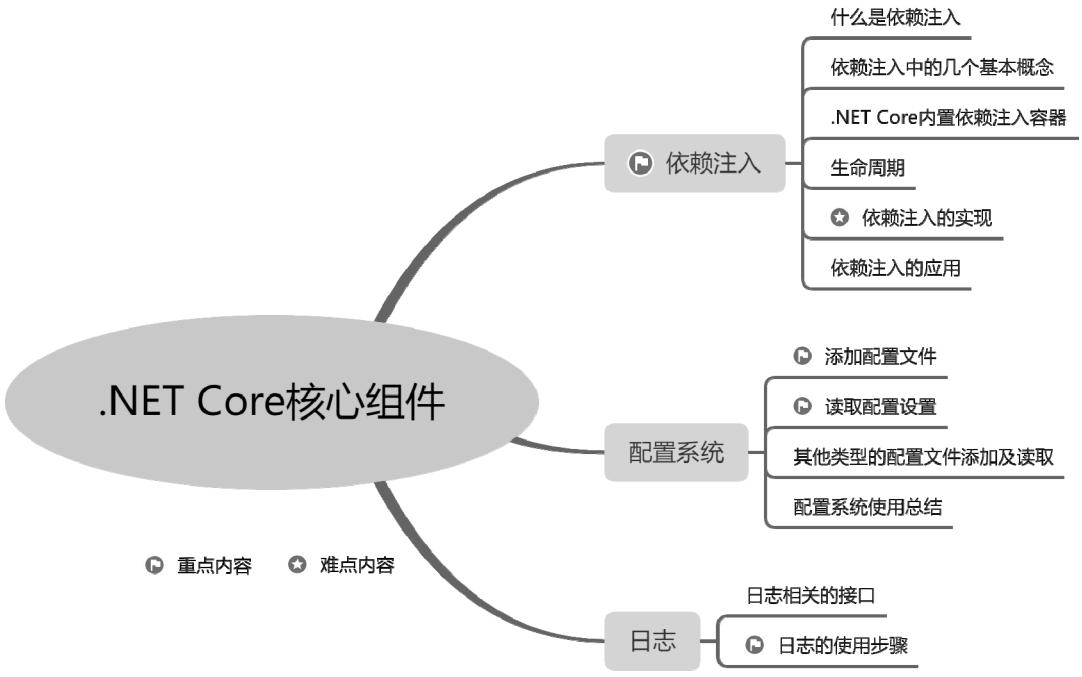
第 7 章

.NET Core 核心组件

依赖注入、配置系统以及日志是.NET Core 开发的三大核心组件，通过在.NET Core 开发中使用这三大组件，可以降低程序的耦合性，增强程序的健壮性，并方便后期运维人员的配置维护。本章将分别对这三大核心组件的原理及使用进行详细讲解。

本章知识架构及重点、难点如下。



7.1 依 赖 注 入

7.1.1 什么是依赖注入

依赖注入（dependency injection，简称 DI），是一种在类及其依赖项之间实现控制反转的技术，它允许开发人员将类的依赖项（如其他类、接口等）注入类中，而不是在类中直接实例化依赖项，这样做的好处是可以使代码更加松散耦合，易于测试和维护。

说明

控制反转（inversion of control），简称 IoC，是面向对象编程中的一种设计原则，可以用来减少程序代码间的耦合程度，它的目的是把创建和组装对象的操作从业务逻辑层转移到框架中，这样在业务逻辑层，程序只需要说明要使用某个类型的对象，框架就会自动创建这个对象。

例如，下面以汽车类和引擎类进行说明，任何汽车都需要一个引擎，按照传统的方式，我们会编写如下代码：

```
public class Car
{
    public Car()
    {
        QYEngine engine = new QYEngine();
        engine.Start();
    }
}

public class QYEngine
{
    public void Start()
    {
        Console.WriteLine("引擎启动……");
    }
}
```

通过分析上面代码，我们发现 Car 类依赖于 QYEngine 类，如果随着时代的进步，引擎进行了升级，比如，引擎升级为纯电引擎 CDEngine，这时如果 Car 类需要进行引擎升级，就需要重写，代码如下：

```
public class Car
{
    public Car()
    {
        CDEngine engine = new CDEngine();
        engine.Start();
    }
}

public class CDEngine
{
    public void Start()
    {
        Console.WriteLine("纯电引擎启动……");
    }
}
```

以此类推，如果引擎一直升级，那么我们的 Car 类就需要一直重写，这导致汽车类与引擎类之间的耦合太紧，而依赖注入可以消除这种类之间的依赖关系。比如，使用传统定义接口的方式进行注入，代码如下：

```
interface IEngine
{
    public void Start();
}
```

```
public class QYEngine: IEngine
{
    public void Start()
    {
        Console.WriteLine("引擎启动……");
    }
}

public class CDEngine: IEngine
{
    public void Start()
    {
        Console.WriteLine("纯电引擎启动……");
    }
}

public class Car
{
    private IEngine _engine;
    public Car(IEngine engine)
    {
        _engine = engine;
}
    public void DoSomething()
    {
        _engine.Start();
    }
}
```

上面代码中，在 Car 类的构造函数中注入了 IEngine 接口对象，这样 Car 类就不再关心具体的引擎实现类。在.NET Core 中使用依赖注入有以下几个好处：

- ☑ 松散的耦合：依赖注入是实现松散耦合的一种方法。开发者可以通过注入接口或抽象类，使代码更加灵活和可扩展，以实现各种需要。
- ☑ 测试：依赖注入是一个可测试代码的强有力工具。开发者可以轻松地使用替代伪装对象，而不需要将目标代码耦合起来。
- ☑ 单一职责和依赖反转原则：从代码开发这一角度来看，这些原则要求开发者将责任划分为小而有用的单元，然后将它们组装在一起来构建软件。
- ☑ 更容易的代码维护：使用依赖注入可以轻松地将代码分离成不同的单元，这使得源码更容易管理和维护。另外，开发者可以轻松地设计独立的模块，而不必担心它们之间的交互或依赖。

7.1.2 依赖注入中的几个基本概念

.NET Core 的依赖注入是一种设计模式，它提供了一种将对象的创建和使用进行分离的方法，它为开发者提供了以一种容易维护和可测试的方式创建组件化应用程序的能力。要学习.NET Core 依赖注入，首先应该了解以下几个概念：

- ☑ 依赖注入容器（DI container）：依赖注入容器是一个对象，它能够自动化地在应用程序中创建并且管理其他对象。
- ☑ 服务（service）：一个服务就是一个类，它提供程序中某个组件的功能。
- ☑ 依赖（dependency）：依赖是指一个服务依赖于另一个服务。

☑　组件（component）：组件是服务和它所依赖的所有服务的集合。

7.1.3　.NET Core 内置依赖注入容器

在.NET Core 中实现依赖注入，可以使用内置的依赖注入容器，也可以使用第三方的依赖注入容器，要使用内置的依赖注入容器，需要安装以下两个 NuGet 包：

```
Microsoft.Extensions.DependencyInjection.Abstractions
Microsoft.Extensions.DependencyInjection
```

.NET Core 内置依赖注入容器的主要命名空间为 Microsoft.Extensions.DependencyInjection，其中核心的类型是 ServiceDescriptor、IServiceCollection、IServiceProvider，它们的关系如图 7.1 所示。

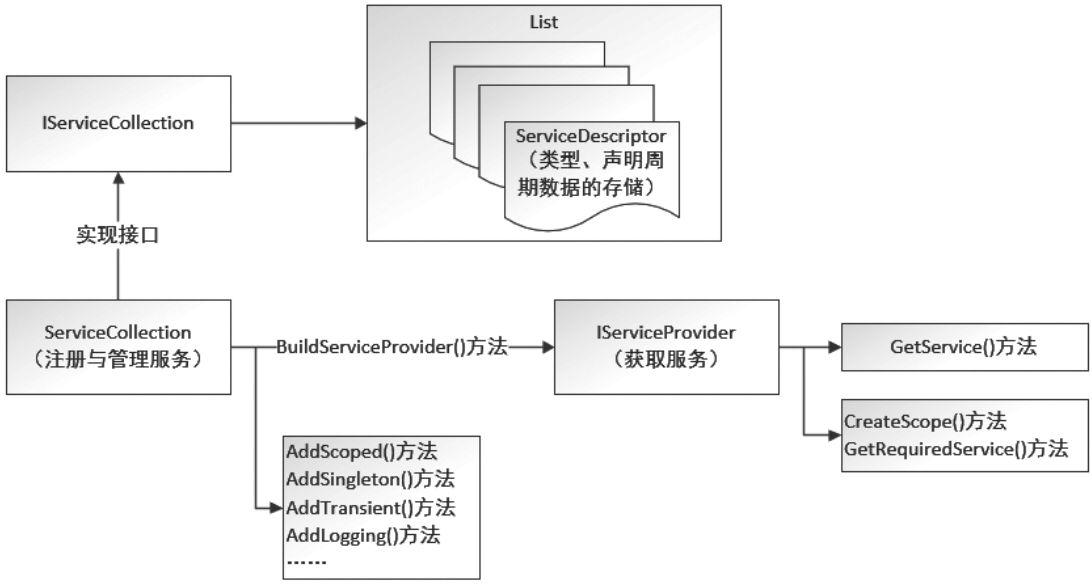


图 7.1　.NET Core 内置依赖注入容器中的类关系

ServiceDescriptor、IServiceCollection、IServiceProvider 的作用分别如下：

☑　ServiceDescriptor：每个服务的描述。

☑　IServiceCollection：服务集合。

☑　IServiceProvider：向外提供服务的类型。

下面分别介绍 ServiceDescriptor、IServiceCollection、IServiceProvider 中常用的属性以及方法，以便为依赖注入实现打下理论基础。

1．ServiceDescriptor

ServiceDescriptor 位于 Microsoft.Extensions.DependencyInjection 命名空间下，用来描述一种服务，包括该服务的类型、实现和生存期。ServiceDescriptor 类的构造函数如下：

```
public ServiceDescriptor (Type serviceType, Func<IServiceProvider,object> factory,
    Microsoft.Extensions.DependencyInjection.ServiceLifetime lifetime);
public ServiceDescriptor (Type serviceType, object instance);
```

```
public ServiceDescriptor (Type serviceType, Type implementationType,
    Microsoft.Extensions.DependencyInjection.ServiceLifetime lifetime);
```

ServiceDescriptor 类有 5 个属性，其说明如表 7.1 所示。

表 7.1　ServiceDescriptor 类的属性及说明

属　　性	说　　明
ServiceType	表示要注册的类型，也就是将来要获取的实例的类型，既可以是接口、抽象类，也可以是普通的类型
Lifetime	表示服务的生存期，它有 3 个枚举值：Singleton（单例）、Scoped（范围）和 Transient（瞬态）
ImplementationType	表示实际上要创建的类型
ImplementationInstance	表示在注册时已经获得了一个指定类的实例并将它注册进来，将来要获取该类的实例时，将 ImplementationInstance 返回给调用者即可
ImplementationFactory	表示注册了一个用于创建 ServiceType 指定的类型的工厂，当需要从容器获取该类实例时，由这个工厂负责创建该类的实例

ServiceDescriptor 类的常用方法及说明如表 7.2 所示。

表 7.2　ServiceDescriptor 类的方法及说明

方　　法	说　　明
Describe()	创建具有指定属性的 ServiceDescriptor 实例
Scoped()	创建具有指定属性的 ServiceDescriptor 实例（生存期为 Scoped）
Singleton()	创建具有指定属性的 ServiceDescriptor 实例（生存期为 Singleton）
Transient()	创建具有指定属性的 ServiceDescriptor 实例（生存期为 Transient）

例如，下面代码分别用来创建 3 种不同生存期的服务注册实例：

```
public static ServiceDescriptor Scoped<TService, TImplementation>();        //创建范围服务注册实例
public static ServiceDescriptor Singleton<TService, TImplementation>();     //创建单例服务注册实例
public static ServiceDescriptor Transient<TService, TImplementation>();     //创建瞬态服务注册实例
```

2．IServiceCollection

IServiceCollection 位于 Microsoft.Extensions.DependencyInjection 命名空间下，用来为服务集合指定协定，它是一个 IList<ServiceDescriptor>类型的集合，ServiceDescriptor 是该集合中的子项。IServiceCollection 常用扩展方法及说明如表 7.3 所示。

表 7.3　IServiceCollection 接口扩展方法及说明

扩 展 方 法	说　　明
AddWebEncoders()	将 HtmlEncoder、JavaScriptEncoder 和 UrlEncoder 添加到指定的服务集合中
Add()	将 ServiceDescriptor 添加到服务集合中
RemoveAll()	删除 IServiceCollection 中所有指定类型的服务
Replace()	删除 IServiceCollection 中与 ServiceDescriptor 的服务类型相同的第一个服务，并将 ServiceDescriptor 添加到集合中

续表

扩 展 方 法	说　　明
TryAdd()	如果服务类型尚未注册，则将指定的 ServiceDescriptor 添加到服务集合中
TryAddEnumerable()	如果现有描述符具有相同 ServiceType 和服务集合中尚不存在的实现，则添加指定的 ServiceDescriptor
TryAddScoped()	如果服务类型尚未注册，则将指定的服务作为范围服务添加到服务集合中
TryAddSingleton()	如果服务类型尚未注册，则将指定的服务作为单例服务添加到服务集合中
TryAddTransient()	如果服务类型尚未注册，则将指定的服务作为瞬态服务添加到服务集合中
AddHttpClient()	将 IHttpClientFactory 和相关服务添加到 IServiceCollection
AddLocalization()	添加应用程序本地化所需的服务
AddLogging()	将日志记录服务添加到指定的 IServiceCollection
Configure<TOptions>()	注册将对其绑定 TOptions 的配置实例
AddOptions()	添加使用选项所需的服务
BuildServiceProvider()	创建一个 ServiceProvider，它包含提供的 IServiceCollection 中的服务
AddScoped()	将指定类型的范围服务添加到指定的 IServiceCollection 中
AddSingleton()	将指定类型的单例服务添加到指定的 IServiceCollection 中
AddTransient()	将指定类型的瞬态服务添加到指定的 IServiceCollection 中

例如，下面代码首先创建一个 ServiceDescriptor 对象，然后将其添加到 IServiceCollection 服务集合中：

```
var serviceDescriptor = ServiceDescriptor.Singleton<IMyService, MyService>();
IServiceCollection services = new ServiceCollection();
services.Add(serviceDescriptor);
```

上面代码由于创建的 ServiceDescriptor 为单例服务，因此添加到 IServiceCollection 集合中的也是单例服务，如果使用 IServiceCollection 集合的扩展方法，则可以替换为如下代码：

```
IServiceCollection services = new ServiceCollection();
services.AddSingleton<IMyService,MyService>();
```

3．IServiceProvider

IServiceProvider 位于 System 命名空间下，是向其他对象提供自定义支持的对象，它是一个接口，默认提供了一个 GetService()方法，用来获取指定类型的服务对象；而在依赖注入组件中，由类 ServiceProvider 实现接口 IServiceProvider，它位于 Microsoft.Extensions.DependencyInjection 包中，它提供了一组扩展方法，让开发者可以更方便地编写获取对象的代码，尤其是泛型方法，它可以直接获得特定类型的返回值，而无须进行类型转换。ServiceProvider 类扩展方法及说明如表 7.4 所示。

表 7.4　ServiceProvider 类扩展方法及说明

扩 展 方 法	说　　明
GetService(Type)	获取指定类型的服务对象
CreateAsyncScope(IServiceProvider)	新建可用于解析范围内服务的 AsyncServiceScope
CreateScope(IServiceProvider)	新建可用于解析范围内服务的 IServiceScope
GetRequiredService(IServiceProvider, Type)	从 IServiceProvider 获取类型 Type 的服务
GetRequiredService<T>(IServiceProvider)	从 IServiceProvider 获取类型 T 的服务，如果服务未注册，将抛出异常

续表

扩 展 方 法	说　明
GetService<T>(IServiceProvider)	从 IServiceProvider 获取类型 T 的服务，如果服务未注册，其不会抛出异常，而是返回 null 或者默认值
GetServices(IServiceProvider, Type)	从 IServiceProvider 获取 Type 类型服务的枚举
GetServices<T>(IServiceProvider)	从 IServiceProvider 获取 T 类型服务的枚举

ServiceProvider 对象由 IServiceCollection 的扩展方法 BuildServiceProvider()创建，当需要它提供某个服务时，会根据创建它的 IServiceCollection 中对应的 ServiceDescriptor 提供相应的服务实例。例如，下面代码创建一个 ServiceProvider 对象：

```
IServiceCollection services = new ServiceCollection();
IServiceProvider serviceProvider = services.BuildServiceProvider();
```

下面代码使用创建的 ServiceProvider 对象来获取指定服务的实例：

```
//获取指定类型的服务实例
object myservice = serviceProvider.GetService(typeof(IMyService));

//获取泛型方法指定类型的服务实例
IMyService myservice = serviceProvider.GetService<IMyService>();
```

7.1.4　生命周期

7.1.3 节提到创建 ServiceDescriptor 对象时，可以指定 3 种生存期，分别为 Singleton（单例）、Scoped（范围）和 Transient（瞬态），这也是.NET Core 中的依赖注入所支持的 3 个服务的生命周期，下面分别对它们进行介绍。

- ☑ 瞬态（Transient）：瞬态服务是每次从服务容器请求时创建的，即每次注入都会自动用 new 新建一个对象，这种生命周期适合轻量级、无状态的服务，它可以避免多段代码使用同一对象而造成混乱，但缺点是生成的对象比较多，可能浪费内存。
- ☑ 范围（Scoped）：对 Web 应用来说，范围服务的生命周期就是每次请求，请求开始后的第一次注入，就是它生命的开始，直到请求结束，因此，对于这种类型的依赖注入，在同一次 HTTP 请求中，不同的注入会获得同一个对象，在不同的 HTTP 请求中，不同的注入会获得不同的对象。这种方式适用于在同一个范围内共享同一个对象的情况。
- ☑ 单例（Singleton）：来自依赖关系注入容器的服务实现的每一个后续请求都使用同一个实例。如果应用需要单一实例行为，则允许服务容器管理服务的生命周期。这种生命周期会全局共享同一个服务对象，这样可以节省创建新对象的资源。单例服务必须是线程安全的，并且通常在无状态服务中使用。

说明

在处理请求的应用中，当应用关闭并释放 ServiceProvider 时，会释放单例服务。由于应用关闭之前不释放内存，因此需要考虑单例服务的内存使用。

例如，下面代码用来注册不同生命周期的服务：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<IMyService, MyService>();
    services.AddScoped<IMyService, MyService>();
    services.AddSingleton<IMyService, MyService>();
}
```

注意

不能在长生命周期的对象中引用生命周期比它短的对象，比如，不能在单例服务中引用范围服务，否则可能会导致被引用的对象被释放，或者内存泄漏。

7.1.5　依赖注入的实现

上面讲解了.NET Core 中内置依赖注入容器的基础知识，本节将介绍如何在.NET Core 中实现依赖注入。实现依赖注入可以通过在 Startup 类中配置服务集合来实现，其具体步骤如下。

1．注册服务

在.NET Core 中，要使用依赖注入需要先注册服务，注册服务主要通过 IServiceCollection 对象的 AddXXX()扩展方法实现，例如：

```
ServiceCollection services = new ServiceCollection();
services.AddTransient<IService, Service>();
```

说明

ASP.NET Core 中的注册服务通常在 Startup 类的 ConfigureServices()方法中实现。

上面代码将 IService 接口和 Service 类进行了绑定，表示当 IService 接口被请求时，可以自动返回 Service 类的一个新实例。这里使用了 AddTransient()方法来注册服务，表示瞬态服务，即每次请求都会返回一个新的实例。

另外，也可以使用默认的服务提供程序注册服务，例如：

```
ServiceCollection services = new ServiceCollection();
services.AddLogging();
```

上面代码中并没有手动指定服务的实现，而是使用了默认的日志记录服务。

2．解析服务

一旦在依赖注入容器中注册了服务，就可以解析这些服务。解析服务意味着使用依赖注入容器创建对象实例并将其注入类中来自动解决依赖关系。解析服务通常是在类的构造函数中通过从容器中获取参数类型来实现的，例如：

```
public class MyClass
{
    private readonly IService _service;
```

```
    public MyClass(IServiceProvider serviceProvider)
    {
        _service = serviceProvider.GetService<IService>();
    }
    public void DoSomething()
    {
        _service.DoSomeWork();
    }
}
```

上面代码中，通过从服务提供程序获取所需的服务实例将_service 变量实例化。通过将 IServiceProvider 接口注入 MyClass 的构造函数，调用 GetService<T>()方法来获取所需的服务，而在 DoSomething()方法中，可以使用_service 变量来调用 IService 的方法。

另外，在.NET Core 中，还可以在控制器中使用构造函数注入依赖。例如：

```
public class MyController : Controller
{
    private readonly IMyService _myService;
    public MyController(IMyService myService)
    {
        _myService = myService;
    }
}
```

上面代码将 IMyService 接口声明为 MyController 控制器类的一个构造函数参数，在创建 MyController 实例时，依赖注入容器将自动解析 IMyService 的实例并传递给构造函数。

例如，下面代码直接使用_myService 实例的方法，而不需要在 MyController 类中实例化 IMyService 的具体实现。

```
public IActionResult Index()
{
    var data = _myService.DoSomeWork ();
    return View(data);
}
```

说明

注入服务还可以在视图中实现，这需要使用 @inject 指令。视图注入适合用在直接在视图中使用逻辑的场景，比如本地化或者只用于视图的服务。视图注入方式如下：

```
@inject WebApplication.Services.MyService
```

从上面步骤可以看出，.NET Core 的依赖注入使代码更加模块化、可测试和可维护，通过使用服务集合和声明依赖关系，开发人员可以更好地管理代码的结构和依赖项。

7.1.6 依赖注入的应用

【例 7.1】通过依赖注入实现商品信息的查询及添加（**实例位置：资源包\Code\07\01**）

新建一个.NET 控制台应用，首先定义需要注册的服务，这里主要定义了商品实体类、数据操作业

务逻辑接口和数据访问接口。代码如下：

```
record Goods(string ID, string Name, double Price,int Num);

interface IGoodsDAO
{
    public Goods? GetByID(string ID);                                    //查询指定编号的商品信息
    public void AddGoods(string ID, string Name, double Price, int Num);  //添加商品信息
}

interface IGoodsBLL
{
    public bool CheckID(string ID);                                      //检查指定 ID 是否存在
}
```

首先实现数据操作业务逻辑接口，编写其实现类，在该类中通过构造函数注入了一个 IDbConnection 接口对象，代码如下：

```
using System.Data;

class GoodsDAO : IGoodsDAO
{
    private readonly IDbConnection conn;

    public GoodsDAO(IDbConnection conn)
    {
        this.conn = conn;
    }

    public Goods? GetByID(string ID)
    {
        using var dt = SqlHelper.ExecuteQuery(conn,
            $"select * from tb_Goods where ID={ID}");
        if (dt.Rows.Count <= 0)
        {
            return null;
        }
        DataRow row = dt.Rows[0];
        string id = (string)row["ID"];
        string name = (string)row["Name"];
        double price = (double)row["Price"];
        int num = (int)row["Num"];
        return new Goods(id, name, price,num);
    }

    public void AddGoods(string ID,string Name,double Price,int Num)
    {
        SqlHelper.ExecuteQuery(conn,$"insert into tb_Goods values({ID},{Name},{Price},{Num})");
    }
}
```

然后实现数据访问接口，编写其实现类，该类中通过构造函数注入了一个 IGoodsDAO 接口对象，代码如下：

```
class GoodsBLL : IGoodsBLL
{
    private readonly IGoodsDAO GoodsDao;

    public GoodsBLL(IGoodsDAO GoodsDao)
```

```
    {
        this.GoodsDao = GoodsDao;
    }

    public bool CheckID(string ID)
    {
        var Goods = GoodsDao.GetByID(ID);
        if (Goods == null)
        {
            return false;
        }
        else
        {
            return Goods.ID == ID;
        }
    }
}
```

定义完服务之后，接下来就可以使用了，在 Program.cs 主类中，首先定义服务集合，注册需要使用的服务 IDbConnection、IGoodsDAO 和 IGoodsBLL，这里分别将 GoodsDAO 和 GoodsBLL 注册为 IGoodsDAO 和 IGoodsBLL 的实现类，然后使用 ServiceProvider 对象的 GetService()扩展方法对 IGoodsDAO 和 IGoodsBLL 进行解析，并调用相应的方法实现查询和添加商品数据的功能。代码如下：

```
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Data;
using System.Data.SqlClient;

ServiceCollection services = new ServiceCollection();

services.AddScoped<IDbConnection>(sp => {
    string connStr = "Data Source=.;Initial Catalog=MR_TEST;Integrated Security=true";
    var conn = new SqlConnection(connStr);
    conn.Open();
    return conn;
});
services.AddScoped<IGoodsDAO, GoodsDAO>();
services.AddScoped<IGoodsBLL, GoodsBLL>();
using (ServiceProvider sp = services.BuildServiceProvider())
{
    var goodsDAO = sp.GetService<IGoodsDAO>();
    var goodsBLL = sp.GetService<IGoodsBLL>();
    Console.WriteLine("————————————查询商品信息————————————");
    var goods = goodsDAO.GetByID("GD20230001");
    Console.WriteLine(goods);
    Console.WriteLine("————————————添加商品信息————————————");
    if (!goodsBLL.CheckID("GD20230003"))
    {
        goodsDAO.AddGoods("GD20230003","C#从入门到精通（第 7 版）",89.8,300);
        Console.WriteLine("添加成功！添加的数据如下：\n");
        goods = goodsDAO.GetByID("GD20230003");
        Console.WriteLine(goods);
    }
    else
        Console.WriteLine("要添加的数据已经存在！");
}
```

查看上面代码，我们发现，除了使用 new SqlConnection()之外，其他对象都没有通过 new 关键字

创建，而是通过依赖注入容器获取的，因此，在实际开发中，如果使用了依赖注入，应该尽量避免直接使用 new 关键字创建对象。

运行程序，效果如图 7.2 所示。

图 7.2　通过依赖注入实现商品信息的查询及添加

如果再次运行程序，则会提示“要添加的数据已经存在!”，效果如图 7.3 所示。

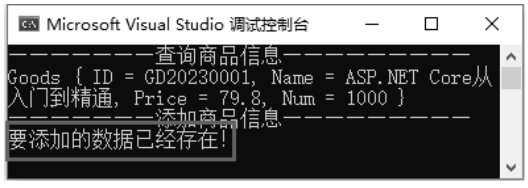


图 7.3　重复运行程序时的提示

7.2　配　置　系　统

.NET Core 的配置系统是一个强大的工具，它允许开发人员在应用程序中轻松地使用和管理配置信息。要使用.NET Core 的配置系统，需要安装以下 NuGet 包：

```
Microsoft.Extensions.Configuration
```

使用.NET Core 配置系统的基本步骤如下。

（1）添加配置文件。

（2）读取配置设置。

下面对.NET Core 中配置系统的使用进行详细讲解。

7.2.1　添加配置文件

.NET Core 中的配置系统提供了多个源来获取配置信息，包括 JSON 文件、环境变量、命令行参数、INI 文件、XML 文件等。在应用程序启动时，配置系统会将这些源中的配置信息合并到一个统一的配置对象中，并使其可用于整个应用程序。

例如，下面代码是一个.NET Core 程序中的配置文件，该文件使用 JSON 文件格式：

```
{
  "ConnectionStrings": {
    "DefaultConnection": "server=localhost;user id=root;password=123456;database=myDatabase"
  },
```

```
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting": "Information"
    }
  }
}
```

要使.NET Core 程序默认加载该配置文件，需要将文件复制到.exe 文件所在文件夹中（即程序输出文件夹），一种方法是手动复制，另外一种是自动复制，比如，上面的 JSON 文件命名为 appsettings.json，在 Visual Studio 开发工具的“解决方案资源管理器”中选中该文件，单击鼠标右键，在弹出的快捷菜单中选择“属性”命令，然后在弹出的“属性”对话框中将“复制到输出目录”设置为“如果较新则复制”，如图 7.4 所示。

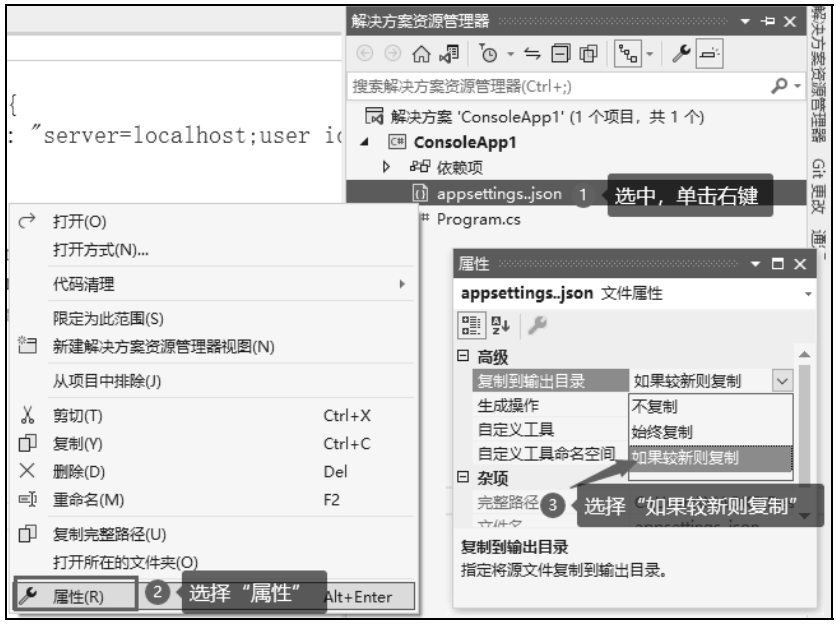


图 7.4　设置将 JSON 文件复制到输出目录

通过以上步骤，我们就在.NET Core 程序中添加了一个配置文件。

7.2.2　读取配置设置

添加配置文件后，接下来就需要读取配置文件中的设置了。在.NET Core 中，读取配置文件中的设置有两种方式，分别是 Configuration API 或 Options API，下面分别讲解。

1．Configuration API 方式读取

Configuration API 使开发人员能够使用键/值对形式或者 API 方法方式读取配置设置，在使用 Configuration API 时，需要使用 ConfigurationBuilder 类和 IConfigurationRoot 对象，下面分别对它们进行介绍。

☑　ConfigurationBuilder 类

ConfigurationBuilder 类位于 Microsoft.Extensions.Configuration 命名空间下，用于生成基于键/值的配置设置，以便在应用程序中使用。ConfigurationBuilder 类的构造函数如下：

```
public ConfigurationBuilder ();
```

ConfigurationBuilder 类有两个属性，其说明如表 7.5 所示。

表 7.5　ConfigurationBuilder 类的属性及说明

属　　性	说　　明
Properties	获取可用于在 IConfigurationBuilder 和已注册的配置提供程序之间共享数据的键/值集合
Sources	获取用于获取配置值的源

ConfigurationBuilder 类的方法及说明如表 7.6 所示。

表 7.6　ConfigurationBuilder 类的方法及说明

方　　法	说　　明
Add(IConfigurationSource)	添加一个新的配置源
Build()	使用在 Sources 中注册的提供程序集中的键和值生成 IConfiguration
AddConfiguration()	将现有配置添加到 IConfigurationProvider
AddCommandLine()	添加从命令行读取配置值的 IConfigurationProvider 或 CommandLineConfigurationProvider
AddEnvironmentVariables()	添加从环境变量读取配置值的 IConfigurationProvider
AddIniFile()	将 INI 配置源添加到 IConfigurationProvider，主要通过读取文件内容的方式来实现
AddIniStream()	将 INI 配置源添加到 IConfigurationProvider，主要通过读取内存流的方式来实现
AddJsonFile()	将 JSON 配置源添加到 IConfigurationProvider，主要通过读取文件内容的方式来实现
AddJsonStream()	将 JSON 配置源添加到 IConfigurationProvider，主要通过读取内存流的方式来实现
AddKeyPerFile()	生成基于键/值的配置设置，以便在应用程序中使用
AddInMemoryCollection()	将内存配置提供程序添加到 IConfigurationProvider
AddUserSecrets()	添加用户机密配置源
AddXmlFile()	将 XML 配置源添加到 IConfigurationProvider，主要通过读取文件内容的方式来实现
AddXmlStream()	将 XML 配置源添加到 IConfigurationProvider，主要通过读取内存流的方式来实现

☑　IConfigurationRoot 接口

IConfigurationRoot 接口位于 Microsoft.Extensions.Configuration 命名空间下，表示 IConfiguration 层次结构的根，该接口提供了两个属性，其说明如表 7.7 所示。

表 7.7　IConfigurationRoot 接口的属性及说明

属　　性	说　　明
Item[String]	获取或设置配置值
Providers	获取用于获取配置值的源

IConfigurationRoot 接口的方法及说明如表 7.8 所示。

表 7.8　IConfigurationRoot 接口的方法及说明

方　法	说　明
GetChildren()	获取直接后代配置子节
GetSection(String)	获取具有指定键的配置子节，如果配置子节不存在，则返回一个空值
Reload()	强制从基础 IConfigurationProvider 重新加载配置值
Bind()	尝试通过按递归方式根据配置键匹配属性名称，将给定的对象实例绑定到配置值
Get()	尝试将配置实例绑定到类型为 T 的新实例。如果此配置节仅包含一个值，则将使用该值。否则，通过按递归方式根据配置键匹配属性名称来进行绑定
GetValue()	提取具有指定键的值，并将其转换为指定的类型
AsEnumerable()	获取 IConfiguration 键/值对的枚举
GetConnectionString()	从配置源的节检索 ConnectionStrings 具有指定键的值，该方法与 GetSection("ConnectionStrings")[name]等效
GetRequiredSection()	获取具有指定键的配置子节，当配置子节不存在时，抛出一个异常
GetDebugView()	生成可读的配置视图，其中显示每个值的来源

例如，下面代码中首先创建了一个 ConfigurationBuilder 对象，并使用其 AddJsonFile()扩展方法添加了一个待解析的 JSON 配置文件，然后使用 IConfigurationRoot 对象读取配置项，读取配置项时有两种方法，一种是使用[]形式（root["ConnectionStrings:DefaultConnection"]），另一种是使用 GetSection()方法。代码如下：

```
using Microsoft.Extensions.Configuration;

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.AddJsonFile("appsettings.json", false, true);
IConfigurationRoot root = builder.Build();
var defaultConnection = root["ConnectionStrings:DefaultConnection"];
Console.WriteLine(defaultConnection);
var logLevel = root.GetSection("Logging").GetSection("LogLevel").GetSection("Default").Value;
Console.WriteLine(logLevel);
```

说明

上面代码中由于要读取 JSON 文件，因此需使用 NuGet 命令安装 Microsoft.Extensions.Configuration.Json 开发包。

上面代码运行效果如图 7.5 所示。

图 7.5　读取 JSON 配置文件内容

2．Options API 方式读取

Options API 允许将配置设置直接绑定到对象和类中，这使得开发人员能够以一种类型安全和优雅的方式访问和管理配置值，这也是.NET Core 中推荐的方式，使用这种方式，需要使用 NuGet 命令安

装以下两个开发包：

```
Microsoft.Extensions.Options
Microsoft.Extensions.Configuration.Binder
```

例如，还是读取 7.2.1 节中的 appsettings.json 配置文件，我们可以将“ConnectionStrings”部分中的配置设置绑定到一个名为 ConnectionOptions 的类中，代码如下：

```
public class ConnectionOptions
{
    public string DefaultConnection { get; set; }
}
```

使用 Options API 方式读取配置时，需要与依赖注入一起使用，因此需要创建一个类，用来获取注入的选项值，这里声明接收选项注入的对象类型不能直接使用上面定义的 ConnectionOptions 类，而应该使用 IOptions<T>、IOptionsMonitor<T>或者 IOptionsSnapshot<T>等泛型接口，这 3 个泛型接口的区别如下：

- ☑ IOptions<T>：配置改变后，不能读取新值，而必须重新运行程序才能读取新值，因此占用资源少，适用于服务器启动后，配置不会再发生改变的情况。
- ☑ IOptionsMonitor<T>：配置改变后，可以读取到新值，但可能会导致同一个请求中，前后读取的配置值不同。
- ☑ IOptionsSnapshot<T>：配置改变后，可以读取到新值，它可以保证在同一个请求中前后读取的配置值相同，而只有在不同的请求中，配置值才有可能不同，因此，通常使用 IOptionsSnapshot<T>作为接收选项注入的类型。

【例 7.2】使用 Options API 方式读取配置文件内容（**实例位置：资源包\Code\07\02**）

例如，这里使用 IOptionsSnapshot<T>接收选项注入类型，并定义读取配置的测试类，代码如下：

```
using Microsoft.Extensions.Options;
class Demo
{
    private readonly IOptionsSnapshot<ConnectionOptions> connectionOptions;
    public Demo(IOptionsSnapshot<ConnectionOptions> connectionOptions)
    {
        this.connectionOptions = connectionOptions;
    }
    public void Test()
    {
        var con = connectionOptions.Value;
        Console.WriteLine(con.DefaultConnection);
    }
}
```

接下来，编写注入服务到容器的代码，首先引入依赖注入和配置系统相关命名空间，然后加载 appsettings.json 配置文件，使用 AddOptions()方法注册与选项相关的服务，并把 ConnectionStrings 节点的内容绑定到 ConnectionOptions 类型的模型对象上，接下来将 Demo 测试类注册为瞬态服务，最后通过服务获取配置文件中相应的值。代码如下：

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.AddJsonFile("appsettings.json", false, true);
```

```
IConfigurationRoot root = builder.Build();
ServiceCollection services=new ServiceCollection();
services.AddOptions().Configure<ConnectionOptions>(e=>root.GetSection("ConnectionStrings").Bind(e));
services.AddTransient<Demo>();
using (var scope = services.BuildServiceProvider())
{
    while (true)
    {
        var demo = scope.GetService<Demo>();
        demo.Test();
        Console.Read();
    }
}
```

运行上面代码，效果如图 7.6 所示。

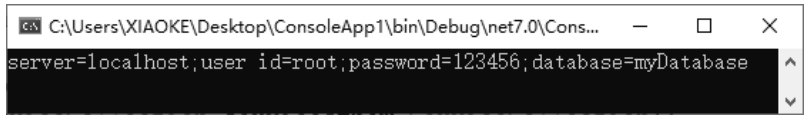


图 7.6　使用 Options API 方式读取 JSON 配置文件内容

我们修改程序生成目录下的 appsettions.json 文件后，再次运行程序，即可输出修改后的值，如图 7.7 所示。

图 7.7　读取修改后的 JSON 配置文件内容

7.2.3　其他类型的配置文件添加及读取

前面两节以 JSON 文件为例讲解了.NET Core 程序中配置文件的添加与读取，除了 JSON 文件，.NET Core 程序的配置系统还支持其他配置源，如环境变量、命令行参数、INI 文件、XML 文件等。其中，INI 文件、XML 文件作为配置源时，其添加及读取方式与 JSON 文件类似，这里不再详细介绍；下面主要讲解如何将环境变量和命令行参数作为.NET Core 程序的配置源。

1．使用环境变量作为配置源

使用.NET Core 的应用程序可以根据不同的环境（如测试环境、开发环境、生产环境等）有不同的配置，这时就可以使用环境变量作为配置源。从环境变量中读取配置需要使用 NuGet 命令安装以下开发包：

```
Microsoft.Extensions.Configuration.EnvironmentVariables
```

然后使用 ConfigurationBuilder 类的 AddEnvironmentVariables()扩展方法进行读取，该方法是一个重载方法，重载形式如下：

```
AddEnvironmentVariables()
AddEnvironmentVariables(string prefix)
```

第一种重载形式没有参数，表示可以将系统中的所有环境变量都加载进来；第二种重载形式有一个 string 类型的参数，用来指定环境变量的前缀，这种形式可以加载系统中具有指定前缀的环境变量。

例如，下面代码加载系统中所有的环境变量，并读取名称为“Path”的环境变量的值，代码如下：

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.AddEnvironmentVariables();
IConfigurationRoot root = builder.Build();
Console.WriteLine(root["Path"]);
```

但如果系统环境变量中有很多“PATH_”开头的环境变量，我们想要读取名称为“PATH_NET”的环境变量的值，则可以使用下面代码：

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.AddEnvironmentVariables("PATH_");
IConfigurationRoot root = builder.Build();
Console.WriteLine(root["NET"]);
```

使用上面代码时，必须在系统环境变量中存在“PATH_NET”。

2．使用命令行参数作为配置源

在容器化运行环境中，.NET Core 程序非常适合通过命令行参数来读取配置信息。从命令行参数中读取配置需要使用 NuGet 命令安装以下开发包：

```
Microsoft.Extensions.Configuration.CommandLine
```

然后使用 ConfigurationBuilder 类的 AddCommandLine()扩展方法进行读取，该方法常用语法如下：

```
AddCommandLine(String[] args)
```

参数 args 是一个字符串类型的参数，表示命令行参数。在命令行中传递的值应该是一组以两个短横线（“--”）或者正斜杠（“/”）为前缀的键，然后是值，键和值用等号（“=”）或空格（“ ”）分隔，另外，如果使用等号设定值，可以排除前缀，因此，命令行参数可以有以下 5 种形式：

```
--key1=value1
--key2 value2
/key3=value3
/key4 value4
key5=value5
```

例如，下面代码读取命令行参数中传递的 server 的值，代码如下：

```
using Microsoft.Extensions.Configuration;
public class Test
{
    static void Main(string[] args)
    {
        ConfigurationBuilder builder = new ConfigurationBuilder();
        builder.AddCommandLine(args);
        IConfigurationRoot root = builder.Build();
        Console.WriteLine($"server: {root["server"]}");
    }
}
```

编译上面代码，然后打开系统的“命令提示符”窗口，将 EXE 路径复制到“命令提示符”窗口中，同时传入 server 参数，按回车键，效果如图 7.8 所示。

图 7.8　读取命令行参数

上面是使用“命令提示符”窗口执行程序时传递命令行参数，在 Visual Studio 中还能以可视化方式传递命令行参数，具体步骤为：在 Visual Studio 的菜单栏中选择“项目”→“***属性”，在打开的属性对话框中单击“调试”/“常规”选项卡，然后单击“打开调试启动配置文件 UI”超链接，打开“启动配置文件”对话框，在“命令行参数”文本框中即可输入要设置的命令行参数，如图 7.9 所示。

图 7.9　以可视化方式设置命令行参数

命令行参数设置完成后，直接在 Visual Studio 中运行上面代码，效果如图 7.10 所示。

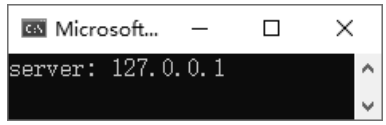


图 7.10　读取可视化方式设置的命令行参数

7.2.4　配置系统使用总结

.NET Core 的配置系统是一个灵活而强大的框架组成部分，它主要用来简化程序的配置管理，支持 JSON 文件、环境变量、命令行参数、INI 文件、XML 文件等多种数据源，并与不同类型的使用模式和应用程序类型兼容，使用它可以减少硬编码并提高可维护性。

另外，.NET Core 程序支持一个程序中同时使用多种数据源，在读取时，如果后添加的数据源有与先添加的数据源中重复的选项，后添加的配置会覆盖先添加的配置，比如，使用 JSON 文件添加的配置信息中有 server，值为 127.0.0.1，但后面又通过命令行参数添加了 server，值为 www.mingrisoft.com，则在程序中，最后读取到的 server 值为 www.mingrisoft.com。

7.3　日　　志

.NET Core 中的日志是指应用程序记录和保存运行时信息的机制，应用程序的运行时信息可能包括调试信息、异常信息、业务操作信息等，通过日志记录可以帮助开发人员分析应用程序的运行情况，快速定位问题，提高开发效率。要使用.NET Core 的日志，需要安装以下 NuGet 包：

```
Microsoft.Extensions.Logging
```

另外，如果要使用控制台方式输出日志，还需要安装 Microsoft.Extensions.Logging.Console 开发包。

本节将对.NET Core 中日志的使用进行详细讲解。

7.3.1　日志相关的接口

在.NET Core 程序中使用日志时，需要用到 ILogger 接口和 ILoggerFactory 接口，下面分别对它们进行介绍。

1．ILogger 接口

ILogger 接口位于 Microsoft.Extensions.Logging 命名空间下，表示用于执行日志记录的类型。ILogger 接口提供了 3 个公共方法，如表 7.9 所示。

表 7.9　ILogger 接口的公共方法及说明

公 共 方 法	说　　明
BeginScope<TState>(TState)	开始逻辑操作范围
IsEnabled(LogLevel)	检查是否已启用给定 logLevel
Log<TState>(LogLevel, EventId, TState, Exception, Func<TState,Exception,String>)	写入日志项

除了上面的公共方法，ILogger 接口还提供了一些扩展方法，用来操作日志，其说明如表 7.10 所示。

表 7.10　ILogger 接口的扩展方法及说明

扩 展 方 法	说　　明
BeginScope()	设置消息格式并创建范围
Log()	在指定的日志级别设置日志消息格式并写入该消息
LogCritical()	设置关键日志消息格式并写入该消息
LogDebug()	设置调试日志消息格式并写入该消息
LogError()	设置错误日志消息格式并写入该消息
LogInformation()	设置信息日志消息格式并写入该消息
LogTrace()	设置跟踪日志消息格式并写入该消息
LogWarning()	设置警告日志消息格式并写入该消息

2．ILoggerFactory 接口

ILoggerFactory 接口位于 Microsoft.Extensions.Logging 命名空间下，表示一个类型，该类型用于配置日志记录系统并从已注册的 ILoggerProvider 创建 ILogger 的实例。ILoggerFactory 接口常用方法及说明如表 7.11 所示。

表 7.11　ILoggerFactory 接口的方法及说明

方　　法	说　　明
AddProvider(ILoggerProvider)	将指定的提供程序添加到创建 ILogger 实例时使用的提供程序集合
CreateLogger(String)	创建具有指定 categoryName 的 ILogger
CreateLogger(ILoggerFactory, Type)	使用给定 Type 的全名创建一个新的 ILogger 实例
CreateLogger<T>(ILoggerFactory)	使用给定类型的全名创建一个新的 ILogger 实例

7.3.2　日志的使用步骤

.NET Core 提供了丰富的日志记录 API，它支持各类日志系统，包括控制台、文件、系统事件日志、日志服务器等。下面介绍.NET Core 中日志的使用方法。

1．引入相关命名空间

在.NET Core 中使用日志，需要引入以下命名空间：

```
using Microsoft.Extensions.Logging;
```

2．配置日志记录器

在.NET Core 程序中使用日志时，首先需要先配置日志记录器的选项，包括选择日志记录的输出方式、日志的格式等选项，这主要通过在 ServiceCollection 集合的 AddLogging()方法中，使用 ILoggingBuilder 的 AddXXX()方法实现。

例如，下面代码通过 ILoggingBuilder 对象的 AddConsole()、AddDebug()、AddFile()方法添加了 3 个日志记录器，这 3 个日志记录器分别将日志记录到控制台、调试器输出窗口以及文件中。代码如下：

```
ServiceCollection services = new ServiceCollection();
services.AddLogging(logBuilder =>
{
    logBuilder.AddConsole();
    logBuilder.AddDebug();
    logBuilder.AddFile("log.txt");
});
```

3．创建日志记录器

使用 ILoggerFactory 接口的 CreateLogger()方法可以创建对应名称的日志记录器，例如，下面代码创建了一个名为 Program 的日志记录器：

```
ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
```

4．记录日志

在.NET Core 程序的需要记录日志的代码中，可以通过创建好的日志记录器对象来记录不同等级的日志，常用的日志等级有 Debug（调试）、Information（信息）、Warning（警告）、Error（错误）、Trace（跟踪）、Critical（关键）等。

例如，下面代码用来记录程序中的提示、警告和错误信息：

```
logger.LogInformation("程序提示信息");
logger.LogWarning("警告信息");
logger.LogError("错误信息");
```

以上就是.NET Core 中使用日志的基本流程，开发人员可以根据实际情况灵活选择不同的日志输出方式。

7.4　要 点 回 顾

本章主要对.NET Core 的三大核心组件：依赖注入、配置系统以及日志的原理及使用进行了详细讲解。其中，依赖注入是.NET Core 的核心，.NET Core 程序的各个部分都是通过依赖注入方式被组装在一起的；配置系统允许运维人员通过后期配置对程序的很多选项进行设置，避免程序中出现过多的硬性编码；日志则可以帮助程序开发人员更好地发现程序问题。本章所讲内容是.NET Core 应用开发的基础及核心，因此，必须熟练掌握。