FileStream trong C#, làm việc với file và thư mục

FileStream là một loại stream đặc biệt chuyên dùng để đọc ghi dữ liệu với file. Đây là những khái niệm tương đối mới và khá đặc thù của C# và .NET. Bài học này sẽ giúp bạn nắm được kỹ thuật đọc ghi file với FileStream và cách làm việc với file/thư mục trong C#.

NỘI DUNG CỦA BÀI [Ẩn]

Làm việc với file và thư mục

Trước hết chúng ta sẽ học cách sử dụng dụng các lớp .NET hỗ trợ làm việc với hệ thống file của windows.

Tất cả các lớp để làm việc với file trong .NET nằm trong không gian tên `System.IO`. Ba class chính để làm việc với hệ thống file là `Directory` (làm việc với thư mục), `File` (làm việc với file), `Path` (làm việc với đường dẫn).

Lớp Directory

Lớp Directory chứa hầu hết các phương thức tĩnh giúp làm việc với file và thư mục. Dưới đây là một số phương thức của lớp này giúp kiểm tra đường dẫn và giúp lấy danh sách file trong một thư mục.

Phương thức tĩnh `GetFiles`: tìm tất cả các file trong thư mục có phần mở rộng theo yêu cầu. Ví dụ dưới đây tìm tất cả các file exe trong thư mục E:\Catalogue:

```
1.  Directory.GetFiles(@"E:\CATALOGUE", "*.exe", SearchOption.AllDirectories)
2.  string[149] { "E:\\CATALOGUE\\Client PWI\\uninstall.exe", "E:\\CATALOGUE\\Client PWI\\el
3.  >
```

Phương thức này sử dụng ba tham số:

1. đường dẫn tới thư mục;
2. mẫu tìm kiếm: mẫu văn bản mà phương thức `GetFiles` sử dụng trong quá trình tìm kiếm. `GetFiles` chỉ trả lại những file mà tên phù hợp với mẫu văn bản của tham số này.
3. phạm vi tìm kiếm: xác định xem phương thức `GetFiles` chỉ tìm trong thư mục được chỉ định (`TopDirectoryOnly`) hay tìm cả trong các thư mục con của nó (`AllDirectories`).

Kết quả thực hiện của phương thức này là một mảng string chứa tên đầy đủ (bao gồm cả đường dẫn) của các file tìm thấy.

Tương thự, phương thức GetDirectories trả về danh sách tất cả các thư mục con trong một thư mục.

Phương thức tĩnh `Exists` : kiểm tra xem một đường dẫn tới thư mục có tồn tại hoặc chính xác không.

```
C# Interactive
1.  > Directory.Exists(@"C:\Program Files")
2.  true
3.  >
```

Phương thức CreateDirectory: tạo thư mục mới.

Phương thức Delete: xóa thư mục.

Bạn có thể dễ dàng tìm hiểu được cách sử dụng của các phương thức còn lại của lớp này.

Lớp Path

Lớp Path cũng chứa hầu hết các phương thức tĩnh giúp phân tích đường dẫn tới file hoặc thư mục. Dưới đây là cách sử dụng một phương thức của lớp này:

Phương thức `GetDirectoryName` trả lại phần tên thư mục trong đường dẫn tới file.

Phương thức GetFileName trích ra phần tên file trong một đường dẫn tới file, bỏ phần đường dẫn thư mục.

Phương thức `GetFileNameWithoutExtension` trích ra phần tên của file, bỏ phần đường dẫn và phần mở rộng.

Phương thức `GetExtension` trả về phần mở rộng của tên file hoặc thư mục.

Các phương thức của lớp Path đều tương đối dễ sử dụng. Bạn đọc có thể tự mình tìm hiểu các phương thức khác.

Đọc/ghi dữ liệu với file trong C#, FileStream

Ở phần trước chúng ta đã xem xét tổng thể về stream trong .NET framework. Trong phần này chúng ta sẽ làm việc với một loại luồng backing store cụ thể trong C#: FileStream.

Khởi tạo FileStream

Trong C# bạn có thể khởi tạo FileStream theo nhiều cách khác nhau:

```
1.  // sử dụng hàm tạo của lớp FileStream
2.  FileStream fs = new FileStream("data1.bin", FileMode.Create);
3.
4.  // sử dụng các phương thức tĩnh của lớp File
5.  FileStream fs1 = File.OpenRead("data1.bin"); // Read-only
6.  FileStream fs2 = File.OpenWrite("data2.bin"); // Write-only
7.  FileStream fs3 = File.Create("data3.bin"); // Read/write
```

Tất cả các cách trên có điểm chung là bắt buộc phải cung cấp một đường dẫn tới file.

Cách thứ nhất là linh hoạt nhất, cho phép lựa chọn chế độ làm việc với file, FileMode. Ba phương pháp còn lại đều là các "lối tắt" giúp đơn giản hóa việc mở file. Thực chất, chúng tương đương với một số chế độ của FileMode ở phương pháp thứ nhất.

Sau khi khởi tạo có thể bắt đầu đọc/ghi dữ liệu với file. Tuy nhiên, hiện tại bạn chỉ có đọc và xử lý các byte thô trực tiếp từ FileStream. Để có thể xử lý trong chương trình, bạn phải tự mình biến đổi các byte đó về kiểu dữ liệu mà chương trình cần đến.

Ghi vào file qua FileStream

Hãy xem ví dụ sau:

```
1.  int i = 1234;
2.  string str = "Hello world";
3.  fs.Write(BitConverter.GetBytes(i), 0, 4);
4.  fs.Write(Encoding.UTF8.GetBytes(str), 0, Encoding.UTF8.GetByteCount(str));
5.  fs.Flush();
6.  fs.Close();
```

Trong ví dụ này, bạn ghi vào file một số nguyên i có giá trị 1234 và một chuỗi có giá trị "Hello world".

Như bạn đã biết từ bài học về stream, các luồng backing store hoàn toàn làm việc với byte hoặc mảng byte. Chúng không biết về các loại giá trị cấp cao như `int`, `string`, `bool` hay các object. Do đó bạn phải biến đổi tất cả các giá trị về mảng byte.

Đối với các kiểu dữ liệu cơ sở (int, bool, char, v.v.), .NET framework cung cấp lớp `BitConverter` để biến đổi về mảng byte và ngược lại. Đối với dữ liệu văn bản cần sử dụng lớp `Encoding`.

Kiểu byte chỉ sử dụng 1 byte để biểu diễn, do đó biểu diễn ở dạng mảng byte của giá trị thuộc kiểu byte là một mảng có 1 phần từ và chứa đúng giá trị đó.

- Quá trình biến đổi một giá trị sang mảng byte phức tạp hơn đối với các kiểu dữ liệu kích thước lớn:
- Đối với kiểu int (sử dụng 4 byte để biểu diễn 1 giá trị), mảng byte này chứa 4 phần tử (bất kể số nguyên đó có giá trị bao nhiêu). Đối với kiểu long (sử dụng 8 byte), mảng byte phải chứa 8 phần tử.

> Đến đây phát sinh vấn đề: trật tự của các phần tử trong mảng, gọi là endianness. Có hai xu hướng khác nhau để viết thứ tự các byte trong mảng:
> (1) Lối viết **big-endian** (sử dụng trong Mac và Linux): byte bên trái có giá trị hơn, giống cách chúng ta đọc số;
> (2) Lối viết **little-endian** (sử dụng trong Windows): byte bên phải có giá trị hơn, ngược lại cách chúng ta đọc số.

Phương thức Write của FileStream thực thi phương thức abstract tương ứng của lớp Stream cho phép ghi một mảng byte vào luồng. Phương thức này chỉ ghi <count> byte bắt đầu từ vị trí <offset>, trong đó offset và count lần lượt là tham số thứ 2 và thứ 3 của phương thức này.

Trong ví dụ trên, phương thức `GetByte` của `BitConverter` chuyển biến i thành một mảng 4 byte (do int là kiểu dữ liệu biểu diễn bằng 4 byte). Mảng này được ghi trọn vẹn vào file, do đó offset = 0, count = 4.

Đối với kiểu string, biểu diễn dạng mảng byte của nó phụ thuộc vào cách mã hóa ký tự (encoding). Nếu dùng mã ASCII, mỗi ký tự là 1 byte nhưng nếu dùng mã hóa nhiều byte như Unicode, số byte cho mỗi ký tự có thể khác nhau. Vì vậy, .NET cung cấp lớp Encoding để thực hiện chuyển đổi này.

Mỗi stream thường cung cấp một bộ nhớ đệm để hỗ trợ đọc ghi dữ liệu. FileStream cũng như vậy. Khi ghi, dữ liệu được lưu tạm ở bộ nhớ đệm trước khi thực sự ghi vào file. Nếu muốn dữ liệu được đẩy ngay vào file có thể gọi phương thức Flush.

Trong suốt quá trình làm việc, file sẽ bị khóa và object khác không thể làm việc với file này. Vì vậy, sau khi kết thúc làm việc với file nên gọi phương thức Close để đóng luồng và giải phóng file.

Đọc từ file qua FileStream

Hãy cùng xem ví dụ sau:

```
1.  var fs = new FileStream("data1.bin", FileMode.OpenOrCreate, FileAccess.Read);
2.  var buffer = new byte[4];
3.  fs.Read(buffer, 0, 4);
4.  int i = BitConverter.ToInt32(buffer, 0);
5.  Console.WriteLine($"i = {i}");
6.  int length = (int)fs.Length - 4;
7.  buffer = new byte[length];
8.  fs.Read(buffer, 0, length);
9.  string str = Encoding.UTF8.GetString(buffer);
10. fs.Close();
11. Console.WriteLine($"str = {str}");
```

Trong ví dụ này, chúng ta mở lại file đã tạo lúc trước và đọc các giá trị lưu ở trong đó, bao gồm một số nguyên và một chuỗi ký tự.

Để đọc ra một giá trị, chúng ta phải tạo ra một mảng đệm trước để luồng file đưa giá trị vào. Mảng đệm này phải có kích thước bằng hoặc lớn hơn dữ liệu được đọc ra.

Với kiểu int, kích thước là cố định (4 byte); với kiểu string, do kích thước không cố định nên ta phải tính toán ra kích thước của nó (bằng tổng số byte trong file trừ đi số byte mà biến int chiếm).

Sau khi đọc được dữ liệu vào mảng đệm, chúng ta sử dụng các phương thức tương ứng của `BitConverter` và `Encoding` để chuyển đổi về kiểu dữ liệu cần thiết.

Các vấn đề liên quan đến FileStream trong C#

Sử dụng stream adapter

Như ở trên chúng ta thấy, việc đọc ghi trực tiếp với FileStream trong C# rất rắc rối, đặc biệt khi cần ghi/đọc những object phức tạp. Để giải quyết một phần vấn đề này, bạn có thể sử dụng các lớp **stream adapter**.

Stream adapter đóng vai trò hỗ trợ sử dụng luồng backing store bằng cách che đi các phương thức làm việc trực tiếp với byte và cung cấp thêm các phương thức để xử lý dữ liệu cấp cao. Tùy thuộc vào kiểu dữ liệu cần làm việc chúng ta lựa chọn các loại adapter khác nhau.

Hãy cùng xem ví dụ sau:

```
1.  FileStream fs = new FileStream("data1.bin", FileMode.Create, FileAccess.ReadWrite);
2.  BinaryWriter bWriter = new BinaryWriter(fs);
3.  bWriter.Write(1234);
4.  StreamWriter sWriter = new StreamWriter(fs);
5.  sWriter.Write("Hello world");
6.  sWriter.Flush();
7.  fs.Close();
8.
9.  fs = new FileStream("data1.bin", FileMode.OpenOrCreate, FileAccess.Read);
10. BinaryReader bReader = new BinaryReader(fs);
11. var i = bReader.ReadInt32();
12. StreamReader sReader = new StreamReader(fs);
13. var str = sReader.ReadToEnd();
14. Console.WriteLine($"i = {i}");
15. Console.WriteLine($"str = {str}");
16. fs.Close();
```

Trong ví dụ này chúng ta sử dụng hai loại adapter: BinaryWriter/BinaryReader để làm việc với các kiểu cơ sở (trừ kiểu string); StreamWriter/StreamReader để làm việc với dữ liệu văn bản.

Khi sử dụng hai loại adapter này, việc đọc/ghi dữ liệu với FileStream được đơn giản hóa rất nhiều vì các adapter đã đứng ra chịu trách nhiệm biến đổi dữ liệu trong quá trình đọc/ghi. Các phương thức của hai loại adapter này cũng rất giống với cách thức đọc/ghi dữ liệu từ giao diện console mà bạn đã quen thuộc.

Một số phương thức “tắt”

Ngoài việc sử dụng các phương pháp “chính thống” như ở trên đã xem xét, lớp File cũng cung cấp cho chúng ta nhiều phương thức “tắt” để đơn giản hóa việc ghi/đọc dữ liệu với file:

- File.WriteAllText,
- File.ReadAllText,
- File.WriteAllBytes,
- File.ReadAllBytes,
- File.WriteAllLines,
- File.ReadAllLines,
- File.OpenRead,
- File.OpenWrite,
- File.Create.

Các phương thức này tuy rằng tiện lợi nhưng có thể làm mất một phần tính hiệu quả của FileStream. Ví dụ, các lệnh đọc tắt này đọc toàn bộ dữ liệu vào bộ nhớ, vốn rất không hiệu quả nếu file lớn.

Sử dụng using block

Trong các ví dụ trên, sau khi kết thúc làm việc với file, chúng ta phải tự mình gọi lệnh đóng luồng file. Đây là một thao tác rất hay bị bỏ quên.

Trong những tình huống khác, chúng ta chỉ cần sử dụng object trong một khối code nhất định, sau đó object bị hủy bỏ hoặc không tiếp tục sử dụng nữa. Để giải phóng người lập trình khỏi việc phải tự mình hủy bỏ các object như vậy, C# cung cấp một cấu trúc mới: using block. Hãy cùng xem ví dụ sau:

```
1.  using (FileStream fs = new FileStream("data1.bin", FileMode.Create, FileAccess.ReadWrite
2.  {
3.      BinaryWriter bWriter = new BinaryWriter(fs);
4.      bWriter.Write(1234);
5.      StreamWriter sWriter = new StreamWriter(fs);
6.      sWriter.Write("Hello world");
7.      sWriter.Flush();
8.  }
9.
10. using (var fs = new FileStream("data1.bin", FileMode.OpenOrCreate, FileAccess.Read))
11. {
12.     BinaryReader bReader = new BinaryReader(fs);
13.     var i = bReader.ReadInt32();
14.     StreamReader sReader = new StreamReader(fs);
15.     var str = sReader.ReadToEnd();
16.     Console.WriteLine($"i = {i}");
17.     Console.WriteLine($"str = {str}");
18. }
```

Ở hai đoạn code này chúng ta không cần tự mình đóng luồng nữa. Biến `fs` được tạo ra trong cấu trúc using và được cấu trúc này theo dõi. Khi kết thúc khối code, biến `fs` sẽ tự bị hủy bỏ. Cấu trúc này rất thường xuyên được sử dụng khi làm việc với luồng.

Khi sử dụng các phương thức "tắt" như `File.WriteAllText`, `File.ReadAllText`, `File.WriteAllBytes`, `File.ReadAllBytes`, `File.WriteAllLines`, `File.ReadAllLines`, file được mở và đóng tự động. Chúng ta không cần tự mình thực hiện các thao tác làm việc với file thông thường nữa. Vì vậy các phương thức này được gọi là các phương thức tắt.

Kết luận

Bài học này đã giúp bạn biết cách làm việc với loại stream đầu tiên là FileStream giúp đọc ghi dữ liệu với file. Bạn cũng học được cách làm việc với hệ thống file của windows từ chương trình C#.

> + 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!