**C#语言基础**

在开始之前我必须声明，该文档为总结性内容，对于某些概念不同的书籍会有不同的解释，尤其翻译过来的书籍，因翻译者的知识储备和对相关概念的理解有稍微不同的描述方式，但本文保证基础语法和对固定概念的描述是正确且严谨的。

1. **.NET Framework**

简单称为底层框架或软件工作平台，所有使用C#语言编写的软件都运行在该框架之上，没有该框架，C#语言就等于没有发动机的壳子，无法运转。（该段为认知描述）

.NET是一个封装了很多代码的代码库。

1. **C#中的数据类型及变量**

C#为强类型语言，即数据类型被强制声明为指定的类型。如：int i=1;

类型分为值类型和引用类型

|  |  |
| --- | --- |
| 值类型 | 引用类型 |
| byte sbyte  short ushort  int uint long ulog  bool  struct enum  double float  decimal | string  class  interface  dynamic  delegate |

值类型*的内容存储在堆栈上分配的内存中，值类型在声明后就在内存中分配了内容空间，如int x=42;值42存储在称为“栈”的内存区域中。*

引用类型（如：类和数组的实例）在另一个称为“堆”的内存区域中分配。下面的示例中，构成数组的所需的10个内存空间实在堆上分配的

int[] number = new int[10];

**补充：**

ref/out：应用于值类型

1. **流程控制**

条件语句

if语句

if (true)

{

Console.WriteLine("恒被执行");

}

if …else …语句

if (true)

{

Console.WriteLine("恒被执行");

}

else

{

Console.WriteLine("恒不执行");

}

if … else if … else 语句

public void IfControl(int inputNumber)

{

int i = 10;

if (i == inputNumber)

{

Console.WriteLine(i);

}

else if (i < inputNumber)

{

Console.WriteLine(inputNumber);

}

else

{

Console.WriteLine("i>inputNumber");

}

}

Switch语句

public int SwitchControl(int value)

{

int res = -1;

switch (value)

{

case 1:

res = 1;

break;

case 2:

res = 2;

break;

case 3:

res = 3;

break;

default:

break;

}

return res;

}

循环语句

循环语句多用于遍历集合或数组元素

For循环

public void ForLoopControl()

{

int[] array = new int[] { 1, 2, 3, 4, 5, 6 };

for (int i = 0; i < array.Length; i++)

{

Console.WriteLine(i);

}

}

do … while 循环

public void DoLoopControl()

{

int i = 0;

do

{

++i;

} while (i > 10);//当i>10的时候跳出循环

}

while循环

public void WhileLoopControl()

{

int i = 10;

while (i >= 10)

{

if (i == 10)

++i; //在这里会继续执行循环

else

i = i - 2;//在这里不再执行循环

}

}

foreach循环

public void ForeachLoopControl()

{

ArrayList arrayList = new ArrayList();

arrayList.Add(1);

arrayList.Add(2);

arrayList.Add("one");

arrayList.Add("two");

foreach (var item in arrayList)

{

Console.WriteLine(item);

}

}

1. **类型转换**

不同类型变量使用不同的模式来表示数据，即：即使可以把一系列的位从一种类型的变量转移到另外一种类型的变量中，结果也可能与期望的不同，因此就需要对数据进行类型转换，而不是将数据位从一个变量一对一映射到另一个变量。

类型转换分：隐式转化、显示转换

|  |  |
| --- | --- |
| 类型 | 可以安全的隐式转换为 |
| byte | short,ushort,int,uint,long,  ulong,float,double,decimal |
| short | int,long,float,double,decimal |
| int | long,float,double,decimal |
| long | float,double,decimal |
| float | double |
| char | ushort,int,uint,long,ulong,float,  double,decimal |

1. **函数（类及方法）**

函数：即必须声明在类中的方法块，在面向对象的程序设计中可以称为对象所具有的功能。

1. 函数分类

不接收任何参数也不返回任何数据的简单函数

public void FuncNoValue()

{

//功能代码块

}

有参数 无返回值的函数/方法

/// <summary>

/// 有参数 无返回值的函数/方法

/// </summary>

/// <param name="i"></param>

public void FuncHasParameter(int i)

{

//方法实现

}

有多个参数 无返回值的函数/方法

/// <summary>

/// 有多个参数 无返回值的函数/方法

/// 因与上面的方法重名，故也被称为方法的重载

/// </summary>

/// <param name="i"></param>

/// <param name="str"></param>

public void FuncHasParameter(int i, string str)

{

//code

}

无参数 有返回值的函数/方法

/// <summary>

/// 无参数 有返回值的函数/方法

/// </summary>

/// <returns>返回值类型为 int 类型</returns>

public int FuncHasReturnValue()

{

return 100;

}

有参数 有返回值的函数/方法

/// <summary>

/// 有参数 有返回值的函数/方法

/// </summary>

/// <param name="i"></param>

/// <returns></returns>

public string FuncHasParamAndReValue(int i)

{

return i.ToString();

}

委托(暂时初学者暂时跳过该函数)

委托即将一个方法作为另外一个方法的签名即参数传递给该方法

1. **异常处理**

1、try…catch

/// <summary>

/// 该方法演示使用try...catch处理异常代码块

/// </summary>

public void ResolveException()

{

string str = "exception";

try

{

//非数字字符串在转换成int类型时会发生异常，

//因此使用try...catch抓取该异常

//这样不会导致程序中断或退出（死掉）

int i = Convert.ToInt32(str);

}

catch (Exception ex)

{

Console.WriteLine("ResolveException函数发生异常：{0}", ex.Message);

}

}

2、try…catch…finally

public int ResolveException(int i)

{

string str = "exception";

try

{

//非数字字符串在转换成int类型时会发生异常，

//因此使用try...catch抓取该异常

//这样不会导致程序中断或退出（死掉）

i = Convert.ToInt32(str);

}

catch (Exception ex)

{

Console.WriteLine("ResolveException函数发生异常：{0}", ex.Message);

}

finally

{

i = 100;//finally中的代码始终会被执行

}

return i;

}

1. **面向对象编程OOP（Object-oriented programming）**
2. 以万物皆对象的概念来理解OOP（如：电脑，手机，手机上的每个App）
3. OOP技术以结构、数据的含义以及数据和数据之间的交互操作为基础。
4. OOP可以方便的拆卸软件的各个功能组件
5. OOP三个主要的特性：封装、继承、多态

下面举个例子，说明对象是什么：

电脑可以看作一个对象，电脑有CPU、内存条、磁盘，你可以把它们看作电脑所具有的一些属性，即CPU有单核和双核，内存条、磁盘都有大小，单核双核就是CPU的特性。

下面用代码演示电脑所具有的这些特性及它所具有的显示换面的功能：

//面向对象解释

public class OOP

{

//电脑

public class Computer

{

//CPU

public int Cpu { get; set; }

//内存

public int memory { get; set; }

//磁盘

public int Disk { get; set; }

/// <summary>

/// 电脑有显示的功能

/// </summary>

/// <param name="i"></param>

/// <returns></returns>

public int DisplayDevice(int i)

{

return i;

}

}

}

实例化电脑对象

/// <summary>

/// 创建对象

/// </summary>

public class InstantiationClass

{

//使用电脑前，需要先将电脑实例化，及创建一个电脑对象

//这样才能让电脑具有不同的配置和性能

public void GetComputer()

{

Computer thinkPadT640s = new Computer();

thinkPadT640s.Cpu = 8;//8核

thinkPadT640s.memory = 8;//8G

thinkPadT640s.Disk = 256;//256G

thinkPadT640s.DisplayDevice(123456);//键盘录入123456，显示器显示123456，这是电脑所具有的功能

}

}

* 1. 对象的生命周期

每个对象都有一个明确定义的生命周期，除了正在使用的“正常对象”外，还有构造阶段和析构阶段

* 1. 6

1. **定义类**

类的概念可以参看OOP及示例和备注

类成员包含：字段、属性、方法、委托、事件

1. 访问修饰符

Public：公开被其修饰的类或其成员

Private：只有类自己可访问，由其派生类的也不可访问

Internal：程序集内可访问

Protected：受保护的访问形式，只有其自身和其派生类可访问

1. 创建类及其成员
2. 静态类和静态成员

1、静态类不能够实例化，其内部不能声明实例成员，所以内部的成员也必须是静态的

1. 回顾OOP
   * 接口

* 接口定义了一个契约，类必须履行这个契约，才能获得接口提供的能力。
* 接口中不能包含字段（也就是数据），如果一个接口要求派生类包含特定的数据，那么它会使用属性而不是字段。
* 每个接口成员都具有与抽象方法相似的行为，从而强迫派生类实现成员，所以，不可以为接口成员显式地使用abstract修饰符。然而成员可以有两种不同的实现：显式和隐式。
* 版本控制：假定一个组件或者应用程序是提供给其他开发者在编程时使用的，那么在创建它的一个新版时，你不应该改动接口。因为接口在负责实现的类和使用接口的类之间定义了一个契约，改动接口相当于改动了契约，这会使基于接口写的代码失效

|  |  |
| --- | --- |
| 抽 象 类 | 接 口 |
| 不能脱离它的派生类来实例化。抽象类构造器只能由它的派生类调用 | 不能实例化 |
| 定义了基类必须实现的抽象成员签名 | 接口的所有成员要在基类中实现。负责实现的类（实现类）不能只实现部分成员 |
| 扩展性比接口好，不会破坏任何版本兼容性。在抽象类中，你可以添加附加的非抽象成员，它们可由所有派生类继承 | 用更多的成员来扩展接口，会破坏版本兼容性 |
| 可以包含存储在字段中的数据 | 不能存储任何数据，只能在派生类中指定字段。解决这个问题的办法是在接口中定义属性，但不能包括实现 |
| 允许包含实现的（virtual）成员，所以能为派生类的成员提供一个默认实现 | 所有成员都自动成为virtual成员，而且不能包含任何实现 |
| 从一个抽象类派生，会用掉子类唯一的一个基类型选项（因为C#只允许单一继承） | 虽然不允许存在默认实现，但实现接口的类可以继续相互派生 |

* 抽象类

namespace ConsoleApp

{

using System;

/// <summary>

/// 抽象类型 与接口相似

/// </summary>

public abstract class AbstractType

{

public AbstractType(string s) { }//这样空的构造器就被抹掉了

private string tag;//字段不能修饰为abstract

public abstract string Tag { get; set; }

public void FuncA() { }

public abstract void FuncB();//抽象方法无实现\*\*\*

public virtual void FuncC() { }//虚方法必须有实现

}

public class ImplementAbstractType : AbstractType

{

//如果AbstractType中没有定义有参的构造，默认是不现实实现抽象类构造的

public ImplementAbstractType(string s) : base(s)

{

}

public override string Tag

{

get { throw new NotImplementedException(); }

set { throw new NotImplementedException(); }

}

public override void FuncB()

{

//AbstractType abstactType = new AbstractType();//error:Cannot create an instance of abstract or interface AbstractType

throw new NotImplementedException();

}

}

/// <summary>

/// 测试抽象类的构造器是否可以在不继承它的类中使用

/// </summary>

public class TestAbstractBuilder

{

}

}

* + 继承
* 继承是OOP中最重要的特性之一。
* C#中规定类只能够单继承，即一个类只能继承一个基类，而可以继承多个接口。
* 任何类都可以从另外一个类中继承，那么这个类就拥有了继承的类的所有成员。继承可以从一个基类扩展更多的特定类。
* 基类的成员可以是虚拟的，通过派生类重写基类成员，即扩展基类的实现。
  + 多态性

多态性，从字面上看是对象拥有了多种形态或状态，或拥有了更多的功能。

接口的多态性，类可以派生自一个类，也可以继承接口，而接口中的方法被派生类实现，每个派生类都有自己独特的实现，而接口变量就可以使用这些不同的实现，来获取不同的功能，即多态。

* + 对象之间的关系

包含关系：A类可以声明在B类的属性中，从而控制通过B类访问另A类的成员

集合关系：使用数组或集合存储相同类型的对象，数组和集合的不同在于，集合需要实现接口定义的一些方法，如：Add()，AddRange()，或RemoveAt()等等

* + 运算符重载
  + 事件

事件是一种特殊的委托。

* + 引用类型和值类型

1. **类成员**
2. 字段

共有字段遵循PascalCasing命名规则，私有字段一般采用camelCasing方式。

字段可以用readonly、static、const修饰，

1. 方法（即：函数）及重构
2. 接口实现

接口成员不允许有任何访问修饰符，接口成员全是公开的。

接口成员不包含发代码体

接口不包含字段

接口成员不能使用static、virtual、abstract、sealed关键字定义

类型定义成员是禁止的

1. **集合、比较和转换**

1

1. **泛型**

1

1. **其他OOP技术**

1

1. **C#语言的改进**

1