这段内存的名字叫做number，当我们使用数字时，不需要知道内存在哪里，只需要使用它的名字number就可以调用数据。



## 变量的声明和使用

### 声明变量的语法

在C#中声明变量的语法格式如下：

|  |
| --- |
| 类型 变量名称；  例如： int a；  声明了一个名称为a的变量，数据类型为 int。 |

### 多重变量声明

可以把多个变量声明在一条单独的声明语句中。多重变量声明中的变量必须类型相同，变量名用逗号隔开。

多重变量声明语法如下：

|  |
| --- |
| int a, b, c = 4, d;//声明多重变量，有的初始化，有的没有初始化 |

### 变量初始化

在声明变量时，还可以用变量初始化语句把它的内存初始化为一个明确的值。

变量初始化语句由一个等号后面跟一个初始值构成，如：

|  |
| --- |
| int a = 11; |

一些类型的变量在声明时没有初始化语句，那么它会被设为默认值，而另一些则不能。不能初始化的变量在程序为其赋值前包含的为undefined值。下表展示了哪种类型的变量会自动初始化以及哪种类型的变量不会被初始化。

|  |  |  |  |
| --- | --- | --- | --- |
| 变量类型 | 存储位置 | 自动初始化 | 用途 |
| 局部变量 | 栈或者栈或堆 | 否 | 用于函数成员部分的计算 |
| 类字段 | 堆 | 是 | 类的成员 |
| 结构字段 | 栈或堆 | 是 | 结构的成员 |
| 参数 |  | 否 | 用于把值传人或传出方法 |
| 数组元素 | 堆 | 是 | 数组的成员 |

### 变量的命名规则

第一个字符必须是字母或下划线，其后的字符可以是任意个数字、字母、下划线。不能使用C#的关键字，比如class、new 、void等。

C#支持中文变量名。

## 变量种类

C#提供了四种变量，我们将会在以后的章节中进行详细的介绍

|  |  |
| --- | --- |
| 名字 | 描述 |
| 局部变量（Local variable） | 在方法的作用域内保存临时数据，不是类型的成员 |
| 字段（Field） | 保存与类型或类型实例相关的数据，是类型的成员 |
| 参数（Parameter） | 用于从一个方法到传递数据另一个方法，不是类型成员 |
| 数组元素（Array element） | 数组的数据项中的成员，可以是类型成员或局部成员。 |

## 常量

与变量一样，常量也是一个被命名的内存位置，但是它的值在程序运行过程中是不会发生改变的。

在C#中，依据其声明的位置不同，常量分为字段常量和局部常量两种。常量的声明方式与字段或变量的声明相似。区别在于：

* 常量声明需要在类型之前增加const关键字
* 必须有初始化语句。初始化值在编译时决定，在程序运行过程中是不会发生改变的。初始化值通常是一个预定义简单类型或由其组成的表达式。它还可以是null引用，但它不能是某对象的引用，因为对象的引用是在运行时决定的。

在下面的示例中，声明了两个常量，一个是字段常量，一个是局部常量。

|  |
| --- |
| public class NewBehaviourScript1 : MonoBehaviour {  const int i = 4;//声明字段常量  void Start()  {  const int b = 4;//声明局部常量  }  } |

用const定义的常量，对于所有类对象而言都是一样的，需要像访问静态成员（11.6.2静态成员）那样去访问const定义的常量，而用对象的成员方式去访问会出变异错误。此外，对于静态常量的访问在编译的时候，是用常量的值去替换常量。也就是说不需要实例化对象就能直接访问常量。

|  |
| --- |
| public class calc  {  public const float PI = 3.14f;    }  class Program  {  void Start()  {  float area = 2 \* calc.PI;//不需要实例化calc对象，就能直接访问到PI常量  print(area);  } |

## 转义字符

转义字符“\” 是一种特殊的字符常量。以反斜线"\"开头，后跟一个或几个字符。具有特定的含义，不同于字符原有的意义，故称“转义”字符。主要用来表示那些用一般字符不便于表示的控制代码。它的作用是消除紧随其后的字符的原有含义。

用一些普通字符的组合来代替一些特殊字符，由于其组合改变了原来字符表示的含义，因此称为“转义”。

ASCII中有一些非打印字符, 像换行，退格等, 这些字符必须直接写入ASCII值才可以输出, 这些ASCII之间没有任何规律，可读性不高, 难于记忆, 为此人们发明了转义字符来代替ASCII值，用以摆脱ASCII的缺点，方便人们的使用。用可以看见的字符表示那不可以看见的字符, 如'\n'表示换行。

下表为常用的转义字符：

|  |  |
| --- | --- |
| 转义字符 | 意义 |
| \’ | 单引号符 |
| \” | 双引号符 |
| \\ | 反斜线符"\" |
| \0 | 结束符 |
| \a | 感叹号 |
| \b | 退格 |
| \f | 走纸换页 |
| \n | 换行 |
| \r | 回车 |
| \t | 横向跳到下一制表位置 |
| \v | 竖向跳格(垂直制表) |

下面的例子是转义字符的几个演示：

|  |  |
| --- | --- |
| using UnityEngine;  using System.Collections;  public class zyf : MonoBehaviour {  // Use this for initialization  void Start () {  string a = "a\nb";//换行 \n  string b = "a\tb";//制表符 \t  string c = "D:\\Unity\\unity.exe";//反斜线符 \\。  string d = @"D:\Unity\unity.exe";//@只对转义符起作用，表示\不当成转义符  print(a);  print(b);  print(c);  print(d);  }  } |  |

## 变量作用域

变量的作用域就是可以访问该变量的代码区域。变量的作用域就是可以访问该变量的代码区域。

可以通过下面的规则来确定变量的作用域：

* 只要字段所属的类在某个作用域内，其字段也在该作用域内。
* 局部变量存在于表示该变量的块语句或者方法结束的封闭花括号之前的作用域。
* 在for、while或类似语句中声明的局部变量存在于该循环体内。

通俗的讲，变量作用域就是离声明该变量最近的那对包含声明语句的大括号所确定的代码区域。

# 数据类型

数据是一切编程语言的基石。所有对象，比如数字、物体的名称、颜色、场景的大小等，都是通过数据来描述的。

计算机语言有成千上万种，但是绝大多数程序设计语言的目标只有一个：处理数据（查看2.2程序的特征）。但是数据本身也是多种多样的。例如在平时生活中我们常用整数来表达东西的数据，在工程、金融等领域往往需要用到很高精度的小数等。除了整型、实型等数值类型外，还包括字符、声音、图像、视频等非数值类型的数据。

数据是描述客观事物的符号，是对某些具有共性的事物的抽象（共性的集合）。而数据类型是一组性质相同的值的集合及定义在此集合上的一些操作的总称[[3]](#footnote-3)。

数据类型是按照值的不同进行划分的。在高级语言中，每个变量、常量和表达式都有各自的取值范围。类型就用来说明变量或表达式的取值范围和所能进行的操作。

当年那些设计计算机语言的人，为什么会考虑到数据类型呢？

比如，大家都需要住房子，也希望房子越大越好。但显然，没有钱，考虑房子是没什么意义的。于是商品房就出现了各种各样的房型，有别墅、有复式、有单间；有一百平米的，也有几十平米的，甚至还有胶囊公寓——只有两平米的房间，这样就满足了不同人的需要。

同样，在计算机中，内存也不是无限大的，你要计算1+1=2、4+5=9这样的整型数字的简单计算，显然不需要开辟很大的适合小数甚至字符运算的内存空间。于是计算机的研究者们就考虑，要对数据进行分类，分出多种数据类型。

在C#中，按照取值的不同，数据类型可以分为两类：

* 预定义类型：即基本类型，是不可以再分解的基本类型。
* 用户自定义类型：结构性的类型，由若干个类型组合而成，是可以再分解的。例如，整型数组是由若干整型数据组成的。

## 预定义类型

C#中有几种类型非常简单，它们被视为其他所有类型的基础，这些类型被称为预定义类型（predefined type）或者基本类型（primitive type）。

C#提供了15种预定义类型，其中包括13种简单类型和2种非简单类型

所有预定义类型的名称都由全小写的字母组成。预定义的简单类型包括：

* 11种数值类型
* 8种整数类型：sbyte、byte、short、ushort、int、unit、long、ulong。
* 3种浮点类型：float、double、decimal。
* 1种Unicode字符类型：char。
* 一种布尔类型：bool。

两种非简单类型是：

* 一个Unicode字符数组：string。
* 所有其他类型的基类： object。

下表列出了这些类型，并同时列出了它们的取值范围、默认值

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 类型 | 大小 | 范围 | 是否有符号 | 默认值 |
| Sbyte | 8bit | -128～127 | 是 | 0 |
| byte | 8bit | 0～255 | 否 | 0 |
| short | 16bit | -32 768～32 767 | 是 | 0 |
| ushort | 16bit | 0～65 535 | 否 | 0 |
| int | 32bit | -2 147 483 648～2 147 483 647 | 是 | 0 |
| uint | 32bit | 0～4 294 967 295 | 否 | 0 |
| long | 64bit | -9 223 372 036 854 775 808～  9 223 372 036 854 775 807 | 是 | 0 |
| ulong | 64bit | 0～18 446 744 073 709 551 615 | 否 | 0 |
| float | 32bit | ±1.5×10-45～±3.4×1038 | 是 | 0.0f |
| double | 64bit | ±5×10-324～±1.7×10308 | 是 | 0.0d |
| decimal | 128bit | ±1.0×1028～±7.9×1028 |  | 0m |
| bool | 布尔型（8bit） | true、false |  | false |
| char | 16bit的Unicode字符 | U+0000～U+FFFF |  | \x0000 |
| string | Unicode字符串 |  |  |  |
| object | 所有其他类型的基类 |  |  |  |

可以看到，这些数据类型除了名字不同，主要的区别就在于位数和数据的范围。由于这些区别，它们的用途不同。

例如，如果要编写一个程序来存储角色的资料，其中包括角色的等级，那么用byte类型是最好的，因为byte类型的范围是0～255，没有角色的等级在这个范围之外。虽然用ushort、ulong也是可以，但是它们所能表达的几万甚至几十亿的范围对于等级来说是没有必要的，况且，它们所需要的存储空间比byte类型多，ushort需要16位、ulong需要64位，而byte只需要8位。

再比如，如要要表示角色的血量，那么用byte就不合适了，这时就需要用ushort或者uint了，甚至需要使用ulong。

|  |
| --- |
| 趣事 |
| 鸟叔的《江南Style》2012年7月被上传至视频分享网站YouTube，点击量如火箭般蹿升。2014年12月4日，它点击量21.5亿次爆表。这已经超过了YouTube计数器的上限，为此，公司不得不紧急进行了升级。谷歌公司表示，在鸟叔之前，“我们从未想过一段视频的点击量会大于32位整数最大值(约为21.47亿)”。 | |

## 用户自定义类型

除了C#提供的预定义类型，还可以创建用户自定义类型，有6种类型可以由用户自己创建

* 类类型（class）
* 结构类型（struct）
* 数组类型（array）
* 枚举类型（enum）
* 委托类型（delegate）
* 接口类型（interface）

类型通过类型声明创建，例如

|  |
| --- |
| Class myClass {} |

一旦声明了类型，就可以创建并使用这种类型的对象。使用用户自定义类型需要2个步骤：先声明类型，然后实例化该类型的对象。具体的使用方法会在以后的章节中进行详细介绍。

## 值类型和引用类型

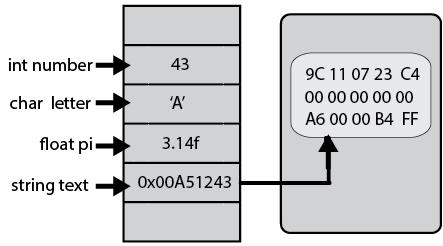
根据数据在内存中的存储方式不同，数据的类型又可以分为：值类型和引用类型。

* 值类型只需要一段单独的内存，用于存储实际的值
* 引用类型需要两段内存：
* 第一段存储实际的数据，它位于堆中。
* 第二段存储的是一个内存位置的引用（内存地址），要去那个位置才能找到真正的数据。

我们声明以下的变量类型

|  |
| --- |
| int number = 43；  char letter = ‘A’；  float pi = 3.14f；  string text = “This is funny”； |

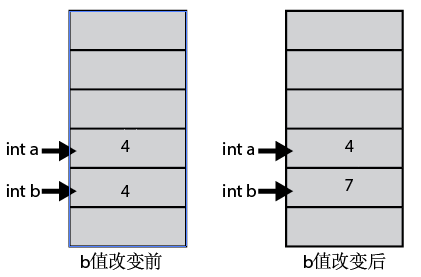
其中int、char、float为值类型，变量指向的内存中存储的是实际的数据；string为引用类型，变量指向的内存中存储是数据所在的地址，真正的数据位于堆中。如下图。



值类型和引用类型在使用上有显著的区别，以值类型int和引用类型int[ ]来做说明，先看值类型int的例子代码：

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript : MonoBehaviour  {  /// <summary>0x00A533C5  /// 值类型的例子  /// </summary>  void Start ()  {  int a = 4;//声明变量a，赋值为4.  int b = a;//声明变量b，并将a的值赋给b  b = 7;//改变变量b的值为7.  print("a的值是：" + a);  print("b的值是：" + b);  }  /\*  \* 输出结果：  \* a的值是4  \* b的值是7  \*/  } |

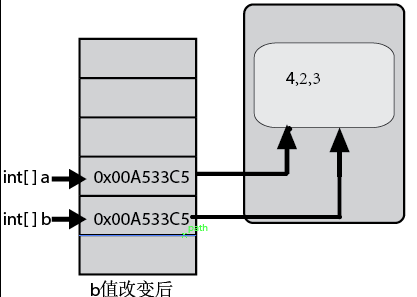
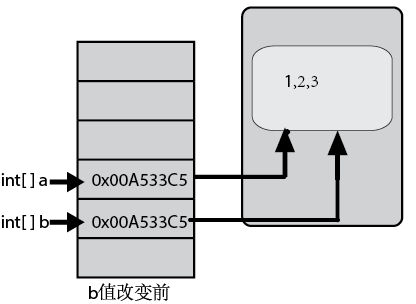
根据结果可以发现，把a的值赋给b后，改变了b的值，a的值并没有因为b值的改变而变化。



再看引用类型的情况，以int[ ]数组为例：

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript : MonoBehaviour  {  int y = 0;  /// <summary>  /// 引用类型的例子  /// </summary>  void Start()  {  int[] a = new int[] { 1, 2, 3 };//声明变量ar，新建一个数组[1,2,3]赋值给它。  int[] b = a;//声明变量b，把变量a的引用赋值给它  b[0] = 4;//改变b数组第一个元素的值为4.  //循环输出a的各个元素  foreach (int i in a)  {  print("a[" + y + "]的值是：" + i);  y++;  }  }  /\* 输出结果：  \* a[0]的值是4  \* a[1]的值是2  \* a[2]的值是3  \*/  } |

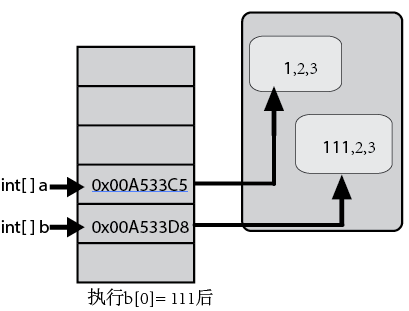
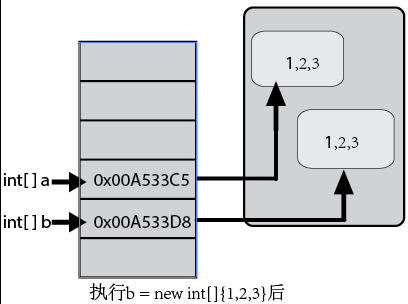
同样是把a的值赋给b，但b改变后，a值也改变了，原因就在于引用类型数据存储是引用。当“int[ ] b = a”这一局执行时，实际上是新建了一个数组变量b，并将a持有的引用（而不是值）赋值给了b。变量a和b同时指向了一个数组对象。因此任何一个变量做出了什么操作，另外一个变量也会受到影响。



再来看下面这种情况：

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript : MonoBehaviour  {  int y = 0;  int x = 0;  /// <summary>  /// 引用类型的例子  /// </summary>  void Start()  {  int[] a = new int[] { 1, 2, 3 };//声明变量ar，新建一个数组[1,2,3]赋值给它。  int[] b = a;//声明变量b，把变量a的引用赋值给它  b = new int[] {1, 2, 3 };//新建一个数组[1,2,3]，将引用赋值给b  b[0] = 111;  //循环输出a的各个元素  foreach (int i in a)  {  print("a[" + y + "]的值是：" + i);  y++;  }  //循环输出b的各个元素  foreach (int i in b)  {  print("b[" + x + "]的值是：" + i);  x++;  }  }  /\* 输出结果：  \* a[0]的值是1  \* a[1]的值是2  \* a[2]的值是3  \* b[0]的值是111  \* b[1]的值是2  \* b[2]的值是3  \*/  } |

我们看到a、b的值不互相干扰了。原因是什么？原来当“b = new int[ ]{1,2,3}”这一句执行时，用new关键字又创建了一个新的数组[1,2,3],并让b的引用指向了它。所以a，b的引用指向不再一样了，分别指向了内存中两个不同的数组。



下面图表中是C#中可以使用的所有类型以及他们的类别

|  |  |  |
| --- | --- | --- |
|  | 值类型 | 引用类型 |
| 预定义类型 | sbyte, byte, float, short, ushort, double, int, uint, char, long, ulong, decimal,  bool | object, string |
| 用户自定义类型 | struct, enum | class, interface, delegate, array |

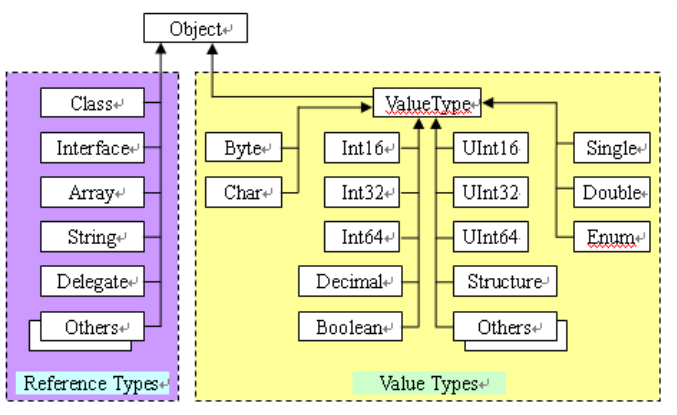
## 数据类型的分配与回收

### 类型的内存分配

C#中的所有类型都是（直接或间接）从System.Object类型派生的。

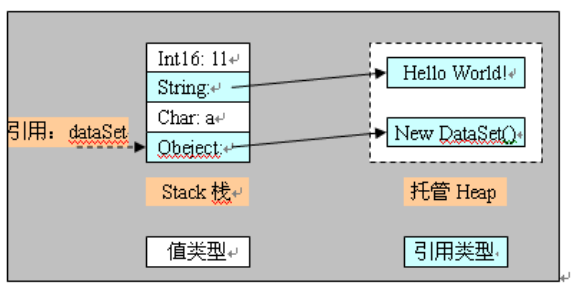
C#中的类型被分成两大类：

* 引用类型（reference type，又叫托管类型[managed type]），分配在内存堆上。
* 值类型（value type，也叫非托管类型[unmanaged type]）分配在堆栈上。



值类型在栈里，先进后出，值类型变量的生命有先后顺序，这个确保了值类型变量在推出作用域以前会释放资源。比引用类型更简单和高效。堆栈是从高地址往低地址分配内存。

引用类型分配在托管堆(Managed Heap)上，声明一个变量在栈上保存，当使用new创建对象时，会把对象的地址存储在这个变量里。托管堆相反，从低地址往高地址分配内存，如图



### 托管类型与非托管类型的区别

托管类型（托管资源）必须接受.NET Framework的CLR(通用语言运行时)的管理(诸如内存类型安全性检查),而非托管资源则不必接受.NET Framework的CLR管理. 需要手动清理垃圾（显式释放）。

托管资源在.NET Framework中又分别存放在两种地方: "堆栈"和"托管堆"(以下简称"堆");规则是,所有的值类型(包括引用和对象实例)和引用类型的引用都存放在"堆栈"中,而所有引用所代表的对象实例都保存在堆中。在C#中,释放托管资源是可以自动通过"垃圾回收器"完成的(注意,"垃圾回收"机制是.NET Framework的特性,而不是C#的).

在C++时代，我们需要自己来管理申请内存和释放内存. 于是有了new, delete关键字. 还有的一些内存申请和释放函数(malloc/free). C++程序必须很好地管理自己的内存, 不然就会造成内存泄漏(Memory leak). 在.net时代, 微软为开发人员提供了一个强有力的机制--垃圾回收. 垃圾回收机制是CLR的一部分, 我们不用操心内存何时释放, 我们可以花更多精力关注应用程序的业务逻辑. CLR里面的垃圾回收机制用一定的算法判断某些内存程序不再使用，回收这些内存并交给我们的程序再使用.[[4]](#footnote-4)

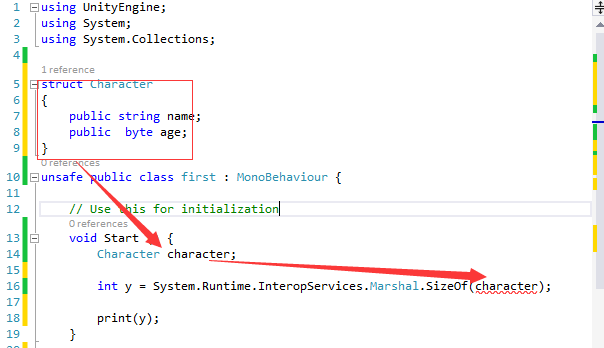
## 类型的大小

对于非托管类型，可以使用sizeof（）获取非托管类型的大小（以字节为单位）。

下面的示例演示如何获取int的大小：

|  |
| --- |
| // Constant value 4:  int intSize = sizeof(int); |

对于包含多个方法、属性的非托管类型（包括含有引用属性、字段的结构（struct））可以使用Marshal.SizeOf（）获取非托管类型的大小。



完整的代码如下：

|  |
| --- |
| using UnityEngine;  using System;  using System.Collections;  struct Character  {  public string name;  public byte age;  }  public class first : MonoBehaviour {  void Start () {  Character character;  character.name = "superman";  character.age = 18;  //获取非托管类型character结构的尺寸大小  int y = System.Runtime.InteropServices.Marshal.SizeOf(character);  print(y);  //获取非托管类型int的尺寸大小  int i = sizeof(int);  print(i);  }  }  /\*------------------------  \* 输出为：8 、4  ------------------------\*/ |

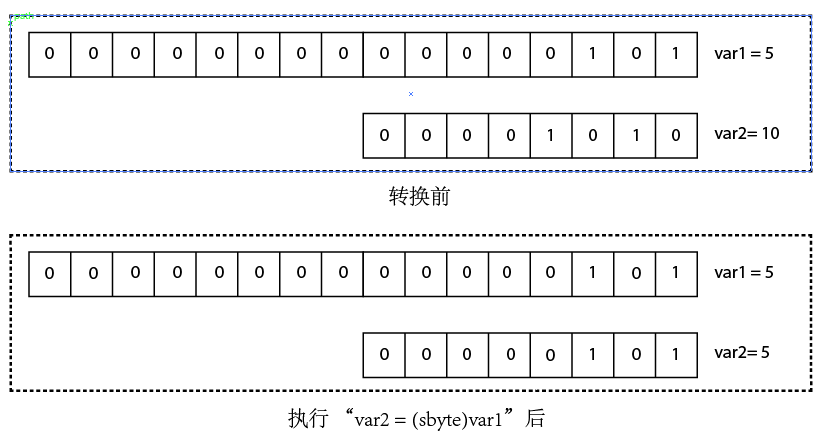
## 类型转换

转换是把一个类型的值作为另一类型的值的过程，转换后的值应当和原值一样，但是它是目标类型。

以下面的代码为例，来看看类型是如何进行转换的。

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript1 : MonoBehaviour {  // Use this for initialization  void Start ()  {  short var1 = 5;  sbyte var2 = 10;  var2 = (sbyte)var1;//类型转换（type casting）后，var1的值转换为8bit的sbyte类型的值，并赋给了var2  print("var2 的值为：" + var2);  }  /\*  \* 输出结果：  \* var2的值为：5；  \*/  } |

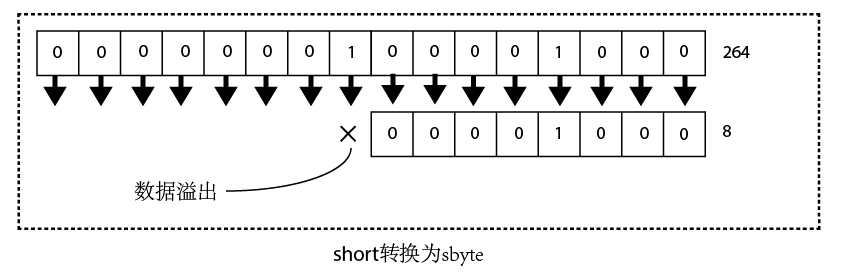
我们知道short类型是以16bit表示的数值，sbyte是以8bit表示的数值。当创建short类型的变量var1和sbyte类型的变量var2时，分别为它们分配了16bit和8bit的内存并把初始值赋给了这两个变量。当执行“var2 =(sbyte)var1”后，var1中的值转换成了8bit表示的数值，并赋给了var2，如下图所示。



short类型能容纳的最大值是32 767，而sbyte类型能容纳的最大值是127。如果我们为var1赋的值大于127会发生什么呢？执行下面的代码：

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript1 : MonoBehaviour {  // Use this for initialization  void Start ()  {  short var1 = 264;  sbyte var2 = 127;  var2 = (sbyte)var1;//类型转换（type casting）后，var1的值转换为8bit的sbyte类型的值，并赋给了var2  print("var2 的值为：" + var2);  }  /\*  \* 输出结果：  \* var2的值为：8；  \*/  } |

这种情况叫做丢失数据，这是长类型向短类型转换时数据溢出造成的。如下图。



有可能造成数据丢失或引发异常（转换失败）的转换都需要执行显示转换（explicit cast）。相反，不会丢失数据，而且不会引发异常的转换都可以隐式转换（implicit cast）。

### 隐式转换

对于预定义的类型，c#会自动将一个取值范围小的数据类型转换为另一个取值范围大的数据类型，。例如，从8bit的sbyte转换为16bit的short类型值时，不会发生数据丢失，只需要指定赋值运算符，转换将隐式地发生。

如下边的代码所示，var1 = var2 指令就是一个隐式转换。

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript1 : MonoBehaviour {  // Use this for initialization  void Start ()  {  short var1 = 13;//short取值范围：-32 768～32 767  sbyte var2 = 127;//sbyte取值范围：-128～127  var1 = var2;// 隐式转换  print("var1 的值为：" + var1);  }  /\*  \* 输出结果：  \* var2的值为：127；  \*/  } |

如果是从short转换为sybyte会发生什么情况呢？看下面的代码

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript1 : MonoBehaviour {  // Use this for initialization  void Start ()  {  short var1 = 13;//short取值范围：-32 768～32 767  sbyte var2 = 127;//sbyte取值范围：-128～127  var2 = var1;// 隐式转换  print("var2 的值为：" + var2);  }  /\*  \* 错误提示：  \* Assets/NewBehaviourScript1.cs(9,9): error CS0266: Cannot implicitly convert type `short' to `sbyte'. An explicit conversion exists (are you missing a cast?)  \*/  } |

在console面板中会出现错误提示：Assets/NewBehaviourScript1.cs(9,9): error CS0266: Cannot implicitly convert type `short' to `sbyte'. An explicit conversion exists (are you missing a cast?)。

如果想要取值范围大的数据类型的值转换为取值范围小的数据类型的值，就需要用到显式转换。

在13.3以及14.2章节也用到了隐式转换。把子类对象赋值给父类变量时实质上是一种隐式转换。如下面的例子：

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class StudyClass : MonoBehaviour  {  void Start()  {  Chinese cn = new Chinese();  Person p = new Chinese();//chinese对象隐式转换为person类。  Person p1 = cn;//cn执行的对象隐式转换为person类。  //Cannot implicitly convert type 'Person' to 'Chinese'. An explicit conversion exists (are you missing a cast?) D:\Documents\New Unity Project 4\Assets\Scripts\StudyClass.cs 13 23 Assembly-CSharp-vs  Chinese cn1 = new Person();//不能把父类隐式转换为子类。可以通过cast显示转换(Chinese)new Person();  Chinese cn2 = p;    }  }  class Person  {  public int age;  public int height;  public string Name { get; set; }  public void Say()  {  Debug.Log("你好，我是：" + this.Name);  }  public void Eat()  {  Debug.Log(Name + "味道真不错！");  }  }  class Chinese:Person  {  public void Kongfu()  {  Debug.Log( Name + "刀枪棍棒、十八般兵器样样精通");  }  }  class Japan:Person  {  public void ArtMovie()  {  Debug.Log(Name + "我精通艺术生活片。");  }  } |

### 显示转换

在c#中可以使用转型运算符来进行显示转换。显示转换可以强制把取值范围大的数据类型的值转换为取值范围小的类型的值。在这种情况下，假如数据丢失或引发异常，责任完全有编程人员承担。

下面的代码是一个显示转换的例子：

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript1 : MonoBehaviour {  // Use this for initialization  void Start ()  {  short var1 = 13333;//short取值范围：-32 768～32 767  sbyte var2 = 127;//sbyte取值范围：-128～127  var2 = (sbyte)var1;// 显式转换  print("var2 的值为：" + var2);  }  /\*  \* 执行结果：  \* var2 的值为：21  \*/  } |

从例子中可以看到，short类型的值显示转换为sbyte类型的值时，产生了数据丢失，但是代码仍然可以执行，编译器不会提示错误。

数据丢失，可能造成程序运行的不正常，为了解决这一情况，c#提供了checked表达式。在显示转换时，使用checked表达式，可以检测数据是否溢出。程序运行时，如果出现数据溢出会引发一个异常。如下面的代码：

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript1 : MonoBehaviour {  // Use this for initialization  void Start ()  {  short var1 = 1333;//short取值范围：-32 768～32 767  short var3 = 12;  sbyte var2 = 127;//sbyte取值范围：-128～127  sbyte var4 = 127;  var4 = checked((sbyte)var3);    print("var2 的值为：" + var4);  var2 = checked((sbyte)var1);// checked表达式检测数据是否溢出  print("var2 的值为：" + var2);  }  /\*  \* 执行结果：  \* var2 的值为：12  \* OverflowException: Number overflow.  \*/  } |

因为，var1的值超过short类型的取值范围，运行时，checked表达式会抛出一个数据溢出异常。Var3的值在short类型的取值范围内，不会引发异常。

#### Convert

还可以使用convert进行数据类型转换。它将一个基本数据类型转换为另一个基本数据类型。

Convert所执行的实际转换操作分为三类：

* 从某类型到它本身的转换只返回该类型。不实际执行任何转换。
* 无法产生有意义的结果的转换引发 InvalidCastException。不实际执行任何转换。下列转换会引发异常：从 Char 转换为 Boolean、Single、Double、Decimal 或 DateTime，以及从这些类型转换为 Char。下列转换会引发异常：从 DateTime 转换为除 String 之外的任何类型，以及从任何类型（String 除外）转换为 DateTime。
* 任何基类型（上面描述的基类型除外）都可以与任何其他基类型进行相互转换。

如果数字类型转换导致精度丢失（即某些最低有效位丢失），不引发异常。但是，如果结果超出了特定转换方法的返回值类型所能表示的范围，则将引发异常。

例如，当将 Double 转换为 Single 时，可能会发生精度丢失，但并不引发异常。但是，如果 Double 的值太大，无法由 Single 表示，则将引发溢出异常。

#### Parse

Parse可以把String转换成int,char,double....等,Convert进行字符串转换时内部调用了Parse方法。

果字符串为空，则抛出ArgumentNullException异常；

如果字符串内容不是数字，则抛出FormatException异常；

如果字符串内容所表示数字超出int类型可表示的范围，则抛出OverflowException异常

例如：

|  |
| --- |
| int i = int.Parse(n2); //把字符串n2转换为int类型的数字  float f = float.Parse(n2);//把字符串n2转换为float类型的数字 |

#### TryParse

不会产生异常，转换成功返回 true，转换失败返回 false。最后一个参数为输出值，如果转换失败，输出值为 0

|  |
| --- |
| string s = "1234";  int n ;  bool nb = int.TryParse(s, out n);  if(nb)  {  Console.WriteLine(n);  }else  {  Console.WriteLine("转换失败");  } |

#### is 运算符

is和as运算符一般用来检测类型转换

is运算符用来检测类型转换是否会成功完成，从而避免盲目尝试转换。is运算符只可以用于引用转换以及装箱和拆箱转换。

is 运算符的语法如下：

|  |
| --- |
| Expr is TargetType |

如果Expr可以转换为目标类型，运算符返回true。

例如，在以下的代码中，使用is运算符来检测Employee类型的变量Smith是否能够转换为person类型，然后进行相应的操作。

|  |
| --- |
| using System;  using UnityEngine;  class Employee : Person  {  }  class Person  {  public string Name = "匿名";  public int age = 25;  }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  Employee Smith = new Employee();  Person p;  if( Smith is Person)// 检测smith是否可以转换为person类型。  {  p = Smith;//smith隐式转换为person类型。  print(p.age);  print(p.Name);  }  }    } |

#### as 运算符

as运算符和强制转换运算符类似，只是它不抛出异常。如果转换失败，它把目标引用设置为null而不引发异常。

as运算符的语法如下：

|  |
| --- |
| Expr as TargetType |

TargetType是目标类型，它必须是引用类型。

例如，我们使用as把Employee类型的变量Smith转换为Person类型，并且赋值给一个Person类型的变量P。在使用之前检测p是否为null。

|  |
| --- |
| using System;  using UnityEngine;  class Employee : Person  {  }  class Person  {  public string Name = "匿名";  public int age = 25;  }  public class NewBehaviourScript1 : MonoBehaviour {    void Start ()  {  Employee Smith = new Employee();  Person p = Smith as Person;// 检测smith是否成功转换为person类型。成功转换的话会把smith转换为person类型并把引用赋给变量p，失败的话变量p的值为null  if( p != null)  {  print(p.age);  print(p.Name);  }  }    } |

#### (int)、Convert.ToInt32和int.Parse的区别

* (int): 能够使用(int)进行强类型转换的只能是数值类型，如long、short、double等。这种转换也叫cast转换，只有在内存存储上存在交集的类型之间才能进行cast。所以，不能转换为string。

如下的代码就行不通：

|  |
| --- |
| string text = "1234";  int i = (int)text; //cannot convert type 'string' to 'int' |

* Convert.ToInt32: 它属于system命名空间，需要引入该命名空间。Convert不再是内存级别的转换，而是考虑数据意义的转换。Convert.ToInt32可以把字符型数字转换为int32类型，如下代码中，text 实际上储存的是一个数值，我们希望把这个数值提取出来并以 int 的形式使用，那么就需要进行内容转换了，即

|  |
| --- |
| string text = "12";  int i = Convert.ToInt32(text); //字符串12转换为int类型 |

此外，Convert.ToInt32 有很多重载版本，例如 Convert.ToInt32(double value)，当用这个版本来把一个double转换成 int 时，ToInt32 会检查被转换的数值是否能够用 int 表示，即是否会发生“越界”，如果是就会抛出OverflowException异常，否则就会转换，但使用 (int) 进行强制转换，如果被转换的数值大于Int32.MaxValue，那么你将得到一个错误的结果，例如下面的代码：

|  |
| --- |
| double d = Int32.MaxValue + 1.1;  int i = (int)d;  //----------输出---------  // -2147483648 |

* int.parse： 与Convert.ToInt32类似，区别在于int.parse只能转换字符型数字。它与convet的最大不同是它们对null值的处理方法。Convert.ToInt32(null)会返回0而不会产生任何异常，但int.Parse(null)则会产生异常。

#### Convert.ToString 与 .ToString()

Convert对于null值不会抛出异常,而直接用ToString的话就会有异常

|  |
| --- |
| object v = null;  Console.WriteLine(Convert.ToString(v));  Console.WriteLine(v.ToString()); |

Convert能自动根据值的类型是否已实现IConvertible或IFormattable接口自动调用接口实现方法.

而ToString则未必会有这种效果!!(除非此方法已被重写),示例:

|  |
| --- |
| namespace ConsoleApplication1  {  public class ConvertTest  : IFormattable  {  #region IFormattable 成员  public string ToString(string format, IFormatProvider formatProvider)  {  return "This is formattable string";  }  #endregion  }  class Test  {  static void Main()  {  ConvertTest v = new ConvertTest();  Console.WriteLine(Convert.ToString(v));  Console.WriteLine(v.ToString());    Console.Read();  }  }  } |

### 引用转换

引用转换可以分为向上转换和向下转换两种

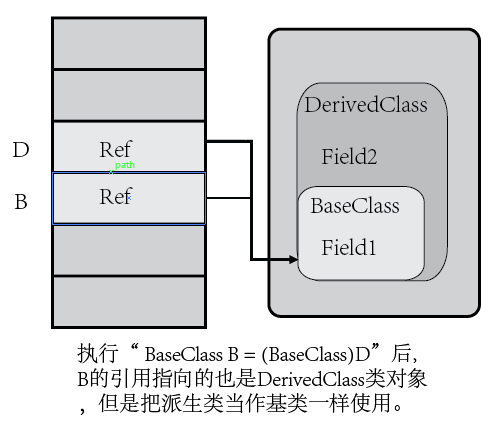
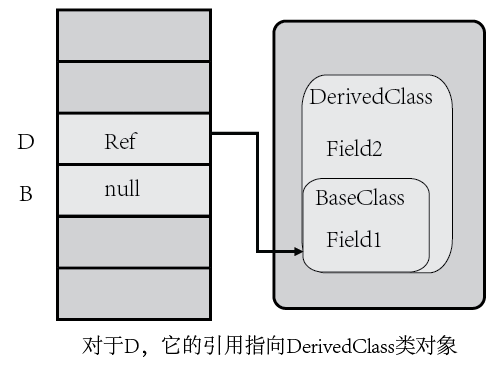
#### 向上转换

在代码编写中，如果把一个对象当成它的基类对象来使用，就称为向上转换（upcasting）。在实际编程中，经常会见到基类引用指向派生类对象，但是看上去像指向基类的对象。这就是著名的里氏替换原则（Liskov Substitution Priciple，LSP）。

例如，下面的代码中有两个引用变量：D和B，它们指向内存中相同的对象。对于D，它引用的对象是Derived类型的对象；对于B，它指向的也是DerivedClass类型的对象，但是操作时却像BaseClass类型的对象一样，它不能看到DerivedClass扩展BaseClass的部分，因此不能看到Field2。

|  |
| --- |
| using UnityEngine;  using System.Collections;  class BaseClass  {  public int Field1;  }  class DerivedClass:BaseClass  {  public int Field2;  }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  DerivedClass D = new DerivedClass();  BaseClass B = (BaseClass)D; //引用转换：把D的引用，赋值给B，B执行派生类对象，但是派生类当成基类的对象来使用，因此只能访问到Field1字段，而不能访问到不属于基类的field2  print(B.Field1);  }  } |

下图阐述了向上转换时，实际上发生了什么



向上转换可以是隐式的，强制转换表达式不是必须的。例如，可以修改“BaseClass B = (BaseClass)D”为：

|  |
| --- |
| BaseClass B = D; //向上转换：可以为隐式转换。 |

#### 向下转换

向下转换，英文为“Downcasting”，与向上转换相反，它是显示引用转换，是从一个普通类型到一个更精确的类型的引用转换，比如从基类到它的派生类的转换、从object类型到任何引用类型的转换。

向下转换通常是不安全的，是有条件限制的，因为如果不受限制，很可能会导致我们尝试引用在内存中实际不存在的类成员。然而，编译器确实允许这样的转换。但是在系统运行时遇到它们会抛出一个异常。

例如下面的代码将基类BaseClass的引用赋值给它的派生类DerivedClass的引用变量D，在编译时，转换是成功的，但在运行时，这样的转换会抛出一个异常。这是因为我们创建BaseClass的对象时，Field2字段部分不会在内存中，进行转换后，D会想访问DerivedClass的对象一样对BaseClass对象进行访问，CLR会检测到这种不安全的转换并抛出异常。

|  |
| --- |
| void Start ()  {  BaseClass B = new BaseClass();  DerivedClass D = (DerivedClass)B;//编译时可以成功，但是到运行时，运行到这一语句会抛出异常  print(D.Field1);  } |

在运行时能够成功进行（也就是不抛出异常）的向下转换有两种情况：

第一种情况：源引用为null。例如，在下面的代码中，即使转换基类的引用到派生类的引用通常是不安全的，但是由于源引用是null，这种转换是允许的，虽然它没有实际意义。

|  |
| --- |
| void Start ()  {  BaseClass B = null;  DerivedClass D = (DerivedClass)B;//运行时可以成功。    } |

第二种情况：源引用指向的实际数据实质上是目标类型的数据。例如下面的代码，第二行的隐式转换使B使用起来像BaseClass类型一样，但是它实际上指向的是DerivedClass类型的数据对象。第三行的显示转换把基类引用转换为派生类的引用，通常，这会产生异常。然而，在这种情况下，D1指向的对象实际上就是DerivedClass类型的对象。

|  |
| --- |
| DerivedClass D = new DerivedClass();  BaseClass B = D;  DerivedClass D1 = (DerivedClass)B;//运行时可以成功 |

### 案例

#### 显示字符编码

|  |
| --- |
| int a = 97;  char aa = (char)a;  Console.WriteLine(aa);//输出为a    char b = 'b';  int bb = (int)b;  Console.WriteLine(bb);//字符编码转换为数字  char hao = '郝';  int i = (int)hao;  Console.WriteLine(i);//郝的编码为37085  Console.ReadLine(); |

### 装箱与拆箱

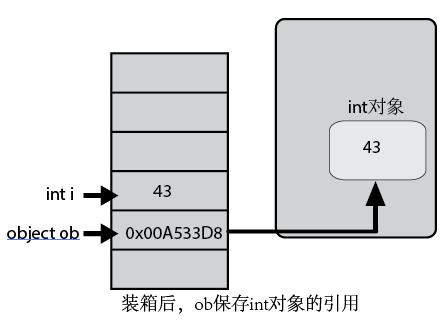
装箱是根据值类型的值在堆上创建一个完整的引用类型对象并返回对象引用的隐式转换。

下面代码是一个装箱的示例：

|  |
| --- |
| int i = 43;//声明int值变量  object ob = null;//声明object引用变量  ob = i;// 装箱； |

当执行“ob = i”后，系统会进行如下的操作：

* 在堆上创建int类型的对象
* 将i的值复制到int对象
* 返回int对象的引用，并赋给ob变量。



拆箱是把装箱后的对象显示转换为值类型的过程。例如，如下代码给出了个拆箱的示例：

|  |
| --- |
| int i = 43;//声明int值变量  object ob = null;//声明object引用变量  ob = i;// 装箱i并把引用赋值给ob；  int j = (int)ob;//拆箱ob并把值复制给j； |

#### 为什么能装箱

装箱对象与值类型要有父子关系，能进行类型转换，才能实现装箱。装箱后堆上存储的值的类型不变。

|  |
| --- |
| int n = 3;  string str = n.ToString();//这个不是装箱，int与string没有关系。里面存储的时字符串  int m = 4;  object o = m;//这个是装箱，m是int类型，就是int32，而int32是一个结构，继承自system.ValueType.而该类又继承自object。所以int类型与object具有父子关系，说以可以发生类型转换。o所指向的堆中存储的值为int类型的4. |

#### 拆箱的类型

拆箱的类型应当与装箱的类型一致

|  |
| --- |
| double d = 99;  object o = d;  int i = (int)o;//抛出System.InvalidCastException异常。装箱类型与拆箱类型应当一致。double i =(double)0;  Console.WriteLine(i);  Console.ReadLine(); |

#### 装箱与拆箱的效率问题

大量装箱与拆箱会产生大量的消耗，应当避免使用装箱与拆箱，下面我们以ArrayList与List<T>来对比下执行的效率。

|  |  |
| --- | --- |
| ArrayLisit | List<T> |
| ArrayList arr = new ArrayList();//ArrayList中的元素为object类型    Stopwatch watch = new Stopwatch();  watch.Start();  for (int i = 0; i < 10000000; i++)  {  arr.Add(i);//装箱  }  watch.Stop();  Console.WriteLine(watch.ElapsedMilliseconds);//2000毫秒左右  Console.ReadLine(); | List<int> list = new List<int>();  Stopwatch watch1 = new Stopwatch();  watch1.Start();  for (int i = 0; i < 10000000; i++)  {  list.Add(i);  }  watch1.Stop();  Console.WriteLine(watch1.ElapsedMilliseconds);//执行时间560毫秒  Console.ReadLine();  #endregion  Console.ReadLine(); |

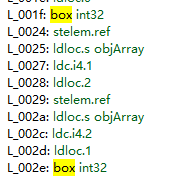
通过对比上面的代码知道，把1千万个数字装到ArrayList需要2000毫秒，而添加到List<int>中只需要560毫秒，速度提升了4倍左右，这就是为什么推荐使用泛型的最主要原因：不需要装箱，拆箱。

#### 字符串与数字拼接时的装箱

|  |
| --- |
| int a = 10;  int b = 90;  string c = "200";  string d = "300";  string s = a + c + b + d;  int r = int.Parse(s);  Console.WriteLine(s);  Console.ReadLine(); |

上面的代码运行时有装箱吗？

有。第5行发生了装箱，用.NET Reflector查看：string s = string.Concat(new object[] { num, str, num2, str2 });。可以知道，在拼接时，首先把int类型装箱成了object类型。那么装箱几次呢？我们再查看IL代码中有几个box（box为装箱）即可：



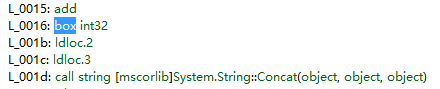
从上面的IL代码可以知道进行了两次装箱，接下来修改第5行的代码为：

|  |
| --- |
| string s = a + b + c + d; |

这时发生了几次装箱：

|  |
| --- |
| string s = (num + num2) + str + str2; |

先把两个数字相加，再装箱，进行了一次装箱



#### 方法调用时的装箱

|  |
| --- |
| int a = 10;  Console.WriteLine("s值为{0}",a);  Console.ReadLine(); |

上面的WriteLine重载时，第二个参数的值需要为obj类型，因此也发生了装箱。



#### 调用GetType时装箱问题

|  |
| --- |
| int a = 10;  string s = a.GetType().ToString();//GetType为object类型中的方法，调用时需要装箱为object  Console.ReadLine(); |

## 强类型和弱类型

强类型: 为所有变量指定数据类型称为“强类型”。强/弱类型是指类型检查的严格程度的。

弱类型：不为变量指定特定的数据类型。

弱类型的检查很弱，仅能严格的区分指令和数据。强类型的则严格的在编译期进行检查。

### 类型推断（var）

var是强类型，在编译的时候，编译器根据“=”右边的数据推断出数据类型，将其替换为对应的数据类型。

var只能用在方法的局部变量中，不能将var用于类的成员变量、方法的参数、方法的返回值中。

### Dynamic

.net4支持弱类型。

|  |
| --- |
| //动态类型，为弱类型  dynamic d = new System.Dynamic.ExpandoObject();  d.name = "hao";  d.age = 18;  Console.WriteLine(d.name);  Console.WriteLine(d.age);  Console.ReadLine(); |

# 运算符与表达式

编程语言必须要清楚描述如何进行数据运算。因此，我们必须通过某种方式告诉电脑进行什么样的数据运算，那么最简单的方法莫过于我们设计一个符号系统来表示它们。这就是运算符最初的设计用意。

在C#中，利用运算符说提供的特定语法，可以针对计算机中涉及的操作数执行各种不同的计算或操作。

## 什么是运算符、表达式？

运算符（operator）指定在表达式中要对一系列操作数（operand）执行哪些运算或操作，以生成一个新值或结果。运算符和操作数的组合，称为表达式（expression），每个表达式都会产生一个值，这个值就是表达式的值。例如，下面的代码中有两个操作数，数字5和3，它们被加法运算符“+”组合到一起，“5+3”就是一个表达式，它返回的值为8，并通过赋值运算符“=”把表达式的值赋给了变量i。

|  |
| --- |
| int i = 5 + 3; |

根据操作数的数量，可以将运算符分为3大类：

* 一元运算符：1个操作数。
* 二元运算符：2个操作数。
* 三元运算符：3个操作数。

## 运算符的本质

运算符本身是一种特殊的函数，操作数就是它的参数，运算结果值就是它的返回值。而每一个表达式都是单个运算符函数的组合，表达式可以看成一个组合成的特殊函数，表达式的值也就是该函数的返回值。

## 赋值运算符

赋值运算符是用的最多的运算符。它将位于运算符右边的值赋给左边的变量。根据其赋值的方式，可以将赋值运算分为简单赋值和复合赋值两类。

简单赋值，计算右边表达式的值，并把返回值赋给左边的变量。而复合赋值将常见的二元运算符与赋值运算符合并到了一起，每一个复合赋值运算都可以转换为等价的简单赋值运算。如下图表：

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符分类 | 运算符 | 运算符例子 | 等价表达式 |
| 简单赋值 | = | a = 3 + 4 |  |
| 复合赋值 | += | a += 3 | a = a + 3 |
| -= | a -= 3 | a = a - 3 |
| \*= | a \*= 3 | a = a \* 3 |
| /= | a /= 3 | a = a / 3 |
| %= | a %= 3 | a = a % 3 |

赋值运算的右值（rvalue）可以是字面量，也可以是一个表达式、函数返回值、变量。而运算符左边只能是变量、属性、索引或事件，不能是字面量，也不能是没有声明的对象的引用。

它常见的合法形式：

|  |
| --- |
| int math(int a, int b)  {  return a + b;  }  void Start ()  {    int i\_1 = 5;//运算符左边为变量，右边为字面量  int i\_2 = 5 + 3;// 运算符左边为变量，右边为字面量表达式  int i\_3 = i\_1 \* i\_2; //运算符左边为变量，右边为变量表达式  object ob = new object();//运算符左边为变量，右边为object对象。“=”赋值运算把object对象的引用赋给变量ob  int i\_4 = math(3, 4);// 运算符左边为变量， 右边为函数。“=”赋值运算会把函数的返回值赋给变量i\_4;  } |

非法形式有（注意：下面的例子是非法例子，无法编译通过）：

|  |
| --- |
| 4 = 5;//左边是字面量，非法  new object() = 5;//左边是没有声明的对象。 非法。 |

## 算术运算符

算术运算符可以分为一元算术运算符和二元算术运算符。

有的时候，我们可能想改变一个数值的正负号。在这种情况下就可以使用一元正运算符“+”或一元负运算符“-”。一元正运算符对值几乎没有影响，只是出于对称性的考虑才加进来，一元负运算符等价于从零减去操作数。例如，下面的表格将当前的值变为一个负值，指明这是一个信用卡的当月需要的还款金额。

|  |
| --- |
| // 信用卡当月还款金额  decimal debt = -1023.25m; |

自增、自减也是一元运算符

|  |
| --- |
| int i = 5;  i++;//等价于 i= i+1  i--;//等价于 i = i-1 |

二元算术运算符需要两个操作数，常见的有加、减、乘、除以及模运算符。加减乘除的运算与我们在小学学习的一样，不用多讲。模运算，实际上就是常用的除法取余数。示例如下：

|  |
| --- |
| int x = 5;  int y = 4;  int i\_1 = x + y;//加法运算符“+”把x、y的值加起来,并且赋给左侧的变量i\_1,  int i\_2 = x - y;//减法运算符“-”把x、y的值相减,并且赋给左侧的变量i\_2  int i\_3 = x \* y;//乘法运算符“\*”把x、y的值乘起来,并且赋给左侧的变量i\_3  int i\_4 = x / y;//除法运算符“/”把x、y的值相除,并且赋给左侧的变量i\_4,  int i\_5 = x % y;//模运算符“%”把x、y的值相除取余数，并且赋给变量i\_5 |

需要注意的是，当使用除非运算符把两个整数相除时，产生的值还是整数，如上面的“int i\_4 = x / y”语句，它的结果为1，而不是1.25。

## 关系运算符

关系运算符可以分为关系比较运算符和相等比较运算符，都是二元运算符。关系运算比较它们的操作数并返回bool型值。关系运算符如图表所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | 运算符 | 名称 | 功能 |
| 关系比较运算符（比较大小） | < | 小于 | 第一个操作数小于第二个操作数，返回值为true，否则返回为false |
| > | 大于 | 第一个操作数大于第二个操作数，返回值为true，否则返回为false |
| <= | 小于等于 | 第一个操作数小于等于第二个操作数，返回值为true，否则返回为false |
| >= | 大于等于 | 第一个操作数大于等于第二个操作数，返回值为true，否则返回为false |
| 相等比较运算符（比较是否相等） | == | 等于 | 第一个操作数等于第二个操作数，返回值为true，否则返回为false |
| != | 不等于 | 第一个操作数不等于第二个操作数，返回值为true，否则返回为false |

看下面的例子：

|  |
| --- |
| int x = 5;  int y = 4;  print(x == y);//输出为fasle；  print(x != y);//输出为true；  print(x > y);//输出为true；  print(x < y);//输出为false |

## 逻辑运算符

逻辑运算符用于比较它们的操作数的逻辑值，并返回结果逻辑值。运算符如下表所示：

|  |  |  |
| --- | --- | --- |
| 运算符 | 名称 | 描述 |
| && | 与（And） | 当两个操作数（通常是表达式）都是true，结果为true，否则为false。 |
| || | 或（Or） | 两个操作数（通常是表达式）只要有一个为true，就会返回true。 |
| ! | 非（Not） | 如果操作数是false，返回true，否则返回false。 |

逻辑AND运算符和逻辑OR运算符是二元运算符，用法看下例：

|  |
| --- |
| var score = 60;  //使用 && 只有同时满足 score大于等于60并且score小于100时，才为真。  if(score >= 60 && score <100)  {  print("you can enter next level");  }  else  {  print("sorry! you are loser");  }  /\*  \* 输出结果：  \* you can enter next level  \*/    var age = 300;  //使用 ||,只要满足age小于0或者age大于200，就为真  if(age < 0 || age >200)  {  print("这不是人啊");  }  else  {  print("是个正常人");  }  /\*  \* 输出结果：  \* 这不是人啊  \*/ |

逻辑NOT运算符是一元运算符，它把操作数的bool值取反。见下例：

|  |
| --- |
| bool val = false;  //vald 值是false，因为前面加了一个！运算符，那么！val的返回值为ture。  if (!val)  {  print("这是真的");  }  /\*输出结果：  \* 这是真的  \*  \*/ |

## 条件运算符（三元运算符）

条件运算符是三元表达式，它的表达形式是：

|  |
| --- |
| 条件表达式 ? 表达式1 ：表达式2 ; |

当条件表达式返回值为true时，返回表达式1的值；当表达式返回值为false时，返回表达式2的值。

看下面的例子：

|  |
| --- |
| int x = 5;  int y = 10;  //当x>y为真时返回字符串“x大于y”，当x>y不为真时返回字符串 "x小于y"  string PrintStr = x > y ? "x大于y" : "x小于y"; |

这个三元运算符相当于简单的if else语句的简写形式。如下面的代码执行的操作与上面的例子相似：

|  |
| --- |
| int x = 5;  int y = 10;  string printStr = "";  if(x > y)  {  printStr = "x 大于y";  }  else  {  printStr = "x 小于y";  } |

使用这个条件运算符（？：）可以达到简化代码的目的。很多优秀的程序员都喜欢代码简洁，所以他们经常使用这个运算符。

## Null合并运算符（？？）

?? 运算符称作 null 合并运算符。如果此运算符的左操作数不为 null，则此运算符将返回左操作数；否则返回右操作数。

|  |
| --- |
| class NullCoalesce { static int? GetNullableInt() { return null; } static string GetStringValue() { return null; } static void Main() { int? x = null; // Set y to the value of x if x is NOT null; otherwise,  // if x = null, set y to -1.  int y = x ?? -1; // Assign i to return value of the method if the method's result  // is NOT null; otherwise, if the result is null, set i to the  // default value of int.  int i = GetNullableInt() ?? default(int); string s = GetStringValue(); // Display the value of s if s is NOT null; otherwise,   // display the string "Unspecified". Console.WriteLine(s ?? "Unspecified"); } } |

## 递增和递减运算符

递增运算符对操作数加1，递减运算符从操作数减1 。它们都是一元运算符，并有两种形式：前置形式和后置形式，它们产生不同的效果。下面的表描述了它们的区别：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 运算符 | 名称 | 表达式：x=10 | 返回给表达式的值 | 计算后变量的值 |
| ++ | 前置递增 | ++x | 11 | 11 |
| 后置递增 | x++ | 10 | 11 |
| -- | 前置递减 | --X | 9 | 9 |
| 后置递减 | x-- | 10 | 9 |

例如，下面是四个不同版本运算符的一个简单示范，为了展示相同情况下的不同结果，操作数x在每个赋值语句之前被重设为10：

|  |
| --- |
| int x = 10, y;  y = ++x;  print(y);//输出：y = 11；  //--------------------  x = 10;  y = x++;  print(y);//输出：y = 10；  //--------------------  x = 10;  y = --x;  print(y);// 输出： y = 9；  //--------------------  x = 10;  y = x--;  print(y);// 输出： y = 10； |

## typeof 运算符

typeof运算符是返回它的参数的System.Type对象。通过这个对象可以得到类型的特征。下面是typeof运算符的示例。Type是system命名空间中的一个类。

|  |
| --- |
| Type t = typeof(int); |

下面的代码运用typeof运算符以获取int类型的信息，并打印出它的公有字段和方法的名称。

|  |
| --- |
| using System;  using UnityEngine;  using System.Reflection;  public class NewBehaviourScript1 : MonoBehaviour {    void Start ()  {  Type t = typeof(int);  MethodInfo[] MI = t.GetMethods();  FieldInfo[] FI = t.GetFields();  foreach(FieldInfo f in FI)  {  print("公共字段:" + f.Name);  }  foreach( MethodInfo m in MI)  {  print("公共方法:" + m.Name);  }    }    } |

typeof运算符还被GetType方法调用，该方法对每个类型的的每个对象都有效。例如，下面的代码获取对象类型的名称：

|  |
| --- |
| int i = 5;  string S= i.GetType().Name;  print(S);  /\*输出结果  \* Int32  \*/ |

## 优先级顺序

从小学学到的知识可以知道，乘法必须在加法之前执行，因为乘法比加法有更高的优先级。在C#中事情跟复杂一点，它由超过45个运算符和14个级别的优先级。

下表是全部的运算符和它们的优先级顺序。该表把最高优先级运算符列在顶端，并持续下降到底端最低优先级运算符

|  |  |
| --- | --- |
| 分类 | 运算符 |
| 初级运算符 | a.x, f(x), a[x], x++, x--, new, typeof, checked, unchecked |
| 一元运算符 | !, ~, ++x, --x, (T)x |
| 乘法 | \*, /, % |
| 加法 | +, - |
| 移位 | <<, >> |
| 关系和类型 | <, >, <= , >=, is, as |
| 相等 | ==, != |
| 位与 | & |
| 位异或 | ^ |
| 位或 | | |
| 条件与 | && |
| 条件或 | || |
| 条件选择 | ?: |
| 赋值运算符 | =, \*=, /=, %=, +=, -=, <<=, >>=, &=, ^= , |= |

优先级顺序，确实很难记忆，可以使用圆括号来代替记忆。圆括号可以改变运算符的优先级，并可以增强代码的可读性。例如下面两个表达式的求值结果是完全不一样的。

|  |
| --- |
| float x = 3 + 4 \* 6 / 2;  print("y =" + x);  float y = (3 + 4) \* (6 / 2);  print("y =" + y);  /\*输出结果：  \* x = 15  \* y = 21；  \*/ |

# 流程控制语句

## 什么是语句

语句是描述一个类型或告诉程序去执行一个动作的源代码指令。在C#中有三种主要的语句种类：

* 声明语句：声明类型或变量的语句。
* 嵌入语句：执行动作或管理控制流的语句。
* 标签语句：控制可以跳转到的语句。

下面的代码给出了每种语句的实例：

|  |
| --- |
| int i = 5; //声明语句  void Start ()  {  int x = 3;//声明语句  x = i;//执行动作的嵌入语句  if (x == 5) //控制流程的嵌入语句，  { //大括号内的代码为嵌入块  i = 3;//执行动作的嵌入语句  top: x++;// 标签语句  } |

本章主要学习控制流程的嵌入语句的使用方法，声明语句以及标签语句会在以后的章节中进行讲解。

## 控制流语句

C#提供与C、C++、java等编程语言共同的控制流结构。可以分为三大类：

* 条件语句：依据一个条件执行或跳过一个代码块。条件语句由以下几种：

○ if

○ if…else

○ switch

* 循环语句：重复执行一个代码块。循环语句如下：

○ while

○ do

○ for

○ foreach

* 跳转语句：把控制流从一个代码块改变到另一个代码块中的指定语句。跳转语句如下：

○ break

○ continue

○ return

○ goto

○ throw

## 条件语句

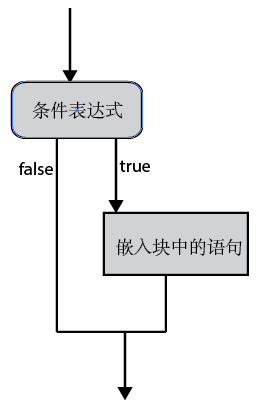
条件语句会根据判断的结果（true或false）来执行不同的分支。

### if 语句

if语句实现按条件执行。如果条件表达式为true（真），执行嵌入块中的语句；如果为false（假），不执行流程语句。If语句的语法如下所示：

|  |
| --- |
| if(条件表达式)  {  嵌入块中的语句；  } |

图阐明了它的流程：



下面的代码展示了if语句的示例：

|  |
| --- |
| int x = 3;  if (x < 5) //3 < 5表达式成立，会执行嵌入块中的语句，  {  int y = x + 3;  print(y);// 输出结果：6  } |

需要注意的是，条件表达式返回的值应该是bool类型。在C#中值类型等数据类型无法转换为bool类型，因此以下的代码会出错：

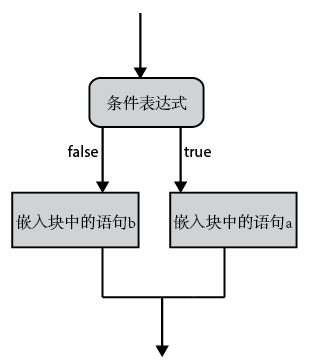
|  |
| --- |
| int x = 3;  if (x) //条件表达式需要的是bool类型，不能使用int类型，因此会出错  {  ;  } |

### if…else语句

if…else语句实现了双路分支。如果条件表达式为true，执行嵌入块中的语句a，如果为假，执行嵌入块中的语句b。if…else语句的语法如下：

|  |
| --- |
| If(条件表达式)  {  嵌入块中的语句a；  }else{  嵌入块中的语句b；  } |

下图阐明了if…else语句的流程：



下面是if…else语句的例子：

|  |
| --- |
| int x = 3;  if (x > 4)  {  int y = x + 4;  }  else  {  int y = x - 4;  } |

### if…else if…else语句

if…else if…else语句可以实现多路分支，它实际上是if…else语句后面接着一个if…else语句而已。、

需要注意的是，一旦有一个if语句中的条件表达式为真，你们该if语句包含的嵌入块语句会执行，当该语句之后而且if…else语句都不会执行了。不注意这一点，会写出有错误的代码。比如，举一个判断成绩的例子：判断一个成绩是优秀、良好、及格还是不及格。看下面的代码：

|  |
| --- |
| int score = 91;  if(score < 60)  {  print("你的成绩不及格");  }  else if(score >80 )  {  print("你的成绩很优秀");  }  else if(score > 90)  {  print("你的成绩是优良");  }  else if (score > 60 && score < 80)  {  print("你及格了");  }  else  {  print("没有你的成绩");  } |

上面例子的输出结果是：你的成绩是优秀。这是因为我们将score >80放在了前面，那么当该表达式为真时，就会执行嵌入块中的语句：print("你的成绩很优秀")。后面的都会被忽略。

如果希望本例中if带给我们更加精确的结果，我们应当把if(score > 90)及其包含的嵌入块放到前面。总结一下，在条件判断有交集时，应当把精确的范围较小的条件表达式提前。比如本例中 score > 90 和score > 80有重叠的地方，就应当把范围较小的表达式score > 90放在前面。

### switch语句

swith语句可以实现多路分支。它的语法结构如下：

|  |
| --- |
| switch(测试表达式)  {  case 情况1:  分支语句;  break;  case 情况2:  分支语句;  break;  case 情况3:  分支语句;  break;  default:  分支语句;  break;  } |

如果情况1的值等于测试表达式的值，那么在分支标签后的分支语句会被执行，直到遇到break语句。每个分支必须包含break语句（或者return语句），它把执行引向switch语句的结尾。同理情况2的值等于测试表达式的值，那么情况2标签下的分支语句会被执行。当情况都不满足时，会执行default中的分支语句。

switch语句常用于选择分支判断。例如，一个箱子中有5种水果。我们用手在箱子中摸一个水果出来。如果是苹果，那么付苹果的钱；如果是香蕉，那么付香蕉的钱，如果是其它水果，那应该付其各自的价钱。下面的代码实现了这一个过程：

|  |
| --- |
| string[] fruits = new string[] { "苹果", "香蕉", "橘子", "番茄", "西瓜"};//把水果放入数组中  float randomFruit = UnityEngine.Random.value \* fruits.Length;// random.value产生0-1之间的随机浮点数；fruits.length返回数组的长度。因此randomFruit值为0-5之间的随机浮点数  print(randomFruit);  string currentFruit = fruits[(int)Math.Floor(randomFruit)];//Math.floor向下取最接近的整数，需要使用(int)显示转换  print("摸到的水果是" + currentFruit );  switch (currentFruit)  {  case "苹果":  print("需要付苹果的钱");  break;  case "香蕉":  print("需要付香蕉的钱");  break;  case "橘子":  print("需要付橘子的钱");  break;  case "番茄":  print("需要付番茄的钱");  break;  case "西瓜":  print("需要付西瓜的钱");  break;  default:  print("没有选择水果");  break;  } |

这个例子描述了switch最常用的形式。可以看出，这种逻辑可以使用if…else if…else语句实现。但是，使用switch语句更加简洁直观。此外，要说明的是，switch语句只能用于等值判断，区间判断需要使用if…else if…else语句。

需要注意的是，case中的值必须是常量，不能是变量、表达式。

## 循环语句

循环语句会在条件为真时，重复执行某些操作。

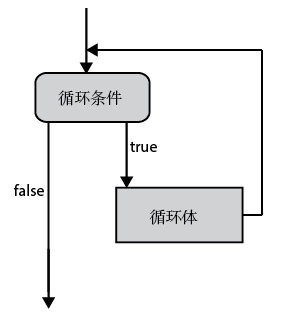
### while循环

while循环是一种简单的循环，它的语法如下：

|  |
| --- |
| while (循环条件)  {  循环体;  } |

首先，循环条件会求值，如果为真，循环体执行，并且循环条件再次求值。每次循环条件为真时，循环体都要再执行一次。当循环条件为假时，循环介绍。

下图阐述了while循环的流程：



我们来看看while循环的示例。循环条件中的变量从3开始在每次迭代中递减。当变量的值小于0时循环退出。

|  |
| --- |
| int number = 3;  while(number >= 0)  {  print("你可以尝试"+ number + "次");  number--;  }  /\*输出结果：  \* 你可以尝试3次  \* 你可以尝试2次  \* 你可以尝试1次  \* 你可以尝试0次  \*/ |

### do-while循环

用while时，会首先判断是否符合循环条件，然后再执行循环体；用do-while时，会先执行循环体，再判断是否符合循环条件，所以do-while和while的唯一的不同点在于，do-while的循环体至少执行一次。

do-while的语法如下：

|  |
| --- |
| do  {  循环体;  }  while (循环条件); |

下面的代码与前面的while循环输出结果一样：

|  |
| --- |
| int number = 3;  do  {  print("你可以尝试" + number + "次");  number--;  }  while (number >= 0); |

do-while循环在实际应用中比较少见。

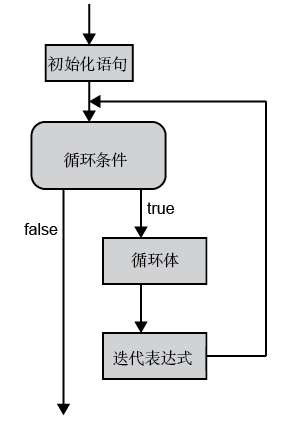
### for循环

for循环比较灵活，应用最为广泛。for循环的语法如下：

|  |
| --- |
| for (初始化语句; 循环条件; 迭代表达式)  {  循环体  } |

在循环的开始，初始化语句被执行一次，然后循环条件求值，如果返回为true，循环体执行，接着执行迭代表达式，然后循环条件再次求值，只要返回为true，循环体和迭代表达式都将被执行，只到循环条件为false时，退出循环。

下图阐明了for语句的控制流程：



下面是一个for循环的示例：

|  |
| --- |
| for (int i = 0; i < 3; i++ )  {  print(i);  }  /\*输出结果：  \* 0  \* 1  \* 2  \*/ |

for循环很灵活，括号内的初始化语句、循环条件、迭代表达式都可以为空，你可以将这些语句分开放到for循环的外面或循环体内。只要确保达到目的并终止循环即可。下面给出的for循环和上面的示例功能相同。

|  |
| --- |
| int i = 0;  for (; ; )  {  print(i);  i++;  if (i < 3) break;  }  /\*输出结果：  \* 0  \* 1  \* 2  \*/ |

初始化语句和迭代表达式还可以包含多个表达式，只要使用逗号隔开。

例如，下面的代码在初始化语句中有两个变量声明，而且在迭代表达式中有两个表达式：

|  |
| --- |
| for (int i = 0, j = 2; (i + j) < 10; i++, j++ )  {  print(i + j);  }  /\*输出结果：  \* 2  \* 4  \* 6  \* 8  \*/ |

最后需要说明的是，初始化语句中声明的变量只在for语句的内部被认识，而循环体内部声明的变量只在循环体内部被认识。

|  |
| --- |
| for (int i = 0; i < 3; i++ )//变量i在大括号和循环体中可以被认识  {  int j = 2;//循环体中的变量只在循环体中可以被认识  print(i+j);  }//跳出循环后，i不再存在  for (int i = 0; i < 10; i++ )//变量i需要重新声明  {  ;  } |

### foreach-in循环

foreach-in语句用来枚举一个集合中所有的元素。它的语法如下：

|  |
| --- |
| foreach(迭代变量 in 数组)  {  循环体;  } |

|  |
| --- |
| 术语 |
| * 集合: 集合是一个广泛的概念，拥有可枚举属性的对象都可以看成集合。 * 枚举：简单的说，就是挨个访问一个集合中的所有成员。 | |

foreach-in语句工作方式如下：

* 从数组的第一个元素开始并把它赋值给迭代变量
* 然后执行循环体。在循环体中，我们可以把迭代变量作为数字元素的只读别名
* 在循环体执行后，foreach-in语句选择数组中的下一个元素并重复处理

这样，它就循环遍历了数组，允许我们逐个访问每一个元素。如下面的示例：

|  |
| --- |
| string[] player = new string[] { "jason", "chen", "xiaoming", "alex"};  foreach(string item in player)  {  print(item);  }  /\*输出结果：  \* jason  \* chen  \* xiaoming  \* alex  \*/ |

有一点需要说明的是，迭代变量是只读的，所以它不能改变。

|  |
| --- |
| int[] number = new int[] { 11, 12, 43, 123};  foreach(int item in number)  {  item++;//错误  print(item);  } |

## 跳转语句(循环中断)

当控制流到达跳转语句时，程序将无条件转移到程序的另一部分。

### break语句

在循环体中，break用来直接跳出循环，不再执行循环。

例如，下面的while循环的循环条件永远为真，将会是一个无限循环。但是在三次循环迭代后，遇到了break语句，就会推出循环。

|  |
| --- |
| byte i = 0;  while(true)  {  i++;  print(i);  if(i >= 3)  {  break;  }  } |

### continue语句

continue语句会跳出当前这一轮的循环，直接跳到下一轮循环，而在这一轮循环中，循环体内continue语句后面的语句不会被执行。

例如：下面的for循环执行了五次，在前三次迭代中，它遇到continue语句后会直接重新开始循序，错过了循环底部的print语句。只在后两次迭代中执行print语句。

|  |
| --- |
| byte i = 0;  while(i< 5)  {  if(i < 3)  {  i++;  continue;//重新开始循环  }  print(i);  } |

### Return语句

Return终止当前执行的函数，后续的代码也不会执行。

### 标签和goto语句

标签语句由一个标识符后面跟着一个冒号再跟着一条语句组成。它的语法如下：

|  |
| --- |
| 标识符：语句 |

标签有它自己的声明空间，所以标识符可以和一个本地变量、参数有相同的名称。如下边的示例：

|  |
| --- |
| void Start ()  {  int id = 12;  id: id++;//标签语句  print(id);  } |

标签语句执行时标识符不执行，仅仅执行冒号后的语句。此外，标签语句只能在块内部，它的有效范围是它声明所在的块以及嵌套在该块内部的快。下面的例子中，if语句块中的goto语句可以跳转到Start方法中声明的标签语句id。

|  |
| --- |
| int id = 12;  id: id++;//标签语句，其声明在块内部以及该块内部的块中可以被认识  if (id <= 15)  {  print(id);  goto id;//跳转到标签id；  }  /\*输出结果：  \* 13  \* 14  \* 15  \*/ |

# 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"); ;  }  } |

# 字段

字段是与类或结构关联的变量。它是类或结构的数据成员，用来存储描述性的数据信息。

字段就是类的状态。比如花这个类，有名称、颜色、价格等字段。

## 字段的声明

字段是隶属于类或结构的变量，它的声明方法与我们在[变量的声明和使用](#_变量的声明和使用)这一节中所讲的方法一样。

例如，下面的代码声明了三个不同类型的字段，用来存储不同类型的信息。

|  |
| --- |
| public class NewBehaviourScript1 : MonoBehaviour {  public int number;//声明int类型字段  public string str;//声明string类型字段  public float randomNumber;//声明float类型字段  } |

## 字段的初始化

只能用字面量，常量或静态的字段变量、方法、属性来初始化字段。

|  |
| --- |
| public class NewBehaviourScript1 : MonoBehaviour {  public int number = 4;//声明并初始化字段值为4  public static int s = 3;// 声明静态字段并赋值为3  public const int c = 2;  /// <summary>  /// 声明一个静态方法  /// </summary>  /// <returns></returns>  static int nm()  {  return 3;  }  //public int number1 = number;//不能把字段变量给字段变量赋值  public int number2 = s;//静态字段可以用来为字段赋值  public int number3 = nm();//静态方法也可用来为字段赋值  public int number4 = c;//常量用来为字段赋值 ，这是最常用的  } |

当多个字段存储值相同时，使用常量来赋值会很方便，易于以后的调试与修改。比如，下面的三个字段用来存储武器的子弹数量，把它与常量关联后，当以后修改武器可以发射的子弹数量时，只需要修改常量AMMO\_COUNT的值就可以了。

|  |
| --- |
| const int AMMO\_COUNT = 10;  int \_ammoForPistol = AMMO\_COUNT;  int \_ammoForRifle = AMMO\_COUNT;  int \_ammoForChainGun = AMMO\_COUNT; |

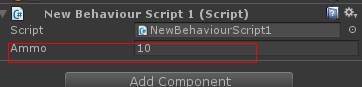
## Unity中公有字段的特殊性

在Unity中，标记为public的字段会在inspector中显示。在编译时和运行时都可以修改这个字段的值。

例如，我们在脚本中创建如下的公有字段：

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class NewBehaviourScript1 : MonoBehaviour {  public int Ammo = 10;  } |

在inspector中，我们可以发现该脚本中显示出了这个公有字段



可以在inspector中直接修改Ammo字段的值，而不需要返回脚本中修改。此外，运行游戏时，也可以修改该字段的值。

## 字段的作用

字段是数据成员，它主要用来存储对象的数据，为行为提供逻辑判断的依据。例如，当字段的值大于零时，会在场景中添加一个立方体。添加立方体这个行为执行的依据是字段中存储的数据。

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class NewBehaviourScript1 : MonoBehaviour {  public int Ammo = 10;  void Update()  {  if (Ammo > 0) //字段的值大于零时，会执行条件语句中的动作。  {  GameObject.CreatePrimitive(PrimitiveType.Cube);//创建一个cube对象  }  Ammo -= 1; //字段的值自减1.  }  } |

## 字段的数据类型

除了支持c#中提供的15中预定义数据类型外，在Unity中，还可以定义字段为GameObject。

# 方法

## 什么是方法

方法是类能够执行的动作，比如问好、吃饭、跑步等。

方法就是函数（面向过程中叫函数、面向对象中方法），是可以进行重用的一种机制。方法是具有名称的代码块。可以使用方法的名称执行方法，也可以把数据传人方法并输出数据。

在C#中，方法是类的函数成员。它由两个主要部分组成：

* 方法头：指定方法的特征，包括以下信息

○ 方法是否返回数据，如果返回，返回什么类型；

○ 方法的名称；

○ 什么类型的输入可以传人方法

* 方法体。包含可执行的语句序列。执行时从方法体的第一条语句开始，一直到整个方法结束

下面的示例展示了方法的基本结构：

有参数和返回值的方法头

|  |
| --- |
| int calculate(int a, int b)  {  int count = a + b;  方法体  return count;  }    无参数和返回值的方法头  void Start ()  {  int result = calculate(4, 5);  print(result);  } |

现在，我们来详细讲解一下方法头的形式：

|  |
| --- |
| int calculate(int a, int b)  **返回类型**  **方法名称**  **参数列表** |

从上边的图表中可以知道，方法头中包括了返回类型、方法名称以及参数列表信息。如果，我们不需要传人数据时，可以省略参数，如下图：

|  |
| --- |
| int calculate() |

方法体是一个代码块，是大括号括起来的语句序列。块中可以包含本地变量、控制流语句、方法调用等语句。

下图展示了一个方法体及其组成的示例：

|  |
| --- |
| void Start ()  {  **本地变量**  int i = 4;  while(i < 6)  {  **控制流语句**  print(i);  i++;  **调用方法**  }  } |

## 局部变量及其有效范围

位于方法内的变量叫做局部变量，用来保存临时的计算数据。它的声明语法与”3.3 变量的声明和使用“中的声明方法一样。

局部变量的有限范围是声明所在的块以及其内嵌的块中。声明执行时，它开始存在，当块完成执行时结束存在。

下面的示例展示了两个局部变量的声明和使用。第一个是值类型变量，第二个是引用类型变量。

|  |
| --- |
| void Start ()  {  int i = 4;//值类型局部变量  string[] text = new string[] {"text1", "text2" };//引用类型局部变量  } |

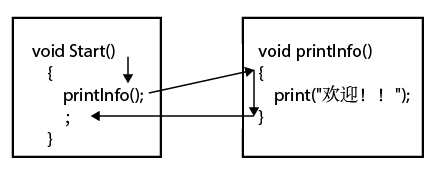
## 方法的调用

在C#中，需要在方法体内部调用其他方法。

例如，下面的类中声明了一个名称为printInfo的方法，它在Start方法体中被调用。

|  |
| --- |
| using System;  using UnityEngine;  public class NewBehaviourScript1 : MonoBehaviour {  /// <summary>  /// 声明了一个printInfo方法  /// </summary>  void printInfo()  {  print("欢迎！！");  }  void Start()  {  printInfo();//调用printInfo方法  }  } |

下图阐明了调用方法时的执行顺序：



当执行到调用的方法时，它会跳到被调用的方法的方法体内，从方法体中的第一条语句开始向下执行，执行完毕后，跳回到调用方法下面的语句继续执行。

## 返回值

方法可以返回数据，叫做返回值，它会保存在调用方法所在的变量中。如果不返回值，必须声明为void类型；如果返回值，必须声明一个返回类型，这个返回类型要与返回值的类型一致。

例如，下面的代码中声明了一个名为GetHour的方法，它返回当前的时间，返回值的类型为int。调用GetHour方法时，它的返回值保存在了nowTime变量中。

|  |
| --- |
| int GetHour()  {  DateTime DT = DateTime.Now;  int hour = DT.Hour;  return hour;  }  void Start()  {  int nowTime = GetHour();  print(nowTime);  } |

## 参数

参数是传人方法的数据，方法会对这些传入的数据进行处理。

通过参数，可以把数据传人方法。参数可以分为形参和实参

### 形参

形参是声明在方法的参数列表中的变量。

下面的方法头展示了形参声明的语法。它声明了两个形参，一个是字符串类型，一个是int类型。

|  |
| --- |
| void student(string name, int age)  {  ;  **形参声明**  } |

形参可以在方法体内使用，例如上面的例子中声明的形参name和age，可以在方法体内使用。

|  |
| --- |
| void student(string name, int age)  {  if (age == 18)  {  print(name);  }  } |

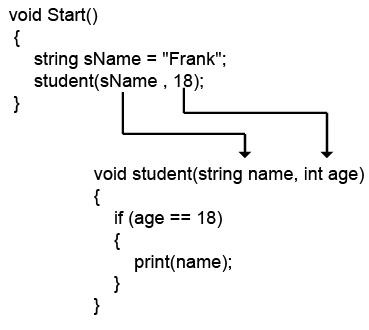
### 实参

调用方法时参数列表中传入的变量称为实参，它的作用是初始化形参。实参可以是表达式或变量。

例如，下面的代码展示了方法student的调用，它有两个实参。

|  |
| --- |
| void Start()  {  string sName = "Frank";  student(sName , 18);  }  **表达式**  **变量** |

当方法被调用时，每个实参的值都被用于初始化相应的形参，方法体随后被执行。下图阐明了实参和形参的关系



调用代码时，数据Frank和18传入了方法，方法体经过执行后，输出结果为：Frank。

## 值参数和引用参数

### 值参数

到目前为止，我们使用的参数都是值参数。使用值参数调用方法时，首先会在栈中为形参分配空间，然后把实参的值复制到形参。对形参的操作不会影响实参。

例如，下面的示例中，形参的值加5，并不影响实参的值。

|  |
| --- |
| void number(int i)  {  i += 5;  print("形参的值是" + i);  }  void Start()  {  int \_i = 20;  number( \_i );  print( "实参的值是" + \_i );  }  /\*输出结果：  \* 形参的值是25  \* 实参的值是20  \*/ |

### 引用参数

与值参数不同的是，使用引用参数不会在栈中为形参分配空间，实参把其持有的引用赋值给了形参，形参指向的空间与实参相同。因此在方法体中对形参的操作会影响到实参。

使用引用参数时需要在方法声明和调用中都使用ref修饰符，而且实参必须是变量，不能是表达式。

把上面的示例经过修改变为引用参数后，当形参的值加5时，实参的值也相应的改变了。

|  |
| --- |
| void number(ref int i)  {  i += 5;  **ref 修饰符**  print("形参的值是" + i);  }  void Start()  {  int \_i = 20;  number( ref \_i );  print( "实参的值是" + \_i );  }  /\*输出结果：  \* 形参的值是25  \* 实参的值是25  \*/ |

### 总结

值传递：传递的是栈中的内容。对于值类型，栈中的内容就是对应的数据，对于引用类型栈中的内容就是对象的引用。

引用传递，传递的是栈本身的地址，形参和实参指向的是同一个栈变量（有多个变量名）。

## 可变数组参数（params）

一般来说，参数个数都是固定的，定义为集群类型的参数可以实现可变数目参数的目的，但是.NET提供了更灵活的机制来实现可变数目参数，这就是使用params修饰符。

可变数目参数的好处就是可以在参数个数不确定情况下进行值传递，例如计算任意数字的加权和，链接任意字符串为一个字符串等。看下面例子：

|  |
| --- |
| void Start()  {  //使用方法1：直接定义一个数组，作为参数传人方法  object[] list = new object[] {"我的年龄",20, "青春年华" };  useParams(list);  //使用方法2：直接输入参数：  useParams("我的年龄", 20, "豆蔻年华");  }  void useParams(params object[] list)  {  foreach (object obj in list)  {  print(obj);  }  } |

Params参数需要注意的几点：

* params修饰的参数必须为一维数组。
* params修饰的参数数组，可以为任何类型，只要设置数组类型为object就可以。
* params必须在参数列表的最后一个，并且只能使用一次。

在下面的例子中，方法有多个参数，parmas必须在最后

|  |
| --- |
| void Start()  {    object[] list = new object[] {"我的年龄",20, "青春年华" };  useParams("hao",list);      }  void useParams(string name,params object[] list)  {  foreach (object obj in list)  {  print(name +obj);  }  } |

## out参数

除了将参数单向传入一个方法（传值），或者同时将参数传入和传出方法（传引用）之外，还可以将数据从一个方法体内部单向传出。为此，需要使用关键字out来修饰参数类型，这种参数叫输出参数。

下面的代码展示了一个out参数的示例。当Number的方法体中的引用参数改变时，实参也改变了。

|  |
| --- |
| void Number(out int i)  {  i = 7; //形参赋值  }  void Start()  {  int x = 5;  Number(out x);  print(x); //输出实参  }  /\*输出结果  \* 7  \*/ |

Out参数中传入的变量是否初始并不重要，不管是否初始化，out参数都当成没有初始化来处理。所以在函数中，首先要对传人的参数进行赋值才能进行运算。

例如，在下面的代码中，outparams函数中，第一步就需要对传人的赋值，不赋值就会出现错误提示： The out parameter 'p' must be assigned to before control leaves the current method。

|  |
| --- |
| void Start()  {  int i = 9; //是否初始化不重要，传人out参数后，会被当作未初始化。  outparams(out i);  }  void outparams(out int p)  {  p = 12;// 首先要对传人的参数赋值（初始化）。  print(p);  } |

与out参数不同，ref参数在传入前必须初始化，否则会出现如下提示： Use of unassigned local variable 'i'

|  |
| --- |
| void Start()  {  int i = 1 ;//必须初始化，才能传人ref参数中  outparams(ref i);  }  void outparams(ref int p)  {  print(p);  } |

Out参数一般用在函数有多个返回值的场所。例如：int.TryParse（string s, out int result）函数，它返回两个值，第一个返回值为bool类型，当第一个参数s能转换为int时，值为真，否则为false；第二个返回值通过out参数返回，当转换成功时，它返回转换后的值，当转换失败时，返回的值为0。如下：

|  |
| --- |
| string str1 = "abc";  string str2 = "123";  int s1;  int s2;  if (int.TryParse(str1, out s1))  {  print("s1的值为：" + s1);  }  else  {  print("s1的值为：" + s1 + " 转换失败");  }  if (int.TryParse(str2, out s2))  {  print("s2的值为：" + s2);  }  else  {  print("s1的值为：" + s2 + " 转换失败");  }  /\*-------------------输出结果-------------------  \* s1的值为：0 转换失败  \* s2的值为：123  --------------------------------------------- \*/ |

而ref参数常用来改变外部的值。例如，在下面的代码中，创建了一个函数swap可以实现参数值之间的交互，通过ref传人数据，经过加工后，修改后的值会返回给原来的变量。

|  |
| --- |
| void swap(ref int x, ref int y)  {  int temp = x;  x = y;  y = temp;  }  void Start()  {  int x = 10;  int y = 20;  swap(ref x, ref y);  print("x的值为：" + x);  print("y的值为：" + y);  }  /\*-------------------输出结果-------------------  \* x的值为：20  \* y的值为：10  --------------------------------------------- \*/ |

## 递归

除了调用其它的方法，方法还能调用自身，这称为递归（recursion）。下面的代码展示了一个递归调用的示例。

|  |
| --- |
| void intVal(int i)  {  if(i == 0)  {  return;  }  print(i);  intVal(i-1);//递归，调用自身。  }  void Start()  {  intVal(3);    } |

用递归实现一个方法时，要注意无限递归（infinite recursion）的错误。假如方法持续地调用自身，永远达不到标志递归结束的位置，就会产生无限递归。所以，必须仔细地检查每一个使用了递归的方法，验证递归调用是有限的而非无限的。

## 方法重载

一个类中有两个或更多相同名称的方法，这叫做方法重载（method overload）。对于重载的方法来说，每个方法必须有一个和其他方法不相同的签名（signature）。

方法的签名在方法声明的方法头中，由以下信息组成：

* 方法的名称
* 参数的数目
* 参数的数据类型和顺序
* 参数修饰符

返回类型和形参的名称不是签名的一部分，认为它是前签名的一部分是一种常见的错误。下图展示了签名的组成：

|  |
| --- |
| int intVal(int i, string Str)  **签名**  **不是签名的一部分** |

下面的代码展示了intVal方法的重载。当参数的数目、类型或类型顺序不同时，可以实现重叠，而参数数目、类型或类型顺序相同时，编译器会产生错误信息。

|  |
| --- |
| int intVal(int a)  {  return a;  }  int intVal(int a, int b)  {  return a + b;  }  int intVal(int b, int a)//调用时会发生错误，因为方法名称不属于签名，会与前面的第二个方法相混淆。  {  return b + a;  }  int intVal(int a, string b)  {  return a;  }  long intVal(int a)//调用时会发生错误，因为返回类型也不属于签名，会与第一个方法相混淆  {  return a;  } |

重载只与方法名和参数有关：方法名必须一致，参数必须不一致。

## 命名参数

当方法有多个多个重载时，为了明确值赋值给了那个参数，可以用命名参数的方式进行赋值，而且赋值时可以打乱参数的顺序。

例如有以下一个方法：

|  |
| --- |
| public string MyInfo(int Age, string Name, string Address)  {  return string.Format("My Name is {0},Age is{1}, Address is{2}", Age, Name, Address);  } |

调用时用命名参数的方式调用：

|  |
| --- |
| MyInfo(Name: "hao", Address: "ZZ", Age: 18); |

# 数组

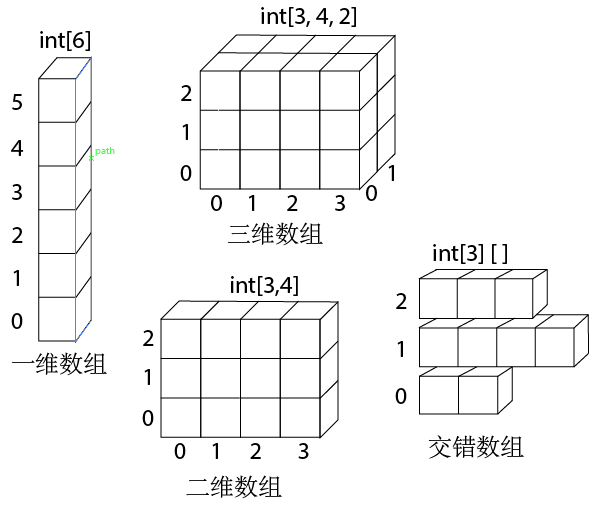
在编程中，会经常碰到一些数据需要放在一起使用，比如说编制一个清单，包含一个班级所有学生的名称，那么这个清单就是一个数组。

利用数组，可以在单个变量中存储同一种类型的的多个数据项，而且还可以利用索引来单独访问这些数据项。

C#主要由三种数组：

* 一维数组
* 多维数组（矩形数组）
* 交错数组

下图演示了C#中的各种数组：



最后需要注意的是，C#不支持动态数组，数组一旦创建，大小就固定了，不能添加或删除元素。此外，c#中数组是基于零（zero based）的，它的索引是从0开始的，数组最后一个元素的索引值要比数组元素的总数小1。

|  |
| --- |
| 术语 |
| * 元素: 数组的独立数据项被称为元素。数组的所有元素必须是相同类型的，或继承自相同的类型。 * 维度数：数组可以有任何为正数的维度数。数组的维度数称作秩（rank）。 * 维度长度：数组的每一个维度都有一个长度，就是这个方向的位置个数。 * 数组长度：数组中所有维度的元素的总和称为数组的长度。 | |

## 数组的声明

在C#中，使用方括号来声明数组变量。首先指定数组中数据项的类型，后跟一对方括号，再输入变量名。

下面的代码将名为number的变量声明为int类型的数组

|  |
| --- |
| int[,] number; |

数组声明中的int表明了数组中存储的元素的类型。方括号内的逗号是秩说明符，它们指定了数组的维度数，维度数是逗号数量加1。在本例中，方括号内有1个逗号，表示我们声明的是一个二维数组。

下面代码展示了一维数组与多维数组声明的示例。需要注意的是：

* 可以使用任意多的秩说明符。
* 不能在方括号内放数组维度长度。
* 数组声明后，维度数就固定了。然而，维度长度直到实例化时才会确定。

|  |
| --- |
| int[] number;//声明一维数组  int[,] DNumber;//声明二维数组  int[, ,] TNumber;//声明三维数组 |

## 数组实例化

实例化数组，需要使用数组创建表达式。数组创建表达式由new运算符，数据类型，一对方括号构成。方括号中以逗号分隔每一个维度的长度。

下面代码是一维数组与多维数组实例化示例：

|  |
| --- |
| int[] number = new int[5];//number一维数组有5个元素  int[,] DNumber = new int[3,4];//DNmuber二维数组有12个元素  int[, ,] TNumber = new int[2,3,4];//TNumber三维数组有24个元素 |

## 数组初始化

实例化后，每一个元素自动初始化为类型的默认值。对于预定义的类型，整型默认值是0，浮点型的默认值是0.0，布尔型的默认值是false，而引用类型的默认值是null。

在C#中可以在声明数组的同时，初始化数组元素的值。下面代码展示了数组声明同时进行赋值的语法：

|  |
| --- |
| int[] i\_1 = new int[3] {1, 2, 3 };//完整的初始化  int[] i\_2 = new int[] { 1, 2, 3 };//数组创建表达式省略维度长度  int[] i\_3 = {1, 2, 3 };//省略数组创建表达式 |

多维数组的初始化使用嵌套的、逗号分隔的初始化列表进行初始化。初始化列表嵌套在花括号内。对于嵌套、分组和逗号的使用，有以下的技巧：

* 逗号用作元素和分组之间的分隔符。
* 逗号不在左花括号之间使用。
* 逗号不在右花括号之前使用。
* 从左向右读秩指示符，指定最后一个数字作为“元素”，其他数字作为“分组”。

下边的代码是多维数组初始化的示例：

|  |
| --- |
| int[,] i\_1 = new int[3,2] {{1, 2}, {2, 3}, {4, 5} };//二维数组初始化  //三维数组初始化  int[,,] i\_2 = new int[3,4,2]{  {{1, 2}, {3, 4}, {5, 6}, {7, 8}},  {{9, 10}, {11, 12}, {13, 14}, {15, 16}},  {{17, 18}, {19, 20}, {21, 22}, {23, 24}}  }; |

## 数组元素的总数

C#中的数组是由System.Array类派生而来的引用对象，它提供一些公共的属性和方法，对数组的操作提供了很大的帮助。其中，Length的返回值是一个32位的整数，表示Array所有维数中元素的总数。

|  |
| --- |
| int[] number = new int[] { 10, 20, 40, 50 };//定义数组  int len = number.Length; //获取元素的总数  print(len); |

## 访问数组元素

数组中的元素是有序的，每一个元素对应一个唯一的下标。第一个元素的下标是0，最后一个元素的下标是数组长度减1。

数组元素访问的代码如下：

|  |
| --- |
| int[] number = new int[] { 10, 20, 40, 50 };  int len = number.Length;//  print(number[0]);//打印第一个元素的值。输出结果为：10  print(number[len - 1]);//打印最后一个元素的值。输出结果为：50 |

### 数组的遍历

使用for循环可以遍历数组元素。代码如下：

|  |
| --- |
| int[] number = new int[] { 10, 20, 40, 50 };  int len = number.Length;  //遍历数组元素  for(int i = 0; i<len;i++)  {  print(number[i]);  }  } |

## 修改数组元素的值

数组元素的修改与元素的访问相似，只需要通过数组名与下标即可。

在下面的代码中，如果数组的元素大于0则将该位置的元素的值加1，如果元素为负则将指定位置的元素的值减1。

|  |
| --- |
| int[] number = new int[] {10,20,-10 };  int len = number.Length;//  for (int i = 0; i < len; i++)  {  //元素的值大于0，则元素值加1。  if (number[i] > 0)  {  number[i]++;//修改第i+1个元素的值  }  //元素的值小于0，则元素值减1。  else  {  number[i]--;//修改第i+1个元素的值  }  } |

## 清空数组

|  |
| --- |
| string[] s = new string[] { "乔丹", "威廉姆斯", "科比" };  Array.Clear(s, 0, s.Length);//清空数组中的内容（数组还在，只是每个值变为null）  for (int i = 0; i < s.Length; i++)  {  Console.WriteLine(s[i]+"☆");  } |

# 字符串

字符串是c#里常用的一个引用型数据类型。它直接从Object类型继承而来。

字符串是一种相对特殊的类型。在应用上，String类型表现为值类型，它的使用方法与值类型一样。，在内存上，String为引用类型，存储在托管堆中。

## 修改字符串内容

因为它的特殊性，它具有以下几个特征：

* 字符串序列一经创建，就不能更改。在需要修改字符串的内容时，是把原来的字符串丢掉，再创建一个字符串。GC会自动回收不再使用的字符串。

例如，下面的代码演示了如何修改字符串内容

|  |
| --- |
| string str = "hello" ; //把hello所在内存的首地址赋值给str变量  string temp = str; //把str变量中存储的hello所在内存的首地址赋值给temp变量  str = "yello"; //在堆中创建一个新的字符串对象“yello”，并把它的地址赋值给str  print(str); //打印str中存储的地址所指向的对象即“yello”。  print(temp); //打印temp中存储的地址所指向的对象即“hello”。 |

* 字符串可以看作是特殊的字符数组，C#提供字符串和字符数组之间的转换方法。参考（10.2字符串是特殊的字符数组）

## 访问字符串中某一个字符

|  |
| --- |
| string str = "hello" ;  Console.WriteLine(str[0]); //输出为： |

## 修改某一个字符的值

可以把字符串看作特殊的字符数组。使用ToCharArray()方法可以把字符串中的内容提取到char类型的数组中，然后修改此数组中的某些元素，最后，用new string(char[]把数组转换为字符串。

实例代码如下：

|  |
| --- |
| string str = "hello" ;  char[] temp = str.ToCharArray(); // 转换为字符数组  temp[0] = 'Y'; //修改数组中第一个元素的值  str = new string(temp); //字符数组转换为字符串对象，并把其地址赋值给str  print(str);  /\*-----------------输出结果----------------------  \* Yello \*  ----------------------------------------------\*/ |

下面的代码中把字符串反向输出：

|  |
| --- |
| string str = "哈喽！我是高大上";  char[] strArray = str.ToCharArray();  for (int i = strArray.Length-1; i >= 0; i--)  {  print(strArray[i]);//输出：上大高是我！喽哈  } |

## 替换字符串

当需要修改一段字符串中的某几个字符串时，可以使用replace来实现。

比如在下面的例子中，我们要打印： 毛泽东对张茜 说：“邓小平是个好同志，他对中国革命和世界革命所作的贡献，是已经下了结论的”。如果我们想要把邓小平改为陈毅。如果重新赋值就需要全部修改，而转换为数组进行修改，我们又不知道具体的下标是什么，这时用replace就可以快速修改。

|  |
| --- |
| string str = "毛泽东对张茜说：“邓小平是个好同志，他对中国革命和世界革命所作的贡献，是已经下了结论的。” ";  str = str.Replace("邓小平", "陈毅"); //修改邓小平为陈毅  print(str); |

## 子字符串

Substring返回一个新的字符串，它是此字符串的一个子字符串。Substring有1个重载,因此可以有2中使用方式。

* substring(int StartIndex)

StartIndex为起始位置，即子字符串从StartIndex个字符开始，一直到字符串结束。字符的索引从0开始，如下面的代码：

|  |
| --- |
| string str = "http://www.microsoft.com/";  str = str.Substring(7);//字符索引的起始位置为7，子字符串为：www.microsoft.com/  print(str); |

* substring(int StartIndex, int Length)

第一个参数为字符串的起始位置（索引从0开始），第二个参数为子字符串的长度。以上面的代码为例，仅想显示microsoft，可以使用下面的方式：

|  |
| --- |
| string str = "http://www.microsoft.com/";  str = str.Substring(11,9);//子字符串为microsoft |

## Contains、StartWith与EndsWith

* Contains

此方法用来检查字符串是否包含某些字符。例如，我们想要屏蔽某些不和谐的用语，可以使用contains来判断真假。下面的例子中，判断字符串中是否有肉欲，如果有则给出警告。

|  |
| --- |
| string Plato = "柏拉图式恋爱，也称为柏拉图式爱情，追求心灵沟通，排斥肉欲。";  if(Plato.Contains("肉欲"))  {  print("含有敏感词汇，请文明用语");  } |

* StartWith(string value)

此方法用来判断字符串是否以子串value开始。比如，在下面的代码中判断一个字符串是否为网页，只需要判断师范以http://或https://的形式开始。

|  |
| --- |
| string website = "http://www.microsoft.com";  if (website.StartsWith("http://") || website.StartsWith("https://"))  {  print("这是个网站");  } |

* EndsWith(string value)

此方法使用方式与StartWith，用来判断字符串是否以字串value结束。

## 获取子字符串的位置

IndexOf（string value）：可以获得子字符串value第一次出现的位置。该值为value中第一个字符出现的位置。

|  |
| --- |
| string website = "哈喽！我是高大上";  int position = website.IndexOf("高大上");//检查高大上出现的位置。  print(position);//输出结果为：5 |

如果value中的子字符串不在字符串中，返回值为-1。

|  |
| --- |
| string website = "哈喽！我是高大上";  int position = website.IndexOf("矮矬穷");  print(position);//输出结果为-1 |

下面的代码使用IndexOf和Substring可以从Email中提取出用户名和网址。

|  |
| --- |
| string str = "hyfdbd@163.com";  int cPosition = str.IndexOf("@");  print("用户名：");  string username = str.Substring(0,cPosition);  print(username);  print("\n");  string website = str.Substring(cPosition + 1);  print("网址：");  print(website); |

下面的例子中使用Index和Replace，可以修改17个字符后的内容为特定的形式。

|  |
| --- |
| string str = "这个文章的标题有点长，马上就要超过了20个字符 作者：西瓜皮";  if(str.IndexOf(" ")>20)//如果空格大于20  {  string temp = str.Substring(17);  str = str.Replace(temp, "...|西瓜皮");  }  print(str); |

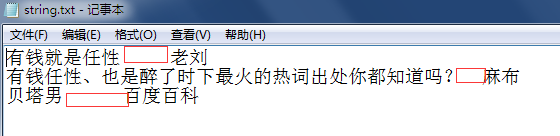
## Split分割字符串

String.Split()方法以指定的字符串或Unicode字符元素作为分割符，把字符串分割为字符串数组。比如可以将空格作为分隔符，把字符串“my name is hyfdbd”分割为一个数组{my,name,is,hyfdbd}，从而可以去掉空格等。

下面的例子中，可以把字符串str中的单词反向输出。

|  |
| --- |
| string str = "my name is hyfdbd";  string[] strArray = str.Split(' ');//以空格作为分隔符，注意是单引号。  for (int i = strArray.Length - 1; i >= 0; i-- )  {  print(strArray[i] + ' ');//输出结果：hyfdbd is name my  } |

Split还能用RemoveEmptyEntries来去除空元素。比如在一个文本文件中，我们以空格对文件进行风格，但是中间的空格分隔符不具体，我们不知道文本输入人员到底用几个空格对文本进行编辑。如下图：



为了使每一行都分割为包含两个元素的数组，我们使用了RemoveEmptyEntries。具体代码如下：

|  |
| --- |
| //使用ReadAllLines，按行读取文件：d:\string.txt，每一行作为一个字符串数组的元素。  string[] lines = System.IO.File.ReadAllLines(@"d:\string.txt", Encoding.Default);  foreach (string str in lines)  {  //第一个参数是char数组，定义分割符，可以有多个分割符，这里只有一个分割符即空格。  //第二个参数当值为RemoveEmptyEntries时，会把包含空字符的数组元素剔除。也就是去除了空格  string[] temp = str.Split(new char[] { ' '}, StringSplitOptions.RemoveEmptyEntries);  string title = temp[0];  string author = temp[1];  if(title.Length>17)  {  //min方法返回2个参数中的最小值。  title = title.Substring(0, Math.Min(17, title.Length));  title = title + "...|";  print(title + author);  }  else  {  title = title + "...|";  print(title + author);  }  } |

## Trim去空格

Trim方法可以去掉字符串前后的空格。

例子如下：

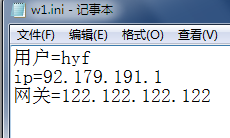
|  |
| --- |
| string str = " hao are you ";  print(str.Trim());//输出结果：hao are you |

## 字符串与字节的转换

|  |
| --- |
| string msg = "Hello! World";  //字符串转换为字节  byte[] bytes = System.Text.Encoding.UTF8.GetBytes(msg);  //字节转换为字符串  string str = System.Text.Encoding.UTF8.GetString(bytes); |

## 综合案例

下面的代码中，从ini文件中读取配置，并显示配置项的值。Ini文件与代码如下



|  |
| --- |
| string GetConfigValue(string filename, string itemname)  { //读取文件  string[] lines = System.IO.File.ReadAllLines(filename, Encoding.Default);  //遍历每一行  foreach(string str in lines)  {  //如果某一行以itemname开始，则执行if语句块  if(str.StartsWith(itemname))  {  int index = str.IndexOf('=');//查询=号所在的索引位置  string value = str.Substring(index + 1);//获取子字符串：从=号以后到结尾  return value.Trim();//返回值  }      }  return "false";  }  void Start()  {  string p = GetConfigValue("d:\\w1.ini", "网关");  print(p);  } |

## 字符串拘留池（intern pool）

  在一个程序中，如果将同一个字符串赋值给不同的字符串引用，那么系统会为多次分配内存空间，然而这些内存中存储的是同一个字符串。这不仅浪费了宝贵的内存，还极大程度的降低了系统的性能。为了改善程序的性能，.net提出了字符串拘留池机制来解决这一问题。

### 字符串拘留池的原理

一旦使用了字符串拘留池，当CLR启动的时候，会在内部创建一个容器（表示为哈希表的键值对），容器的键是字符串内容，值是字符串在托管堆上的引用。当一个新的字符串对象需要分配时，CLR首先检测容器中是否包含了该字符串对象，如果已经包含，就直接返回已经存在的字符串对象的引用；如果不存在，则新分配一个字符串对象，并把它添加到内部容器中同时返回该字符串对象的引用。拘留池只对字符串常量其作用，是当用new关键字显示的申明分配一个字符串对象时，字符串拘留池机制不会再起作用。

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  namespace ConsoleApplication1  {  class Program  {  public static void Main()  {  string a = "abcde";  string b = "abcde";  Console.WriteLine(Object.ReferenceEquals(a, b));  char[] c ={'a','b','c','d','e'};  string cc = new string(c);  Console.WriteLine(Object.ReferenceEquals(a, c));  }  }  } |

输出结果：



      对于CLR保存的字符串拘留池的容器，可以通过System.String类型的两个静态方法进行访问：

        public  static  String  Intern (String s);

                  返回字符串s在字符串拘留池中对应的引用，如果该字符串不在字符串拘留池中，那么会新分配一个字符串对象并添加到字符串拘留池中同时返回该字符串的引用。

       public  static  String  IsInterned (String s);

               功能与上个方法相似。只是当字符串s不在字符串拘留池中时，不会分配新的对象，并返回null。

### 缺点

     当在一个程序中刚好较少地或根本没有使用到重复的字符串时，那么字符串拘留池机制就会带来负面效应。因为CLR额外的保留了一个存储字符串池的容器，并且在每次进行字符串赋值时都要额外的检查这个容器。考虑到这点，.NET提供了字符串拘留池的开关接口，如果程序集标记了一个System.Runtime.CompilerServices.CompilationRelaxationsAtrribute特性，并指定了一个NoStringInterning标志，那么CLR将不会再使用字符串拘留池机制。需要注意的是CLR v.4.0 只在使用ngen.exe编译时才起作用。

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Runtime.CompilerServices;  [assembly: CompilationRelaxationsAttribute(CompilationRelaxations.NoStringInterning)]  namespace ConsoleApplication1  {  class Program  {  public static void Main()  {  string a = "abcde";  string b = "abcde";  Console.WriteLine(Object.ReferenceEquals(a, b));  char[] c ={'a','b','c','d','e'};  string cc = new string(c);  Console.WriteLine(Object.ReferenceEquals(a, c));  }  }  } |

### 添加变量到暂存池

用String.Intern(string)方法，可以把动态字符串添加到哈希表暂存池中。

|  |
| --- |
| String a = new string(new char[] { 'a','b','c' });  string c = new string(char);  String.Intern(a);  String b = "a";  Console.WriteLine(object.ReferenceEquals(a,b)); |

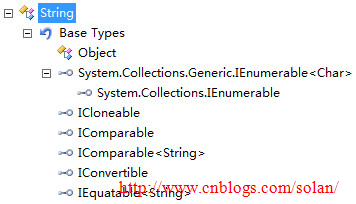
## 字符串不可变特性

字符串不可以变和暂存池带来的问题时当进行字符串拼贴或字符串值常常需要改变时，会造成暂存池的污染（新的字符串常量会添加到暂存池中）以及内存的浪费（值改变需要重新分配内存）

## 字符串（不可变、驻留机制、池、比较）

第一节 字符串的定义（System.String）

字符串被定义为System.String类型的对象，既然它是引用类型，那么一个未初始化的对象声明将保留为null，并且它的内存只能在堆上分配。它在内部维护的是字符Char的集合，所以它有一个属性Length来表示Char集合中元数的个数。来看一下String类型的定义：



String实际上是继承了System.Object类型，同时还实现了一系列接口，如Ienumberable、ICompareable等，所以字符串提供了对字符集合、比较等相关的操作。

尽管它是引用类型，但是编译器不允许使用new根据一个文本常量来创建一个字符串对象，而是必须使用简明的声明语法来声明及初始化，对字符串的初始化值是直接被编译进元数据的。比如如下定义一个字符串变量：

string name1 = "Jack";

IL:

.method **public** hidebysig specialname rtspecialname

instance **void** .ctor() cil managed

{

*// 代码大小 19 (0x13)*

.maxstack 8

IL\_0000: ldarg.0

IL\_0001: ldstr "Jack"

IL\_0006: stfld **string** ConsoleApp.Example07.Code\_07::name1

IL\_000b: ldarg.0

IL\_000c: call instance **void** [mscorlib]System.Object::.ctor()

IL\_0011: nop

IL\_0012: ret

} *// end of method Code\_07::.ctor*

我们知道，通常对于引用类型创建对象是使用newobj指令，但上面的并没有使用该指令，只是使用ldstr指令加载了字符串“Jack”，从IL\_0000 - IL\_0006可以看到是直接加载“Jack”串赋给变量，这是CLR的一种特殊的构造方式。

第二节 字符串的不可变性

字符串 对象一旦创建，在整个进程的生命周期中是不可变的，无法对其进行加长、缩短、改变等操作，既然它不会变，所以也就不存在线程同步的问题，哪怕是皇天老儿创建的线程都无法对其进行改变。如下代码：

**string** str1 = "Jim";

**string** str2 = str1;

Console.WriteLine(**object**.ReferenceEquals(str1, str2)); *//True*

str1 += " China";

Console.WriteLine(**object**.ReferenceEquals(str1, str2)); *//False*

在第一次调用object.ReferenceEquals方法比较的str1和str2，它们指向的是同一个字符串对象引用，所以结果为true，而str1 += " China";的过程是重新创建了一个对象，且把新的对象引用赋给str1，此时str1与str2指向的不是同一个对象引用，所以在第二次调用object.ReferenceEquals方法时返回的是false。无论是使用+=操作符还是其他的对字符串修改的方法，都会引起重新创建字符串对象，并且复制旧的字符串到新的内存区，而不是我们常说的“对XX字符串进行修改”，如果非要说“改变”，那就是对对象引用的改变。对于str1 += " China";操作，CLR会执行以下操作：

1） 开辟新的足够大的临时存储区内存来容纳str1和” China”；

2） 复制str1串到临时区的开始处；

3） 复制” China”到临时存储区的结尾处；

4） 释放str1旧有的内存；

5） 为str1再一次分配内存区；

6） 将临时存储区内的字符值复制到4)新开辟的内存区，将str1指向这个内存区的引用。

所以对字符串的连接操作会大大损伤性能。我们在下面会讲到.NET 提供的一个专门对付字符串连接的类StringBuilder。

第三节 字符串的驻留

通过前面的描述，我们已经知道字符串的内存是分配在托管堆上，且它是不可改变的，而在编程中，我们会大量使用字符串，这就会导致不停地创建字符串对象，不停地分配内存，并且很有可能不停地执行垃圾回收，如此以来会大大损伤性能,所以CLR对字符串进行了特殊的优化机制，下面我们来对这些机制及特性进行描述。

字符串驻留是CLR提供的一种提高性能的对待字符串的机制，它保证在一个进程内的某个字符串在内存中只分配一次。看以下代码：

**string** str1 = "abc";

**string** str2 = "abc";

Console.WriteLine(**object**.ReferenceEquals(str1, str2)); *//True*

明明声明了两个对象str1和str2，调用object.ReferenceEquals方法返回的是True，为什么它们指向的是同一个引用呢？这就说明了CLR的字符串驻留，相同的字符串在托管内存中只分配一次，再次声明相同的字符串对象时，会将后来一次的声明指向第一次声明所引用的对象。那么CLR 如何保证做到的呢？原来，在CLR初始化时创建一个内部的哈希表，我们知道哈希表在处理表内数据时是非常快的，这个表相当于一个字典表（Dictionary<TKey,TValue>），键就是字符串，而值是指向托管堆中该字符串对象的引用，当在声明一个字符串时，会调用对象的Intern方法，该方法接收一个string对象，它会先在哈希表中检查该字符串是否存在？如果存在，则返回这个字符串对应的对象引用；否则，将创建该字符串的副本，并将副本添加到哈希表中，最后返回对该副本对象的引用。String类还提供了一个IsInterned方法，该方法会根据字符串在哈希表中检查是否已经存在相同的串，如果存在，则返回该字符串对象的引用，否则，返回null，但它是它不会向哈希表中添加字符串。我们对上面的代码进行改造：

**void** TestIntern()

{

**string** str1 = "abc";

**string** str2 = "abc";

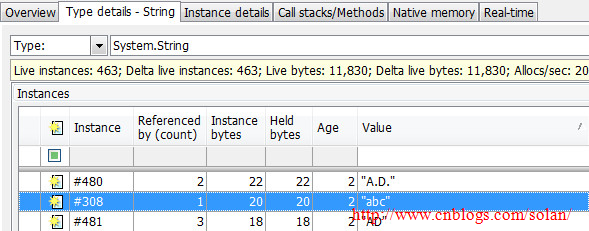
Console.WriteLine(**object**.ReferenceEquals(str1, str2)); *//True*

str1 += str2;

Console.WriteLine(str1);

}

接下来通过内存分析器来看一下字符串驻留：



可以看到字符串”abc”是有驻留的。从前面的讲解中我们知道，+=操作是要重新创建对象的，但CLR对临时计算的新对象“abcabc”没有进行驻留呢。继续改造上面的代码：

**string** str1 = "abc";

**string** str2 = "abc";

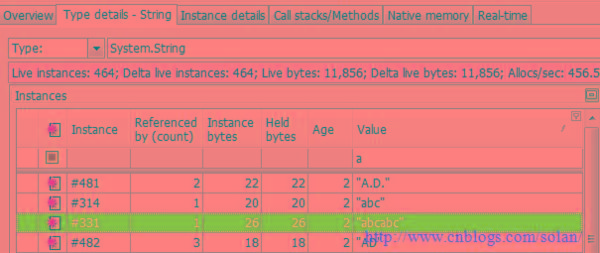
Console.WriteLine(**object**.ReferenceEquals(str1, str2)); *//True*

str1 += str2;

str1 = **string**.Intern(str1);

Console.WriteLine(str1);

这次我们使用了string.Intern方法，再用内存分析器看一下：



这次我们看到字符串”abcabc”是进行了驻留，这里就验证了刚才上面对驻留机制的讨论。

有一点要注意，尽管String.Intern(string)方法的字符串参数（上面代码中的str1）被垃圾回收器回收，但是CLR已将这个str1的副本添加到哈希表中，垃圾回收器是无法对哈希表引用的字符串进行回收。

字符串驻留也有发“懵”的时候，看以下代码：

**void** Test()

{

string str1 = "abc";

string str2 = **new** string(**new** **char**[] { 'a', 'b', 'c' });

Console.WriteLine(object.ReferenceEquals(str1, str2)); *//False*

}

对于变量str2最终的字符串也是”abc”，可是为什么这次object.ReferenceEquals方法返回了False呢？原因是CLR会为new string(char[])创建的字符串对象会重新分配内存，不再使用字符串驻留机制对它进行处理，于是str1和str2就指向了不同的对象引用。

第四节 字符串池

字符串池与字符串驻留机制是分不开的，最能体现字符串池的是在在C#编译器编译的过程中，在编译过程中，同样的字符串（比如上面代码中的”abc”）会被程序中的很多地方使用，通过第一节我们已经知道字符串的声明是直接将字符中写入元数据中，如果编译器在每个使用的地方都将该相同的字符串写入元数据中，则会大大增加元数据的体积，且也没有必要。所以编译器会只在模块的元数据中写入该字符串一次，并将引用该相同字符串的代码都修改为指向这同一个字符串对象，如此以来，就会大大减小元数据的体积。

第五节 字符串的比较

对字符串的比较，通常有两层意义，一个是判断两个字符串对象是否具有相同的引用，另一个是判断两个字符串对象是不具有相同的“值”，我们一般的编程中经常使用后者。

（1） Unicode编码与字符

为了解决使用相同的字符集表示不同的语言，于是一群人就捣鼓出了Unicode，Unicode对每一个字符都提供了唯一的值，它不依赖于任何平台和任何区域语言。Unicode采用2个字节的编码方式，可以表示65536个字符，就是我们常说的16位Unicode编码，然而，仅中文就有85000多个字符，所以16位的Unicode不能满足语言文化的需要。当然如果采用32位的Unicode编码一定能满足，但32位编码的每个字符占4个字节，所以Unicode定制了另一个使用代理对的机制来满足各种语言文化的需要。

.NET Framework使用16位的Unicode编码，每个字符对应一个确定的Unicode码值。在第一节我们已经知道，字符串是由字符组成，所以一个字符串是由一系列的Unicode码组成。我们通常用Int32值来表示一个字符的编码值，如：’a’：97、’A’：65。如果你感兴趣，可以使用如下代码来测试看看每个数值所代表的字符是什么，当然，无对应数值，可能无法转换：

**for** (**int** i = 1; i < 10000; i++)

{

**char** temp = (**char**)i;

Debug.WriteLine(temp + " " + i);

}

计算机只能识别0/1，任何一个数字都可以用0/1对其进行编码，每个字符都是由Unicode码值来表示，所以计算机就能“识别”出所有字符，而字符是组成字符串的元素，所以计算机进而能“识别”出字符串。

字符文本与区域语言文化有很大的依赖关系。

（2） 区域语言文化

由于全世界各地的人类文化不同，也就导致的语言文化的不同，这就是区域语言文化的不同。计算机系统使用国际化标准来处理各种语言文化的差异。.NET Framework为了方便处理各种语言文化，提供了一个System.Globalization.CultureInfo类，它提供对特殊文化信息的支持，如文化名称、相关语言、国家/区域等。如en代表英文、en-CA代表加拿大、zh-CHS代表中文简体。

一个应用程序一般即要处理国际化数据，也要处理本地化数据，CultureInfo为处理这两类数据扮演了重要角色，为了完美地支持这两类数据处理，CultureInfo类提供了两个重要的属性CurrentUICulture和CurrentCulture。CurrentUICulture的值决定了如何加载窗体资源及窗体元素以什么语言来显示。CurrentCulture决定了除CurrentUICulture外的其他方面，如日期格式、数字格式、货币符号、字符串大小写及比较等。CurrentUICulture和CurrentCulture在应用的线程级设定，如果未设定，那么系统将从Windows中获取一个值来进行实初始化，这个值通常在控制面板的语言和区域中设置。

（3） 字符串的比较

我们知道，字符串是由字符组成的，而每个字符可以由一个Int32值来表示，我们当然可以比较两个Int32的值，同样也可以比较两个字符的“大小” ，进而可以比较两个字符串的“大小”，事实上字符串都是表示一系列文本内容的，它们不可能用大小来衡量，只能长度对其本身的特性进行一方面的描述，我们通常所说的比较，是对其内包含的字符对应的Int32值进行比较。在对字符或串排序的时候，这个比较很有用，另外一个，我们通常是比较两个字符串是否相等。

注意：字符串的比较，是按顺序逐个比较每个字符的Int32值。

字符串的比较通常有以下几种方式：

比较符号 ==、实例级和静态的Equals方法、CompareTo方法、String.Compare方法、String.CompareOrdinal方法

字符串与语言文化有很大的依赖关系，所以任何一个的字符串，都会直接或间接的使用某种语言文化信息。下面我们分别介绍一下每个比较方法。

**a) 等于号==**

如下代码：

**void** TestEqualto()

{

**string** str1 = "abc";

**string** str2 = "def";

**bool** chk = str1 == str2;

}

我们来看一下IL：

.method **private** hidebysig instance **void** TestEqualto() cil managed

{

*// 代码大小 22 (0x16)*

.maxstack 2

.locals init ([0] **string** str1,

[1] **string** str2,

[2] **bool** chk)

IL\_0000: nop

IL\_0001: ldstr "abc"

IL\_0006: stloc.0

IL\_0007: ldstr "def"

IL\_000c: stloc.1

IL\_000d: ldloc.0

IL\_000e: ldloc.1

IL\_000f: call **bool** [mscorlib]System.String::op\_Equality(**string**,

**string**)

IL\_0014: stloc.2

IL\_0015: ret

} *// end of method Code\_07::TestEqualto*

它是调用了op\_Equality方法，再来看看op\_Equality方法的IL:

.method **public** hidebysig specialname **static**

**bool** op\_Equality(string a,

string b) cil managed

{

.custom instance **void** System.Runtime.TargetedPatchingOptOutAttribute::.ctor(string) = ( 01 00 3B 50 65 72 66 6F 72 6D 61 6E 63 65 20 63 *// ..;Performance c*

72 69 74 69 63 61 6C 20 74 6F 20 69 6E 6C 69 6E *// ritical to inlin*

65 20 61 63 72 6F 73 73 20 4E 47 65 6E 20 69 6D *// e across NGen im*

61 67 65 20 62 6F 75 6E 64 61 72 69 65 73 00 00 ) *// age boundaries..*

*// 代码大小 8 (0x8)*

.maxstack 8

IL\_0000: ldarg.0

IL\_0001: ldarg.1

IL\_0002: call **bool** System.String::Equals(string,

string)

IL\_0007: ret

} *// end of method String::op\_Equality*

我们可以看到op\_Equality方法内部调用的是Equals方法，也就是说，只要使用==来比较两个字符串，最终调用的还是Equals方法，二者是等价的。

**b) Equals**

Equals有两种：实例方法和类方法，它们是普通的序号比较。来看一下两者的实现。

**public** **bool** Equals(**string** **value**)

{

**if** (**this** == **null**)

{

**throw** **new** NullReferenceException();

}

**if** (**value** == **null**)

{

**return** **false**;

}

**return** (**object**.ReferenceEquals(**this**, **value**) || EqualsHelper(**this**, **value**));

}

**public** **static** **bool** Equals(**string** a, **string** b)

{

**return** ((a == b) || (((a != **null**) && (b != **null**)) && EqualsHelper(a, b)));

}

可以看到，实例方法进行了一次引用的比较，判断两个对象是否指向了同一个对象地址；静态方法是判断两个对象是否相等。对于两个方法，如果前面部分不成立，则它们会继续调用EqualsHelper方法进行逐字符比较。逐字符比较是先比较第一个字符的码值，再比较第二个字符的码值，依次类推，比较过程中以第一个字符串的长度为基准。比如：

**char** a = 'a'; *//97*

**char** b = 'b'; *//98*

**char** c = 'c'; *//99*

"ab"<"ac"

"abc">"ab"

"ab"<"abc"

"ab"="ab"

"ac">"ab"

但是，如果要执行区域敏感规则的比较，可能就不同了，比较中可能会为非字母数字的 Unicode 字符分配特殊权重，使用字词排序规则和特定区域的约定

**c) CompareTo**

CompareTo(string)方法是拿一个字符串对象与另一个串对象进行比较，返回一个Int32值，如果前者的码值小于后者码值，则返回-1，如果相等，则返回0，如果前者码值大于后者码值，则返回1。来看一下它的定义：

**public** **int** CompareTo(**string** strB)

{

**if** (strB == **null**)

{

**return** 1;

}

**return** CultureInfo.CurrentCulture.CompareInfo.Compare(**this**, strB, CompareOptions.None);

}

它在内部使用了包含区域语言特性信息的Compare比较。CompareInfo.Compare方法接收一个CompareOptions枚举，各个枚举的用意可参考 [***MSDN文档***](http://msdn.microsoft.com/zh-cn/system.globalization.compareoptions) 。

**d) String.Compare**

Compare是CompareTo的静态化版本，在内部都是对CultureInfo.CurrentCulture.CompareInfo.Compare方法进行调用。它同样返回Int32值：-1、0、1。Compare的重载版本会另外的参数，如：

public static int Compare(string strA, string strB, bool ignoreCase)

接收一个bool值ignoreCase表示是否忽略大小写。

public static int Compare(string strA, string strB, StringComparison comparisonType)

接收一个StringComparison枚举，表示要使用的区域文化信息、排序规则等。该枚举的详细可参考 [***MSDN文档***](http://technet.microsoft.com/zh-cn/library/system.stringcomparison(VS.90).aspx) 。

**e) String.CompareOrdinal**

CompareOrdial方法执行的是忽略区域文化信息的序号比较，不执行转换和提供国际化支持所需的系统开销，该方法可用于比较文件路径、IP、URL等字符串。很显然它的性能会比提供区域语言文化信息的Compare方法快很多。

最后，如果进行不区分大小写的比较，或是想对字符串中的大小写进行更改，建议使用 ToUpperInvariant()和ToLowerInvariant()方法，这两个方法使用了固定区域性的大小写规则，而ToUpper()方法和ToLower()方法是依赖区域语言文化信息的，所以性能会差一些。

第六节 StringBuilder

在第2节，我们知道字符串的不可变性，如果要对字符串进行更改，则会重新创建字符串对象，这个过程会导致性能问题，CLR提供了驻留和池来提高性能，同时.NET Framework还提供了一个StringBuilder类可以高效地对字符串进行动态管理，大提高了性能，它不像我们像常说的在内部维护一个Char[]数组那么简单。在后面，我们准备专用一个章节来讨论StringBuilder。

## StringBuilder

String对象是不可改变的。每次使用System.String类中的方法之一时，都要在内存中创建一个新的字符串对象，这就需要为该新对象分配新的空间。在需要对字符串执行重复修改的情况下，与创建新的String对象相关的系统开销可能会非常昂贵。如果要修改字符串而不创建新的对象，则可以使用System.Text.StringBuilder类。例如，当在一个循环中将许多字符串连接在一起时，使用StringBuilder类可以提升性能。

我们先用string类进行字符串拼接，诊断它需要的时间：

|  |
| --- |
| DirectoryInfo dir = new DirectoryInfo(Environment.CurrentDirectory);//获取exe所在的目录  string p = dir.Parent.Parent.FullName + "\\";//获取项目所在目录  string msg = string.Empty;  string[] lines = File.ReadAllLines(p + "sbTest.txt", Encoding.Default);  //Stopwatch 可以用来诊断下面for循环需要的时间。  Stopwatch watch = new Stopwatch();  watch.Start();  for (int i = 0; i < lines.Length; i++)  {  msg = msg + lines[i];  }  watch.Stop();  Console.WriteLine(watch.Elapsed);//在这台机器上需要2秒多。  Console.ReadLine(); |

上面的示例中，字符串的拼接需要2秒多（不同机器需要时间不同），接下来，我们再用StringBuilder来测试需要的时间：

|  |
| --- |
| DirectoryInfo dir = new DirectoryInfo(Environment.CurrentDirectory);//获取exe所在的目录  string p = dir.Parent.Parent.FullName + "\\";//获取项目所在目录  StringBuilder msg = new StringBuilder();  string[] lines = File.ReadAllLines(p + "sbTest.txt", Encoding.Default);  //Stopwatch 可以用来诊断下面for循环需要的时间。  Stopwatch watch = new Stopwatch();  watch.Start();  for (int i = 0; i < lines.Length; i++)  {  msg.Append(lines[i]);  }  watch.Stop();  string s = msg.ToString();//把StringBuilder类型转换为string类型，因为大多数字符串操作是针对string的  Console.WriteLine(watch.Elapsed);//在这台机器上仅需要30多毫秒。  Console.ReadLine(); |

可以发现，StringBuilder效率非常高。这是因为每次修改都时修改的同一个块内存中的值，不会重复创建大量对象，也不会产生垃圾内存，所以大大提高了字符串拼接的效率。

### 设置容量和长度

虽然StringBuilder对象是动态对象，允许扩充它所封装的字符串中字符的数量，但是您可以为它可容纳的最大字符数指定一个值。此值称为该对象的容量，不应将它与当前 StringBuilder对象容纳的字符串长度混淆在一起。例如，可以创建StringBuilder类的带有字符串“Hello”（长度为5）的一个新实例，同时可以指定该对象的最大容量为25。当修改 StringBuilder时，在达到容量之前，它不会为其自己重新分配空间。当达到容量时，将自动分配新的空间且容量翻倍。可以使用重载的构造函数之一来指定StringBuilder类的容量。以下代码示例指定可以将MyStringBuilder对象扩充到最大25个空白。

|  |
| --- |
| StringBuilder MyStringBuilder = new StringBuilder("Hello   World!",   25); |

另外，可以使用读/写Capaci属性来设置对象的最大长度。以下代码示例使用Capacity 属性来定义对象的最大长度

|  |
| --- |
| MyStringBuilder.Capacity   =   25; |

EnsureCapacit方法可用来检查当前StringBuilder的容量。如果容量大于传递的值，则不进行任何更改；但是，如果容量小于传递的值，则会更改当前的容量以使其与传递的值匹配。

也可以查看或设置Length属性。如果Length属性设置为大于Capacity属性的值，则自动将Capacity属性更改为与Length属性相同的值。如果将Length属性设置为小于当前 StringBuilder对象内的字符串长度的值，则会缩短该字符串。

### Append

Append方法可用来将文本或对象的字符串表示形式添加到由当前StringBuilder对象表示的字符串的结尾处。以下示例将一个StringBuilder对象初始化为“Hello World”，然后将一些文本追加到该对象的结尾处。将根据需要自动分配空间。

|  |
| --- |
| StringBuilder   MyStringBuilder   =   new   StringBuilder("Hello   World!");   MyStringBuilder.Append("   What   a   beautiful   day.");   Console.WriteLine(MyStringBuilder); |

此示例将Hello World! What a beautiful day.显示到控制台。

### AppendFormat

AppendFormat方法将文本添加到StringBuilder的结尾处，而且实现了IFormattable接口，因此可接受格式化部分中描述的标准格式字符串。可以使用此方法来自定义变量的格式并将这些值追加到StringBuilder的后面。以下示例使用AppendFormat方法将一个设置为货币值格式的整数值放置到StringBuilder的结尾。

|  |
| --- |
| int   MyInt   =   25;  StringBuilder   MyStringBuilder   =   new   StringBuilder("Your   total   is   ");  MyStringBuilder.AppendFormat("{0:C}   ",   MyInt);  Console.WriteLine(MyStringBuilder); |

此示例将 Your total is $25.00 显示到控制台。

### Insert

Insert方法将字符串或对象添加到当前StringBuilder中的指定位置。以下示例使用此方法将一个单词插入到StringBuilder的第六个位置。

|  |
| --- |
| StringBuilder MyStringBuilder = new StringBuilder("Hello World!");  MyStringBuilder.Insert(6,"Beautiful ");  Console.WriteLine(MyStringBuilder); |

此示例将 Hello Beautiful World! 显示到控制台。

### Remove

可以使用Remove方法从当前StringBuilder中移除指定数量的字符，移除过程从指定的从零开始的索引处开始。以下示例使用Remove方法缩短StringBuilder。

|  |
| --- |
| StringBuilder   MyStringBuilder   =   new   StringBuilder("Hello   World!");   MyStringBuilder.Remove(5,7);  Console.WriteLine(MyStringBuilder); |

此示例将 Hello 显示到控制台。

### Replace

使用Replac方法，可以用另一个指定的字符来替换StringBuilder对象内的字符。以下示例使用Replace方法来搜索StringBuilder对象，查找所有的感叹号字符(!)，并用问号字符(?) 来替换它们。

|  |
| --- |
| StringBuilder MyStringBuilder = new StringBuilder("Hello World!");  MyStringBuilder.Replace('!', '?');  Console.WriteLine(MyStringBuilder); |

此示例将 Hello World? 显示到控制台。

## 常用属性与方法

字符串可以看成字符串数组，具有不可变特性

|  |  |
| --- | --- |
| 属性 | |
| Length | 获得字符串的个数 |
| 方法 | |
| IsNullOrEmpty | 判断字符串是否为空，bool类型 |
| ToUpper | 字符串转换为大写，必须接受返回值（因为不可变特性） |
| ToLower | 字符串转换为小写 |
| Equals | 比较两个字符串是否相同 |
| IndexOf | 返回字子符串第一次出现的位置。如果没有找到对应的数据，返回-1 |
| LastIndexOf | 返回子字符串最后出现的位置。如果没有找到对应的数据，返回-1 |
| Substring | 截取子字符串 |
| Split | 分割字符串 |
| Join | 合并字符串 |
| Format | 格式化字符串，与console.writeline类似，不同之处是有返回值 |

### Length

|  |
| --- |
| string msg = "hao";  Console.WriteLine(msg.Length);//获得字符串长度 |

### IsNullOrEmpty

|  |
| --- |
| string msg = "";  if (string.IsNullOrEmpty(msg))  {  Console.WriteLine(msg.Length);//获得长度  } |

它实际上调用了Length属性

|  |
| --- |
| public static [bool](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://mscorlib:4.0.0.0:b77a5c561934e089/System.Boolean) [**IsNullOrEmpty**](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://mscorlib:4.0.0.0:b77a5c561934e089/System.String/IsNullOrEmpty(String):Boolean)([string](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://mscorlib:4.0.0.0:b77a5c561934e089/System.String) value)  {  if (value != null)  {  return (value.[Length](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://mscorlib:4.0.0.0:b77a5c561934e089/System.String/property:Length:Int32) == 0);  }  return true;  } |

### ToUpper

|  |
| --- |
| string msg = "Hao";  msg = msg.ToUpper();  Console.WriteLine(msg); |

### Equals

|  |
| --- |
| string msg = "hao";  string msg1 = "hao";  if (msg.Equals(msg1))  {  Console.WriteLine("msg=msg1");  } |

它内部是进行以下的运算

|  |
| --- |
| public static [bool](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://mscorlib:4.0.0.0:b77a5c561934e089/System.Boolean) [**Equals**](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://mscorlib:4.0.0.0:b77a5c561934e089/System.String/Equals(String,String):Boolean)([string](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://mscorlib:4.0.0.0:b77a5c561934e089/System.String) a, [string](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://mscorlib:4.0.0.0:b77a5c561934e089/System.String) b)  {  if (a == b)  {  return true;  }  if ((a == null) || (b == null))  {  return false;  } |

### IndexOf

|  |
| --- |
| int index = msg.IndexOf("北京");//返回第一个北京的索引起始值即2  int index1 = msg.IndexOf("北京", index+2); //重载，从index+2即4开始搜索北京，返回值为5，即第二个北京的起始索引  Console.WriteLine(index);  Console.WriteLine(index1); |

### Substring

|  |
| --- |
| string msg = "我爱北京，北京不爱我";  msg = msg.Substring(5);  Console.WriteLine(msg);  msg = "我爱北京，北京不爱我";  msg = msg.Substring(5, 2);  Console.WriteLine(msg); |

### Split

|  |
| --- |
| string msg = "乔峰|慕容复|段誉|虚竹";  string[] msgArray = msg.Split('|');  foreach(string s in msgArray)  {  Console.WriteLine(s);  }  Console.ReadKey(); |
| 去除空的元素 |
| string msg = "乔峰||||慕容复|段誉|虚竹";  string[] msgArray = msg.Split(new char[] { '|' }, StringSplitOptions.RemoveEmptyEntries);  foreach(string s in msgArray)  {  Console.WriteLine(s);  }  Console.ReadKey(); |
| 多个分隔符 |
| string date = "2015年5月8日";  string[] dateArray = date.Split(new char[] { '年', '月', '日' },StringSplitOptions.RemoveEmptyEntries); |

### Join

|  |
| --- |
| string msg = "乔峰||||慕容复|段誉|虚竹";  string[] msgArray = msg.Split(new char[] { '|' }, StringSplitOptions.RemoveEmptyEntries);  msg = string.Join("===", msgArray);  Console.WriteLine(msg);  Console.ReadKey(); |

### Format

|  |
| --- |
| string msg = string.Format("我叫{0},年龄{1}", "高富帅", 18);  Console.WriteLine(msg);  Console.ReadKey();  /\*------------输出结果--------------------------------  \* 我叫高富帅，年龄18.  -------------------------------------------\*/ |

### replace

|  |
| --- |
| string str = "毛泽东对张茜说：“邓小平是个好同志，他对中国革命和世界革命所作的贡献，是已经下了结论的。” ";  str = str.Replace("邓小平", "陈毅"); //修改邓小平为陈毅 |

## 字符数组分割

string.join连接指定数组的元素或集合的成员，在每个元素或成员之间使用指定的分隔符。第一参数指定分割符，第二参数为数组。

|  |
| --- |
| *List*<string> list = new *List*<string>() {"1","2","3","4","5"};  string str = string.*Join*(":", list);  *Console*.*WriteLine*(str); |

输出结果：



# 编码

## 编码的发展

ASCII：英文码表，每一个字符占一个字节。

GB2312：兼容ASCII，包含中文，每一个英文占一个字节，中文占两个字节

GBK:简体中文，兼容GB2312，包含更多汉字。英文占1个字节，中文占2个字节。

Big5：繁体中文。

Unicode：国际码表，中英文都占2个字节

UTF-8：英文占一个字节，中文占3个字节。

## ASCII，Unicode和UTF-8

1. ASCII码

我们知道，在计算机内部，所有的信息最终都表示为一个二进制的字符串。每一个二进制位（bit）有0和1两种状态，因此八个二进制位就可以组合出256种状态，这被称为一个字节（byte）。也就是说，一个字节一共可以用来表示256种不同的状态，每一个状态对应一个符号，就是256个符号，从0000000到11111111。

上个世纪60年代，美国制定了一套字符编码，对英语字符与二进制位之间的关系，做了统一规定。这被称为ASCII码，一直沿用至今。

ASCII码一共规定了128个字符的编码，比如空格"SPACE"是32（二进制00100000），大写的字母A是65（二进制01000001）。这128个符号（包括32个不能打印出来的控制符号），只占用了一个字节的后面7位，最前面的1位统一规定为0。

2、非ASCII编码

英语用128个符号编码就够了，但是用来表示其他语言，128个符号是不够的。比如，在法语中，字母上方有注音符号，它就无法用ASCII码表示。于是，一些欧洲国家就决定，利用字节中闲置的最高位编入新的符号。比如，法语中的é的编码为130（二进制10000010）。这样一来，这些欧洲国家使用的编码体系，可以表示最多256个符号。

但是，这里又出现了新的问题。不同的国家有不同的字母，因此，哪怕它们都使用256个符号的编码方式，代表的字母却不一样。比如，130在法语编码中代表了é，在希伯来语编码中却代表了字母Gimel (ג)，在俄语编码中又会代表另一个符号。但是不管怎样，所有这些编码方式中，0--127表示的符号是一样的，不一样的只是128--255的这一段。

至于亚洲国家的文字，使用的符号就更多了，汉字就多达10万左右。一个字节只能表示256种符号，肯定是不够的，就必须使用多个字节表达一个符号。比如，简体中文常见的编码方式是GB2312，使用两个字节表示一个汉字，所以理论上最多可以表示256x256=65536个符号。

中文编码的问题需要专文讨论，这篇笔记不涉及。这里只指出，虽然都是用多个字节表示一个符号，但是GB类的汉字编码与后文的Unicode和UTF-8是毫无关系的。

3.Unicode

正如上一节所说，世界上存在着多种编码方式，同一个二进制数字可以被解释成不同的符号。因此，要想打开一个文本文件，就必须知道它的编码方式，否则用错误的编码方式解读，就会出现乱码。为什么电子邮件常常出现乱码？就是因为发信人和收信人使用的编码方式不一样。

可以想象，如果有一种编码，将世界上所有的符号都纳入其中。每一个符号都给予一个独一无二的编码，那么乱码问题就会消失。这就是Unicode，就像它的名字都表示的，这是一种所有符号的编码。

Unicode当然是一个很大的集合，现在的规模可以容纳100多万个符号。每个符号的编码都不一样，比如，U+0639表示阿拉伯字母Ain，U+0041表示英语的大写字母A，U+4E25表示汉字"严"。具体的符号对应表，可以查询unicode.org，或者专门的汉字对应表。

4. Unicode的问题

需要注意的是，Unicode只是一个符号集，它只规定了符号的二进制代码，却没有规定这个二进制代码应该如何存储。

比如，汉字"严"的unicode是十六进制数4E25，转换成二进制数足足有15位（100111000100101），也就是说这个符号的表示至少需要2个字节。表示其他更大的符号，可能需要3个字节或者4个字节，甚至更多。

这里就有两个严重的问题，第一个问题是，如何才能区别Unicode和ASCII？计算机怎么知道三个字节表示一个符号，而不是分别表示三个符号呢？第二个问题是，我们已经知道，英文字母只用一个字节表示就够了，如果Unicode统一规定，每个符号用三个或四个字节表示，那么每个英文字母前都必然有二到三个字节是0，这对于存储来说是极大的浪费，文本文件的大小会因此大出二三倍，这是无法接受的。

它们造成的结果是：1）出现了Unicode的多种存储方式，也就是说有许多种不同的二进制格式，可以用来表示Unicode。2）Unicode在很长一段时间内无法推广，直到互联网的出现。

5.UTF-8

互联网的普及，强烈要求出现一种统一的编码方式。UTF-8就是在互联网上使用最广的一种Unicode的实现方式。其他实现方式还包括UTF-16（字符用两个字节或四个字节表示）和UTF-32（字符用四个字节表示），不过在互联网上基本不用。重复一遍，这里的关系是，UTF-8是Unicode的实现方式之一。

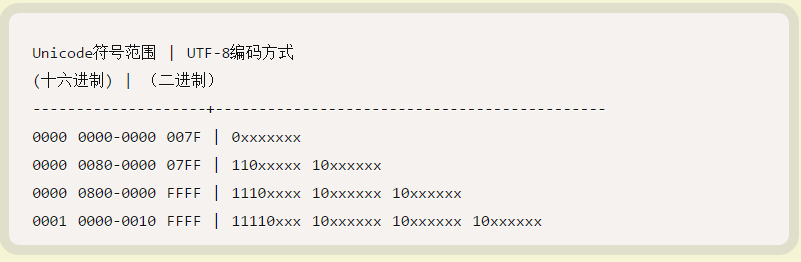
UTF-8最大的一个特点，就是它是一种变长的编码方式。它可以使用1~4个字节表示一个符号，根据不同的符号而变化字节长度。

UTF-8的编码规则很简单，只有二条：

1）对于单字节的符号，字节的第一位设为0，后面7位为这个符号的unicode码。因此对于英语字母，UTF-8编码和ASCII码是相同的。

2）对于n字节的符号（n>1），第一个字节的前n位都设为1，第n+1位设为0，后面字节的前两位一律设为10。剩下的没有提及的二进制位，全部为这个符号的unicode码。

下表总结了编码规则，字母x表示可用编码的位。



Unicode符号范围 | UTF-8编码方式

(十六进制) | （二进制）

--------------------+---------------------------------------------

0000 0000-0000 007F | 0xxxxxxx

0000 0080-0000 07FF | 110xxxxx 10xxxxxx

0000 0800-0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx

0001 0000-0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

跟据上表，解读UTF-8编码非常简单。如果一个字节的第一位是0，则这个字节单独就是一个字符；如果第一位是1，则连续有多少个1，就表示当前字符占用多少个字节。

下面，还是以汉字"严"为例，演示如何实现UTF-8编码。

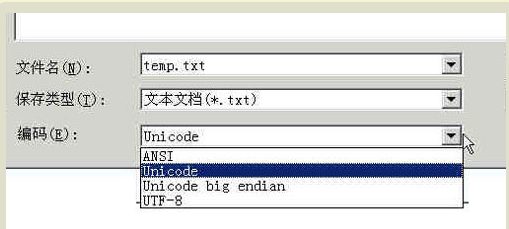
已知"严"的unicode是4E25（100111000100101），根据上表，可以发现4E25处在第三行的范围内（0000 0800-0000 FFFF），因此"严"的UTF-8编码需要三个字节，即格式是"1110xxxx 10xxxxxx 10xxxxxx"。然后，从"严"的最后一个二进制位开始，依次从后向前填入格式中的x，多出的位补0。这样就得到了，"严"的UTF-8编码是"11100100 10111000 10100101"，转换成十六进制就是E4B8A5。

6. Unicode与UTF-8之间的转换

通过上一节的例子，可以看到"严"的Unicode码是4E25，UTF-8编码是E4B8A5，两者是不一样的。它们之间的转换可以通过程序实现。

在Windows平台下，有一个最简单的转化方法，就是使用内置的记事本小程序Notepad.exe。打开文件后，点击"文件"菜单中的"另存为"命令，会跳出一个对话框，在最底部有一个"编码"的下拉条。

bg2007102801.jpg



里面有四个选项：ANSI，Unicode，Unicode big endian 和 UTF-8。

1）ANSI是默认的编码方式。对于英文文件是ASCII编码，对于简体中文文件是GB2312编码（只针对Windows简体中文版，如果是繁体中文版会采用Big5码）。

2）Unicode编码指的是UCS-2编码方式，即直接用两个字节存入字符的Unicode码。这个选项用的little endian格式。

3）Unicode big endian编码与上一个选项相对应。我在下一节会解释little endian和big endian的涵义。

4）UTF-8编码，也就是上一节谈到的编码方法。

选择完"编码方式"后，点击"保存"按钮，文件的编码方式就立刻转换好了。

7. Little endian和Big endian

上一节已经提到，Unicode码可以采用UCS-2格式直接存储。以汉字"严"为例，Unicode码是4E25，需要用两个字节存储，一个字节是4E，另一个字节是25。存储的时候，4E在前，25在后，就是Big endian方式；25在前，4E在后，就是Little endian方式。

这两个古怪的名称来自英国作家斯威夫特的《格列佛游记》。在该书中，小人国里爆发了内战，战争起因是人们争论，吃鸡蛋时究竟是从大头(Big-Endian)敲开还是从小头(Little-Endian)敲开。为了这件事情，前后爆发了六次战争，一个皇帝送了命，另一个皇帝丢了王位。

因此，第一个字节在前，就是"大头方式"（Big endian），第二个字节在前就是"小头方式"（Little endian）。

那么很自然的，就会出现一个问题：计算机怎么知道某一个文件到底采用哪一种方式编码？

Unicode规范中定义，每一个文件的最前面分别加入一个表示编码顺序的字符，这个字符的名字叫做"零宽度非换行空格"（ZERO WIDTH NO-BREAK SPACE），用FEFF表示。这正好是两个字节，而且FF比FE大1。

如果一个文本文件的头两个字节是FE FF，就表示该文件采用大头方式；如果头两个字节是FF FE，就表示该文件采用小头方式。

8. 实例

下面，举一个实例。

打开"记事本"程序Notepad.exe，新建一个文本文件，内容就是一个"严"字，依次采用ANSI，Unicode，Unicode big endian 和 UTF-8编码方式保存。

然后，用文本编辑软件UltraEdit中的"十六进制功能"，观察该文件的内部编码方式。

1）ANSI：文件的编码就是两个字节"D1 CF"，这正是"严"的GB2312编码，这也暗示GB2312是采用大头方式存储的。

2）Unicode：编码是四个字节"FF FE 25 4E"，其中"FF FE"表明是小头方式存储，真正的编码是4E25。

3）Unicode big endian：编码是四个字节"FE FF 4E 25"，其中"FE FF"表明是大头方式存储。

4）UTF-8：编码是六个字节"EF BB BF E4 B8 A5"，前三个字节"EF BB BF"表示这是UTF-8编码，后三个"E4B8A5"就是"严"的具体编码，它的存储顺序与编码顺序是一致的。

## .输出所有的的编码

Encoding.GetEncodings()，可以输出.net支持的所有的编码

|  |
| --- |
| StringBuilder sb = new StringBuilder();  EncodingInfo[] infos = Encoding.GetEncodings();//输出所有编码  sb.AppendLine("编码页"+"\t"+"编码"+"\t"+"显示名称");  foreach (var item in infos)  {  sb.AppendLine(item.CodePage + "\t" + item.Name + "\t" + item.DisplayName);  }  File.WriteAllText("encoding.txt", sb.ToString()); |

## 以指定的编码读取

|  |
| --- |
| string msg = File.ReadAllText("encoding.txt", Encoding.Default);//当前系统默认编码中文为：GB2312。  string msg1 = File.ReadAllText("encoding.txt");//默认的编码为UTF-8；  //还可以输入特定的编码  string msg2 = File.ReadAllText("encoding.txt", Encoding.GetEncoding("GB18030"));  //常用的ASCII，Unicode可以直接被封装为属性。  string msg3 = File.ReadAllText("encoding.txt", Encoding.ASCII); |

## 通过文件头判断编码

UTF-8的前三个字节为文件头，指明了它的编码格式，unicode（utf-16）的前两个字节为文件头。而ASCII的编码文件没有头文件。

用ReadAllBytes进行读取时，因为文件头存在，则可能出现乱码。示例代码如下：

|  |
| --- |
| string txt = "1.txt";//内容为：Hello,world  byte[] bytes = File.ReadAllBytes(txt);  string msg = System.Text.Encoding.UTF8.GetString(bytes);  Console.WriteLine(msg);//输出为：？Hello,world  Console.ReadLine(); |

上面的代码中读取的“1.txt”编码格式为utf-8，当读取为字节，再编码为UTF8时则出现了乱码” ？Hello,world “。

1. Print与Debug.log的区别：Print方法属于UnityEngine.MonoBehavior类，而Debug.log属于UnityEngine命名空间。Print方法只有继承了MonoBehavior的类才能使用，而Debug.log方法几乎可以在所有引入了UnityEngine命名空间的Unity脚本的类中使用， [↑](#footnote-ref-1)
2. 关于命名空间的更多介绍，请参考[命名空间和程序集](#_命名空间和程序集) [↑](#footnote-ref-2)
3. 程杰： 《大话数据结构》 ，第11页。 [↑](#footnote-ref-3)
4. http://jingyan.baidu.com/article/642c9d34d7ec8f644a46f7fb.html [↑](#footnote-ref-4)