第 5 章
ASP. NET Core 和 ABP 框架的基础设施

ASP. NET Core 和 ABP 框架都为现代应用程序开发提供了许多构件和功能。本章将介绍最基本的构件，从而使读者能够理解应用程序配置和初始化的过程。

本章从 ASP. NET Core 中的 Startup 类开始，阐述为什么需要模块化系统，以及 ABP 框架如何通过提供一个模块化的方式来配置和初始化应用程序；然后，讨论 ASP. NET Core 提供的依赖注入系统和 ABP 框架提供的根据预定义规则实现的自动化依赖注入注册方式；接着介绍 ASP. NET Core 的配置和选项的模式以学习配置 ASP. NET Core 应用程序和其他库的方式。

5.1 准备工作

如果想要运行示例程序，需要安装用于构建 ASP. NET Core 项目的 IDE/编辑器（如 Visual Studio）。读者可以从 GitHub 仓库 https://github.com/PacktPublishing/Mastering-ABP-Framework 下载本章的示例代码。

5.2 模块化

模块化是一种将大型软件的功能分解成更小的部分，并允许每个部分根据需要通过标准化接口与其他部分通信的设计技术。模块化有以下 3 个优点。

（1）由于模块之间是互相隔离的，仅能通过预先定义且有限的接口进行通信，因此模块化能够降低软件的复杂性。

（2）模块之间是松散耦合的，为程序开发提供了灵活性。之后开发者可以重构或者替换某个模块。

（3）模块能够独立于应用程序，从而提高了它的重用性。

大部分企业软件系统都是以模块化的方式进行设计的。然而，实现模块化设计并不容易，并且 ASP. NET Core 在模块化设计上没有提供相关的支持。ABP 框架的主要目标之一是为开发真正的模块化系统提供基础设施和工具。第 15 章将详细探讨模块化应用程序开发，本节仅介绍一些 ABP 框架的模块的基础知识。

5.2.1　Startup 类

在定义模块类之前，首先回顾 ASP.NET Core 中的 Startup 类，从而了解模块类需要完成的工作。示例代码如下所示：

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
        services.AddTransient<MyService>();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseRouting();
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

ConfigureServices 方法用于配置其他服务或者向依赖注入系统注册新服务。Configure 方法用于配置 ASP.NET Core 的请求处理管道（ASP.NET Core request pipeline），它由一系列处理 HTTP 请求的中间件构成。

通常在配置位于 Program.cs 文件中的 HostBuilder 时注册 Startup 类，以便应用程序启动时可以执行这段程序，代码如下：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[]args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

这段代码已经包含在 ASP.NET Core 的启动模板中，无须手动编写。

Startup 类的问题在于它的唯一性。也就是说，所有应用程序服务只能在一个位置进行配置和初始化。然而，在模块化应用程序中，每个模块都应该进行配置和初始化与该模块相

关的服务。此外，一个模块通常需要使用或依赖其他模块，因此这些模块需要按照正确的顺序进行配置和初始化。ABP 框架的模块定义类解决了这些问题。

5.2.2　定义模块类

ABP 框架的模块由一组一起开发和发布的类型（如类和接口）组成。它是一个程序集（Visual Studio 中的一个项目），包含一个派生自 AbpModule 的模块类。模块类负责配置和初始化该模块，并在必要时配置它所依赖的模块。

以下代码是一个简单的短信服务（Short Messaging Service，SMS）的模块定义类。

```
using Microsoft.Extensions.DependencyInjection;
using Volo.Abp.Modularity;
namespace SmsSending
{
    public class SmsSendingModule : AbpModule
    {
        public override void ConfigureServices(ServiceConfigurationContext context)
        {
            context.Services.AddTransient<SmsService>();
        }
    }
}
```

每个模块都可以重写 ConfigureServices 方法，来把它的服务注册到依赖注入系统中并配置其他模块。以上代码以瞬时服务的方式把 SmsService 注册到依赖注入系统中，并在模块定义类中实现了服务注册，功能与 5.2.1 节中的 Startup 类一样。然而，大多数情况下，开发者不需要手动注册这些服务，因为 ABP 框架的 DI 系统能够自动化地完成这些工作，参阅 5.3.2 节。

AbpModule 类定义了 OnApplicationInitialization 方法，该方法在服务注册完成并且应用程序做好运行准备后执行。该方法可以在应用程序启动时执行一些必要的操作，如初始化一个服务，代码如下：

```
public class SmsSendingModule : AbpModule
{
    //...
    public override void OnApplicationInitialization(
        ApplicationInitializationContext context)
    {
        var service = context.ServiceProvider.GetRequiredService<SmsService>();
        service.Initialize();
    }
}
```

这段代码使用 context.ServiceProvider 方法从依赖注入系统中获取一个服务并初始化该服务。由于此时 DI 系统已经准备好，因此可以直接从中获取服务。也可以把 OnApplicationInitialization 方法看作 Startup 类的 Configure 方法，因此开发者可以在该方法中配置 ASP.NET Core 的请求处理管道。当然，通常情况下应该在启动模块中配置请求处理管道。

5.2.3　模块依赖和启动模块

一个业务应用程序通常包含多个模块，ABP 框架允许开发者声明模块之间的依赖关系。一个应用程序总是有一个启动模块(startup module)。启动模块可以依赖某些模块，这些模块也可以依赖其他一些模块，以此类推。

图 5.1 展示了一个简单的模块依赖关系。

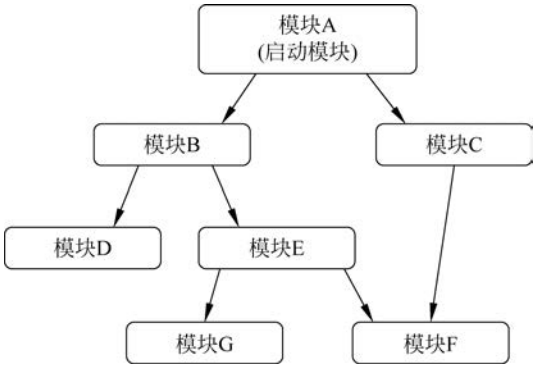

图 5.1　模块依赖关系

ABP 框架根据模块依赖关系初始化模块。如果模块 A 依赖模块 B，那么模块 B 总是在模块 A 之前初始化。这样，模块 A 就能够使用、设置、更改或覆盖模块 B 定义的配置和服务。图 5.1 中的模块的初始化顺序是 G→F→E→D→B→C→A。开发者不需要知道确切的初始化顺序，只要知道如果某个模块依赖模块 X，那么模块 X 一定会在该模块之前初始化。

使用[DependsOn]特性声明模块间的依赖关系，代码如下：

```
[DependsOn(typeof(ModuleB), typeof(ModuleC))]
public class ModuleA : AbpModule
{
}
```

这段代码中，ModuleA 使用[DependsOn]特性声明其依赖 ModuleB 和 ModuleC。

对于 ASP.NET Core 应用程序来说，启动模块(示例中的 ModuleA)负责初始化 ASP.Net Core 请求处理管道，代码如下：

```
[DependsOn(typeof(ModuleB), typeof(ModuleC))]
public class ModuleA : AbpModule
{
    //...
      public override void OnApplicationInitialization(ApplicationInitializationContext
context)
    {
        var app = context.GetApplicationBuilder();
        var env = context.GetEnvironment();

        app.UseRouting();
        if (env.IsDevelopment())
        {
```

```
            app.UseDeveloperExceptionPage();
        }
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();
        });
    }
}
```

这段代码构建了一个与 5.2.1 节相同的 ASP. NET Core 请求处理管道。context. GetApplicationBuilder()和 context. GetEnvironment()只是从依赖注入系统中获取标准的 IApplicationBuilder 服务和 IWebHostEnvironment 服务的快捷方法。

然后，开发者可以在 ASP. NET Core 的 Startup 类中使用该模块，从而把 ABP 框架和 ASP. NET Core 集成在一起，代码如下所示：

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddApplication<ModuleA>();
    }

    public void Configure(IApplicationBuilder app)
    {
        app.InitializeApplication();
    }
}
```

services. AddApplication()方法是由 ABP 框架定义的，用于配置模块。它基本上是按照模块依赖关系的顺序执行所有模块的 ConfigureServices 方法。ABP 框架也定义了 app. InitializeApplication()方法。同样，它也是根据模块依赖关系的顺序执行所有模块的 OnApplicationInitialization 方法。ConfigureServices 方法和 OnApplicationInitialization 方法是模块类中最常用的方法。

5.2.4　模块的生命周期

AbpModule 类定义了一些方法，开发者可以通过重写这些方法在应用程序启动和关闭时执行一些自定义代码。模块包含以下 7 个生命周期方法。

(1) PreConfigureServices：该方法在 ConfigureServices 方法之前执行。在该方法中定义了一些需要在依赖模块的 ConfigureServices 方法之前执行的代码。

(2) ConfigureServices：如 5.2.3 节所述，该方法主要用于配置模块和注册服务。

(3) PostConfigureServices：该方法在所有模块(依赖于该模块的所有模块)的 ConfigureServices 方法执行之后调用，因此可以执行最后的配置。

(4) OnPreApplicationInitialization：该方法在 OnApplicationInitialization 之前调用。该阶段可以从依赖注入系统中获取服务。

(5) OnApplicationInitialization：如 5.2.3 节所述，在该方法中可以配置 ASP. NET Core 的请求处理管道和初始化服务。

（6）OnPostApplicationInitialization：该方法在初始化阶段调用。

（7）OnApplicationShutdown：在必要的时候，开发者可以在该方法中实现模块的关闭逻辑。

带 Pre 和 Post 前缀的方法（如 PreConfigureServices 和 PostConfigureServices）与相应的不带前缀的方法具有相同的作用。这些方法很少被使用，但是提供了一种可以在其他所有模块之前或之后执行一些配置/初始化代码的方式。

> **生命周期方法的异步版本**
>
> 本节中介绍的生命周期方法都是同步的。在本书撰写时，ABP 框架开发团队正在致力于在 ABP 框架的 5.1 版本中引入异步生命周期方法[①]，详情参阅 https://github.com/abpframework/abp/pull/10928。

5.3　依赖注入系统

依赖注入是一种获取类所依赖的对象的技术，实现了类对象创建和使用的分离。

假如有一个 UserRegistrationService 类，它使用 SmsService 发送验证码，代码如下：

```
public class UserRegistrationService
{
    private readonly SmsService _smsService;

    public UserRegistrationService(SmsService smsService)
    {
        _smsService = smsService;
    }

    public async Task RegisterAsync(
        string username,
        string password,
        string phoneNumber)
    {
        //把用户信息保存在数据库中
        await _smsService.SendAsync(
            phoneNumber,
            "Your verification code: 1234"
        );
    }
}
```

在这段代码中，SmsService 是通过构造函数注入模式（constructor-injection pattern）获得的。构造函数注入模式意味着需要先为类定义一个带参数的构造函数，参数就是需要注入的类对象。依赖注入系统在实例化该类时，先实例化依赖项，并把它传递给该类的构造函数，然后把这些对象实例赋值给类中的字段，以便在后面的代码中使用它们。在该示例中，RegisterAsync 方法在把用户信息保存到数据库后，使用 SmsService 发送验证码。

ASP.NET Core 自带一个依赖注入框架，ABP 框架直接使用了该框架而没有引入第三方依赖注入框架。一旦把所有的服务都注册到依赖注入系统中，任意服务都可以通过构造

① 译者注：模块的异步生命周期方法可参阅 https://docs.abp.io/en/abp/5.2/Module-Development-Basics。

函数注入的方式注入目标服务中，而不需要手动创建它们及它们的依赖项。

5.3.1　服务的生命周期

在设计服务时，应该考虑的最重要的事情是服务的生命周期。ASP. NET Core 在服务注册时提供了以下 3 个生命周期以供选择，开发者在注册服务时需要为每个服务选择一个生命周期。

(1) 瞬时(Transient)：只要注入瞬时服务，就会重新创建一个该服务的实例。每次获取或注入瞬时服务，都会创建一个新实例。

(2) 作用域(Scoped)：作用域服务是根据每个作用域的范围创建的。HTTP 请求的生命周期通常作为参考，因此在 ASP. NET Core 中每个 HTTP 请求都会创建一个作用域。在同一个作用域中共享相同的实例，在不同的作用域中得到的是不同的实例。

(3) 单例(Singleton)：单例服务在整个应用程序中只有一个实例。所有的请求和服务的使用者都共用同一个实例。该对象在第一次获取的时候创建，在后续获取该对象时将重用这个已创建的实例。

以下代码中定义的模块注册了两个服务，其中一个是瞬时服务，另一个是单例服务。

```
public class MyModule : AbpModule
{
    public override void ConfigureServices(ServiceConfigurationContext context)
    {
        context.Services.AddTransient<ISmsService,SmsService>();
        context.Services.AddSingleton<OtherService>();
    }
}
```

context. Services 是一个 IServiceCollection 类型的实例，所有 ASP. NET Core 的扩展方法都可以被用于手动注册和配置服务。在这段示例代码中，AddTransient < ISmsService, SmsService >()实现了把 SmsService 注册为 ISmsService 接口的服务。由于注册的是瞬时服务，因此每当需要注入 ISmsService 时，依赖注入系统就会新建一个 SmsService 对象。AddSingleton < OtherService >()把 OtherService 注册为单例服务。在需要使用这个服务时，应该注入 OtherService 类的引用。

> **作用域依赖和 ASP. NET Core 的依赖注入文档**
>
> 默认情况下，对于 ASP. NET Core 应用程序来说，需要为每个 HTTP 请求创建作用域服务。对于非 ASP. NET Core 应用程序来说，开发者需要自己管理作用域。关于依赖注入的所有细节，可以参阅 ASP. NET Core 的官方文档 https://docs. microsoft. com/en-us/aspnet/core/fundamentals/dependency-injection。

5.3.2　约定优先的服务注册方式

得益于 ABP 框架的约定优先和声明式的服务注册系统，在使用 ABP 框架时，开发者不必过多地考虑服务注册。

如 5. 3. 1 节所述，在 ASP. NET Core 中，开发者需要显式地把所有服务注册到

IServiceCollection 中。然而，这些注册代码大部分都是重复代码，能够自动完成。

ABP 框架自动把以下 8 种类型的服务自动注册到依赖注入系统中。

(1) MVC 控制器。

(2) Razor Page 模块。

(3) 视图组件。

(4) Razor 组件。

(5) SignalR 中心(SignalR hub)。

(6) 应用服务。

(7) 领域服务。

(8) 仓储。

所有这些服务都注册为瞬时生命周期服务。因此，开发者不需要关心这些类的服务注册问题。如果有其他类型的服务，开发者可以使用某个 DI 接口或 Dependency 特性来注册服务。

5.3.3　与依赖注入相关的接口

开发者可以通过实现 ITransientDependency、IScopedDependency 或 ISingletonDependency 接口的方式把服务注册到依赖注入系统中。例如，以下代码段以单例模式注册服务，因此在整个应用程序的生命周期中只创建一个共享实例。

```
public class UserPermissionCache : ISingletonDependency
{}
```

这 3 个与依赖注入相关的接口十分简单，也是大多数场景下建议使用的方法，但是与[Dependency]特性相比，它们有一定的局限性。

5.3.4　[Dependency]特性

[Dependency]特性可以通过以下 3 个属性来精细地控制依赖注册过程。

(1) Lifetime(枚举类型)：服务的生命周期，可选值包括 Singleton、Transient 或 Scoped。

(2) TryRegister(布尔类型)：只有在服务未注册的情况下才注册该服务。

(3) ReplaceServices(布尔类型)：如果服务已经注册了，则替换这个服务。

以下示例代码使用[Dependency]特性注册服务。

```
using Microsoft.Extensions.DependencyInjection;
using Volo.Abp.DependencyInjection;
namespace UserManagement
{
    [Dependency(ServiceLifetime.Transient,TryRegister = true)]
    public class UserPermissionCache
    {}
}
```

这段代码使用[Dependency]特性，并设置 Lifetime 属性为 Transient，还设置了

TryRegister 属性，把这个类注册到依赖注入系统中。

> **[Dependency]特性和依赖注入接口**
>
> [Dependency]特性可以与依赖注入接口一起使用。如果[Dependency]特性设置了 Lifetime 属性，那么它与依赖注入接口相比具有更高的优先级。

只有把一个类注册到依赖注入系统中，才能在应用程序中通过依赖注入获得该类的实例。然而，一个类可以使用不同的类或者接口的引用来注入其他类中，这取决于该类要公开的服务类型。

5.3.5 [ExposeServices]特性

如果一个类没有实现任何接口，那么就只能注入该类的实例。5.3.4 节介绍的 UserPermissionCache 类就是直接使用该类本身注入的。但是，一般情况下应该为服务定义接口。

假设抽象的短信发送接口代码如下：

```
public interface ISmsService
{
    Task SendAsync(string phoneNumber, string message);
}
```

这是一个非常简单的接口，只有一个发送短信的方法。假设想要使用 Azure 实现 ISmsService 接口，代码如下所示：

```
public class AzureSmsService : ISmsService,
ITransientDependency
{
    public async Task SendAsync(string phoneNumber, string message)
    {
        //TODO: ...
    }
}
```

AzureSmsService 类实现了 ISmsService 接口和 ITransientDependency 接口。ITransientDependency 接口仅用于将服务注册到依赖注入系统中。

开发者通常希望通过注入 ISmsService 接口来使用 AzureSmsService 类。ABP 框架足够“聪明”，能够理解开发者的意图，并在依赖注入系统中自动把 AzureSmsService 类注册为 ISmsService 接口的服务。开发者既可以通过注入 ISmsService 接口，也可以通过注入 AzureSmsService 的引用，来使用 AzureSmsService 类。在依赖注入系统中自动把 AzureSmsService 类注册为 ISmsService 接口的服务是通过命名约定实现的：ISmsService 接口是 AzureSmsService 类的默认接口，因为该类以 SmsService 后缀结尾。

假设有一个类，它实现了多个接口，代码如下所示：

```
public class PdfExporter: IExporter, IPdfExporter, ICanExport,ITransientDependency
{}
```

可以通过注入 IPdfExporter 接口、IExporter 接口或者 PdfExporter 类的引用的方式使

用 PdfExporter 服务。但是，因为 PdfExporter 类的名字不是以 CanExport 结尾的，所以不能使用 ICanExport 接口注入该服务。

如果需要改变以上的默认规则，那么可以使用[ExposeServices]特性，代码如下：

```
[ExposeServices(typeof(IPdfExporter))]
public class PdfExporter: IExporter, IPdfExporter, ICanExport,ITransientDependency
{}
```

这样就只能通过注入 IPdfExporter 接口来使用 PdfExporter 类了。

> **问题：应该为每个服务定义接口吗？**
>
> 开发者可能会困惑是否应该为每个服务都定义接口，并且使用接口注入这些服务。ABP 框架并不强迫开发者这样做，并且最佳实践是在遇到以下情况时定义接口：实现服务之间的松耦合；一个服务存在多个实现；在单元测试中方便地模拟服务；在物理上实现接口与实现的分离（例如在 Application.Contracts 项目中定义应用服务接口，在 Application 项目中实现这些服务接口，又如在领域层定义仓储接口，但是在基础设施层中实现这些接口）等。

至此已经介绍完注册和使用服务的方法。有些服务或者库存在一些可配置的选项，需要在使用它们之前对它们进行配置。5.4 节和 5.5 节将探讨如何基于选项模式配置这些服务和库。

5.4　应用程序配置

ASP.NET Core 的配置（configuration）系统为应用程序提供了一种方便的方式来读取基于键值对的配置信息。它是一个可扩展的系统，可以从各种资源中读取键值对，如 JSON 配置文件、环境变量、命令行参数和 Azure Key Vault[①]。

> **ABP 框架和 ASP.NET Core 的配置系统**
>
> ABP 框架没有向 ASP.NET Core 的配置系统添加新功能。不过，要正确地使用 ASP.NET Core 和 ABP 框架，理解配置系统的工作原理还是至关重要的。本书将介绍配置系统的基础知识，详情参阅 ASP.NET Core 的官网文档 https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration。

5.4.1　设置配置值

设置配置值最简单的方法是使用 appsettings.json 文件。假设构建一个使用 Azure 发送短信验证码的服务，需要使用以下两个配置值。

（1）Sender：展示给目标用户的发信人号码。

（2）ConnectionString：Azure 资源的连接字符串。

开发者可以在 appsettings.json 文件的配置部分定义这些值，代码如下：

---

① 由 Azure 提供的一项云服务，详情参阅 https://azure.microsoft.com/zh-cn/services/key-vault/。

```
{
    ...
    "AzureSmsService": {
      "Sender": " + 901112223344",
      "ConnectionString": "..."
    }
}
```

配置节的名称(这段代码中的 AzureSmsService)和键的名称(这段代码中的 Sender 和 ConnectionString)可以是任意值,只要和代码中使用的名称相同即可。

一旦在配置文件中定义了这些值,开发者就能够方便地在应用程序代码中读取它们。

5.4.2　读取配置值

当需要读取配置值时,可以注入和使用 IConfiguration 服务。获取 Azure 配置,并使用 AzureSmsService 类发送短信的代码如下:

```
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Volo.Abp.DependencyInjection;
namespace SmsSending
{
    public class AzureSmsService : ISmsService,ITransientDependency
    {
        private readonly IConfiguration _configuration;

        public AzureSmsService(IConfiguration configuration)
        {
            _configuration = configuration;
        }

        public async Task SendAsync(string phoneNumber, string message)
        {
            string sender = _configuration["AzureSmsService:Sender"];
            string ConnectionString = _configuration["AzureSmsService:ConnectionString"];
            //TODO: 使用 Azure 发送信息
      }
    }
}
```

该类使用 IConfiguration 服务获取配置值,并且使用符号“:”访问嵌套部分的值。在该示例中,AzureSmsService:Sender 用于获取 AzureSmsService 节中 Sender 的值。

在模块的 ConfigureServices 方法中也可以使用 IConfiguration 服务,代码如下:

```
public override void ConfigureServices(ServiceConfigurationContext context)
{
    IConfiguration configuration = context.Services.GetConfiguration();
    string sender = configuration["AzureSmsService:Sender"];
}
```

通过这种方式,开发者甚至可以在依赖注入注册阶段完成前访问配置值。

配置系统为配置应用程序和获取键值对样式的设置信息提供了一种优雅的方式。然而,如果构建的是可重用库,那么选项模式为定义类型安全的选项提供了一种更好的方法。

5.5　选项模式

在选项模式中,可以使用一个普通的类,有时也称作 POCO(Plain Old C# Object,简单传统的 C#对象)来定义一组相关的选项。接下来介绍如何基于选项模式定义、设置和使用配置信息。

5.5.1　定义选项类

选项类是一个简单的 C#类。以下代码为 Azure SMS 服务定义了一个选项类。

```
public class AzureSmsServiceOptions
{
    public string Sender { get; set; }
    public string ConnectionString { get; set; }
}
```

在选项类中添加 Options 后缀是一种惯例。一旦定义了这样的类,任何使用该服务的模块都可以方便地配置这些选项的值。

5.5.2　配置选项

如 5.2 节所述,可以在模块的 ConfigureServices 方法中配置它所依赖的模块的服务。可以在该方法中调用 IServiceCollection.Configure 扩展方法为任意选项类配置值。以下代码展示了配置 AzureSmsServiceOptions 的方法。

```
[DependsOn(typeof(SmsSendingModule))]
public class MyStartupModule : AbpModule
{
    public override void ConfigureServices(ServiceConfigurationContext context)
    {
        context.Services.Configure<AzureSmsServiceOptions>(options =>
        {
            options.Sender = "+901112223344";
            options.ConnectionString = "...";
        });
    }
}
```

context.Services.Configure 方法是一个泛型方法,选项类是它的泛型参数。它的参数是一个委托(Action),用于设置选项的值。这段示例代码在 lambda 表达式中设置了 Sender 和 ConnectionString 的值,从而实现配置 AzureSmsServiceOptions 的目的。

AbpModule 基类提供了一个 Configure 方法作为 context.Services.Configure 方法的快捷方式,所以上述代码可以重写为如下代码:

```
public override void ConfigureServices(ServiceConfigurationContext context)
{
    Configure < AzureSmsServiceOptions >(options = >
    {
        options. Sender = " + 901112223344";
        options. ConnectionString = "...";
    });
}
```

这段代码只是把 context. Services. Configure <…>方法替换为了 Configure <…>。配置选项的值很简单，下面介绍如何使用这些配置的值。

多次配置选项

在应用程序中可以多次配置相同的选项。所有委托得到的选项是同一个实例，所以后面的配置值将覆盖之前的配置值。如果多个模块配置了相同的选项，则以最后一个模块的配置值为准。需要注意的是，模块是按照依赖顺序初始化的。

5.5.3　使用选项值

ASP. NET Core 提供了一个用于注入选项类的 IOptions < T >接口来读取配置的值。以下代码重写了 AzureSmsService 类，注入了 AzureSmsServiceOptions 服务，而不是 IConfiguration 服务。

```
public class AzureSmsService : ISmsService,
ITransientDependency
{
    private readonly AzureSmsServiceOptions _options;

    public AzureSmsService(IOptions < AzureSmsServiceOptions > options)
    {
        _options = options. Value;
    }

    public async Task SendAsync(string phoneNumber, string message)
    {
        string sender = _options. Sender;
        string ConnectionString = _options. ConnectionString;
        //TODO...
    }
}
```

在这段代码中，通过注入 IOptions < AzureSmsServiceOptions >得到注入对象的 Value 属性，从而得到 AzureSmsServiceOptions 实例。IOptions < T >接口由 Microsoft. Extensions. Options 包定义，是注入选项类的标准方式。它在内部执行所有的 Configure 方法，并向外部提供一个已配置好的选项类的实例。如果直接注入 AzureSmsServiceOptions 类，系统将会抛出一个依赖注入的异常，因此必须注入 IOptions < AzureSmsServiceOptions >。

至此已经介绍了选项的定义、配置和使用，接下来将探讨如何使用配置系统来设置使用

选项模式定义的选项的值。

5.5.4　通过配置系统设置选项值

选项模式允许通过多种方式设置选项的值。这意味着可以使用 IConfiguration 服务读取应用程序的配置信息来设置选项的值。以下代码展示了从配置服务获取配置信息来设置 AzureSmsServiceOptions 的值的方法。

```
[DependsOn(typeof(SmsSendingModule))]
public class MyStartupModule : AbpModule
{
    public override void ConfigureServices(ServiceConfigurationContext context)
    {
        var configuration = context.Services.GetConfiguration();
        Configure<AzureSmsServiceOptions>(options =>
        {
            options.Sender = configuration["AzureSmsService:Sender"];
            options.ConnectionString = configuration["AzureSmsService:ConnectionString"];
        });
    }
}
```

这段代码使用 context.Services.GetConfiguration()方法获取 IConfiguration 接口，然后使用配置值来设置选项值。

由于这种用法非常常见，因此以上功能有更简便的实现方式，代码如下：

```
public override void ConfigureServices(ServiceConfigurationContext context)
{
    var configuration = context.Services.GetConfiguration();
    Configure<AzureSmsServiceOptions>(configuration.GetSection("AzureSmsService"));
}
```

在这种用法中，Configure 方法使用配置节的名称作为参数，而不是使用一个委托。它通过命名约定自动匹配配置键和选项类的属性。如果配置信息中没有定义 AzureSmsService 节，那么这段代码对选项的值没有任何影响。

选项模式为开发者提供了更多的灵活性，开发者可以从 IConfiguration 或任何其他源设置这些选项的值。

> **默认从配置信息设置选项的值**
>
> 在开发可重用模块时，最好尽可能从配置信息[①]中设置选项的值。通过这种方式，开发者能够从 appsettings.json 文件中配置模块。

5.5.5　ABP 框架和 ASP.NET Core 的选项

ABP 框架和 ASP.NET Core 强烈推荐使用选项模式来配置应用程序。以下代码展示

① 从 context.Services.GetConfiguration 获取配置信息，尽量避免在代码中硬编码选项值。

了配置 ABP 框架中一个选项的方法。

```
Configure < AbpAuditingOptions >(options = >
{
    options. IgnoredTypes. Add(typeof(ProductDto));
});
```

AbpAuditingOptions 由 ABP 框架的审计日志系统定义。上述代码用于在审计日志中忽略 ProductDto。

以下代码展示了配置 ASP. NET Core 中一个选项的方法。

```
Configure < MvcOptions >(options = >
{
    options. RespectBrowserAcceptHeader = true;
});
```

MvcOptions 由 ASP. NET Core 定义，用于配置 ASP. NET Core MVC 框架的行为。

> **选项类中的复杂类型**
>
> 注意，AbpAuditingOptions. IgnoredTypes 是一个 Type 类型的列表容器，它不是一个可以在 appsettings. json 文件中定义的简单的原始类型。这是选项模式的优点之一：可以定义具有复杂类型的属性，甚至可以是一个回调委托。

配置系统和选项模式提供了一种方便的方式来配置和定制使用的服务的行为。开发者可以配置 ASP. NET Core 和 ABP 框架，并为自定义的服务定义配置选项。

5.6　日志

记录日志是一项在每个应用程序中都很常用的功能。ASP. NET Core 提供了一个简单而高效的日志系统，它可以集成一些流行的日志库，如 NLog、Log4Net 和 Serilog。

Serilog 是一个广泛使用的库，提供了许多记录日志的目标载体，包括控制台、文本文件和 Elasticsearch。ABP 框架的启动模板已经安装并配置好了 Serilog 库。它把日志写入应用程序所在文件夹的 Logs 子目录中，因此开发者可以直接在自己的服务中使用日志记录系统。当然，也可以通过配置使 Serilog 把日志写入不同的目标载体中，可以参阅 Serilog 的文档。Serilog 所有的配置信息都包含在了 ABP 框架的启动模板中。Serilog 不是 ABP 框架的核心依赖项，因此开发者可以方便地把其替换为其他日志记录库。

在 ASP. NET Core 应用程序中，ILogger < T >接口用于记录日志，其中 T 通常是服务的类型。

服务类记录日志的示例代码如下：

```
public class AzureSmsService : ISmsService, ITransientDependency
{
    private readonly ILogger < AzureSmsService > _logger;

    public AzureSmsService(ILogger < AzureSmsService > logger)
```

```
        {
            _logger = logger;
        }

        public async Task SendAsync(string phoneNumber, string message)
        {
            _logger.LogInformation( $ "Sending SMS to {phoneNumber}: {message}");
            //TODO...
        }
    }
```

ILogger < AzureSmsService >服务从构造函数注入 AzureSmsService 类中，并在该类中调用 LogInformation 方法把信息级别的日志写入日志系统中。

ILogger 接口还有一些其他方法用于输出不同严重级别的日志，如 LogError 和 LogDebug，详情可参阅 ASP. NET Core 的官方文档 https://docs. microsoft. com/en-us/aspnet/core/fundamentals/logging。

5.7　小结

本章主要介绍了 ASP. NET Core 和 ABP 框架的核心构件，讨论了在应用程序启动时使用 Startup 类、配置系统和选项模式来配置 ASP. NET Core 和 ABP 框架的服务并在需要时实现自己的配置选项的方法。

ABP 框架提供的模块化系统扩展了 ASP. NET Core 的初始化系统和配置系统，从而支持创建多模块应用程序，其中每个模块仅负责初始化本模块的服务并配置本模块的依赖。通过这种方式，开发者可以把应用程序划分为多个模块以便更好地组织代码库，或者可以创建在不同应用程序中重用的模块。

依赖注入系统是 ASP. NET Core 应用程序中最基本的基础设置。一个服务可以使用依赖注入系统调用其他服务。本章介绍了依赖注入系统的基本用法，并探讨了 ABP 框架是如何简化服务注册的。

第 6 章将重点介绍数据访问基础设施，这是业务应用程序中一个非常重要的组件，还将探讨 ABP 框架如何规范实体的定义，以及如何使用仓储来抽象和执行数据库操作，并自动管理数据库连接和事务。