C# 点阵列（ BitArray ） C# 匿名方法

C# 泛型（ Generic ）

泛型（ Generic ） 允许您延迟编写类或方法中的编程元素的数据类型的规范，直到实际在程序中使用它的时候。换句话说，泛型允许您编写一个可以与任何数据类型一起工作的类或方法。

您可以通过数据类型的替代参数编写类或方法的规范。当编译器遇到类的构造函数或方法的函数调用时，它会生成代码来处理指定的数据类型。下面这个简单的实例将有助于您理解这个概念：

实例

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

namespace GenericApplication
{
    public class MyGenericArray<T>
    {
        private T[] array;
        public MyGenericArray(int size)
        {
            array = new T[size + 1];
        }
        public T getItem(int index)
        {
            return array[index];
        }
        public void setItem(int index, T value)
        {
            array[index] = value;
        }
    }

    class Tester
    {
        static void Main(string[] args)
        {
            // 声明一个整型数组
            MyGenericArray<int> intArray = new MyGenericArray<int>(5);
            // 设置值
            for (int c = 0; c < 5; c++)
            {
                intArray.setItem(c, c*5);
            }
            // 获取值
            for (int c = 0; c < 5; c++)
            {
                Console.Write(intArray.getItem(c) + " ");
            }
            Console.WriteLine();
            // 声明一个字符数组
```

```
            MyGenericArray<char> charArray = new MyGenericArray<char>(5);
            // 设置值
            for (int c = 0; c < 5; c++)
            {
                charArray.setItem(c, (char)(c+97));
            }
            // 获取值
            for (int c = 0; c < 5; c++)
            {
                Console.Write(charArray.getItem(c) + " ");
            }
            Console.WriteLine();
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
0 5 10 15 20
a b c d e
```

泛型（Generic）的特性

使用泛型是一种增强程序功能的技术，具体表现在以下几个方面：

- 它有助于您最大限度地重用代码、保护类型的安全以及提高性能。
- 您可以创建泛型集合类。.NET 框架类库在 *System.Collections.Generic* 命名空间中包含了一些新的泛型集合类。您可以使用这些泛型集合类来替代 *System.Collections* 中的集合类。
- 您可以创建自己的泛型接口、泛型类、泛型方法、泛型事件和泛型委托。
- 您可以对泛型类进行约束以访问特定数据类型的方法。
- 关于泛型数据类型中使用的类型的信息可在运行时通过使用反射获取。

泛型（Generic）方法

在上面的实例中，我们已经使用了泛型类，我们可以通过类型参数声明泛型方法。下面的程序说明了这个概念：

实例

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

namespace GenericMethodAppl
{
    class Program
    {
        static void Swap<T>(ref T lhs, ref T rhs)
        {
            T temp;
```

```
            temp = lhs;
            lhs = rhs;
            rhs = temp;
        }
        static void Main(string[] args)
        {
            int a, b;
            char c, d;
            a = 10;
            b = 20;
            c = 'I';
            d = 'V';

            // 在交换之前显示值
            Console.WriteLine("Int values before calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values before calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);

            // 调用 swap
            Swap<int>(ref a, ref b);
            Swap<char>(ref c, ref d);

            // 在交换之后显示值
            Console.WriteLine("Int values after calling swap:");
            Console.WriteLine("a = {0}, b = {1}", a, b);
            Console.WriteLine("Char values after calling swap:");
            Console.WriteLine("c = {0}, d = {1}", c, d);
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Int values before calling swap:
a = 10, b = 20
Char values before calling swap:
c = I, d = V
Int values after calling swap:
a = 20, b = 10
Char values after calling swap:
c = V, d = I
```

泛型（Generic）委托

您可以通过类型参数定义泛型委托。例如：

```
delegate T NumberChanger<T>(T n);
```

下面的实例演示了委托的使用：

实例

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

delegate T NumberChanger<T>(T n);
namespace GenericDelegateAppl
{
    class TestDelegate
    {
        static int num = 10;
        public static int AddNum(int p)
        {
            num += p;
            return num;
        }

        public static int MultNum(int q)
        {
            num *= q;
            return num;
        }
        public static int getNum()
        {
            return num;
        }

        static void Main(string[] args)
        {
            // 创建委托实例
            NumberChanger<int> nc1 = new NumberChanger<int>(AddNum);
            NumberChanger<int> nc2 = new NumberChanger<int>(MultNum);
            // 使用委托对象调用方法
            nc1(25);
            Console.WriteLine("Value of Num: {0}", getNum());
            nc2(5);
            Console.WriteLine("Value of Num: {0}", getNum());
            Console.ReadKey();
        }
    }
}
```

当上面的代码被编译和执行时，它会产生下列结果：

```
Value of Num: 35
Value of Num: 175
```

← C# 点阵列（BitArray）

C# 匿名方法 →

2 篇笔记

写笔记

在声明泛型方法/泛型类的时候，可以给泛型加上一定的约束来满足我们特定的一些条件。

比如：

```
using System;
using System.Web.Caching;

namespace Demo.CacheManager
{
   public class CacheHelper<T> where T:new()
   {

   }
}
```

泛型限定条件：

- 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）
- 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

不吃梓梓 1年前 (2018-01-17)

在封装公共组件的时候，很多时候我们的类/方法不需要关注调用者传递的实体是"什么"，这个时候就可以使用泛型。

比如：

```
using System;
using System.Web.Caching;

namespace Xy.CacheManager
{
  public class CacheHelper<T>
  {
      //获取缓存实体
      public static T Get(Cache cache,string cacheKey)
      {
        //....缓存操作
      }
      //插入缓存
      public static void Set(Cache cache T tEntity,string cacheKey)
```

```
        {
            //....缓存操作
        }
    }
}
```

不吃梓梓 1年前 (2018-01-17)