## [解读ASP.NET 5 & MVC6系列（7）：依赖注入](http://www.cnblogs.com/TomXu/p/4496440.html)

2015-05-20 09:10 by 汤姆大叔, 22725 阅读, 30 评论, [收藏](http://www.cnblogs.com/TomXu/p/4496440.html), [编辑](https://i.cnblogs.com/EditPosts.aspx?postid=4496440)

在前面的章节（Middleware章节）中，我们提到了依赖注入功能（Dependency Injection），ASP.NET 5正式将依赖注入进行了全功能的实现，以便开发人员能够开发更具弹性的组件程序，MVC6也利用了依赖注入的功能重新对Controller和View的服务注入功能进行了重新设计；未来的依赖注入功能还可能提供更多的API，所有如果还没有开始接触依赖注入的话，就得好好学一下了。

在之前版本的依赖注入功能里，依赖注入的入口有MVC中的IControllerFactory和Web API中的IHttpControllerActivator中，在新版ASP.NET5中，依赖注入变成了最底层的基础支撑，MVC、Routing、SignalR、Entity Framrwork等都依赖于依赖注入的IServiceProvider接口，针对该接口微软给出了默认的实现ServiceProvider，以及Ninject和AutoFac版本的包装，当然你也可以使用其它第三方的依赖注入容器，如Castle Windsor等；一旦应用了第三方容器，所有的依赖解析都会被路由到该第三方容器上。

针对通用的依赖类型的解析与创建，微软默认定义了4种类别的生命周期，分别如下：

| **类型** | **描述** |
| --- | --- |
| Instance | 任何时间都只能使用特定的实例对象，开发人员需要负责该对象的初始化工作。 |
| Transient | 每次都重新创建一个实例。 |
| Singleton | 创建一个单例，以后每次调用的时候都返回该单例对象。 |
| Scoped | 在当前作用域内，不管调用多少次，都是一个实例，换了作用域就会再次创建实例，类似于特定作用内的单例。 |

# **类型注册与示例**

依赖注入类型的注册一般是在程序启动的入口中，如Startup.cs中的ConfigureServices中，该类的主要目的就是注册依赖注入的类型。由于依赖注入的主要体现是接口编程，所以本例中，我以接口和实现类的方式来举例。

首先声明一个接口ITodoRepository和实现类TodoRepository1，代码如下：

public interface ITodoRepository

{

IEnumerable<TodoItem> AllItems { get; }

void Add(TodoItem item);

TodoItem GetById(int id);

bool TryDelete(int id);

}

public class TodoItem

{

public int Id { get; set; }

public string Name { get; set; }

}

public class TodoRepository : ITodoRepository

{

readonly List<TodoItem> \_items = new List<TodoItem>();

public IEnumerable<TodoItem> AllItems

{

get { return \_items; }

}

public TodoItem GetById(int id)

{

return \_items.FirstOrDefault(x => x.Id == id);

}

public void Add(TodoItem item)

{

item.Id = 1 + \_items.Max(x => (int?)x.Id) ?? 0;

\_items.Add(item);

}

public bool TryDelete(int id)

{

var item = GetById(id);

if (item == null) { return false; }

\_items.Remove(item);

return true;

}

}

为了演示不同的声明周期类型，建议多实现几个类，比如TodoRepository2、TodoRepository3、TodoRepository4等，以便进行演示。

然后在ConfigureServices方法内注册接口ITodoRepository类型和对应的实现类，本例中根据不同的生命周期注册了不同的实现类，具体示例如下：

//注册单例模式，整个应用程序周期内ITodoRepository接口的示例都是TodoRepository1的一个单例实例

services.AddSingleton<ITodoRepository, TodoRepository1>();

services.AddSingleton(typeof(ITodoRepository), typeof(TodoRepository1)); // 等价形式

//注册特定实例模型，整个应用程序周期内ITodoRepository接口的示例都是固定初始化好的一个单例实例

TodoRepository2

services.AddInstance<ITodoRepository>(new TodoRepository2());

services.AddInstance(typeof(ITodoRepository), new TodoRepository2()); // 等价形式

//注册作用域型的类型，在特定作用域内ITodoRepository的示例是TodoRepository3

services.AddScoped<ITodoRepository, TodoRepository3>();

services.AddScoped(typeof(ITodoRepository), typeof(TodoRepository3));// 等价形式

//获取该ITodoRepository实例时，每次都要实例化一次TodoRepository4类

services.AddTransient<ITodoRepository, TodoRepository4>();

services.AddTransient(typeof(ITodoRepository), typeof(TodoRepository));// 等价形式

//如果要注入的类没有接口，那你可以直接注入自身类型，比如：

services.AddTransient<LoggingHelper>();

依赖注入的在MVC中的使用方式目前有三种，分别是Controller的构造函数、属性以及View中的Inject形式。其中构造函数注入和之前的MVC中的是一样的，示例代码如下：

public class TodoController : Controller

{

private readonly ITodoRepository \_repository;

/// 依赖注入框架会自动找到ITodoRepository实现类的示例，赋值给该构造函数

public TodoController(ITodoRepository repository)

{

\_repository = repository;

}

[HttpGet]

public IEnumerable<TodoItem> GetAll()

{

return \_repository.AllItems; //这里就可以使用该对象了

}

}

属性注入，则是通过在属性上加一个[FromServices]属性即可实现自动获取实例。

public class TodoController : Controller

{

// 依赖注入框架会自动找到ITodoRepository实现类的示例，赋值给该属性

[FromServices]

public ITodoRepository Repository { get; set; }

[HttpGet]

public IEnumerable<TodoItem> GetAll()

{

return Repository.AllItems;

}

}

注意：这种方式，目前只适用于Controller以及子类，不适用于普通类  
同时：通过这种方式，你可以获取到更多的系统实例对象，如ActionContext、HttpContext、HttpRequest、HttpResponse、 ViewDataDictionary、以及ActionBindingContext。

在视图中，则可以通过@inject关键字来实现注入类型的实例提取，示例如下：

@using WebApplication1

@inject ITodoRepository repository<div>

@repository.AllItems.Count()</div>

而最一般的使用方式，则是获取IServiceProvider的实例，获取该IServiceProvider实例的方式目前有如下几种（但范围不同）：

var provider1 = this.Request.HttpContext.ApplicationServices; 当前应用程序里注册的Servicevar provider2 = Context.RequestServices; // Controller中，当前请求作用域内注册的Servicevar provider3 = Resolver; //Controller中

然后通过GetService和GetRequiredService方法来获取指定类型的实例，示例如下：

var \_repository1 = provider1.GetService(typeof(ITodoRepository));var \_repository2 = provider1.GetService<LoggingHelper>();//等价形式//上述2个对象可能为空

var \_repository3 = provider1.GetRequiredService(typeof(ITodoRepository));var \_repository4 = provider1.GetRequiredService<LoggingHelper>();//等价形式//上述2个对象肯定不为空，因为如果为空的话，会自动抛异常出来

# **普通类的依赖注入**

在新版的ASP.NET5中，不仅支持上面我们所说的接口类的依赖注入，还支持普通的类型的依赖注入，比如我们生命一个普通类，示例如下：

public class AppSettings

{

public string SiteTitle { get; set; }

}

上述普通类要保证有无参数构造函数，那么注册的用法，就应该像如下这样：

services.Configure<AppSettings>(app =>

{

app.SiteTitle = "111";

});

使用的时候，则需要获取IOptions<AppSettings>类型的实例，然后其Options属性即是AppSettings的实例，代码如下：

var appSettings = app.ApplicationServices.GetRequiredService<IOptions<AppSettings>>().Options;

当然，我们也可以在视图中，使用@inject语法来获取实例，示例代码如下：

@inject IOptions<AppSettings> AppSettings

<title>@AppSettings.Options.SiteTitle</title>

# **基于Scope生命周期的依赖注入**

## **普通的Scope依赖注入**

基于Scope作用域的实例在创建的时候需要先创建作用域，然后在该作用域内再获取特定的实例，我们看看一个示例并对其进行验证。首先，注册依赖注入类型，代码如下：

services.AddScoped<ITodoRepository, TodoRepository>();

然后创建作用域，并在该作用域内获取实例：

var serviceProvider = Resolver;

var scopeFactory = serviceProvider.GetService<IServiceScopeFactory>(); //获取Scope工厂类using (var scope = scopeFactory.CreateScope()) // 创建一个Scope作用域

{

var containerScopedService = serviceProvider.GetService<ITodoRepository>(); //获取普通的实例

var scopedService1 = scope.ServiceProvider.GetService<ITodoRepository>(); //获取当前Scope的实例

Thread.Sleep(200);

var scopedService2 = scope.ServiceProvider.GetService<ITodoRepository>(); //获取当前Scope的实例

Console.WriteLine(containerScopedService == scopedService1); // 输出：False

Console.WriteLine(scopedService1 == scopedService2); //输出：True

}

另外，Scope也可以进行嵌套，嵌套的内外作用域所获取的实例也是不相同的，实例代码如下：

var serviceProvider = Resolver;

var outerScopeFactory = serviceProvider.GetService<IServiceScopeFactory>();using (var outerScope = outerScopeFactory.CreateScope()) //外部Scope作用域

{

var innerScopeFactory = outerScope.ServiceProvider.GetService<IServiceScopeFactory>();

using (var innerScope = innerScopeFactory.CreateScope()) //内部Scope作用域

{

var outerScopedService = outerScope.ServiceProvider.GetService<ITodoRepository>();

var innerScopedService = innerScope.ServiceProvider.GetService<ITodoRepository>();

Console.WriteLine(outerScopedService == innerScopedService); // 输出：False

}

}

## **基于HTTP请求的Scope依赖注入**

在之前很多流行的DI容器中，针对每个请求，在该请求作用域内保留一个单实例对象是很流行的，也就是在每次请求期间一个类型的对象实例只会创建一次，这样可以大大提高性能。

在ASP.NET5中，基于HTTP请求的Scope依赖注入是通过一个ContainerMiddleware来实现的，调用该Middleware时，会创建一个限定作用域的DI容器，用于替换当前请求中已有的默认DI容器。在该管线中，所有后续的Middleware都会使用这个新的DI容器，在请求走完整个Pipeline管线以后，该ContainerMiddleware的作用就结束了，此时作用域会被销毁，并且在该作用域内创建的实例对象也都会销毁释放。

ContainerMiddleware的时序图如下所示：

具体的使用方式如下：

app.Use(new Func<RequestDelegate, RequestDelegate>(nextApp => new ContainerMiddleware(nextApp, app.ApplicationServices).Invoke));

# **普通类的依赖注入处理**

目前普通类的依赖注入，只支持构造函数，比如我们定于一个TestService类，代码如下：

public class TestService

{

private ITodoRepository \_repository;

public TestService(ITodoRepository r)

{

\_repository = r;

}

public void Show()

{

Console.WriteLine(\_repository.AllItems);

}

}

通过在构造函数里传入ITodoRepository类的参数来使用该实例，使用的时候需要先将该类注册到DI容器中，代码如下：

services.AddScoped<ITodoRepository, TodoRepository>();

services.AddSingleton<TestService>();

然后调用如下语句即可使用：

var service = serviceProvider.GetRequiredService<TestService>();

另外，需要注意，在目前的情况下，不能使用[FromServices]来使用依赖注入功能，比如，如下代码在获取TestService2实例的过程中会出现错误：

public class TestService2

{

[FromServices]

public ITodoRepository Repository { get; set; }

public void Show()

{

Console.WriteLine(Repository.AllItems);

}

}

## **普通类中获取HttpContext实例**

在MVC6中，我们没办法通过HttpContent.Current来获取上下文对象了，所以在普通类中使用的时候就会出问题，要想在普通类中使用该上下文对象，需要通过依赖注入来获取HttpContext实例，微软在ASP.NET5中，提供了IHttpContextAccessor接口用于获取该上下文对象。也就是说，我们可以将该类型的参数放在构造函数中，以获取上下文实例，代码如下：

public class TestService3

{

private IHttpContextAccessor \_httpContextAccessor;

public TestService3(IHttpContextAccessor httpContextAccessor)

{

\_httpContextAccessor = httpContextAccessor;

}

public void Show()

{

var httpContext = \_httpContextAccessor.HttpContext;//获取上下文对象实例

Console.WriteLine(httpContext.Request.Host.Value);

}

}

而使用的时候，则直接通过如下语句就可以了，代码如下：

var service = serviceProvider.GetRequiredService<TestService3>();

service.Show();

提示：普通类的构造函数中，可以传入多个DI容器支持的数据类似作为参数。

# **使用第三方DI容器**

目前，.NETCore不支持，只能在全功能版的.NET framework上才能使用，所以使用的时候需要注意一下。第三方DI容器的替换通常是在Startup.cs的Configure方法中进行的，在方法的开始处进行替换，以便后续的Middleware会使用相关的依赖注入功能。

首先要引入第三方的容器，以Autofac为例，引入Microsoft.Framework.DependencyInjection.Autofac，然后加入如下示例中的替换代码即可：

app.UseServices(services =>

{

services.AddMvc();// AddMvc要在这里注册

var builder = new ContainerBuilder();// 构造容器构建类

builder.Populate(services);//将现有的Services路由到Autofac的管理集合中

IContainer container = builder.Build();

return container.Resolve<IServiceProvider>();//返回AutoFac实现的IServiceProvider

});

注意，使用上述方法的时候，要把Mvc的注册代码services.AddMvc();必须要从ConfigureServices中挪到该表达式内，否则会报异常，等待微软解决。

另外，还有一个方式，微软目前的实例项目中还没有公开，通过分析一些代码，我们可以发现，在Microsoft.AspNet.Hosting程序中的StartupLoader.cs负责程序入口点的执行，在该文件中，我们知道首先是调用Startup.cs中的ConfigureServices方法，然后再调用Configure方法；我们可以看到示例中的ConfigureServices的返回值是void类型的，但在源码分析中发现，在根据约定解析ConfigureServices方法的时候，其首先判断有没有返回类型是IServiceProvider的，如果有则执行该方法，用使用该返回中返回的新IServiceProvider实例；没有的话，再继续查找void类型的ConfigureServices方法。所以，我们可以通过这种方式，来替换第三方的DI容器，实例代码如下：

// 需要先删除void类型的ConfigureServices方法public IServiceProvider ConfigureServices(IServiceCollection services){

var builder = new ContainerBuilder(); // 构造容器构建类

builder.Populate(services); //将现有的Services路由到Autofac的管理集合中

IContainer container = builder.Build();

return container.Resolve<IServiceProvider>(); //返回AutoFac实现的IServiceProvider

}

这样，你就可以像以往一样，使用Autofac的方式进行依赖类型的管理了，示例如下：

public class AutofacModule : Module

{

protected override void Load(ContainerBuilder builder)

{

builder.Register(c => new Logger())

.As<ILogger>()

.InstancePerLifetimeScope();

builder.Register(c => new ValuesService(c.Resolve<ILogger>()))

.As<IValuesService>()

.InstancePerLifetimeScope();

}

}

地址：<https://github.com/aspnet/Hosting/blob/dev/src/Microsoft.AspNet.Hosting/Startup/StartupLoader.cs>  
另外一个关于Autofac集成的案例：<http://alexmg.com/autofac-4-0-alpha-1-for-asp-net-5-0-beta-3/>

# **最佳实践**

在使用依赖注入的的时候，我们应该遵守如下最佳实践。

1. 做任何事情之前，务必在程序入口点提前注册所有的依赖类型。
2. 避免直接使用IServiceProvider接口，相反，在构造函数里显式添加需要依赖的类型即可，让依赖注入引擎自己来解析实例，一旦依赖很难管理的话，就使用抽象工厂。
3. 基于接口进行编程，而不是基于实现进行编程。

http://www.cnblogs.com/TomXu/p/4496440.html