Lập trình C# (C Sharp)
Attribute Annotation (Bài trước)
(Bài tiếp) (Multithreading) Parallel

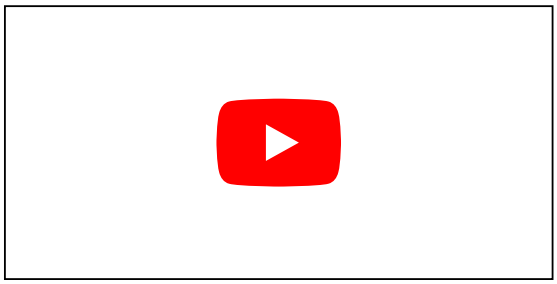
Dependency injection (DI) trong C# với ServiceCollection

Khái niệm về DI, các kiểu Inject, sử dụng DI container mặc định của NET ServiceCollection, khởi tạo dịch vụ với Factory, hàm khởi tạo, Inject thiết lập và sử dụng config file định dạng JSON cấu hình để thiết lập thuộc tính đối tượng

- *Sử dụng Options khởi tạo dịch vụ*
- *Sử dụng cấu hình từ file thiết lập*

Inversion of Control (IoC) / Dependency inversion

Inversion of Control (IoC - Đảo ngược điều khiển) là một nguyên lý thiết kế trong công nghệ phần mềm trong đó các thành phần nó dựa vào để làm việc bị đảo ngược quyền điều khiển khi so sánh với lập trình hướng thủ thục truyền thống.

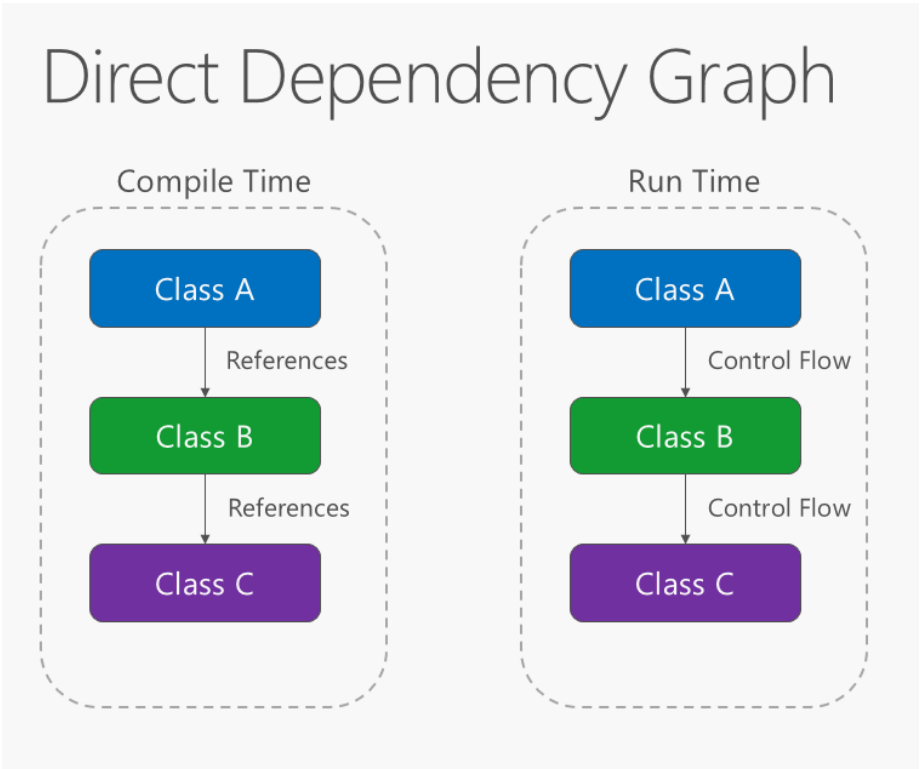


Khi áp dụng cho các đối tượng lớp (dịch vụ) có thể gọi nó là **Dependency inversion** (đảo ngược phụ thuộc), để diễn giải trước tiên cần nắm rõ khái niệm **Dependency** (phụ thuộc)

dependency : Giả sử bạn có một lớp **classA**, lớp này có sử dụng một chức năng từ đối tượng lớp **classB** (classA hoạt động dựa vào classB). Lúc đó **classB** gọi là **phụ thuộc (dependency)** *(của classA)*

Thiết kế truyền thống - tham chiếu trực tiếp đến Dependency

Có lớp *class A* có sử dụng một chức năng (gọi hàm ào đó) của *class B*, lớp *class B* lại tham chiếu và gọi các chức năng có trong *class C*. Ta thấy *class A* dựa vào *class B* để hoạt động, *class B* dựa vào *class C*. Nếu vậy khi thiết kế theo cách thông thường, viết code thì *class A* có tham chiếu trực tiếp (cứng) đến *class B* và trong *class B* có tham chiếu đến *class C* (thể hiện như hình dưới).



Sự phụ thuộc đối tượng này vào đối tượng khác ở thời điểm viết code và thời điểm thực thi là hoàn toàn giống nhau.

```
class ClassC {
    public void ActionC() => Console.WriteLine("Action in ClassC");
```

```
}

class ClassB {
    // Phụ thuộc của ClassB là ClassC
    ClassC c_dependency;

    public ClassB(ClassC classc) => c_dependency = classc;
    public void ActionB()
    {
        Console.WriteLine("Action in ClassB");
        c_dependency.ActionC();
    }
}

class ClassA {
    // Phụ thuộc của ClassA là ClassB
    ClassB b_dependency;

    public ClassA(ClassB classb) => b_dependency = classb;
    public void ActionA()
    {
        Console.WriteLine("Action in ClassA");
        b_dependency.ActionB();
    }
}
```

Khi sử dụng:

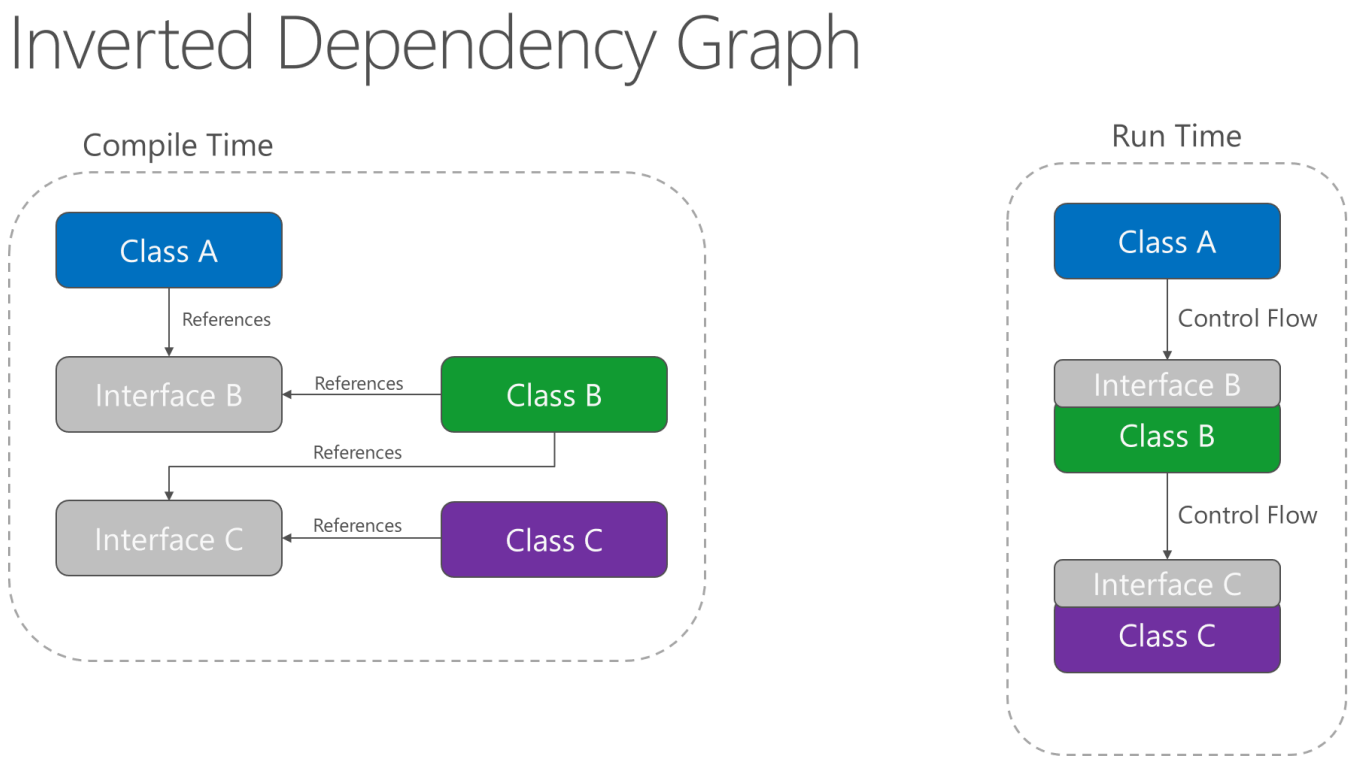
```
ClassC objectC = new ClassC();
ClassB objectB = new ClassB(objectC);
ClassA objectA = new ClassA(objectB);

objectA.ActionA();

// Kết quả:
// Action in ClassA
// Action in ClassB
// Action in ClassC
```

Thiết kế theo cách đảo ngược phụ thuộc Inverse Dependency

Cách viết code này, ở thời điểm thực thi thì *class A* vẫn gọi được hàm có *class B*, *class B* vẫn gọi hàm có *class C* nghĩa là kết quả không đổi. Tuy nhiên, khi thiết kế ở thời điểm viết code (trong code) *class A* không tham chiếu trực tiếp đến *class B* mà nó lại sử dụng **interface** (hoặc lớp **abstruct**) mà *classB* triển khai. Điều này dẫn tới sự phụ thuộc **lỏng lẻo** giữa *classA* và *classB* (xem hình)



Khi thực thi, *classB* có thể được thay thế bởi bất kỳ lớp nào triển khai từ giao diện *interface B*, *classB* cụ thể mà *classA* sử dụng được **quyết định và điều khiển** bởi *interface B* (điều này có nghĩa tại sao gọi là đảo ngược phụ thuộc)

```
interface IClassB {
    public void ActionB();
}
interface IClassC {
    public void ActionC();
}

class ClassC : IClassC {
    public ClassC() => Console.WriteLine ("ClassC is created");
    public void ActionC() => Console.WriteLine("Action in ClassC");
}

class ClassB : IClassB {
    IClassC c_dependency;
    public ClassB(IClassC classc)
    {
        c_dependency = classc;
        Console.WriteLine("ClassB is created");
    }
    public void ActionB()
    {
        Console.WriteLine("Action in ClassB");
        c_dependency.ActionC();
    }
}


class ClassA {
    IClassB b_dependency;
    public ClassA(IClassB classb)
    {
        b_dependency = classb;
        Console.WriteLine("ClassA is created");
    }
    public void ActionA()
    {
        Console.WriteLine("Action in ClassA");
        b_dependency.ActionB();
```

```
    }
}
```

Kết quả sử dụng khi chạy là tương tự

```
IClassC objectC = new ClassC();
IClassB objectB = new ClassB(objectC);
ClassA objectA = new ClassA(objectB);

objectA.ActionA();
```

Code dễ dàng thay các phụ thuộc, ví dụ, định nghĩa thêm:

```
class ClassC1 : IClassC
{
    public ClassC1() => Console.WriteLine ("ClassC1 is created");
    public void ActionC()
    {
        Console.WriteLine("Action in C1");
    }
}

class ClassB1 : IClassB
{
    IClassC c_dependency;
    public ClassB1(IClassC classc)
    {
        c_dependency = classc;
        Console.WriteLine("ClassB1 is created");
    }
    public void ActionB()
    {
        Console.WriteLine("Action in B1");
        c_dependency.ActionC();
    }
}
```

Khi sử dụng, có thể thay thế các dependency tùy thuộc vào mục đích sử dụng:

```
IClassC objectC = new ClassC1();            // new ClassC();
IClassB objectB = new ClassB1(objectC);     // new ClassB();
ClassA objectA = new ClassA(objectB);

objectA.ActionA();
```

Kỹ thuật lập trình Dependency injection

Dependency injection (DI) là một kỹ thuật trong lập trình, nó là một hình thức cụ thể của **Inverse of Control (Dependency Inverse)** đã nói ở trên. **DI** thiết kế sao cho các dependency (phụ thuộc) của một đối tượng **CÓ THỂ** được đưa vào, tiêm vào đối tượng đó (Injection) khi nó cần tới (khi đối tượng khởi tạo). Cụ thể cần làm:

- Xây dựng các lớp (dịch vụ) có sự phụ thuộc nhau một cách lỏng lẻo, và dependency có thể tiêm vào đối tượng (injection) - thường qua phương thức khởi tạo constructor, property, setter
- Xây dựng được một thư viện có thể tự động tạo ra các đối tượng, các dependency tiêm vào đối tượng đó, thường là áp dụng kỹ thuật Reflection của C# (xem thêm lớp type): Thường là thư viện này quá phức tạp để tự phát triển nên có thể sử dụng các thư viện có sẵn như: Microsoft.Extensions.DependencyInjection hoặc thư viện bên thứ ba như Windsor, Unity Ninject ...

Nói chung, các khái niệm về DI rất trừu tượng và khó hiểu! Để rõ hơn cần thực hiện từng bước qua các ví dụ.

Giả sử có lớp `Car` có chức năng (phương thức) `Beep()` - để phát ra tiếng còi xe, mà để phát ra tiếng còi - nó lại dựa vào vào lớp `Horn` chuyên tạo ra tiếng còi - lúc đó ta nói lớp `Car` có một phụ thuộc (dependency Horn) là lớp Horn, Horn là dependency của Car.

Muốn lớp Car hoạt động thì nó phải có đối tượng (dịch vụ) từ Horn. Vậy khi thiết kế, thường có hai cách:

- Trong lớp Car thiết kế code mà nó phụ thuộc cứng vào lớp Horn - tự khởi tạo Horn, cách thiết kế này không có khả năng áp dụng kỹ thuật DI
- Trong lớp Car, dependency Horn không do Car trực tiếp khởi tạo mà nó được đưa vào qua phương thức khởi tạo, qua setter, qua gán property. Các thiết kế này linh hoạt và có **KHẢ NĂNG** để áp dụng DI

Ví dụ code cho 2 trường hợp này như sau:

Viết Code mà không có khả năng áp dụng DI

```
public class Horn {
    public void Beep () => Console.WriteLine ("Beep - beep - beep ...");
}

public class Car {
    public void Beep () {
        // chức năng Beep xây dựng có định với Horn
        // tự tạo đối tượng horn (new) và dùng nó
        Horn horn = new Horn ();
        horn.Beep ();
    }
}
```

Code viết như trên khá thông dụng và vẫn chạy tốt. Bấm còi xe vẫn kêu Beep, beep ...

```
var car = new Car();
car.Beep();         // Beep - beep - beep ...
```

Nhưng code trên có một vấn đề là tính linh hoạt khi sử dụng. Chức năng `Beep()` của `Car` nó tự tạo ra đối tượng `Horn` và sử dụng nó - làm cho Car gắn cứng vào Horn với cấu trúc khởi tạo hiện thời.

Nếu lớp Horn sửa lại, ví dụ muốn khởi tạo Horn phải chỉ ra một tham số nào đó, ví dụ như độ lớn tiếng còi `level`

```
public class Horn {
    int level; // độ lớn của còi xe
    public Horn (int level) => this.level = level; // thêm khởi tạo level
    public void Beep () => Console.WriteLine ($"(level {level}) Beep - beep - beep ...");
}
```

Việc thay đổi `Horn` làm cho `Car` không còn dùng được nữa, nếu muốn `Car` hoạt động cần sửa lại code của Car, ví dụ tại Beep sửa thành

```
Horn horn = new Horn(10);     // Khởi tạo với Horn với tham số level
horn.Beep();
```

Viết code có KHẢ NĂNG áp dụng DI

Xây dựng lại ví dụ trên sao cho phụ thuộc của Car có thể đưa vào nó từ bên ngoài.

```
public class Horn {
    public void Beep () => Console.WriteLine ("Beep - beep - beep ...");
}

public class Car {
    // horn là một Dependecy của Car
    Horn horn;

    // dependency Horn được đưa vào Car qua hàm khởi tạo
    public Car(Horn horn) => this.horn = horn;

    public void Beep () {
        // Sử dụng Dependecy đã được Inject
        horn.Beep ();
```

```
    }
}
```

Khi sử dụng:

```
Horn horn = new Horn();

var car = new Car(horn); // horn inject vào car
car.Beep(); // Beep - beep - beep ...
```

Code trên hoạt động tương tự trường hợp thứ nhất. Bằng cách khai bảo Horn là một biến thành viên trong Car, Car đã có một dependency là đối tượng lớp Horn, dependency này không phải do Car tạo ra, nó được bơm vào (cung cấp) thông qua phương thức khởi tạo của nó.

Kết quả gọi `car.Beep();` có vẻ kết quả vẫn như trên, nhưng code mới này có một số lợi ích. Ví dụ, nếu sửa cập nhật lại Horn bằng cách sửa phương thức khởi tạo của nó, thì lớp Car không phải sửa gì!

```
public class Horn {
    int level; // thêm độ lớn còi xe
    public Horn (int level) => this.level = level; // thêm khởi tạo level

    public void Beep () => Console.WriteLine ("Beep - beep - beep ...");
}
```

```
Horn horn = new Horn(10);

var car = new Car(horn); // horn inject vào car
car.Beep(); // Beep - beep - beep ...
```

Như vậy, viết code mà các dependency có thể đưa vào từ bên ngoài (chủ yếu qua phương thức khởi tạo), giúp cho các dịch vụ tương đối độc lập nhau. Nó là cơ sở để có thể dùng các Framework hỗ trợ DI (tự động phân tích tạo dịch vụ, dependency)

Các kiểu Dependency Injection

Từ cách thức một dependency được đưa vào đối tường cần nó thì được phân chia có ba kiểu DI:

- **Inject thông qua phương thức khởi tạo:** cung cấp các Dependency cho đối tượng thông qua hàm khởi tạo (như đã thực hiện ở ví dụ trên) - tập trung vào cách này vì thư viện .NET hỗ trợ sẵn
- **Inject thông qua setter:** tức các Dependency như là thuộc tính của lớp, sau đó inject bằng gán thuộc tính cho Depedency `object.denpendency = obj;`
- **Inject thông qua các Interface** - xây dựng Interface có chứa các phương thức Setter để thiết lập dependency, interface này sử dụng bởi các lớp triển khai, lớp triển khai phải định nghĩa các setter quy định trong interface

Trong ba kiểu Inject thì Inject qua phương thức khởi tạo rất phổ biến vì tính linh hoạt, mềm dẻo, dễ xây dựng thư viện DI...

Ví dụ, xây dựng lại code phần trên với kỹ thuật **INJECT BẰNG HÀM TẠO** kết hợp với thiết kế phụ thuộc lỏng lẻo giữa các dependency (Dependency Inverse) ở trên.

Đầu tiên xây dựng một interface là IHorn

```
public interface IHorn {
    void Beep ();
}
```

Lớp Car được xây dựng để sử dụng IHorn như là Dependency

```
public class Car {
    IHorn horn;                                 // IHorn (Interface) là một Dependecy của Car
    public Car (IHorn horn) => this.horn = horn; // Inject từ hàm  tạo
```

```
    public void Beep () => horn.Beep ();
}
```

Với cách triển khai DI bằng phương thức khởi tạo như vậy, kết hợp với sự phụ thuộc lỏng giữa Car và các lớp triển khai IHorn. Thì sử dụng Car tạo ra các đối tượng cụ thể rất linh hoạt và độc lập với nhiều loại đối tượng triển khai IHorn, ví dụ thử tạo ra hai loại còi một cái loại lơn và một cái loại nhẹ

```
public class HeavyHorn : IHorn
{
    public void Beep() => Console.WriteLine("(kêu to lắm) BEEP BEEP BEEP ...");
}

public class LightHorn : IHorn
{
    public void Beep() => Console.WriteLine("(kêu bé lắm) beep  bep  bep ...");
}
```

Lúc này khi sử dụng, Car của bạn có dùng loại còi nào thì dùng - logic code giống nhau

```
Car car1 = new Car(new HeavyHorn());
car1.Beep();                            // (kểu to lắm) BEEP BEEP BEEP ...

Car car2 = new Car(new LightHorn());
car2.Beep();                            // (kểu bé lắm) beep  bep  bep ...
```

Inject bằng phương thức khởi tạo nên tập trung vào đó, vì các thư viện DI hỗ trợ tốt

Toàn bộ phần trên là lý thuyết cơ bản, triển khai thực tế thì cần có một dịch vụ trung tâm gọi là DI Container, tại đó các lớp (dịch vụ) đăng ký vào, sau đó khi sư dụng dịch vụ nào nó tự động tạo ra dịch vụ đó, nếu dịch vụ đó cần dependency nào nó cũng tự tạo dependency và tự động bơm vào dịch vụ cho chung ta. Để tự xây dựng ra một DI Container rất phức tạp, nên ở đây ta không cố gắng xây dựng một DI Container riêng, thay vào đó ta sẽ sử dụng các thư viện hỗ trợ sẵn cho .NET

DI Container

Mục đích sử dụng DI, để tạo ra các đối tượng dịch vụ kéo theo là các Dependency của đối tượng đó. Để làm điều này ta cần sử dụng đến các thư viện, có rất nhiều thư viện DI - Container (cơ chứa chứa và quản lý các dependency) như: Windsor, Unity Ninject, DependencyInjection ...

Trong đó DependencyInjection là DI Container mặc định của `ASP.NET Core`, phần này tìm hiểu về DI Container này Microsoft.Extensions.DependencyInjection

Trước tiên phải đảm bảo tích hợp Package Microsoft.Extensions.DependencyInjection vào dự án

```
dotnet add package Microsoft.Extensions.DependencyInjection
```

Sau đó sử dụng namespace

```
using  Microsoft.Extensions.DependencyInjection;
```

Từ đây các đối tượng lớp, các dependency ta gọi chúng là các dịch vụ (service)!

Lớp ServiceCollection

`ServiceCollection` là lớp triển khai giao diện `IServiceCollection` nó có chức năng quản lý các dịch vụ (đăng ký dịch vụ - tạo dịch vụ - tự động inject - và các dependency của đích vụ ...). ServiceCollection là trung tâm của kỹ thuật DI, nó là thành phần rất quan trọng trong ứng dụng ASP.NET

Các sử dụng cơ bản như sau:

- Khởi tạo đối tượng `ServiceCollection`, sau đó đăng ký (lớp) các dịch vụ vào `ServiceCollection`

- Từ ServiceCollection phát sinh ra đối tượng `ServiceProvider`, từ đối tượng này truy vấn lấy ra các dịch vụ cụ thể khi cần.

ServiceLifetime: Mỗi dịch vụ (lớp) khi đăng ký vào `ServiceCollection` thì có một đối tượng `ServiceDescriptor` chứa thông tin về dịch vụ đó, căn cứ vào ServiceDescriptor để ServiceCollection khởi tạo dịch vụ khi cần. Trong ServiceDescriptor có thuộc tính Lifetime để xác định dịch vụ tạo ra tồn tại trog bao lâu. Lifetime có kiểu `ServiceLifetime` (kiểu enum) có các giá trị cụ thể:

`Scoped`	1	Một bản thực thi (instance) của dịch vụ (Class) được tạo ra cho mỗi phạm vi, tức tồn tại cùng với sự tồn tại của một đối tượng kiểu `ServiceScope` (đối tượng này tạo bằng cách gọi `ServiceProvider.CreateScope`, đối tượng này hủy thì dịch vụ cũng bị hủy).
`Singleton`	0	Duy nhất một phiên bản thực thi (instance of class) (dịch vụ) được tạo ra cho hết vòng đời của ServiceProvider
`Transient`	2	Một phiên bản của dịch vụ được tạo mỗi khi được yêu cầu

Để bắt đầu sử dụng, khởi tạo ServiceCollection như sau

```
var services = new ServiceCollection();
```

Khi đã có đối tượng bạn có thể thực hiện các thao tác như đăng ký dịch vụ vào ServiceCollection (DI container), lấy đối tượng lớp `ServiceProvider` qua đó để truy vấn lấy các dịch vụ ...

Một số phương thức của **ServiceCollection**, trong các phương thức có tham số thì kiểu như sau:

- `ServiceType` : Kiểu (tên lớp) dịch vụ
- `ImplementationType` : Kiểu (tên lớp) sẽ tạo ra đối tượng dịch vụ theo tên `ServiceType`, cần đảm bảo ImplementationType là một lớp triển khai / kế thừa từ ServiceType, hoặc chính là ServiceType

Phương thức	Diễn giả
`AddSingleton<ServiceType, ImplementationType>()`	Đăng ký dịch vụ kiểu `Singleton`. Ví dụ: `services.AddSingleton<IHorn, HeavyHorn>();`
Nếu ServiceType giống ImplementationType có thể viết `AddSingleton<ServiceType>()`	Đăng ký dịch vụ kiểu `IHorn`, mà khi dịch vụ `IHorn` được yêu cầu nó tạo ra và trả về đối tượng kiểu `HeavyHorn`, do là Singleton chỉ một đối tượng của dịch vụ được tạo, nếu đã có yêu cầu sau trả về đối tượng lần trước tạo (gọi ra thế nào ở phần sau).
`AddTransient<ServiceType, ImplementationType>()` Hoặc `AddTransient<ServiceType>()`	Đăng ký dịch vụ thuộc loại `Transient`, luôn tạo mới mỗi khi có yêu cầu lấy dịch vụ.
`AddScoped<ServiceType, ImplementationType>()`	Đăng ký vào hệ thống dịch vụ kiểu `Scoped`
`BuildServiceProvider()`	Tạo ra đối tượng lớp `ServiceProvider`, đối tượng này dùng để triệu gọi, tạo các dịch vụ thiết lập ở trên.

Các phương thức `AddSingleton`, `AddTransient`, `AddScoped` còn có bản quá tải mà tham số là một callback delegate tạo đối tượng. Nó là cách triển khai pattern factory

Lớp ServiceProvider

Lớp ServiceProvider cung cấp cơ chế để lấy ra (tạo và inject nếu cần) các dịch vụ đăng ký trong ServiceCollection. Đối tượng ServiceProvider được tạo ra bằng cách gọi phương thức `BuildServiceProvider()` của `ServiceCollection`

```
var serviceprovider = services.BuildServiceProvider();
```

Một số phương thức trong ServiceProvider

Phương thức	Diễn giả

`GetService<ServiceType>()`	Lấy dịch vụ có kiểu ServiceType - trả về `null` nếu dịch vụ không tồn tại
	`// lấy đối tượng triển khai IHorn`
	`services.GetService<IHorn>()`

	Lấy dịch vụ có kiểu ServiceType - phát sinh Exception nếu dịch vụ không tồn tại
`GetRequiredService(ServiceType)`	`// lấy đối tượng kiểu Car` `services.GetRequiredService(Car)`
	Tạo một phạm vi mới, thường dùng khi sử dụng những dịch vụ có sự ảnh hưởng theo `Scoped`, sử dụng cơ bản:

`CreateScope()`	`using (var scope = serviceprovider.CreateScope())`
	`{`
	`  //  Lấy Service trong một pham vi`
	`  var myservice = scope.ServiceProvider.GetService<ServiceType>();`
	
	`  // ...`
	`  // ... Ra khỏi phạm vị, các Service kiểu Scoped tạo ra trong phạm vi`
	`  //     using ...  sẽ bị hủy`
	`}`

Để cụ thể hơn sử dụng DI với thư viện ServiceCollection, ta sẽ thực hành cho vài trường hợp:

Sử dụng ServiceCollection cơ bản

Xét lại các lớp IClassA, IClassB, ClassA ... đã xây dựng ở trên, áp dụng vài trường hợp riêng lẻ sau:

Dịch vụ được đăng ký là Singleton

```
// Đăng ký dịch vụ IClassC tương ứng với đối tượng ClassC
services.AddSingleton<IClassC, ClassC>();

var provider = services.BuildServiceProvider();

for (int i = 0; i < 5; i++)
{
    var service = provider.GetService<IClassC>();
    Console.WriteLine(service.GetHashCode());
}
// ClassC is created
// 32854180
// 32854180
// 32854180
// 32854180
// 32854180
// Gọi 5 lần chỉ 1 dịch vụ (đối tượng) được tạo ra
```

Dịch vụ được đăng ký là Transient

```
services.AddTransient<IClassC, ClassC>();

var provider = services.BuildServiceProvider();

for (int i = 0; i < 5; i++)
{
    var service = provider.GetService<IClassC>();
    Console.WriteLine(service.GetHashCode());
}
// ClassC is created
// 32854180
// ClassC is created
// 27252167
// ClassC is created
// 43942917
// ClassC is created
// 59941933
// ClassC is created
// 2606490
// Gọi 5 lần có 5 dịch vụ được tạo ra
```

Dịch vụ được đăng ký là Scoped

```
ServiceCollection services = new ServiceCollection();

// Đăng ký dịch vụ IClassC tương ứng với đối tượng ClassC
services.AddScoped<IClassC, ClassC>();

var provider = services.BuildServiceProvider();

// Lấy dịch vụ trong scope toàn cục
for (int i = 0; i < 5; i++)
{
    var service = provider.GetService<IClassC>();
    Console.WriteLine(service.GetHashCode());
}

// Tạo ra scope mới
using (var scope = provider.CreateScope())
{
    // Lấy dịch vụ trong scope
    for (int i = 0; i < 5; i++)
    {
        var service = scope.ServiceProvider.GetService<IClassC>();
        Console.WriteLine(service.GetHashCode());
    }
}
// ClassC is created
// 32854180
// 32854180
// 32854180
// 32854180
// 32854180
// ClassC is created
// 27252167
// 27252167
// 27252167
// 27252167
// 27252167
// Mỗi scope tạo ra một loại dịch vụ
```

Kiểm tra tạo và inject các dịch vụ đăng ký trong ServiceCollection

```
// ClassA
// IClassB -> ClassB,  ClassB1
// IClassC -> ClassC,  ClassC1

ServiceCollection services = new ServiceCollection();

services.AddSingleton<ClassA, ClassA>();
services.AddSingleton<IClassC, ClassC>();
services.AddSingleton<IClassB, ClassB>();

var provider = services.BuildServiceProvider();

ClassA service_a = provider.GetService<ClassA>();

service_a.ActionA();

// ClassC is created
// ClassB is created
// ClassA is created
// Action in ClassA
// Action in ClassB
// Action in ClassC
```

Sử dụng Delegate / Factory đăng ký dịch vụ

Sử dụng Delegate đăng ký

Các phương thức để đăng dịch vụ vào ServiceCollection như AddSingleton, AddSingleton, AddTransient còn có phiên bản *(nạp chồng)* nó nhận tham số là delegate trả về đối tượng dịch vụ có kiểu ImplementationType. Ví dụ AddSingleton, cú pháp đó là:

```
services.AddSingleton<ServiceType>((IServiceProvider provider) => {
    // các chỉ thị
    // ...
    return (đối tượng kiểu ImplementationType);
});
```

Trong cú pháp trên thì Delegate đó là

```
(IServiceProvider provider) => {
    // các chỉ thị
    // ...
    return (đối tượng kiểu ImplementationType);
}
```

Nó nhận tham số là `IServiceProvider` (chính là đối tượng được sinh ra bởi ServiceCollection.BuildServiceProvider()), Delegate phải trả về một đối tượng triển khai từ ServiceType

Ví dụ, tạo ra thêm lớp ClassB2

```
class ClassB2 : IClassB
{
    IClassC c_dependency;
    string message;
    public ClassB2(IClassC classc, string mgs)
    {
        c_dependency = classc;
        message = mgs;
        Console.WriteLine("ClassB2 is created");
    }
    public void ActionB()
    {
        Console.WriteLine(message);
        c_dependency.ActionC();
    }
}
```

Lớp trên khi khởi tạo cần có hai tham số. Vậy khi đăng ký vào dịch vụ theo cách:

```
services.AddSingleton<IClassB, ClassB2>();
```

Thì tham số khởi tạo IClassC được inject, trong khi đó tham số chuỗi string không đăng ký sẽ dẫn tới lỗi. Lúc này có thể đăng ký với Delegate và truyền chuỗi khởi tạo cụ thể, ví dụ:

```
services.AddSingleton<IClassB>((IServiceProvider serviceprovider) => {
    var service_c = serviceprovider.GetService<IClassC>();
    var sv = new ClassB2(service_c, "Thực hiện trong ClassB2");
    return sv;
});
```

Lúc này nếu lấy ra dịch vụ `IClassB` (hoặc khi nó Inject vào dịch vụ khác) , nếu dịch vụ đó chưa có nó sẽ thi hành Delegate để tạo dịch vụ.

Sử dụng Factory đăng ký

Delegate trên bạn có thể khai báo thành một phương thức, một phương thức cung cấp cơ chế để tạo ra đối tượng mong muốn gọi là Factory.

```
// Factory nhận tham số là IServiceProvider và trả về đối tượng dịch vụ cần tạo
public static ClassB2 CreateB2Factory(IServiceProvider serviceprovider)
{
    var service_c = serviceprovider.GetService<IClassC>();
    var sv = new ClassB2(service_c, "Thực hiện trong ClassB2");
```

```
    return sv;
}
```

Lúc này có thể sử dụng Factory trên để đăng ký IClassB.

```
services.AddSingleton<IClassB>(CreateB2Factory);
```

Sử dụng Options khởi tạo dịch vụ trong DI

Khi một dịch vụ đăng ký trong DI, nếu nó cần các tham số để khởi tạo thì ta có thể Inject các tham số khởi tạo là các đối tượng như cách làm ở trên. Tuy nhiên để tách bạch giữa các dịch vụ và các thiết lập truyền vào để khởi tạo dịch vụ thì trong ServiceCollection hỗ trợ sử dụng giao diện IOptions.

Trước tiên cần thêm package `Microsoft.Extensions.Options`

```
dotnet add package Microsoft.Extensions.Options
```

Sử dụng namepace:

```
using Microsoft.Extensions.Options;
```

Các thiết lập cho một dịch vụ thường thiết kế là một lớp chứa các thuộc tính, ví dụ có lớp là `MyService`, khi nó khởi tạo thì cần một đối tượng lớp `MyServiceOptions`

```
public class MyServiceOptions
{
    public string data1 { get; set; }
    public int data2 { get; set; }
}
```

Để có thể Inject MyServiceOptions vào MyService theo nguyên tắc của ServiceCollection thì lớp MyService thiết kế sử dụng IOption làm tham số khởi tạo như sau:

```
public class MyService
{
    public string data1 { get; set; }
    public int data2 { get; set;}

    // Tham số khởi tạo là IOptions, các tham số khởi tạo khác nếu có khai báo như bình thường
    public MyService(IOptions<MyServiceOptions> options)
    {
        // Đọc được MyServiceOptions từ IOptions
        MyServiceOptions opts = options.Value;
        data1 = opts.data1;
        data2 = opts.data2;
    }
    public void PrintData() => Console.WriteLine($"{data1} / {data2}");
}
```

Khi tham số khởi tạo có kiểu IOptions, thì nó được Inject vào từ một tập hợp các IOptions riêng biệt với các dịch vụ, và các IOptions nạp vào ServiceCollection bằng phương thức **Configure**, cách làm như sau:

Để đăng ký vào hệ thống một lớp `T` là cấu hình, sau này sẽ có các đối tượng kiểu `IOptions<T>` có thể Inject vào thì làm như sau:

```
services.Configure<T>(
    (T options)
    {
        // T là tên lớp chứa các thiết lập
        // Hãy thiết lập các giá trị cho options
    }
);
```

Cụ thể đăng ký `MyServiceOptions` tạo ra đối tượng cụ thể để sau này có thể Inject vào MyService:

```
services.Configure<MyServiceOptions>(
    options => {
        options.data1 = "Xin chao cac ban";
        options.data2  = 2021;
    }
);
```

Lúc này sử dụng

```
services.AddSingleton<MyService>();
var provider = services.BuildServiceProvider();

var myservice = provider.GetService<MyService>();
myservice.PrintData();

// Kết quả:
// Xin chao cac ban / 2021
```

Như vậy DI Container, đã giúp tạo Config (đối tượng MyServiceOptions) và Inject nó cho dịch vụ khi tạo (dịch vụ MyService)

Lưu ý 1: nếu muốn lấy đối tượng lớp `MyServiceOptions` trong DI Container, thì:

```
var config = serviceprovider.GetService<IOptions<MyServiceOptions>>()
MyServiceOptions myServiceOptions = config.Value;
```

Lưu ý 2: nếu muốn tạo trực tiếp đối tượng `IOptions<MyServiceOptions>`, dành cho trường hợp muốn tạo `MyService` trực tiếp không thông qua DI Container. Thì dùng phương thức Factory `Options.Create(obj)`, ví dụ:

```
var opts = Options.Create(new MyServiceOptions() {
    data1 = "DATA-DATA-DATA-DATA-DATA",
    data2 = 12345
});
MyService myService = new MyService(opts);
myService.ShowData();
```

Sử dụng cấu hình từ File cho DI Container

Ở ví dụ trên, các giá trị dữ liệu trong `MyServiceOptions` (như `data1`, `data2`) có thể lưu ở file sau đó nạp vào khi chương trình thực thi. Các file cấu hình này hỗ trợ nhiều định dạng như XML, INI, JSON ... (cần cài đặt gói tương ứng)

Trước tiên thêm package Microsoft.Extensions.Configuration và Microsoft.Extensions.Options.ConfigurationExtensions

```
dotnet add package Microsoft.Extensions.Configuration
dotnet add package Microsoft.Extensions.Options.ConfigurationExtensions
```

Sau đó, muốn dùng định dạng nào thì thêm Package tương ứng, ví dụ dùng JSON:

```
dotnet add package Microsoft.Extensions.Configuration.Json
dotnet add package Microsoft.Extensions.Configuration.Ini
dotnet add package Microsoft.Extensions.Configuration.Xml
```

Sử dụng namespace với kiểu file Json

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Configuration.Json;
```

ConfigurationBuilder

Lớp **ConfigurationBuilder**, giúp nạp các cấu hình lưu trong file config, từ đó build ra đối tượng `ConfigurationRoot`, đối tượng này truy cập đến các cấu hình bằng chỉ toán tử chỉ số `[key]`

Giá sử cấu hình lưu tại file `appsettings.json`, thì nạp cấu hình đó để có được `ConfigurationRoot`

```
var configBuilder = new ConfigurationBuilder()
                        .SetBasePath(Directory.GetCurrentDirectory())      // file config ở thư mục hiện tại
                        .AddJsonFile("appsettings.json");                  // nạp config định dạng JSON
var configurationroot = configBuilder.Build();                             // Tạo configurationroot
```

Khi có đối tượng `ConfigurationRoot`, lấy một Section nào đó bằng phương thức `GetSection(key)`, nó trả về đối tượng biểu diễn nút cấu hình (JSON), giá trị của nút truy cập bằng thuộc tính `Value`

Ví dụ, tạo file `appsettings.json` với nội dung

```
{
    "MyServiceOptions" : {
        "data1" : "ABCDE",
        "data2" : 123456
    },

    "Option2" : {
        "key1" : "Test",
        "Key2" : 789
    }
}
```

Truy cập config

```
var cf1 = configurationroot.GetSection("Option2").GetSection("key1").Value; // Test
var cf2 = configurationroot.GetSection("Option2").GetSection("key2").Value; // 789
var cf3 = configurationroot.GetSection("Option2").GetSection("key3").Value; // null, không tồn tại
```

Như vậy đã có thể nạp và đọc các giá trị lưu trong file cấu hình json.

Nạp config json vào IOption

Trong file JSON có một Section có tên `MyServiceOptions`, ta có thể gán các giá trị trong Section đó vào MyServiceOptions trong ServiceCollection bằng cách

```
// Nạp mở phương thức mở rộng
services.AddOptions();
services.Configure<MyServiceOptions>(configurationroot.GetSection("MyServiceOptions"));

// Lưu ý: phải cài package  ConfigurationExtensions
// dotnet add package Microsoft.Extensions.Options.ConfigurationExtensions
```

Hoàn thiện ví dụ trên

```
var configBuilder = new ConfigurationBuilder()
           .SetBasePath(Directory.GetCurrentDirectory())      // file config ở thư mục hiện tại
           .AddJsonFile("appsettings.json");                  // nạp config định dạng JSON
var configurationroot = configBuilder.Build();                // Tạo configurationroot

ServiceCollection services = new ServiceCollection();

services.AddOptions();
services.Configure<MyServiceOptions>(configurationroot.GetSection("MyServiceOptions"));

services.AddSingleton<MyService>();

var provider = services.BuildServiceProvider();

var myservice = provider.GetService<MyService>();
myservice.PrintData();

// Kết quả:
// ABCDE / 123456
```

Kỹ thuật DI với thư viện DependencyInjection ở trên là kiến rất cần nắm vững, nó là cơ sở để học các các mô hình lập trình hiện đại, nhất là sau này áp dụng với Asp.Net Core bạn cần hiểu nó.

Mục lục bài viết

7 bình luận Sắp xếp theo **Mới nhất**

Viết bình luận...

Dương Chiến
cảm ơn thầy
Thích · Phản hồi · 22 tuần

Nguyễn Hữu Tâm
Bài viết rất chi tiết và đầy đủ
Mình chưa biết nhiều về C# nên ko hiểu ở ví dụ mà lớp B có tham số là lớp A ở hàm khởi tạo
Nếu request tạo 1 đối tượng B thì framework sẽ tự động inject 1 đối tương A vào
Nhưng làm sao framework biết được lớp B cần lớp A ở hàm khởi tạo mà ko phải là cần 1 lớp nào khác ?
Thích · Phản hồi · 3 năm

Văn Sơn Nguyễn
Thì do có tham số ở hàm khởi tạo nên framework tự detect ra thôi bạn
Thích · Phản hồi · 1 · 2 năm

Born To Try
Bạn đã đăng kí khi inject vào ServiceClollection đó mấy cái service.AddSingleton, scope... : services.AddSingleton<IClassB>(CreateB2Factory); . Khi đó nó sẽ tìm trong ServicesCollection và lấy ra thôi.
Thích · Phản hồi · 1 năm

Cao Van Thuy
thanks
Thích · Phản hồi · 3 năm

Huỳnh Công
thanks
Thích · Phản hồi · 2 năm

Le Viet
Bài viết rất chi tiết. Thanks Bạn
Thích · Phản hồi · 2 năm

Top-level statement trong lập trình C# .NET 6 Tạo các Requirement và Authorization handler chứng thực quyền truy cập Authorize trong ASP.NET Core Lớp Uri Dns Ping và các lớp về Networking trong lập trình C# NET Core Code C# chuyển số thành chữ (ASP.NET Core MVC) Triển khai ứng dụng ASP.NET trên Server Linux với Kestrel Apache Nginx Lập trình bất đồng bộ asynchronou C# C Sharp với bất đồng bộ theo mô hình tác vụ Linq trong lập trình C# .NET - thực hình ví dụ Linq Kiểu vô danh và kiểu động dynamic trong C# C Sharp Các phương thức mở rộng trong C# C Sharp Lớp Dictionary HashSet trong C# C Sharp
Attribute Annotation (Bài trước)
(Bài tiếp) (Multithreading) Parallel
Giới thiệuPrivacyTừ điển Anh - ViệtChạy SQLRegExpCubic-bezierUnix timestampKý tự HTMLcalories, chỉ số BMRchỉ số khối cơ thể BMITạo QR CodeLịch vạn niên Liên hệ RSS

Đây là blog cá nhân, tôi ghi chép và chia sẻ những gì tôi học được ở đây về kiến thức lập trình PHP, Java, JavaScript, Android, C# ... và các kiến thức công nghệ khác
Developed by XuanThuLab