探索 ABP 基础架

**前言**

为了了解应用程序是如何配置和初始化，本文将探讨ASP.NET Core和ABP框架最基本的构建模块。我们将从 ASP.NET Core 的 Startup类开始了解为什么我们需要模块化系统，以及 ABP 如何提供模块化方式来配置和初始化应用程序。

然后我们将探索 ASP.NET Core 的依赖注入，以及ABP是如何使用预定义规则（predefined rules）自动进行依赖注入。

最后，我们将了解 ASP.NET Core 的配置和选项框架，以及其他类库。

以下是本文的所有主题：

* 了解模块化
* 使用依赖注入系统
* 配置应用程序
* 实现选项模式
* 日志系统

**一、了解模块化**

**模块化**是一种将大型软件按功能分解为更小的部分，并允许每个部分通过标准化接口进行通信。模块化有以下主要好处：

* 模块按规则进行隔离后，大大降低了系统复杂性。
* 模块之间松散耦合，提供了更大的灵活性。因为模块是可组装、可替换的。
* 因为模块是独立的，所以它允许跨应用被重用。

大多数企业的软件被设计成模块化，但是，实现模块化并不容易。ABP 框架的主要目标之一是为模块化提供基础设施和工具。我们将在后面详细介绍模块化开发，本节只介绍 ABP 模块的基础知识。

**Startup 类**

在定义ABP的模块之前，建议先熟悉 ASP.NET Core 中的StartUp类，我们看下ASP.NET Core 的Startup类：

public class Startup  
{  
 public void ConfigureServices(IServiceCollection services)  
 {  
 services.AddMvc();//注入MVC组件，自带服务注入  
 services.AddTransient<MyService>();  
 }  
 public void Configure(IApplicationBuilder app, IWebHostEnvironment env)  
 {  
 app.UseRouting();//指定使用路由  
 if (env.IsDevelopment())  
 {  
 app.UseDeveloperExceptionPage();  
 }  
 app.UseEndpoints(endpoints =>  
 {  
 endpoints.MapControllers();  
 });  
 }  
}

ConfigureServices方法用于配置服务并将新服务注册到依赖注入系统。另一方面，Configure方法用于配置 ASP.NET Core 管道中间件，用于处理 HTTP 请求。在应用程序启动之前，我们需要在Program.cs中配置Startup类：

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>();  
            });  
}

这个Startup类是独一无二的，我们只有一个点来配置和初始化所有的服务。但是，在模块化应用程序中，我们希望每个模块都能独立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于其他模块，因此模块配置顺序和初始化就非常重要了。我们来看下 ABP 的模块是如何定义的

**模块定义**

ABP 模块是一组类型（比如类或接口），它们一同开发一同交付的。它是一个程序集（一般来说是Visual Studio 中的一个*项目*），派生自AbpModule，模块类负责配置和初始化，并在必要时配置依赖模块。

下面是一个短信发送模块的简单定义：

**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，以便将其服务注册到依赖注入系统。此示例中的SmsService服务被注册为瞬态生命周期。该示例和上面Startup类似。但是，大多时候，您不需要手动注册服务，这要归功ABP 框架的按约定注册系统。

OnApplicationInitialization方法用在服务注册完成后，并且在应用准备就绪后执行。使用此方法，您可以在应用启动时执行任何操作。例如，您可以初始化一个服务：

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

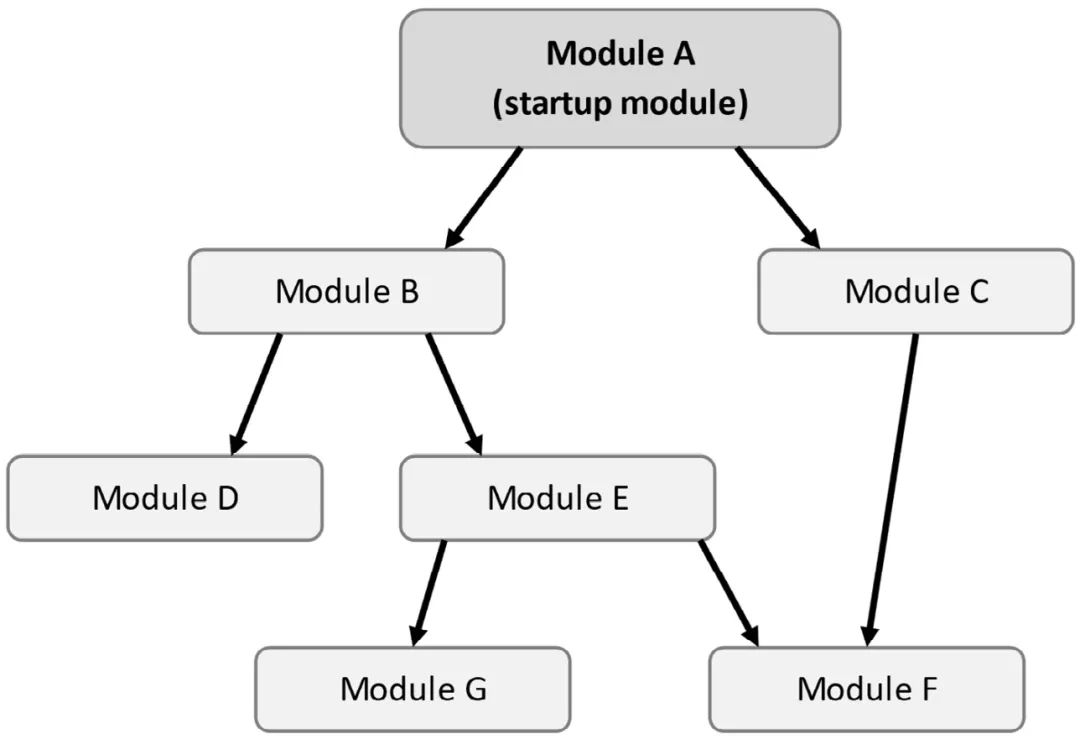
这里，我们使用context.ServiceProvider从依赖注入系统请求并初始化服务。可见，此时服务已经完成注册。

您也可以将OnApplicationInitialization方法等同于Startup类的Configure方法。

您可以在此处构建 ASP.NET Core 请求管道。但是，通常我们会在**启动模块中配置请求管道**，如下一节所述。

**模块依赖和启动模块**

一个业务应用通常由多个模块组成，ABP 框架允许您声明模块之间的依赖关系。一个应用必须要有一个**启动模块**。启动模块可以依赖于其他模块，其他模块可以再依赖于其他模块，以此类推。

下图是一个简单的模块依赖关系图：

如图所示，如果模块 A 依赖于模块 B，则模块 B 总是在模块 A 之前初始化。这允许模块 A 使用、设置、更改或覆盖模块 B 定义的配置和服务。

对于示例图，模块初始化的顺序应该是：G、F、E、D、B、C、A。

您不必知道确切的初始化顺序；只需要知道如果你的模块依赖于模块xx，那么模块xx在你的模块之前被初始化。

ABP使用[DependsOn]（属性声明）方式来定义模块依赖：

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

这里，ModuleA通过[DependsOn]依赖于ModuleB和ModuleC。本例中，启动模块ModuleA负责设置ASP.NET Core 的请求管道：

[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();  
        });  
    }  
}

[代码块和之前ASP.NET Core的 *Startup类* 创建请求管道相同。context.GetApplicationBuilder()和context.GetEnvironment()用于从依赖注入中获IApplicationBuilder和IWebHostEnvironment服务。

最后，我们在Startup里将ASP.NET Core 和 ABP 框架进行集成：

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

//添加用于将Http请求重定向到Https的中间件

app.UseHttpsRedirection();

//为当前请求路径启用静态文件服务

app.UseStaticFiles();

app.UseMvc(routes =>{

routes.MapRoute(

name:”default”,

template: “{controller=Home}/{action}/{id?}”);

});

app.InitializeApplication();  
    }  
}

services.AddApplication()方法由 ABP 框架定义，用于ABP的模块配置。它按顺序执行了所有模块的ConfigureServices方法。而app.InitializeApplication()方法也是由 ABP 框架定义，它也是按照模块依赖的顺序来执行所有模块的OnApplicationInitialization方法。

ConfigureServices和OnApplicationInitialization方法是模块类中最常用的方法。

**模块生命周期**

AbpModule中定义的生命周期方法，除了上面看到的ConfigureServices和OnApplicationInitialization，下面罗列其他生命周期相关方法：

* PreConfigureServices: 这个方法在ConfigureServices方法之前被调用。它允许您配置服务之前执行的代码。
* ConfigureServices：这是配置模块和注册服务的主要方法。
* PostConfigureServices: 该方法在ConfigureServices之后调用（包括依赖于您模块的模块），这里可以配置服务后执行的代码。
* OnPreApplicationInitialization: 这个方法在OnApplicationInitialization之前被调用。在这个阶段，您可以从依赖注入中解析服务，因为服务已经被初始化。
* OnApplicationInitialization：此方法用来配置 ASP.NET Core 请求管道并初始化您的服务。
* OnPostApplicationInitialization: 这个方法在初始化阶段后被调用。
* OnApplicationShutdown：您可以根据需要自己实现模块的关闭逻辑。带Pre…和Post…前缀的方法与原始方法具有相同的目的。它们提供了一种在模块之前或之后执行的一些配置/初始化代码，一般情况下我们很少使用到。

**异步生命周期方法**

本节介绍的生命周期方法是同步的。在编写本书时，ABP 框架团队正努力在 框架 5.1 版本引入异步生命周期方法。

如前所述，模块类主要包含**注册和配置**与该模块相关的服务的代码。在下一节中，我们将介绍如何使用 ABP 框架注册服务。

**二、使用依赖注入系统**

**.NET 原生依赖注入**

依赖注入是一种获取类的依赖的技术，它将创建类与使用该类分开。

假设我们有一个UserRegistrationService类，它调用SmsService类来发送验证短信，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)  
    {  
        *//...save user in the database*  
        await \_smsService.SendAsync(  
            phoneNumber,  
            "Your verification code: 1234"  
        );  
    }  
}

这里的SmsService使用**构造函数注入**来获取实例。也就是说，依赖注入系统会自动帮我们实例化类的依赖项，并将它们赋值给我们的\_smsService。

注意：ABP采用的是ASP.NET Core原生的依赖注入框架，他自己并没有发明依赖注入框架。

在设计服务时，我们还要考虑另外一件重要的事情：服务生命周期。ASP.NET Core 为服务注册提供了三个生命周期选项：

* **Transient（瞬态）**：每次您请求/注入服务时，都会创建一个新实例。
* **Scoped（范围）**: 通常这由请求生命周期来评估，您只有在同一范围内才能共享相同的实例。
* **Singleton（单例）**：在应用内有且仅有一个实例。所有请求都使用相同的实例。该对象在第一次请求创建。以下模块注册了两个服务，一个是瞬态的，另一个是单例的：

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

context.Services的类型是IServiceCollection，它是一个扩展方法。

在第一个示例中使用接口注册，第二个示例使用引用类注册为单例。

**ABP的依赖注入**

使用 ABP 框架时，您不必考虑服务注册，这要归功于 ABP 框架独特的服务注册系统。

**1、约定式注册**

在 ASP.NET Core 中，所有服务需要显式注册到IServiceCollection，如上一节所示。这些注册大多重复，完全可以自动化操作。

ABP 对于以下类型采用自动注册：

* MVC controllers
* Razor page models
* View components
* Razor components
* SignalR hubs
* Application services
* Domain services
* Repositories 以上类型均使用瞬态生命周期自动注册。如果您还有别的类型，可以考虑接口注册。

**2、接口注册**

您可以实现以下三种接口来注册：

* ITransientDependency
* IScopedDependency
* ISingletonDependency

例如，在下面代码块中，我们将服务注册为单例：

public class UserPermissionCache : ISingletonDependency  
{ }

接口注册很容易并且是推荐的方式，但与下面的属性注册相比，它有一定的局限性。

**3、属性注册**

属性注册更精细，下面是和属性注册相关的配置参数

* Lifetime(enum): 服务的生命周期，包括Singleton,Transient和Scoped
* TryRegister(bool)：仅当服务尚未注册时才注册
* ReplaceServices(bool)：如果服务已经注册，则替换之前的注册

示例代码：

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

}

**4、接口属性混合注册**

属性接口一起使用。如果属性定义了属性，属性比接口优先级更高。

如果一个类可能被注入不同的类或接口，具体取决于暴露的类型。

**暴露服务**

当一个类没有实现接口时，只能通过类引用注入。上一节中的UserPermissionCache类就是通过注入类引用来使用的。

假设我们有一个抽象 SMS 发送的接口：

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

假设您要ISmsService实现 Azure 服务：

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

这里的AzureSmsService实现了ISmsService和ITransientDependency两个接口。而ITransientDependency接口才是用于自动注册到依赖注入中的。这里的注入主要通过**命名约定**来实现，因为AzureSmsService以SmsService作为后缀结尾。我们再举一个通过命名约定的例子，假设我们有一个实现多个接口的类：

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

PdfExporter服务可以通过注入IPdfExporter和IExporter接口来使用，也可以直接注入PdfExporter类引用来使用。但是，您不能使用ICanExport接口注入它，因为名称PdfExporter不以CanExport为后缀。

一旦您使用该ExposeServices属性来暴露服务，如以下代码块所示：

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

现在，您只能通过注入IPdfExporter接口来使用PdfExporter类。

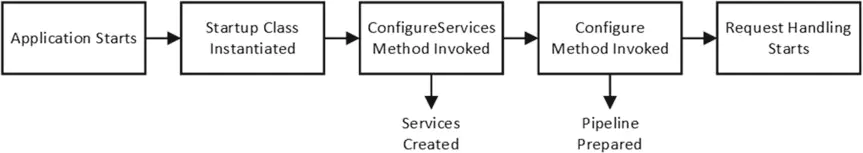
我应该为每个服务定义接口吗？

ABP 不会强迫你这么做，但是通用接口来定义是最佳实践：如果你想松散地耦合你的服务。比如，在单元测试中可以轻松模拟测试数据。

这就是为什么我们将接口与实现物理分离（例如，我们在项目中定义Application.Contracts接口，并在Application项目中实现它们，或者在领域层中定义存储库接口，在基础设施层中实现它们）。

我们已经了解了如何注册和消费服务。另外，某些服务具有选项配置，您需要在使用它们之前对其进行配置。接下来的两节将展开介绍。

应用启动的流程如下图所示：



**关于DDD和三层架构区别**

最大的区别在于设计思想的不同：DDD以领域为中心；三层以数据库为中心。一个是领域驱动；一个是数据驱动。DDD使用领域对象封装和实现需求的复杂性，领域可以复用；三层架构基于数据库建模，常规操作为：CRUD。