这篇文章主要讲解C#中的泛型，泛型在C#中有很重要的地位，尤其是在搭建项目框架的时候。

一、什么是泛型

泛型是C#2.0推出的新语法，不是语法糖，而是2.0由框架升级提供的功能。

我们在编程程序时，经常会遇到功能非常相似的模块，只是它们处理的数据不一样。但我们没有办法，只能分别写多个方法来处理不同的数据类型。这个时候，那么问题来了，有没有一种办法，用同一个方法来处理传入不同种类型参数的办法呢？泛型的出现就是专门来解决这个问题的。

二、为什么使用泛型

先来看下面一个例子：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MyGeneric

{

public class CommonMethod

{

/// <summary>

/// 打印个int值

///

/// 因为方法声明的时候，写死了参数类型

/// 已婚的男人 Eleven San

/// </summary>

/// <param name="iParameter"></param>

public static void ShowInt(int iParameter)

{

Console.WriteLine("This is {0},parameter={1},type={2}",

typeof(CommonMethod).Name, iParameter.GetType().Name, iParameter);

}

/// <summary>

/// 打印个string值

/// </summary>

/// <param name="sParameter"></param>

public static void ShowString(string sParameter)

{

Console.WriteLine("This is {0},parameter={1},type={2}",

typeof(CommonMethod).Name, sParameter.GetType().Name, sParameter);

}

/// <summary>

/// 打印个DateTime值

/// </summary>

/// <param name="oParameter"></param>

public static void ShowDateTime(DateTime dtParameter)

{

Console.WriteLine("This is {0},parameter={1},type={2}",

typeof(CommonMethod).Name, dtParameter.GetType().Name, dtParameter);

}

}

}

[复制代码](javascript:void(0);)

 结果：

https://images2018.cnblogs.com/blog/1033738/201805/1033738-20180514070216936-605954407.png

从上面的结果中我们可以看出这三个方法，除了传入的参数不同外，其里面实现的功能都是一样的。在1.0版的时候，还没有泛型这个概念，那么怎么办呢。相信很多人会想到了OOP三大特性之一的继承，我们知道，C#语言中，object是所有类型的基类，将上面的代码进行以下优化：

public static void ShowObject(object oParameter)

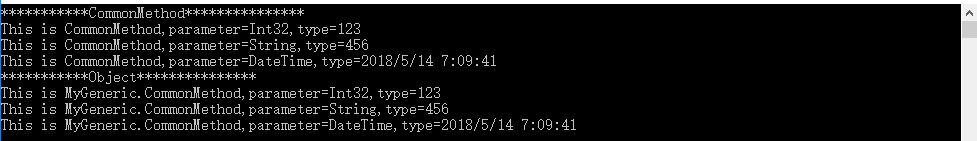
{

Console.WriteLine("This is {0},parameter={1},type={2}",

typeof(CommonMethod), oParameter.GetType().Name, oParameter);

}

 结果：



从上面的结果中我们可以看出，使用Object类型达到了我们的要求，解决了代码的可复用。可能有人会问定义的是object类型的，为什么可以传入int、string等类型呢？原因有二：

1、object类型是一切类型的父类。

2、通过继承，子类拥有父类的一切属性和行为，任何父类出现的地方，都可以用子类来代替。

但是上面object类型的方法又会带来另外一个问题：装箱和拆箱，会损耗程序的性能。

微软在C#2.0的时候推出了泛型，可以很好的解决上面的问题。

三、泛型类型参数

在泛型类型或方法定义中，类型参数是在其实例化泛型类型的一个变量时，客户端指定的特定类型的占位符。 泛型类( GenericList<T>)无法按原样使用，因为它不是真正的类型；它更像是类型的蓝图。 若要使用 GenericList<T>，客户端代码必须通过指定尖括号内的类型参数来声明并实例化构造类型。 此特定类的类型参数可以是编译器可识别的任何类型。 可创建任意数量的构造类型实例，其中每个使用不同的类型参数。

上面例子中的代码可以修改如下：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 public class GenericMethod

10 {

11 /// <summary>

12 /// 泛型方法

13 /// </summary>

14 /// <typeparam name="T"></typeparam>

15 /// <param name="tParameter"></param>

16 public static void Show<T>(T tParameter)

17 {

18 Console.WriteLine("This is {0},parameter={1},type={2}",

19 typeof(GenericMethod), tParameter.GetType().Name, tParameter.ToString());

20 }

21 }

22 }

[复制代码](javascript:void(0);)

 调用：

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace MyGeneric

{

class Program

{

static void Main(string[] args)

{

int iValue = 123;

string sValue = "456";

DateTime dtValue = DateTime.Now;

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*CommonMethod\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

CommonMethod.ShowInt(iValue);

CommonMethod.ShowString(sValue);

CommonMethod.ShowDateTime(dtValue);

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*Object\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

CommonMethod.ShowObject(iValue);

CommonMethod.ShowObject(sValue);

CommonMethod.ShowObject(dtValue);

Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*Generic\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

GenericMethod.Show<int>(iValue);

GenericMethod.Show<string>(sValue);

GenericMethod.Show<DateTime>(dtValue);

Console.ReadKey();

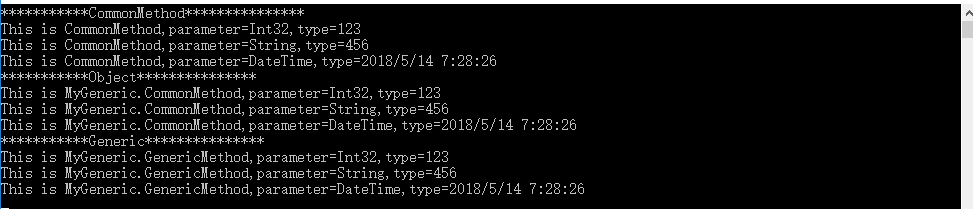
}

}

}

[复制代码](javascript:void(0);)

显示结果：



为什么泛型可以解决上面的问题呢？

泛型是延迟声明的：即定义的时候没有指定具体的参数类型，把参数类型的声明推迟到了调用的时候才指定参数类型。 延迟思想在程序架构设计的时候很受欢迎。例如：分布式缓存队列、EF的延迟加载等等。

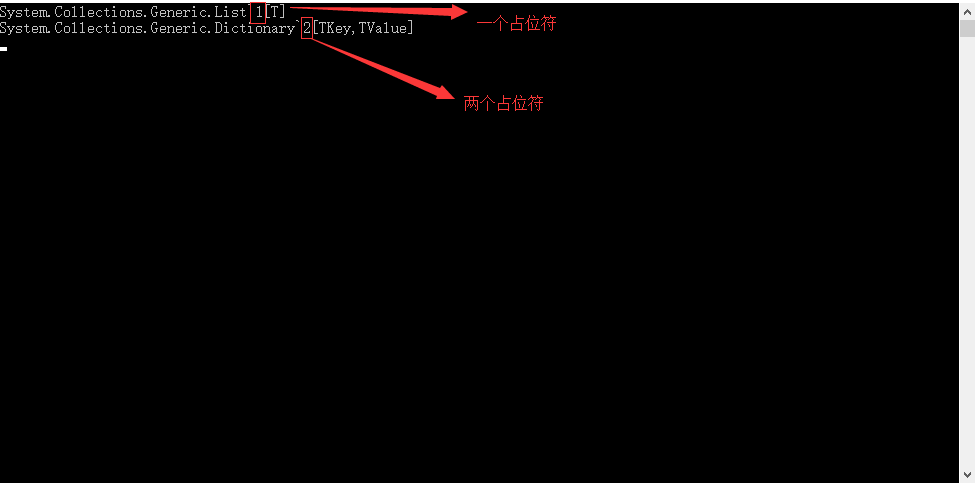
泛型究竟是如何工作的呢？

控制台程序最终会编译成一个exe程序，exe被点击的时候，会经过JIT(即时编译器)的编译，最终生成二进制代码，才能被计算机执行。泛型加入到语法以后，VS自带的编译器又做了升级，升级之后编译时遇到泛型，会做特殊的处理：生成占位符。再次经过JIT编译的时候，会把上面编译生成的占位符替换成具体的数据类型。请看下面一个例子：

1 Console.WriteLine(typeof(List<>));

2 Console.WriteLine(typeof(Dictionary<,>));

 结果：



从上面的截图中可以看出：泛型在编译之后会生成占位符。

注意：占位符需要在英文输入法状态下才能输入，只需要按一次波浪线(数字1左边的键位)的键位即可，不需要按Shift键。

1、泛型性能问题

请看一下的一个例子，比较普通方法、Object参数类型的方法、泛型方法的性能。

添加一个Monitor类，让三种方法执行同样的操作，比较用时长短：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 public class Monitor

10 {

11 public static void Show()

12 {

13 Console.WriteLine("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Monitor\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

14 {

15 int iValue = 12345;

16 long commonSecond = 0;

17 long objectSecond = 0;

18 long genericSecond = 0;

19

20 {

21 Stopwatch watch = new Stopwatch();

22 watch.Start();

23 for (int i = 0; i < 100000000; i++)

24 {

25 ShowInt(iValue);

26 }

27 watch.Stop();

28 commonSecond = watch.ElapsedMilliseconds;

29 }

30 {

31 Stopwatch watch = new Stopwatch();

32 watch.Start();

33 for (int i = 0; i < 100000000; i++)

34 {

35 ShowObject(iValue);

36 }

37 watch.Stop();

38 objectSecond = watch.ElapsedMilliseconds;

39 }

40 {

41 Stopwatch watch = new Stopwatch();

42 watch.Start();

43 for (int i = 0; i < 100000000; i++)

44 {

45 Show<int>(iValue);

46 }

47 watch.Stop();

48 genericSecond = watch.ElapsedMilliseconds;

49 }

50 Console.WriteLine("commonSecond={0},objectSecond={1},genericSecond={2}"

51 , commonSecond, objectSecond, genericSecond);

52 }

53 }

54

55 #region PrivateMethod

56 private static void ShowInt(int iParameter)

57 {

58 //do nothing

59 }

60 private static void ShowObject(object oParameter)

61 {

62 //do nothing

63 }

64 private static void Show<T>(T tParameter)

65 {

66 //do nothing

67 }

68 #endregion

69

70 }

71 }

[复制代码](javascript:void(0);)

 Main()方法调用：

1 Monitor.Show();

 结果：



从结果中可以看出：泛型方法的性能最高，其次是普通方法，object方法的性能最低。

四、泛型类

除了方法可以是泛型以外，类也可以是泛型的，例如：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 /// <summary>

10 /// 泛型类

11 /// </summary>

12 /// <typeparam name="T"></typeparam>

13 public class GenericClass<T>

14 {

15 public T \_T;

16 }

17 }

[复制代码](javascript:void(0);)

 Main()方法中调用：

[复制代码](javascript:void(0);)

1 // T是int类型

2 GenericClass<int> genericInt = new GenericClass<int>();

3 genericInt.\_T = 123;

4 // T是string类型

5 GenericClass<string> genericString = new GenericClass<string>();

6 genericString.\_T = "123";

[复制代码](javascript:void(0);)

 除了可以有泛型类，也可以有泛型接口，例如：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 /// <summary>

10 /// 泛型接口

11 /// </summary>

12 public interface IGenericInterface<T>

13 {

14 //泛型类型的返回值

15 T GetT(T t);

16 }

17 }

[复制代码](javascript:void(0);)

 也可以有泛型委托：

1 public delegate void SayHi<T>(T t);//泛型委托

 注意：

1、泛型在声明的时候可以不指定具体的类型，但是在使用的时候必须指定具体类型，例如：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 /// <summary>

10 /// 使用泛型的时候必须指定具体类型，

11 /// 这里的具体类型是int

12 /// </summary>

13 public class CommonClass :GenericClass<int>

14 {

15 }

16 }

[复制代码](javascript:void(0);)

 如果子类也是泛型的，那么继承的时候可以不指定具体类型，例如：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 /// <summary>

10 /// 使用泛型的时候必须指定具体类型，

11 /// 这里的具体类型是int

12 /// </summary>

13 public class CommonClass :GenericClass<int>

14 {

15 }

16

17 /// <summary>

18 /// 子类也是泛型的，继承的时候可以不指定具体类型

19 /// </summary>

20 /// <typeparam name="T"></typeparam>

21 public class CommonClassChild<T>:GenericClass<T>

22 {

23

24 }

25 }

[复制代码](javascript:void(0);)

 2、类实现泛型接口也是这种情况，例如：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 /// <summary>

10 /// 必须指定具体类型

11 /// </summary>

12 public class Common : IGenericInterface<string>

13 {

14 public string GetT(string t)

15 {

16 throw new NotImplementedException();

17 }

18 }

19

20 /// <summary>

21 /// 可以不知道具体类型，但是子类也必须是泛型的

22 /// </summary>

23 /// <typeparam name="T"></typeparam>

24 public class CommonChild<T> : IGenericInterface<T>

25 {

26 public T GetT(T t)

27 {

28 throw new NotImplementedException();

29 }

30 }

31 }

[复制代码](javascript:void(0);)

 五、泛型约束

先来看看下面的一个例子：

定义一个People类，里面有属性和方法：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 public interface ISports

10 {

11 void Pingpang();

12 }

13

14 public interface IWork

15 {

16 void Work();

17 }

18

19

20 public class People

21 {

22 public int Id { get; set; }

23 public string Name { get; set; }

24

25 public void Hi()

26 {

27 Console.WriteLine("Hi");

28 }

29 }

30

31 public class Chinese : People, ISports, IWork

32 {

33 public void Tradition()

34 {

35 Console.WriteLine("仁义礼智信，温良恭俭让");

36 }

37 public void SayHi()

38 {

39 Console.WriteLine("吃了么？");

40 }

41

42 public void Pingpang()

43 {

44 Console.WriteLine("打乒乓球...");

45 }

46

47 public void Work()

48 {

49 throw new NotImplementedException();

50 }

51 }

52

53 public class Hubei : Chinese

54 {

55 public Hubei(int version)

56 { }

57

58 public string Changjiang { get; set; }

59 public void Majiang()

60 {

61 Console.WriteLine("打麻将啦。。");

62 }

63 }

64

65

66 public class Japanese : ISports

67 {

68 public int Id { get; set; }

69 public string Name { get; set; }

70 public void Hi()

71 {

72 Console.WriteLine("Hi");

73 }

74 public void Pingpang()

75 {

76 Console.WriteLine("打乒乓球...");

77 }

78 }

79 }

[复制代码](javascript:void(0);)

 在Main()方法里面实例化：

[复制代码](javascript:void(0);)

1 People people = new People()

2 {

3 Id = 123,

4 Name = "走自己的路"

5 };

6 Chinese chinese = new Chinese()

7 {

8 Id = 234,

9 Name = "晴天"

10 };

11 Hubei hubei = new Hubei(123)

12 {

13 Id = 345,

14 Name = "流年"

15 };

16 Japanese japanese = new Japanese()

17 {

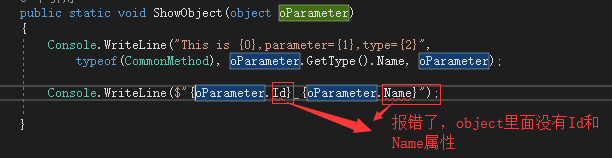
18 Id = 7654,

19 Name = "werwer"

20 };

[复制代码](javascript:void(0);)

 这时有一个需求：需要打印出Id和Name属性的值，将ShowObject()方法修改如下：



但是这样修改报错了：object类里面没有Id和Name属性，可能会有人说，强制类型转换一下就行了啊：

[复制代码](javascript:void(0);)

1 public static void ShowObject(object oParameter)

2 {

3 Console.WriteLine("This is {0},parameter={1},type={2}",

4 typeof(CommonMethod), oParameter.GetType().Name, oParameter);

5

6 Console.WriteLine($"{((People)oParameter).Id}\_{((People)oParameter).Name}");

7 }

[复制代码](javascript:void(0);)

 这样修改以后，代码不会报错了，这时我们在Main()方法里面调用：

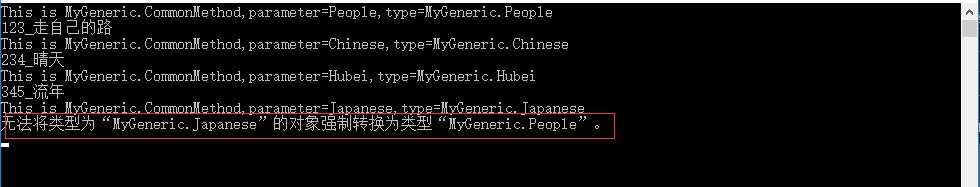
1 CommonMethod.ShowObject(people);

2 CommonMethod.ShowObject(chinese);

3 CommonMethod.ShowObject(hubei);

4 CommonMethod.ShowObject(japanese);

 结果：



可以看出程序报错了，因为Japanese没有继承自People，这里类型转换的时候失败了。这样会造成类型不安全的问题。那么怎么解决类型不安全的问题呢？那就是使用泛型约束。

所谓的泛型约束，实际上就是约束的类型T。使T必须遵循一定的规则。比如T必须继承自某个类，或者T必须实现某个接口等等。那么怎么给泛型指定约束？其实也很简单，只需要where关键字，加上约束的条件。

泛型约束总共有五种。

|  |  |
| --- | --- |
| 约束 | s说明 |
| T：结构 | 类型参数必须是值类型 |
| T：类 | 类型参数必须是引用类型；这一点也适用于任何类、接口、委托或数组类型。 |
| T：new() | 类型参数必须具有无参数的公共构造函数。 当与其他约束一起使用时，new() 约束必须最后指定。 |
| T：<基类名> | 类型参数必须是指定的基类或派生自指定的基类。 |
| T：<接口名称> | 类型参数必须是指定的接口或实现指定的接口。 可以指定多个接口约束。 约束接口也可以是泛型的。 |

先看下面的示例代码：

[复制代码](javascript:void(0);)

class MyList<T>

{

List<T> list = new List<T>();

public T this[int i]

{

get { return list[i]; }

set { this.list[i] = value; }

}

}

class Person

{

public string Name { get; set; }

}

[复制代码](javascript:void(0);)

**1 指定泛型参数为值类型**

class MyList<T> where T:struct

{

...代码省略部分

}

看看加上约束后的效果怎么样，按下面方式 实例化MyList 类 ：

MyList<Person> list = new MyList<Person>();

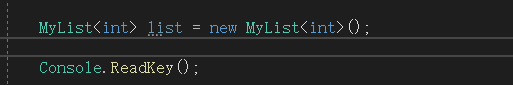
你会发现，将会产生下面的错误提示：



使用下面的方式实例化MyList 类 ，将一切正常：

MyList<int> list = new MyList<int>();

如图所示：



**2  指定泛型参数为引用类型**

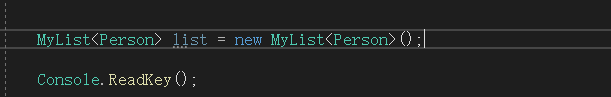
class MyList<T> where T:class

{

...代码省略部分

}

修改泛型约束为 引用类型后，前面的错误提示消失，因为 Person 类是引用类型，满足泛型约束。



**3  指定泛型参数有无参的公共的构造函数**

class MyList<T> where T:new()

{

...代码省略部分

}

为Person类增加私有的无参构造函数，代码如下：

[复制代码](javascript:void(0);)

class Person

{

public string Name { get; set; }

private Person()

{

//do nothing

}

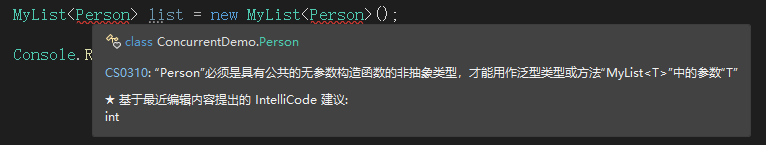
}

[复制代码](javascript:void(0);)

实例化 MyList < Person > 类：

MyList<Person> list = new MyList<Person>();

出现编译错误，提示：



把Person类里面私有的构造函数去掉就可以了。

**4  指定泛型参数 必须派生于指定基类**

增加抽象类 SeniorAnimal

abstract class SeniorAnimal//高级动物

{

public abstract void Speak();//会说话

public abstract void UseTool();//会使用工具

}

指定泛型参数 必须派生于基类 SeniorAnimal

class MyList<T> where T : SeniorAnimal

{

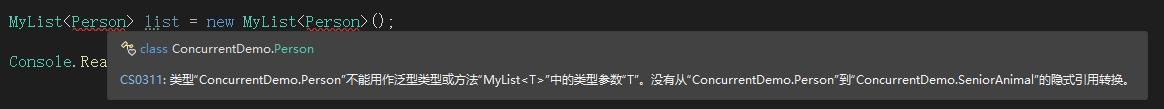
...代码省略部分

}

实例化 MyList < Person > 类（此时Person类还未继承自 SeniorAnimal 类 ）：

MyList<Person> list = new MyList<Person>();

出现编译错误，提示：



修改代码，使Person类继承自 SeniorAnimal 类：

[复制代码](javascript:void(0);)

class Person : SeniorAnimal

{

public string Name { get; set; }

public override void Speak()

{

Console.WriteLine("我会说脏话！");

}

public override void UseTool()

{

Console.WriteLine("我会用砍刀！");

}

}

[复制代码](javascript:void(0);)

再次编译，一切正常。

**5 指定泛型参数 必须实现指定接口**

演示情况和第4点类似，就不提供演示代码了。

1、基类约束

上面打印的方法约束T类型必须是People类型。

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 基类约束：约束T必须是People类型或者是People的子类

3 /// </summary>

4 /// <typeparam name="T"></typeparam>

5 /// <param name="tParameter"></param>

6 public static void Show<T>(T tParameter) where T : People

7 {

8 Console.WriteLine($"{tParameter.Id}\_{tParameter.Name}");

9 tParameter.Hi();

10 }

[复制代码](javascript:void(0);)

 注意：

基类约束时，基类不能是密封类，即不能是sealed类。sealed类表示该类不能被继承，在这里用作约束就无任何意义，因为sealed类没有子类。

2、接口约束

[复制代码](javascript:void(0);)

/// <summary>

/// 接口约束

/// </summary>

/// <typeparam name="T"></typeparam>

/// <param name="t"></param>

/// <returns></returns>

public static T Get<T>(T t) where T : ISports

{

t.Pingpang();

return t;

}

[复制代码](javascript:void(0);)

 3、引用类型约束 class

引用类型约束保证T一定是引用类型的。

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 引用类型约束

3 /// </summary>

4 /// <typeparam name="T"></typeparam>

5 /// <param name="t"></param>

6 /// <returns></returns>

7 public static T Get<T>(T t) where T : class

8 {

9 return t;

10 }

[复制代码](javascript:void(0);)

 4、值类型约束  struct

值类型约束保证T一定是值类型的。

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 值类型类型约束

3 /// </summary>

4 /// <typeparam name="T"></typeparam>

5 /// <param name="t"></param>

6 /// <returns></returns>

7 public static T Get<T>(T t) where T : struct

8 {

9 return t;

10 }

[复制代码](javascript:void(0);)

 5、无参数构造函数约束  new()

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// new()约束

3 /// </summary>

4 /// <typeparam name="T"></typeparam>

5 /// <param name="t"></param>

6 /// <returns></returns>

7 public static T Get<T>(T t) where T : new()

8 {

9 return t;

10 }

[复制代码](javascript:void(0);)

 泛型约束也可以同时约束多个，例如：

[复制代码](javascript:void(0);)

1 public static void Show<T>(T tParameter)

2 where T : People, ISports, IWork, new()

3 {

4 Console.WriteLine($"{tParameter.Id}\_{tParameter.Name}");

5 tParameter.Hi();

6 tParameter.Pingpang();

7 tParameter.Work();

8 }

[复制代码](javascript:void(0);)

 注意：有多个泛型约束时，new()约束一定是在最后。

六、泛型的协变和逆变

协变和逆变是在.NET 4.0的时候出现的，只能放在接口或者委托的泛型参数前面，out 协变covariant，用来修饰返回值；in：逆变contravariant，用来修饰传入参数。

先看下面的一个例子：

定义一个Animal类：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 public class Animal

10 {

11 public int Id { get; set; }

12 }

13 }

[复制代码](javascript:void(0);)

 然后在定义一个Cat类继承自Animal类：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 public class Cat :Animal

10 {

11 public string Name { get; set; }

12 }

13 }

[复制代码](javascript:void(0);)

 在Main()方法可以这样调用：

[复制代码](javascript:void(0);)

1 // 直接声明Animal类

2 Animal animal = new Animal();

3 // 直接声明Cat类

4 Cat cat = new Cat();

5 // 声明子类对象指向父类

6 Animal animal2 = new Cat();

7 // 声明Animal类的集合

8 List<Animal> listAnimal = new List<Animal>();

9 // 声明Cat类的集合

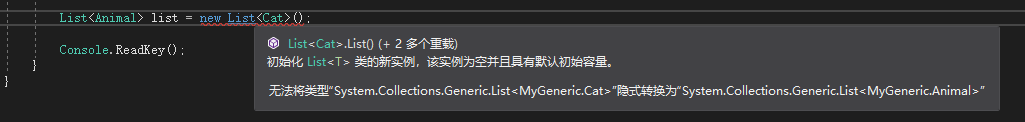
10 List<Cat> listCat = new List<Cat>();

[复制代码](javascript:void(0);)

 那么问题来了：下面的一句代码是不是正确的呢？

1 List<Animal> list = new List<Cat>();

 可能有人会认为是正确的：因为一只Cat属于Animal，那么一群Cat也应该属于Animal啊。但是实际上这样声明是错误的：因为List<Cat>和List<Animal>之间没有父子关系。



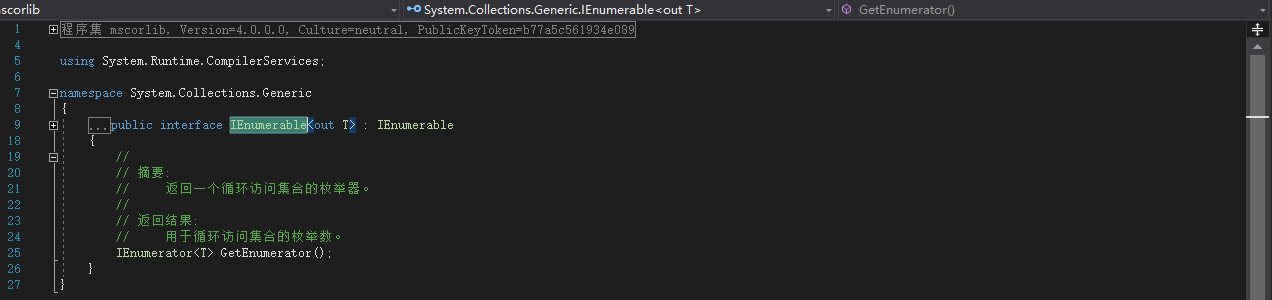
这时就可以用到协变和逆变了。

1 // 协变

2 IEnumerable<Animal> List1 = new List<Animal>();

3 IEnumerable<Animal> List2 = new List<Cat>();

 F12查看定义：



可以看到，在泛型接口的T前面有一个out关键字修饰，而且T只能是返回值类型，不能作为参数类型，这就是协变。使用了协变以后，左边声明的是基类，右边可以声明基类或者基类的子类。

协变除了可以用在接口上面，也可以用在委托上面：

1 Func<Animal> func = new Func<Cat>(() => null);

 除了使用.NET框架定义好的以为，我们还可以自定义协变，例如：

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// out 协变 只能是返回结果

3 /// </summary>

4 /// <typeparam name="T"></typeparam>

5 public interface ICustomerListOut<out T>

6 {

7 T Get();

8 }

9

10 public class CustomerListOut<T> : ICustomerListOut<T>

11 {

12 public T Get()

13 {

14 return default(T);

15 }

16 }

[复制代码](javascript:void(0);)

 使用自定义的协变：

1 // 使用自定义协变

2 ICustomerListOut<Animal> customerList1 = new CustomerListOut<Animal>();

3 ICustomerListOut<Animal> customerList2 = new CustomerListOut<Cat>();

 在来看看逆变。

在泛型接口的T前面有一个In关键字修饰，而且T只能方法参数，不能作为返回值类型，这就是逆变。请看下面的自定义逆变：

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// 逆变 只能是方法参数

3 /// </summary>

4 /// <typeparam name="T"></typeparam>

5 public interface ICustomerListIn<in T>

6 {

7 void Show(T t);

8 }

9

10 public class CustomerListIn<T> : ICustomerListIn<T>

11 {

12 public void Show(T t)

13 {

14 }

15 }

[复制代码](javascript:void(0);)

 使用自定义逆变：

1 // 使用自定义逆变

2 ICustomerListIn<Cat> customerListCat1 = new CustomerListIn<Cat>();

3 ICustomerListIn<Cat> customerListCat2 = new CustomerListIn<Animal>();

 协变和逆变也可以同时使用，看看下面的例子：

[复制代码](javascript:void(0);)

1 /// <summary>

2 /// inT 逆变

3 /// outT 协变

4 /// </summary>

5 /// <typeparam name="inT"></typeparam>

6 /// <typeparam name="outT"></typeparam>

7 public interface IMyList<in inT, out outT>

8 {

9 void Show(inT t);

10 outT Get();

11 outT Do(inT t);

12 }

13

14 public class MyList<T1, T2> : IMyList<T1, T2>

15 {

16

17 public void Show(T1 t)

18 {

19 Console.WriteLine(t.GetType().Name);

20 }

21

22 public T2 Get()

23 {

24 Console.WriteLine(typeof(T2).Name);

25 return default(T2);

26 }

27

28 public T2 Do(T1 t)

29 {

30 Console.WriteLine(t.GetType().Name);

31 Console.WriteLine(typeof(T2).Name);

32 return default(T2);

33 }

34 }

[复制代码](javascript:void(0);)

 使用：

1 IMyList<Cat, Animal> myList1 = new MyList<Cat, Animal>();

2 IMyList<Cat, Animal> myList2 = new MyList<Cat, Cat>();//协变

3 IMyList<Cat, Animal> myList3 = new MyList<Animal, Animal>();//逆变

4 IMyList<Cat, Animal> myList4 = new MyList<Animal, Cat>();//逆变+协变

 七、泛型缓存

在前面我们学习过，类中的静态类型无论实例化多少次，在内存中只会有一个。静态构造函数只会执行一次。在泛型类中，T类型不同，每个不同的T类型，都会产生一个不同的副本，所以会产生不同的静态属性、不同的静态构造函数，请看下面的例子：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading.Tasks;

6

7 namespace MyGeneric

8 {

9 public class GenericCache<T>

10 {

11 static GenericCache()

12 {

13 Console.WriteLine("This is GenericCache 静态构造函数");

14 \_TypeTime = string.Format("{0}\_{1}", typeof(T).FullName, DateTime.Now.ToString("yyyyMMddHHmmss.fff"));

15 }

16

17 private static string \_TypeTime = "";

18

19 public static string GetCache()

20 {

21 return \_TypeTime;

22 }

23 }

24 }

[复制代码](javascript:void(0);)

 然后新建一个测试类，用来测试GenericCache类的执行顺序：

[复制代码](javascript:void(0);)

1 using System;

2 using System.Collections.Generic;

3 using System.Linq;

4 using System.Text;

5 using System.Threading;

6 using System.Threading.Tasks;

7

8 namespace MyGeneric

9 {

10 public class GenericCacheTest

11 {

12 public static void Show()

13 {

14 for (int i = 0; i < 5; i++)

15 {

16 Console.WriteLine(GenericCache<int>.GetCache());

17 Thread.Sleep(10);

18 Console.WriteLine(GenericCache<long>.GetCache());

19 Thread.Sleep(10);

20 Console.WriteLine(GenericCache<DateTime>.GetCache());

21 Thread.Sleep(10);

22 Console.WriteLine(GenericCache<string>.GetCache());

23 Thread.Sleep(10);

24 Console.WriteLine(GenericCache<GenericCacheTest>.GetCache());

25 Thread.Sleep(10);

26 }

27 }

28 }

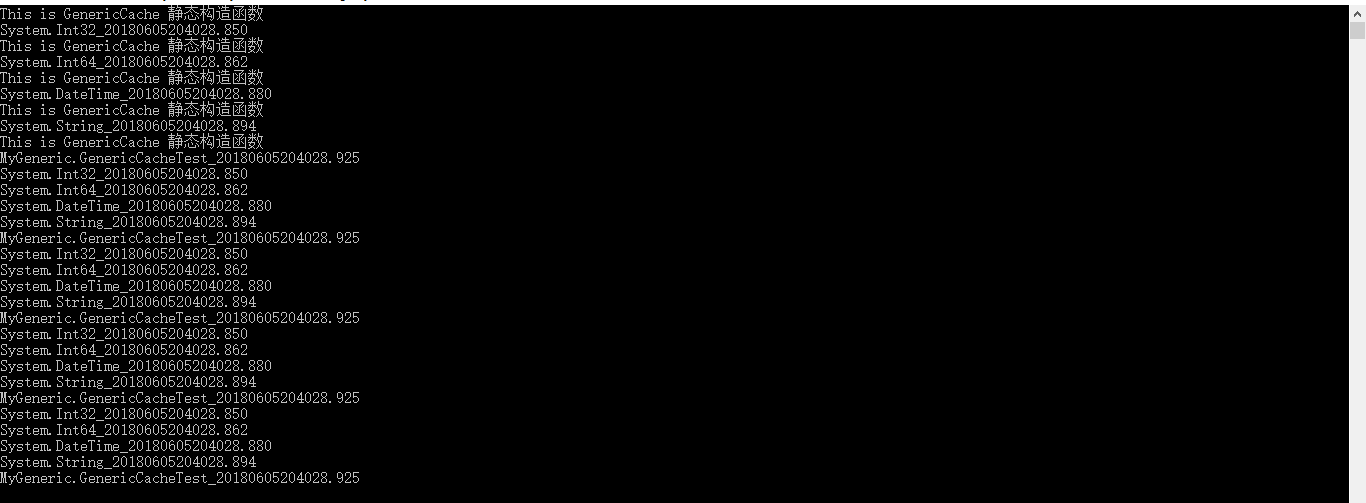
29 }

[复制代码](javascript:void(0);)

 Main()方法里面调用：

1 GenericCacheTest.Show();

 结果：



从上面的截图中可以看出，泛型会为不同的类型都创建一个副本，所以静态构造函数会执行5次。 而且每次静态属性的值都是一样的。利用泛型的这一特性，可以实现缓存。

注意：只能为不同的类型缓存一次。泛型缓存比字典缓存效率高。泛型缓存不能主动释放