深入解析.NET依赖注入框架：方法、步骤及实例代码一网打尽

作者：微信公众号：【架构师老卢】 1-24 9:0 133

概述：本文详解.NET中多种依赖注入框架（如Castle Windsor、Unity等）的原理、方法和步骤，并提供每个框架的实例源代码。开发者可通过深入了解这些框架，提升代码质量和灵活性。

依赖注入（Dependency Injection，DI）是一种软件设计模式，用于实现松耦合的组件之间的依赖关系。在.NET生态系统中，有多个依赖注入框架可供选择，每个框架都有其独特的原理和用法。在本文中，我们将详细讨论以下依赖注入框架：

- Castle Windsor
- Unity
- Autofac
- Dryloc
- Ninject
- Spring.Net
- Lamar
- LightInject
- Simple Injector
- Microsoft.Extensions.DependencyInjection
- Scrutor
- VS MEF
- TinyIoC
- Stashbox

依赖注入框架原理

Castle Windsor

Castle Windsor是一个开源的.NET依赖注入框架，采用IoC（Inversion of Control）容器的思想。其原理基于反射和配置文件，通过注册和解析服务来管理对象的生命周期。

Unity

Unity是由Microsoft维护的开源依赖注入框架，通过配置容器来管理对象之间的依赖关系。Unity使用配置文件或者代码方式进行注册和解析。

Autofac

Autofac是一个轻量级的IoC容器，采用Lambda表达式进行注册和解析服务。它支持构造函数注入、属性注入和方法注入。

Dryloc

Dryloc是一个快速、轻量级的依赖注入框架，采用编译时生成代码的方式实现高性能的依赖注入。它支持构造函数注入、属性注入和方法注入。

Ninject

Ninject是一个轻量级的依赖注入框架，通过模块化的方式进行配置。它支持构造函数注入和属性注入。

Spring.Net

Spring.Net是基于Java中Spring框架的一个.NET版本，它提供了完整的IoC容器。Spring.Net使用XML或者属性进行配置。

Lamar

Lamar是一个快速、灵活的IoC容器，与ASP.NET Core兼容。它支持构造函数注入、属性注入和方法注入。

LightInject

LightInject是一个轻量级的依赖注入框架，采用属性注入的方式。它支持构造函数注入和属性注入。

Simple Injector

Simple Injector是一个简单、高性能的IoC容器，通过约定进行注册。它主张在启动时发现并验证所有依赖关系。

Microsoft.Extensions.DependencyInjection

Microsoft.Extensions.DependencyInjection是ASP.NET Core官方推荐的依赖注入框架，简单易用。它通过扩展方法进行注册和解析服务。

Scrutor

Scrutor是一个用于扩展Microsoft.Extensions.DependencyInjection的库，提供了更多的功能，如自动注册服务。

VS MEF

VS MEF是Visual Studio自带的MEF（Managed Extensibility Framework）实现，用于插件化开发。它支持构造函数注入和属性注入。

TinyIoC

TinyIoC是一个极简的依赖注入框架，适用于资源受限的环境。它支持构造函数注入和属性注入。

Stashbox

Stashbox是一个功能强大、灵活的依赖注入框架，支持构造函数注入、属性注入和方法注入。它具有高性能和低内存消耗的特点。

依赖注入方法

注册服务

在依赖注入框架中，注册服务是将服务类型与其实现类型进行关联的过程。具体的方法取决于每个框架的语法和规范。

解析服务

解析服务是从依赖注入容器中获取已注册的服务实例的过程。通过依赖注入框架提供的方法，可以根据服务类型获取相应的实例。

生命周期管理

依赖注入框架通常支持多种生命周期管理方式，如瞬时（Transient）、单例（Singleton）、作用域（Scoped）等。生命周期管理决定了服务实例的生命周期和管理方式。

依赖注入步骤

1. **引入依赖注入框架包**
   - 使用NuGet或其他包管理工具引入相应的依赖注入框架包。
2. **注册服务**
   - 根据框架的语法，使用注册方法将服务类型和实现类型关联起来。这通常在应用程序启动时完成。
3. **解析服务**
   - 在需要使用服务的地方，通过依赖注入容器解析服务，获取服务实例。
4. **生命周期管理**
   - 根据应用程序的需求，选择合适的生命周期管理方式，确保服务的生命周期符合预期。

实例源代码

Castle Windsor

```
// 安装Castle.Windsor NuGet包
// 注册服务
var container = new WindsorContainer();
container.Register(Component.For<IService>().ImplementedBy<MyService>());
// 解析服务
var service = container.Resolve<IService>();
```

Unity

```
// 安装Unity NuGet包
// 注册服务
var container = new UnityContainer();
container.RegisterType<IService, MyService>();
// 解析服务
var service = container.Resolve<IService>();
```

Autofac

```
// 安装Autofac NuGet包
// 注册服务
var builder = new ContainerBuilder();
builder.RegisterType<MyService>().As<IService>();
var container = builder.Build();
// 解析服务
var service = container.Resolve<IService>();
```

Dryloc

```
// 安装Dryloc NuGet包
// 注册服务
var container = new Container();
container.Register<IService, MyService>();
// 解析服务
var service = container.Resolve<IService>();
```

Ninject

```
// 安装Ninject NuGet包
// 注册服务
var kernel = new StandardKernel();
kernel.Bind<IService>().To<MyService>();
// 解析服务
var service = kernel.Get<IService>();
```

Spring.Net

```
// 安装Spring.Net NuGet包
// 配置文件中定义服务和实现
// 解析服务
var context = new XmlApplicationContext("spring-config.xml");
var service = (IService)context["myService"];
```

Lamar

```
// 安装Lamar NuGet包
// 注册服务
var container = new Container(x => x.For<IService>().Use<MyService>());
// 解析服务
var service = container.GetInstance<IService>();
```

LightInject

```
// 安装LightInject NuGet包
// 注册服务
var container = new ServiceContainer();
container.Register<IService, MyService>();
// 解析服务
var service = container.GetInstance<IService>();
```

Simple Injector

```
// 安装SimpleInjector NuGet包
// 注册服务
var container = new Container();
container.Register<IService, MyService>();
container.Verify(); // 验证依赖关系
// 解析服务
var service = container.GetInstance<IService>();
```

Microsoft.Extensions.DependencyInjection

```
// ASP.NET Core中，无需安装包，已内置
// 注册服务
var serviceProvider = new ServiceCollection()
    .AddTransient<IService, MyService>()
    .BuildServiceProvider();
// 解析服务
var service = serviceProvider.GetService<IService>();
```

Scrutor

```
// 安装Scrutor NuGet包
// 在Microsoft.Extensions.DependencyInjection基础上使用
// 自动注册服务
var serviceProvider = new ServiceCollection()
    .Scan(scan => scan.FromAssemblyOf<MyService>().AddClasses().AsImplementedInterfaces().WithTransientLifetime())
    .BuildServiceProvider();
// 解析服务
var service = serviceProvider.GetService<IService>();
```

VS MEF

```
// 使用Visual Studio MEF框架，无需安装包，已内置
// 导入Microsoft.VisualStudio.Composition命名空间
// 注册服务
var container = new CompositionContainer(new AssemblyCatalog(Assembly.GetExecutingAssembly()));
var service = container.GetExportedValue<IService>();
```

TinyIoC

```
// 安装TinyIoC NuGet包
// 注册服务
var container = new TinyIoCContainer();
container.Register<IService, MyService>();
// 解析服务
var service = container.Resolve<IService>();
```

Stashbox

```
// 安装Stashbox NuGet包
// 注册服务
var container = new StashboxContainer();
container.Register<IService, MyService>();
// 解析服务
var service = container.Resolve<IService>();
```

注意事项及建议

- **生命周期选择**
  - 选择适当的生命周期管理方式，确保服务的生命周期符合应用程序的需求，避免产生意外的副作用。
- **注册方式**
  - 根据框架的语法和规范选择适当的注册方式，有的框架支持Lambda表达式、XML配置、属性配置等不同方式。
- **验证依赖关系**
  - 在使用一些框架时，建议在启动时验证依赖关系，确保注册的服务能够正确解析。
- **性能考虑**
  - 对于性能要求较高的应用程序，可以选择一些轻量级、快速的依赖注入框架，避免不必要的性能开销。

依赖注入是一种优秀的设计模式，能够提高代码的可维护性、可测试性和可扩展性。选择适合项目需求的依赖注入框架是很重要的一步。本文详细介绍了.NET生态系统中常见的依赖注入框架的原理、方法和步骤，并提供了每个框架的实例源代码。在实施过程中，注意事项和建议也对开发者有一定的指导意义。通过深入了解这些依赖注入框架，开发者可以更好地应用它们来构建健壮、灵活的应用程序。