**C# 泛型详解(泛型类，方法，接口，委托，约束，反射 )**

本文详细介绍了C#中的泛型，包括泛型的基本概念、为何使用泛型、泛型与Object类型的区别、泛型类、泛型方法、泛型接口、泛型委托、泛型约束以及泛型与反射的结合使用。通过实例展示了泛型在提高代码复用性和安全性上的优势，并提供了一个利用泛型和反射检查对象属性是否改变的实用示例。

摘要由CSDN通过智能技术生成

**一、什么是泛型**

先看一段介绍

泛型(Generic)，是将不确定的类型预先定义下来的一种C#高级语法，我们在使用一个类，接口或者方法前，不知道用户将来传什么类型，或者我们写的类，接口或方法相同的代码可以服务[不同的](https://so.csdn.net/so/search?q=%E4%B8%8D%E5%90%8C%E7%9A%84&spm=1001.2101.3001.7020)类型，就可以定义为泛型。这会大大简化我们的代码结构，同时让后期维护变得容易。

泛型很适用于集合，我们常见的泛型集合有：List<T>,Dictionary<K,V>等等（T，K，V就代表不确定的类型，它是一种类型占位符），无一不是利用的泛型这一特性，若没有泛型，我们会多出很多重载方法，以解决类型不同，但是执行逻辑相同的情况。

那么泛型到底长啥样？，看到下面代码中的 “T” 没有，那就是泛型。

1. namespace 泛型
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. Test<string>();
9. Console.ReadKey();
10. }
12. static void Test<T>()
13. {
14. Console.WriteLine(typeof(T).FullName);
15. }
16. }
17. }

输出：System.String

你可能会问，这个 “T” 就是泛型啊？把 “T” 改成鸡，那鸡不也是泛型了？

1. namespace 泛型
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. Test<string>();
8. Console.ReadKey();
9. }
11. static void Test<鸡>()
12. {
13. Console.WriteLine("鸡你太美");
14. }
15. }
16. }

是的，没错，输出：鸡你太美

**二、为什么要用泛型**

先看一个例子

1. namespace 泛型
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. int result = Test.Add(3, 4);
8. Console.WriteLine(result);
10. Console.ReadKey();
11. }
13. }
15. public class Test
16. {
17. public static int Add(int x, int y)
18. {
19. return x + y;
20. }
22. public static int Add(string x, string y)
23. {
24. return int.Parse(x) + int.Parse(y);
25. }
26. }
27. }

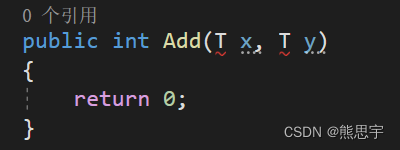
https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

比如，我们需要封装一个加法的算法，但是传入的类型，可能有 int 类型，也可能有 string 类型，还可能有其他的类型，但是每多一种类型，你就要多加一个方法，非常的麻烦，臃肿。

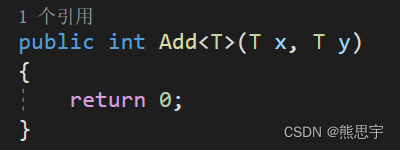
那么有没有方法来解决这个问题呢，于是后面C#2.0中，就有了泛型这个概念，它并不是语法糖，看字面意思像 “泛滥的类型” 的意思。

那么下面就用泛型的方式来封装一下这个加法运算。

注意一下写法，在 Add 后面要加上 <T> 这句，不然会报错。



正确方式



下面是封装加法的完整代码

1. public class Test
2. {
3. public static int Add<T>(T x,T y)
4. {
5. try
6. {
7. return int.Parse(x.ToString()) + int.Parse(y.ToString());
8. }
9. catch (Exception ex)
10. {
11. Console.WriteLine(ex.Message);
12. }
13. return -1;
14. }
15. }

测试第一种

1. class Program
2. {
3. static void Main(string[] args)
4. {
5. int result = Test.Add("3", "4");
6. Console.WriteLine(result);
8. Console.ReadKey();
9. }
10. }

输出：7

测试第二种

1. class Program
2. {
3. static void Main(string[] args)
4. {
5. int result = Test.Add(4, 5);
6. Console.WriteLine(result);
8. Console.ReadKey();
9. }
10. }

输出：9

**三、泛型和Object类型的区别**

看了上面的案例，你会觉得，泛型是不是 Object 类型不是也差不多？下面是一些介绍：

C# 中 Object 是一切类型的基类，可以用来表示所有类型，而泛型是指将类型参数化以达到代码复用提高软件开发工作效率的一种数据类型。

你可以将泛型理解成替换，在使用的时候将泛型参数替换成具体的类型，这个过程是在编译的时候进行的，使用泛型编译器依然能够检测出类型错误。泛型不用拆箱装箱。

而 Object 表示其他类型是通过类型转换来完成的，而所有类型转化为 Object 类型都是合法的，所以即使你先将 Object 对象赋值为一个整数再赋值为一个字符串，编译器都认为是合法的。

**Object 类型**

> 优点：  
> 1. object类型可以用来引用任何类型的实例；  
> 2. object类型可以存储任何类型的值；  
> 3. 可以定义object类型的参数；  
> 4. 可以把object作为返回类型。

> 缺点：  
> 1. 会因为程序员没有记住使用的类型而出错，造成类型不兼容；  
> 2. 值类型和引用类型的互化即装箱拆箱使系统性能下降。

泛型，Object 类型，var 类型，这三种类型，看着很类似，但其实泛型的作用更不止如此，还有[泛型类](https://so.csdn.net/so/search?q=%E6%B3%9B%E5%9E%8B%E7%B1%BB&spm=1001.2101.3001.7020)，泛型方法，泛型接口，泛型约束等，下面分别来介绍这几个功能

**四、泛型类**

泛型类是指这个类的某些字段的类型是不确定的，只有在构造的时候才能确定下的类，泛型类一般在数据结构中用的比较多，比如 C# 自带的 List<T>，Dictionary<TKey, TValue> 等，我也写过自定义 List 相关的帖子，有兴趣的可以去看下

[C# 自定义List\_熊思宇的博客-CSDN博客\_c#定义list](https://blog.csdn.net/qq_38693757/article/details/126263502?spm=1001.2014.3001.5501)

**1.泛型类案例**

下面看一个例子

1. namespace 泛型
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. Test<int> test = new Test<int>();
8. test.Add(1);
9. test.Add(3);
10. Console.WriteLine(test.Count);
12. Console.ReadKey();
13. }
14. }
16. public class Test<T>
17. {
18. private List<T> list = new List<T>();
20. public int Count => list.Count;
22. public void Add(T t)
23. {
24. if(t != null)
25. list.Add(t);
26. }
27. }
28. }

运行后输出：2

Test 类此时并没有添加任何的约束，这个是不推荐的，至少应该加一个，如下

1. public class Test<T> where T : new()
2. {
3. private List<T> list = new List<T>();
5. public int Count => list.Count;
7. public void Add(T t)
8. {
9. if(t != null)
10. list.Add(t);
11. }
12. }

关于泛型约束，在后面的章节中会有详细的介绍。

**2.泛型类继承**

泛型类型继承写法：

1. public class Test1<T>
2. {
4. }
6. public class Test2<T> : Test1<T>
7. {
9. }

指定基类的类型写法：

1. public class Test1<T>
2. {
4. }
6. public class Test2<T> : Test1<int>
7. {
9. }

**五、**[**泛型方法**](https://so.csdn.net/so/search?q=%E6%B3%9B%E5%9E%8B%E6%96%B9%E6%B3%95&spm=1001.2101.3001.7020)

泛型方法是指通过泛型来约束方法中的参数类型，如果没有泛型，每次方法中的参数类型都是固定的，不能随意更改，在使用泛型后，方法中的数据类型则有指定的泛型来约束，即可以根据提供的泛型来传递不同类型的参数。

泛型方法的返回类型和参数类型都可以使用泛型。如下:

1. public static T GetT<T>(T a)
2. {
3. return a;
4. }

泛型方法同样可以使用泛型约束，只是泛型方法中的 T 只能用在方法内部和返回值，而泛型类中的 T 可以应用于整个类的字段和方法。

1. public class Test
2. {
3. public void Add<T>(T t) where T : new()
4. {
5. List<T> list = new List<T>();
6. if (t != null) list.Add(t);
7. }
8. }

如果要对 T 类型作一些操作，则需要用到反射。

**六、泛型接口**

泛型接口如下

1. public interface IFace<T>
2. {
3. void SayHi(T msg);
4. }

同样的，泛型接口也可以使用泛型约束

1. public interface IFace<T> where T : new()
2. {
3. void SayHi(T msg);
4. }

实现泛型接口有两种情况

1. */// <summary>*
2. */// 1.普通类实现泛型接口*
3. */// </summary>*
4. public class MyClass1 : IFace<string>
5. {
6. public void SayHi(string msg)
7. {
8. Console.WriteLine(msg);
9. }
10. }
12. */// <summary>*
13. */// 2.泛型类继承泛型接口*
14. */// </summary>*
15. */// <typeparam name="T"></typeparam>*
16. public class MyClass2<T> : IFace<T>
17. {
18. public void SayHi(T msg)
19. {
20. Console.WriteLine(msg);
21. }
22. }

案例

1. namespace 泛型
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. IFace<Msg> face = new Test();
8. face.SayHi(new Msg());
9. Console.ReadKey();
10. }
11. }
13. public interface IFace<T> where T : new()
14. {
15. void SayHi(T msg);
16. }
18. public class Test : IFace<Msg>
19. {
20. public void SayHi(Msg msg)
21. {
22. Console.WriteLine(msg.MyName);
23. }
24. }
26. public class Msg
27. {
28. public string MyName = "张三";
29. }
30. }

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

输出：张三

**七、泛型委托**

泛型委托和普通委托差别不大，只是参数由具体类型变成了 "T"

public delegate void MyDelegate<T>(T args);

案例

1. namespace 泛型
2. {
3. class Program
4. {
5. delegate void MyDelegate<T>(T args);
7. static void Main(string[] args)
8. {
9. MyDelegate<string> myDelegate = SayHi;
10. myDelegate("哈喽");
12. Console.ReadKey();
13. }
15. static void SayHi(string msg)
16. {
17. Console.WriteLine(msg);
18. }
19. }
20. }

输出：哈喽

**八、泛型约束**

泛型约束就是对 “T” 类型的数据做一定的限制，防止在运用过程中，做一些违规的操作，来保证数据的安全。

**六种类型的约束：**

|  |  |
| --- | --- |
| T：结构 | 类型参数必须是值类型。可以指定除 Nullable 以外的任何值类型。有关更多信息，请参见使用可空类型（C# 编程指南）。 |
| T：类 | 类型参数必须是引用类型，包括任何类、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。 |
| T：U | 为 T 提供的类型参数必须是为 U 提供的参数或派生自为 U 提供的参数。这称为裸类型约束。 |

案例

1. namespace 泛型
2. {
3. class Program
4. {
5. static void Main(string[] args)
6. {
7. Console.ReadKey();
8. }
9. }
11. */// <summary>*
12. */// 泛型接口*
13. */// </summary>*
14. */// <typeparam name="T"></typeparam>*
15. public interface IFace<T>
16. {
17. void SayHi(T msg);
18. }
20. */// <summary>*
21. */// 泛型约束*
22. */// </summary>*
23. public class MyClass1<T, K, V, W, X,Y>
24. where T : struct *//约束 T 必须是值类型*
25. where K : class *//约束 K 必须是引用类型*
26. where V : IFace<T> *//约束 V 必须实现 IFace 接口*
27. where W : K *//约束 W 必须是 K 类型，或者是 K 类型的子类*
28. where X : class, new() *//约束 X 必须是引用类型，并且有一个无参数的构造函数，当有多个约束时，new()必须写在最后*
29. where Y : MyClass2 *//约束 Y 必须是 MyClass2 类型，或者继承于 MyClass2 类*
30. {
31. public void Add(T num)
32. {
33. Console.WriteLine(num);
34. }
35. }
37. public class MyClass2
38. {
40. }
41. }

这个案例，几乎介绍了所有的泛型约束的使用方法，有兴趣的可以亲自动手写一写。

**九、泛型配合反射**

现在有一个需求，用户修改个人资料，在提交这里必须要判断是否有修改内容，如果没有修改，点击提交按钮则不提交，那要怎么判断呢？

有人可能会说，用 if...else 去判断就好了，可以这么写没错，但资料如果有几百个地方改了，不会还用 if...else 吧，那整篇代码全是 if...else 了，就像我同事开玩笑一样：“我不会高数，我只会写 if...else...”

下面这个案例就教你如何来解决这个问题。

代码

1. using System;
2. using System.Collections.Generic;
3. using System.Linq;
4. using System.Reflection;
5. using System.Text;
6. using System.Threading.Tasks;
8. namespace 泛型
9. {
10. class Program
11. {
12. static void Main(string[] args)
13. {
14. Test test1 = new Test();
15. test1.Names = "张三";
16. test1.Age = 24;
17. test1.Height = 179.9f;
18. Test test2 = new Test();
19. test2.Names = "李四";
20. test2.Age = 31;
21. test2.Height = 163.4f;
23. bool result = CompareType(test1, test2);
24. Console.WriteLine("是否相同：" + result);
26. Console.ReadKey();
27. }
29. */// <summary>*
30. */// 比较--两个类型一样的实体类对象的值*
31. */// </summary>*
32. */// <param name="oneT"></param>*
33. */// <returns>返回true表示两个对象的数据相同，返回false表示不相同</returns>*
34. public static bool CompareType<T>(T oneT, T twoT)
35. {
36. bool result = true;*//两个类型作比较时使用,如果有不一样的就false*
37. Type typeOne = oneT.GetType();
38. Type typeTwo = twoT.GetType();
39. *//如果两个T类型不一样 就不作比较*
40. if (!typeOne.Equals(typeTwo)) { return false; }
41. PropertyInfo[] pisOne = typeOne.GetProperties(); *//获取所有公共属性(Public)*
42. PropertyInfo[] pisTwo = typeTwo.GetProperties();
43. *//如果长度为0返回false*
44. if (pisOne.Length <= 0 || pisTwo.Length <= 0)
45. {
46. return false;
47. }
48. *//如果长度不一样，返回false*
49. if (!(pisOne.Length.Equals(pisTwo.Length))) { return false; }
50. *//遍历两个T类型，遍历属性，并作比较*
51. for (int i = 0; i < pisOne.Length; i++)
52. {
53. *//获取属性名*
54. string oneName = pisOne[i].Name;
55. string twoName = pisTwo[i].Name;
56. *//获取属性的值*
57. object oneValue = pisOne[i].GetValue(oneT, null);
58. object twoValue = pisTwo[i].GetValue(twoT, null);
59. *//比较,只比较值类型*
60. if ((pisOne[i].PropertyType.IsValueType || pisOne[i].PropertyType.Name.StartsWith("String")) && (pisTwo[i].PropertyType.IsValueType || pisTwo[i].PropertyType.Name.StartsWith("String")))
61. {
62. if (oneName.Equals(twoName))
63. {
64. if (oneValue == null)
65. {
66. if (twoValue != null)
67. {
68. result = false;
69. break; *//如果有不一样的就退出循环*
70. }
71. }
72. else if (oneValue != null)
73. {
74. if (twoValue != null)
75. {
76. if (!oneValue.Equals(twoValue))
77. {
78. result = false;
79. break; *//如果有不一样的就退出循环*
80. }
81. }
82. else if (twoValue == null)
83. {
84. result = false;
85. break; *//如果有不一样的就退出循环*
86. }
87. }
88. }
89. else
90. {
91. result = false;
92. break;
93. }
94. }
95. else
96. {
97. *//如果对象中的属性是实体类对象，递归遍历比较*
98. bool b = CompareType(oneValue, twoValue);
99. if (!b) { result = b; break; }
100. }
101. }
102. return result;
103. }
104. }
106. public class Test
107. {
108. public string Names { get; set; }
109. public int Age { get; set; }
110. public float Height { get; set; }
111. }
112. }

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

输出：是否相同：False

将这两个 Test 对象的值改为一样试试

1. static void Main(string[] args)
2. {
3. Test test1 = new Test();
4. test1.Names = "张三";
5. test1.Age = 24;
6. test1.Height = 179.9f;
7. Test test2 = new Test();
8. test2.Names = "张三";
9. test2.Age = 24;
10. test2.Height = 179.9f;
12. bool result = CompareType(test1, test2);
13. Console.WriteLine("是否相同：" + result);
15. Console.ReadKey();
16. }

https://csdnimg.cn/release/blogv2/dist/pc/img/newCodeMoreWhite.png

输出：是否相同：True

可以看到，输出结果是对的

**结束**