# 泛型约束表

在定义泛型类时，可以对客户端代码能够在实例化类时用于类型参数的类型种类施加限制。如果客户端代码尝试使用某个约束所不允许的类型来实例化类，则会产生编译时错误。这些限制称为约束。约束是使用 **where** 上下文关键字指定的。下表列出了六种类型的约束：

| **约束** | **说明** |
| --- | --- |
| T：结构 | 类型参数必须是值类型。可以指定除 [Nullable](https://msdn.microsoft.com/zh-cn/library/system.nullable.aspx) 以外的任何值类型。有关更多信息，请参见[使用可以为 null 的类型（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/2cf62fcy.aspx)。 |
| 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。 |
| 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。 |

## 派生约束

1.常见的

public class MyClass5<T> where T :IComparable { }

2.约束放在类的实际派生之后

public class B { }

public class MyClass6<T> : B where T : IComparable { }

3.可以继承一个基类和多个接口，且基类在接口前面

public class B { }

public class MyClass7<T> where T : B, IComparable, ICloneable { }

## 构造函数约束

1.常见的

public class MyClass8<T> where T :  new() { }

2.可以将构造函数约束和派生约束组合起来,前提是构造函数约束出现在约束列表的最后

public class MyClass8<T> where T : IComparable, new() { }

## 值约束

1.常见的

public class MyClass9<T> where T : struct { }

2.与接口约束同时使用，在最前面(不能与基类约束,构造函数约束一起使用)

public class MyClass11<T> where T : struct, IComparable { }

## 引用约束

常见的

public class MyClass10<T> where T : class { }

## 多个泛型参数

 public class MyClass12<T, U> where T : IComparable  where U : class { }

## 二、 继承和泛型

public class B<T>{ }

1. 在从泛型基类派生时,可以提供类型实参,而不是基类泛型参数

    public class SubClass11 : B<int>  
    { }

2.如果子类是泛型,而非具体的类型实参,则可以使用子类泛型参数作为泛型基类的指定类型

    public class SubClass12<R> : B<R>  
    { }

3.在子类重复基类的约束(在使用子类泛型参数时,必须在子类级别重复在基类级别规定的任何约束)  
    public class B<T> where T : ISomeInterface { }  
    public class SubClass2<T> : B<T> where T : ISomeInterface { }

4.构造函数约束  
    public class B<T> where T : new()  
    {  
        public T SomeMethod()  
        {  
            return new T();  
        }  
    }  
    public class SubClass3<T> : B<T> where T : new(){ }

## 三、泛型方法

(C#2.0泛型机制支持在"方法声名上包含类型参数",这就是泛型方法)

1.泛型方法既可以包含在泛型类型中,又可以包含在非泛型类型中

public class MyClass5  
    {

        public void MyMethod<T>(T t){ }  
    }

2.泛型方法的声明与调用

[复制代码](javascript:void(0);)

public class MyClass5

{

public void MyMethod<T>(T t){ }

}

public class App5

{

public void CallMethod()

{

MyClass5 myclass5 = new MyClass5();

myclass5.MyMethod<int>(3);

}

}

[复制代码](javascript:void(0);)

3.泛型方法的重载

[复制代码](javascript:void(0);)

//第一组重载

void MyMethod1<T>(T t, int i){ }

void MyMethod1<U>(U u, int i){ }

//第二组重载

void MyMethod2<T>(int i){ }

void MyMethod2(int i){ }

//第三组重载，假设有两个泛型参数

void MyMethod3<T>(T t) where T : A { }

void MyMethod3<T>(T t) where T : B { }

//第四组重载

public class MyClass8<T,U>

{

public T MyMothed(T a, U b)

{

return a;

}

public T MyMothed(U a, T b)

{

return b;

}

public int MyMothed(int a, int b)

{

return a + b;

}

}

[复制代码](javascript:void(0);)

4.泛型方法的覆写

(1)public class MyBaseClass1  
    {  
        public virtual void MyMothed<T>(T t) where T : new() { }  
    }  
    public class MySubClass1:MyBaseClass1  
    {  
        public override void MyMothed<T>(T t) //不能重复任何约束  
        { }  
    }

(2)public class MyBaseClass2  
    {  
        public virtual void MyMothed<T>(T t)  
        { }  
    }  
    public class MySubClass2 : MyBaseClass2  
    {  
        public override void MyMothed<T>(T t) //重新定义泛型参数T  
        { }  
    }

## 四、虚拟方法

[复制代码](javascript:void(0);)

public class BaseClass4<T>

{

public virtual T SomeMethod()

{

return default(T);

}

}

public class SubClass4 : BaseClass4<int> //使用实参继承的时候方法要使用实参的类型

{

public override int SomeMethod()

{

return 0;

}

}

public class SubClass5<T> : BaseClass4<T> //使用泛型继承时,方法也是泛型

{

public override T SomeMethod()

{

return default(T);

}

}

[复制代码](javascript:void(0);)

## 五、泛型参数隐式强制转换

编译器只允许将泛型参数隐式强制转换到 Object 或约束指定的类型。

[复制代码](javascript:void(0);)

class MyClass<T> where T : BaseClass, ISomeInterface

{

void SomeMethod(T t)

{

ISomeInterface obj1 = t;

BaseClass obj2 = t;

object obj3 = t;

}

}

[复制代码](javascript:void(0);)

变通方法:使用临时的 Object 变量，将泛型参数强制转换到其他任何类型

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class MyClass2<T>  {      void SomeMethod(T t)      {          object temp = t;          BaseClass obj = (BaseClass)temp;      }  } |

## 六、 泛型参数显式强制转换

编译器允许您将泛型参数显式强制转换到其它任何接口，但不能将其转换到类

[复制代码](javascript:void(0);)

class MyClass1<T>

{

void SomeMethod(T t)

{

ISomeInterface obj1 = (ISomeInterface)t;

//BaseClass obj2 = (BaseClass)t; //不能通过编译

}

}

[复制代码](javascript:void(0);)

## 七、 泛型参数强制转换到其他任何类型

使用临时的 Object 变量，将泛型参数强制转换到其他任何类型

[复制代码](javascript:void(0);)

class MyClass2<T>

{

void SomeMethod(T t)

{

object temp = t;

BaseClass obj = (BaseClass)temp;

}

}

[复制代码](javascript:void(0);)

## 八、使用is和as运算符

[复制代码](javascript:void(0);)

public class MyClass3<T>

{

public void SomeMethod(T t)

{

if (t is int) { }

if (t is LinkedList<int>) { }

string str = t as string;

if (str != null) { }

LinkedList<int> list = t as LinkedList<int>;

if (list != null) { }

}

}

**使用约束的原因**

如果要检查泛型列表中的某个项以确定它是否有效，或者将它与其他某个项进行比较，则编译器必须在一定程度上保证它需要调用的运算符或方法将受到客户端代码可能指定的任何类型参数的支持。这种保证是通过对泛型类定义应用一个或多个约束获得的。例如，基类约束告诉编译器：仅此类型的对象或从此类型派生的对象才可用作类型参数。一旦编译器有了这个保证，它就能够允许在泛型类中调用该类型的方法。约束是使用上下文关键字 **where** 应用的。下面的代码示例演示可通过应用基类约束添加到 GenericList<T> 类（在[泛型介绍（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/0x6a29h6.aspx)中）的功能。

public class Employee

{

private string name;

private int id;

public Employee(string s, int i)

{

name = s;

id = i;

}

public string Name

{

get { return name; }

set { name = value; }

}

public int ID

{

get { return id; }

set { id = value; }

}

}

public class GenericList<T> where T : Employee

{

private class Node

{

private Node next;

private T data;

public Node(T t)

{

next = null;

data = t;

}

public Node Next

{

get { return next; }

set { next = value; }

}

public T Data

{

get { return data; }

set { data = value; }

}

}

private Node head;

public GenericList() //constructor

{

head = null;

}

public void AddHead(T t)

{

Node n = new Node(t);

n.Next = head;

head = n;

}

public IEnumerator<T> GetEnumerator()

{

Node current = head;

while (current != null)

{

yield return current.Data;

current = current.Next;

}

}

public T FindFirstOccurrence(string s)

{

Node current = head;

T t = null;

while (current != null)

{

//The constraint enables access to the Name property.

if (current.Data.Name == s)

{

t = current.Data;

break;

}

else

{

current = current.Next;

}

}

return t;

}

}

约束使得泛型类能够使用 Employee.Name 属性，因为类型为 T 的所有项都保证是 Employee 对象或从 Employee 继承的对象。

可以对同一类型参数应用多个约束，并且约束自身可以是泛型类型，如下所示：

class EmployeeList<T> where T : Employee, IEmployee, System.IComparable<T>, new()

{

// ...

}

通过约束类型参数，可以增加约束类型及其继承层次结构中的所有类型所支持的允许操作和方法调用的数量。因此，在设计泛型类或方法时，如果要对泛型成员执行除简单赋值之外的任何操作或调用 **System.Object** 不支持的任何方法，您将需要对该类型参数应用约束。

在应用 where T : class 约束时，避免对类型参数使用 **==** 和 **!=** 运算符，因为这些运算符仅测试引用同一性而不测试值相等性。即使在用作参数的类型中重载这些运算符也是如此。下面的代码说明了这一点；即使 [String](https://msdn.microsoft.com/zh-cn/library/system.string.aspx) 类重载 **==** 运算符，输出也为 false。

public static void OpTest<T>(T s, T t) where T : class

{

System.Console.WriteLine(s == t);

}

static void Main()

{

string s1 = "target";

System.Text.StringBuilder sb = new System.Text.StringBuilder("target");

string s2 = sb.ToString();

OpTest<string>(s1, s2);

}

这种情况的原因在于，编译器在编译时仅知道 T 是引用类型，因此必须使用对所有引用类型都有效的默认运算符。如果必须测试值相等性，建议的方法是同时应用 where T : IComparable<T> 约束，并在将用于构造泛型类的任何类中实现该接口。

**约束多个参数**

可以对多个参数应用约束，并对一个参数应用多个约束，如下面的示例所示：

class Base { }

class Test<T, U>

where U : struct

where T : Base, new() { }

**未绑定的类型参数**

没有约束的类型参数（如公共类 SampleClass<T>{} 中的 T）称为未绑定的类型参数。未绑定的类型参数具有以下规则：

* 不能使用 **!=** 和 **==** 运算符，因为无法保证具体类型参数能支持这些运算符。
* 可以在它们与 **System.Object** 之间来回转换，或将它们显式转换为任何接口类型。
* 可以将它们与 [null](https://msdn.microsoft.com/zh-cn/library/edakx9da.aspx) 进行比较。将未绑定的参数与 **null** 进行比较时，如果类型参数为值类型，则该比较将始终返回 false。

**作为约束的类型参数**

将泛型类型参数作为约束使用，在具有自己类型参数的成员函数必须将该参数约束为包含类型的类型参数时非常有用，如下示例所示：

class List<T>

{

void Add<U>(List<U> items) where U : T {/\*...\*/}

}

在上面的示例中，T 在 **Add** 方法的上下文中是一个类型约束，而在 **List** 类的上下文中是一个未绑定的类型参数。

类型参数还可在泛型类定义中用作约束。请注意，必须在尖括号中声明此类型参数与任何其他类型的参数：

//Type parameter V is used as a type constraint.

public class SampleClass<T, U, V> where T : V { }

泛型类的类型参数约束的作用非常有限，因为编译器除了假设类型参数派生自 **System.Object** 以外，不会做其他任何假设。在希望强制两个类型参数之间的继承关系的情况下，可对泛型类使用参数类型约束。

**请参阅**

[System.Collections.Generic](https://msdn.microsoft.com/zh-cn/library/system.collections.generic.aspx)

[C# 编程指南](https://msdn.microsoft.com/zh-cn/library/67ef8sbd.aspx)

[泛型介绍（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/0x6a29h6.aspx)

[泛型类（C# 编程指南）](https://msdn.microsoft.com/zh-cn/library/sz6zd40f.aspx)

[new 约束（C# 参考）](https://msdn.microsoft.com/zh-cn/library/sd2w2ew5.aspx)