7/16

笔记本: 暑期实习
创建时间: 2021/7/16 10:04 **更新时间:** 2021/7/20 23:44
作者: 134exetj717
URL: http://wiki.suncaper.net/pages/viewpage.action?pageId=39860218

- 泛型：指通过参数化类型来实现在同一份代码上操作多种数据类型。
    - 什么是泛型？：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。

```
数据的抽象：
int、double数据
int MAX(int a, int b)
{
    return a > b ? a : b;
}
double MAX1(double a, double b)
{
    return a > b ? a : b;
}

重载
int MAX(int a, int b)
{
    return a > b ? a : b;
}
double MAX(double a, double b)
{
    return a > b ? a : b;
}

泛型
数据类型的抽象：
T MAX<T>(T a, T b)
{
    return a > b ? a : b;
}
```

    - 定义泛型的语法格式：

```
[访问修饰符][返回类型]泛型名称<类型参数列表>
```

    - 常用泛型类和泛型方法：

```
class Stack<T>                    //声明泛型类
{
```

```
    T data[MaxSize];
    int top;
    //...
}
void swap<T>(ref T a, ref T b)     //定义泛型方法
{
    T tmp = a;
    a = b;
    b = tmp;
}

调用方法:
Stack<Teacher> s4 = new Stack<Teacher>();//定义教师栈
```

- 泛型的类型参数约束
    1. 使用泛型的过程中，可能会存在两个参数类型不一致，导致运行泛型表达式的过程中出现错误：比如字符串和整数比较大小
    2. 约束是使用上下文关键字where应用的。一般格式如下：

```
where 类型参数：约束1，约束2......
```

- 可为任意类型参数指定任意数量的接口约束，但类类型约束只能限定一个（如果类可实现任意数量的接口，但只从一个类派生）。所有约束都在一个以逗号分隔的列表中声明。约束列表跟在泛型类型名称和一个冒号之后。如果有多个类型参数，每个类型参数前面都要使用where关键字。

```
例如，以下泛型有3个类型参数，T1是未绑定的（没有约束），T2只有MyClass1类型或从它派生的类或MyClass2类型或从它派生的类才能用作类型实参，T3只有MyClass3类型或从它派生的类才能用作类型实参：
class MyClass<T1, T2, T3>
    where T2 : MyClass1, MyClass2
    where T3 : MyClass3
{
    //  …
}
```

- 泛型的继承
    - C#中除了单独声明泛型类型，还可以在基类中声明泛型类型，但如果基类是泛型，要么已实例化，要么来源于子类（同样是泛型类型）声明的类型参数

```
例如，若声明了如下泛型：
class C<U,V>
{
       //…
}

则以下声明是正确的：
class D:C<string,int>    //继承的类型已实例化
{
      // …
}
class E<U,V>:C<U,V>  //E类型为C类型提供了U、V，即来源于子类
{
      // …
}
class F<U,V>:C<string,int>     //F类型继承于C<string,int>，可看成F继承一个非泛型的类
{
      // …
}

而以下声明是错误的：
class G:C<U,V> //因为G类型不是泛型，C是泛型，G无法给C提供泛型的实例化
```

```
{
      // …
}
```

- 泛型委托

```
delegate bool MyDelegate<T>(T value);
class MyClass
{
    static bool method1(int i) { … }
    static bool method2(string s) { … }
    static void Main()
    {
        MyDelegate<string> p2 = method2;
        MyDelegate<int> p1 = new MyDelegate<int>(method1);
    }
}
```

- 枚举器概述

```
例如，有以下代码：
int[] myarr = { 1, 2, 3, 4, 5 };
foreach (int item in myarr)
    Console.Write("{0} ", item);
Console.WriteLine();

其输出是：1，2，3，4，5。为什么会这样呢？
这是因为数组可以按需提供一个称为枚举器（enumerator，或枚举数）的对象。
枚举器可用于依次读取数组中的元素，但不能用于修改基础集合，所以，不能用迭代变量（或枚举变量）item修改myarr的元素。
Array类有一个GetEnumerator方法用于返回当前使用的枚举器，除了Array类外，还有一些其他类型提供了GetEnumerator方法，凡是提供了GetEnumerator方法的类型称为可枚举类型，显然，数组是可枚举类型。
```

- IEnumerator接口

枚举器是实现IEnumerator接口的类对象。

IEnumerator接口支持对非泛型集合的简单迭代，是所有非泛型枚举器的基接口，它位于命名空间 System.Collections中。IEnumerator 接口有如下public成员：

- Current属性：获取集合中的当前元素。
- MoveNext方法：将枚举器推进到集合的下一个元素。
- Reset方法：将枚举器设置为其初始位置，该位置位于集合中第一个元素之前。

最初，枚举器被定位于集合中**第一个元素的前面**。Reset 方法用于将枚举器返回到此位置。在此位置上，未定义 Current。 因此，**在读取 Current 的值之前，必须调用 MoveNext将枚举数定位到集合的第一个元素。**

再次调用MoveNext方法将 Current属性定位到下一个元素。**如果 MoveNext越过集合的末尾，则枚举器将定位到集合中最后一个元素的后面，而且MoveNext返回 false。**

对于前面的foreach语句的代码，其执行过程如下：

①调用arr.GetEnumerator()返回一个IEnumerator引用。

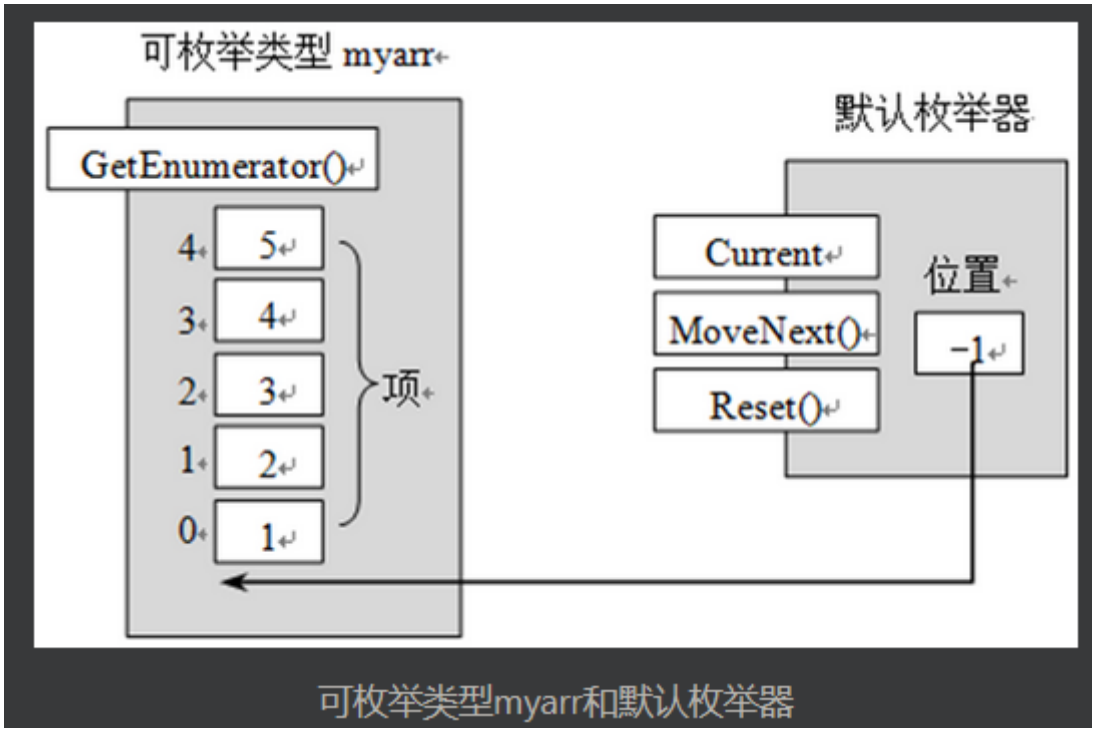
②调用所返回的IEnumerator接口的MoveNex方法。

③如果MoveNex方法返回true，就使用IEnumerator接口的属性来获取arr的一个元素，用于foreach循环。

④重复②和③的步骤，直到MoveNex方法返回false为止，此时循环停止。

前面foreach语句代码的功能与以下代码是相同的：

```
int[] myarr = { 1, 2, 3, 4, 5 };
Enumerator ie = myarr.GetEnumerator();
while (ie.MoveNext())
    Console.Write("{0} ",ie.Current);
Console.WriteLine();
```



- IEnumerable接口

可枚举类型是指提供了GetEnumerator方法的类型，而GetEnumerator方法是IEnumerable接口的成员，因此**可枚举类型是指实现了IEnumerable接口的类型。**

IEnumerable接口支持在非泛型集合上进行简单迭代，它位于 System.Collections命名空间。IEnumerable接口只有一个public成员，即GetEnumerator方法，用于返回一个循环访问集合的枚举器IEnumerator，而

IEnumerator 可以通过集合循环显示 Current 属性和 MoveNext 和 Reset 方法。

【例】设计一个学生类Student和一个People类，People类包含若干学生对象，通过从IEnumerable 接口继承使其成为可枚举类型，并设计相应的枚举器类PeopleEnum（从IEnumerator接口继承）。最后用foreach语句对People类对象执行枚举。

```
using System;
using System.Collections;
namespace proj8_1
{
    public class Student                //声明Student类
    {
        public int id;                      //学号
        public string name;                 //姓名
        public Student(int id, string name)     //构造函数
        {
            this.id = id;
```

```
            this.name = name;
        }
    }
    public class People : IEnumerable           //声明可枚举类
    {
        private Student[] sts;                  //sts为Student对象数组
        public People(Student[] pArray)         //People类的构造函数,创建sts
        {
            sts = new Student[pArray.Length];
            for (int i = 0; i < pArray.Length; i++)
                sts[i] = pArray[i];
        }
        IEnumerator IEnumerable.GetEnumerator()//实现IEnumerable的GetEnumerator方
法
        {
            return (IEnumerator)GetEnumerator();
            //调用People类的GetEnumerator方法，并将结果转换为枚举器对象
        }
        public PeopleEnum GetEnumerator()      //定义People类的GetEnumerator
        {
            return new PeopleEnum(sts);
        }
    }
    public class PeopleEnum : IEnumerator       //声明枚举器类
    {
        public Student[] sts;
        int position = -1;                          //位置字段，初始为-1
        public PeopleEnum(Student[] list)       //构造函数
        {
            sts = list;
        }
        public bool MoveNext()    //定义PeopleEnum的MoveNext方法
        {
            position++;
            return (position < sts.Length);
        }
        public void Reset()                 //定义PeopleEnum的Reset方法
        {
            position = -1;
        }
        object IEnumerator.Current   //实现IEnumerator的Current属性
        {
            get
            { return Current; }      //返回PeopleEnum的Current属性
        }
        public Student Current               //定义PeopleEnum的Current属性
        {
            get
            { return sts[position]; }  //返回sts中position位置的Student对象
        }
    }
    class Program
    {
        static void Main()
        {
            Student[] starry = new Student[4]
               {   new Student(1, "Smith"),new Student(2, "Johnson"),
                           new Student(3,"Mary"),new Student(4,"Hammer") };
            People peopleList = new People(starry);
            foreach (Student p in peopleList)
                Console.WriteLine(p.id.ToString() + " " + p.name);
        }
    }
}
```

- 泛型枚举接口

对于**非泛型接口**版本：

- IEnumerable接口的GetEnumerator方法返回实现IEnumerator枚举器类的实例。
- 实现IEnumerator枚举器的类实现了Current属性，它返回object的引用，然后需要把它转换为实际类型的对象。

对于**泛型接口**版本：

- IEnumerable<T>接口的GetEnumerator方法返回实现IEnumerator<T>枚举器类的实例。IEnumerable<T> 是从 IEnumerable继承的。
- 实现IEnumerator<T>枚举器的类实现了Current属性，它返回实际类型的对象引用，不需要进行转换操作。IEnumerator<T>是从Enumerator继承的。

- 迭代器
  - **迭代器（iterator）**也是用于对集合如列表和数组等进行迭代，它是一个代码块，按顺序提供要在foreach循环中使用的所有值。
  - 一般情况下，这个代码块是一个方法，称为迭代器方法，也可以使用含 get访问器的属性来实现。这里的迭代器方法或 get 访问器使用yield语句产生在foreach循环中使用的值。
  - yield 语句的如下两种形式：
  - yield return 表达式：使用一个 yield return 语句一次返回一个元素。foreach 循环的每次迭代都调用迭代器方法。 当 遇到迭代器方法中的一个yield return 语句时，返回“表达式”的值，并且保留代码的当前位置。当下次调用迭代器方法时从该位置重新启动。
  - yield break：使用 yield break 语句结束迭代。
  - 在迭代器的迭代器方法或 get访问器中，yield return 语句的“表达式”类型必须能够隐式转换到迭代器返回类型。迭代器方法或 get 访问器的声明必须满足以下要求：
  - 返回类型必须是 IEnumerable、IEnumerable<T>、IEnumerator 或 IEnumerator<T>。
  - 该声明不能有任何 ref 或out 参数。

迭代器方法不是在同一时间执行所有语句，它只是描述了希望编译器创建的枚举器的行为，也就是说，迭代器方法的代码描述了如何迭代元素。
1. 用迭代器方法实现可枚举类型
通过以下项目tmp来说明采用迭代器方法实现可枚举类型的原理：

```
using System;
using System.Collections.Generic;
namespace tmp
{
    class Program
    {
        static void Main()
        {
            foreach (int number in SomeNumbers())
                Console.Write(number.ToString() + " ");
        }
        public static IEnumerable<int> SomeNumbers()    //迭代器方法
        {
            yield return 3;
            yield return 5;
            yield return 8;
        }
    }
}
```

【例】设计一个程序，采用迭代器方法实现可枚举类型的方式，输出5～18之间的所有偶数。

```
using System;
using System.Collections;
```

```
using System.Collections.Generic;
namespace Proj8_2
{
    class Program
    {
        public static IEnumerable<int> EvenSequence(int i, int j) //迭代器方法
        {
            for (int number = i; number <= j; number++)
            {
                if (number % 2 == 0)
                    yield return number;
            }
        }
        static void Main()
        {
            foreach (int number in EvenSequence(5, 18))
                Console.Write("{0} ", number);
            Console.WriteLine();
        }
    }
}
```

2. 用迭代器方法实现枚举器
可以采用创建枚举器的方式，即声明一个包含GetEnumerator方法的类，由GetEnumerator方法返回IEnumerator<int>，它通过调用迭代器方法来实现。前面的tmp项目采用迭代器方法实现枚举器如下：

```
using System;
using System.Collections;
using System.Collections.Generic;
namespace tmp
{
    class MyClass
    {
        public IEnumerator<int> GetEnumerator()
        {
            return itmethod();
        }              //返回枚举器
        public IEnumerator<int> itmethod()      //迭代器方法
        {
            yield return 3;
            yield return 5;
            yield return 8;
        }
    }
    class Program
    {
        static void Main()
        {
            MyClass s = new MyClass();
            foreach (int item in s)
                Console.Write("{0} ", item);
            Console.WriteLine();
        }
    }
}
```

【例】设计一个程序，采用迭代器方法实现枚举器的方式，输出100以内的素数。

```
using System;
using System.Collections;
using System.Collections.Generic;
namespace Proj8_3
{
    public class Primes
    {
        int n;
        public Primes(int n)            //构造函数
        {
            this.n = n;
        }
        public IEnumerator<int> GetEnumerator()
```

```
        {
            return itmethod();
        }              //返回枚举器
        public IEnumerator<int> itmethod()      //迭代器方法
        {
            int i, j; bool isprime;
            for (i = 3; i <= n; i++)
            {
                isprime = true;
                for (j = 2; j <= (int)Math.Floor(Math.Sqrt(i)); j++)
                {
                    if (i % j == 0)
                    {
                        isprime = false; break;
                    }
                }
                if (isprime) yield return i;
            }
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            int i = 1;
            Primes ps = new Primes(100);
            Console.WriteLine("100以内素数:");
            foreach (int item in ps)
            {
                Console.Write("{0,4:d}", item);
                if (i % 10 == 0)        //某输出10个整数换一行
                    Console.WriteLine();
                i++;
            }
            Console.WriteLine();
        }
    }
}
```

从上看出，当创建类的迭代器时，不必实现整个 IEnumerator 接口，当编译器检测到迭代器时，会自动生成 Current属性、MoveNext方法和 IEnumerator 或 IEnumerator<T> 接口的 Dispose 方法。
注意迭代器不支持 IEnumerator.Reset 方法，如果要重置迭代器，必须获取新的迭代器。因此使用迭代器方法大大简化了可枚举类型和枚举器的设计过程。