C#中的泛型和C++中的模板尽管都是泛型，但有本质的不同。

C#泛型和C++类模板的区别：参考：<http://blog.csdn.net/nabila/article/details/8133245>

C#的泛型更接近类，只是带有类型参数，而C++模板更接近宏。

C#的泛型在运行时进行实例化，而C++模板在运行之前进行实例化。

C#中的泛型是强类型的，在泛型中定义的方法必须满足所有的类型参数。而C++模板则允许定义只对某些类型参数有效的方法。

开放类型：具有泛型类型参数的类型

封闭类型：开放类型以外的类型都是封闭类型。

泛型实现类型安全：创建类型安全的代码，并在编译时捕获类型不匹配的错误。

例：程序generics\_test1

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test1

{

class TwoGen<T, V>

{

T obj1;

V obj2;

public TwoGen(T o1, V o2)

{

obj1 = o1;

obj2 = o2;

}

public void ShowType()

{

*Console*.*WriteLine*("Type of T is " + typeof(T));

*Console*.*WriteLine*("Type of V is " + typeof(V));

}

public T GetObj1()

{

return obj1;

}

public V GetObj2()

{

return obj2;

}

}

class GenericsTest

{

static void Main(string[] args)

{

TwoGen<int, string> obj = new TwoGen<int, string>(119, "Alpha");

obj.ShowType();

}

}

}

类型约束：可以限制用做类型实参的类型。

Class class-name<type-param> where type-param : constrains

其中constrains是一个以逗号分隔的约束列表。

C#定义的类型约束：

1. 基类约束：指定某个（可能有多个）类型实参必须继承的基类。

使用基类约束，允许在泛型类中使用有约束指定的基类所定义的成员。

例：程序generics\_test2

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型约束：基类约束

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test2

{

class A

{

public void Hello()

{

*Console*.*WriteLine*("Hello");

}

}

class B : A { }

class C { }

class Gen<T> where T : A

{

T obj;

public Gen(T o)

{

obj = o;

}

public void SayHello()

{

obj.Hello();

}

}

class GenericsTest

{

static void Main(string[] args)

{

A a = new A();

B b = new B();

C c = new C();

Gen<A> g1 = new Gen<A>(a);

g1.SayHello();

Gen<B> g2 = new Gen<B>(b);

g2.SayHello();

// error，C不是A的派生类

// Gen<C> g3 = new Gen<C>(c);

}

}

}

1. 接口约束：指某个类型参数必须实现的接口。

例：程序generics\_test3

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型类型约束：接口约束

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test3

{

class NotFoundException : *Exception*

{

public NotFoundException() : base() { }

public NotFoundException(string message) : base(message) { }

public NotFoundException(string message, *Exception* inner)

: base(message, inner) { }

protected NotFoundException(

*System*.*Runtime*.*Serialization*.*SerializationInfo* info,

*System*.*Runtime*.*Serialization*.*StreamingContext* context)

: base(info, context) { }

}

public interface IPhoneNumber

{

// 接口中的属性在语法上和自动实现属性类似，

// 但有本质的不同，这是抽象的属性

// 接口中的属性可以只有get或set

// 自动实现属性必须具有get和set

string Number

{

get;

set;

}

string Name

{

get;

set;

}

}

class Friend : IPhoneNumber

{

public Friend(string n, string num, bool wk)

{

Name = n;

Number = num;

IsWorkNumber = wk;

}

public bool IsWorkNumber { get; private set; }

public string Number { get; set; }

public string Name { get; set; }

}

class EmailFriend { }

class PhoneList<T> where T : IPhoneNumber

{

T[] list;

int end;

public PhoneList()

{

list = new T[10];

end = 0;

}

public bool Add(T new\_entry)

{

if (end == 10)

{

return false;

}

list[end] = new\_entry;

++end;

return true;

}

public T FindByName(string name)

{

for (int i = 0; i < end; ++i)

{

if (list[i].Name == name)

{

return list[i];

}

}

throw new NotFoundException();

}

public T FindByNumber(string number)

{

for (int i = 0; i < end; ++i)

{

if (list[i].Number == number)

{

return list[i];

}

}

throw new NotFoundException();

}

}

class GenericsTest

{

static void Main(string[] args)

{

PhoneList<Friend> flist = new PhoneList<Friend>();

flist.Add(new Friend("Tom", "555-1234", true));

flist.Add(new Friend("Matt", "555-9527", false));

try

{

Friend frnd = flist.FindByName("Tom");

*Console*.*WriteLine*(frnd.Name + ": " + frnd.Number);

Friend frnd1 = flist.FindByName("Gary");

*Console*.*WriteLine*(frnd1.Name + ": " + frnd1.Number);

} catch (NotFoundException)

{

*Console*.*WriteLine*("Not Found");

}

// error，EmailFriend没有继承IPhoneNumber

// PhoneList<EmailFriend> elist = new PhoneList<EmailFriend>();

}

}

}

输出结果为：

Tom: 555-1234

Not Found

1. new()构造函数约束

类型实参必须提供一个无参数的公有构造函数（可以是自动提供的默认构造函数）。

new()约束可以和其他约束一起使用，但必须位于约束列表的末端。

new()约束只允许使用无参数构造函数构造对象，不允许给类型形参的构造函数传递实参。

new()约束不能同时和值类型约束一起使用。

例：程序generics\_test4

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型约束：new()约束

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace generics\_test4

{

class MyClass

{ }

class MyClass2

{

public MyClass2()

{

}

public MyClass2(int i)

{

}

}

class Gen<T> where T : new()

{

T obj;

public Gen()

{

obj = new T();

}

}

class GenericsTest

{

static void Main(string[] args)

{

// MyClass有默认的构造函数

Gen<MyClass> x = new Gen<MyClass>();

// MyClass2必须显示提供一个无参构造函数

Gen<MyClass2> y = new Gen<MyClass2>();

}

}

}

1. 引用类型约束和值类型约束

引用类型约束：where T : class

值类型约束：where T : struct

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型约束：引用类型约束和值类型约束

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace generics\_test5

{

class MyClass { }

struct MyStruct { }

class Gen<T> where T : class

{

T obj;

public Gen()

{

obj = null;

}

}

class Gen1<T> where T : struct

{

T obj;

public Gen1(T x)

{

obj = x;

}

}

class GenericsTest

{

static void Main(string[] args)

{

Gen<MyClass> x = new Gen<MyClass>();

// error，类型实参必须为引用类型

// Gen<int> x1 = new Gen<int>();

Gen1<int> y = new Gen1<int>(5);

Gen1<MyStruct> y1 = new Gen1<MyStruct>(new MyStruct());

// error，类型实参必须为值类型

// Gen1<MyClass> y2 = new Gen1<MyClass>(new MyClass());

}

}

}

类型形参的默认值:default(T)

例：程序generics\_test6

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型：类型参数的默认值

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test6

{

class MyClass

{

}

class Gen<T>

{

public T obj;

public Gen()

{

// 类型参数的默认值

obj = default(T);

}

}

class GenericsTest

{

static void Main(string[] args)

{

Gen<MyClass> x = new Gen<MyClass>();

// 引用类型的默认值为null

if (x.obj == null)

{

*Console*.*WriteLine*("x.obj is null.");

}

Gen<int> y = new Gen<int>();

// 整型的默认值为0

if (y.obj == 0)

{

*Console*.*WriteLine*("y.obj is 0.");

}

Gen<bool> z = new Gen<bool>();

// bool型默认值为false

if (z.obj == false)

{

*Console*.*WriteLine*("z.obj is false.");

}

}

}

}

输出结果：

x.obj is null.

y.obj is 0.

z.obj is false.

C#允许创建泛型结构，语法与泛型类完全相同，而且也可以使用各种约束。

泛型方法：可以在非泛型类中创建泛型方法。

例：程序generics\_test7

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型方法

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test7

{

class ArrayUtils

{

// 非泛型类的泛型方法

public static bool CopyInsert<T>(T e, uint idx, T[] src, T[] target)

{

if (target.*Length* < src.*Length* + 1)

return false;

for (int i = 0, j = 0; i < src.*Length*;++i, ++j )

{

if (i == idx)

{

target[j] = e;

++j;

}

target[j] = src[i];

}

return true;

}

}

class GenericsTest

{

static void Main(string[] args)

{

int[] src = { 1, 2, 3};

int[] target = new int[4];

*Console*.*Write*("Contents of src: ");

foreach (int x in src)

*Console*.*Write*(x + " ");

*Console*.*WriteLine*();

// 泛型方法

ArrayUtils.CopyInsert(99, 2, src, target);

*Console*.*Write*("Contents of target: ");

foreach (int x in target)

*Console*.*Write*(x + " ");

*Console*.*WriteLine*();

}

}

}

输出结果：

Contents of src: 1 2 3

Contents of target: 1 2 99 3

从上例可以看出，调用泛型方法的语法可以和普通方法相似，会自动进行类型推断。

在调用泛型方法时，也可以显式指定类型形参：

ArrayUtils.CopyInsert<int>(99, 2, src, target);

可以为泛型方法指定约束，与泛型类的约束类似。

public static bool CopyInsert<T>(T e, uint idx, T[] src, T[] target) where T : class

{

该泛型方法只能处理引用类型。

泛型委托：

delegate ret-type delegate-name<type-parameter-list>(arg-list);

例：generics\_test8

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型委托

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test8

{

// 泛型委托

delegate T Op<T>(T v);

class MyClass

{

public int Sum(int v)

{

int result = 0;

for (int i = v; i > 0; --i)

result += i;

return result;

}

public string Reflect(string src)

{

string result = "";

foreach (char ch in src)

result = ch + result;

return result;

}

}

class GenericsTest

{

static void Main(string[] args)

{

MyClass obj = new MyClass();

Op<int> del\_int = obj.Sum;

*Console*.*WriteLine*(del\_int(3));

Op<string> del\_str = obj.Reflect;

*Console*.*WriteLine*(del\_str("Hello"));

}

}

}

泛型接口：语法与泛型类相似。

实现泛型接口的类本身必须是泛型的。

泛型类的继承：泛型基类所需的类型实参必须通过派生类沿着层次结构向上传递。

例：程序generics\_test9

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型类的继承

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test9

{

class Gen<T>

{

T obj;

public Gen(T o)

{

obj = o;

}

public T GetObj()

{

return obj;

}

}

// 派生类可以有自己自有的类型参数

// 但必须给泛型基类传递基类的类型参数

class Gen2<T, V> : Gen<T>

{

V obj;

public Gen2(T o, V o2) : base(o)

{

obj = o2;

}

public V GetObj2()

{

return obj;

}

}

class GenericsTest

{

static void Main(string[] args)

{

Gen2<string, int> x = new Gen2<string, int>("Value is: ", 99);

*Console*.*WriteLine*(x.GetObj());

*Console*.*WriteLine*(x.GetObj2());

*Console*.*WriteLine*();

Gen<string> base\_x = x;

*Console*.*WriteLine*(base\_x.GetObj());

// error，基类的类型实参与派生类的类型实参不一致

// Gen<int> base\_y = x;

}

}

}

输出结果

Value is:

99

Value is:

非泛型类可以作为泛型类的基类。

例：程序generics\_test10

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型继承：非泛型类为基类

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test10

{

class NonGen

{

int num;

public NonGen(int i)

{

num = i;

}

public int GetNum()

{

return num;

}

}

// 泛型类继承非泛型类

class Gen<T> : NonGen

{

T obj;

public Gen(T o, int i) : base(i)

{

obj = o;

}

public T GetObj()

{

return obj;

}

}

class GenericsTest

{

static void Main(string[] args)

{

Gen<string> w = new Gen<string>("Hello", 47);

*Console*.*Write*(w.GetObj() + " ");

*Console*.*WriteLine*(w.GetNum());

}

}

}

重载使用类型形参的方法：

C#普通接口没有逆变和协变。

在C#中的泛型编程中，协变和逆变只能用于泛型接口和泛型委托。

协变和逆变是互斥的，泛型参数如果可以逆变，就不能协变，如果可以协变，就不能逆变。

泛型接口中的协变：

接口中的方法可以返回派生自类型形参的类的类型。

例：程序generics\_test11

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型接口的协变

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test11

{

// 协变的泛型接口，使用out关键字

public interface IMyCoVarGen<out T>

{

T GetObject();

//error， T为协变，不能用于方法形参

// void SetObject(T obj);

}

// 继承协变的泛型接口，接口中不再用

// out关键字，泛型类中也不能用out，

// 协变只能在泛型接口和泛型委托

class MyClass<T> : IMyCoVarGen<T>

{

T obj;

public MyClass(T v) { obj = v; }

public T GetObject() { return obj; }

}

class Alpha

{

string name;

public Alpha(string n) { name = n; }

public string GetName() { return name; }

}

class Beta : Alpha

{

public Beta(string n) : base(n) { }

}

class GenericsTest

{

static void Main(string[] args)

{

IMyCoVarGen<Alpha> alpha\_ref

= new MyClass<Alpha>(new Alpha("Alpha #1"));

*Console*.*WriteLine*("Name of object is " + alpha\_ref.GetObject().GetName());

// 因为接口IMyCoVarGen是协变，所以可以这样赋值

alpha\_ref = new MyClass<Beta>(new Beta("Beta #1"));

*Console*.*WriteLine*("Name of object is " + alpha\_ref.GetObject().GetName());

MyClass<Alpha> alpha = new MyClass<Alpha>(new Alpha("Alpha #2"));

MyClass<Beta> beta = new MyClass<Beta>(new Beta("Beta #2"));

// error，泛型类没有协变和逆变

// alpha = beta;

}

}

}

输出结果为：

Name of object is Alpha #1

Name of object is Beta #1

协变类型形参的接口也可以被其它接口继承。

public interface IMyCoVarGen2<out T> : IMyCoVarGen<T> {}

IMyCoVarGen<T>是协变类型接口，IMyCoVarGen2<T>也是协变类型接口。在继承时，如果IMyCoVarGen2不加out关键字，则就不再是协变类型接口，同时也无法使用IMyCoVarGen的协变特性。

例：程序generics\_test12

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型接口的协变

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test12

{

public interface IMyCoVarGen<out T>

{

T GetObject();

}

public interface IMyCoVarGen2<T> : IMyCoVarGen<T>

{

void ShowType();

}

class MyClass<T> : IMyCoVarGen2<T>

{

T obj;

public MyClass(T v) { obj = v; }

public T GetObject() { return obj; }

public void ShowType()

{

*Console*.*WriteLine*("T is " + typeof(T));

}

}

class Alpha

{

string name;

public Alpha(string n) { name = n; }

public string GetName() { return name; }

}

class Beta : Alpha

{

public Beta(string n) : base(n) { }

}

class GenericsTest

{

static void Main(string[] args)

{

IMyCoVarGen2<Alpha> alpha\_ref

= new MyClass<Alpha>(new Alpha("Alpha #1"));

*Console*.*WriteLine*("Name of object is " + alpha\_ref.GetObject().GetName());

// error，IMyCoVarGen2不是协变，

// 也无法扩展IMyCoVarGen协变特性

// alpha\_ref = new MyClass<Beta>(new Beta("Beta #1"));

}

}

}

泛型接口中的协变形参只能用于方法返回类型。out关键字不能用于声明方法形参的类型形参。

协变值适用于引用类型，对值类型无效。

协变类型形参不能做接口方法中的约束。

public interface IMyCoVarGen<out T> {

// void Method<V> () where V:T // error，协变形参不能作为方法约束

}

在泛型接口中使用逆变：用in关键字声明类型形参

例：程序generics\_test13

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型接口的逆变

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test13

{

public interface IMyContraVarGen<in T>

{

// 逆变类型形参只能应用于方法实参

void Show(T obj);

}

class MyClass<T> : IMyContraVarGen<T>

{

public void Show(T x)

{

// 会自动调用T的ToString()方法

*Console*.*WriteLine*(x);

}

}

class Alpha

{

// 重写object中的ToString()

public override string ToString()

{

return "This is an Alpha object.";

}

}

class Beta : Alpha

{

public override string ToString()

{

return "This is a Beta object.";

}

}

class GenericsTest

{

static void Main(string[] args)

{

IMyContraVarGen<Alpha> alpha\_ref = new MyClass<Alpha>();

IMyContraVarGen<Beta> beta\_ref = new MyClass<Beta>();

// 因为逆变，所以可以给接口的引用的类型参数赋一个基类类型

IMyContraVarGen<Beta> beta\_ref2;

beta\_ref2 = new MyClass<Alpha>();

beta\_ref.Show(new Beta());

// 在断点调试时，看到beta\_ref2的类型形参T的实际类型为Alpha

// Show方法能接收一个Beta对象是因为T是逆变，Beta可以转换为Alpha

// 但不清楚Show方法不能接收一个Alpha对象？？

beta\_ref2.Show(new Beta());

// error，在编译时报错beta\_ref2在编译时T为beta;

// beta\_ref2.Show(new Alpha());

}

}

}

逆变只适用于引用类型，并且逆变类型形参只能（这里的“只能”表示，如果使用逆变类型形参就只能做方法实参，也可以不使用逆变类型形参）用于方法实参。

既有协变，又有逆变：

例：程序generics\_test14

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型接口的协变和逆变

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace generics\_test14

{

// 既有协变，又有逆变的泛型接口

public interface IMyIF<out T, in V>

{

T GetObject();

void Show(V v\_obj);

}

public class MyClass<T, V> : IMyIF<T, V>

{

T obj;

public MyClass(T o) { obj = o; }

public T GetObject() { return obj; }

public void Show(V v\_obj) { *Console*.*WriteLine*(v\_obj); }

}

class AlphaT

{

string name;

public AlphaT(string n) { name = n; }

public string GetName() { return name; }

}

class BetaT : AlphaT

{

public BetaT(string n) : base(n) { }

}

class GenericsTest

{

static void Main(string[] args)

{

// IMyIF接口的T类型为协变类型

// 所以可以传递一个派生类类型

// IMyIF接口的V类型为逆变类型

// 所以可以传递一个基类类型

IMyIF<AlphaT, BetaT> alpha\_ref

= new MyClass<BetaT, AlphaT>(new BetaT("BetaT #1"));

*Console*.*WriteLine*("Name of object is " + alpha\_ref.GetObject().GetName());

alpha\_ref.Show(new BetaT("Beta T #2"));

// error，alpha\_ref的Show()接收BetaT对象

// alpha\_ref.Show(new AlphaT("AlphaT #1"));

}

}

}

泛型接口协变和逆变的相互作用：

例：程序generics\_test16

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型接口协变和逆变的相互作用

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace generics\_test16

{

// 逆变

public interface IFoo<in T>

{

}

// error

//public interface IBar<in T>

//{

// void Test(IFoo<T> foo);

//}

// 协变

public interface IBar<out T>

{

// IFoo必须逆变类型形参T

void Test(IFoo<T> foo);

}

class MyClass<T> : IBar<T>

{

public void Test(IFoo<T> foo) { }

}

class Foo<T> : IFoo<T> { }

class GenericsTest

{

static void Main(string[] args)

{

// 通过断点调试可以看到obj\_bar的T的实际类型为string

// Test方法的实参类型为IFoo<string>，

// 通过obj\_bar调用Test方法，传递的实参为IFoo<object>

// 如果传递IFoo<string>会发生编译错误，这样就要求

// IFoo接口必须具有逆变类型形参，才能从IFoo<object>

// 转换成IFoo<string>

IBar<object> obj\_bar = new MyClass<string>();

IFoo<object> obj\_foo = new Foo<object>();

obj\_bar.Test(obj\_foo);

}

}

}

泛型委托：非泛型委托也支持方法返回类型的协变和方法参数类型的逆变。

例：程序generics\_test15

// Copyright 2016.刘珅珅

// author：刘珅珅

// 泛型委托的协变和逆变

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace generics\_test15

{

delegate bool Op<in T>(T obj);

// error，逆变类型形参只能作为方法形参

// delegate T Op1<in T>(T obj);

delegate T AnotherOp<out T, V>(V obj);

// error，协变类型形参只能为方法的返回值

// delegate T AnotherOp1<out T, V>(T o, V obj);

// OK，如果使用协变类型形参就必须做方法的返回值

// 也可以不用协变类型形参

// delegate bool AnotherOp1<out T, V>(V obj);

class Alpha

{

public int Val { get; set; }

public Alpha(int v) { Val = v; }

}

class Beta : Alpha

{

public Beta(int v) : base(v) { }

}

class GenericsTest

{

static bool IsEven(Alpha obj)

{

if ((obj.Val % 2) == 0)

return true;

return false;

}

static Beta ChangeIt(Alpha obj)

{

return new Beta(obj.Val + 2);

}

static void Main(string[] args)

{

Alpha objA = new Alpha(4);

Beta objB = new Beta(9);

Op<Alpha> del\_alpha = IsEven;

// 逆变

Op<Beta> del\_beta = del\_alpha;

Console.WriteLine(del\_alpha(objA));

Console.WriteLine(del\_beta(objB));

AnotherOp<Beta, Alpha> del\_another\_beta = ChangeIt;

// 协变

AnotherOp<Alpha, Alpha> del\_another\_alpha = del\_another\_beta;

objA = del\_another\_alpha(objA);

Console.WriteLine(objA.Val);

}

}

}

输出结果为：

True

False

6