Francisco Hao

[公司名称][公司地址]

C#面向对象篇

目录

[第 1 章 概述 4](#_Toc421389707)

[1.1 面向对象的三大特性 4](#_Toc421389708)

[1.2 面向对象编程的五大原则 5](#_Toc421389709)

[1.2.1 单一职责原则（Single Responsibility Principle） 5](#_Toc421389710)

[1.2.2 开闭原则（the Open Closed Principle OCP） 5](#_Toc421389711)

[1.2.3 里氏替换原则 (the Liskov Substitution Principle LSP) 5](#_Toc421389712)

[1.2.4 依赖倒置原则 (the Dependency Inversion Principle DIP) 6](#_Toc421389713)

[1.2.5 接口隔离原则（（the Interface Segregation Principle ISP）） 6](#_Toc421389714)

[第 2 章 类和对象 6](#_Toc421389715)

[2.1 类是一种数据结构 6](#_Toc421389716)

[2.2 类（class）、对象（Object）和实例（instance） 7](#_Toc421389717)

[2.3 类的声明和实例化 7](#_Toc421389718)

[2.3.1 类的声明 7](#_Toc421389719)

[2.3.2 类的实例化 7](#_Toc421389720)

[2.4 字段和方法 9](#_Toc421389721)

[2.4.1 字段 9](#_Toc421389722)

[2.4.2 方法 9](#_Toc421389723)

[2.5 访问类的成员 10](#_Toc421389724)

[2.6 实例成员和静态成员 10](#_Toc421389725)

[2.6.1 实例成员 10](#_Toc421389726)

[2.6.2 静态成员 11](#_Toc421389727)

[2.6.3 静态成员特点 12](#_Toc421389728)

[2.7 构造函数 12](#_Toc421389729)

[2.7.1 实例构造函数 12](#_Toc421389730)

[2.7.2 静态构造函数 14](#_Toc421389731)

[2.7.3 构造函数的可访问性 15](#_Toc421389732)

[2.7.4 对象初始化列表 15](#_Toc421389733)

[2.8 属性 16](#_Toc421389734)

[2.8.1 属性的访问器 17](#_Toc421389735)

[2.8.2 一个属性的示例 18](#_Toc421389736)

[2.8.3 使用属性 19](#_Toc421389737)

[2.8.4 关联字段与执行运算 19](#_Toc421389738)

[2.8.5 只读和只写属性 20](#_Toc421389739)

[2.8.6 自动实现属性 20](#_Toc421389740)

[2.8.7 静态属性 21](#_Toc421389741)

[2.8.8 访问器的访问修饰符 22](#_Toc421389742)

[2.8.9 属性的作用 23](#_Toc421389743)

[2.8.10 常见错误 24](#_Toc421389744)

[2.9 索引器 25](#_Toc421389745)

[2.9.1 声明索引 25](#_Toc421389746)

[2.9.2 使用索引 26](#_Toc421389747)

[2.10 其他类成员 27](#_Toc421389748)

[2.11 综合示例：类的基本构成 27](#_Toc421389749)

[第 3 章 索引器 28](#_Toc421389750)

[3.1 声明和使用索引 28](#_Toc421389751)

[3.2 使用索引 30](#_Toc421389752)

[3.3 示例演示 30](#_Toc421389753)

[第 4 章 封装 31](#_Toc421389754)

[4.1 什么是封装 31](#_Toc421389755)

[4.2 为什么需要封装 32](#_Toc421389756)

[4.2.1 封装把程序分成一个个相对独立的单元 32](#_Toc421389757)

[4.2.2 封装使修改代码更加容易和安全 32](#_Toc421389758)

[4.2.3 封装使整个程序开发的复杂度大大降低 33](#_Toc421389759)

[4.3 访问修饰符 33](#_Toc421389760)

[4.3.1 类访问修饰符 33](#_Toc421389761)

[4.3.2 类成员访问修饰符 34](#_Toc421389762)

[4.4 访问修饰符与封装 36](#_Toc421389763)

[4.5 访问级别约束 37](#_Toc421389764)

[4.5.1 子类的访问级别不能比父类的高 37](#_Toc421389765)

[4.5.2 类中属性或字段的访问级别不能比其参数的访问级别高 38](#_Toc421389766)

[4.5.3 方法的访问级别不能比返回值的访问级别高 38](#_Toc421389767)

[4.6 命名空间和程序集 38](#_Toc421389768)

[4.6.1 命名空间 39](#_Toc421389769)

[4.6.2 using指令 41](#_Toc421389770)

[4.6.3 创建并使用程序集 42](#_Toc421389771)

[第 5 章 复合与继承 44](#_Toc421389772)

[5.1 复合 45](#_Toc421389773)

[5.2 初始化复合对象 48](#_Toc421389774)

[5.3 继承 49](#_Toc421389775)

[5.3.1 如何判断继承是否合理 50](#_Toc421389776)

[5.3.2 继承的好处 50](#_Toc421389777)

[5.3.3 继承的特性 50](#_Toc421389778)

[5.4 访问继承的成员 53](#_Toc421389779)

[5.5 继承与访问控制 54](#_Toc421389780)

[5.6 隐藏基类成员 55](#_Toc421389781)

[5.7 何时用复合，何时用继承 56](#_Toc421389782)

[5.8 生物分类学与类 57](#_Toc421389783)

[第 6 章 多态 60](#_Toc421389784)

[6.1 里氏替换原则 60](#_Toc421389785)

[6.2 多态的基础：虚（virtual）方法和覆写（override）方法 60](#_Toc421389786)

[6.3 基类引用 63](#_Toc421389787)

[6.4 虚方法特点总结 65](#_Toc421389788)

[6.5 抽象类和抽象方法 65](#_Toc421389789)

[6.5.1 抽象类特点 67](#_Toc421389790)

[6.6 接口 69](#_Toc421389791)

[6.6.1 什么是接口 69](#_Toc421389792)

[6.6.2 什么时候用接口 71](#_Toc421389793)

[6.6.3 声明接口 73](#_Toc421389794)

[6.6.4 实现接口 73](#_Toc421389795)

[6.6.5 接口引用和as运算符 75](#_Toc421389796)

[6.6.6 多个接口的实现与引用 76](#_Toc421389797)

[6.6.7 接口成员的实现 77](#_Toc421389798)

[6.6.8 接口可以继承接口 80](#_Toc421389799)

[6.7 谈C#多态的魅力（虚方法，抽象，接口实现） 81](#_Toc421389800)

[6.8 多态的意义与目的 95](#_Toc421389801)

[6.9 面向对象的编程的使用建议 95](#_Toc421389802)

[第 7 章 类型操作 96](#_Toc421389803)

[7.1 GetType方法获取类的类型 96](#_Toc421389804)

[7.2 判断向下转换是否成功 96](#_Toc421389805)

[7.3 判断是否是同一个对象 96](#_Toc421389806)

# 概述

在面向对象编程中，我们之前学过的所有结构化编程构造仍然适用。然而，将那些构造封装在类中，可以创建更大，更有条理以及更容易维护的程序。从结构化的、基于流程控制的程序转向面向对象的程序，是思维模式的一个根本性变化。

与面向过程相比，面向对象的一大优势是，不需要完全从头创建新的程序。相反，可以将现有的一系列对象组装到一起，用新的功能来扩展类，添加更多的类，然后将一切重新组装起来，以提供新的功能。

那么什么是面向对象编程呢？简单来说面向对象编程是“以对象为中心的编程”，其英文是“Object-Oriented Programming”，也就是从物体的角度来编程。通过面向对象，现实世界事物的特性和功能被组织为逻辑上相关的数据项和函数的封装集合，称之为类。

比如，我们创建一个宠物狗养成游戏。这个狗的皮毛，身材都会随着时间而变化；此外，它还可以打滚、吐舌头，吠叫等。在C#的面向对象思想中，皮毛、身材特性被封装为数据成员，而打滚等动作被封装为函数成员。下面的代码实现了狗的模拟。

|  |
| --- |
| public class Dog{  public int dogSize = 1;//数据成员： 存储狗的大小信息  public string dogSkin = "red";//数据成员： 存储毛发颜色  public void roll()//函数成员：实现打滚动作  {  ;  }  public void stick()//函数成员:实现吐舌头动作  {  ;  }  public void bark()//函数成员：实现吠叫动作  {  ;  }  } |

## 面向对象的三大特性

面向对象的三大特性：封装、继承、多态

所谓封装，也就是把客观事物封装成抽象的类，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。封装是面向对象的特征之一，是对象和类概念的主要特性。 简单的说，一个类就是一个封装了数据以及操作这些数据的代码的逻辑实体。在一个对象内部，某些代码或某些数据可以是私有的，不能被外界访问。通过这种方式，对象对内部数据提供了不同级别的保护，以防止程序中无关的部分意外的改变或错误的使用了对象的私有部分。

所谓继承是指可以让某个类型的对象获得另一个类型的对象的属性的方法。它支持按级分类的概念。继承是指这样一种能力：它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展。 通过继承创建的新类称为“子类”或“派生类”，被继承的类称为“基类”、“父类”或“超类”。继承的过程，就是从一般到特殊的过程。要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。继承概念的实现方式有二类：实现继承与接口继承。实现继承是指直接使用基类的属性和方法而无需额外编码的能力；接口继承是指仅使用属性和方法的名称、但是子类必须提供实现的能力；

所谓多态就是指一个类实例的相同方法在不同情形有不同表现形式。多态机制使具有不同内部结构的对象可以共享相同的外部接口。这意味着，虽然针对不同对象的具体操作不同，但通过一个公共的类，它们（那些操作）可以通过相同的方式予以调用。

## 面向对象编程的五大原则

### 单一职责原则（Single Responsibility Principle）

单一职责原则，其核心思想为：一个类，最好只做一件事，只有一个引起它的变化原因。单一职责原则可以看做是低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。职责过多，可能引起它变化的原因就越多，这将导致职责依赖，相互之间就产生影响，从而大大损伤其内聚性和耦合度。通常意义下的单一职责，就是指只有一种单一功能，不要为类实现过多的功能点，以保证实体只有一个引起它变化的原因。

### 开闭原则（the Open Closed Principle OCP）

一个模块在扩展性方面应该是开放的而在更改性方面应该是封闭的。因此在进行面向对象设计时要尽量考虑接口封装机制、抽象机制和多态技术。该原则同样适合于非面向对象设计的方法，是软件工程设计方法的重要原则之一。我们以收音机的例子为例，讲述面向对象的开闭原则。我们收听节目时需要打开收音机电源，对准 电台频率和进行音量调节。但是对于不同的收音机，实现这三个步骤的细节往往有所不同。比如自动收缩电台的收音机和按钮式收缩在操作细节上并不相同。因此， 我们不太可能针对每种不同类型的收音机通过一个收音机类来实现（通过重载）这些不同的操作方式。但是我们可以定义一个收音机接口，提供开机、关机、增加频 率、降低频率、增加音量、降低音量六个抽象方法。不同的收音机继承并实现这六个抽象方法。这样新增收音机类型不会影响其它原有的收音机类型，收音机类型扩 展极为方便。此外，已存在的收音机类型在修改其操作方法时也不会影响到其它类型的收音机。

### 里氏替换原则 (the Liskov Substitution Principle LSP)

子类应当可以替换父类并出现在父类能够出现的任何地方。这个原则是Liskov于1987年提出的设计原则。它同样可以从Bertrand Meyer 的DBC (Design by Contract) 的概念推出。

　　我们以学生为例，夜校生为学生的子类，因此在任何学生可以出现的地方，夜校生均可出现。这个例子有些牵强，一个能够反映这个原则的例子时圆和椭圆，圆是椭圆的一个特殊子类。因此任何出现椭圆的地方，圆均可以出现。但反过来就可能行不通。

　　运用替换原则时，我们尽量把类B设计为抽象类或者接口，让C类继承类B（接口B）并实现操作A和操作B，运行时，类C实例替换B，这样我们即可进行新类的扩展（继承类B或接口B），同时无须对类A进行修改。

### 依赖倒置原则 (the Dependency Inversion Principle DIP)

在进行业务设计时，与特定业务有关的依赖关系应该尽量依赖接口和抽象类，而不是依赖于具体类。具体类只负责相关业务的实现，修改具体类不影响与特定业务有关的依赖关系。

在结构化设计中，我们可以看到底层的模块是对高层抽象模块的实现（高层抽象模块通过调用底层模块），这说明，抽象的模块要依赖具体实现相关的模块，底层模块的具体实现发生变动时将会严重影响高层抽象的模块，显然这是结构化方法的一个"硬伤"。

面向对象方法的依赖关系刚好相反，具体实现类依赖于抽象类和接口。

为此，我们在进行业务设计时，应尽量在接口或抽象类中定义业务方法的原型，并通过具体的实现类（子类）来实现该业务方法，业务方法内容的修改将不会影响到运行时业务方法的调用。

### 接口隔离原则（（the Interface Segregation Principle ISP））

采用多个与特定客户类有关的接口比采用一个通用的涵盖多个业务方法的接口要好。

ISP原则是另外一个支持诸如COM等组件化的使能技术。缺少ISP，组件、类的可用性和移植性将大打折扣。

这个原则的本质相当简单。如果你拥有一个针对多个客户的类，为每一个客户创建特定业务接口，然后使该客户类继承多个特定业务接口将比直接加载客户所需所有方法有效。

# 类和对象

类是抽象的，对象是具体的，对象可以叫做类的实例。类不占内存，对象才占内存。

“人”是类，而“张三”就是“人”这个类的对象，“汽车”是类，你买的那个车（实例）就叫做“车”的对象。

## 类是一种数据结构

类是封装的最小单元，它是逻辑相关的数据和函数的封装，它是一种能存储数据并执行代码的数据结构。它包含以下内容：

* 数据成员：存储与类或类的实例相关的数据。数据成员通常模拟该类所表现的现实世界事物的特性。
* 函数成员：它执行代码。函数成员通常模拟类所表示的现实世界事物的功能和操作。

## 类（class）、对象（Object）和实例（instance）

在非正式的场合，类和对象这两个词经常互换着使用。然而，对象和类具有截然不同的含义。类是一个模板，它定义了一个对象在实例化的时候看起来是什么样子。而对象是类的一个实例。类就像一个模具，它定义了一个零件的样子。对象就是用这个模具创建的零件。我们将从类创建对象的过程称为实例化（instantiation）。

## 类的声明和实例化

### 类的声明

类的声明用来定义新类的成员，它并不创建类的实例，而是创建了用于创建实例的模板。

类的声明需要以关键字class开始，下面的代码是类声明的语法：

|  |
| --- |
| class 类名  {  成员声明;  } |

例如，下面的代码展示了一个myClass类的声明，它包含了一个名为i的数据成员声明和一个名为myMethod的函数成员声明。

|  |
| --- |
| class myClass  {  public int i = 3;//数据成员声明  public void myMethod()//函数成员声明  {  ;  }  } |

### 类的实例化

一旦类被声明，就可以创建类的实例。C#使用new关键字来实例化一个对象。

类是引用类型，这意味着要实例化时为数据引用和实际数据两者都申请内存。所以，类的实例化要分为两个步骤：

* 创建类类型的变量，用来保存实例化对象的引用。

|  |
| --- |
| myClass myC;  **变量名称**  **类名称** |

* 使用new关键字为指定类型的实例分配并初始化内存，之后把引用赋值给类类型的变量。

|  |
| --- |
| myC = new myClass();  **圆括号是必需的**  **类型**  **关键字** |

还可以在实例化时把两个步骤结合在一起：

|  |
| --- |
| myClass myC = new myClass(); |

下边的代码展示了一个名为myClass的类的声明，之后它在aboutClass中被实例化了，创建了该类的对象。

|  |
| --- |
| using UnityEngine;  using System.Collections;  /// <summary>  /// 声明了一个myClass类，它由一个数据成员和一个函数成员  /// </summary>  public class myClass  {  public int myInt = 2;  public void myMethod()  {  Debug.Log("this is myclass");  }  }  public class aboutClass : MonoBehaviour {  // Use this for initialization  void Start ()  {  myClass useClass;//声明类类型变量  useClass = new myClass();//为myClass类分配内存，创建myClass的实例，最后赋值给类变量  }    } |

## 字段和方法

字段和方法是最重要的类成员。字段是数据成员，方法是函数成员。

### 字段

字段是隶属于类的变量，必须在类声明内部声明。作为数据成员，它可以被写入或读取。声明一个字段的语法如下：

|  |
| --- |
| class myField()  {  类型 字段名称;  } |

例如下面的类myField中声明了两个字段：isField和isField2，并将字段isField2初始化，没有初始化语句的字段isField的默认值为0。

|  |
| --- |
| class myField()  {  int isField;//隐式初始化字段的值为int类型的默认值：0  int isField2 = 1;//显式初始化字段的值为1  } |

### 方法

从第7章的学习中，我们知道方法是具有名称的可执行代码，与C和C++中的函数类似，它可以在程序的很多不同地方执行，甚至从其他程序中执行；当方法被调用时，它执行自己所含的代码，然后返回到调用它的代码。与c和c++不同的是，C#中没有全局函数，方法的声明必须声明在类声明的内部。

下面的代码展示了方法应当声明的位置：

|  |
| --- |
| using UnityEngine;  using System.Collections;  /// <summary>  /// 错误的声明位置，c#中没有全局方法。需要在类内部声明  /// </summary>  void myMethod1()  {  ;  }  public class NewBehaviourScript : MonoBehaviour {  /// <summary>  /// 正确的声明位置，在NewBehaviourScript类内部声明  /// </summary>  void myMethod2()  {  ;  }  } |

方法的详细介绍请参考第七章：[方法](#_方法)。

## 扩展方法

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  Person p = new Person();  p.Name = "hao";  //调用扩展方法  p.SayHello();  Console.Read();  }  }  public class Person  {  string \_name;  public string Name  {  get { return \_name; }  set { \_name = value; }  }  }  //1.增加扩展方法第一步：增加一个静态类。  //该静态类与将来要用扩展方法的地方在同一个命名空间下。即便命名空间不一样，用的时候也必须导入该命名空间否则不能使用。  //  public static class ExtMethod  {  //2.向静态类中增加一个静态方法。  //该静态方法的第一个参数就表示要给那个类型增加该方法  //第一个修饰符this是必须的，这里的obj就表示将来调用该方法的那个对象。  public static void SayHello(this Person obj)  {  Console.WriteLine("Hello\t" + obj.Name);  }  } |

案例，添加扩展方法判断是否邮箱

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  string msg = "hyfdbd@163.com";  bool b = msg.IsEmail();  Console.WriteLine(b);  Console.Read();  }  }  public static class Email  {  public static bool IsEmail(this string str)  {  return Regex.IsMatch(str, @"[-0-9a-zA-Z.\_]+@[0-9a-zA-Z.\_]+(\.[0-9a-zA-Z]+)+");  }  } |

## 访问类的成员

在类的内部，只需要使用类成员的名称就可以访问它们。

例如，下面的类声明中声明了两个字段numberA、numberB。在声明的方法sumNumber中使用通过字段名称使用了它们。

|  |
| --- |
| class myClass  {  int numberA = 3;  int numberB = 5;  void sumNumber()  {  int sum = numberA + numberB;//类内部：通过成员名称访问类的成员。  }  } |

从类的外部访问实例成员时，需要使用变量名称和成员名称，中间用点运算符（.）分隔。

例如，下面代码的展示了一个从类的外部访问方法的示例：

|  |
| --- |
| static void Start () {  myClass mc = new myClass();//创建类的实例  mc.sumNumber();// 访问myClass类的函数成员sumNUmber  } |

从类的外部访问静态成员时，需要使用类名称和成员名称，中间也用点运算符分开。

例如，下面的代码展示了访问静态成员的方法。关于静态成员请参考：[静态成员](#_静态成员)。

|  |
| --- |
| myClass.myField = 5;//访问静态成员需要使用类名,而不是变量名称。 |

## 实例成员和静态成员

### 实例成员

类的每一个实例都是不同的实体（有不同的内存分配），它们有自己的一组成员，这些成员关联到该实例，改变该实例的成员的值不会影响到其他的实例。这些关联到实例的成员被称为实例成员。

例如，下面的代码声明一个类myClass，它带有唯一的字段型成员myField。后在Start方法中为类myClass创建了两个实例，每一个实例都有自己的字段型成员myField的拷贝，改变一个实例的字段型成员的拷贝的值不影响其他实例的拷贝的值。

|  |
| --- |
| using UnityEngine;  using System.Collections;  class myClass  {  public int myField = 4;  }  public class NewBehaviourScript : MonoBehaviour {  // Use this for initialization  void Start () {  myClass class01 = new myClass();//创建一个类的实例  myClass class02 = new myClass();//创建第二个类的实例  class01.myField = 5;//改变myClass类的实例class01的myfield字段的值为5；  print(class02.myField);//输出实例class02的字段myField的值。它的值为初始化值4。    }  } |

### 静态成员

默认情况下，成员被关联到一个实例，这样的成员叫实例成员。成员也可以关联到类本身，而不是关联到实例，这样的成员叫静态成员。

静态成员不依赖实例而独立存在。简单地说，即使不创建一个类的实例，我们也可以访问到该类的静态成员。（类似于全局变量，在程序运行时创建，程序结束时销毁）

以下面的代码为例，声明了一个类myClass，它由一个静态成员myField。在Start方法中使用静态成员myFields，不需要实例化类型。

|  |
| --- |
| using UnityEngine;  using System.Collections;  class myClass  {  public static int myField = 3;  }  public class NewBehaviourScript : MonoBehaviour {  static void tart () {  myClass.myField = 5;//访问静态成员需要使用类名，不需要实例化  }  } |

静态成员在平常的OOP编程中与高级的OOP设计中都有重要的地位。

在日常的OOP编程中，有时我们只是想用类来存储一些数据，并不想创建任何类的实例。一方面，为了减少麻烦，不想用new创建实例；另一方面，希望数据集中管理，不想因为不小心创建了多个实例而导致数据存储的分散。那么这时使用静态属性就很方便，只需要在类名后面加上静态成员的名称就可以访问到所需要的数据。

有时我们只希望某个类提供一些服务，不需要创建具体实例时，也可以用静态成员。例如C#中的Math类就包含了多个静态方法用来提供工具性的方法。

除了以上两点外，在高级OOP编程的设计模式（Design Pattern）中静态成员也经常用到。比如应用广泛的工厂模式，著名的Singleton模式、享元模式等。

### 静态成员特点

* 静态成员在整个应用程序中共享同一个值
* 只有当程序退出后，才会释放静态资源的内存
* 静态成员只能通过类名来直接访问，不能通过对象名访问
* 静态类不能实例化对象

所以当多个实例共享一个成员时，可以使用静态成员。当使用一个工具类的时候，可以使用静态类把相关的方法封装到一个类中，方便调用。

## 构造函数

对象的创建和删除是所有OOP语言关注的重要课题。在C#中，创建对象时会自动调用构造函数；删除则是由垃圾回收机按照自己计划执行。

什么是构造函数（构造方法）？我们在使用一个对象前，往往需要初始化这个新生的对象状态。这个初始化动作怎么执行？放在哪里？难道要用户在创建时每一个对象都要手动调用吗？为了解决这些问题，现在成熟的OOP语言中都采用了构造函数。一个类只要含有构造函数，那么编译器会告知这个对象在创建时需要调用这个函数，完成我们指定的初始化动作。简单地说，当我们使用new关键字加类名时，构造函数就会执行。

构造函数主要作用就是对对象进行初始化。

### 实例构造函数

实例构造函数是一个特殊的方法，它在类的每个实例创建的时候执行。构造函数的声明方法看起来很像方法的声明方法，但是与普通方法不同的是构造方法的名称必须与类的名称一致，而且没有返回类型：

|  |
| --- |
| class Class01  **声明为public，可以从类的外部访问**  {  public Class01()  {    **没有返回类型**  **与类名称一致**  }  } |

构造函数可以有参数，通过给构造函数传入参数来初始化成员是常见的方法。如果类中没有构造函数，那么编译器会自动生成一个默认的空的构造函数。例如，在下面的代码中，创建了三个类：Class01，Class02，Class03并为它们创建了3种形式的构造函数，最后在Start方法中调用了各自的构造函数。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  class Class01  {  public DateTime time;  public Class01()//构造函数：用来初始化当前时间  {  time = DateTime.Now;  }  }  class Class02  {  public string name;  public Class02(string str)//带参数的构造函数  {  name = str;  }  }  class Class03//空的构造函数，默认会创建一个 空的构造函数：public Class03(){};  {  public int i;  }  public class NewBehaviourScript : MonoBehaviour {  // Use this for initialization  void Start () {  Class01 mc01 = new Class01();//调用构造函数来实例化对象。  Class02 mc02 = new Class02("frank");//实例化对象过程中赋值  Class03 mc03 = new Class03();//使用默认构造函数来实例化。  print(mc01.time);  print(mc02.name);  print(mc03.i);  }  /\*输出结果  \* 12/29/2013 11:13:49 AM  \* frank  \* 0  \*/  } |

构造函数支持重载，但如果定义了一个或多个构造函数，那么编译器将不会为类定义默认的构造函数。例如。在下面的代码中，Class01有两个构造函数：一个带int参数另一个带string参数。Main使用不带参数的默认构造函数时会报错，因为编译器不会为该类定义默认构造函数。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  class Class01  {  int id;  string name;  public Class01(int i)//构造函数1：带int参数  {  id = i;  name = "Frank";  }  public Class01(string str)//构造函数2：带str参数  {  name = str;  }  public void printClass()  {  Debug.Log(id);  Debug.Log(name);  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  Class01 mc01 = new Class01(1);//调用构造函数1  Class01 mc02 = new Class01("Frank");//调用构造函数2  Class01 mc03 = new Class01();//会报错，不能使用默认构造函数  mc01.printClass();  mc02.printClass();  }  } |

### 静态构造函数

构造函数也可以声明为static。实例构造函数初始化类的每个新实例，而静态构造函数通常用来初始化类的静态字段。

与实例构造函数相似的是，静态构造函数的名称必须和类名相同，构造函数不能有返回值；与实例构造函数不同的是，静态构造函数声明中使用static关键字，类只能有一个静态构造函数，而且不能带参数，静态构造函数不能有访问修饰符。静态构造函数的语法如下：

|  |
| --- |
| class Class01  {  static Class01()//静态构造函数  {  }  } |

静态构造函数不能访问所在类的实例成员，也不能使用this访问器。此外，静态构造函数也不能从程序中显示的调用，它们会被系统自动调用。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  class Class01  {  static int number;  static Class01()//静态构造函数:初始化静态字段  {  number = 100;  }  public void printNumber()  {  Debug.Log(number);  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  Class01 mc01 = new Class01();//  mc01.printNumber();  }  } |

### 构造函数的可访问性

与Javascript中只能使用public访问修饰符不同，在C#中可以为实例构造函数指派访问修饰符，就想对其他成员所做的那样。

### 对象初始化列表

对象初始化列表允许在创建新的对象实例时设置字段或属性的值。

在此之前的内容我们已经看到，对象创建表达式由关键字new后面跟着一个类构造函数组成。对象初始化列表扩展了创建语法，在表达式的尾部放置了一组成员初始化列表。该语法有两种形式：一种形式包含构造函数的参数列表，另一种不包括。如下：

|  |
| --- |
| **对象初始化列表**  new ClassName(Arglist) {FieldOrProp = initExpr;FieldOrProp = initExpr}  new ClassName {FieldOrProp = initExpr;FieldOrProp = initExpr}  **字段或属性初始化表达式**  **字段或属性初始化表达式** |

下面的代码展示了一个使用对象初始化列表的示例。在Start中，默认构造函数把字段的值设为1和2，然后对象初始化列表语句把它们改为3和4。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  class Class01  {  public int x = 1;  public int y = 2;  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  Class01 mc = new Class01 {x = 3, y = 4 };//对象初始化列表  }  } |

## 构造函数重写与调用

|  |
| --- |
| /\*----------------------------------------------------------  \* 定义父类Father（三个属性：LastName，property，bloodtype），  \* 儿子类Son（PLaygame方法），女儿类Daughter（Dance方法）。调用父类为子类赋值  --------------------------------------------------------------\*/  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace \_02\_01\_调用父类构造函数为子类赋值  {  class Program  {  static void Main(string[] args)  {  Son son = new Son("www", 12, "B");  Console.WriteLine(son.Lastname);  Console.ReadKey();  }  }  class Father  {  public string Lastname { get; set; }  public double Property { get; set; }  public string BloodType { get; set; }  public Father(string name, double property, string bloodType)  {  this.Lastname = name;  this.Property = property;  this.BloodType = bloodType;  }  }  class Son : Father  {  public Son(string name, int property, string bloodType)  : base(name, property, bloodType)  {  }  public void Playgame()  { }  }  class Daughter : Father  {  public Daughter(string name, int property, string bloodType)  : base(name,property,bloodType)  {  }  public void Dance()  {  }  }  } |

## 属性

定义字段为public，可以从类的外部来访问该字段，定义字段为private，将禁止从类的外部访问。然而，这种形式的封装（参考封装的[访问修饰符](#_访问修饰符)）通常过于彻底。例如，你可能希望字段从外部只读，但是从内部可以更改，又类如，你可以需要允许对类中的一些数据进行操作，但需要验证对数据进行的更改。

以前，编程语言为了实现这些要求，采取的方法是将字段标记为私有，然后提供取值和赋值方法来访问和修改数据。下面的代码中将\_gameName改为私有字段，公共方法GetName和SetName用于访问和修改字段\_gameName的值。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  class MClass  {  private string \_gameName = "GM";//私有字段  /// <summary>  /// 获取私有字段的值  /// </summary>  /// <returns></returns>  public string GetName()  {  return \_gameName;  }  /// <summary>  /// 修改私有字段的值  /// </summary>  /// <param name="newName"></param>  public void SetName(string newName)  {  if (newName != null && newName != "")  {  \_gameName = newName;  }  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  MClass mc = new MClass();  mc.SetName("Frank");  }  } |

遗憾的是，这一更改会影响到MClass类的可读性。无法再用赋值运算符来设置类中的数据。另外，要想访问数据，只能调用方法来进行。C#提供了一种语法可以用来访问或修改数据，这种方法叫做属性（property）。

属性用来代表类的实例或类中的一个数据项成员。与字段相同的是，它是命名的类成员、有类型、可以被赋值和读取；然而，和字段不同的是，属性是一个函数成员，它不为数据存储分配内存，它是执行的代码。

本质上，属性是拥有一组访问器的方法。下面是属性声明的语法，它声明了一个名为MValue的属性，它由两个访问get和set，用来访问和修改数据的值。

### 属性的访问器

set和get访问器有预定义的语法和语义。可以把访问器想象成方法，但是访问器声明既没有显示的参数，也没有返回类型，因为它们已经在属性的类型中隐含了。set访问器有一个隐式值参value，可以用它发送数据到访问器块内；set访问器的块内必须包含一条return语句，返回一个属性类型的值。

set和get访问器可以任何顺序声明，并且，除了这两个访问器外在属性上不允许有其他方法。

下面的代码详细介绍了属性声明的语法以及访问器的用法：

|  |
| --- |
| **属性类型**  **属性名**  int MValue  {  get  {  **总是返回属性类型的值**  return something;  }  set  {  **返回类型为void**  }  **没有显示的参数声明，隐式参数名称为value，并与属性有相同的类型**  } |

### 一个属性的示例

下面的代码展示了一个名称为MClass的类，它有一个名为GameName的属性。需要说明的是，属性是执行的代码，不会为其分配内存。

|  |
| --- |
| class MClass  {  private string \_gameName = "GM";//私有字段,分配内存  /// <summary>  /// 属性：用来获取和设置私有字段的值；不会其分配内存。  /// </summary>  public string GameName  {  get  {  return \_gameName;  }  set  {  \_gameName = value;//value代表用户赋值过来的值  }  }  } |

### 使用属性

写入和读取属性的方法与访问字段一样。访问器被隐式调用。

* 写入属性：在赋值语句的左边使用属性的名称。
* 读取属性：在表达式中使用属性的名称。

例如，下面的代码通过属性设置\_gameNam的值为“SocialGame”，并用属性访问它的值。

|  |
| --- |
| public class NewBehaviourScript : MonoBehaviour {  void Start () {  MClass mc = new MClass();  print(mc.GameName);//输出结果为：GM  mc.GameName = "Social Game";//写入属性  string Gname = mc.GameName;// 读取属性  print(Gname);//输出结果为：Social Game  }  } |

当写入或读取属性时，适当的访问器会被隐式调用，不能显示地调用访问器。试图这样做会产生一个编译错误。

### 关联字段与执行运算

从前面几节的学习中，我们知道属性的属性的一个重要作用：用来关联私有字段，并声明一个public属性以提供受控的从类外部对字段的访问。和属性关联的字段被称为后备字段或后备存储。

属性和它们的后备字段有两种命名约定。一种约定是两个名称使用相同的内容，但字段使用Camel大小写（首字母小写），属性使用Pascal大小写。另一种约定是属性使用Pascal大小写，对于字段，使用Camel大小写，并以下划线开始。

下面的代码展示了两种约定：

|  |
| --- |
| class MClass  {  private string \_firstField;//下划线与Camel大小写  public string FirstField//Pascal大小写  {  set { \_firstField = value; }  get { return \_firstField; }  }  private string secondField;//amel大小写  public string SecondField//Pascal大小写  {  set { secondField = value; }  get { return secondField; }  } |

属性并不局限于仅仅对关联的后备字段传进传出数据。它还能执行其他的运算。下面的代码展示了一个名为Age的属性，set访问器在设置关联字段之前实现过滤。当设置的值大于18后，它返回的值为18。

|  |
| --- |
| private int \_age;//私有字段  /// <summary>  /// 属性  /// </summary>  public int Age  {  set //设置字段值  {  \_age = value > 18 //确保值不大于18  ? 18  : value;  }  get //获取字段值  {  return \_age;  }  } |

### 只读和只写属性

可以通过忽略访问器的声明，以设置属性为只读或只写。只有get访问器的属性称为只读属性，这是一种安全的方法，可以把一项数据从类或类的实例中传出，而不允许太多的访问；只有set访问器的属性称为只写属性，它可以把一项数据从外部传入类而不允许太多访问的方法。

两个访问器中至少有一个必须定义，否则编译器会产生一条错误信息。

### 自动实现属性

因为属性经常被关联到后备字段，C#中提供了自动实现属性（auto-implemented property），允许只声明属性而不声明后备字段。

自动实现属性不需要声明后备字段，访问器的方法体被简单声明为分号。下面的代码展示了一个自动实现属性的示例。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  class MClass  {  public int MNmuber//自动实现属性会分配内存  {  set;//访问器简单声明为分号  get;  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  MClass mc = new MClass();  print(mc.MNmuber);//输出值为：0  mc.MNmuber = 20;//设置属性的值为20  print(mc.MNmuber);//输出为：20  }  } |

### 静态属性

属性也可以声明为static。静态属性的访问器和所有静态成员一样：

* 它不能访问类的实例成员——虽然它们能被实例成员访问
* 不论类是否有实例，它都存在
* 当从类的外部访问时，必须使用类名引用

例如，下面的代码中有一个名为MClass的类，它带有名称为MNumber的静态属性，并关联到静态字段\_mNumber。之后再Start方法中，用类名加属性名称来访问静态属性，而不是用实例名称的方式来访问。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  class MClass  {  private static int \_mNumber;  public static int MNmuber//自动实现属性会分配内存  {  set { \_mNumber = value; }  get { return \_mNumber; }  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  print(MClass.MNmuber);  MClass.MNmuber = 20;  print(MClass.MNmuber);  }  } |

### 访问器的访问修饰符

默认情况下，成员的两个访问器有和成员自身相同的访问级别。也就是说，如果一个属性有public访问级别，那么它的两个访问器都有同样的访问级别，对索引也一样。

在特定情况下，成员的访问器可以有不同的访问级别。访问器的访问修饰符有以下几项限制：

* 仅当成员（属性或索引）既有get访问器也有set访问器时，其访问器才能有访问修饰符。
* 虽然两个访问器都必须出现，但它们中只能有一个有访问修饰符。
* 访问器的访问修饰符修饰符必须比成员的访问级别有更严格的限制性。下图阐明了访问级别的层次。访问器的访问级别在图表中的位置必须比成员的访问级别的位置低。例如，一个属性的访问级别是protected，那么访问器能使用的修饰符只有private。但是当属性访问级别是public时，访问器可以使用任意一个级别的修饰符。

|  |
| --- |
| **private**  **protected**  **internal**  **protected internal**  **public** |

例如，在下面的代码中，属性Name有public访问级别，但是set访问器有protected访问级别。

|  |
| --- |
| class MClass  {  private string \_name;  public string Name  {  get { return \_name; }  protected set { \_name = value; }//访问器有更低的访问级别  }  } |

### 属性的作用

属性的最大作用是对字段的赋值进行限制。比如有一个age字段，输入的应该是大于0小于60，这时就可以在属性中进行条件判断来限制赋值（摒除非法值）。如下：

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class StudyClass : MonoBehaviour  {  void Start()  {  Person hyfdbd = new Person();//实例化类，即创建对象  hyfdbd.Age = -1;  print(hyfdbd.Age);  }  }  class Person  {  private int age;  public int Age  {  get  {  return age;  }  set  {  //如个年龄小于0或大于60直接返回。不赋值。默认值为0.  if (value < 0 || value > 60)  {  return;  }  this.age = value;  }  }  } |

### 常见错误

属性在使用上与字段相似，但是在构造上它是函数。因此在调用或赋值时，看起来好像是为属性的变量赋值，实质上是调用属性中的get和set访问器（也可以看作是方法）。

下面的代码是一个迭代死循环，从中可以看到属性的本质

|  |
| --- |
| public class StudyClass : MonoBehaviour  {  void Start()  {  Person hyfdbd = new Person();//实例化类，即创建对象  hyfdbd.Age = 30;//属性赋值实质上是调用了Age属性的set访问器。  }  }  class Person  {  private int age;  public int Age  {  get  {  return this.Age; // 这是个迭代，看起来是返回Age的值，实质上是调用Age属性的sget访问器。这是一个迭代死循环。  }  set  {  this.Age = value;// 这是个迭代，看起来是为Age进行赋值，实质上是调用Age属性的set访问器。它是一个迭代死循环。  }  }  } |

## 索引器

索引和属性在很多方面相似，和属性一样，索引不需要分配内存来存储；它们都主要被用来访问其他数据的成员，但属性通常访问的是单独的数据成员，而索引通常访问多个数据成员。可以把索引想象成提供获取或设置类的多个数据成员的属性。

### 声明索引

声明索引的语法如下表。需要注意以下几点：

● 索引没有名称。在名称的位置是关键字this

● 索引参数在方括号中间

● 参数列表中至少要声明一个参数。

|  |
| --- |
| ReturnType this[type param，……]  {  get  {  ...  }  set  {  ...  }  } |

下面是一个索引声明的例子，这个索引与字段\_name，\_grade以及\_city相关联。

|  |
| --- |
| class MClass  {  private string \_name;  private string \_grade;  private string \_city;  public string this[int index] //索引 用来与字段相关联  {  set  {  switch (index)  {  case 0:  \_name = value;  break;  case 1:  \_grade = value;  break;  case 2:  \_city = value;  break;  default:  throw new ArgumentOutOfRangeException("index");  break;  }  }  get  {  switch (index)  {  case 0:  return \_name;  case 1:  return \_grade;  case 2:  return \_city;  default:  throw new ArgumentOutOfRangeException("index");  break;  }  }  }  } |

### 使用索引

索引的使用非常简单，可以通过使用实例名称加索引值来访问或设置相应的关联字段。例如在下面的代码中，为MClass声明了一个实例，之后通过索引对MClass的字段进行访问。

|  |
| --- |
| void Start () {  MClass mc = new MClass();  mc[0] = "Frank";//用索引的方式为 字段\_name 设置值为 Frank  mc[1] = "3 Grade";//用索引的方式为 字段\_grade 设置值为 3 Grade  mc[2] = "BeiJing";//用索引的方式为 字段\_city 设置值为BeiJing  print(mc[0] + mc[1]+ mc[2]);  } |

## 其他类成员

## 综合示例：类的基本构成

下面的代码，创建了一个类Person，这个类模拟了人的身高、名字、年纪的属性，并且模拟了说话的能力，最后通过实例化创建出了这个类的对象hyfdbd，并且对对象进行了具体设置。

|  |
| --- |
| public class StudyClass : MonoBehaviour {  // Use this for initialization  void Start ()  {  Person hyfdbd = new Person();//实例化类，即创建对象  //设置对象的字段与方法。  hyfdbd.Height = 170;  hyfdbd.Name = "高大上";  hyfdbd.Say("你好！");    }    }  class Person  {  /// <summary>  /// 定义字段：Person的数据属性，即状态。  /// </summary>  public int Height;  int Age;  public string Name;  /// <summary>  /// 定义方法：Person类可以执行的动作  /// </summary>  /// <param name="str">说的话</param>  public void Say(string str)  {  Debug.Log(Name + "说: " + str);  }  } |

# 索引器

索引器是类的一种成员，它使得对象可以像数组一样被索引，使程序看起来更直观，更容易编写。

类似于属性的访问器，索引器也是由一组get和set访问器组成。

索引与属性的对比如下：

* 两者都不用分配内存来存储
* 都用来访问其他数据成员。它们为这些成员提供设置和获取访问。属性通常访问单独的数据成员。索引通常访问多个数据成员。
* 两者都可以只有一个访问器，也可以两者都有。
* 实现set和get的访问器都不必一定关联到某个字段或属性。这段代码可以做什么也可以什么都不做，只要get访问器返回某个指定类型的值即可。
* 与属性不同的是，索引总是实例成员，不能声明为static。

## 声明和使用索引

声明索引的语法如下。请注意以下几点：

* 索引没有名称，在名称的位置是关键字this。
* 参数列表在方括号中间。
* 参数列表中至少必须声明一个参数。

|  |
| --- |
| Returntype this(Type param, …)  **关键字**  **返回类型**  **参数列表** |

例子如下：

|  |
| --- |
| private string[] weapons = new string[5];  public string this[int index]  {  get  {  if (!(index >= 2 || index < 0))  {  return weapons[index];  }  else  {  return "只能携带2个武器" ;  }    }  set  {  if(!(index>=2||index<-1))  {//  weapons[index] = value;  }    }  } |

索引不仅能设置为int类型还能设置为其它数据类型，如string

|  |
| --- |
| private string weapon1;  private string weapon2;  public string this[string weapon]  {  get  {  if(weapon =="gun1")  {  return weapon1;  }  else if(weapon =="gun2")  {  return weapon2;  }  else  {  return "错误";  }  }  set  {  if (weapon == "gun1")  {  weapon1 = value;  }  else if (weapon == "gun2")  {  weapon2 = value;  }    }  } |

## 使用索引

索引器的使用方法和数组相似，一般形式如下：

|  |
| --- |
| **对象名[索引]** |

其中索引的数据类型必须与索引器的索引类型相同。例如：

|  |
| --- |
| Weapon w = new Weapon();  w[0] = "knife";  w[1] = "gun";  w[2] = "sword";  print("你携带的武器有"+w[0] +","+ w[1]+","+ w[2] ); |

以及

|  |
| --- |
| Weapon w = new Weapon();  w["gun1"] = "倚天剑";  w["gun2"] = "屠龙刀";  print("你携带的武器有" + w["gun1"] + "," + w["gun2"]); |

## 示例演示

下面的例子中，通过索引器对weapons数组进行了设置和访问。

|  |
| --- |
| class Weapon  {  private string[] weapons = new string[5];  //使用索引可以为多个数据成员、数组的设置和访问进行限制。  public string this[int index]  {  get  {  if (!(index >= 2 || index < 0))  {  return weapons[index];  }  else  {  return "只能携带2个武器" ;  }    }  set  {  if(!(index>=2||index<-1))  {  weapons[index] = value;  }    }  }  }  class Program  {  void Start()  {  Weapon w = new Weapon();  w[0] = "knife";  w[1] = "gun";  w[2] = "sword";  print("你携带的武器有"+w[0] +","+ w[1]+","+ w[2] );  }  } |

# 封装

在上一章，我们详细讲解了类以及对象的基本结构和使用方法。从这一章开始，将真正进入到面向对象思想的核心。面向对象有三大特征：封装、继承和多态。而封装正是面向对象的核心概念之一。

本章主要讲述如何在C#中实现封装。C#中的封装包括：命名空间的使用、类的访问控制、类成员的访问控制。设计这些语法的根本用意是更好更灵活的隐藏细节。

## 什么是封装

封装（encapsulation），又叫隐藏实现（Hiding the implementation），具体的意思就是将实现的细节隐藏起来，只将必要的功能接口对外公开。使用这些代码单元的用户只知道这些代码单元可以提供哪些服务，却不知道这些代码单元中的具体逻辑是怎样的。

封装：封装复杂的、变化的东西。暴露给用户简单的、稳定的。

举一个赛车游戏的例子：当我们按键盘或手柄上的向上按键时汽车会前进，按向下的按键时会减速，按空格键会停止，按shift键会加速。但是，为什么按向上按键会前进，按向下按钮会减速？这些实现的细节我们并不知道。对玩家来说，也不需要知道。

我们把向下、向上、空格键、shift等按键称为和使用者交流的接口。封装是什么？封装就像游戏中的赛车，只将必要的功能通过接口暴露，而玩家不需要了解其内部的实现机制。

那么如何让一部分接口暴露呢？如何隐藏不对外公开的方法呢？使用访问修饰符，这是实现封装的一个重要方法。访问修饰符用来告诉外部使用者，本代码单元有哪些成员可以被外部访问到，哪些不能访问到。C#中根据访问级别的不同分为：public、protected internal、 protected、internal 、private。一个优秀的面向对象程序，会尽可能将代码的细节隐藏起来。对代码单元访问控制越严格，以后修改起来的自由越大。

## 为什么需要封装

看了上面的介绍，对封装有个模糊印象之后，再来详细说说为什么需要封装，也就是为什么会产生封装这个概念，以及封装到底是为了解决什么问题。

### 封装把程序分成一个个相对独立的单元

通过封装，可以把具有相同功能或行为的代码分成一个单元。最小的单元就是类，多个类共同执行一个任务。多个类组成的代码块叫做模块。一个程序由多个模块组成。

在设计程序时，我们首先要做的是打好框架、划分好各个模块、然后才是具体的各个模块的代码的编写。编写过程中，只需要抽象地思考模块之间的协同工作方式即可，不需要细化到每个流程的具体实现，模块之间依赖程度大大降低。这正是面向对象编程相对于流程式编程的巨大优点之一。

### 封装使修改代码更加容易和安全

开发程序时，会经常调试和修改程序。一般来说，软件的修改和调试的时间可能会与开发时间相等，甚至更长。因此，程序开发者所写的代码要经得起不断的修改。尤其是当编写的代码要提供给第三方使用时，更要小心你所做的修改，确保你的修改不会影响到其他代码。

如何做到不影响其他代码呢？每个类都有那么多的成员（字段、方法、构造函数、属性、事件等），你如何知道其他程序员使用了哪些成员？不知道就无从修改！如果有个标识，标识哪些是第三方可以访问的，哪些是第三方不能访问和修改的就好了。这就要靠访问修饰符来控制访问权限了。第三方开发者只能修改那些可以访问和修改的代码，这样调试和修改时会更加安全。

此外，封装消除了命名冲突。同名的方法，由于封装在不同的对象中，不会冲突；同名的类，封装在不同的命名空间中，就可以导入命名空间来调用，从而避免冲突。在使用第三方的类库，或者程序开发规模大时，多个人员之间容易使用相同名称命名，如果没有进行类和命名空间的封装，那么解决命名冲突也是一件很麻烦的事。

### 封装使整个程序开发的复杂度大大降低

通过封装，将代码分为一个个相对独立的单元以及将实现的细节隐藏起来，让程序的复杂程度大大降低。

这种封装的思想，有一个经典的术语——“黑箱”（Black Box）。在黑箱理论中，将所有的对象都看成一个个封装好的黑箱子，对象公开一些供外部访问的公有成员。程序员通过这些公有成员与对象打交道，而并不知道对象内部是如何实现的。通过忽略细节以及划分为独立的单元，减轻了脑力负担，从而使我们能在更高的层次上考虑软件的架构设计。越复杂的系统，越能体现这种面向对象思维方式的巨大优势。

从根本上说，降低复杂度才是封装这个思想产生的最真实原因。

## 访问修饰符

所有类型和类型成员都具有可访问性级别，用来控制是否可以在程序集的其他代码中或其他程序集中使用它们。 访问性级别是在声明类或类成员时，由访问修饰符控制的。

C#中类的访问级别，有两种修饰符：internal、public。

成员的访问级别有五种：

* Public
* Private
* Protected
* Internal
* Protected internal

### 类访问修饰符

在使用C#进行unity编程时，类的访问只有两个级别：public和internal。

标记为public的类可以被其他程序集访问，而标记为internal的类只能被它自己所在的程序集内的类访问，这是默认的可访问级别。

先来看一下在程序集外部如何访问到类 。在下面的例子中，左侧的代码编译为一个程序集，myAssembly.dll。它有两个类：一个为public类MC01，一个为internal类MC02。右侧的myClass02类使用了myAssembly程序集中的类，只有MC01类能被使用，不能访问到MC02，因为MC02类的修饰符为internal，只能在该程序集内被访问。

|  |  |
| --- | --- |
| namespace myAssembly  {  public class MC01  {  public int Add(int x, int y)  {  return x + y;  }  }  internal class MC02  {  public int Sub(int a, int b)  {  return a + b;  }  }  } | using UnityEngine;  using System.Collections;  using myAssembly;//引用命名空间myAssembly  class myClass02 : MonoBehaviour {  // Use this for initialization  void Start () {  MC01 mc = new MC01();//声明MC01类  print(mc.Add(3, 4));//调用MC01的Add方法  }  }  } |

在程序集内部，无论类的修饰符是public还是internal，都能被其他的类访问到，即使类不在同一个脚本文件中。

例如，下面的表格中，左边的代码定义了一个internal类myClass01，这个类是在脚本文件myClass01.cs中。右边的是脚本文件myClass02.cs中的代码，它访问了myClass01类，即使它是internal的。

|  |  |
| --- | --- |
| using UnityEngine;  using System.Collections;  internal class myClass01 : MonoBehaviour {  public void myMethod()  {  print("this is myClass01");  }  } | using UnityEngine;  using System.Collections;  class myClass02 : MonoBehaviour {  void Start ()  {  myClass01 mc = new myClass01();//在myClass02.cs脚本文件中调用myClass01.cs文件中的 myClass01类  mc.myMethod();//调用myClass01类中的myMethod方法  }  } |

### 类成员访问修饰符

声明在类中的成员有五个访问级别，分别使用了修饰符： Public、Private、Protected、Internal、Protected internal。

在讲述类成员访问性的细节之前，我们首先阐述一些通用内容：

* 无论它们的访问修饰符是什么，声明在类声明中的成员是互相可见的。
* 不指定访问修饰符的话，它的隐式访问级别为private。
* 类成员不能比它的类更可访问。也就是说，如果一个类的可访问性限于它所在的程序集，无论成员的修饰符是什么，类成员的个体也不能从程序集的外部访问到。

例如，下面的示例中创建了一个internal类myMember，并声明了五个级别的字段成员。即使修饰符为public，成员m1也不能在程序集外部使用；即使修饰符为private，成员m5也能被其他的成员访问。

|  |
| --- |
| using UnityEngine;  using System.Collections;  internal class myMember : MonoBehaviour {  public int m1;//即使修饰符为public，因为类访问级别为internal，所以它也不能从程序集外部访问到。  protected internal int m2;  protected int m3;  internal int m4;  private int m5;  int m6;//默认为private。  void Start()  {  int i = m5;//即使m5为private也可以被类内的其他成员访问  print(i);  }  } |

下面，我们来详细讲解一下成员访问的五个级别。

1. Public

Public是访问级别限制最少的。Public（公有）类的public（公有）成员可以被同一程序集或其他程序集的所有类访问。Internal（内部）类的public成员可以被同一程序集的所有类访问。

要声明一个公有成员，使用public修饰符，语法如下：

|  |
| --- |
| public int m1; |

1. Private

private（私有）成员是访问级别限制最严格的。任何类的private只能被它自己的类（或嵌套类）的成员可见。它不能被其他的类成员，包括继承它的类。

声明私有成员，使用private修饰符，如下：

|  |
| --- |
| private int m5; |

1. Protected

Protected（受保护）成员对它自己的类的成员或派生类的成员可见。如果受保护成员是属于public类的，那么派生类甚至可以在其他程序集中。

声明受保护的成员，使用protected 修饰符，如下：

|  |
| --- |
| internal int m4; |

1. Internal

标记为internal（内部）的成员对程序集内的所有类可见，但对程序集外部的类不可见。

声明为内部的成员，使用internal修饰符，如下：

|  |
| --- |
| internal int m4; |

1. Protected internal

标记为protected internal（受保护的内部）的成员可以被所有继承该类的类以及所有程序集内部的类可见。

声明受保护内部成员，使用protected internal修饰符，如下：

|  |
| --- |
| protected internal int m2; |

下面的表格概括了五种访问级别的特征，并直观描述了修饰符的作用。

|  |  |
| --- | --- |
| 修饰符 | 含义 |
| private | 只在类的内部可以使用 |
| internal | 对该程序集内所有类可访问 |
| protected | 对所有继承该类的类可见 |
| protected internal | 对所有继承该类或在该程序集内声明的类可见 |
| Public | 对任何类可见 |

## 访问修饰符与封装

封装可以说通过访问修饰符实现的。通过访问修饰符把不需要对外公开的字段、属性、方法等隐藏起来，开发者只需要关注公开的字段、属性、方法，从而使程序开发复杂度降低，修改与阅读代码也更加容易与安全。

例如，我们不希望在外部对代码的年龄进行设置，但是能实现有条件的对年龄赋值，就可以通过下面的方式实现：

|  |
| --- |
| using UnityEngine;  using System.Collections;  public class StudyClass : MonoBehaviour {  // Use this for initialization  void Start ()  {  Person hyfdbd = new Person();//实例化类，即创建对象  hyfdbd.gaveName("高大上");//通过方法对人名进行赋值。  hyfdbd.printName();  }    }  class Person  {  private string Name;// 外部不能访问name字段  /// <summary>  /// 通过公共的方法对name进行有条件的赋值  /// </summary>  /// <param name="name">人的名字</param>  public void gaveName(string name)  {  //如果姓名不包含张，接受输入结果。姓名包含张，则输出：我才不要姓张  if(!name.Contains("张"))  {  this.Name = name;  }  else  {  this.Name = "我才不要姓张";  }  }  /// <summary>  /// 打印人的名字。  /// </summary>  public void printName()  {  Debug.Log(this.Name);  }  } |

## 访问级别约束

### 子类的访问级别不能比父类的高

会暴露父类的成员。子类的访问修饰符必须与父类的访问修饰符兼容。

|  |
| --- |
| class Parent  {  protected string name { get; set; }  }  //子类的访问级别不能比父类高  public class child :Parent  {  } |

### 类中属性或字段的访问级别不能比其参数的访问级别高

方法的参数的访问修饰符必须与方法的访问修饰符兼容

|  |
| --- |
| class myClass  {  protected string name { get; set; }  }  public class MyClass1  {  //方法的访问级别不能比其参数的访问级别高  public void M1(myClass mc)  {  Console.WriteLine("w");  }  } |

### 方法的访问级别不能比返回值的访问级别高

方法的返回值的类型修饰符必须与方法的访问修饰符兼容。

|  |
| --- |
| class myClass  {  protected string name { get; set; }  }  public class MyClass1  {  //方法的访问级别不能比其返回值的访问级别高  public myClass M1()  {  Console.WriteLine("w");  return null;  }  } |

## 命名空间和程序集

命名空间用于解决

命名空间用于对类型进行逻辑分组，主要针对开发人员。程序集则是程序的物理分组，主要针对安装和部署，对应于一个dll或exe文件。

同一个命名空间中可以在多个程序集中，比如一个公司分别由不同的部门共同开发同一个命名空间的不同组件。那么不同的部门可以把他们开发的东西编译成各自的dll文件。使用时把几个dll一起引用即可。

一个程序集中也可以有多个命名空间，比如程序的整体框架，可能调用了多个命名空间中的代码来完成最终的功能。

### 命名空间

在编写程序、特别是大型程序时，经常会遇到这样的情况：需要使用到第三方开发者编写的程序集或脚本文件。很可能有一定数量的类名重复。如果仅仅因为类名重复，就不能在一个程序中同时使用这两个类，这是极为浪费和不方面的。命名空间就是为了解决这个问题而存在的。

下面展示了声明一个命名空间的语法。声明在大括号中间的所有类都是该命名空间的成员。

|  |
| --- |
| namespace sampleNamespace  {  class A  **关键字**  {  **命名空间名称**  }  class B  {  }  } |

当第三方开发者编写的程序集有相同的名称时，使用完全限定名就可以区分使用的那个开发者开发的类。

完全限定名由命名空间名称和类名组成，中间用点运算符隔开。

例如，下面的示例中声明了两个名称相同的类playerClass，这两个类属于不同的命名空间。使用完全限定名，就可以区分出类playerClass到底是哪一个。

|  |
| --- |
| public class NewBehaviourScript : MonoBehaviour {  **完全限定名**  **完全限定名**  // Use this for initialization  void Start () {  corpA.playerClass = new corpA.playerClass();// 声明playerClass并创建实例  corpB.playerClass = new corpB.playerClass();// 声明playerClass并创建实例    }  **类名称**  **命名空间名称**  } |

命名空间把多个类分组在一起并给它们一个名称，称为命名空间名称。命名空间名称的命名规则是它的名称应该描述命名空间的内容并和其他命名空间名称相区别。下面是命名空间命名方法的建议：

* 使用公司名开始命名空间名称。
* 在公司名之后跟着技术名称。
* 不要把空间名称命名为与类或类成员相同的名称。

例如，ZYCG公司在下面的三个命名空间中开发软件。如下面代码所示：

* ZYCG.Media
* ZYCG.Game
* ZYCG.Display

|  |
| --- |
| namespace ZZCG.Game.GameControl  {  class A  {  }  class B  {  }  } |

一个命名空间还可以是另一个命名空间的成员。这个成员被称为嵌套的命名空间。有一点需要注意的是，虽然嵌套的命名空间是外层命名空间的成员，但它的成员不是外层命名空间的成员。

有两种方式声明一个嵌套的命名空间：

* 原文嵌套: 把命名空间的声明放在一个封装的命名空间声明体内部，从而创建一个嵌套命名空间。

|  |
| --- |
| namespace sampleNamespace  {  namespace nestNamespace//嵌套命名空间  {  }  } |

* 分离的声明：嵌套声明也可以以分离的方式声明，但必须在声明中使用完全限定名称。

|  |
| --- |
| namespace sampleNamespace  {  }  namespace sampleNamespace.nestNamespace//使用完全限定名创建嵌套命名空间  {  } |

### using指令

完全限定名可能非常长，在代码中多次使用时会显得臃肿、难读、增加调试的难度。然而，有两个指令可以避免不得不使用完全限定名：

* using命名空间指令
* using别名指令

关于using指令有两个要点：

* 它们必须放在源文件的顶端，在任何类型声明之前。
* 它们应用于当前源文件中的所有命名空间。

using命名空间指令由关键字using跟着一个命名空间标识符组成。

|  |
| --- |
| using UnityEngine;  **命名空间名称**  **关键字** |

using命名空间指令的作用是告诉编译器你将要使用来自某个指定命名空间的类。当使用到这个指定命名空间中的类时，就不需要使用完全限定名，只需要类名就可以。

我们在前面使用到的log方法，它是类Debug的成员，在UnityEngine命名空间中。在代码的顶端使用using命名空间指令以描述使用来自UnityEngine命名空间的类，就不再需要使用完全限定名了。如下面的代码所示：

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript : MonoBehaviour  {  void Start ()  {  Debug.Log("welcome");//如果不在源代码顶端使用using指令，指明使用UnityEngine中的类，就需要使用完全限定名: UnityEngine.Debug.Log()  }  } |

Using别名指令允许给一个命名空间或命名空间内的一个类起一个别名。

例如，下面的代码展示了using别名指令的两个使用方法。第一个指令告诉编译器标识符UE是命名空间UnityEngine的别名，第二个指令告诉编译器标识符UD是类UnityEngine.Debug的别名。

|  |
| --- |
| **命名空间**  **别名**  **关键字**  using UE = UnityEngine;  using UD = UnityEngine.Debug;  **类名称**  **关键字**  **别名** |

下面的代码使用这些代码。在Start方法中都调用了log方法。

|  |
| --- |
| using UE = UnityEngine;  using UD = UnityEngine.Debug;  public class NewBehaviourScript : UE.MonoBehaviour {  void Start ()  {  UnityEngine.Debug.Log("welcome");//使用完全限定名  UE.Debug.Log("welcome");//使用命名空间UnityEngine的别名UE  UD.Log("welcome");//使用类 UnityEngine.Debug的别名UD  }  } |

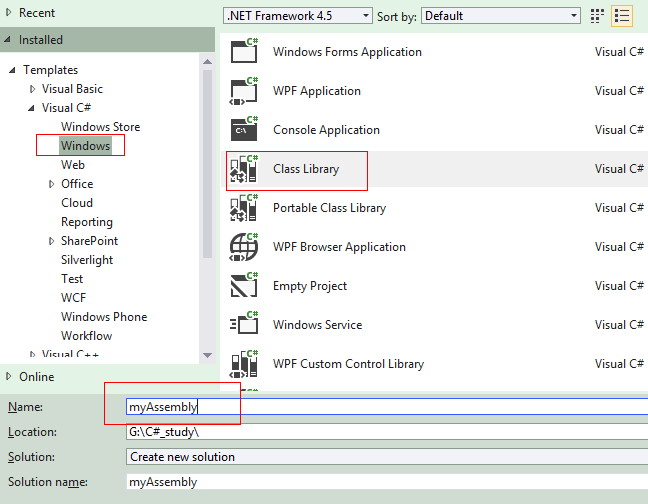
### 创建并使用程序集

程序集包含描述它们自己的内部版本号和它们包含的所有数据和对象类型的详细信息的元数据。程序集仅在需要时才加载。如果不使用程序集，则不会加载。这意味着程序集可能是在大型项目中管理资源的有效途径。程序集可以包含一个或多个模块。例如，计划较大的项目时，可以让几个各个开发人员负责单独的模块，并通过组合所有这些模块来创建单个程序集。

创建程序集的方法很简单，它相当于一个项目。下面的步骤创建了一个名为myAssembly的程序集：

1. 创建类库文件。

在visualstudio中选择“File> New>Project ”命令，在弹出的窗口中，从模板中选择class Library，并修改名字为myAssembly，最后点击ok创建类库

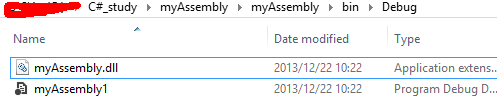


1. 在class1.cs文件中添加如下代码：

|  |
| --- |
| namespace myAssembly  {  public class myClass  {  public int Add(int a, int b)  {  return a + b;  }  }  } |

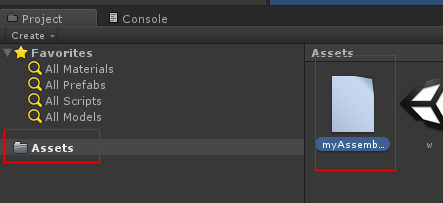
1. 创建解决方案，并找到名为myAssembly.dll的文件。

在visualstudio中选择“Build>Build Solution”命令，创建解决方案。之后在项目的Debug文件夹中找到myAssembly.dll文件。



1. 添加程序集到unity中。

拖拽myAssembly.dll到unity的project面板的assets文件夹中。



1. 在脚本文件中使用程序集。

在unity中添加脚本文件，并使用using指令引用程序集中的命名空间，然后就可以想在同一个程序集内部一样，使用myAssembly指令集中的类了。

下面的代码展示了程序集的使用方法：

|  |
| --- |
| using UnityEngine;  using System.Collections;  using myAssembly;//引用myAssembly命名空间  public class NewBehaviourScript : MonoBehaviour {  // Use this for initialization  void Start () {  myClass mc = new myClass();//创建myClass的实例  int i = mc.Add(4, 3);//使用Add方法  print(i);  }  } |

# 复合与继承

复合和继承的核心思想是重用现有的代码。重用的思想在各种语言中都有相应的实现机制，在面向对象编程中一般是指重用现有的类（class）。这些现有的类或许是自己以前就写好并调试成功的、或许是第三方开发的类库（Class Library）。重用的机制是创立一些新的类，来利用这些现有的类。

创建新类的途径有两种：

* 复合（composition）：直接在新类中创建现有Class的对象，成为新类的一部分。这样可以直接使用这些对象所提供的功能。但这些对象外部的接口形式不一定被采用。
* 继承（inheritance）：声明一个新类作为现有的类的之类，然后这个新类就具有了现有类的功能。在使用时，新类的使用形式和现有类一样，而在新类中修改和增加代码也不需要改动现有类的代码。

这两种方式固然目标一致，但实现机理不同，使用上也有所区别。下面开始讲述这两者在C#中的应用。

## 复合

复合的原理是在新创建的类中有一个或多个对象的引用，而这些对象是现有类的实例。在实际编程中，使用复合的频率要远远超过继承。复合比继承灵活，而且更加直白，易于管理。

在下面的例子中，有一个复合类Human。Human类中组合了Ear类对象（耳）、Eye类对象（眼）、Brain类对象（脑）和Mouth类对象（嘴）；就像我们人有耳朵、眼睛、大脑和嘴巴一样。

还有两个独立的类：Book和Song类。眼睛可以看书，耳朵可以听歌，嘴巴可以说话。但是，这书和歌这两种对象不能直接被Mouth对象理解，只有书和歌的内容转成字符串形式的信息，才能被Mouth对象理解，所以单个的Mouth对象无法说出书或歌的内容。

但是复合类Human的对象就可以。Human类组合了Eye类对象、Ear类对象、Mouth类对象以及Brain类对象。Ear类对象可以听歌、Eye类对象可以看书，Brain对象可以记住眼睛和耳朵返回的信息，Mouth类可以读出Brain记住的信息。一加一远远大于二，通过组合的对象互相写作，从而完成单个对象不能完成的事情，正是复合的巨大优点之一。

复合类的外部的表现形式比较灵活。复合类可以不采用组合类的对外接口的名称，比如听歌在Human类中就是enjoy()，而不是Ear类的hear()；但是可以采用组合类的对外接口名称，比如Human类看书的接口read()和类Eye类的接口read就相同。

下面的代码虽稍长但非常易读，分为三大块：

* 文档类NewBehaviourScript1：初始化Human类，并让Human做一些事情。
* 复合类Human：组合了Ear、Eye、Mouth、Brain等4个类的对象，并新建了3个对外接口，即read、enjoy以及tell。
* 4个被复合的类：Ear、Eye、Mouth、Brain。两个独立的类：Book和Song。

|  |
| --- |
| using UnityEngine;  using System.Collections;  /// <summary>  /// 复合类Human，复合了嘴、眼、耳、脑等类  /// </summary>  class Human  {  private string \_name;  private Mouth \_mouth = new Mouth();  private Eye \_eye = new Eye();  private Ear \_ear = new Ear();  private Brain \_brain = new Brain();  public Human(string name)  {  \_name = name;  Debug.Log(\_name);  }  public void read(Book abook)  {  \_eye.read(abook);  \_brain.remember(\_eye.info);  }  public void enjoy(Song aSong)  {  \_ear.hear(aSong);  \_brain.remember(\_ear.info);  }  public void tell()  {  \_mouth.speak(\_brain.memory);  }  }  /// <summary>  /// 嘴，拥有对外接口speak();  /// </summary>  class Mouth  {  public void speak(string msg)  {  Debug.Log("Mouth speak:" + msg);  }  }  /// <summary>  /// 眼：拥有对外接口read()，可以接受Book类的对象，并理解成info。  /// </summary>  class Eye  {  public string info;  public void read(Book aBook)  {  info = aBook.content;  }  }  /// <summary>  /// 耳：拥有对外接口hear()，可以接受Song类的对象，并理解成info。  /// </summary>  class Ear  {  public string info;  public void hear(Song aSong)  {  info = aSong.content;  }  }  /// <summary>  /// 脑：拥有对外接口remember()，存储字符串信息。  /// </summary>  class Brain  {  public string memory;  public void remember(string msg)  {  memory = msg;  }  }  /// <summary>  /// 书：存储书籍内容  /// </summary>  class Book  {  public string content = "The content of a book";  }  /// <summary>  /// 歌曲：存储歌曲信息  /// </summary>  class Song  {  public string content = "The lyric of a music";  }  public class NewBehaviourScript1 : MonoBehaviour {  void Start () {  Human aHuman = new Human("Frank");//创建一个名为Frank的人  Book book = new Book();//创建一本书  Song song = new Song();//创建一首歌  aHuman.read(book);//读书  aHuman.enjoy(song);//唱歌  aHuman.tell();//读出内容  }  void Update () {    }  } |

## 初始化复合对象

有三种方式初始化复合对象

* 定义属性时就初始化对象

这种方式的优点是确保复合对象一定被初始化；缺点是无论复合对象有没有用到，一定会初始化，导致资源浪费。如果复合的对象占用资源大或者稀缺，比如位图对象、网络连接，那么会占用大量资源。另外，如果需要在运行时创建复合对象，这种方式也不能被使用。

下面是定义属性时就初始化复合对象的例子：

|  |
| --- |
| class Human  {  private string \_name;  private Mouth \_mouth = new Mouth();  private Eye \_eye = new Eye();  private Ear \_ear = new Ear();  private Brain \_brain = new Brain();  } |

* 在构造函数中初始化

这是最常用的一种方式。在复合类对象被创建时，被复合的对象才会被创建。这种情况满足绝大部分的应用需要。

在构造函数中的初始化语句最好单独抽离到一个方法中。一来可以保证代码的可读性，二来较为灵活便于日后的修改。

抽离出的初始化语句放入的方法建议以init开头。如下面的例子：

|  |
| --- |
| class Human  {  private string \_name;  private Mouth \_mouth;  private Eye \_eye;  private Ear \_ear;  private Brain \_brain;  void initParts()  {  \_mouth = new Mouth();  \_ear = new Ear();  \_eye = new Eye();  \_brain = new Brain();  }  public Human(string name)  {  initParts();  } |

* 在使用对象时再初始化

当使用到复合对象时，才初始化。这种到了用时才创建的做法，一般称为懒汉式初始化。相对地，刚刚讲到的第一种初始化方法就叫饿汉式初始化。

这种初始化方式的缺点是要多写一些代码确保正确初始化，优点是节省资源。所以如果被复合的对象资源占用巨大，或者资源稀缺时，就会采用这种初始化方式。示例如下：

|  |
| --- |
| class Human  {  private Mouth \_mouth;  public void tell()  {  if(\_mouth == null)  {  \_mouth = new Mouth();  \_mouth.speak(\_brain.memory);  }  }  } |

## 继承

通过继承我们可以定义一个新类。新类拥有与原有类相同的对外接口[[1]](#footnote-1)，或许还新增了一些接口。使用原有类对象的地方，都可以被替换成新类对象。符合这两个条件的，才能称为正确的继承。

被继承的类，称为基类（Base Class），通俗称为父类。用继承创建的新类，称为派生类（Derived Class），通俗称为子类（Subclass）。

其实继承的用法，我们已经很属性了。在之前的例子中，我们看到新建脚本文件的文档类都是继承自MonoBehaviour类的。见下面的代码。

|  |
| --- |
| using UnityEngine;  public class MyScript :MonoBehaviour {  void Start ()  {  print("Hello");  }  } |

其中MyScrip是派生类，它的基类为MonoBehaviour。派生类MyScrip拥有MonoBehaviour类对外公开的所有接口。比如print方法就是基类中对外开放的一个方法。

派生类声明的方法是在派生类名称后加冒号，之后跟着基类的名称。语法如下：

|  |
| --- |
| public class DerivedClass : BaseClass  {  }  **冒号**  **基类名称**  **派生类名称** |

### 如何判断继承是否合理

判断子类是否是父类（is a）即可。比如一个鸟类有一个飞的方法，那么天鹅是否可以继承鸟类呢？因为天鹅是鸟，所以继承时合理的。那么如果飞机是否适合继承呢？飞机不是鸟，所以不适合继承。

### 继承的好处

继承有2个好处：

* 代码重用
* 多态

如果即不需要代码重用，也不需要多态，就不需要继承。

### 继承的特性

* 单根性

一个类只能有一个父类，不允许多继承。（继承多个类需要用接口实现）。

* 传递性
* 构造函数不能被继承

由于在创建子类对象时，一定会调用子类的构造函数，而任何一个子类构造函数默认情况下都会去调用父类的无参数的构造函数，所以当父类中没有无参数的构造函数时就会报错。

有3种解决方法：

1. 在父类中增加一个无参数的构造函数

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace \_02\_继承  {  class Program  {  static void Main(string[] args)  {  Teacher teacher = new Teacher("hao", 18);  Console.WriteLine("老师的名字是{0}，年龄：{1}.",teacher.Name,teacher.Age);  Console.ReadKey();  }  }  class Person  {  //创建无参数的构造函数  public Person()  {  Console.WriteLine("=====================================================");  }  public string Name { get; set; }  public int Age { get; set; }  }  class Teacher:Person  {  public Teacher(string name, int age)  {  this.Name = name;  this.Age = age;  }  }  }  /\*-------------------输出结果--------------------------  =====================================================  老师的名字是hao，年龄：18.  -------------------------------------------------- \*/ |

1. 在子类中指定子类的构造函数来调用父类中的有参数的构造函数。

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace \_02\_继承  {  class Program  {  static void Main(string[] args)  {  Teacher teacher = new Teacher("hao", 18);  Console.WriteLine("老师的名字是{0}，年龄：{1}.",teacher.Name,teacher.Age);  Console.ReadKey();  }  }  class Person  {  //创建有参数的构造函数  public Person(string name, int age)  {  this.Age = age;  this.Name = name;  }  public string Name { get; set; }  public int Age { get; set; }  }  class Teacher:Person  {  //在子类的构造函数中指明调用有参数的构造函数  public Teacher(string name, int age):base("hao",18)  {  //this.Name = name;  //this.Age = age;  }  }  }  /\*-------------------输出结果--------------------------  老师的名字是hao，年龄：18.  -------------------------------------------------- \*/ |

1. 调用自己类中的重载的构造函数

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace \_02\_继承  {  class Program  {  static void Main(string[] args)  {  Teacher teacher = new Teacher();  Console.WriteLine("老师的名字是{0}，年龄：{1}.",teacher.Name,teacher.Age);  Console.ReadKey();  }  }  class Person  {    public string Name { get; set; }  public int Age { get; set; }  }  class Teacher:Person  {    public Teacher(string name, int age)  {  this.Name = name;  this.Age = age;  }  //调用自己类中重载的构造函数  public Teacher():this("we",18)  {  }  }  }  /\*-------------------输出结果--------------------------  老师的名字是hao，年龄：18.  -------------------------------------------------- \*/ |

## 访问继承的成员

前面我们已经讲过，派生类拥有与基类相同的对外接口。就像是它们是派生类自己声明的一样。

下面的示例包含了三个类：BaseClass、DerivedClass以及NewBehaviourScript类。其中BaseClass包含了5个不同访问级别的字段。DerivedClass是BaseClass的派生类，它继承了除字段e以外的所有成员。NewBehaviourScript类中创建了DerivedClass的实例，并且使用了它对外开放的接口a、b、d以及MethodA()。这里需要注意的是受保护级别（protected level）的成员，不能被NewBehaviourScript访问。只能在基类以及派生类内部可见。关于访问级别的详情请参考[类成员访问修饰符](#_类成员访问修饰符)

|  |
| --- |
| using UnityEngine;  public class NewBehaviourScript :MonoBehaviour {  void Start ()  {  DerivedClass DC = new DerivedClass();  /// <summary>  /// 即使DerivedClass中没有声明，a、b、d以及MethodA()也可以使用。  /// </summary>  DC.a = 1;  DC.b = 2;  DC.d = 4;  DC.MehodA()  DC.c = 3;//会有执行错误，因为 protected类不能被非继承的类访问  }  }  public class BaseClass  {  public int a = 1;//可以被所有类可见  internal int b = 2;//可以被同一个程序集中的所有类访问。  protected int c = 3;//只能在派生类内部以及基类内部访问。  protected internal int d = 4;//对所有继承该类或在该程序集内声明的类可见  private int e = 5;//只在类的内部可以使用,不能被继承  }  public class DerivedClass:BaseClass  {  public void MehodA()  {  DerivedClass D = new DerivedClass();  D.c = 3;// c字段为protected，能在继承类内部访问，但是不能被非继承的类访问。  }  } |

## 继承与访问控制

上面说过，基类中非private实例成员都会被子类继承。换句话说，在子类中可以访问到父类所有的非private的实例成员。在上类中，派生类DerivedClass可以访问到a、b、c、d等字段。但是派生类不能访问基类中的private成员e。

基类中的protected成员，在派生类内部可以访问到，即使基类和派生类不在同一个程序集中，派生类也可以访基类的protected成员。如果没有继承，那么protected访问修饰符的作用 和private一样。

Public、internal以及protected internal的类成员都会被派生类继承。所不同的是，public的成员可以被任何第三方类访问，internal的类成员只有在同一个程序集中的类才能访问，protected internal的类成员可以被同一程序集中的类以及派生类访问。

下面结合继承，讲一下在设计类时，5个访问控制如何选择。

什么时候用public类成员？当这个类成员需要对外部可见，且需要作为类和外部的接口时，才可以使用public。不能仅仅因为图访问方便，而将类成员设为public，这是很不好的做法。Public类成员可以被派生类访问。

什么时候用internal类成员？首先，这个类成员不需要对包外可见，即不需要包外访问。本着访问权限越低越好的原则，能用internal绝不用public，除非是必需公开的对外接口。Internal类成员可以被派生类访问。

什么时候用protected类成员？有一个类成员需要被子类访问，而且这个类成员不能对基类和派生类的外部可见，这时候可以用protected。有时使用protected是唯一的选择。比如，基类和派生类不在同一个程序集中，如果派生类访问的类成员不能作为public从而完全公开，那么就只有使用protected访问修饰符来运行派生类跨包访问这个类成员。

什么时候用private类成员？当这个类成员只是基类的内部实现机制，派生类和其他类都不应该知道，这时就应该使用private。根据OOP惯例，使用private应该是最优先考虑的。

什么时候用 protected internal类成员？如果当前类可能会有不在同一个程序集中的子类，而且又有一个类成员必须可以被同一个程序集中的其他类访问时，就可以使用protected internal。

## 隐藏基类成员

在派生类中，经常要改变继承自基类的某个成员，但不改变名称，从而达到同名成员实现不同功能的多态效果。这时就需要隐藏基类成员。

隐藏一个继承的数据成员，需要声明一个新的相同类型的名称，并使用相同的名称；隐藏一个继承的函数成员，需要声明一个新的带有相同签名（不包括返回类型）的函数成员；显示的隐藏继承的成员，需要使用new修饰符，没有它，程序也可以编译成功，但编译器会警告你隐藏了一个基类成员。

下面的代码声明了一个基类和一个派生类，它们都有一个名称为myFiled的成员。在派生类使用new修饰符显示地告诉编译器隐藏了基类成员。

|  |
| --- |
| using UnityEngine;  using System.Collections;  /// <summary>  /// 基类  /// </summary>  class BaseClass  {  public string myField = "This is BaseClass";//声明字段myField  }  /// <summary>  /// 派生类  /// </summary>  class DerivedClass:BaseClass  {  new public string myField = "This is DerivedClass";//用new修饰符隐藏基类成员  public void printField()  {  Debug.Log(myField);//访问派生类成员，输出结果为：This is DerivedClass  Debug.Log(base.myField);//访问基类成员，输出结果为：This is BaseClass  }  }  /// <summary>  /// 文档类  /// </summary>  public class NewBehaviourScript2 : MonoBehaviour  {  void Start () {  DerivedClass MC = new DerivedClass();  MC.printField();  }  } |

有时，派生类需要访问被隐藏的继承成员。可以使用基类访问表达式访问隐藏的基类成员。基类访问表达式由关键字base后面跟着一个点和成员的名称组成。例如上面例子中的语句：

|  |
| --- |
| Debug.Log(base.myField)  **基类访问表达式** |

## 何时用复合，何时用继承

这是OOP编程中的一个经典的讨论，直至今天仍然争论不休。本书给出以下三个指导原则：

* 谨慎使用基础。新手中滥用继承是比较普遍的现象。虽然继承号称面向对象的三大基石之一，但不正确的使用继承不仅仅是代码维护的灾难，也是真实逻辑的扭曲。在现实OOP编程中，使用继承的情况是不多的。不用继承就不能带来所需要的好处时，才会考虑采用继承。
* 在编程中，需要用到向上转换（Upcasting）时，请考虑采用继承。简单地说，当代码中需要将子类对象当成父类对象使用时，请采用继承。向上转型也是多态的好处之一。将在下一章中详细讲述。
* 用“has-a”和“ia-a”来区分复合和继承，用中文说，就是用“有一个”和“是一个”来区分复合和继承。包含的关系用复合，特殊种类用继承。这是大部分OOP书籍中都要说到的一个通俗的判断方法。详细地说就是：如果觉得新类是原有类中一种特殊的分类，需要具有原有类的一切对外接口，那么采用继承；如果觉得新类对象只包含原有类对象，那么采用复合。打个比方，孔雀是鸟的一种，那么孔雀应当是鸟类的子类，使用基础比较合适。轮胎是汽车的一部分，不能因为汽车使用了论坛，就说汽车是一种特殊的轮胎。所以，轮胎对象应当复合到汽车对象中。

即使有以上3条原则，也不一定能理清思路，因为在实际编程中碰到的情况是相当复杂的，逻辑关系并不一定像比喻的例子那样清楚。碰到这种情况时，请先考虑自己的原有类和新类设计是否恰当？是否容纳了过多的内容，导致判断不清？如果这样，可否做成一个复合类，将其中的逻辑分成功能更细的类独立起来。如果还是不行，那么考虑是否需要向上转型。需要，则优先考虑继承实现，否则采用复合。如果需要向上转型，但是采用继承并不符合逻辑关系，那么这时候可以采用接口（interface）来解决。

## 生物分类学与类

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace \_09\_生物分类学的抽象  {  class Program  {  static void Main(string[] args)  {  Tiger tiger = new Tiger();  tiger.BirthPlace();//定义在Tiger类中,所有老虎都具有的特征  tiger.ToothProperties();//定义在Carnivora中，所有食肉目的动物都具有的特征。  tiger.Live();//生存方式，定义在Animalia中，所有动物都具有的特征  Lion lion = new Lion();  lion.BirthPlace();//定义在Lion类中，所有狮子都具有的特征。、  lion.ToothProperties();//定义在Carnivora中，所有食肉目的动物都具有的特征。  lion.Live();//生存方式，定义在Animalia中，所有动物都具有的特征    }  }  /// <summary>  /// 动物界  /// </summary>  public class Animalia  {  public virtual void Live()  {  Console.WriteLine("只能以光合作用来生存，只能靠捕食植物或其他动物");  }  }  /// <summary>  /// 脊索门  /// </summary>  public class Chordata:Animalia  {  public virtual void HavingChordata()  {  Console.WriteLine("具有脊索；具有背神经管；具有咽鳃裂等");  }  }  /// <summary>  ///哺乳纲  /// </summary>  public class Mammalia:Chordata  {  public virtual void bodyStructure()  {  Console.WriteLine("哺乳动物是全身披毛、运动快速、恒温胎生、体内有膈的脊椎动物，是脊椎动物中躯体结构、功能行为最为复杂的最高级动物类群，因能通过乳腺分泌乳汁来给幼体哺乳而得名");  }  }  /// <summary>  /// 食肉目  /// </summary>  public class Carnivora:Mammalia  {  public virtual void ToothProperties()  {  Console.WriteLine("牙齿尖锐而有力，具食肉齿（裂齿），可将韧带、软骨切断。大齿异常粗大，长而尖，颇锋利，起穿刺作用。");  }  }  /// <summary>  /// 猫科  /// </summary>  public class Felidae:Carnivora  {  public virtual void LimbsProperties()  {  Console.WriteLine("四肢较短些，粗壮而沉重。尾长、末端钝圆。趾行性，足下有数个球形肉垫，均等的承负着体重，形成猫类轻快的的步态。");  }  }  /// <summary>  /// 猫属  /// </summary>  public class Felis:Felidae  {  public virtual void FelisProperties()  {  Console.WriteLine("头圆、颜面部短，前肢五指，后肢四趾，趾端具锐利而弯曲的爪，爪能伸缩。夜行性");  }  }  /// <summary>  /// 种：家猫  /// </summary>  public class FelisDomesticus:Felis  {  public virtual void FelisDomesticusProperties()  {  Console.WriteLine("人类驯养的一种以肉食为主的杂食性动物，由几种旧大陆的小型野猫的杂交后代中通过自然选择而最后衍生的，有一些变种，主要的区别在于皮毛的长短，体形以及尾的有无。");  }  }  /// <summary>  /// 老虎属于猫科  /// </summary>  public class Tiger:Felidae  {  public void BirthPlace ()  {  Console.WriteLine("生在深山");  }  }  public class Lion:Felidae  {  public void BirthPlace()  {  Console.WriteLine("生在草原");  }  }  } |

# 多态

多态，英文叫做“Polymorphism”，意思是多种形态。所谓形态，具体是指数据类型（Data type）。通过继承，一个类可以被当作不止一个数据类型（type）使用，它可以被用做自身代表的数据类型，还可以被当作它的基类所代表的数据类型，乃至它实现的接口的数据类型这一机制称为"多态"。

在C#中，所有的数据类型都是多态的。任意一个数据类型都可以被当作自身来使用，也可以当作Object类型来使用，因为任何数据类型都自动以Object为自己的基类。

使用多态可以解决项目中紧偶合的问题,提高程序的可扩展性，大部分的设计模式都是基于多态而创建的。

多态的主要目的就是程序的可扩展性。多态符合“开放与封闭原则（对修改封闭，对扩展开放）”。

多态的作用：把不同的子类对象都当作父类来看，可以屏蔽不同子类对象之间的差异，写出通用的代码，做出通用的编程，以适应需求的不断变化。

## 里氏替换原则

要的是一个父类类型对象，可以给的是一个子类对象。

## 多态的基础：虚（virtual）方法和覆写（override）方法

使用虚方法可以使基类的引用访问派生类的方法。也就是基类的引用指向派生类的对象。

例如，下面的代码中，有一个基类Car，它的两个派生类Bus和Carriage。还有一个类Person，拥有一个方法参数为基类类型的driver方法。之后再Start方法中，创建了一个person对象，这个对象的driver方法根据其实参的不同程序不同的状态，这就是多态。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  /// <summary>  /// 基类car,拥有虚方法Run  /// </summary>  class Car  {  virtual public void Run()  {  Debug.Log("this is the running car ");  }  }  /// <summary>  /// 派生类carriage，拥有覆写方法Run  /// </summary>  class Carriage:Car  {  override public void Run()  {  Debug.Log("this is the running carriage");  }  }  /// <summary>  /// 派生类Bus，拥有覆写方法Run  /// </summary>  using UnityEngine;  using System.Collections;  using System;  /// <summary>  /// 基类car,拥有虚方法Run  /// </summary>  class Car  {  virtual public void Run()  {  Debug.Log("this is the running car ");  }  }  /// <summary>  /// 派生类carriage，拥有覆写方法Run  /// </summary>  class Carriage:Car  {  override public void Run()  {  Debug.Log("this is the running carriage");  }  }  /// <summary>  /// 派生类Bus，拥有覆写方法Run  /// </summary>  class Bus:Car  {  override public void Run()  {  Debug.Log("this is the running bus");  }  }  /// <summary>  /// 类person拥有driver方法，它可以调用car类中的run方法。  /// </summary>  class person  {  public void Driver(Car car)//形参的数据类型为Car，car参数持有car对象的引用。  {  car.Run();//调用run方法  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  person P = new person();//创建person的实例  Bus B = new Bus();//创建bus的实例  P.Driver(B);//把Bus的引用作为实参传递给形参car（car参数持有car对象的引用）  Carriage C = new Carriage();//创建carriage的实例  P.Driver(C);//同样的对象，同样的Driver方法，不同的行为，这就是多态  }  }/// <summary>  /// 类person拥有driver方法，它可以调用car类中的run方法。  /// </summary>  class person  {  public void Driver(Car car)//形参的数据类型为Car，car参数持有car对象的引用。  {  car.Run();//调用run方法  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  person P = new person();//创建person的实例  Bus B = new Bus();//创建bus的实例  P.Driver(B);//把Bus的引用作为实参传递给形参car（car参数持有car对象的引用）  Carriage C = new Carriage();//创建carriage的实例  P.Driver(C);//同样的对象，同样的Driver方法，不同的行为，这就是多态  }  } |

当我们添加一个新的继承类Truck时，只需要创建一个新的实例，把实例名称作为参数赋值给Driver方法，即可扩展Driver方法，这就是多态的具有的最大的优点。

|  |
| --- |
| class Truck:Car  {  override public void Run()  {  Debug.Log("this is the running bus");  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  person P = new person();  Truck T = new Truck();  P.Driver(T);  }  } |

## 基类引用

派生类的实例由基类的实例加上派生类附加的成员组成。派生类的引用指向整个类对象，包括基类部分。通过派生类对象的引用就可以获取该对象基类部分的引用。如何获取基类的引用呢？使用类型转换运算符把派生类的引用转换为基类类型即可。

例如，下面的代码中，执行“BaseClass B = D”语句后，D持有的引用隐式转换为BaseClass类型的引用，并赋值给B，变量B就可以访问基类中的成员了。

|  |
| --- |
| using UnityEngine;  using System.Collections;  class BaseClass  {  public string Field1 = "this is the BaseClass";  }  class DerivedClass:BaseClass  {    }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  DerivedClass D = new DerivedClass();  BaseClass B = D;  print(B.Field1);//输出结果：this is the BaseClass  }  } |

在[隐藏基类成员](#_隐藏基类成员)这一节中，我们知道可以使用new关键字来隐藏基类中的成员，那么当创建派生类对象后，使用基类引用的结果是什么呢？我们先看看下面的代码。

|  |
| --- |
| using UnityEngine;  using System.Collections;  class BaseClass  {  public string Field1 = "this is the BaseClass";  }  class DerivedClass:BaseClass  {  new public string Field1 = "this is the DerivedClass";    }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  DerivedClass D = new DerivedClass();  BaseClass B = D;  print(B.Field1);//输出结果：this is the BaseClass  }  } |

在执行代码后，输出的结果是：this is the BaseClass。可以得出这样的结论：基类引用访问的是被隐藏的基类成员。也就是说，在创建派生类对象时，基类的成员Field1会在内存中分配位置，只是被隐藏起来，不能通过派生类对象访问。

当覆写了基类中的方法后，又发生什么情况呢？先看看下面的代码。

|  |
| --- |
| using UnityEngine;  using System.Collections;  class BaseClass  {  virtual public void Method()  {  Debug.Log("this is the BaseClass");  }  }  class DerivedClass:BaseClass  {  override public void Method()  {  Debug.Log("this is the DerivedClass");  }    }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  DerivedClass D = new DerivedClass();  BaseClass B = D;  B.Method();//调用Method方法，输出：this is the DerivedClass  }  } |

调用基类中的方法Method后，输出的是覆写中实现的方法，而不再是基类中的方法了。

总结一下，基类的引用即可以访问基类中的成员，也可以访问到派生类中的成员。基类可以调用派生类对象中的成员，这是多态的基础。

## 虚方法特点总结

1. 在父类中把需要设置为虚方法的方法前加virtual标记。
2. 虚方法在父类中必须有实现
3. 子类继承父类后，可以选择对虚方法进行重写也可以选择不重写
4. 当子类重写父类中的方法的时候，必须保证重写后的方法与原方法的访问修饰符、返回值类型、方法名、参数列表完全一致。
5. override标记的方法也可以被重写。也就是说子类的子类也可以重写。

## 抽象类和抽象方法

抽象方法是必需被派生类覆写的方法。它没有实现的方法体，方法体被分号代替。抽象方法只能被声明在抽象类中。抽象方法所在的抽象类不能实例化。抽象方法和抽象类使用abstract修饰符声明。抽象类不能实例化，抽象类中既可以有抽象成员，也可以有实例成员，抽象成员不能时私有的。

抽象方法和虚方法有很多相似之处，常常可以互换。但是它们也有区别：

* 抽象方法只能在抽象类中声明，虚方法则不是。
* 抽象方法必须在派生类中重写，而虚方法则不必。
* 抽象方法不能声明方法实体，虚方法则可以。

下面的代码，使用抽象方法实现了与上例同样的功能：

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  /// <summary>  /// 抽象类car,拥有抽象方法  /// </summary>  abstract class Car  {  abstract public void Run();//抽象方法没有实现体 用分号代替    }  /// <summary>  /// 派生类carriage，拥有覆写方法Run  /// </summary>  class Carriage:Car  {  override public void Run()  {  Debug.Log("this is the running carriage");  }  }  /// <summary>  /// 派生类Bus，拥有覆写方法Run  /// </summary>  class Bus:Car  {  override public void Run()  {  Debug.Log("this is the running bus");  }  }  /// <summary>  /// 类person拥有driver方法，它可以调用car类中的run方法。  /// </summary>  class person  {  public void Driver(Car car)//形参的数据类型为Car，car参数持有car对象的引用。  {  car.Run();//调用run方法  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  person P = new person();  Bus B = new Bus();  P.Driver(B);  Carriage C = new Carriage();  P.Driver(C);  // Car c = new Car(); 会出现错误，因为 抽象类不能实例化  }  } |

### 抽象类特点

抽象类是不能被实例化的类（光说不做）

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace \_05\_抽象类计算圆\_正方形\_矩形的面积和周长  {  class Program  {  static void Main(string[] args)  {  Shape p = new Circle(10);  Console.WriteLine(p.ToString());//输出对象字符串  Console.WriteLine(p.GetArea());//输出面积  Console.WriteLine(p.GetGirth());//输出周长  Console.ReadKey();  }  }  public abstract class Shape  {  /// <summary>  /// 计算面积  /// </summary>  /// <returns></returns>  public abstract double GetArea();  /// <summary>  /// 计算周长  /// </summary>  /// <returns></returns>  public abstract double GetGirth();  }  public class Circle : Shape  {  public Circle(double radius)  {  this.Radius = radius;  }  public double Radius  {  get;  set;  }  public override double GetArea()  {  return Math.PI \* this.Radius \* this.Radius;  }  public override double GetGirth()  {  return 2 \* Math.PI \* this.Radius;  }  }  public class Square : Shape  {  public Square(double side)  {  this.SideLength = side;  }  public double SideLength  {  get;  set;  }  public override double GetArea()  {  return this.SideLength \* this.SideLength;  }  public override double GetGirth()  {  return this.SideLength \* 4;  }  }  public class Rectangle : Shape  {  public Rectangle(double h, double w)  {  this.Height = h;  this.Width = w;  }  public double Height  {  get;  set;  }  public double Width  {  get;  set;  }  public override double GetArea()  {  return this.Width \* this.Height;  }  public override double GetGirth()  {  return 2 \* (this.Width + this.Height);  }  }  } |

## 接口

### 什么是接口

接口就是一种规范、协议，约定好遵循某种规范就可以写出通用的代码。实现了接口的类就具有了规范中定义的能力。

以秦始皇统一六国后实行的“书同文、车同轨”为例。秦统一六国前，诸侯国各自为政，文字的形体极其紊乱。这给政令的推行和文化交流造成了严重障碍。进过整理、参照六国文字，最后创造出一种形体匀圆齐整、笔画简略的新文字，称为“秦篆”，又称“小篆”，作为官方规范文字，同时废除其他异体字。秦篆就是一个接口，只要采用了秦篆这个接口，不论身在何地，都能进行书面交流；战国时期，各诸侯国的道路车轨宽窄不一，严重影响交通往来，秦始皇统一中国后，修建驰道，宽50步，车轨宽6尺。只要生产的车遵循车轨宽6尺的规范，就都能在驰道上行驶。

总结起来，秦篆、车轨与车道的宽度等规范就时规范，而用秦篆书写的文字记录、遵从车轨宽度的马车就是规范的实现（接口的实现）。

—————————————————————————————————————

接口是仅包含一组方法声明，而没有具体的实现代码的引用类型。实现接口的类必须按照接口的定义实现这些方法，从而，实现同一个接口的类都具有这个接口的特征。

接口用来定义“行为规范”。而实现接口的派生类，需要完成接口定义的行为规范。比如，对于动物、人来说，它们都可以run。接口IRun中定义的是run的规范。可以这样说，接口IRun为不相关的类Animal和Human提供了通用功能run；Animal和Human随后实现了run这一功能。

|  |
| --- |
| using UnityEngine;  using System.Collections;  using System;  /// <summary>  /// 接口ICar  /// </summary>  interface IRun  {  void Run();//    }  /// <summary>  /// 实现接口  /// </summary>  class Animal:IRun  {  public void Run()  {  Debug.Log("this is the running Animal");  }  }  /// <summary>  /// 实现接口  /// </summary>  class Human:IRun  {  public void Run()  {  Debug.Log("this is the running Human");  }  }  /// <summary>  /// 类person拥有driver方法，它可以调用Icar类中的run方法。  /// </summary>  class Player  {  public void Driver(IRun iRun)//形参的数据类型为Car，car参数持有car对象的引用。  {  iRun.Run();//调用run方法  }  }  public class NewBehaviourScript : MonoBehaviour {  void Start () {  Player P = new Player();  Human B = new Human();  P.Driver(B);  Animal C = new Animal();  P.Driver(C);  }  } |

通过上面的例子，我们对接口的使用有了简单的了解。接下来我们来详细研究下如何声明与使用接口。

### 什么时候用接口

* 当多个类型不能抽象出合理的父类的时候，但是又要对某些方法进行多态，此时可以考虑使用接口来实现多态。即把公共的方法抽象到一个接口中，然后让不同的子类实现该接口。

|  |
| --- |
| public interface IFlyable  {  void Fly();  }  public class SpiderMan:IFlyable  {  public void Fly()  {  Console.WriteLine("蜘蛛侠在飞……");  }  }  public class Airplane:IFlyable  {  public void Fly()  {  Console.WriteLine("飞机在飞……");  }  }  public class bee:IFlyable  {  public void Fly()  {  Console.WriteLine("蜜蜂在飞……");  }  } |

蜘蛛侠、飞机、蜜蜂，都会飞，但是不能抽象出一个公共的父类。这里我们抽象出一个公共的规范（IFlyable）。实现了这个接口的类都具有规范所给予的能力“飞”。

* 接口可以多实现，解决类的单继承问题，当一个类需要同时继承多个类的行为时，可以考虑使用接口来进行多实现。

|  |
| --- |
| public interface IStopable  {  void Stop();  }  public interface IFlyable  {  void Fly();  }  public class SpiderMan:IFlyable,IStopable  {  public void Fly()  {  Console.WriteLine("蜘蛛侠在飞……");  }  public void Stop()  {  Console.WriteLine("蜘蛛侠停到楼顶");  }  } |

Spiderman类实现了IFlyable和IStopable两个接口，具有了这两个接口规范中给予的能力。

#### 抽象类与接口的区别

抽象类适用于同一系列，并且需要继承的成员

接口适用于不同系列具有相同的动作（行为，动作，方法）。

对于不是相同的系列，但具有相同的行为，这个考虑用接口。

接口解决了类不能多继承问题。

### 声明接口

接口声明可以有任何的访问修饰符。接口的成员只能是函数成员（包含静态函数成员），它不允许有任何访问修饰符（默认为public），而且不能包含任何实现代码。

下面是接口声明的语法，它包含了两个方法成员。

|  |
| --- |
| **接口名称**  **接口关键字**  **访问修饰符**  public interface IMyInterface  {  **分号代替了方法体**  int DoSomething(int val1, int val2);  void DoOtherthing();  } |

### 实现接口

只有类和结构可以实现接口。实现接口的类或结构必须满足以下条件:

* 在基类列表中包含接口名称。例如下面代码演示了MyClass的新声明，它实现了前面内容中声明的IMyInterface接口。注意，冒号后面的基类列表中需要有接口名称。

**接口名称**

**冒号**

|  |
| --- |
| class MyClass : IMyInterface  {  public int DoSomething(int val1, int val2)  {  **实现代码**  return val1 + val2;  }  public void DoOtherthing()  {  Debug.Log("this is implements of interface");  **实现代码**  }  } |

* 如果类实现了接口，它必须实现接口的所有成员。例如上面代码中实现了接口中的所有成员DoSomething和DoOtherthing。如果类不实现接口中成员，需要类为抽象类，成员需要添加abstract修饰符。

|  |
| --- |
| class abstract MyClass : IMyInterface //抽象类  {  public abstract int DoSomething(int val1, int val2)(); //方法没有实现    public void DoOtherthing();//方法没有实现  } |

* 如果类从基类继承并实现接口，基类列表中的基类名称必须放在任何接口之前，如下表所示。

|  |
| --- |
| class MyClass : BaseClass , IMyInterface, IEnumerator  **接口名称**  **基类必须放在最前面** |

#### 显式实现接口

实现接口的时候在方法前加接口名就叫做显式实现接口。

|  |
| --- |
| class Program  {  static void Main(string[] args)  {  I1 i1 = new MyClass();  i1.M1();//i1的类型为I1，通过接口名能访问到私有的I1.M1()。但是I2.M1()对i1来讲是访问不到的。因此这里访问的就是I1.M1()  I2 i2 = new MyClass();  i2.M1();  Console.ReadKey();  /\*----------------输出结果---------------------  \* 这是I1的显式实现  \* 这是I2的显式实现  --------------------------------------------\*/  }  }  public interface I1  {  void M1();  }  public interface I2  {  void M1();  }  public class MyClass:I1,I2  {  //显式实现接口时私有的，必须通过接口名才能访问的到。  void I1.M1()//添加接口名来显式实现接口  {  Console.WriteLine("这是I1的显式实现");  }  void I2.M1()  {  Console.WriteLine("这是I2的显式实现");  }  } |

在上表中，i1和i2调用同样的M1，但是返回了不一样的结果。解决了多个接口具有相同的方法名在调用时的歧义问题。

### 接口引用和as运算符

接口没有实例，接口的引用不能直接访问接口中的方法，然而，我们可以把类对象引用转换为接口类型的引用，就可以来调用接口中的方法了。

先来看看下面的代码。

|  |
| --- |
| using UnityEngine;  using System.Collections;  interface IBaseClass  {  void Method();    }  class DerivedClass:IBaseClass  {  public void Method()  {  Debug.Log("this is the DerivedClass");  }    }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  DerivedClass D = new DerivedClass();  //IBaseClass b1 = new IBaseClass();//不能创建接口的实例。接口的引用不能直接访问接口对象。  IBaseClass B = D;  B.Method();//调用Method方法，输出：this is the DerivedClass    }  } |

其中，Start方法中的语句就是从类对象引用转换为接口引用的示例：

第一行的D变量是一个实现了IBaseClass接口的类对象的引用。第二行中转换类对象DerivedClass的引用为接口IBaseClass的引用，并将它赋值给变量B。第三行使用指向接口的引用来调用实现的方法。

|  |
| --- |
| DerivedClass D = new DerivedClass();  IBaseClass B = D ;  B.Method(); |

除了使用类型转换来获取对象接口的使用，另一个更好的方式是使用as运算符。如果我们使用类型转换将类对象引用转换为类未实现的接口的引用，运行时会抛出一个异常。我们可以通过使用as运算符来避免这个问题。

如果类实现了接口，as所在的表达式返回的接口的引用；如果类没有实现接口，表达式返回null，而不是异常。

如下面的代码演示了as运算符的运用。第二行使用as运算符从类对象引用来获取接口引用。表达式的值会把B的值设置为null或 IBaseClass接口的引用；第三行代码检测了B的值，如果不是null，则执行命令来调用接口的成员。

|  |
| --- |
| DerivedClass D = new DerivedClass();  IBaseClass B = D as IBaseClass;  if(B != null)  {  B.Method();//调用Method方法，输出：this is the DerivedClass  } |

### 多个接口的实现与引用

类或结构可以实现任意数量的接口。所有要实现的接口必须列在基类列表中并以逗号隔开。

由于类可以实现任意数量的接口，有可能两个或多个接口成员都有相同的签名或返回类型。那么如何实现重复的接口成员呢？只需要实现单个成员来满足所有包含重复成员的接口。

在之前的内容中我们知道，通过类型转换可以把对象引用转换为接口类型的引用。如果类实现了多个接口，我们可以获取每一个独立的接口的引用。

例如，下面的类ImplementClass实现了两个接口：IInterface01和IInterface02。其中Method方法是这两个接口中重复的方法。在Start方法中，使用类型转换使使对象的引用分别转换为IInterface01的引用和IInterface02的引用。

|  |
| --- |
| using UnityEngine;  using System.Collections;  interface IInterface01 //声明接口1，有两个规范（方法）  {  void Method01();  void Method(string s);    }  interface IInterface02 //声明接口2，有两个规范（方法）  { ImplementClass  void Method02();  void Method(string str);  }  class ImplementClass:IInterface01, IInterface02  {  public void Method01()//实现接口1的方法  {  Debug.Log("this is the implements of IInterface01");  }  public void Method02()//实现接口2的方法  {  Debug.Log("this is the implements of IInterface02");  }  public void Method(string str)//实现接口1和接口2的重复方法  {  Debug.Log(str);  }    }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  ImplementClass I = new ImplementClass();  IInterface01 i1 = (IInterface01)I;//获取接口1的引用  IInterface02 i2 = (IInterface02)I;//获取接口2的引用  i1.Method01();//调用接口1的实现方法  i2.Method02();//调用接口2的实现方法  }  } |

### 接口成员的实现

接口的成员通常是在实现接口的类或结构中实现的。有时，我们也可以从实现接口的类的基类中继承实现的代码。

例如，在下面的代码中，IInterface接口的成员MPrint与BaseClass的成员MPrint相匹配。DerivedClass派生自BaseClass，而且在基类列表中包含IInterface。即使DerivedClass的声明主体是空的，它也靠基类中的MPrint方法实现了接口IInterface。

|  |
| --- |
| using UnityEngine;  using System.Collections;  interface IInterface  {  void MPrint();  }  class BaseClass  {  public void MPrint()  {  Debug.Log("this is a implement");  }  }  class DerivedClass : BaseClass, IInterface  {  }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  DerivedClass D = new DerivedClass();  IInterface I = (IInterface)D;  I.MPrint();//输出：this is a implement  }  } |

在上一节中，我们已经看到单个成员可以实现多个接口中重复的成员。但是，如果我们希望为每一个接口设值独立的实现方法时该怎么做呢？在这种情况下，我们可以创建显示接口成员实现。它使用限定接口名称（由接口名称和成员名称以及它们中间的点号分隔符构成）来声明，可以为不同的接口实现不同版本的方法。

显示实现的语法样式如下：

|  |
| --- |
| void IInterface01.Method()  {  } |

例如，下面的代码中为两个接口的重复成员Method声明了显示接口成员实现。

|  |
| --- |
| using UnityEngine;  using System.Collections;  interface IInterface01 //声明接口1，有两个规范（方法）  {  void Method();    }  interface IInterface02 //声明接口2，有两个规范（方法）  {  void Method();  }  class ImplementClass:IInterface01, IInterface02  {  void IInterface01.Method()//显示实现接口1的方法  {  Debug.Log("this is the implements of IInterface01");  }  void IInterface02.Method()//显示实现接口2的方法  {  Debug.Log("this is the implements of IInterface02");  }    }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  ImplementClass I = new ImplementClass();  IInterface01 i1 = (IInterface01)I;//获取接口1的引用  IInterface02 i2 = (IInterface02)I;//获取接口2的引用  i1.Method();//调用接口1的显示实现  i2.Method();//调用接口2的显示实现    }  } |

显示实现的接口成员只可以通过指向接口的引用来访问，也就是说，其他的类成员都不可以直接访问它们。

例如，下面的代码演示了类ImplementClass的声明，它使用显示实现实现了IInterface01接口。注意，即使是ImplementClass的另一个成员MPrint，也不可以直接访问显示实现。

|  |
| --- |
| using UnityEngine;  using System.Collections;  interface IInterface01 //声明接口1  {  void Method();    }  class ImplementClass:IInterface01  {  void IInterface01.Method()//显示实现接口1的方法  {  Debug.Log("this is the implements of IInterface01");  }  public void MPrint()  {  ((IInterface01)this).Method();//强制转换当前对象的引用（this）为接口类型的引用，并使用这个指向接口的引用调用显示接口实现  }  } |

### 接口可以继承接口

接口本身可以从一个或多个接口继承。要指定某个接口继承其他的接口，应在接口声明中把基接口以逗号分隔的列表形式放在接口名称的冒号之后。与类不同的是，类的基类列表中只能有一个类名，而接口可以在接口列表中有任意多个接口。

接口继承的声明语法如下:

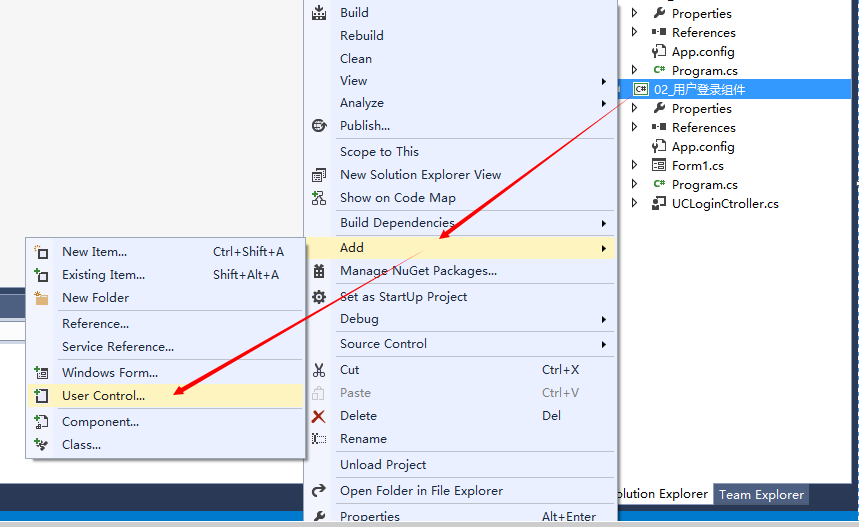
|  |
| --- |
| interface IDerivedInterface : IInterface01, IInterface02  {  }  **接口列表**  **冒号**  **接口名称** |

下面的代码演示了三个接口的声明，IDerivedInterface接口从前两个继承。

|  |
| --- |
| using UnityEngine;  using System.Collections;  interface IInterface01 //声明接口1  {  void SetMethod();    }  interface IInterface02 //声明接口1  {  void GetMethod();  }  /// <summary>  /// 即使接口的成员声明的主体是空的，它也有两个接口成员 SetMethod和GetMethod  /// </summary>  interface IDerivedInterface : IInterface01, IInterface02  {  }  class ImplementClass:IDerivedInterface  {  public void SetMethod()  {  Debug.Log("this is derived from IInterface01");  }  public void GetMethod()  {  Debug.Log("this is derived from IInterface02");  }    }  public class NewBehaviourScript1 : MonoBehaviour {  void Start ()  {  ImplementClass I = new ImplementClass();  IDerivedInterface D = (IDerivedInterface)I;//转换为IDerivedInterface接口引用  D.SetMethod();//调用接口的实现方法  D.GetMethod();//调用接口的实现方法  }  } |

### 自定义控件中通过接口进行验证

1. 添加User Control



1. 设计登录界面

3个label，2个textbox，一个button



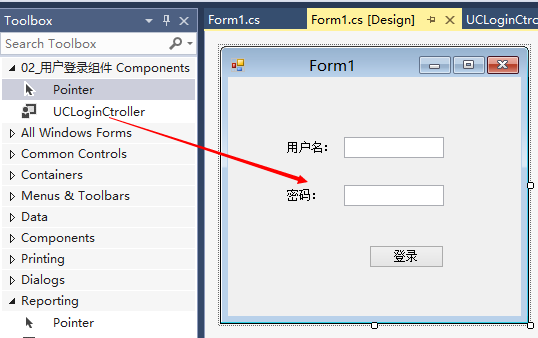
1. 设计一个接口ICheckValid，一个类LoginEventArgs。

|  |
| --- |
| //接口  public interface ICheckValid  {  void validatingUserLogin(object sender, LoginEventArgs e);  }  //事件需要的参数  public class LoginEventArgs  {  public string LoginUid { get; set; }  public string LoginPwd { get; set; }  public bool IsValid { get; set; }  } |

1. 在用户控件的类中对数据的采集以及判断后的行为进行处理。

|  |
| --- |
| public partial class UCUserLogin : UserControl  {  public UCUserLogin()  {  InitializeComponent();  }  //接口变量  public ICheckValid CheckValid;  private void button1\_Click(object sender, EventArgs e)  {  //1.采集数据  string uid = txt\_Uid.Text.Trim();  string pwd = txt\_Pwd.Text;  //2.判断接口变量是否为空  if (CheckValid != null)  {  LoginEventArgs evt = new LoginEventArgs();  evt.LoginUid = uid;  evt.LoginPwd = pwd;  //这里是多态。  CheckValid.validatingUserLogin(this, evt);  if (evt.IsValid)  {  this.BackColor = Color.Green;  }else  {  this.BackColor = Color.Gray;  }  }  else  {  this.BackColor = Color.Red;  }  } |

1. 把自定义控件添加到窗口中



1. 添加接口实现类，在其中对输入进行判断。

|  |
| --- |
| public class MyEventHandler : ICheckValid  {  //接口处理程序  public void validatingUserLogin(object sender, LoginEventArgs e)  {  if (e.LoginUid == "admin" && e.LoginPwd == "888")  {  e.IsValid = true;  }  else  {  e.IsValid = false;  }  }  } |

1. 窗口加载时，为接口变量赋值，，给的是接口的实现类。

|  |
| --- |
| private void Form1\_Load(object sender, EventArgs e)  {  //窗口加载时，为接口变量赋值，给的是子类  ucUserLogin1.CheckValid = new MyEventHandler();  } |

## 谈C#多态的魅力（虚方法，抽象，接口实现）

前言：我们都知道面向对象的三大特性：封装，继承，多态。封装和继承对于初学者而言比较好理解，但要理解多态，尤其是深入理解，初学者往往存在有很多困惑，为什么这样就可以？有时候感觉很不可思议，由此，面向对象的魅力体现了出来，那就是多态，多态用的好，可以提高程序的扩展性。常用的设计模式，比如简单工厂设计模式，核心就是多态。

其实多态就是：允许将子类类型的指针赋值给父类类型的指针。也就是同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。在运行时，可以通过指向基类的指针，来调用实现派生类中的方法。如果这边不理解可以先放一放，先看下面的事例，看完之后再来理解这句话，就很容易懂了。  
理解多态之前首先要对面向对象的里氏替换原则和开放封闭原则有所了解。

里氏替换原则（Liskov Substitution Principle）：派生类（子类）对象能够替换其基类（超类）对象被使用。通俗一点的理解就是“子类是父类”，举个例子，“男人是人，人不一定是男人”，当需要一个父类类型的对象的时候可以给一个子类类型的对象；当需要一个子类类型对象的时候给一个父类类型对象是不可以的！

开放封闭原则（Open Closed Principle）：封装变化、降低耦合，软件实体应该是可扩展，而不可修改的。也就是说，对扩展是开放的，而对修改是封闭的。因此，开放封闭原则主要体现在两个方面：对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改。

对这两个原则有一定了解之后就能更好的理解多态。

首先，我们先来看下怎样用虚方法实现多态。

我们都知道，喜鹊（Magpie）、老鹰（Eagle）、企鹅（Penguin）都是属于鸟类，我们可以根据这三者的共有特性提取出鸟类（Bird）做为父类，喜鹊喜欢吃虫子，老鹰喜欢吃肉，企鹅喜欢吃鱼。

创建基类Bird如下，添加一个虚方法Eat():

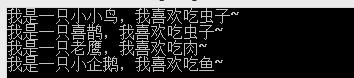
    /// <summary>  
    /// 鸟类：父类  
    /// </summary>  
    public class Bird  
    {  
        /// <summary>  
        /// 吃：虚方法  
        /// </summary>  
        public virtual void Eat()  
        {  
            Console.WriteLine("我是一只小小鸟，我喜欢吃虫子~");  
        }  
    }  
创建子类Magpie如下，继承父类Bird，重写父类Bird中的虚方法Eat()：

    /// <summary>  
    /// 喜鹊：子类  
    /// </summary>  
    public  class Magpie:Bird  
    {  
        /// <summary>  
        /// 重写父类中Eat方法  
        /// </summary>  
        public override void Eat()  
        {  
            Console.WriteLine("我是一只喜鹊，我喜欢吃虫子~");  
        }  
    }  
创建一个子类Eagle如下，继承父类Bird，重写父类Bird中的虚方法Eat()：

    /// <summary>  
    /// 老鹰：子类  
    /// </summary>  
    public  class Eagle:Bird  
    {  
        /// <summary>  
        /// 重写父类中Eat方法  
        /// </summary>  
        public override void Eat()  
        {  
            Console.WriteLine("我是一只老鹰，我喜欢吃肉~");  
        }  
    }  
创建一个子类Penguin如下，继承父类Bird，重写父类Bird中的虚方法Eat()：

    /// <summary>  
    /// 企鹅：子类  
    /// </summary>  
    public  class Penguin:Bird  
    {  
        /// <summary>  
        /// 重写父类中Eat方法  
        /// </summary>  
        public override void Eat()  
        {  
            Console.WriteLine("我是一只小企鹅，我喜欢吃鱼~");  
        }  
    }  
到此，一个基类，三个子类已经创建完毕，接下来我们在主函数中来看下多态是怎样体现的。

    static void Main(string[] args)  
    {  
        //创建一个Bird基类数组，添加基类Bird对象，Magpie对象，Eagle对象，Penguin对象  
        Bird[] birds = {  
                       new Bird(),  
                       new Magpie(),  
                       new Eagle(),  
                       new Penguin()  
        };  
        //遍历一下birds数组  
        foreach (Bird bird in birds)  
        {  
            bird.Eat();  
        }  
        Console.ReadKey();  
    }  
运行结果：



由此可见，子类Magpie，Eagle，Penguin对象可以赋值给父类对象，也就是说父类类型指针可以指向子类类型对象，这里体现了里氏替换原则。

父类对象调用自己的Eat()方法，实际上显示的是父类类型指针指向的子类类型对象重写父类Eat后的方法。这就是多态。

多态的作用到底是什么呢？  
其实多态的作用就是把不同的子类对象都当作父类来看，可以屏蔽不同子类对象之间的差异，写出通用的代码，做出通用的[编程](http://www.2cto.com/kf)，以适应需求的不断变化。  
以上程序也体现了开放封闭原则，如果后面的同事需要扩展我这个程序，还想再添加一个猫头鹰（Owl），很容易，只需要添加一个Owl类文件，继承Bird，重写Eat()方法，添加给父类对象就可以了。至此，该程序的扩展性得到了提升，而又不需要查看源代码是如何实现的就可以扩展新功能。这就是多态带来的好处。

我们再来看下利用抽象如何来实现多态。

还是刚才的例子，我们发现Bird这个父类，我们根本不需要使用它创建的对象，它存在的意义就是供子类来继承。所以我们可以用抽象类来优化它。  
我们把Bird父类改成抽象类，Eat()方法改成抽象方法。代码如下：

    /// <summary>  
    /// 鸟类：基类  
    /// </summary>  
    public abstract class Bird  
    {  
        /// <summary>  
        /// 吃：抽象方法  
        /// </summary>  
        public abstract void Eat();  
    }  
抽象类Bird内添加一个Eat()抽象方法，没有方法体。也不能实例化。  
其他类Magpie，Eagle，Penguin代码不变，子类也是用override关键字来重写父类中抽象方法。  
Main主函数中Bird就不能创建对象了，代码稍微修改如下：

        static void Main(string[] args)  
        {  
            //创建一个Bird基类数组，添加 Magpie对象，Eagle对象，Penguin对象  
            Bird[] birds = {  
                           new Magpie(),  
                           new Eagle(),  
                           new Penguin()  
            };  
            //遍历一下birds数组  
            foreach (Bird bird in birds)  
            {  
                bird.Eat();  
            }  
            Console.ReadKey();  
        }  
执行结果：

\

由此可见，我们选择使用虚方法实现多态还是抽象类抽象方法实现多态，取决于我们是否需要使用基类实例化的对象.

比如说 现在有一个Employee类作为基类，ProjectManager类继承自Employee，这个时候我们就需要使用虚方法来实现多态了，因为我们要使用Employee创建的对象，这些对象就是普通员工对象。  
再比如说 现在有一个Person类作为基类，Student，Teacher 类继承Person，我们需要使用的是Student和Teacher创建的对象，根本不需要使用Person创建的对象，  
所以在这里Person完全可以写成抽象类。

总而言之，是使用虚方法，或者抽象类抽象方法实现多态，视情况而定，什么情况？以上我说的两点~

接下来~~~~

我要问一个问题，喜鹊和老鹰都可以飞，这个飞的能力，我怎么来实现呢？

XXX答：“在父类Bird中添加一个Fly方法不就好了~~”

我再问：“好的，照你说的，企鹅继承父类Bird，但是不能企鹅不能飞啊，这样在父类Bird中添加Fly方法是不是不合适呢？”

XXX答：“那就在能飞的鸟类中分别添加Fly方法不就可以了吗？”

对，这样是可以，功能完全可以实现，可是这样违背了面向对象开放封闭原则，下次我要再扩展一个鸟类比如猫头鹰（Owl），我还要去源代码中看下Fly是怎么实现的，然后在Owl中再次添加Fly方法，相同的功能，重复的代码，这样是不合理的，程序也不便于扩展；

其次，如果我还要添加一个飞机类（Plane），我继承Bird父类，合适吗？

很显然，不合适！所以我们需要一种规则，那就是接口了，喜鹊，老鹰，飞机，我都实现这个接口，那就可以飞了，而企鹅我不实现这个接口，它就不能飞~~

好，接下来介绍一下接口如何实现多态~

添加一个接口IFlyable，代码如下：

    /// <summary>  
    /// 飞 接口  
    /// </summary>  
    public interface IFlyable  
    {  
        void Fly();  
    }  
喜鹊Magpie实现IFlyable接口，代码如下：

    /// <summary>  
    /// 喜鹊：子类，实现IFlyable接口  
    /// </summary>  
    public  class Magpie:Bird,IFlyable  
    {  
        /// <summary>  
        /// 重写父类Bird中Eat方法  
        /// </summary>  
        public override void Eat()  
        {  
            Console.WriteLine("我是一只喜鹊，我喜欢吃虫子~");  
        }  
        /// <summary>  
        /// 实现 IFlyable接口方法  
        /// </summary>  
        public void Fly()  
        {  
            Console.WriteLine("我是一只喜鹊，我可以飞哦~~");  
        }  
    }  
老鹰Eagle实现IFlyable接口，代码如下：

    /// <summary>  
    /// 老鹰：子类实现飞接口  
    /// </summary>  
    public  class Eagle:Bird,IFlyable  
    {  
        /// <summary>  
        /// 重写父类Bird中Eat方法  
        /// </summary>  
        public override void Eat()  
        {  
            Console.WriteLine("我是一只老鹰，我喜欢吃肉~");  
        }

        /// <summary>  
        /// 实现 IFlyable接口方法  
        /// </summary>  
        public void Fly()  
        {  
            Console.WriteLine("我是一只老鹰，我可以飞哦~~");  
        }  
    }  
在Main主函数中，创建一个IFlyable接口数组，代码实现如下：

    static void Main(string[] args)  
    {  
        //创建一个IFlyable接口数组，添加 Magpie对象，Eagle对象  
        IFlyable[] flys = {  
                       new Magpie(),  
                       new Eagle()  
        };  
        //遍历一下flys数组  
        foreach (IFlyable fly in flys)  
        {  
            fly.Fly();  
        }  
        Console.ReadKey();  
    }  
执行结果：

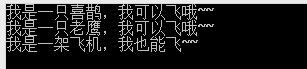
\

由于企鹅Penguin没有实现IFlyable接口，所以企鹅不能对象不能赋值给IFlyable接口对象，所以企鹅，不能飞~

好了，刚才我提到了飞机也能飞，继承Bird不合适的问题，现在有了接口，这个问题也可以解决了。如下，我添加一个飞机Plane类，实现IFlyable接口，代码如下：

    /// <summary>  
    /// 飞机类，实现IFlyable接口  
    /// </summary>  
    public  class Plane:IFlyable  
    {  
        /// <summary>  
        /// 实现接口方法  
        /// </summary>  
        public void Fly()  
        {  
            Console.WriteLine("我是一架飞机，我也能飞~~");  
        }  
    }  
在Main主函数中，接口IFlyable数组，添加Plane对象：

    class Program  
    {  
        static void Main(string[] args)  
        {  
            //创建一个IFlyable接口数组，添加 Magpie对象，Eagle对象，Plane对象  
            IFlyable[] flys = {  
                           new Magpie(),  
                           new Eagle(),  
                           new Plane()  
            };  
            //遍历一下flys数组  
            foreach (IFlyable fly in flys)  
            {  
                fly.Fly();  
            }  
            Console.ReadKey();  
        }  
    }  
执行结果：



由此，可以看出用接口实现多态程序的扩展性得到了大大提升，以后不管是再扩展一个蝴蝶（Butterfly），还是鸟人（Birder）创建一个类，实现这个接口，在主函数中添加该对象就可以了。  
也不需要查看源代码是如何实现的，体现了开放封闭原则！

接口充分体现了多态的魅力~~

以上通过一些小的事例，给大家介绍了面向对象中三种实现多态的方式，或许有人会问，在项目中怎么使用多态呢？多态的魅力在项目中如何体现？  
那么接下来我做一个面向对象的简单计算器，来Show一下多态在项目中使用吧！

加减乘除运算，我们可以根据共性提取出一个计算类，里面包含两个属性 Number1和Number2，还有一个抽象方法Compute();代码如下：

    /// <summary>  
    /// 计算父类  
    /// </summary>  
    public abstract class Calculate  
    {  
        public int Number1  
        {  
            get;  
            set;  
        }  
        public int Number2  
        {  
            get;  
            set;  
        }  
        public abstract int Compute();  
    }  
接下来，我们添加一个加法器，继承计算Calculate父类：

    /// <summary>  
    /// 加法器  
    /// </summary>  
    public class Addition : Calculate  
    {  
        /// <summary>  
        /// 实现父类计算方法  
        /// </summary>  
        /// <returns>加法计算结果</returns>  
        public override int Compute()  
        {  
            return Number1 + Number2;  
        }  
    }  
再添加一个减法器，继承计算Calculate父类：

    /// <summary>  
    /// 减法器  
    /// </summary>  
    public class Subtraction : Calculate  
    {  
        /// <summary>  
        /// 实现父类计算方法  
        /// </summary>  
        /// <returns>减法计算结果</returns>  
        public override int Compute()  
        {  
            return Number1 - Number2;  
        }  
    }  
在主窗体FormMain中，编写计算事件btn\_Compute\_Click，代码如下：

    private void btn\_Compute\_Click(object sender, EventArgs e)  
    {  
        //获取两个参数  
        int number1 = Convert.ToInt32(this.txt\_Number1.Text.Trim());  
        int number2 = Convert.ToInt32(this.txt\_Number2.Text.Trim());  
        //获取运算符  
        string operation = cbb\_Operator.Text.Trim();  
        //通过运算符，返回父类类型  
        Calculate calculate = GetCalculateResult(operation);  
        calculate.Number1 = number1;  
        calculate.Number2 = number2;  
        //利用多态，返回运算结果  
        string result = calculate.Compute().ToString();  
        this.lab\_Result.Text = result;  
    }  
    /// <summary>  
    /// 通过运算符，返回父类类型  
    /// </summary>  
    /// <param name="operation"></param>  
    /// <returns></returns>  
    private Calculate GetCalculateResult(string operation)  
    {  
        Calculate calculate = null;  
        switch (operation)  
        {  
            case "+":  
                calculate = new Addition();  
                break;  
            case "-":  
                calculate = new Subtraction();  
                break;  
        }  
        return calculate;  
    }  
在该事件中主要调用GetCalculateResult方法，通过运算符，创建一个对应的加减乘除计算器子类，然后赋值给父类，其实这就是设计模式中的简单工厂设计模式，我给你一个运算符你给我生产一个对应的加减乘除计算器子类，返回给我。。其实大多数的设计模式的核心就是多态，掌握好多态，设计模式看起来也很轻松。

现阶段工作已经完成，但是过了一段时间，又添加新的需求了，我还要扩展一个乘法了，那好，很简单只要创建一个乘法计算器继承Calculate父类即可，看代码：

    /// <summary>  
    /// 乘法计算器  
    /// </summary>  
    public  class Multiplication:Calculate  
    {  
        public override int Compute()  
        {  
            return Number1\*Number2;  
        }  
    }  
然后在GetCalculateResult函数中添加一个case 就好了：

    switch (operation)  
    {  
        case "+":  
            calculate = new Addition();  
            break;  
        case "-":  
            calculate = new Subtraction();  
            break;  
        case "\*":  
            calculate = new Multiplication();  
            break;  
    }  
执行结果：

## 多态的意义与目的

多态的意义：程序的可扩展性。

多态的最终目的：节省成本、提高效率。

## 面向对象的编程的使用建议

在编程时，要面向抽象进行编程，使用抽象（父类，抽象类，接口），而不使用具体。原因请参考1.2.4依赖倒置原则 (the Dependency Inversion Principle DIP)

在定义方法参数、返回值、声明变量的时候能用抽象就不用具体。能使用接口就不使用抽象，能用抽象就不用类，能用父类就不用子类。原因参考1.2.3里氏替换原则 (the Liskov Substitution Principle LSP)以及1.2.2开闭原则（the Open Closed Principle OCP）。

避免使用“体积庞大的接口”、”多功能接口“，会造成”接口污染“（因为实现借口需要实现所有的接口中定义的方法，多余的方法造成污染）。只把相关联的一组成员定义到一个接口中。这是单一职责原则应用在接口上，叫做接口隔离原则（（the Interface Segregation Principle ISP））。

当有多种功能时，可以定义多个职责单一的接口（组合使用）。就好像印刷术与活字印刷术一样，具有可扩展性。符合1.2.1单一职责原则（Single Responsibility Principle）。

# 匿名类型

匿名类型是方便程序员编程用的，编译器遍编译时会自动加上类名，并且自动生成只读的字段和属性。

在只使用一次的情况下，可以用匿名类型来加快开发速度。

|  |
| --- |
| //匿名类型  var car = new {Brand="布加迪威龙",Speed="350",Price="1300万"};  Console.WriteLine(car.Brand);//访问你们类型的属性。是只读的  Console.WriteLine(car.Price);  Console.WriteLine(car.Speed);  Console.ReadKey(); |

# 浅拷贝和深拷贝

拷贝：将对象在内存中重新创建一份。

创建2个类，代码如下：

|  |
| --- |
| class Person  {  private string \_name;  public string Name  {  get { return \_name; }  set { \_name = value; }  }  private string \_gender;  public string Gender  {  get { return \_gender; }  set { \_gender = value; }  }  private int \_age;  public int Age  {  get { return \_age; }  set { \_age = value; }  }  public Car Mycar;  }  class Car  {  private string \_brand;  public string Brand  {  get { return \_brand; }  set { \_brand = value; }  }  } |

之后在入口函数进行如下操作：

|  |
| --- |
| static void Main(string[] args)  {  Person p1 = new Person();  Person p2 = p1;//没有发生拷贝  } |

在p1赋值为p2时，并没有发生拷贝，因为p1、p2指向的是同一个对象。

## **浅拷贝**

当针对一个对象浅拷贝的时候，对于对象的值类型成员，会复制其本身，对于对象的引用类型成员，仅仅复制对象引用，这个引用指向托管堆上的对象实例。

在Person类中添加一个浅拷贝的方法

|  |
| --- |
| public Object clone()  {  return MemberwiseClone();  } |

在主函数中的代码如下:

|  |
| --- |
| Person p1 = new Person();  p1.Age = 12;  p1.Mycar = new Car();  p1.Mycar.Brand = "audi";  Person p2 = (Person)p1.clone();//浅拷贝  p2.Age = 14;//修改值类型  p2.Mycar.Brand = "QQ";//修改引用类型  Console.WriteLine(p1.Age);//12  Console.WriteLine(p1.Mycar.Brand);//QQ.引用类型  Console.ReadLine(); |

调用clone方法进行拷贝时，值类型成员，会复制其本身，对于对象的引用类型成员，仅仅复制对象引用

所以，浅拷贝关键点是对引用类型拷贝的是对象引用，这个引用指向托管堆上的对象实例。改变原对应引用类型的值，会影响到复制对象。

## 深拷贝

对引用成员指向的对象也进行复制，在托管堆上赋值原先对象实例所包含的数据，再在托管堆上创建新的对象实例。

在Person类中添加一个深拷贝的方法

|  |
| --- |
| public Object clone1()  {  Person p = new Person();  p.Name = this.Name;  p.Age = this.Age;  p.Gender = this.Gender;  p.Mycar = new Car();  p.Mycar.Brand = this.Mycar.Brand;  return p;  } |

在主函数中的代码如下:

|  |
| --- |
| Person p = new Person();  p.Mycar = new Car();  p.Mycar.Brand = "audi";  Person p1 = (Person)p.clone1();  p1.Mycar.Brand = "QQ";  Console.WriteLine(p.Mycar.Brand);//还是原来的值：audi  Console.ReadLine(); |

深拷贝后，两个对象的引用成员已经分离，改变原先对象引用类型成员的值并不会对复制对象的引用类型成员值造成影响。

### 序列化和深拷贝

当类的成员比较多时，一个个赋值速度会很忙，这时可以用序列化来快速实现深拷贝。

序列化需要实现序列化特性：

|  |
| --- |
| [Serializable]  class Person  {  private string \_name;  public string Name  {  get { return \_name; }  set { \_name = value; }  }  private string \_gender;  public string Gender  {  get { return \_gender; }  set { \_gender = value; }  }  private int \_age;  public int Age  {  get { return \_age; }  set { \_age = value; }  }  public Car Mycar;  }  [Serializable]  class Car  {  private string \_brand;  public string Brand  {  get { return \_brand; }  set { \_brand = value; }  }  }  } |

在Person类中添加如下代码：

|  |
| --- |
| /\*  \* //使用序列化和反序列化进行复制  public object Clone1()  {  BinaryFormatter bf = new BinaryFormatter();  MemoryStream ms = new MemoryStream();  bf.Serialize(ms, this); //复制到流中  ms.Position = 0;  return (bf.Deserialize(ms));  }  \*/  public object SerializeClone()  {  //创建二进制序列器  BinaryFormatter bf = new BinaryFormatter();  //创建内存缓冲区  byte[] bytes = new byte[2000];  //创建内存流，内存流写入区为缓冲区  using (MemoryStream ms = new MemoryStream(bytes))  { //通过流，序列化当前类到内存缓冲区。  bf.Serialize(ms, this);  }  object o;  using (MemoryStream ms = new MemoryStream(bytes))  { //通过流，反序列化  o = bf.Deserialize(ms) as object;  }  return o;  } |

在主函数中调用SerializeClone进行深拷贝

|  |
| --- |
| Person p1 = new Person() { Age = 19, Name = "hao", Gender = "male" };  p1.Mycar = new Car();  p1.Mycar.Brand = "audi";  Person p2 = (Person)p1.SerializeClone();  p2.Mycar.Brand = "QQ";  Console.WriteLine(p1.Mycar.Brand);//audi  Console.ReadLine(); |

# 类型操作

## GetType方法获取类的类型

|  |
| --- |
| Person p = new Student();  string t = p.GetType().ToString();//输出p的类型  string tp = p.GetType().BaseType.ToString();//输出p的父亲的类型  string tpp = p.GetType().BaseType.BaseType.ToString();//输出p的父亲的父亲类型  string tppp = p.GetType().BaseType.BaseType.BaseType.ToString();//输出p的父亲的父亲的父亲类型  Console.WriteLine(t);  Console.WriteLine(tp);  Console.WriteLine(tpp);  Console.WriteLine();  Console.ReadLine(); |

## 判断向下转换是否成功

向下转换通是不安全的，是有条件限制的，当我们尝试引用在内存中实际不存在的类成员时，编译器不会报异常，但是在系统运行时遇到它们会抛出一个异常。可以用is和as来判断是否转换成功，转换成功再执行代码，这样就不会引发异常。

|  |  |
| --- | --- |
| is | as |
| Person p = new Student();  if (p is Student)  {  Student s = (Student)p;  }  else  {  Console.WriteLine("类型转换失败");  } | Person p = new Student();  Student s = (Student)p as Student;  if(s != null)  {  Console.WriteLine("ok");  } |

在上面的代码中，推荐使用as运算符。因为as只需要一次类型转换，而is需要两次类型转换。

## 判断是否是同一个对象

判断是否是同一个对象有三种方式：

* Equals
* ==
* object.ReferenceEquals

我们先来研究前两种方法：

|  |  |
| --- | --- |
| public virtual bool **Equals**(object obj)  {  return RuntimeHelpers.Equals(this, obj);  } | public static bool **operator** **==** (string a, string b)  {  return Equals(a, b);  } |

从上面可以发现，“==”进行判断实际上就是用Equals进行判断。而Equals方法是个虚方法，它判断对象是否相同的依据是“是否指向同一个对象”。以下面的代码为例：

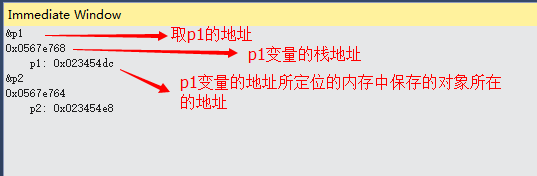
|  |
| --- |
| class Program  {  class Person  {  public int age;  }  static void Main(string[] args)  {  Person p1 = new Person();  p1.age = 18;  Person p2 = new Person();  p2.age = 20;  bool e1 = p1.Equals(p2);  bool e2 = (p1 == p2);    Console.WriteLine(e1); //输出为false  Console.WriteLine(e2); //输出为false  Console.ReadKey();  }      } |

用Equals和“=”进行判断的结果都是false。因为p1和p2指向的堆并不是同一块地址。p1和p2指向的地址，可以用vs的immediate window（即时窗口）进行调试。

首先，打个断点，开始调试:



然后在即时窗口中输入： &p1， &p2查看这两个变量中保存的对象在堆中的地址。

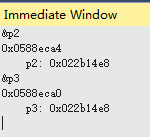


从上图中，可以发现，p1和p2指向的并不是同一个对象，所以返回值为false。

如果对以下的2个对象进行判断，返回值应该是什么？

|  |
| --- |
| Person p2 = new Person();  p2.age = 20;  Person p3 = p2;  bool e3 = p2.Equals(p3);  Console.WriteLine(e3);//输出为True |

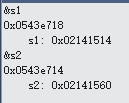
我们再用即时窗口调试，就知道为什么会true了“



接下来我们来看下特殊的对象：字符串。

|  |
| --- |
| string s1 = new string(new char[]{'h','a','o'});  string s2 = new string(new char[] { 'h', 'a', 'o' }); Console.WriteLine(s1.Equals(s2)); |

上面代码输出的结果是什么呢？ 我们用immediate window查看一下：



s1和s2指向的并不是同一个对象，那么返回值应该时flase喽!，但是实际上返回的却是true，这是为什么呢？

因为在开头我们研究Equals方法时就说过，在object中定义的Equals是个虚方法，在string类型中对其进行了重写，判断两个对象是否相等，不再是判断是否同一块内存了，而是判断2个对象中的值是否相同。

我们也来重写一下类，使本来要返回false的地方返回值为true.

|  |
| --- |
| class Program  {  class Person  {  public int age;  /// <summary>  /// 重写Equals方法  /// </summary>  /// <param name="obj"></param>  /// <returns></returns>  public override bool Equals(object obj)  {  Person p = obj as Person;  if(p!=null)  {  if(p.age==this.age)  {  return true;  }else  {  return false;  }  }else  {  return false;  }    }  }  static void Main(string[] args)  {  Person p1 = new Person();  p1.age = 19;  Person p2 = new Person();  p2.age = 19;  Console.WriteLine(p1.Equals(p2));//即时不是一个对象，返回值却为true    Console.ReadKey();  }      } |

所以用Equals并不是一定安全与正确的，用静态类object.ReferenceEquals来判断两个对象是否同一个对象，却能保证百分百正确，因为它不支持重写，永远判断的是两者是否指向同一个对象。

|  |
| --- |
| string s1 = new string(new char[] { 'h', 'a', 'o' });  string s2 = new string(new char[] { 'h', 'a', 'o' });  Console.WriteLine(object.ReferenceEquals(s1,s2)); //返回值为false |

1. 注意：这里的对外接口不是OOP编程中的接口类型。这里的对外接口表示类公开的成员（字段、方法、属性等） [↑](#footnote-ref-1)