

# Router (1): Kiểu từ điển, nạp chồng toán tử

Hướng dẫn tự học lập trình C# toàn tập > Router (1): Kiểu từ điển, nạp chồng toán tử

Trong bài học này chúng ta sẽ học cách sử dụng kiểu từ điển (Dictionary) để xây dựng lớp Router giúp tiếp nhận và xử lý truy vấn của người dùng.

Trong các bài trước chúng ta đã xây dựng được các thành phần chủ chốt để tạo nên các chức năng quản lý dữ liệu cơ bản của ứng dụng. Tuy nhiên, các thành phần này đang hoạt động rời rạc và khả năng tương tác với người dùng hạn chế.

## NỘI DUNG CỦA BÀI [ Ẩn ]

1. Xử lý truy vấn và router
  - 1.1. Vai trò của router
  - 1.2. Đặc điểm của console
  - 1.3. Cấu trúc truy vấn đề xuất
  - 1.4. Ví dụ
2. Thực hành: xây dựng lớp hỗ trợ lưu tham số từ truy vấn
  - 2.1. Kiểu dữ liệu Dictionary
  - 2.2. Nạp chồng phép toán indexer
3. Kết luận

## Xử lý truy vấn và router

Ứng dụng của chúng ta hiện nay gặp hai vấn đề nghiêm trọng.

Thứ nhất, ứng dụng chỉ tiếp nhận được một số truy vấn đơn giản (single, create, list, update) và không tiếp nhận/xử lý được tham số đi cùng truy vấn. Đây là một vấn đề nghiêm trọng vì nếu người dùng không cung cấp được tham số cho lệnh sẽ hạn chế rất nhiều việc tương tác của người dùng với ứng dụng. Ví dụ, người dùng hiện nay gọi được lệnh single nhưng không thể cấp giá trị id của cuốn sách cần xem.

Thứ hai, các giao diện chưa thể truyền dữ liệu trở lại cho controller xử lý. Chúng ta có lớp BookCreateView, BookUpdateView để tiếp nhận dữ liệu của người dùng nhưng dữ liệu này chưa thể trả về BookController để xử lý. Theo nguyên tắc của MVC, dữ liệu không được phép xử lý ở lớp giao diện mà phải trả về cho controller xử lý.

Ngoài ra, trong phân tích hệ thống ở [bài 1](#), chúng ta còn phải xây dựng thêm nhiều tính năng khác. Tất cả đều yêu cầu phải tiếp nhận các truy vấn tương đối phức tạp từ người dùng.

## Vai trò của router

Theo nguyên tắc của MVC, tất cả yêu cầu của người dùng hoặc lệnh phát ra từ giao diện đều được chuyển cho **controller** xử lý. Điều này có nghĩa là phải có một cơ chế cho phép ánh xạ mỗi truy vấn của người dùng với việc thực thi một phương thức của controller.

Trong các MVC framework đều có một loại class đặc biệt, thường được gọi là *router*, làm nhiệm vụ này.

*Router* trong các MVC framework có nhiệm vụ tiếp nhận *truy vấn* của người dùng, phân tích ra các thành phần chính, và gọi phương thức tương ứng của controller. Mỗi phương thức của controller thường được gọi tắt là một *action* (hành động). Một truy vấn thường gọi tới một action.

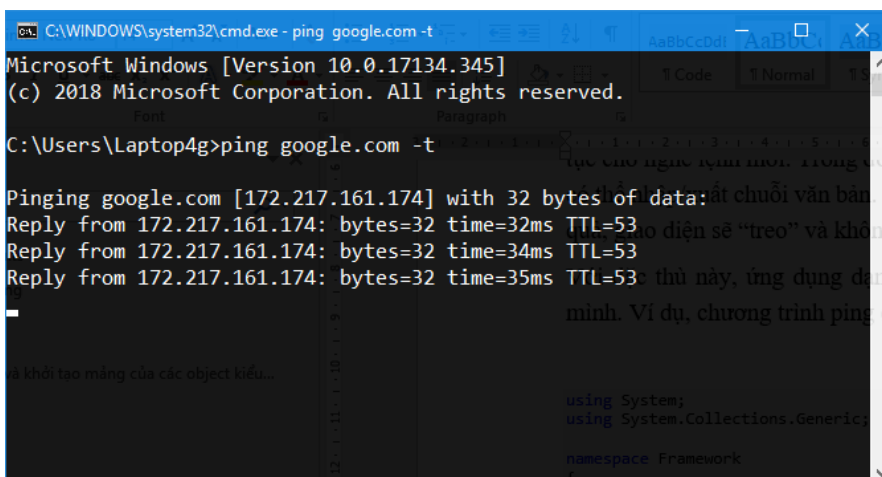
Trong bài học này chúng ta sẽ xây dựng các class để tạo ra một router đáp ứng các yêu cầu của một ứng dụng console.

## Đặc điểm của console

Ứng dụng console thông thường hoạt động theo chế độ request/response, nghĩa là người dùng nhập vào một truy vấn, ứng dụng thực hiện lệnh tương ứng và trả kết quả trở lại, sau đó lại tiếp tục chờ nghe truy vấn mới.

Trong đó, mỗi truy vấn đều là một chuỗi văn bản vì console chỉ có thể nhập/xuất chuỗi văn bản. Trong quá trình thực hiện lệnh cho đến lúc nhận lại kết quả, giao diện sẽ “treo” và không thể tiếp nhận bất kỳ thông tin gì.

Với đặc thù này, ứng dụng dạng console thường đưa ra cấu trúc truy vấn của riêng mình. Ví dụ, chương trình ping của Windows có cấu trúc lệnh như sau:



Gọi chương trình ping của windows

Tuy nhiên, truy vấn của console có một số đặc điểm chung. Ví dụ, thường có phần “lệnh” và phần “tham số”. Phần lệnh cho chương trình biết cần làm gì; phần tham số cung cấp thông tin cần cho việc thực hiện lệnh.

Các phần này cần được phân tích hợp lý để chương trình có thể phân tích và lấy ra những

Các phần này cần được phân tách hợp lý để chương trình có thể phân tích và lấy ra những thông tin cần thiết.

## Cấu trúc truy vấn đề xuất

Trong dự án này chúng ta sẽ đưa ra cấu trúc truy vấn như sau:

```
lệnh ? khóa_1 = giá_trị_1 & khóa_2 = giá_trị_2
```

nghĩa là, một truy vấn của chúng ta sẽ bao gồm hai thành phần:

1. phần lệnh (sẽ gọi là route): là một chuỗi ký tự bất kỳ không được chứa ký tự "?";
2. phần tham số (sẽ gọi là parameter): là chuỗi ký tự được viết theo quy tắc "khóa = giá\_trị"; mỗi cặp này được gọi là một tham số; các tham số viết tách nhau bởi ký tự "&";
3. phần lệnh và phần tham số viết tách nhau bởi ký tự "?" (vì lý do này, ký tự "?" không được phép có mặt trong phần lệnh).

Ngoài ra cấu trúc trên còn phải có các đặc điểm sau:

1. lệnh không phân biệt ký tự hoa/thường;
2. khóa của tham số có phân biệt hoa/thường;
3. số lượng dấu cách giữa các thành phần của truy vấn là không quan trọng.

## Ví dụ

Ví dụ, chúng ta dự kiến các lệnh single, list, create, update sẽ có dạng như sau:

```
1. single ? id = 1
2. list
3. create
4. update ? id = 2
```

Sau này chúng ta sẽ xây dựng tiếp phần trợ giúp với cấu trúc truy vấn dự kiến như sau:

```
1. Help
2. Help ? cmd = single
3. Help ? cmd = update
```

Cấu trúc truy vấn này mô phỏng lại cấu trúc truy vấn GET của giao thức HTTP.

Cấu trúc này cho phép dễ dàng phân tích các thành phần.

Với cấu trúc này, chúng ta có nhiệm vụ:

1. phân tích một truy vấn ra thành phần lệnh và thành phần tham số;
2. phân tích thành phần tham số và chuyển đổi về một kiểu dữ liệu khác tiện lợi hơn cho việc lập trình.

## Thực hành: xây dựng lớp hỗ trợ lưu tham số từ truy vấn

Trong phần thực hành này, chúng ta sẽ xây dựng một class cho phép chuyển đổi phần tham số của truy vấn thành một kiểu dữ liệu khác để dễ dàng hơn cho việc lập trình. Việc tách một truy vấn thành phần lệnh và phần tham số chúng ta sẽ thực hiện ở phần thực hành tiếp theo.

Tạo file Parameter.cs trong thư mục Framework với lớp Parameter như sau:

```
1. using System;
2. using System.Collections.Generic;
3.
4. namespace Framework
5. {
6.     /// <summary>
7.     /// lưu các cặp khóa-giá trị người dùng nhập;
8.     /// chuỗi tham số cần viết ở dạng khóa=giá trị;
9.     /// nếu có nhiều tham số thì viết tách nhau bằng ký tự &
10.    /// </summary>
11.    public class Parameter
12.    {
13.        private readonly Dictionary<string, string> _pairs = new Dictionary<string, string>();
14.
15.        /// <summary>
16.        /// nạp chồng phép toán indexing []; cho phép truy xuất giá trị theo kiểu biến[key]
17.        /// </summary>
18.        /// <param name="key">khóa</param>
19.        /// <returns>giá trị tương ứng</returns>
20.        public string this[string key] // để nạp chồng phép toán indexing phải viết hai lần
21.        {
22.            get {
23.                if (_pairs.ContainsKey(key))
24.                    return _pairs[key];
25.                else return null;
26.            } // phương thức get trả lại giá trị từ dictionary
27.            set => _pairs[key] = value; // phương thức set gán giá trị cho dictionary
28.        }
29.
30.        /// <summary>
31.        /// Kiểm tra xem một khóa có trong danh sách tham số không
32.        /// </summary>
33.        /// <param name="key">khóa cần kiểm tra</param>
34.        /// <returns></returns>
35.        public bool ContainsKey(string key)
36.        {
37.            return _pairs.ContainsKey(key);
38.        }
39.
40.        /// <summary>
41.        /// nhận chuỗi ký tự và phân tích, chuyển thành các cặp khóa-giá trị
42.        /// </summary>
43.        /// <param name="parameter">chuỗi ký tự theo quy tắc khóa_1=giá_trị_1&khóa_2=giá_trị_2
44.        public Parameter(string parameter)
45.        {
46.            // cắt chuỗi theo mặc là ký tự &
47.            // kết quả của phép toán này là một mảng, mỗi phần tử là một chuỗi có dạng k=v
48.            var pairs = parameter.Split(new[] { '&' }, StringSplitOptions.RemoveEmptyEntries);
49.            foreach (var pair in pairs)
50.            {
51.                var p = pair.Split('='); // cắt mỗi phần tử lấy mốc là ký tự =
52.                if (p.Length == 2) // một cặp khóa = giá_trị đúng sau khi cắt sẽ phải có 2 phần
53.                {
54.                    var key = p[0].Trim(); // phần tử thứ nhất là khóa
55.                    var value = p[1].Trim(); // phần tử thứ hai là giá trị
56.                    this[key] = value; // lưu cặp khóa-giá trị này lại sử dụng phép toán indexing
57.
58.                    // cũng có thể viết theo kiểu khác, trực tiếp sử dụng biến _pairs
59.                    // _pairs[key] = value;
60.                }
61.            }
62.        }
63.    }
64. }
```

## Kiểu dữ liệu Dictionary

Trong code ở phần thực hành trên chúng ta đã khai báo một biến thuộc kiểu

`Dictionary<string, string>` để lưu các tham số người dùng nhập.

Biến này dùng để lưu các cặp khóa/giá trị, trong đó khóa và giá trị đều có kiểu string. Khi người dùng nhập một truy vấn (ở dạng chuỗi văn bản), phần tham số sẽ được tách riêng ra, sau đó lại tách tiếp từng cặp khóa = giá trị để lưu vào từ điển.

`Dictionary` là một kiểu dữ liệu tập hợp tổng quát (generic collection) tương tự như `List<T>` nhưng được dùng cho lưu trữ danh sách các cặp **khóa – giá trị**. Khóa và giá trị có thể thuộc bất kỳ kiểu dữ liệu nào của .NET.

Kiểu dữ liệu này được mô tả đầy đủ là `Dictionary<TKey, TValue>`, trong đó `TKey` là kiểu của khóa, `TValue` là kiểu của giá trị. Lớp `Dictionary<TKey, TValue>` được định nghĩa trong không gian tên `System.Collections.Generic`.

`Dictionary` có thể hình dung như bộ từ điển song ngữ, ví dụ, từ điển Anh – Việt, trong đó từ tiếng Anh là khóa, nghĩa trong tiếng Việt là giá trị.

Lưu ý, khi sử dụng từ điển không được phép sử dụng lặp khóa hoặc để khóa có giá trị null. Khóa bắt buộc phải là duy nhất (tương tự như trong từ điển song ngữ). Nếu trùng lặp khóa sẽ báo lỗi ở giai đoạn runtime.

## Nạp chồng phép toán indexer

Trong lớp `Parameter` chúng ta gặp một phương thức có khai báo lạ mắt

```
1. public string this[string key] // để nạp chồng phép toán indexing phải viết hai phương t
2. {
3.     get => _pairs[key]; // phương thức get trả lại giá trị từ dictionary
4.     set => _pairs[key] = value; // phương thức set gán giá trị cho dictionary
5. }
```

Phương thức này có ý nghĩa đặc biệt: nạp chồng toán tử indexer (phép toán indexer, phép toán chỉ mục).

*Indexer* là một phép toán giúp client code sử dụng object tương tự như khi sử dụng mảng. Indexer thường được sử dụng với các kiểu dữ liệu chứa trong nó một tập hợp dữ liệu (collection hoặc array). Indexer giúp đơn giản hóa việc sử dụng ở client code.

Phép toán indexer giúp client code có thể truy xuất biến của kiểu `Parameter` như sau:

```
1. Parameter p = new Parameter("id=1&title=A new book");
2. Var id = p["id"];
3. p["title"] = "C# programming for dummy";
```

Client code không biết gì về dữ liệu kiểu từ điển chứa trong `Parameter` nhưng có thể sử dụng phép toán indexer để dễ dàng truy xuất dữ liệu của từ điển thông qua tên biến kiểu `Parameter`.

Như vậy áp dụng phép toán indexer rất tiện lợi cho việc truy xuất các cặp khóa-giá trị chứa trong `Parameter`.

**Expression body** là một lỗi viết xuất hiện từ C# 6: nếu thân của phương thức chỉ chứa một lệnh duy nhất có thể sử dụng cấu trúc như sau để viết:

```
Tên_phương_thức() => lệnh;
```

Từ C# 7 có thể sử dụng expression body cho cả phương thức get và set của property.

Trong code của indexer ở trên chúng ta đã sử dụng cấu trúc này cho ngắn gọn. Từ giờ về sau, ở những chỗ phù hợp chúng ta sẽ sử dụng cấu trúc expression body.

## Kết luận

---

Trong bài này chúng ta bắt đầu xây dựng lớp Router giúp ánh xạ truy vấn người dùng sang phương thức của controller. Chúng ta xử lý vấn đề đầu tiên là cấu trúc của truy vấn, đồng thời xây dựng class Parameter giúp lưu trữ dữ liệu của truy vấn. Qua đó, chúng ta học kỹ thuật sử dụng lớp Dictionary.

Trong các bài tiếp theo chúng ta sẽ lần lượt hoàn thiện lớp Router.

- + 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!