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