第 3 章 依 赖 注 入

本章要点

- 服务容器与 ServiceProvider
- 服务的生存期
- 注入多个服务实例

3.1 依赖注入与服务容器

假设有 3 个类——A、B、C，A 类中需要引用 B 类的实例，B 类中需要引用 C 类的实例，比较常见的方案是通过类的构造函数传递被引用的对象。例如：

```
public class A
{
    private B _b;

    public A(B b)
    {
        _b = b;
    }
}

public class B
{
    C _c;

    public B(C c)
    {
        _c = c;
    }
}

public class C { }
```

如此一来，A、B、C 类之间就存在了依赖关系，即创建 B 类实例前要先创建 C 类实例，创建 A 类实例前要先创建 B 类实例。

```
C xc = new C();
B xb = new B(xc);
A xa = new A(xb);
```

随着需求的变化，A 类所依赖的类型可能不仅仅是 B 类，而是结构与 B 类相近的其他类型。于是，可以将 A 类对 B 类的依赖关系修改为 A 类对 ITest 接口类型的依赖，然后 B 类实现 ITest 接口。

```
public class A
{
    private ITest _t;

    public A(ITest ts)
    {
        _t = ts;
    }
}

// 公共接口
public interface ITest { }

public class B : ITest
{
     …
}
```

将来因功能需要，可能会扩展出 D 类。D 类同样实现 ITest 接口，因此在创建 A 类实例时，既可以向构造函数传递 B 类实例，也可以传递 D 类实例。

```
ITest _e1 = new B();
A var1 = new A(_e1);

ITest _e2 = new D();
A var2 = new A(_e2);
```

采用抽象类/实现类的设计模式也是可行的，如

```
public abstract class LoaderBase
{
    public abstract string GetData();
}

public class XMLLoader : LoaderBase
{
    public override string GetData()
    {
        return "XML 数据";
    }
```

```
}

public class JSONLoader : LoaderBase
{
    public override string GetData()
    {
        return "JSON 数据";
    }
}

public class CSVLoader : LoaderBase
{
    public override string GetData()
    {
        return "CSV 数据";
    }
}
```

LoaderBase 是一个抽象类，包含 GetData()抽象方法。假设该类的作用是通过 GetData()方法从数据文件中加载数据。常见的数据文件可以是 TXT、JSON、XML、INI、CSV 等格式。于是，从 LoaderBase 类可以派生出具有相应功能的类。上述代码中，XMLLoader 类用于加载 XML 格式的数据，JSONLoader 类用于加载 JSON 格式的数据，CSVLoader 类则用于加载 CSV 格式的数据。

再假设有一个 ConfigReader 类，专用于读取文件数据。

```
public sealed class ConfigReader
{
    private readonly LoaderBase _loader;

    public ConfigReader(LoaderBase loader)
    {
        _loader = loader;
    }
}
…
```

ConfigReader 类依赖的是 LoaderBase 类，所以它不需要考虑被读取的数据来自什么格式的文件，具体的文件加载工作由 LoaderBase 的实现类去完成。在创建 ConfigReader 类的实例时，可以向构造函数传递 JSONLoader 实例或 CSVLoader 实例。

```
ConfigReader reader = new(new JSONLoader());
ConfigReader reader = new(new CSVLoader());
```

通过构造函数参数或属性向调用者传递它所依赖的对象，称为依赖注入（Dependency Injection，DI）。给调用者赋值其所依赖对象的过程即是“注入”。此过程是通过一种特定的管理机制主动实现的，而不是调用者直接去获得。上述示例代码编写方式只适合少量对象间的依赖注入，如果一个应用程序中存在很多类型，并且类型之间的依赖关系非常复杂，手动编写代

码的方式不仅工作量大，而且混乱，也不利于维护和扩展。于是，就需要一种机制，可以自动管理类型之间的依赖关系，当调用方需要某个类型的实例时，该管理机制能够自动创建该类型的实例，自动创建该类型所依赖的类型实例并完成注入。

这种机制包括以下两部分。

（1）一个容器，即服务容器，可以将存在依赖关系的类型添加到该容器中，称为服务。

（2）自动激活实例。当调用者需要容器中某个类型实例时，服务容器会自动调用类型构造函数创建实例，并根据构造函数的参数列表自动创建和引用所依赖的类型实例。

3.1.1　ServiceCollection 类

ServiceCollection 类是.NET 内置的服务容器，它实现了 IServiceCollection 接口。依据接口的继承层次，可知 ServiceCollection 类也实现了 IList<T>、ICollection<T>、IEnumerable<T>等接口。也就是说，ServiceCollection 类的使用方法与其他的泛型集合相同。其中，类型参数 T 为 ServiceDescriptor 类。因此，被添加到服务容器中的服务类型将通过 ServiceDescriptor 对象来描述。

ServiceDescriptor 类公开了以下属性。

（1）ServiceType：表示服务类型的 Type 对象。

（2）ImplementationType：如果服务的实现方式为“接口+实现类”或“抽象类+派生类”，那么 ServiceType 属性表示服务的接口或抽象类，ImplementationType 属性表示服务的实现类。ImplementationType 属性是可选的，可以不指定。

（3）ImplementationInstance：表示服务类型的实例对象。

（4）Lifetime：表示服务对象的生存期类型。

为了使服务容器更容易访问，ServiceCollection 类提供了以下扩展方法（在 ServiceCollection-DescriptorExtensions 和 ServiceCollectionServiceExtensions 类中定义）。

（1）Add()、TryAdd()：将单个或多个服务添加到服务容器中。

（2）Replace()：替换容器中的 ServiceDescriptor 实例。

（3）RemoveAll()：删除容器所有与指定类型相同的服务。

（4）AddSingleton()、TryAddSingleton()：将服务添加到容器中，并且服务实例的生存期为单个实例。

（5）AddScoped()、TryAddScoped()：将服务添加到容器中，服务实例的生存期被限定为“作用域”（Scoped）范围内。

（6）AddTransient()、TryAddTransient()：将服务添加到容器中，服务实例的生存期为瞬时性。

上述扩展方法中，以 TryAdd 开头命名的方法表示只有当目标服务类型未在容器中注册时才会添加服务，否则不进行任何处理。

3.1.2 ServiceProvider 类

当服务类型注册到容器后，可使用 ServiceProvider 类获取服务类型的实例。服务类型的实例化过程由 ServiceProvider 类负责完成，调用代码只须告诉 ServiceProvider 类想要获取哪个类型的服务即可。

ServiceProvider 类实现了 IServiceProvider 接口。在编写代码过程中，应通过 IServiceProvider 接口类型的变量引用 ServiceProvider 实例，因为 ServiceProviderServiceExtensions 类中定义的扩展方法均面向 IServiceProvider 接口（GetService、CreateScope 等）。

ServiceProvider 类没有公共构造函数，不能直接使用 new 运算符实例化，而是调用 IServiceCollection 类型的扩展方法 BuildServiceProvider()获得 ServiceProvider 实例。

3.2 .NET 项目中的依赖注入

非 ASP.NET Core 项目在默认情况下不包含与依赖注入相关的引用，需要手动添加 Nuget 包引用。

在应用程序项目所在的目录下执行以下命令，可添加对 Microsoft.Extensions.DependencyInjection 包的引用。

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

或者直接编辑项目文件，在<Project>下添加以下内容。

```
<ItemGroup>
<PackageReference Include="Microsoft.Extensions.DependencyInjection"
Version="x.y.z" />
</ItemGroup>
```

其中，Version="x.y.z"表示要引用的 Nuget 包的版本号。

在代码文件中引入相关的命名空间。

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

然后创建容器实例，并调用 BuildServiceProvider()扩展方法产生 ServiceProvider 实例。

```
IServiceCollection services = new ServiceCollection();
// 添加服务
services.AddTransient<ITestService, MyService>();
IServiceProvider serviceProvider = services.BuildServiceProvider();
```

当需要访问服务实例时，可通过 serviceProvider 对象获取。

```
ITestService? theservice = serviceProvider.GetService<ITestService>();
```

GetService()方法将返回指定服务类型的实例，泛型参数用于指定要获取的服务类型，上述代码中，服务类型为 ITestService。如果服务容器中并不存在指定的服务类型，该方法将返回 null。因为在使用服务实例时需要进行 null 检查。

```
theservice?.DoSomething();
```

也可以调用非泛型的 GetService()方法，通过传递方法参数指定期望的服务类型。

```
ITestService? theservice = serviceProvider.GetService(typeof(ITestService))
as ITestService;
```

3.3　ASP.NET Core 项目中的依赖注入

在 ASP.NET Core 项目中，默认已包含对 Microsoft.Extensions.DependencyInjection 包的引用，开发人员不需要手动添加相关的 Nuget 包引用。

调用 WebApplication.CreateBuilder()方法会自动创建服务容器（ServiceCollection）实例，并向容器添加一些必要的基础服务。创建 WebApplicationBuilder 实例后，通过 Services 属性可以访问服务容器实例。

WebApplicationBuilder 类的内部封装了对 HostApplicationBuilder（位于 Microsoft.Extensions.Hosting 命名空间）实例的引用。ServiceCollection 容器的实例也是在 HostApplicationBuilder 类中初始化。以下是 HostApplicationBuilder 类的部分源代码。

```
public sealed class HostApplicationBuilder
{
    …
    private readonly ServiceCollection _serviceCollection = new();

    private Func<IServiceProvider> _createServiceProvider;
    private Action<object> _configureContainer = _ => { };
    private HostBuilderAdapter? _hostBuilderAdapter;

    private IServiceProvider? _appServices;
    …
}
```

通过 Services 属性将服务容器实例对外公开。

```
public IServiceCollection Services => _serviceCollection;
```

_createServiceProvider 字段是委托类型，用于创建 ServiceProvider 实例。当调用 Build()方法时，该委托会被执行。创建的 ServiceProvider 实例将赋值给_appServices 字段。

```
_appServices = _createServiceProvider();
```

同时，调用 MakeReadOnly()方法使服务容器（ServiceCollection）变为只读集合。之后就不能再向容器添加服务了，当然也不能删除或替换服务。

```
_serviceCollection.MakeReadOnly();
```

最后，把 HostApplicationBuilder 实例封装到 WebApplicationBuilder 类中。

```
public sealed class WebApplicationBuilder
{
    …
```

```
    private readonly HostApplicationBuilder _hostApplicationBuilder;
    …
}
```

通过 Services 属性继续对外公开服务容器。

```
public IServiceCollection Services => _hostApplicationBuilder.Services;
```

在 Build()方法中也调用了 HostApplicationBuilder 类的 Build()方法。

```
public WebApplication Build()
{
    …
    _builtApplication = new WebApplication(_hostApplicationBuilder.Build());
    return _builtApplication;
}
```

ServiceCollection 对象的传递过程如图 3-1 所示。

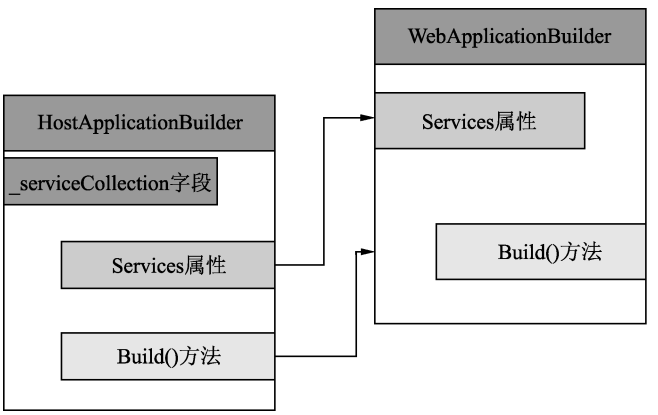

图 3-1　ServiceCollection 对象的传递过程

待 WebApplication 对象构建完成后，服务容器处于只读状态，不能被修改。此时可以访问 app.Services 属性获得对 ServiceProvider 实例的引用，再通过调用 GetService()方法获取要使用的服务实例。

```
// 获取服务实例
ILoggerFactory? logfac = app.Services.GetService<ILoggerFactory>();
// 使用服务实例
if(logfac != null)
{
    // 创建日志记录器
    ILogger logger = logfac.CreateLogger("示例日志");
    // 写入一条日志记录
    logger.LogInformation("此条记录仅用于测试");
}
```

上述代码从服务容器中获取 ILoggerFactory 类型的服务实例，再调用它的 CreateLogger()

方法创建一个用于写入应用程序日志的对象，最后调用 LogInformation()方法写入一条测试日志。

3.4　构建存在依赖关系的服务

本节将通过一个示例演示如何在应用程序中定义存在依赖关系的服务类型。

定义一个名为 DataSender 的服务类，调用它的 Sendout()方法后会将字符串数据发回给客户端。DataSender 类依赖名为 IDataWriter 的服务。IDataWriter 定义为接口，包含一个 Output()方法，可将字符串数据以特定的格式输出。

```
public interface IDataWriter
{
    ValueTask Output(string data);
}
```

本示例中还定义了两个实现了 IDataWriter 接口的类，它们的功能都是将字符串数据写入 HTTP 响应流中。其中，JsonDataWriter 类将数据以 JSON 格式写入响应流，XmlDataWriter 类则将数据以 XML 格式写入响应流。

```
public class JsonDataWriter : IDataWriter
{
    readonly HttpContext _httpContext;
    public JsonDataWriter(IHttpContextAccessor accessor)
    {
        _httpContext = accessor.HttpContext!;
        // 设置响应头
        _httpContext.Response.ContentType = "application/json;charset=
        UTF-8";
    }

    public async ValueTask Output(string data)
    {
        string res = $"{{ \"data\": \"{data}\" }}";
        // 写入响应内容
        await _httpContext.Response.WriteAsync(res);
    }
}

public class XmlDataWriter : IDataWriter
{
    readonly HttpContext _httpContext;
    public XmlDataWriter(IHttpContextAccessor accessor)
    {
        _httpContext = accessor.HttpContext!;
        // 设置响应头
        _httpContext.Response.ContentType = "application/xml;charset=
        UTF-8";
```

```
    }

    public async ValueTask Output(string data)
    {
        string odata = $"<data>{data}</data>";
        // 写入响应内容
        await _httpContext.Response.WriteAsync(odata);
    }
}
```

这两个类在结构上和实现逻辑上都很相似，区别在于 JsonDataWriter 类生成的数据为 JSON 格式，而 XmlDataWriter 类生成的数据为 XML 格式。

这两个类都依赖 IHttpContextAccessor 接口类型的服务，作用是通过该服务可以引用与当前 HTTP 通信关联的 HttpContext 对象，然后再通过 HttpContext 对象将数据写入 HTTP 的响应流中。最后，DataSender 类依赖一个 IDataWriter 类型的服务对象，其具体的实例类型可能是 XmlDataWriter 或 JsonDataWriter。

将相关的服务添加到服务容器中。

```
var builder = WebApplication.CreateBuilder(args);

// 注册服务
builder.Services.AddHttpContextAccessor();
builder.Services.AddSingleton<IDataWriter, JsonDataWriter>();
//builder.Services.AddSingleton<IDataWriter, XmlDataWriter>();
builder.Services.AddTransient<DataSender>();
…
```

因为本示例需要从服务容器中获取 IHttpContextAccessor 服务，所以要调用 AddHttpContext-Accessor()扩展方法注册默认的 HttpContextAccessor 服务。JsonDataWriter 和 XmlDataWriter 类型的服务可以同时加入容器中，在获取服务实例时，优先获取最后添加到容器中的类型。假设先添加 JsonDataWriter 服务，再添加 XmlDataWriter 服务，那么最终被注入 DataSender 类构造函数的是 XmlDataWriter 服务。

```
builder.Services.AddSingleton<IDataWriter, JsonDataWriter>();
builder.Services.AddSingleton<IDataWriter, XmlDataWriter>();
```

以下代码尝试从服务容器中获得 DataSender 实例，并调用它的 Sendout()方法。

```
app.MapGet("/", async () =>
{
    // 获取 DataSender 实例
    DataSender ds = app.Services.GetRequiredService<DataSender>();
    // 使用服务实例
    await ds.Sendout("示例数据");
});
```

得到的结果如下。

```
<data>示例数据</data>
```

可以对示例做一个扩展，定义一个选项类，用于配置数据输出的格式。

```
// 枚举
public enum Format { Json, Xml}

// 选项类
public class DataOutputOptions
{
    public Format OutputFormat { get; set; }
}
```

将选项类对象也添加到服务容器中。

```
builder.Services.Configure<DataOutputOptions>(options =>
{
    options.OutputFormat = Format.Json;
});
```

在注册 IDataWriter 服务时，可以根据 DataOutputOptions 选项类的配置决定产生 JsonDataWriter 实例还是 XmlDataWriter 实例。

```
builder.Services.AddSingleton<IDataWriter>(provider =>
{
    // 获得选项类的实例引用
    IOptions<DataOutputOptions> opt = provider.GetRequiredService
    <IOptions<DataOutputOptions>>();
    // 获取 IHttpContextAccessor
    IHttpContextAccessor accessor = provider.GetRequiredService
    <IHttpContextAccessor>();
    // 根据 OutputFormat 属性的值创建不同的类型实例
    IDataWriter datawriter = opt.Value.OutputFormat switch
    {
        Format.Json => new JsonDataWriter(accessor),
        Format.Xml => new XmlDataWriter(accessor),
    };
    return datawriter;
});
```

调用 JsonDataWriter 类或 XmlDataWriter 类的构造函数时需要注入 IHttpContextAccessor 类型的服务实例，由于此处是手动调用构造函数，所以要先从服务容器中获取 IHttpContext-Accessor 服务实例，再手动传递给构造函数。

此时运行示例程序，将得到 JSON 格式的数据。

```
{ "data": "示例数据" }
```

若需要 XML 格式的数据，可以将选项类配置为 XML 格式。

```
builder.Services.Configure<DataOutputOptions>(options =>
{
    options.OutputFormat = Format.Xml;
});
```

3.5 服务的生存期

服务注册到容器时，有 3 种生存期可供选择。

（1）单实例服务：生存期最长，实例生存期与服务容器相同。在整个容器的生存期内，服务仅创建一个实例。

（2）瞬时服务（暂时性服务）：生存期最短，每次从服务容器中获取时都会创建新的服务实例。

（3）作用域服务：从容器根部的 ServiceProvider 上创建子作用域。在该作用域的生存期内，服务只创建一个实例。当该作用域的生命周期结束时会释放服务实例。

下面将通过一个示例验证服务在不同生存期的实例化行为。代码定义一个服务类，在其构造函数中生成一个新的 GUID 值。该 GUID 可以通过 MyID 属性获得。

```
public class TestService : IDisposable
{
    readonly Guid _guid;
    public TestService()
    {
        _guid = Guid.NewGuid();
        Console.WriteLine("{0}实例化", GetType().Name);
    }

    // 获取与实例相关的 GUID
    public Guid MyID => _guid;

    public void Dispose()
    {
        Console.WriteLine("{0}实例即将释放", GetType().Name);
    }
}
```

此服务类实现了 IDisposable 接口，当实例被释放时会向屏幕输出提示信息。

首先，测试一下单实例服务的实例化行为。

```
// 创建服务容器
var services = new ServiceCollection();
// 注册服务
services.AddSingleton<TestService>();
// 构建 ServiceProvider
using(var provider = services.BuildServiceProvider())
{
    // 获取 3 次服务实例
    var x1 = provider.GetRequiredService<TestService>();
    var x2 = provider.GetRequiredService<TestService>();
    var x3 = provider.GetRequiredService<TestService>();
    // 输出 GUID
```

```
        Console.WriteLine("变量 1: {0}", x1.MyID);
        Console.WriteLine("变量 2: {0}", x2.MyID);
        Console.WriteLine("变量 3: {0}", x3.MyID);
    }
```

上述代码将从服务容器中获取 3 次 TestService 实例，接着依次输出它们的 GUID 值。运行代码后发现 3 次输出的 GUID 相同，说明从容器中获取的引用为同一个服务实例，如图 3-2 所示。

接下来测试一下瞬时性服务的实例化行为。

```
// 创建服务容器
var services = new ServiceCollection();
// 添加服务
services.AddTransient<TestService>();
// 构建 ServiceProvider
using var svcPrvd = services.BuildServiceProvider();
// 获取 4 次服务实例
var x1 = svcPrvd.GetRequiredService<TestService>();
var x2 = svcPrvd.GetRequiredService<TestService>();
var x3 = svcPrvd.GetRequiredService<TestService>();
var x4 = svcPrvd.GetRequiredService<TestService>();
// 依次输出它们的 GUID
Console.WriteLine($"变量 1: {x1.MyID}");
Console.WriteLine($"变量 2: {x2.MyID}");
Console.WriteLine($"变量 3: {x3.MyID}");
Console.WriteLine($"变量 4: {x4.MyID}");
```

代码运行后，可以看到 4 个变量的 GUID 均不相同，这表明容器为服务创建了 4 个实例，如图 3-3 所示。

```
====== 单实例服务 ======
TestService实例化
变量1: 72a5e23d-55cd-4d01-b8fb-5e8b9cad097a
变量2: 72a5e23d-55cd-4d01-b8fb-5e8b9cad097a
变量3: 72a5e23d-55cd-4d01-b8fb-5e8b9cad097a
TestService实例即将释放
```

图 3-2　3 个变量的 GUID 相同

```
====== 瞬时服务 ======
TestService实例化
TestService实例化
TestService实例化
TestService实例化
变量1: dfcdb48c-f06e-4fe1-8f87-8e439fdfc6d2
变量2: 9eade92e-3236-417a-b897-32c6ec7f806e
变量3: 5f216710-2a6a-4fe2-ac2a-b9a5daed058b
变量4: 38f7b1f9-c8a8-431b-ace2-0071e7524e5c
TestService实例即将释放
TestService实例即将释放
TestService实例即将释放
TestService实例即将释放
```

图 3-3　4 个变量的 GUID 均不相同

最后测试作用域服务的实例化行为。

```
// 创建服务容器
var container = new ServiceCollection();
```

```
// 注册服务
container.AddScoped<TestService>();
// 构建位于根部的 ServiceProvider
using var rootProvider = container.BuildServiceProvider();

Console.WriteLine("----- 开始 子作用域-1 -----");
// 创建第 1 个子作用域
using (var scoped1 = rootProvider.CreateScope())
{
    // 获取 3 次服务实例
    var v1 = scoped1.ServiceProvider.GetRequiredService<TestService>();
    var v2 = scoped1.ServiceProvider.GetRequiredService<TestService>();
    var v3 = scoped1.ServiceProvider.GetRequiredService<TestService>();
    // 输出 GUID
    Console.WriteLine($"变量 1: {v1.MyID}");
    Console.WriteLine($"变量 2: {v2.MyID}");
    Console.WriteLine($"变量 3: {v3.MyID}");
}
Console.WriteLine("----- 结束 子作用域-1 -----");

Console.WriteLine("----- 开始 子作用域-2 -----");
// 创建第 2 个子作用域
using (var scoped2 = rootProvider.CreateScope())
{
    // 获取 4 次服务实例
    var k1 = scoped2.ServiceProvider.GetRequiredService<TestService>();
    var k2 = scoped2.ServiceProvider.GetRequiredService<TestService>();
    var k3 = scoped2.ServiceProvider.GetRequiredService<TestService>();
    var k4 = scoped2.ServiceProvider.GetRequiredService<TestService>();
    // 输出 GUID
    Console.WriteLine($"变量 1: {k1.MyID}");
    Console.WriteLine($"变量 2: {k2.MyID}");
    Console.WriteLine($"变量 3: {k3.MyID}");
    Console.WriteLine($"变量 4: {k4.MyID}");
}
Console.WriteLine("----- 结束 子作用域-2 -----");
```

上述代码创建了两个子作用域。在第 1 个子作用域中，获取 3 次服务实例并向屏幕输出 GUID；在第 2 个子作用域中，获取了 4 次服务实例并输出 GUID，如图 3-4 所示。

从结果中可以看到，在同一个作用域内，各变量输出的 GUID 相同，表明它引用的是同一个服务实例。

对于作用域服务，需要注意在默认情况下是不能通过位于根部的 ServiceProvider 对象获取服务实例的，只能先调用 CreateScope()（或 CreateAsyncScope()）方法创建子作用域，然后通过子作用域的 ServiceProvider 对象获取服务实例。

图 3-4　同一个作用域内 GUID 相同

若确实需要从根 ServiceProvider 处获取作用域服务的实例，可以在调用 BuildService-Provider()方法时将 validateScopes 参数设置为 false，如

```
// 创建容器
var container = new ServiceCollection();
// 注册服务
container.AddScoped<IDriver, MyDriver>();
// 创建位于容器根部的 ServiceProvider
using var serviceProvider = container.BuildServiceProvider
(validateScopes: false);
// 获取服务
IDriver drv = serviceProvider.GetRequiredService<IDriver>();
```

3.6　GetService()方法与 GetRequiredService()方法的区别

GetService()方法与 GetRequiredService()方法都可以通过 ServiceProvider 类从容器中获取服务实例，其区别如下。

（1）GetService()方法：如果服务已在容器中注册，就返回其实例，否则返回 null，不会引发异常。程序代码需要进行 null 检查。

（2）GetRequiredService()方法：如果服务已在容器中注册，就返回服务类型的实例，否则会引发 InvalidOperationException 异常。

也就是说，GetRequiredService()方法要求服务必须已经注册到容器中。

若使用 GetService()方法获取服务实例，随后需要在代码中验证服务实例是否为 null，如

```
ITest? obj = serviceProvider.GetService<ITest>();
if(obj != null)
{
```

```
    // 使用服务实例
}
```

若使用的是 GetRequiredService()方法，建议将代码写在 try…catch 语句块中，以捕捉可能发生的异常，如

```
try
{
    ITest obj2 = serviceProvider.GetRequiredService<ITest>();
    _ = obj2.GetNum();                    //使用服务
}
catch
{
    // 处理异常
}
```

3.7 注入多个服务实例

假设有一个 IDemo 接口，它有两个实现类 A 和 B。

```
public interface IDemo { }

public class A : IDemo { }

public class B : IDemo { }
```

将 A、B 类添加到服务容器中。

```
container.AddTransient<IDemo, A>();
container.AddTransient<IDemo, B>();
```

当请求一个服务实例时，默认返回最后添加到容器中的服务类的实例。

```
IDemo sv = serviceProvider.GetRequiredService<IDemo>();
```

在该示例中，GetRequiredService()方法返回的是 B 类的实例。

在有些特殊应用场合，开发人员希望同时获取 A、B 类的实例，此时应该调用 GetServices()方法，它将返回一个服务实例列表，其中包含 A、B 类的实例。

```
IEnumerable<IDemo> insts = serviceProvider.GetServices<IDemo>();
foreach(var s in insts)
{
    …
}
```

接下来给出一个比较完整的示例。该示例将实现向 MVC 控制器的构造函数注入多个服务实例。

先定义名为 ICalculator 的接口类型，它表示一个计算器程序的通用模型。该接口包含一个 GetResult()的方法，用于获取计算结果（输入参数和返回结果均为 double 类型）。

```
public interface ICalculator
{
    double GetResult(double x);
}
```

再定义 3 个类，它们都实现 ICalculator 接口。

```
public class CalculatorA : ICalculator
{
    public double GetResult(double x)
    {
        return Math.Pow(x, 2d);
    }
}

public class CalculatorB : ICalculator
{
    public double GetResult(double x)
    {
        return Math.Pow(x, 3d);
    }
}

public class CalculatorC : ICalculator
{
    public double GetResult(double x)
    {
        return Math.Pow(x, 4d);
    }
}
```

CalculatorA 类求 x 的平方（二次方），CalculatorB 类求 x 的立方（三次方），CalculatorC 类求 x 的四次方。

将以上 3 个类添加到服务容器中，其生存期均为瞬时（暂时）服务。

```
builder.Services.AddTransient<ICalculator, CalculatorA>();
builder.Services.AddTransient<ICalculator, CalculatorB>();
builder.Services.AddTransient<ICalculator, CalculatorC>();
```

本示例将使用到 MVC 功能，因此也需要将相关的功能服务添加到容器中。

```
builder.Services.AddControllersWithViews();
```

调用 Build()方法创建 Web 应用程序实例后，还需要向 HTTP 管线添加 MVC 相关的终结点。

```
var app = builder.Build();

// 使用 MVC 终结点
app.MapControllerRoute("mvc", "{controller=Home}/{action=Default}");

app.Run();
```

MapControllerRoute()方法在添加 MVC 终结点时需要指定一条路由规则。本示例中将路由规则命名为 mvc，规则中大括号内的字符串表示路由参数的名称。controller 表示控制器的名字，此处分配一个默认值 Home；action 表示控制器内要调用的操作方法的名称，默认为 Default。当客户端请求的 URL 为根地址（/）时，路由规则中的参数会使用默认值，使请求的 URL 变为/Home/Default，即调用 Home 控制器中的 Default 方法。例如，访问 https://somehost/就会解析为 https://somehost/Home/Default。

创建一个类，派生自 Controller 类（位于 Microsoft.AspNetCore.Mvc 命名空间），该类就会被识别为 MVC 控制器。本示例中控制器名为 Home。

```
public class HomeController : Controller
{
    readonly IEnumerable<ICalculator> _calculators;
    public HomeController(IEnumerable<ICalculator> cals)
    {
        _calculators = cals;
    }

    public IActionResult Default()
    {
        double input = 0.5d;
        // 存入 ViewData 中，以便在视图代码中读取
        ViewData.Add("number", input);
        IList<double> results = new List<double>();
        // 分别计算
        foreach(ICalculator cal in _calculators)
        {
            double r = cal.GetResult(input);
            // 存储计算结果
            results.Add(r);
        }
        return View(results);
    }
}
```

HomeController 类定义了类型为 IEnumerable<ICalculator>的私有字段，该字段用于从类构造函数的参数中接收注入的服务实例。在运行阶段，它将包含 3 个服务实例，分别为 CalculatorA、CalculatorB、CalculatorC 类型。

在 Default 方法中，依次调用这些服务实例的 GetResult()方法，然后把计算结果存放到一个 double 列表中。最后返回与此方法关联的视图（View），同时将 double 列表对象作为模型对象（Model）传递给视图。

新建一个目录，命名为 Views，再在 Views 目录下新建一个子目录 Home。在 Home 目录下添加一个 Razor 视图文件，文件名与 Default()方法相同，即 Default.cshtml。文件内容如下。

```
@model IList<double>
<html>
```

```
<head>
<title>示例</title>
<meta charset="utf-8" />
</head>
<body>
    @{
        // 取出 ViewData 中存放的内容
        if(ViewData.TryGetValue("number", out var v))
        {
            // 显示内容
            <p>输入数字：@v</p>
        }
    }
    @if (Model.Count == 0)              //没有计算结果
    {
<div>无计算结果</div>
    }
    else
    {
        foreach(double val in Model)
        {
            <div>@val</div>
        }
    }
</body>
</html>
```

第 1 行的@model 指令声明此视图的模型类型为 double 列表，而模型的值将从 Model 属性获取（Default()方法末尾调用 View()方法时传递）。

视图中使用 foreach 语句访问 double 列表中的所有计算结果，并呈现在 元素中。运行应用程序，结果如图 3-5 所示。

图 3-5　呈现 3 个计算结果

3.8 容易被忽略的问题

由于容器中的服务存在生存期的差异，在设计服务类时需要考虑它们之间的依赖关系是否会导致生存周期的改变。下面的例子将反映该问题。

有两个服务类：Service1 和 Service2。Service2 类需要引用 Service1 类，即在服务容器中，Service1 类的实例会被注入 Service2 类的构造函数中。

```
public class Service1 : IDisposable
{
    readonly Guid _id;
    public Service1()
    {
        _id = Guid.NewGuid();
    }

    // 获取实例标识
    public Guid ID => _id;

    public void Dispose()
    {
        Console.WriteLine("{0}实例即将被清理", GetType().Name);
    }
}

public class Service2 : IDisposable
{
    readonly Guid _id;
    readonly Service1 _service1;
    public Service2(Service1 sv)
    {
        _service1 = sv;
        _id = Guid.NewGuid();
    }

    // 获取实例标识
    public Guid ID => _id;
    // 获取 Service1 实例的引用
    public Service1 OtherService => _service1;

    public void Dispose()
    {
        Console.WriteLine("{0}实例即将被清理", GetType().Name);
    }
}
```

Service1 类在容器中注册为瞬时服务，而 Service2 类则注册为单个实例服务。

```
// 创建服务容器
var services = new ServiceCollection();
// 注册服务
services.AddTransient<Service1>();
services.AddSingleton<Service2>();
// 创建 ServiceProvider 实例
using var svProvd = services.BuildServiceProvider();
```

下面的代码将从容器中连续获取 3 次 Service2 服务实例。

```
for (int i = 1; i < 4; i++)
{
    var svobj = svProvd.GetRequiredService<Service2>();
    Console.WriteLine("第{0}轮：", i);
    Console.WriteLine($"Service1.ID:  {svobj.OtherService.ID}");
    Console.WriteLine($"Service2.ID:  {svobj.ID}");
    Console.WriteLine("\n");
}
```

运行后的屏幕输出结果如图 3-6 所示。

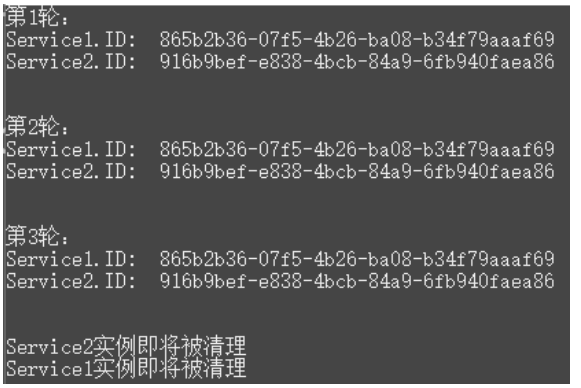

图 3-6　瞬时服务意外变成了单实例服务

由运行结果可以发现，由于 Service2 是单个实例服务，它内部对 Service1 的引用会导致 Service1 的生存期被延长——从瞬时服务变成了单实例服务（3 次获取到的 Service1 实例的 GUID 相同，属于同一个实例）。

因此，在处理服务类之间的依赖关系时，应避免将生存期短的服务注入生存期较长的服务中。