Generic trong C# – Lập trình tổng quát, tham số hóa kiểu dữ liệu

Khi đọc tài liệu lập trình C#, bạn đã bao giờ gặp những lối viết ***lạ mắt*** chứa những chữ T, T1, T2, như List<T>, Swap<T>, Action<T1, T2>? Trong C#, những thứ có <T>, <T1, T2> đi đằng sau như vậy được gọi chung là **generic** (hay **generics**). Kỹ thuật lập trình như vậy gọi là **lập trình generic** hay lập trình tổng quát/lập trình khái quát.

Tôi đoán rằng, bạn hẳn đã từng làm việc với generic trong khi học lập trình C#. Ít nhất bạn cũng đã sử dụng kiểu List<T>. Nếu bạn từng nghe đến và sử dụng LINQ thì chắc chắn rằng bạn đã từng làm việc với generic. Chỉ có điều bạn chưa biết đến tên gọi của nó mà thôi.

Bạn có biết rằng, generics là loại kỹ thuật lập trình đặc biệt phổ biến và hữu dụng trong C#? Thực tế bạn có thể sử dụng generic với kiểu (type, class), phương thức (method), giao diện (interface), đại diện (delegate), tập hợp (collection), trong kế thừa. Không nắm kỹ về generic, bạn đã bỏ mất một công cụ đặc biệt mạnh mẽ và thông dụng trong C#.

Bài viết này sẽ cung cấp cho bạn những thông tin quan trọng nhất và đầy đủ nhất về lập trình generic trong C# .NET.

NỘI DUNG CỦA BÀI [Ẩn]

Lập trình generic trong C# là gì?

Trong C#, *lập trình tổng quát* (generic programming, generics), còn gọi là *lập trình khái quát*, là một dạng lập trình đặc biệt, trong đó **kiểu dữ liệu** (của biến thành viên, biến cục bộ, tham số, kiểu trả về của phương thức, v.v.) không được xác định trong giai đoạn xây dựng xây dựng đơn vị code (như class, phương thức, v.v.) mà chỉ được xác định ở giai đoạn khởi tạo và sử dụng.

Có thể hình dung bản chất của generics trong C# là **tham số hóa kiểu dữ liệu**. Nói cách khác, với generics, kiểu dữ liệu cũng là một tham số. Khi kiểu dữ liệu là một tham số, chúng ta có thể tạo ra class, phương thức, interface hay delegate để sử dụng với nhiều kiểu dữ liệu khác nhau mà không cần phải viết lại code cho từng kiểu dữ liệu riêng rẽ. Qua đó generics giúp tái sử dụng code hiệu quả.

Để thực hiện ý tưởng trên, ở giai đoạn ĐỊNH NGHĨA (KHAI BÁO), người ta dùng một **kiểu dữ liệu giả**. Ở giai đoạn SỬ DỤNG, kiểu dữ liệu giả này sẽ được thay thế bằng **kiểu dữ liệu thực**. Các chữ T, T1, T2 mà bạn có thể đã thấy chính là kiểu dữ liệu giả.

Chúng ta dùng từ "kiểu dữ liệu giả" ở đây để nghe cho dân dã. Thuật ngữ chính thức của nó là **tham số kiểu** (type parameter), đôi khi cũng được gọi đơn giản là **placeholder**. Kiểu giả không nhất thiết phải đặt là T mà có thể là bất kỳ ký tự/cụm ký tự nào. Tuy nhiên, người ta thường dùng nhất là T (viết tắt của Type) hoặc các ký tự ở cuối bảng chữ cái.

> Bạn có thể nhận ra sự tương tự giữa tham số của phương thức với tham số kiểu của generics. Tham số của phương thức cũng là một loại "giá trị giả" mà ở giai đoạn xây dựng phương thức bạn có thể sử dụng. Chỉ ở giai đoạn gọi phương thức, "giá trị thật" mới được truyền vào.

Khi nào nên sử dụng generics?

Khi lập trình, nếu gặp một trong hai tình huống dưới đây thì hãy nghĩ ngay đến generics:

Nếu có sự trùng lặp code về mặt logic và cách xử lý dữ liệu, chỉ khác biệt về kiểu dữ liệu: hãy nghĩ đến generics để tránh lặp code. Bạn sẽ gặp tình huống này ngay trong phần giới thiệu về generic method dưới đây.

Nếu lúc xây dựng class chưa xác định được kiểu dữ liệu của các biến thành viên, thuộc tính hoặc biến cục bộ (của phương thức) thì cần sử dụng lập trình generic. Bạn sẽ gặp tình huống này khi xem xét generic class ở phần sau.

Một số đặc điểm của generics trong C#

Dưới đây là tóm lược một số đặc điểm cần lưu ý khi sử dụng generics trong C#.

1. Phương thức/lớp tổng quát cho phép lựa chọn kiểu dữ liệu ở giai đoạn sử dụng, không phải ở giai đoạn định nghĩa;
2. Lập trình generic yêu cầu phải cung cấp một kiểu dữ liệu "giả" thay thế đặt trong cặp dấu <>; tên kiểu dữ liệu giả thường là một chữ cái in hoa nằm cuối bảng chữ cái (thông dụng nhất là T, U, V);
3. Trong code có thể sử dụng kiểu dữ liệu giả này tương tự như bất kỳ kiểu dữ liệu "thật" nào;
4. Số lượng kiểu dữ liệu giả không giới hạn; nếu có nhiều kiểu giả thì phân tách bởi dấu phẩy;
5. Có thể giới hạn kiểu giả (sẽ xem xét ở phần sau);

Đừng lo lắng nếu bạn chưa hiểu hết các vấn đề trên vì sau đây chúng ta sẽ đi vào từng ví dụ cụ thể. Sau khi đọc hết bài, bạn hãy quay lại đây một lần nữa nhé.

Generic được áp dụng cho các đối tượng nào trong C#?

Trong C#, generics có thể áp dụng cho: (1) class, (2) method, (3) interface, (4) delegate.

Tùy vào đối tượng áp dụng, kiểu giả của generic chỉ khác biệt về phạm vi tác dụng.

Đối với class và interface, kiểu giả tác dụng trong toàn bộ code của class. Kiểu giả này có thể sử dụng làm kiểu cho biến thành viên, thuộc tính, kiểu trả về của phương thức thành viên, kiểu tham số của phương thức thành viên.

Đối với method, kiểu giả chỉ có tác dụng trong phạm vi code của method đó. Nghĩa là kiểu giả có thể được sử dụng làm kiểu của biến cục bộ, kiểu trả về, kiểu tham số của method.

Đối với delegate, kiểu giả có thể sử dụng làm kiểu của tham số và kiểu kết quả trả về.

Kỹ thuật cụ thể với từng loại đối tượng mời bạn đọc tiếp ở các phần dưới đây.

Generic method trong C#

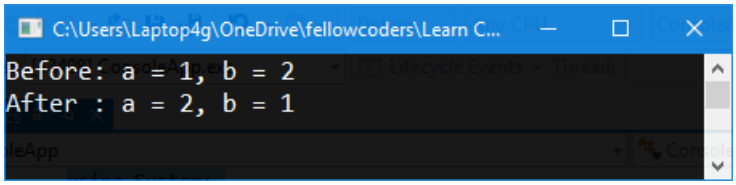
Ví dụ minh họa – Swap

Hẳn bạn đều biết loại phương thức Swap dùng để tráo giá trị của hai biến.

Giờ chúng ta cùng xây dựng một phương thức Swap đơn giản như sau:

```
1.  using System;
2.  namespace ConsoleApp
3.  {
4.      internal class Program
5.      {
6.          private static void Swap (ref int a, ref int b)
7.          {
8.              var temp = b;
9.              b = a;
10.             a = temp;
11.         }
12.         private static void Main(string[] args)
13.         {
14.             int a = 1, b = 2;
15.             Console.WriteLine($"Before: a = {a}, b = {b}");
16.             Swap(ref a, ref b);
17.             Console.WriteLine($"After : a = {a}, b = {b}");
18.             Console.ReadKey();
19.         }
20.     }
21. }
```

Trong ví dụ này chúng ta viết một phương thức Swap để tráo giá trị của hai biến kiểu int. Phương thức này được sử dụng trong phương thức Main cho kết quả như sau:



Kết quả chạy chương trình swap

Giả sử bạn cần hoán đổi giá trị của hai biến kiểu bool, bạn sẽ phải viết thêm một phương thức có code tương tự, chỉ thay duy nhất int bằng bool. Nếu muốn hoán đổi hai biến kiểu char (ký tự), bạn lại phải viết thêm một phương thức nữa tương tự.

Rõ ràng ở đây các phương thức có cùng một logic, chỉ khác biệt duy nhất kiểu dữ liệu mà nó xử lý. Trong tình huống này, generics là một lựa chọn hợp lý giúp chống lặp code.

Bây giờ chúng ta viết lại phương thức Swap ở trên theo cách sau:

```
1.  using System;
2.  namespace ConsoleApp
3.  {
4.      internal class Program
5.      {
6.          private static void Swap<T>(ref T a, ref T b)
7.          {
8.              var temp = b;
9.              b = a;
10.             a = temp;
11.         }
12.         private static void Main(string[] args)
13.         {
14.             int a = 1, b = 2;
15.             Console.WriteLine($"type = {a.GetType()}");
16.             Console.WriteLine($"Before: a = {a}, b = {b}");
17.             Swap(ref a, ref b);
18.             Console.WriteLine($"After : a = {a}, b = {b}");
19.
20.             bool aa = true, bb = false;
21.             Console.WriteLine($"type = {aa.GetType()}");
22.             Console.WriteLine($"Before: a = {aa}, b = {bb}");
23.             Swap(ref aa, ref bb);
24.             Console.WriteLine($"After : a = {aa}, b = {bb}");
25.             Console.ReadKey();
26.         }
27.     }
28. }
```

Đây là một ví dụ về cách xây dựng và sử dụng *generic method* trong C#.

Chúng ta có thể thấy cùng một phương thức Swap<T> giờ có thể dùng cho cả kiểu int và bool mà không cần viết lại cho mỗi kiểu dữ liệu cụ thể. Nói cách khác, bản thân kiểu dữ liệu của tham số giờ cũng lại là một loại tham số mà chúng ta có thể cung cấp khi gọi phương thức.

Cách lập trình generic method

Ở giai đoạn định nghĩa phương thức Swap<T> mới, chúng ta không biết được người dùng muốn sử dụng kiểu dữ liệu cụ thể nào. Vì vậy, chúng ta sử dụng một kiểu dữ liệu giả T. Cú pháp generic trong C# quy định kiểu giả phải đặt trong cặp dấu ngoặc <T>. Ở đây, hai biến phải có cùng kiểu dữ liệu, do đó Swap chỉ dùng một kiểu giả T. Nếu có nhiều kiểu giả, chúng ta có thể viết gộp vào cùng cặp ngoặc <T1, T2, T3>.

Tất cả các thao tác trên dữ liệu thuộc kiểu <T> này thực hiện giống như nó là một kiểu dữ liệu thực thụ.

Đối với generic method, kiểu giả chỉ có ý nghĩa và sử dụng được trong thân của phương thức. Ở phương thức Swap, chúng ta đã dùng kiểu T để khai báo biến tạm temp.

Kiểu giả có thể sử dụng làm kiểu của tham số. Hai tham số a và b của Swap<T> ở trên đều thuộc kiểu giả T. Ngoài ra, kiểu giả có thể sử dụng làm kiểu trả về của phương thức như bất kỳ kiểu dữ liệu bình thường nào.

Phương thức generic phải được ghi rõ với cặp dấu ngoặc với kiểu giả, Swap<T>, thay vì Swap. Bởi vì Swap<T> và Swap là hai phương thức hoàn toàn khác nhau.

Đến giai đoạn sử dụng Swap<T> người ta mới thay T bằng một kiểu dữ liệu cụ thể.

Nói tóm lại, generic method nên được xem xét sử dụng nếu:

1. Logic của các phương thức giống hệt nhau, chỉ khác biệt về kiểu dữ liệu: có thể chuyển đổi về generic method để tránh lặp code.
2. Ở giai đoạn định nghĩa phương thức chúng ta xác định phải sử dụng cho nhiều loại kiểu dữ liệu khác nhau.
3. Chưa xác định được kiểu dữ liệu cụ thể khi định nghĩa class.

Generic class trong C#

Ví dụ minh họa

Hãy cùng xem xét ví dụ sau đây:

```
using System;
namespace ConsoleApp
{
    class ListInt
    {
        private int[] _data;
        public int Count => _data.Length;
        public ListInt(int size) => _data = new int[size];
        public void Set(int index, int value)
        {
            if (index >= 0 && index < _data.Length) _data[index] = value;
        }
        public int Get(int index)
        {
            if (index >= 0 && index < _data.Length) return _data[index];
            return default(int);
        }
    }
    class ListChar
    {
        private char[] _data;
        public int Count => _data.Length;
        public ListChar(int size) => _data = new char[size];
        public void Set(int index, char value)
        {
            if (index >= 0 && index < _data.Length) _data[index] = value;
        }
        public char Get(int index)
        {
            if (index >= 0 && index < _data.Length) return _data[index];
```

```
            return default(char);
        }
    }
    internal class Program
    {
        private static void Main(string[] args)
        {
            var listInt = new ListInt(10);
            for (var i = 0; i < listInt.Count; i++)
                Console.Write($"{listInt.Get(i)}t");
            var listChar = new ListChar(10);
            for (var i = 0; i < listChar.Count; i++)
                Console.Write($"{listChar.Get(i)}t");
            Console.ReadKey();
        }
    }
}
```

Trong ví dụ trên chúng ta xây dựng hai class để "bao bọc" một mảng, đồng thời cung cấp phương thức để truy xuất mảng thay vì để người sử dụng trực tiếp truy xuất. Một lớp dành cho kiểu int, một lớp dành cho kiểu char.

Giả sử chúng ta cần xử lý thêm kiểu bool thì sẽ lại phải viết thêm một class riêng nữa.

Điều đáng lưu ý là logic của các class đều giống nhau. Sự khác biệt duy nhất nằm ở kiểu dữ liệu cụ thể mà class đó xử lý. Rõ ràng là có tình trạng lặp code ở đây.

Bây giờ chúng ta sẽ thay đổi code theo cách sau đây:

```
using System;
namespace ConsoleApp
{
    class List<T>
    {
        private T[] _data;
        public int Count => _data.Length;
        public List(int size) => _data = new T[size];
        public void Set(int index, T value)
        {
            if (index >= 0 && index < _data.Length) _data[index] = value;
        }
        public T Get(int index)
        {
            if (index >= 0 && index < _data.Length) return _data[index];
            return default(T);
        }
    }
    internal class Program
    {
        private static void Main(string[] args)
        {
            var listInt = new List<int>(10);
            for (var i = 0; i < listInt.Count; i++)
                Console.Write($"{listInt.Get(i)}t");
            var listChar = new List<char>(10);
            for (var i = 0; i < listChar.Count; i++)
                Console.Write($"{listChar.Get(i)}t");
            Console.ReadKey();
        }
    }
}
```

Nếu dịch và chạy thử cả hai đoạn code cho ra cùng một kết quả.

Đây là ví dụ về cách khai báo và sử dụng generic class.

Có thể nhận xét rằng, bản thân kiểu dữ liệu của biến cục bộ _data giờ cũng là một tham số, thay vì là một kiểu cố định. Giá trị của tham số kiểu này sẽ được cung cấp khi khởi tạo

object của class.

Cách lập trình generic class

So với generic method mà chúng ta đã xem xét ở phần trên, cách sử dụng kiểu giả đối với generic class là không khác biệt. Sự khác biệt lớn nhất nằm ở chỗ: phạm vi có ý nghĩa của kiểu giả bây giờ là toàn bộ class, thay vì chỉ trong một phương thức.

Như vậy chúng ta có thể thấy, nếu nhiều class có chung logic, chỉ khác biệt về một hoặc nhiều kiểu dữ liệu cần xử lý thì có thể viết một generic class thay cho viết nhiều class riêng rẽ. Nó sẽ giúp chúng ta tránh phải viết code lặp nhiều lần.

Hãy xem xét một góc nhìn khác: giả sử chúng ta phải xây dựng một lớp List để chứa một danh sách các giá trị để về sau chúng ta hoặc một lập trình viên khác sử dụng.

Tuy nhiên, lúc xây dựng lớp List này chúng ta muốn nó có khả năng chứa được nhiều kiểu dữ liệu khác nhau. Có những kiểu có thể tại thời điểm viết lớp List thậm chí còn chưa được định nghĩa! Generics là giải pháp cho tình huống này.

Nói tóm lại, lúc xây dựng lớp List chúng ta không xác định được kiểu dữ liệu của các phần tử sẽ chứa trong nó là gì. Khi đó, chúng ta nên nghĩ tới sử dụng generic.

> **TẠM KẾT**
>
> Qua phần trình bày về generic method và generic class chúng ta giờ có thể dễ dàng hình dung hơn về generics trong C# .NET: đó là sự tham số hóa kiểu dữ liệu.
> (1) Ở trong đơn vị code nào mà kiểu dữ liệu không xác định, hoặc không cố định, kiểu dữ liệu đó sẽ chuyển thành tham số, gọi là **tham số kiểu**.
> (2) Cách viết tham số kiểu tuân thủ cú pháp của C#, <T1, T2, T3, … >, và đứng ngay sau đơn vị code.
>
> Ngoài ra, việc áp dụng generics cho **interface** giống hệt như đối với class, áp dụng cho **delegate** giống như đối với method.

Giới hạn kiểu trong generic

Trong các ví dụ trên, kiểu giả T về sau có thể thay bằng bất kỳ kiểu dữ liệu nào, dù là kiểu có sẵn (built-in) hoặc kiểu do người dùng tự định nghĩa. Điều này có lợi là chúng ta về sau không chịu ràng buộc gì về kiểu dữ liệu thực.

Nhưng đi cùng với nó là những hạn chế khi dùng kiểu giả trong code.

Nếu bạn sử dụng intellisense sẽ thấy, biến thuộc kiểu T có đúng những phương thức và thuộc tính của kiểu Object!

> Có thể bạn đã biết, `Object` (hay `object`) là kiểu cha của mọi loại kiểu dữ liệu trong C#. Khi T không bị giới hạn, nó có thể nhận cả kiểu Object. Do vậy, chúng ta chỉ có thể sử dụng được các

> phương thức và thuộc tính tương tự như của Object trên biến kiểu T nếu T không bị giới hạn.

Giả sử bạn cần so sánh các biến thuộc kiểu T. Bạn thử code xem có được không? Tôi cam đoan là không được. Chúng ta không thể so sánh hai object bất kỳ trong C#.

Từ đây đặt ra yêu cầu về giới hạn kiểu giả T trong generic để biến thuộc kiểu giả này có những đặc điểm chúng ta mong muốn.

Nghe có vẻ lý thuyết quá phải không ạ! Hãy cùng thực hiện một ví dụ.

Ví dụ minh họa

Dưới đây là code cài đặt của thuật toán sắp xếp chọn (selection sort) trích ra từ khóa học Cấu trúc dữ liệu và giải thuật với C# trên site.

Dĩ nhiên, bạn không cần bận tâm về thuật toán này làm gì. Hãy xem cách sử dụng generic method thôi.

```
1.  using System;
2.  namespace P01_SelectionSort
3.  {
4.      class Program
5.      {
6.          static void Main(string[] args)
7.          {
8.              Console.Title = "Selection Sort";
9.              var numbers = new[] {10, 3, 1, 7, 9, 2, 0};
10.             Sort(numbers);
11.             Console.ReadKey();
12.         }
13.         static void Swap<T>(T[] array, int i, int m)
14.         {
15.             T temp = array[i];
16.             array[i] = array[m];
17.             array[m] = temp;
18.         }
19.         static void Print<T>(T[] array)
20.         {
21.             Console.WriteLine(string.Join("\t", array));
22.         }
23.         static void Sort<T>(T[] array) where T : IComparable
24.         {
25.             for (int i = 0; i < array.Length - 1; i++)
26.             {
27.                 int m = i;
28.                 T minValue = array[i];
29.                 for (int j = i + 1; j < array.Length; j++)
30.                 {
31.                     if (array[j].CompareTo(minValue) < 0)
32.                     {
33.                         m = j;
34.                         minValue = array[j];
35.                     }
36.                 }
37.                 Swap(array, i, m);
38.                 Console.ForegroundColor = ConsoleColor.Yellow;
39.                 Console.WriteLine($"Step {i+1}: i = {i}, m = {m}, min = {minValue}");
40.                 Console.ResetColor();
41.                 Print(array);
42.                 Console.WriteLine();
43.             }
44.         }
45.     }
46. }
```

Hãy nhìn dòng số 23 và 31.

Ở dòng số 23 xuất hiện một lệnh lạ mắt: `where T : IComparable`. Đây là cú pháp để giới hạn kiểu (type constraint) thực mà kiểu giả T có thể nhận. Cụ thể trong trường hợp này, T chỉ có thể được thay thế bằng những class thực thi giao diện IComparable.

> IComparable là giao diện mà nếu class nào thực thi thì ta có thể trực tiếp so sánh object của nó. Ví dụ, bạn có thể so sánh các số (nguyên, thực), so sánh hai chuỗi, so sánh hai ký tự.

Rõ ràng, để sắp xếp các phần tử của mảng thì ta phải so sánh được giá trị các phần tử. Nhưng để so sánh được giá trị của hai object thì class của object đó bắt buộc phải thực thi giao diện IComparable. Từ đây dẫn đến yêu cầu là kiểu giả T bắt buộc phải là các class thực thi giao diện IComparable. Cấu trúc `where T : IComparable` chính là để thực hiện giới hạn này.

Việc đặt ra giới hạn kiểu giúp viết code an toàn hơn. Nếu có vi phạm giới hạn kiểu, compiler sẽ báo lỗi ngay trong quá trình dịch. Thậm chí Intellisense hỗ trợ kiểm tra lỗi ngay trong giai đoạn viết code.

Các loại giới hạn kiểu thường gặp

Ở trên chúng ta đã gặp một loại giới hạn kiểu: kiểu chính thức phải thực thi một giao diện (interface) nào đó.

Ngoài ra, C# cung cấp nhiều loại giới hạn kiểu khác nhau. Sau đây là một số thường gặp.

Giới hạn kiểu là class

```
where T : class
```

Loại giới hạn này yêu cầu kiểu thực thay thế cho T không được phép là các kiểu như int, double, struct, enum. Tức là, T phải là các kiểu reference (các class built-in hoặc class tự tạo), chứ không được là các kiểu value.

Giới hạn kiểu value

```
where T : struct
```

Đây là loại giới hạn ngược lại so với trường hợp trên. Ở đây T bắt buộc phải là các kiểu value (int, double, struct, enum, v.v.), không được phép là class.

Giới hạn về constructor của class

```
where T: new()
```

Giới hạn này yêu cầu lớp thay thế cho T phải có hàm tạo (constructor) không tham số. Yêu cầu này sử dụng khi cần thực hiện khởi tạo object của T trong code generic.

Giới hạn kiểu con

```
where T: <base class name>
Ví dụ:
where T: Bird
```

Giới hạn này yêu cầu T phải là class con của một lớp khác. Trong ví dụ trên, T bắt buộc phải là lớp con của Bird.

Nhiều giới hạn đồng thời nhiều kiểu giả

```
where T: class where U:struct
```

Nếu có nhiều kiểu giả, mỗi kiểu giả được viết giới hạn riêng rẽ.

Từ khóa default – giá trị mặc định của kiểu dữ liệu

Có thể bạn đã biết, khi khai báo một biến, C# sẽ tự động gán cho biến đó một giá trị ban đầu. Giá trị đó gọi là giá trị mặc định của kiểu (default value). Ví dụ, kiểu int có giá trị mặc định là 0, kiểu bool là false, tất cả các kiểu tham chiếu (như string, DateTime, các class do người dùng xây dựng) là null.

Hãy đặt vào một tình huống khác. Một biến của bạn đã được thay đổi giá trị, giờ bạn muốn nó nhận lại giá trị mặc định. Hay nói cách khác, bạn đang muốn reset lại giá trị của biến đó về giá trị mặc định. Tình huống này chắc chắn không hiếm gặp phải không ạ?

Khi đó bạn cho nó nhận giá trị mặc định nào đây? Bạn đâu có biết kiểu thực sự của biến đó là gì. Vì bản thân kiểu dữ liệu cũng đang là tham số kia mà.

C# cung cấp cho chúng ta từ khóa `default`, dùng để tự động xác định giá trị mặc định của một kiểu bất kỳ, kể cả khi kiểu đó chỉ là một tham số.

Giả sử tham số kiểu bạn đặt là <T> như mọi khi. Các lệnh sau:

```
x = default(T);
y = default(T);
```

sẽ gán giá trị mặc định của kiểu T về cho x và y. Còn T là gì thì kệ nó.

Một số ứng dụng của generics trong C#

Các kiểu generic collection trong C# .NET

Để tiện lợi cho người lập trình trong xử lý dữ liệu, C# cung cấp nhiều kiểu dữ liệu tập hợp khác nhau. Tất cả các kiểu dữ liệu dạng generic collection được đặt trong không gian tên `System.Collections.Generics`. Trong không gian tên này chứa nhiều class khác nhau như List<T>, Stack<T>, Queue<T>, LinkedList<T>.

`List<T>` thuộc loại danh sách, là một dạng mảng động, với phần tử thuộc kiểu `T`, trong đó `T` có thể là bất kỳ kiểu dữ liệu nào của C# và .NET. Kiểu của phần tử được xác định

trong lúc khai báo và khởi tạo object (vì đây là kiểu generic). `T` có thể là bất kỳ kiểu dữ liệu nào của C# và .NET.

Tương tự như vậy, Stack<T> là lớp cài đặt cấu trúc ngăn xếp (stack) trong C#.

Queue<T> là lớp cài đặt cấu trúc dữ liệu hàng đợi (queue).

LinkedList<T> là lớp cài đặt cấu trúc dữ liệu danh sách liên kết (linked list).

Dictionary<TKey, TValue> là lớp cài đặt cấu trúc dữ liệu từ điển (dictionary) với hai kiểu giả: TKey là kiểu của khóa, TValue là kiểu của giá trị.

LINQ

Generics là một trong 4 thành phần tạo ra thư viện LINQ.

> Ba anh còn lại lần lượt là Phương thức mở rộng (extension method), kiểu đại diện (delegate), biểu thức lambda (lambda expression).

Kết quả của hầu hết các loại truy vấn LINQ là danh sách. Ai mà biết được người dùng viết truy vấn để lấy ra phần tử thuộc kiểu nào. Thậm chí, kiểu của phần tử còn không xác định rõ (gọi là kiểu vô danh – anonymous type). Do vậy, generics là không thể thiếu khi tạo ra các class chứa kết quả truy vấn.

Generic delegate

C# hỗ trợ người lập trình bằng cách định nghĩa ra một loạt kiểu dữ liệu *generic delegate* mà chúng ta có thể trực tiếp sử dụng ngay để khai báo biến. Sử dụng generic delegate giúp bỏ qua giai đoạn khai báo kiểu delegate.

> Đọc bài viết này để hiểu delegate trong c# là gì.

Về cơ bản, generic delegate là các kiểu delegate đã được định nghĩa sẵn sử dụng cơ chế generic. .NET framework định nghĩa 3 nhóm generic delegate: Actions, Funcs, Predicates.

Actions là các kiểu delegate tương ứng với các phương thức không trả về dữ liệu (đầu ra là void). Các kiểu Action được định nghĩa trong không gian tên System như sau:

```
1.  namespace System
2.  {
3.      public delegate void Action();
4.      public delegate void Action<in T>(T obj);
5.      public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
6.      // còn các delegate tương tự như vậy nữa
7.      // .NET framework định nghĩa tổng cộng 16 delegate như vậy với số lượng tham số đầu
8.  }
```

Funcs là các kiểu delegate tương ứng với các phương thức có trả về dữ liệu. Các kiểu funcs được định nghĩa trong không gian tên System như sau:

```
namespace System
{
    public delegate TResult Func<out TResult>();
    public delegate TResult Func<in T, out TResult>(T arg);
    public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
    // có 17 delegate tương tự như vậy
}
```

Predicate là kiểu delegate được định nghĩa sẵn như sau (trong System):

```
public delegate bool Predicate<in T>(T obj);
```

Kết luận

Cảm ơn bạn đã đi đến cùng với một bài viết dài lê thê (hơn 3000 từ và khơ khớ code ví dụ).

Bài viết này đã cung cấp cho bạn hầu như tất cả những gì quan trọng bạn cần biết về lập trình generic trong C#. Dĩ nhiên, còn rất nhiều tiểu tiết khác nữa bạn có thể tìm được trên mạng về vấn đề này.

Tuy nhiên tôi cho rằng nếu bạn nắm được hết bấy nhiêu đây thôi, bạn sẽ yên tâm xử lý đẹp generic nếu gặp phải nó.

Chúc bạn học tốt!

> + Nếu bạn thấy site hữu ích, trước khi rời đi hãy **giúp đỡ** site bằng một hành động nhỏ để site có thể phát triển và phục vụ bạn tốt hơn.
> + Nếu bạn thấy bài viết hữu ích, hãy giúp **chia sẻ** tới mọi người.
> + Nếu có thắc mắc hoặc cần trao đổi thêm, mời bạn viết trong phần **thảo luận** cuối trang.
> Cảm ơn bạn!