第5章

视频讲解

书山有路勤为径　学海无涯苦作舟
——MAUI数据访问

5.1　本地数据库

5.1.1　环境搭建

第2章中讲述了Sqlite数据库，作为一款轻量级的关系数据库，其资源占用小，对资源有限的智能手机来说，Sqlite数据库特别适合嵌入式开发的场景。

搭建Sqlite数据库环境非常简单，安装Sqlite相关的软件包即可。NuGet作为Visual Studio的扩展，是一款开源的包管理开发工具，使用该工具可以非常方便地将第三方的组件库整合至项目中，类似于Maven和Npm等工具。

第一种安装程序包的方法是采用命令行的方式进行。在Visual Studio中选择“工具”→“NuGet包管理器(N)”→“程序包管理控制台(O)”选项后，打开PowerShell控制台执行安装命令安装相关库。

【例5-1】 使用PowerShell安装软件包。

```
1  Install-Package sqlite-net-pcl
2  Install-Package SQLitePCLRaw.bundle_green
```

第二种安装程序包的方法是采用图形化界面的方式进行。在Visual Studio中选择“工具”→“NuGet包管理器(N)”→“管理解决方案的NuGet程序包(N)”选项，打开NuGet管理向导后，选择相应的项目，搜索需要安装的软件包安装即可。管理解决方案的NuGet程序包如图5-1所示。

第三种安装程序包的方法是采用修改工程配置文件的方式进行。双击项目打开MAUIDemo.csproj配置文件，找到相应的配置节点并追加如下配置。建议安装最新稳定版本。

【例5-2】 工程方式配置Sqlite。

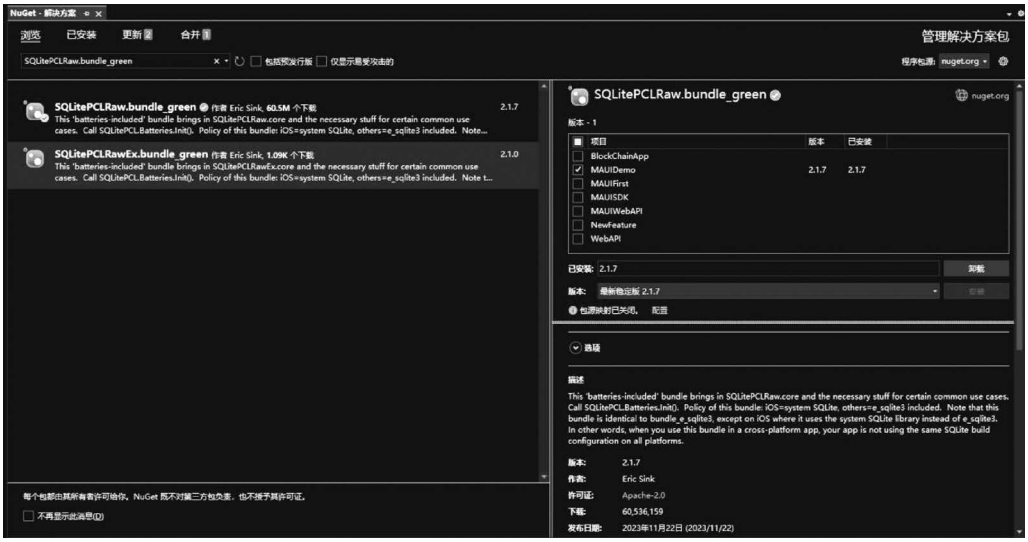

图 5-1　管理解决方案的 NuGet 程序包

MAUIDemo.csproj 代码如下：

```
<ItemGroup>
  <PackageReference Include="sqlite-net-pcl" Version="1.8.116" />
  <PackageReference Include="SQLitePCLRaw.bundle_green" Version="2.1.7" />
</ItemGroup>
```

5.1.2　功能封装

Sqlite 数据库支持多种编程语言，本节采用 C# 进行相关功能的封装。软件工程的核心思想之一是封装，封装的目的是便于后期各种工程复用。对于任何包含主键的表，需要定义基本模型。

【例 5-3】 Sqlite 数据库功能封装。

BaseModel.cs 代码如下：

```
using SQLite;

namespace MAUISDK.Models
{
    public class BaseModel
    {
        [PrimaryKey, AutoIncrement]
        public long Id
        {
            get;
            set;
        }
    }
}
```

上述基本模型仅有一个字段，Id 代表数据库索引，所以添加注解 PrimaryKey 表示主键，注解 AutoIncrement 表示自动增长，这样无须用户手工维护主键索引。基本模型能

够满足大部分建表需求。定义业务对象时，只需要继承基本模型即可。

MAUILearning.cs 代码如下：

```
namespace MAUISDK.Models
{
    public class MAUILearning : BaseModel
    {
        public string ?Learning
        {
            get;
            set;
        }
    }
}
```

MAUILearning 类作为 Sqlite 数据库操作示例模型，继承 BaseModel，为简化起见，仅定义一个 Learning 字段。

IMAUIRepository.cs 代码如下：

```
using MAUISDK.Models;

namespace MAUISDK.Interfaces
{
    public interface IMAUIRepository<T> where T : BaseModel
    {
        bool Exist(long Id);
        T Find(long Id);
        void Insert(T Item);
        void Update(T Item);
        void Remove(long Id);
    }
}
```

在接口文件夹中定义 IMAUIRepository 接口，仓储接口定义了增、删、查、改 4 个基本操作，同时定义了 Exist()方法判断搜索对象是否存在。为保证封装的通用性，这里采用泛型机制，添加 where T ：BaseModel 约束表明泛型对象需要继承 BaseModel 类。

MAUIRepository.cs 代码如下：

```
using MAUISDK.Interfaces;
using MAUISDK.Models;

namespace MAUISDK.Services
{
    public class MAUIRepository<T> : IMAUIRepository<T> where T : BaseModel
    {
        public List<T> ItemList;
        public MAUIRepository()
        {
            ItemList = new();
        }
        public void Remove(long Id)
        {
            ItemList.Remove(Find(Id));
        }
        public bool Exist(long Id)
```

```
18          {
19              return ItemList.Any(item => item.Id == Id);
20          }
21          public T Find(long Id)
22          {
23              return ItemList.FirstOrDefault(item => item.Id == Id)!;
24          }
25          public void Insert(T Item)
26          {
27              ItemList.Add(Item);
28          }
29          public void Update(T Item)
30          {
31              int index = ItemList.IndexOf(Find(Item.Id));
32              ItemList.RemoveAt(index);
33              ItemList.Insert(index, Item);
34          }
35      }
36  }
```

上述代码实现了 IMAUIRepository 接口，对 Lambda 表达式还不熟悉的读者可以参阅本书 1.4 节中的相关内容。仓储中的核心数据结构是 ItemList 列表，用于维护一系列对象。

BaseViewModel.cs 代码如下：

```
1   using System.ComponentModel;
2   using System.Runtime.CompilerServices;
3
4   namespace MAUIDemo.ViewModels
5   {
6       public class BaseViewModel : IQueryAttributable, INotifyPropertyChanged
7       {
8           public event PropertyChangedEventHandler PropertyChanged;
9    public virtual void ApplyQueryAttributes(IDictionary<string, object> query)
10          {
11          }
12          protected virtual void OnPropertyChanged([CallerMemberName] string propertyName =
    null)
13          {
14  PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
15          }
16      }
17  }
```

与基本模型相对应，上述代码中定义了基本视图模型 BaseViewModel。BaseViewModel 继承了 IQueryAttributable 和 INotifyPropertyChanged 接口，目的是监听数据动态变化，及时通知视图层。实现了接口的 OnPropertyChanged()方法，定义事件委托 PropertyChangedEventHandler 调用 Invoke()方法触发属性变更动作。

MAUILearningDAO.cs 代码如下：

```
1   using MAUISDK.Core;
2   using MAUISDK.Models;
3   using SQLite;
```

```
4
5   namespace MAUIDemo.Core
6   {
7       public class MAUILearningDAO
8       {
9           private SQLiteAsyncConnection Database;
10          public MAUILearningDAO()
11          {
12          }
13          public async Task LogAsync()
14          {
15              await Database.EnableWriteAheadLoggingAsync();
16          }
17          public async Task Init()
18          {
19              if (Database is not null)
20                  return;
21           Database = new(GlobalValues.LocalDatabasePath, GlobalValues.Flags);
22              _ = await Database.CreateTableAsync<MAUILearning>();
23          }
24          public async Task<List<MAUILearning>> GetItemsAsync()
25          {
26              await Init();
27              return await Database.Table<MAUILearning>().ToListAsync();
28          }
29          public async Task<MAUILearning> GetItemAsync(int Id)
30          {
31              await Init();
32              return await Database.Table<MAUILearning>().Where(i => i.Id == Id).
    FirstOrDefaultAsync();
33          }
34          public async Task<int> SaveItemAsync(MAUILearning model)
35          {
36              await Init();
37              return await Database.InsertAsync(model);
38          }
39          public async Task<int> DeleteItemAsync(object primaryKey)
40          {
41              await Init();
42              return await Database.DeleteAsync(primaryKey);
43          }
44      }
45  }
```

DAO(Data Access Objects,数据访问对象)属于面向对象中的一种设计模式,该设计模式将底层的数据访问逻辑与上层的业务逻辑进行分离。数据访问逻辑专注于数据访问相关的操作,业务逻辑专注于业务执行流程。MAUILearningDAO 是针对 MAUILearning 类对应的数据表进行操作。内部引入 SQLiteAsyncConnection Sqlite 数据库异步连接对象,Init()方法完成初始化。GlobalValues.LocalDatabasePath 是自定义 MAUIDemo.db3 文件的路径。MAUIDemo.db3 文件是用于存储数据的 Sqlite 数据库文件,数据库相关的所有操作均会影响此文件内容。Gitee 的.gitignore 文件可添加 *.db3 配置,屏蔽上传 Sqlite 数据文件。与本地文件相对应的是本地数据库缓存路径,本地数据库缓存路径

是 Sqlite 数据库操作默认缓存文件的位置，针对不同的操作系统有一定的差异性。笔者的本地数据库缓存路径是 C:\Users\Administrator\AppData\Local\Packages\ *** \LocalState。

MAUILearningViewModel.cs 代码如下：

```
using MAUIDemo.Core;
using MAUISDK.Models;
using System.Collections.ObjectModel;
using System.Windows.Input;

namespace MAUIDemo.ViewModels
{
    public class MAUILearningViewModel : BaseViewModel
    {
        private MAUILearningDAO dao;
        public ObservableCollection<MAUILearning> MAUILearnings
        {
            get;
            set;
        }
        public ICommand OnSaveAsync
        {
            set;
            get;
        }
        public ICommand OnDeleteAsync
        {
            set;
            get;
        }
        public ICommand OnFlushAsync
        {
            set;
            get;
        }
        public MAUILearningViewModel()
        {
            MAUILearnings = new();
            dao = new();
            OnSaveAsync = new Command(async (text) =>
            {
              if (text != null && !String.IsNullOrWhiteSpace(text.ToString()))
                {
                    MAUILearning Item = new()
                    {
                        Learning = text.ToString()
                    };
                    await dao.SaveItemAsync(Item);
                }
                OnFlushAsync.Execute(null);
            });
            OnDeleteAsync = new Command(async (text) =>
            {
                if (text != null)
                {
```

```
51                      int del = -1;
52                      for (int i = 0; i < MAUILearnings.Count; i++)
53                      {
54                  if (MAUILearnings[i].Id.Equals(long.Parse(text.ToString())))
55                          {
56                              del = i;
57                              break;
58                          }
59                      }//for
60                      if (del != -1)
61                      {
62                          await dao.DeleteItemAsync(MAUILearnings[del]);
63                      }
64                  }
65                  OnFlushAsync.Execute(null);
66              });
67              OnFlushAsync = new Command(async () =>
68              {
69                  var items = await dao.GetItemsAsync();
70                  MainThread.BeginInvokeOnMainThread(() =>
71                  {
72                      MAUILearnings.Clear();
73                      foreach (var item in items)
74                      {
75                          MAUILearnings.Add(item);
76                      }
77                  });
78                  OnPropertyChanged("MAUILearnings");
79              });
80          }
81      }
82  }
```

MAUILearningViewModel 类继承 BaseViewModel 类。内部的核心成员 MAUILearningDAO 用于实际数据库操作。MAUILearnings 列表是 ObservableCollection 可观察的列表，底层使用观察者设计模式。观察者设计模式也称为发布-订阅模式，多个对象间存在一对多的依赖关系，当一个对象的状态发生改变时，所有依赖它的对象都将得到通知并且会被自动更新。观察者设计模式是一种数据更新的触发机制，可以降低目标与观察者之间的耦合关系。MAUI 视图模型中大量使用观察者设计模式，以保证数据的双向绑定。

MAUILearningViewModel 类相关方法如下。

OnSaveAsync()。用于新增数据对象。当界面输入数据不为空时，构造 MAUILearning 对象，调用 DAO 层封装的 SaveItemAsync()方法完成新增数据对象逻辑。最后调用 OnFlushAsync()方法实现界面层数据刷新。

OnDeleteAsync()。用于删除数据对象。依次遍历观察者列表中的所有内容，如果满足删除条件，调用 DAO 层封装的 DeleteItemAsync()方法完成删除数据对象逻辑。最后调用 OnFlushAsync()方法实现界面层数据刷新。

OnFlushAsync()。用于刷新数据对象，同步界面层数据显示。首先调用 GetItemsAsync()方法获取当前数据库中的全部对象，因为这里涉及多线程间互操作界面更新问题。多线程间互操作修改线程数据会导致线程不安全，MAUI 提供了解决此问题的一种思路，采

用主线程方式调用 MainThread.BeginInvokeOnMainThread()方法。该方法的参数是一个委托,委托内部完成 MAUILearnings 可观察数据对象的更新操作,最后调用父类的 OnPropertyChanged()方法完成属性更新的通知操作。

MAUILearningConfig.cs 代码如下:

```
using MAUISDK.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace MAUISDK.Configs
{
    public class MAUILearningConfig : IEntityTypeConfiguration<MAUILearning>
    {
        public void Configure(EntityTypeBuilder<MAUILearning> builder)
        {
            builder.Property(e => e.Learning).IsRequired();
        }
    }
}
```

上述 MAUILearningConfig 配置类完成数据表的自动生成,Configure()方法完成数据表的自动创建过程。内部根据需求配置表中的各个字段,如 IsRequired()表示该字段必须设置,不能为空。至此,完成了数据库相关的功能封装。

5.1.3　应用调用

应用调用逻辑在 SQLitePage 页面演示。

【例 5-4】 Sqlite 应用调用。

SQLitePage.xaml 代码如下:

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
    x:Class="MAUIDemo.Pages.SQLitePage"
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:models="clr-namespace:MAUIDemo.Models"
    xmlns:viewmodels="clr-namespace:MAUIDemo.ViewModels"
    Title="数据操作">
    <ContentPage.BindingContext>
        <viewmodels:MAUILearningViewModel />
    </ContentPage.BindingContext>
    <ContentPage.Resources>
        <Style TargetType="Label">
            <Setter Property="VerticalTextAlignment" Value="Center" />
            <Setter Property="HorizontalTextAlignment" Value="Center" />
            <Setter Property="HorizontalOptions" Value="Center" />
            <Setter Property="WidthRequest" Value="300" />
            <Setter Property="HeightRequest" Value="40" />
            <Setter Property="FontAttributes" Value="Bold" />
            <Setter Property="FontSize" Value="20" />
        </Style>
        <Style TargetType="Button">
            <Setter Property="HorizontalOptions" Value="Center" />
```

```
24              <Setter Property="WidthRequest" Value="300" />
25              <Setter Property="HeightRequest" Value="40" />
26              <Setter Property="BackgroundColor" Value="#512BD4" />
27          </Style>
28          <Style TargetType="Entry">
29              <Setter Property="HorizontalOptions" Value="Center" />
30              <Setter Property="HorizontalTextAlignment" Value="Center" />
31              <Setter Property="WidthRequest" Value="300" />
32              <Setter Property="HeightRequest" Value="40" />
33          </Style>
34          <DataTemplate x:Key="data">
35              <HorizontalStackLayout>
36                  <Label Text="{Binding Id}" WidthRequest="100" />
37                  <Label Text="{Binding Learning}" WidthRequest="200">
38                      <Label.GestureRecognizers>
39                          <TapGestureRecognizer Tapped="OnTapped" />
40                      </Label.GestureRecognizers>
41                  </Label>
42              </HorizontalStackLayout>
43          </DataTemplate>
44      </ContentPage.Resources>
45      <VerticalStackLayout>
46          <VerticalStackLayout
47              BindableLayout.ItemTemplate="{StaticResource data}"
48              BindableLayout.ItemsSource="{Binding MAUILearnings}"
49              HorizontalOptions="Center" />
50          <Entry x:Name="entry" Placeholder="输入信息" />
51          <Button
52              Command="{Binding OnSaveAsync}"
53            CommandParameter="{Binding Source={x:Reference entry}, Path=Text}"
54              Text="保存" />
55          <Button
56              Command="{Binding OnDeleteAsync}"
57            CommandParameter="{Binding Source={x:Reference entry}, Path=Text}"
58              Text="删除" />
59          <Button Command="{Binding OnFlushAsync}" Text="刷新" />
60      </VerticalStackLayout>
61  </ContentPage>
```

SQLitePage 页面说明如下。

ContentPage.BindingContext 配置节中定义了绑定的视图模型为 MAUILearningViewModel。

ContentPage.Resources 配置节中定义了 Label、Button 和 Entry 控件的通用样式。DataTemplate 定义了数据模板，数据模板名称为 data，后面引入时使用此名称。该数据模板使用了 GestureRecognizers 手势操作。当 Tapped 触摸事件触发时，执行 OnTapped 事件逻辑。

界面部分以 VerticalStackLayout 垂直布局方式进行展示，内部嵌套 VerticalStackLayout 垂直布局用于展示数据列表，BindableLayout.ItemTemplate 绑定的数据模板 ItemTemplate 引用了前面的数据模板资源 data。BindableLayout.ItemsSource 绑定的数据源绑定了可观察列表 MAUILearnings。接着定义了输入信息输入条目控件和两个按钮控件。两个按钮控件绑定的参数引用了上方的输入条目控件，Path 属性指明了输入条目控件的 Text 属性，这样输入条目控件的 Text 属性发生变化时，作为绑定参数传递

给相应的事件。

SQLitePage.xaml.cs 代码如下：

```
namespace MAUIDemo.Pages;

public partial class SQLitePage : ContentPage
{
    public SQLitePage()
    {
        InitializeComponent();
    }
    private async void OnTapped(object sender, EventArgs e)
    {
        await DisplayAlert("选择", ((Label)sender).Text, "确认");
    }
}
```

页面逻辑大部分通过视图模型进行控制，仅有数据模板中用于展示列表数据信息的 OnTapped()事件方法需要上述代码进行实现。这里直接使用异步调用 DisplayAlert()方法展示用户选中的信息，sender 传递对象强制转换为 Label 标签控件，获取其 Text 文本参数进行弹框显示。本地操作数据库的运行效果如图 5-2 所示。

图 5-2　本地操作数据库的运行效果

5.2　.NET Core Web API

5.2.1　.NET Core 最小化 API

.NET Core 最小化 API(minimal API)是从.NET 6 开始引入的。不同于传统 Program.cs 的静态 Main()方法，最小化 API 去除了 Main()方法，直接采用顶级语句进行编程。这样大大简化了主流程的书写过程，配置过程全部通过顶级语句完成。读者可以使用最少的代码行数搭建出自己心仪的网站。下面进行最小化 API 演示。

在 Visual Studio 中选择“解决方案”→“添加”→“新建项目”选项，在“添加新项目”窗

口中，选择项目类型为 ASP.NET Core Web API，如图 5-3 所示。

图 5-3　新建项目

单击“下一步”按钮，进入如图 5-4 所示的“配置新项目”窗口，在其中输入项目名称 WebAPI，单击“下一步”按钮。

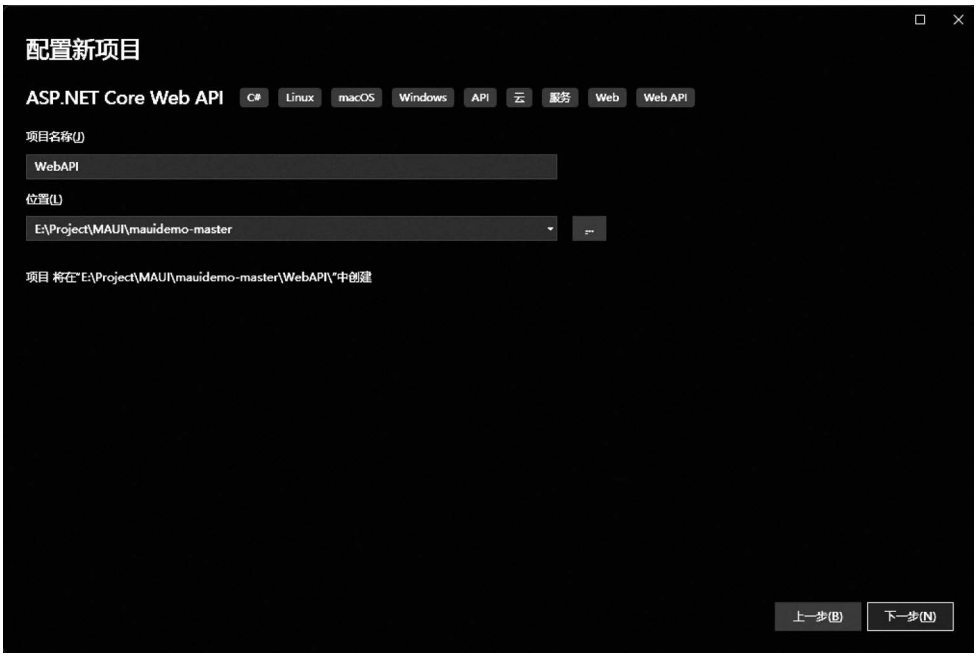

图 5-4　“配置新项目”窗口

配置其他信息，选择框架为“.NET 8.0(长期支持)”，根据需求配置 HTTPS 等选项，单击“创建”按钮。其他信息如图 5-5 所示。

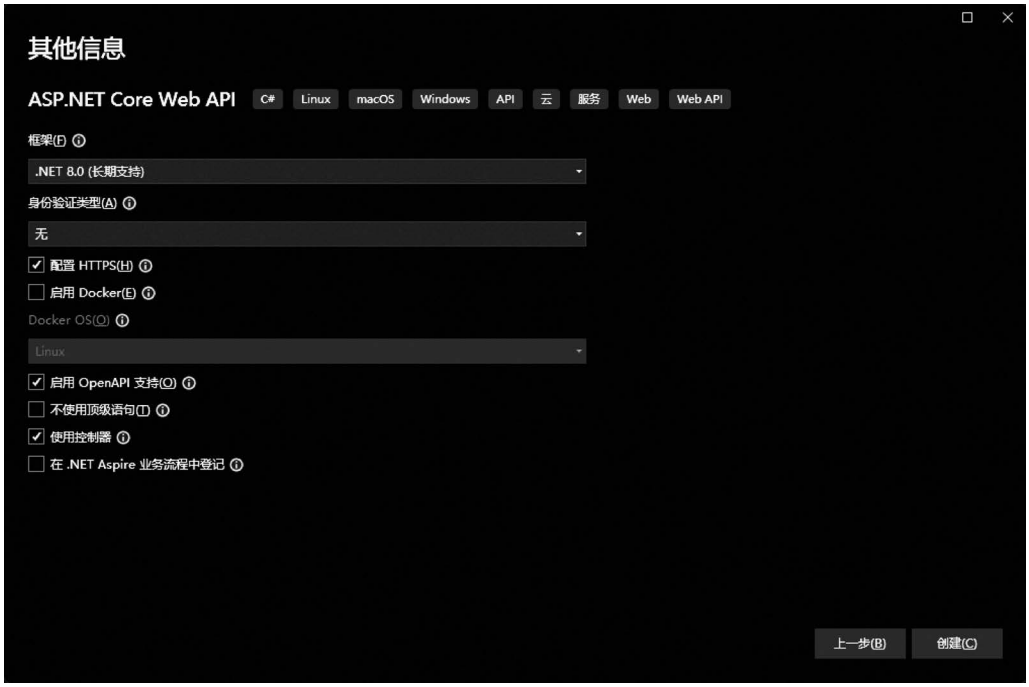

图 5-5　其他信息

创新项目后，项目默认结构如图 5-6 所示。

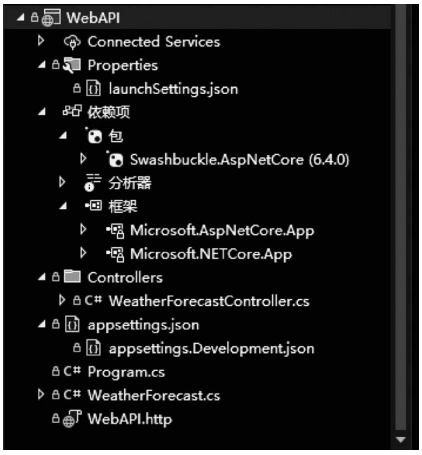

图 5-6　项目默认结构

运行项目，Swagger 启动界面如图 5-7 所示。

在浏览器中输入 URL，访问相应的控制器，运行效果如图 5-8 所示。

launchSettings 是启动配置文件。

【例 5-5】　最小化 ASP.NET Core Web API。

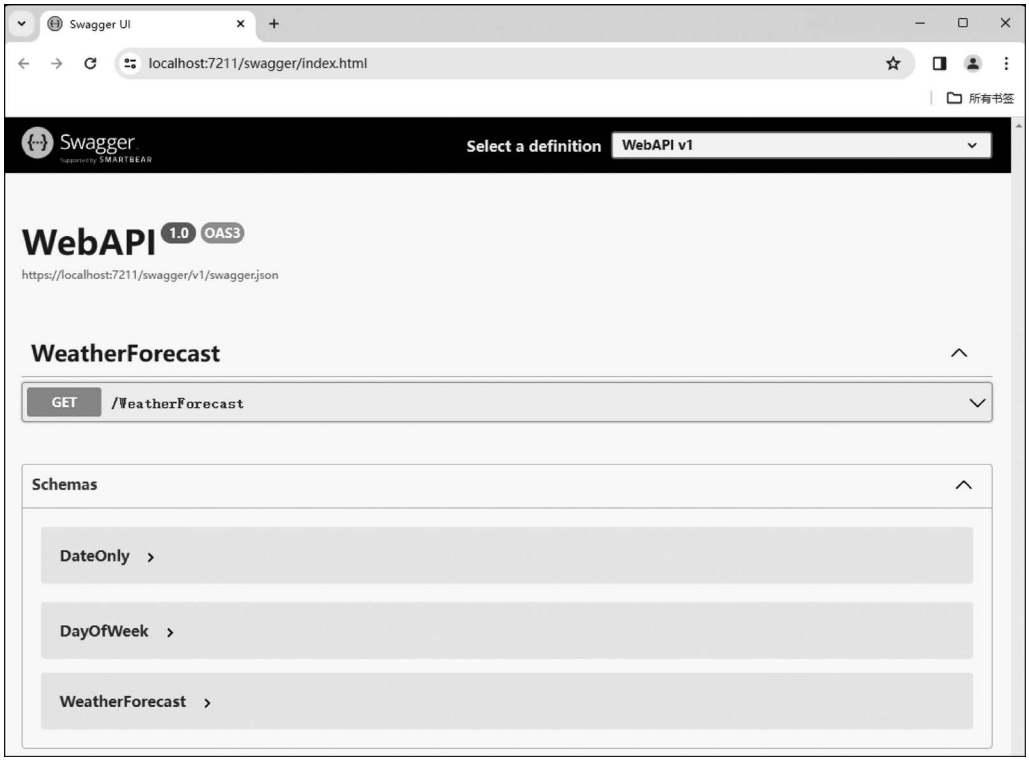

图 5-7　Swagger 启动界面

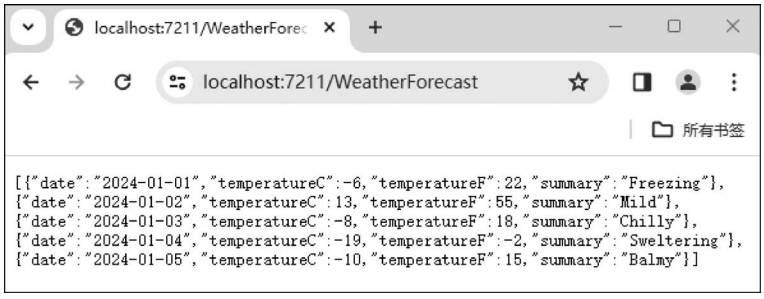

图 5-8　运行效果

launchSettings.json 代码如下：

```
{
  "$schema": "http://json.schemastore.org/launchsettings.json",
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:49874",
      "sslPort": 44386
    }
  },
  "profiles": {
```

```
    "http": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "http://localhost:5281",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "https": {
      "commandName": "Project",
      "dotnetRunMessages": true,
      "launchBrowser": true,
      "launchUrl": "swagger",
      "applicationUrl": "https://localhost:7211;http://localhost:5281",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "swagger",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

上述代码是 JSON 结构。

iisSettings 配置节。IIS(Internet Information Services,因特网信息服务)相关的配置：windowsAuthentication(Windows 验证选项)、anonymousAuthentication(匿名验证选项)、applicationUrl(应用访问路径)、sslPort SSL(Secure Socket Layer,安全套接字层端口)。

profiles 配置节。针对不同协议进行的配置。

http 配置节。commandName(命令名称)、dotnetRunMessages(是否运行消息)、launchBrowser(是否启动浏览器)、launchUrl(启动 URL)、environmentVariables(环境变量),环境变量的子节点是键值对。

https 配置节。与 http 类似,不同的是 applicationUrl(应用访问路径)的协议采用 https。

IIS Express 配置节。与 http 配置节和 https 配置节类似,不同的是 commandName(命令名称)为 IISExpress。

下面是 WeatherForecast(天气预报)数据结构代码。

WeatherForecast.cs 代码如下：

```
namespace WebAPI
{
```

```
    public class WeatherForecast
    {
        public DateOnly Date { get; set; }
        public int TemperatureC { get; set; }
        public int TemperatureF => 32 + (int)(TemperatureC / 0.5556);
        public string? Summary { get; set; }
    }
}
```

WeatherForecast 类的各个字段说明如下。

Date。当前日期。从.NET 6 开始,引入两种新的数据结构 DateOnly 和 TimeOnly,将原来的日期对象划分为日期部分和时间部分。

TemperatureC。摄氏温度。

TemperatureF。华氏温度。

Summary。信息汇总。

WeatherForecastController.cs 代码如下:

```
using Microsoft.AspNetCore.Mvc;

namespace WebAPI.Controllers
{
    [ApiController]
    [Route("[controller]")]
    public class WeatherForecastController : ControllerBase
    {
        private static readonly string[] Summaries = new[]
        {
            "Freezing", "Bracing", "Chilly", "Cool", "Mild", "Warm", "Balmy", "Hot",
"Sweltering", "Scorching"
        };
        private readonly ILogger<WeatherForecastController> _logger;
        public WeatherForecastController(ILogger<WeatherForecastController> logger)
        {
            _logger = logger;
        }
        [HttpGet(Name = "GetWeatherForecast")]
        public IEnumerable<WeatherForecast> Get()
        {
            return Enumerable.Range(1, 5).Select(index => new WeatherForecast
            {
                Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
                TemperatureC = Random.Shared.Next(-20, 55),
                Summary = Summaries[Random.Shared.Next(Summaries.Length)]
            })
            .ToArray();
        }
    }
}
```

针对 WeatherForecast 对象构造 WeatherForecastController 控制器。该控制器中的 Summaries 成员变量是只读的数据模板。构造方法通过依赖注入方式注入了 ILogger<WeatherForecastController>日志对象。Get()方法获取当天的天气描述数据。内部通过

Enumerable.Range(1,5)随机产生了5条相关数据。模拟产生WeatherForecast对象,将该对象的3个属性进行填充。填充时使用随机产生器Random.Shared.Next()随机生成。最后调用ToArray()方法转换为IEnumerable类型进行返回。WeatherForecastController控制器使用ApiController注解,代表RESTful API方式。接口Get()方法注解HttpGet的参数name指明了路由。访问路径需要在https://localhost:7211/后面追加GetWeatherForecast。

Program.cs代码如下:

```
var builder = WebApplication.CreateBuilder(args);
// Add services to the container.
builder.Services.AddControllers();
// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen();
var app = builder.Build();
// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
app.Run();
```

Program.cs是最小化API的入口类,省略了Main()方法,全部采用顶级语句。WebApplication.CreateBuilder()构造builder对象。对builder对象进行配置,builder对象配置逻辑如下。

builder.Services.AddControllers()。添加控制器中间件。

builder.Services.AddEndpointsApiExplorer()。引入端点路由。

builder.Services.AddSwaggerGen()。添加Swagger生成器。Swagger是一款API的文档管理工具。

builder.Build()。完成应用构建,返回应用对象。

app对象配置逻辑如下。

app.Environment.IsDevelopment()。判断是否为开发模式。

app.UseSwagger()。使用Swagger中间件。

app.UseSwaggerUI()。使用SwaggerUI(Swagger图形用户界面)中间件。

app.UseHttpsRedirection()。使用Https重定向中间件。完成HTTPS协议的重定向机制。

app.UseAuthorization()。使用授权机制中间件。通过策略和授权中间件的配置来决定是否允许请求继续执行。

app.MapControllers()。开启控制器映射。

app.Run()。运行应用。

appsettings是应用程序配置文件。Logging配置节配置了日志相关信息。Default默认记录日志等级Information信息级别。AllowedHosts允许访问主机配置项,这里采

用正则表达式表明全部允许。

appsettings.json 代码如下：

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

.NET 8 项目中还新增了 WebAPI.http 文件。花括号插值运算符引入 URL 变量 WebAPI_HostAddress。

WebAPI.http 代码如下：

```
@WebAPI_HostAddress = http://localhost:5281
GET {{WebAPI_HostAddress}}/weatherforecast/
Accept: application/json
###
```

至此介绍完了.NET Core 最小化 API 项目结构。MAUI 框架与其有很多相通之处，如依赖注入、配置、中间件等机制。

5.2.2 .NET Core Web API 管道模型

.NET Core Web API 管道模型是责任链设计模式的变体。责任链设计模式(Chain of Responsibility Pattern)属于行为设计模式，多个处理者构成链式结构，允许将请求沿着处理者链进行发送。每个处理者接收请求后，均可对请求进行处理，或者处理后将其传递给链上的下个处理者。管道模型可以提升系统可维护性、可测试性，避免代码冗余，能够并发执行，而且可以方便地增加和删除管道中的处理者，动态根据需求进行配置。.NET Core 内置了多个中间件，不同中间件完成不同的需求。.NET Core Web API 中间件管道模型如图 5-9 所示。

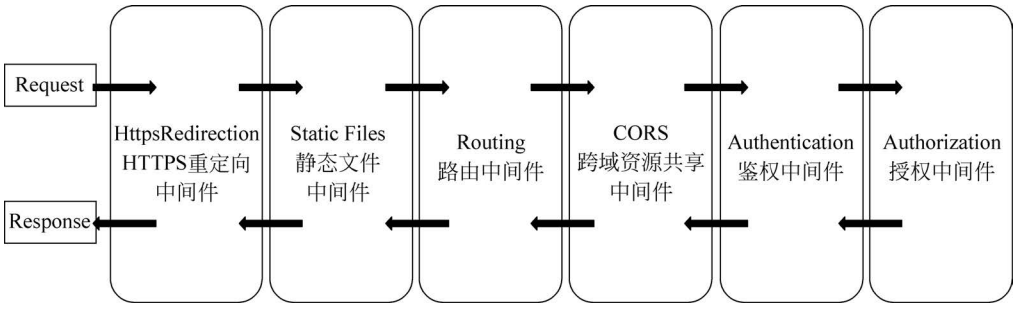

图 5-9 .NET Core WebAPI 中间件管道模型

.NET Core Web API 管道模型内置中间件如下。

app.UseAntiforgery()。使用防伪造中间件。防止恶意的第三方进行篡改。

app.UseAuthentication()。使用鉴权机制中间件。通过鉴权机制控制准入逻辑。

app.UseAuthorization()。使用授权机制中间件。通过策略和授权中间件的配置来决定是否允许请求继续执行。

app.UseCertificateForwarding()。使用证书转发中间件。寻找证书请求解码,更新HttpContext.Connection.ClientCertificate参数。

app.UseCookiePolicy()。使用Cookie策略中间件。Cookie是网站为了辨别用户身份,进行会话跟踪而存储在客户端上的数据。Cookie策略中间件用于修改Cookie相关的策略。

app.UseCors()。使用跨域中间件。配置策略,允许跨域请求。

app.UseDefaultFiles()。使用默认文件中间件。开启当前路径的默认映射。

app.UseDeveloperExceptionPage()。使用开发异常页面中间件。在开发环境中,使用该中间件针对异步或同步方法抛出的异常产生用户自定义友好的HTML页面。

app.UseDirectoryBrowser()。使用目录访问中间件。开启当前路径目录访问功能。

app.UseEndpoints()。使用终端中间件。执行匹配的端点。与路由机制中间件相分离,增强灵活性。

app.UseExceptionHandler()。使用异常处理中间件。抛出异常后进行日志记录,并重新执行其他中间件,如果响应已经开始则不执行其他中间件。

app.UseFileServer()。使用文件服务中间件。除目录服务器外,启用所有的静态文件。

app.UseForwardedHeaders()。使用转发头中间件。用于处理转发请求字段相匹配的请求。

app.UseHealthChecks()。使用健康检查中间件。

app.UseHostFiltering()。使用主机过滤中间件。过滤不满足条件的主机。

app.UseHsts()。使用严格的传输安全头部中间件。该中间件用于开启HTTP严格传输安全(Strict-Transport-Security,STS)相关的功能。

app.UseHttpLogging()。使用HTTP日志中间件。

app.UseHttpMethodOverride()。使用HTTP方法重写中间件。

app.UseHttpsRedirection()。使用HTTPS重定向中间件。完成HTTPS重定向机制。

app.UseOutputCache()。使用输出缓存中间件。

app.UseMiddleware()。使用中间件。

app.UseMvc()。使用MVC请求中间件。使用.NET Core模型、视图、控制器框架。

app.UseMvcWithDefaultRoute()。使用MVC请求中间件且为默认路由。

app.UsePathBase()。使用基路径中间件。

app.UseRateLimiter()。使用限流中间件。限流和削峰的目的是降低QPS(Queries-Per-Second,每秒查询率),延缓用户请求,防止高并发带来的服务器压力过大导致宕机。

app.UseRequestDecompression()。使用请求压缩中间件。

app.UseRequestLocalization()。使用请求本地化中间件。

app.UseRequestTimeouts()。使用请求超时中间件。

app. UseResponseCaching()。使用响应缓存中间件。

app. UseResponseCompression()。使用响应压缩中间件。

app. UseRewriter()。使用重定向中间件。

app. UseRouter()。使用路由机制中间件。特化 IRouter 实例。

app. UseRouting()。使用路由机制中间件。特化 IApplicationBuilder 实例。

app. UseSession()。使用会话中间件。Session 是利用对象存储特定用户会话所需的属性及配置信息的对象。该方法封装了会话相关的管理。

app. UseStaticFiles()。使用静态文件中间件。通过配置静态文件中间件,产生类似 IIS 中虚拟目录的效果。

app. UseStatusCodePages()。使用状态码中间件。

app. UseStatusCodePagesWithRedirects()。使用状态码转发中间件。

app. UseStatusCodePagesWithReExecute()。使用状态码执行中间件。

app. UseSwagger()。使用 Swagger 中间件。

app. UseSwaggerUI()。使用 SwaggerUI(Swagger 图形用户界面)中间件。

app. UseW3CLogging()。使用 W3C 日志中间件。

app. UseWebSockets()。使用 WebSocket 网络通信协议。WebSocket 协议是单 TCP 连接上的全双工通信协议,仅需一次握手即可创建持久性连接,简化客户端和服务器端之间的数据交互。

app. UseWelcomePage()。使用欢迎页面中间件。

app. UseWhen()。使用条件判断中间件。

5.2.3 EFCore

EntityFrameworkCore(EFCore)是轻量级的数据访问技术,可用作对象关系映射(Object Relational Mapping,ORM)程序,类似 Java 世界中大名鼎鼎的 Hibernate 框架。EntityFrameworkCore 最方便之处在于代码优先或数据优先模式能够自动完成对应部分的生成。如代码优先(Code First)方式,用户仅需写 C# 代码,PowerShell 执行命令或应用程序启动时会自动在数据库中生成相应的数据表结构。同理,数据优先(DataBase First)方式,用户仅需写 SQL 代码,应用层自动完成 C# 相关代码同步。

1. EFCore 框架的主要优点

易切换。跨数据库能力强大,用户仅需要修改相应的配置即可完成数据库之间的切换。

易使用。EFCore 提供强大的模型设计器功能,支持复杂表之间的关联关系,以及灵活的导航属性机制。

高效率。用户无须编写复杂的 SQL 语句,强大的 ORM 机制自动完成相关操作。对于复杂的查询和修改操作需要通过 EFCore 的 ExecuteSqlCommand 完成相关操作。

延迟查。延迟查询加载和懒加载机制提升性能。

2. EFCore 框架的主要缺点

性能低。与大部分 ORM 固有的缺点一样,针对大批量的数据操作效率低,尤其是复

杂查询的性能会下降。

启动慢。首次预热时启动慢，对象关系映射加载至内存后性能会提升。

NuGet 安装 EFCore 框架相关组件或者按照如下方式对项目文件进行配置。如果使用其他类型的数据库，进行相应的修改即可。

【例 5-6】 安装 EFCore 框架相关组件。

代码如下：

```
<ItemGroup>
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite" Version="8.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite.Core" Version="8.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite.NetTopologySuite"
Version="8.0.0" />
  <PackageReference Include="Microsoft.EntityFrameworkCore.Tools" Version="8.0.0">
</ItemGroup>
```

3. EFCore 常用命令

Add-Migration。增加迁移，每次迁移前需要先执行此命令。后面的参数指定迁移的操作名称。

Update-Database。更新数据库，包括数据库中的表和字段。

Script-Migration。生成从开始到最新迁移的 SQL 语句。

5.3　网络数据库

5.3.1　核心层

实际工程中网络数据库的开发大部分需要使用 ORM 框架。核心层主要完成数据操作逻辑的封装，为上层业务应用提供相应的接口。

【例 5-7】 网络数据库核心层。

EFCoreDbContext.cs 代码如下：

```
using MAUISDK.Core;
using MAUISDK.Models;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;

namespace MAUISDK.Repositories
{
    public class EFCoreDbContext : DbContext
    {
        public DbSet<MAUILearning> MAUILearning
        {
            get;
            set;
        }
        /// <summary>
        /// 构造方法
        /// </summary>
```

```
        /// <param name="options">EFCore 选项</param>
        public EFCoreDbContext(DbContextOptions<EFCoreDbContext> options)
            : base(options!)
        {
        }
        /// <summary>
        /// 模型构建
        /// </summary>
        /// <param name="builder">构建器</param>
        protected override void OnModelCreating(ModelBuilder builder)
        {
            var cfg = builder.Entity<MAUILearning>();
            cfg.Property(p => p.Learning).IsRequired();
        }
        /// <summary>
        /// 构建
        /// </summary>
        /// <param name="builder">构建器</param>
      protected override void OnConfiguring(DbContextOptionsBuilder builder)
        {
            string connection = GlobalValues.ConnectionStr;
            builder.UseSqlite(connection)
                .LogTo(Console.WriteLine, LogLevel.Information);
        }
    }
}
```

定义 EFCoreDbContext 是 EFCore 框架数据库上下文类，继承 DbContext 数据库上下文类。成员变量 MAUILearning 是 DbSet 类型，对应数据库中的一张表。重写 OnModelCreating()方法，构造 MAUILearning 数据表以及内部字段。重写 OnConfiguring()方法，构建数据库连接配置信息。调用构建器的 UseSqlite()方法，配置 Sqlite 数据库连接字符串信息。

IEFCoreRepository.cs 代码如下：

```
/// <summary>
/// EFCore 仓储接口
/// </summary>
/// <typeparam name="T">数据泛型</typeparam>
/// <typeparam name="Context">仓储上下文</typeparam>
public interface IEFCoreRepository<T, Context> : IDisposable
        where T : class
        where Context : DbContext
{
}
```

IEFCoreRepository 定义了仓储接口，泛型参数 T 表示数据表，泛型参数 Context 表示数据上下文。

EFCoreRepository.cs 代码结构如下：

```
using Microsoft.EntityFrameworkCore;
using System.Data;
using System.Linq.Expressions;

namespace MAUISDK.Repositories
```

```
{
    /// <summary>
    /// EFCore 仓储
    /// </summary>
    /// <typeparam name="T">数据泛型</typeparam>
    /// <typeparam name="Context">仓储上下文</typeparam>
    public class EFCoreRepository<T, Context> : IEFCoreRepository<T, Context>
            where T : class
            where Context : DbContext
    {
        #region 构造析构
        // 此处省略相关代码
        #endregion
        #region 事务操作
        // 此处省略相关代码
        #endregion
        #region 同步版本
        // 此处省略相关代码
        #endregion
        #region 异步版本
        // 此处省略相关代码
        #endregion
    }
}
```

EFCoreRepository EFCore 仓储继承了之前定义的 IEFCoreRepository 接口，涉及 EFCoreRepository 的代码非常多，为了从宏观上进行把握，分为四大类代码，分别用 C# 区域运算符进行划分。构造析构类方法完成 EFCore 仓储的构造析构操作，事务操作完成 EFCore 仓储的保障数据业务完整性和一致性的事务类操作，同步版本完成 EFCore 仓储涉及数据库同步类的相关操作，异步版本完成 EFCore 仓储涉及数据库异步类的相关操作。

EFCoreRepository.cs 构造析构部分代码如下：

```
#region 构造析构
private Context dbContext;
private readonly DbSet<T> dbSet;
/// <summary>
/// 构造方法
/// </summary>
/// <param name="dbContext">数据库上下文</param>
public EFCoreRepository(Context dbContext)
{
    this.dbContext = dbContext;
    dbSet = dbContext.Set<T>();
}
/// <summary>
/// 释放资源
/// </summary>
public void Dispose()
{
    if (dbContext != null)
        dbContext.Dispose();
```

```
20  }
21  #endregion
```

EFCore 仓储的构造部分初始化数据表，析构部分释放数据表相关资源。

EFCoreRepository.cs 事务操作部分代码如下：

```
1   /// <summary>
2   /// 开始事务
3   /// </summary>
4   /// <param name="isolationLevel">隔离级别</param>
5   public void BeginTransaction(IsolationLevel isolationLevel = IsolationLevel.Unspecified)
6   {
7       if (dbContext.Database.CurrentTransaction == null)
8       {
9           dbContext.Database.BeginTransaction(isolationLevel);
10      }
11  }
12  /// <summary>
13  /// 提交事务
14  /// </summary>
15  public void Commit()
16  {
17      var transaction = dbContext.Database.CurrentTransaction;
18      if (transaction != null)
19      {
20          try
21          {
22              transaction.Commit();
23          }
24          catch (Exception)
25          {
26              transaction.Rollback();
27              throw;
28          }
29      }
30  }
31  /// <summary>
32  /// 回滚事务
33  /// </summary>
34  public void Rollback()
35  {
36      if (dbContext.Database.CurrentTransaction != null)
37      {
38          dbContext.Database.CurrentTransaction.Rollback();
39      }
40  }
```

EFCore 仓储的事务操作部分封装了下述 3 个核心方法。

BeginTransaction()。开始事务。根据事务的隔离级别参数 IsolationLevel 进行配置。

Commit()。提交事务。获取当前事务对象，如果提交成功则不进行回滚；如果提交不成功则回滚后抛出异常供上层程序处理。

Rollback()。回滚事务。手动调用此方法完成当前事务的回滚操作。

EFCore仓储的同步和异步操作包含很多方法。同步操作指所有的逻辑都执行完，才最终返回给用户，可能导致用户在线等待的时间过长，造成一种阻塞或死机的体验。异步操作是将用户请求放入消息队列后，直接返回，一个任务不需要等待另一个任务的完成就能够继续执行，这种方式不会造成任务阻塞，可以提高系统的响应速度。下面仅介绍几个常用的方法，其余的读者自行查看代码，根据需要也可以自行进行扩充和完善。

根据条件分页查询代码如下：

```
1   /// <summary>
2   /// 根据条件分页查询
3   /// </summary>
4   /// <typeparam name="TOrder">排序约束</typeparam>
5   /// <param name="where">过滤条件</param>
6   /// <param name="order">排序条件</param>
7   /// <param name="pageIndex">当前页码</param>
8   /// <param name="pageSize">每页记录条数</param>
9   /// <param name="count">返回总条数</param>
10  /// <param name="isDesc">是否倒序</param>
11  /// <returns>查询结果集</returns>
12  public IEnumerable<T> Where<TOrder>(Func<T, bool> @where, Func<T, TOrder>
    order, int pageIndex, int pageSize, out int count, bool isDesc = false)
13  {
14      count = Count();
15      if (isDesc)
16      {
17      return dbSet.Where(@where).OrderByDescending(order).Skip((pageIndex - 1) *
    pageSize).Take(pageSize);
18      }
19      else
20      {
21          return dbSet.Where(@where).OrderBy(order).Skip((pageIndex - 1) *
    pageSize).Take(pageSize);
22      }
23  }
```

根据条件分页查询是数据库的常用操作之一。泛型参数TOrder是排序要求，函数委托参数@where是过滤条件，函数委托参数order是排序规则，pageIndex为页码，pageSize为每页大小，count作为输出参数返回总条数，isDesc表示是否进行降序。通过链式调用方式实现，Skip()方法跳过指定记录，Take()方法获取指定记录数，如果不足则返回剩余所有。返回的结果集是IEnumerable可枚举类型。

插入实体对象代码如下：

```
1   /// <summary>
2   /// 插入实体对象
3   /// </summary>
4   /// <param name="entity">实体对象</param>
5   /// <param name="save">是否保存</param>
6   /// <returns>实体对象</returns>
7   public async Task<T> AddAsync(T entity, bool save = true)
8   {
9       await dbSet.AddAsync(entity);
```

```
    if (save)
    {
        await this.SaveChangesAsync();
    }
    return entity;
}
```

数据库的插入操作是数据库的常用操作之一。通过 save 参数判断是否需要同步至实体数据库。

删除实体对象代码如下：

```
/// <summary>
/// 删除实体对象
/// </summary>
/// <param name="entity">实体对象</param>
/// <param name="save">是否保存</param>
public async Task DeleteAsync(T entity, bool save = true)
{
    dbSet.Remove(entity);
    if (save)
    {
        await this.SaveChangesAsync();
    }
}
```

数据库的删除操作是数据库的常用操作之一。通过 save 参数判断是否需要同步至实体数据库。

根据编号查询对象代码如下：

```
/// <summary>
/// 根据编号查询对象
/// </summary>
/// <typeparam name="Ttype">字段类型</typeparam>
/// <param name="id">编号</param>
/// <returns></returns>
public async Task<T> GetByIdAsync<Ttype>(Ttype id)
{
    return await dbSet.FindAsync(id);
}
```

数据库的查询操作是数据库的常用操作之一。上述代码实现了基于主键的查询。

修改对象代码如下：

```
/// <summary>
/// 修改对象
/// </summary>
/// <param name="entity">实体对象</param>
/// <param name="save">是否保存</param>
public async Task UpdateAsync(T entity, bool save = true)
{
    dbSet.Update(entity);
    if (save)
    {
        await this.SaveChangesAsync();
    }
}
```

数据库的修改操作是数据库的常用操作之一。通过 save 参数判断是否需要同步至实体数据库。

5.3.2　服务层

核心层完成了底层数据交互的逻辑封装，服务层则进行业务逻辑的封装。首先定义 RESTful API 操作风格的接口。

【例 5-8】　网络数据库服务层。

IRestService.cs 代码如下：

```
using MAUISDK.Models;

namespace MAUISDK.Interfaces
{
    public interface IRestService<T> where T : BaseModel
    {
        Task<List<T>> QueryAsync();
        Task SaveAsync(T Item, bool newData = true);
        Task RemoveAsync(long Id);
    }
}
```

IRestService 接口的泛型参数继承了 BaseModel 基本模型类，需要符合 BaseModel 基本模型类的规范。下面是其中的 3 个方法。

QueryAsync()。数据查询接口。

SaveAsync()。数据保存接口。参数 Item 是待保存的对象，参数 newData 指明是否为新增数据，用于区分新增操作还是修改操作。

RemoveAsync()。数据删除接口。参数 Id 对应数据对象的主键。

RestService.cs 代码如下：

```
using MAUISDK.Interfaces;
using MAUISDK.Models;
using System.Text.Json;
using System.Text;
using MAUISDK.Core;

namespace MAUISDK.Services
{
    public class RestService<T> : IRestService<T> where T : BaseModel
    {
        private HttpClient client;
        private JsonSerializerOptions serializerOptions;
        private IHttpsClientHandlerService httpsClientHandlerService;
        public List<T> ItemList;
  public RestService(IHttpsClientHandlerService httpsClientHandlerService)
        {
#if DEBUG
            this.httpsClientHandlerService = httpsClientHandlerService;
            HttpMessageHandler handler =
httpsClientHandlerService.GetPlatformMessageHandler();
            if (handler != null)
```

```
22                  client = new(handler);
23              else
24                  client = new();
25  #else
26              client = new HttpClient();
27  #endif
28              serializerOptions = new JsonSerializerOptions
29              {
30                  PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
31                  WriteIndented = true
32              };
33          }
34          public async Task<List<T>> QueryAsync()
35          {
36              ItemList = [];
37              Uri uri = new(GlobalValues.RestUrl);
38              try
39              {
40                  HttpResponseMessage response = await client.GetAsync(uri);
41                  if (response.IsSuccessStatusCode)
42                  {
43                    string content = await response.Content.ReadAsStringAsync();
44                      ItemList = JsonSerializer.Deserialize<List<T>>(content,
    serializerOptions)!;
45                  }
46              }
47              catch
48              {
49              }
50              return ItemList;
51          }
52
53          public async Task SaveAsync(T Item, bool newData = true)
54          {
55              Uri uri = new(GlobalValues.RestUrl);
56              try
57              {
58            string json = JsonSerializer.Serialize<T>(Item, serializerOptions);
59          StringContent content = new(json, Encoding.UTF8, "application/json");
60                  HttpResponseMessage response;
61                  if (newData)
62                  {
63                      response = await client.PostAsync(uri, content);
64                  }
65                  else
66                  {
67                      response = await client.PutAsync(uri, content);
68                  }
69              }
70              catch
71              {
72              }
73          }
74          public async Task RemoveAsync(long Id)
75          {
76              Uri uri = new(String.Format(GlobalValues.RemoveItem, Id));
```

```
77              try
78              {
79                  HttpResponseMessage response = await client.DeleteAsync(uri);
80              }
81              catch
82              {
83              }
84          }
85      }
86  }
```

RestService 继承了 IRestService 接口。构造方法完成 IHttpsClientHandlerService 对象实例的依赖注入，并构造出 HttpClient 与服务器交互的对象，同时完成了 JsonSerializerOptions JSON 序列化对象选项配置。PropertyNamingPolicy JSON 属性名称策略指定为 CamelCase(骆驼拼写法)，根据单词的大小写拼写复合词的方法。骆驼拼写法又分为小骆驼拼写法和大骆驼拼写法。小骆驼拼写法是指第一个词的首字母小写，后面单词首字母大写，单词除首字母后的字母均为小写。大骆驼拼写法是指第一个词的首字母大写，后面单词首字母大写，单词除首字母后的字母均为小写。WriteIndented 参数指明了 JSON 是否采用格式缩进。

QueryAsync()。数据查询方法。通过 HttpClient 对象获取结果后，因为服务器的控制器返回的是 JSON 格式，所以需要调用 JsonSerializer 类的 Deserialize()反序列化方法转换为对象列表。

SaveAsync()。数据保存方法。提交参数时，根据后台的[FromForm] 注解，配套使用 StringContent 对象进行提交 POST 请求。

RemoveAsync()。数据删除方法。直接调用 HttpClient 对象的 DeleteAsync()方法。

5.3.3　控制层

控制层主要由控制器实现对服务器的封装。配置了路由参数 [Route("api/[controller]")]，通过控制器类型进行路由匹配。

【例 5-9】 网络数据库控制层。

MAUILearningController.cs 代码如下：

```
1   [ApiController]
2   [Route("api/[controller]")]
3   public class MAUILearningController : ControllerBase
4   {
5       private readonly IEFCoreRepository<MAUILearning, EFCoreDbContext> repository;
6       public MAUILearningController(IEFCoreRepository<MAUILearning, EFCoreDbContext>
    repository)
7       {
8           this.repository = repository;
9       }
10      [HttpGet]
11      public IActionResult List()
12      {
13          return Ok(repository.GetAll());
14      }
```

```
15      [HttpPost]
16      public IActionResult Create([FromBody] MAUILearning Item)
17      {
18          try
19          {
20              if (Item == null || !ModelState.IsValid)
21              {
22                  return BadRequest("InValid");
23              }
24              repository.AddAsync(Item);
25          }
26          catch (Exception)
27          {
28              return BadRequest("Error");
29          }
30          return Ok(Item);
31      }
32      [HttpPut]
33      public IActionResult Update([FromBody] MAUILearning Item)
34      {
35          try
36          {
37              if (Item == null || !ModelState.IsValid)
38              {
39                  return BadRequest("InValid");
40              }
41              repository.UpdateAsync(Item);
42          }
43          catch (Exception)
44          {
45              return BadRequest("Error");
46          }
47          return NoContent();
48      }
49      [HttpDelete("{Id}")]
50      public IActionResult Remove(long Id)
51      {
52          try
53          {
54              repository.DeleteAsync(Id);
55          }
56          catch (Exception)
57          {
58              return BadRequest("Error");
59          }
60          return NoContent();
61      }
62  }
```

控制器的构造方法通过依赖注入机制完成仓储对象的初始化，增、删、查、改操作均是通过调用之前仓储对象封装的方法完成。控制器主要完成异常检测功能，如果抛出异常，返回 BadRequest 类型。IActionResult 是控制器方法执行后返回的结果类型，Ok()方法返回的结果类型是正常服务返回的类型，BadRequest()方法返回的结果类型是异常服务返回的类型，NoContent()方法返回的结果类型是无内容类型。

客户端界面层定义 WebPage.xaml 代码如下：

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
< ContentPage
    x:Class = "MAUIDemo.Pages.WebPage"
    xmlns = "http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x = "http://schemas.microsoft.com/winfx/2009/xaml"
    Title = "网络访问">
    < ContentPage.Resources >
        < Style TargetType = "Entry">
            < Setter Property = "HorizontalOptions" Value = "Center" />
            < Setter Property = "HorizontalTextAlignment" Value = "Center" />
            < Setter Property = "WidthRequest" Value = "300" />
            < Setter Property = "HeightRequest" Value = "40" />
        </Style >
        < Style TargetType = "Button">
            < Setter Property = "HorizontalOptions" Value = "Center" />
            < Setter Property = "WidthRequest" Value = "300" />
            < Setter Property = "HeightRequest" Value = "40" />
            < Setter Property = "BackgroundColor" Value = "＃512BD4" />
        </Style >
    </ContentPage.Resources >
    < VerticalStackLayout >
        < Entry x:Name = "entry" Placeholder = "输入信息" />
        < Button
            Command = "{Binding OnSaveAsync}"
          CommandParameter = "{Binding Source = {x:Reference entry}, Path = Text}"
            Text = "保存" />
        < Button
            Command = "{Binding OnRemoveAsync}"
          CommandParameter = "{Binding Source = {x:Reference entry}, Path = Text}"
            Text = "删除" />
        < Button Command = "{Binding OnFlushAsync}" Text = "刷新" />
        < CollectionView
            x:Name = "collectionView"
            HorizontalOptions = "Center"
            SelectionMode = "Single"
            WidthRequest = "300">
            < CollectionView.ItemTemplate >
                < DataTemplate >
                    < Grid ColumnDefinitions = "Auto">
                        < Label
                            HorizontalOptions = "Center"
                            HorizontalTextAlignment = "Center"
                            Text = "{Binding Learning}"
                            VerticalTextAlignment = "Center" />
                    </Grid >
                </DataTemplate >
            </CollectionView.ItemTemplate >
        </CollectionView >
    </VerticalStackLayout >
</ContentPage >
```

资源定义部分定义了 Entry 和 Button 控件的页面级通用样式。主体部分通过 VerticalStackLayout 垂直布局定义了输入条目、保存和删除按钮。CollectionView 控件用于展示数据信息。DataTemplate 内部使用 Grid 布局展示标签，标签绑定了数据的

Learning 属性。

WebPage. xaml. cs 代码如下：

```
using MAUISDK.Interfaces;
using MAUISDK.Models;
using System.Collections.ObjectModel;
using System.Windows.Input;

namespace MAUIDemo.Pages;

public partial class WebPage : ContentPage
{
    private readonly IRestService<MAUILearning> restService;
    public ObservableCollection<MAUILearning> MAUILearnings
    {
        get;
        set;
    }
    public ICommand OnSaveAsync
    {
        get;
        set;
    }
    public ICommand OnRemoveAsync
    {
        get;
        set;
    }
    public ICommand OnFlushAsync
    {
        get;
        set;
    }
    public WebPage(IRestService<MAUILearning> restService)
    {
        this.restService = restService;
        InitializeComponent();
        OnSaveAsync = new Command(async (arg) =>
        {
            MAUILearning Item = new()
            {
                Learning = entry.Text.Trim()
            };
            await restService.SaveAsync(Item);
            await DisplayAlert("信息", "保存成功", "确认");
            OnFlushAsync.Execute(null);
        });
        OnRemoveAsync = new Command(async (arg) =>
        {
            long Id = long.Parse(arg.ToString());
            await restService.RemoveAsync(Id);
            await DisplayAlert("信息", "删除" + Id.ToString() + "成功", "确认");
            OnFlushAsync.Execute(null);
        });
        OnFlushAsync = new Command(async (arg) =>
```

```
53          {
54              var Items = await restService.QueryAsync();
55              MainThread.BeginInvokeOnMainThread(() =>
56              {
57                  MAUILearnings = new();
58                  foreach (var Item in Items)
59                  {
60                      MAUILearnings.Add(Item);
61                  }
62              });
63              OnPropertyChanged("MAUILearnings");
64              collectionView.ItemsSource = MAUILearnings;
65          });
66          BindingContext = this;
67      }
68      protected override void OnAppearing()
69      {
70          base.OnAppearing();
71          OnFlushAsync.Execute(null);
72      }
73  }
```

与本地数据库界面逻辑代码实现完全类似，刷新操作也使用了 MAUI 涉及界面多线程更新数据时保障线程安全的技巧。网络操作数据库的示例效果如图 5-10 所示。

图 5-10　网络操作数据库的示例效果

本示例以极简的方式完成了 MAUI 网络数据库编程相关的全流程操作，起到演示作用。作为抛砖引玉，相信读者能够开发出比 MAUI 网络数据库示例更为复杂的业务场景。