第3章 ASP.NET Core开发环境

Visual Studio是微软公司推出一套支持Visual Basic、Visual C#和Visual C++语言的集成开发环境(IDE),用于生成ASP.NET Core Web应用程序、XML Web Services、桌面应用程序和移动应用程序。本章将主要介绍Visual Studio 2019管理软件包工具NuGet、JavaScript和CSS的概念和部署以及日志管理。

本章主要学习目标如下:

- 掌握NuGet工具的应用。
- 掌握JavaScript文件的部署。
- 掌握CSS文件的部署。
- 掌握日志文件的配置。

3.1 管理软件包

3.1.1 NuGet工具

ASP.NET Core框架下开发的项目有时需要依赖不同类型的软件包。为了管理项目中的开源软件包,Visual Studio 2019提供了图形工具——NuGet。NuGet是用于微软.NET开发平台的软件包管理器,是Visual Studio的扩展。NuGet包是具有.nupkg扩展的单个ZIP文件,此扩展包含编译代码(Dll)、与该代码相关的其他文件以及描述性清单(包含包版本号等信息)。在使用Visual Studio 2019进行项目的开发时,NuGet可以使在项目中添加、移除和更新引用的工作变得更加快捷方便。通过NuGet可以很容易地访问到其他开发者发布的软件包,也可以创建、分享或者发布自己的包到NuGet。微软的EntityFramework、ASP.NET MVC或第三方软件包(Json.NET、NUnit等)都托管到NuGet上,该工具主要包含以下特性。

(1) NuGet提供了专用于托管中心的nuget.org存储库。

NuGet负责在nuget.org上维护中央存储库的各类包,还支持在云Azure DevOps上、私有网络中或者甚至直接在本地文件系统以私密方式托管包,这些包可以被ASP.NET Core开发者使用。

(2) NuGet提供工具。

NuGet为开发人员提供创建、发布和使用包所需的工具,包括dotnet CLI、nuget.exe CLI、包管理器控制台、包管理器UI、管理NuGet UI、MSBuild。这些工具用于在Visual Studio项目中安装和管理包。

（3）管理依赖项。

大部分 NuGet 的用途就是代表系统管理依赖关系树或“关系图”。开发者仅需要关注在项目中如何直接使用包。如果这些包本身还使用了其他包，NuGet 还负责对所有这些下层依赖项进行管理。

（4）跟踪引用和还原包。

NuGet 使用引用列表维护系统所依赖的包，包括顶层和下层的依赖关系。当将某个主机中的包安装到系统中时，NuGet 都将在引用列表中记录包的标识符和版本号。

为了获取更简洁的开发环境并减少存储库容量，NuGet 提供还原所有引用程序包的方法。NuGet 提供可使用包的两种包管理格式。

① PackageReference。

PackageReference 始终由. NET Core 项目使用，使用 PackageReference 结点的包引用可直接在项目文件中管理 NuGet 依赖项，即无须再提供单独的 packages. config 文件。

② packages. config。

一种 XML 文件，包含一个或多个< package >元素，每个元素用于一个引用。用于维护项目中所有依赖项的简单列表，包括其他已安装包的依赖项。

任何系统中所用的包管理格式取决于系统类型以及 NuGet 版本。若要确认当前使用的格式，只需在安装第一个包后在系统根目录中查找 packages. config。如果未找到该文件，则在该系统文件中查找< PackageReference >元素。

（5）NuGet 管理包缓存和全局包文件夹。

包缓存用于避免重复下载已安装的包，全局包文件夹允许多个项目共享同一个已安装的包，使安装和重新安装过程更为快捷。

3.1.2 NuGet 管理软件包

从 Visual Studio 2017 开始，NuGet 和 NuGet 包管理器与任何. NET 相关的应用一起自动安装。通过在 Visual Studio 安装程序中选择“单个组件”→“代码工具”→“NuGet 包管理器”命令就可以单独安装。

【例 3-1】 查找和安装包。

① 打开 Visual Studio 2019，选择“创建新项目”选项，如图 3-1 所示。

② 在“创建新项目”对话框中选择“ASP. NET Core Web 应用程序”选项，单击“下一步”按钮，如图 3-2 所示。

③ 在“配置新项目”对话框中输入项目名称 WebApplicationDemo 并选择安装路径，然后单击“创建”按钮，如图 3-3 所示。

④ 在“创建新的 ASP. NET Core Web 应用程序”对话框中选择“Web 应用程序”选项，然后单击“创建”按钮，如图 3-4 所示。

⑤ 在打开的“解决方案资源管理器”中，右击 WebApplicationDemo 项目，在弹出的快捷菜单中选择“管理 NuGet 程序包”命令，如图 3-5 所示。

⑥ 选择“浏览”选项卡，按当前所选来源的下载量多少显示包，或者使用搜索框搜索特定包，再从列表中选择所需要的包，如图 3-6 所示。

⑦ 从下拉列表中选择所需的版本，然后单击“安装”按钮。Visual Studio 2019 随即将

图 3-1　创建新项目

图 3-2　创建 ASP. NET Core Web 应用程序

包及其依赖项安装到项目中。安装完成后，添加的包将显示在“已安装” 选项卡上，同时也出现在“解决方案资源管理器”的“引用”结点中。这表明可以使用 using 语句在项目中引用了，如图 3-7 所示。

图 3-3　配置新项目

图 3-4　创建 Web 应用程序

【例 3-2】 更新包。

① 在“解决方案资源管理器”中，右击 WebApplicationDemo 项目，在弹出的快捷菜单中选择“管理 NuGet 程序包”命令。

② 选择“更新”选项卡，查看所选包源中可用更新的包。选中“包括预发行版”复选框，以便在更新列表中包含预发布版本的包，如图 3-8 所示。

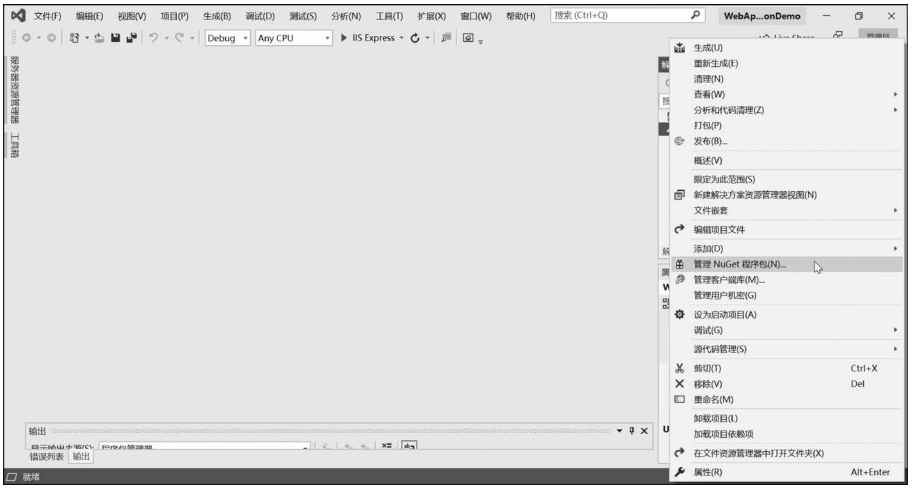

图 3-5　管理 NuGet 程序包

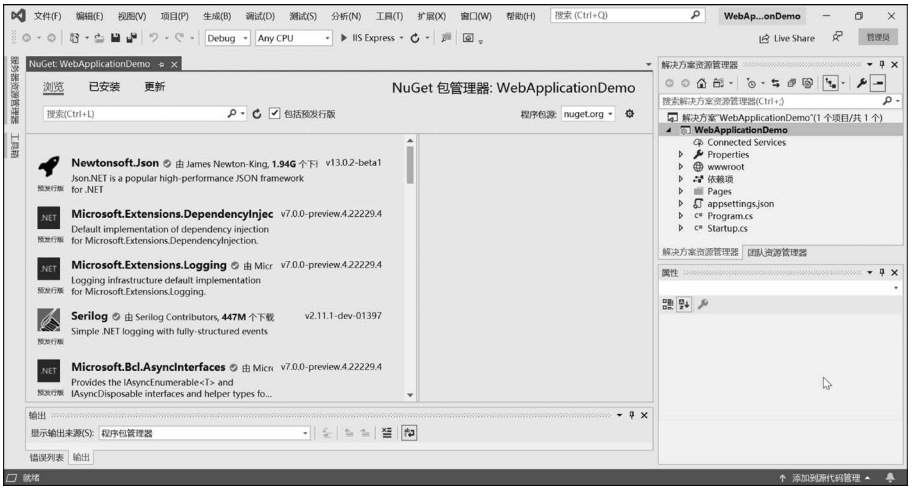

图 3-6　浏览包

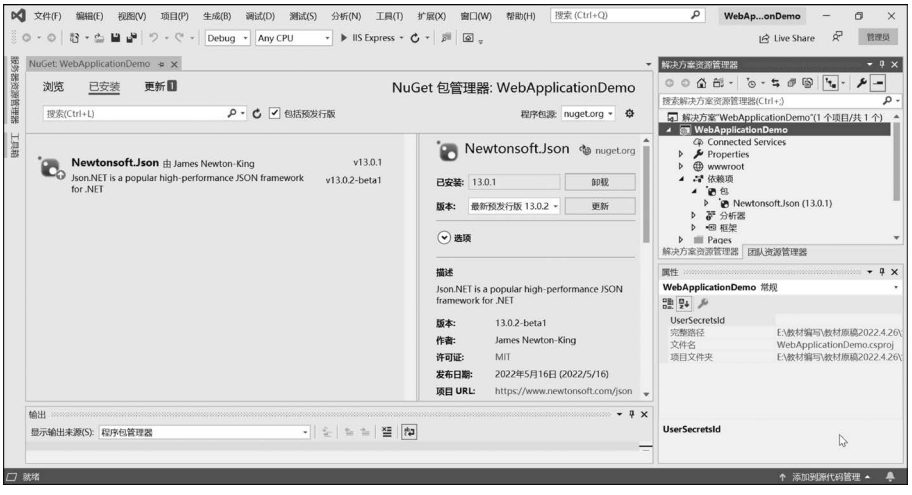

图 3-7　安装包

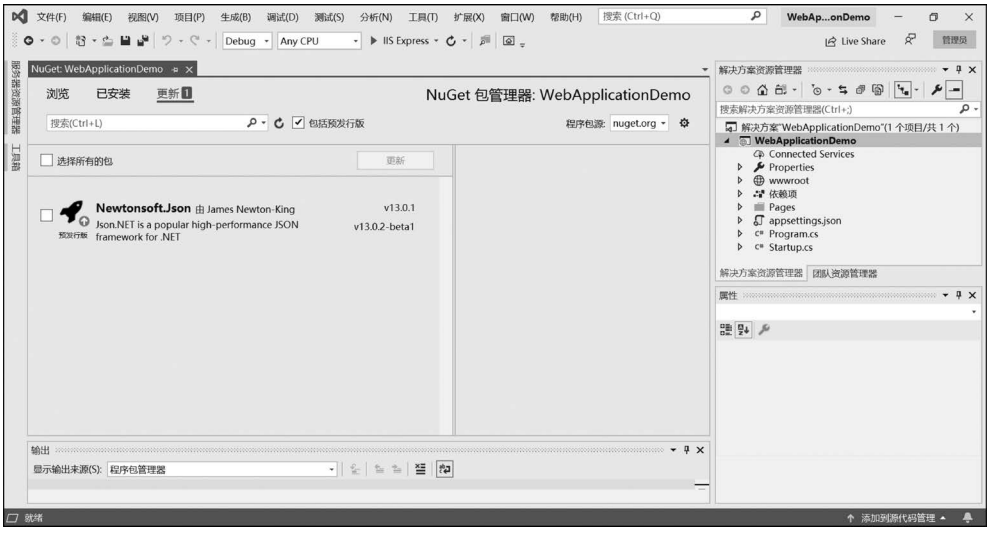

图 3-8　浏览更新包

③ 选择要更新的包，从右侧的下拉列表中选择所需的版本，然后单击“更新”按钮，如图 3-9 所示。

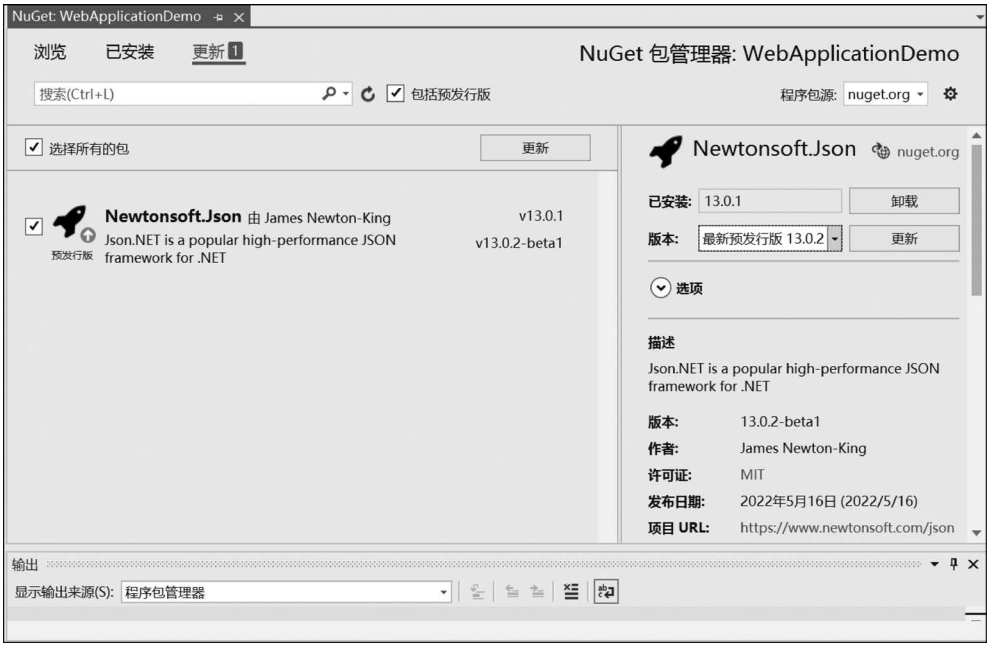

图 3-9　更新软件包

包更新过程中可能存在以下情况。

- 对于某些包，“更新”按钮处于禁用状态，并显示是“由 SDK 隐式引用”（或 AutoReferenced）的消息，表明该包是较大框架或 SDK 的一部分，不能单独更新。
- 若要将多个包更新到其最新版本，则在列表中选中，然后单击列表上方的“更新”按钮。

- 支持从“已安装”选项卡更新单个包。在这种情况下包的详细信息包括“版本”选择和“更新”按钮。

【例 3-3】 管理包源。

① 若要更改 Visual Studio 2019 从中获取包的源，可从源选择器中选择一个源，如图 3-10 所示。

图 3-10　获取程序包源

② 在图 3-10 中单击“程序包源”右侧的设置图标，或使用菜单栏中的“工具”→“选项”命令打开“选项”对话框，选择“NuGet 包管理器”→“程序包源”命令，如图 3-11 所示。

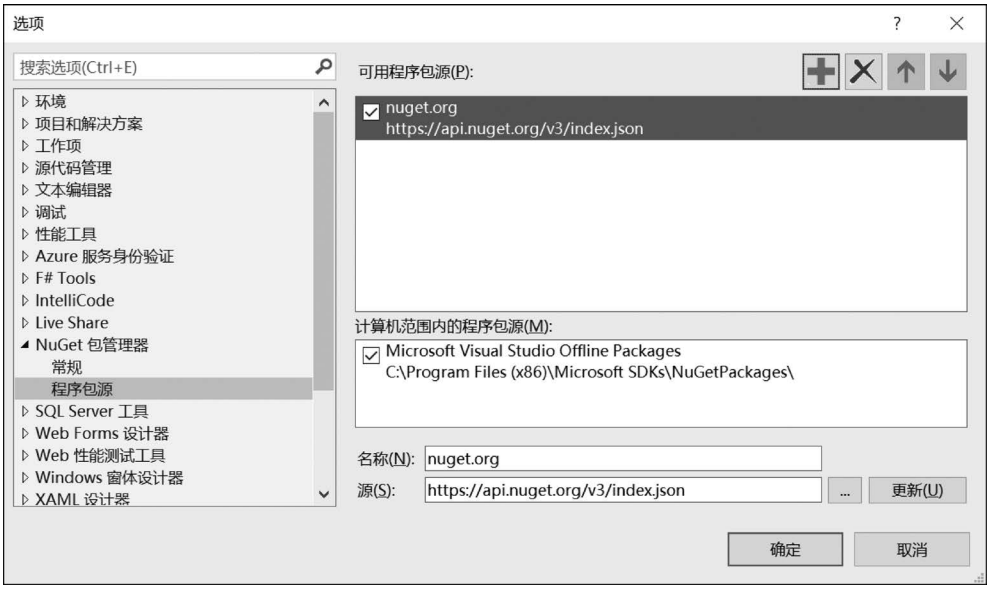

图 3-11　设置程序包源

③ 要添加源，单击＋按钮，编辑名称，在“源”控件中输入 URL 或路径，然后单击“更新”按钮。此时选择器下拉列表中就会显示刚添加的源。

若要更改包源，选中该包，在“名称” 和“源” 文本框中进行编辑，然后单击“更新”按钮。

若要禁用包源，则清除列表中名称左侧的复选框即可。

若要删除包源，首先选中该包，然后单击 ✕ 按钮完成删除。

Visual Studio 2019 会忽略包源的顺序，使用向上或向下箭头按钮不会更改包源的优先级顺序。

【例 3-4】 卸载包。

① 在“解决方案资源管理器”中右击 WebApplicationDemo 项目，在弹出的快捷菜单中选择“管理 NuGet 程序包”命令。

② 选择“已安装”选项卡，查看所选包源中可用更新的包。选中“包括预发行版”复选框，以便在更新列表中包含预发布版本的包。

③ 选择要卸载的包，从右侧的下拉列表中选择所需的版本，然后单击“卸载”按钮，如图 3-12 所示。

图 3-12　卸载包

包卸载过程中存在以下情况。

- 如果选中“删除依赖项”复选框，当此包未在项目中的其他位置引用时则删除所有依赖包。
- 如果选中“在存在依赖项时仍强制卸载”复选框，则即使在项目中引用了该包也会被卸载。此复选框通常与“删除依赖项”一起使用，用于删除包及其安装的任何依赖项。应注意，使用此复选框可能会导致项目中的引用中断。

3.2　JavaScript 和 CSS

3.2.1　静态文件

ASP. NET Core 允许向 Web 客户端提供静态文件，静态文件主要是 JavaScript 和 CSS 文件，这些文件存储在 wwwroot 文件夹内。

ASP. NET Core 允许用户创建自定义 JavaScript 和 CSS 文件，使用静态文件的方法为 StaticFileExtensions. UseStaticFiles( )，命名空间为 Microsoft. AspNetCore. Builder，根据参数不同主要包括以下三个方法，如表 3-1 所示。

表 3-1　创建 JavaScript 和 CSS 文件的方法

UseStaticFiles(IApplicationBuilder)	为当前请求路径启用静态文件服务
UseStaticFiles(IApplicationBuilder,StaticFileOptions)	使用给定的选项启用静态文件服务
UseStaticFiles(IApplicationBuilder,String)	为给定请求路径启用静态文件服务

如果 ASP. NET Core 项目采用的空模板进行创建，则需要在 Startup 类中添加 UseStaticFiles(IApplicationBuilder)，用于提供静态文件。

图 3-13　新建文件夹

3.2.2　部署 JavaScript 和 CSS

【例 3-5】　建立静态文件。

① 创建 ASP. NET Core Web 应用程序，项目名称为 JSCSSApp，模板选择“空”。

② 右击项目名称 JSCSSApp，在弹出的快捷菜单中选择“添加”→“新建文件夹”命令，文件夹的名称为 wwwroot。右击 wwwroot 文件夹，在文件夹中再创建一个 CSS 文件夹，如图 3-13 所示。

③ 右击 css 文件夹，在弹出的快捷菜单中选择“添加”→“新建项”命令，打开“添加新项-

JSCSSApp"对话框,选择"样式表"选项,在"名称"文本框中输入 site1.css,单击"添加"按钮,如图 3-14 所示。

图 3-14 新建样式表

④ 打开 site1.css 文件,并输入如下代码:

```
h2{
    font-size:30pt;
    font-family:sans-serif;
}
table,td {
    border :3px dashed red;
    border-collapse :collapse;
    padding :4px;
    font-family :sans-serif;
}
```

重复上述步骤,在 css 文件夹下创建 site2.css 样式表文件,代码如下:

```
p {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    font-size: 15pt;
    color: red;
    background-color: aqua;
    border: 1px,solid black;
    padding: 3px;
}
```

⑤ 在项目解决方案的 wwwroot 文件夹中创建一个 js 文件夹,选中 js 文件夹并右击,在弹出的快捷菜单中选择"添加"→"新建项"命令,在"添加新项-JSCSSApp"对话框中选择 JavaScript 文件,文件名称为 site1.js,如图 3-15 所示。

⑥ 打开 site1.js 文件,添加代码如下:

```
document.addEventListener("DOMContentLoaded", function ()
{
```

图 3-15　创建 JavaScript 文件

```
    var element = document.createElement("p");
    element.textContent = "第一个 js 文件";
    document.querySelector("body").appendChild(element);
});
```

重复上述步骤，在 js 文件夹下创建 site2.js 样式表，代码如下：

```
document.addEventListener("DOMContentLoaded", function ()
{
var element = document.createElement("p");
element.textContent = "第二个 js 文件";
document.querySelector("body").appendChild(element);
});
```

⑦ 右击 wwwboot 文件夹，在弹出的快捷菜单中选择“添加”→“新建项”命令，打开“添加新项-JSCSSApp”对话框，选择“HTML 页”选项，在“名称”文本框中输入 index.html，单击“添加”按钮完成静态网页文件的添加。打开 index.html 文件，对该文件编辑如下：

```
<!DOCTYPE html>
<html>
<head>
    <link href="css/site1.css" rel="stylesheet" />
    <link href="css/site2.css" rel="stylesheet" />
    <script src="js/site1.js"></script>
    <script src="js/site2.js"></script>
</head>
<body class=" p-1 ">
    <h2>选修课人数统计</h2>
    <table>
        <thead>
            <tr><td>课程名称 </td><td>人数 </td></tr>
        </thead>
        <tbody>
            <tr>
```

```
                <td>计算机网络</td>
                <td>60 人</td>
            </tr>
            <tr>
                <td>数字媒体处理工具</td>
                <td>65 人</td>
            </tr>
            <tr>
                <td>计算机系统结构</td>
                <td>30 人</td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

⑧ 打开 Startup.cs 文件，对 Configure 文件进行编辑，代码如下所示：

```
if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        app.UseDefaultFiles();
        app.UseStaticFiles();
```

⑨ 运行该程序，效果如图 3-16 所示。

图 3-16　页面运行效果

3.2.3　静态文件打包和压缩

在 ASP.NET 中可以使用打包与压缩这两种技术来提高 Web 应用程序页面加载的性能。例 3-5 中，项目中包含 4 个静态文件，浏览器需要发送 4 次请求才能获得这些静态文件，可以通过打包方式减少请求次数。

打包是将多个文件(CSS、JavaScript 等资源文件)合并到单个文件中。文件合并后可减少 Web 资源文件从服务器访问的次数，这样也可提高页面载入的性能。

压缩是将各种不同的代码进行优化，以减少请求资源文件的体积。压缩的常见方法是删除不必要的空格和注释，并将变量名缩减为一个字符。

在 Visual Studio 2019 中，打包和压缩需要用到 Bundler&Minifier，需要开发人员进行

安装。

【例 3-6】 打包和压缩。

① 在 Visual Studio 2019 中，选择“扩展”→“管理扩展”命令，如图 3-17 所示。

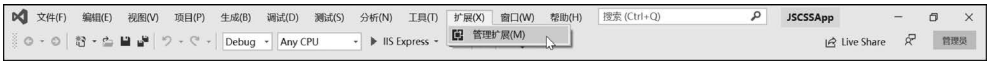

图 3-17 管理扩展

② 在搜索栏中输入 bundle，选中 Bundler&Minifier 选项，单击“下载”按钮，如图 3-18 所示。

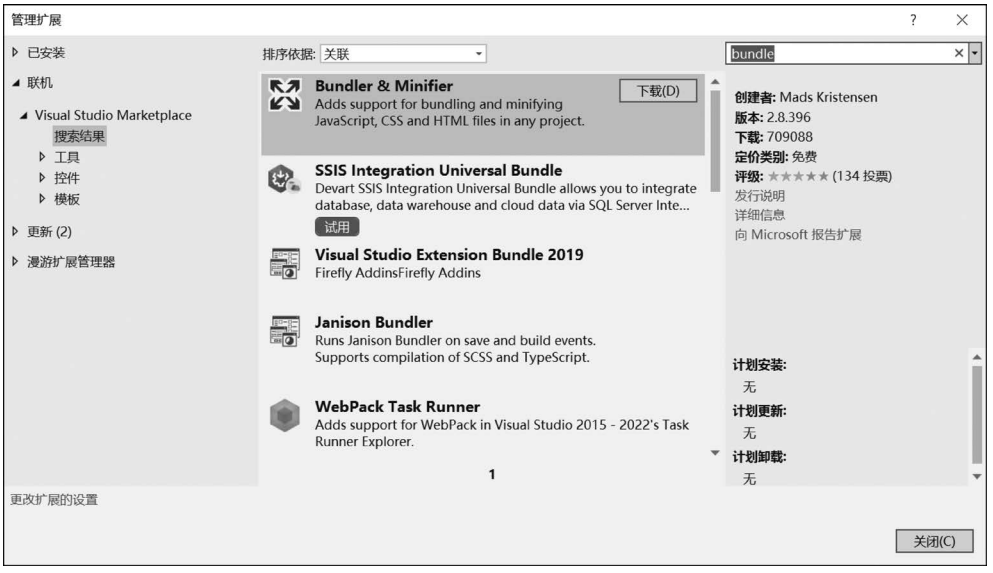

图 3-18 下载 Bundler&Minifier

③ 下载完成后程序会自动进行安装，在图 3-19 所示的对话框中单击 Modify 按钮以完成安装，之后重新启动 Visual Studio 2019。

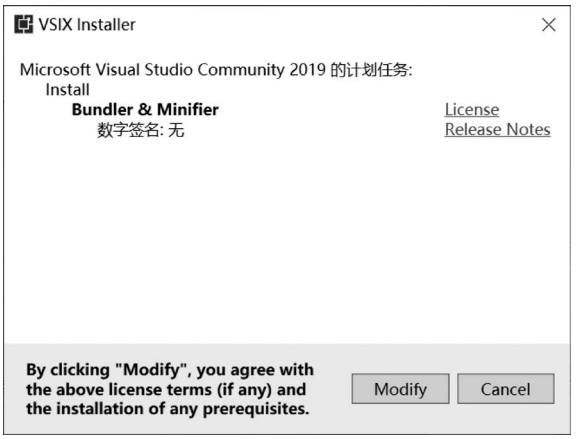

图 3-19 安装 Bundler&Minifier

④ Bundler&Minifier 扩展可以合并同类型的文件。打开例 3-5，选中 site1. css 和

site2.css，右击，在弹出的快捷菜单中选择 Bundler&Minifier→Bundle and Minify Files 命令，如图 3-20 所示。

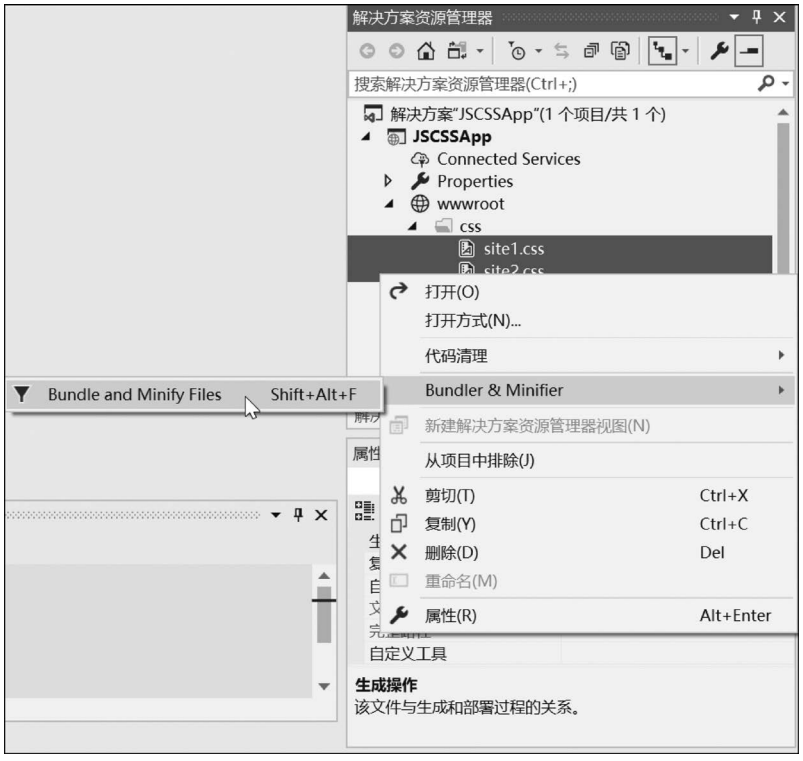

图 3-20 合并样式表文件

⑤ 在"另存为"对话框中将 CSS 文件存为 bundle.css，如图 3-21 所示。

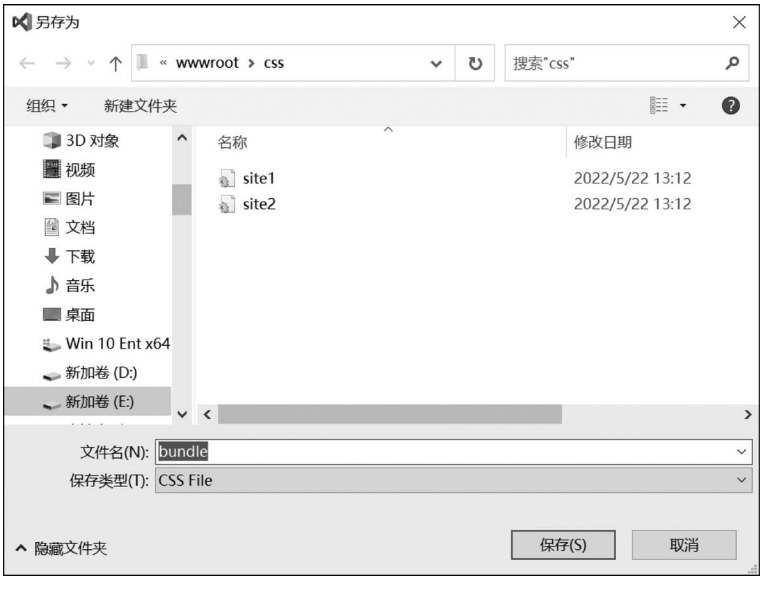

图 3-21 命名合并样式表文件

⑥ 单击“保存”按钮，在 wwwroot 文件夹下生成 bundle.css 文件，bundle.min.css 就是 site1.css 和 site2.css 合并的文件，这个文件所占空间很小，打开 bundle.min.css 文件，代码如下：

```
h2 {
    font-size: 30pt;
    font-family: sans-serif;
}
table, td {
    border: 3px dashed red;
    border-collapse: collapse;
    padding: 4px;
    font-family: sans-serif;
}

p {
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
    font-size: 15pt;
    color: red;
    background-color: aqua;
    border: 1px, solid black;
    padding: 3px;
}
```

⑦ Bundler&Minifier 扩展合并为 JavaScript 文件。选中 site1.js 和 site2.js，右击，在弹出的快捷菜单中选择 Bundler&Minifier→Bundle and Minify Files，将文件保存为 bundle.js，如图 3-22 所示。

图 3-22　合并 JavaScript 文件

⑧ 打开 bundle.min.js 文件，代码如下：

```
document.addEventListener("DOMContentLoaded", function () {
var element = document.createElement("p");
element.textContent = "第二个 JavaScript 文件";
```

```
document.querySelector("body").appendChild(element);
});
document.addEventListener("DOMContentLoaded", function () {
var element = document.createElement("p");
element.textContent = "第一个 JavaScript 文件";
document.querySelector("body").appendChild(element);
});
```

通过 bundle.min.js 和 bundle.min.css 代码可以发现所有的空白符都已被删除，从而减少文件占用的空间。

⑨ 打开 wwwroot 下的 Index.html 文件，修改静态文件的引用文件如下：

```
<head>
    <link href="css/bundle.min.css" rel="stylesheet" />
    <script src="js/bundle.min.js"></script>
</head>
```

运行该网页，效果如图 3-16 所示。

⑩ Bundler&Minifier 扩展将文件的处理记录存放在 bundleconfig.json 文件中，bundleconfig.json 是 ASP.NET Core 项目提供的一个配置文件。默认情况下该文件存放在项目的根目录下。该文件定义了打包的配置选项，实现了自定义脚本文件(wwwroot/js/site.js) 和样式表文件(wwwroot/css/site.css)的配置。其配置信息如下：

```
[
    {
        "outputFileName": "wwwroot/css/bundle.css",
        "inputFiles": [
          "wwwroot/css/site1.css",
          "wwwroot/css/site2.css"
        ]
      },
      {
            "outputFileName": "wwwroot/js/bundle.js",
            "inputFiles": [
            "wwwroot/js/site2.js",
            "wwwroot/js/site1.js"
        ]
    }
]
```

当在开发模式下运行应用程序时可以使用未压缩 CSS 和 JavaScript 脚本文件。在生产环境中可以使用压缩后的资源文件，这样可以提高应用程序的性能。

3.3 日志管理

日志管理在程序中应用十分广泛，可以根据日志提供的信息查看产生的错误，进而对程序进行调试。在本节中将介绍日志的建立和使用。

3.3.1 日志提供程序

日志提供程序的主要作用是将生成的日志通过多种方式进行显示和输出。ASP.NET

Core 支持以下 4 种内置日志记录提供程序。

1. Console

将日志通过控制台进行输出。

2. Debug

调试时在 Visual Studio 的 Debug 窗口中能够看到日志的输出。

3. EventSource

跨平台日志记录，此提供程序可以向事件跟踪器输出日志。

4. EventLog

能够在 Windows 的系统日志中看到输出日志。

下面代码演示了如何使用日志提供程序。

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
            {
                logging.ClearProviders();
                logging.AddConsole();
                logging.AddDebug();
                logging.AddEventSourceLogger();
                bool isWindows = RuntimeInformation.IsOSPlatform(OSPlatform.Windows);
                if (isWindows)
                    {
                        logging.AddEventLog();
                    }
            })
```

在 CreateDefaultBuilder()方法中默认通过调用 ConfigureLogging()方法添加了 Console、Debug、EventSource Logger 和 EventLog(仅 Windows)共 4 种日志记录提供程序。如果不想使用默认添加的日志提供程序，也可以先通过 ClearProviders()清除所有已添加的日志记录提供程序，然后再添加需要的。

3.3.2 日志分类

为了简化配置根据日志名称将把日志分成系统日志、EFCore 日志以及应用日志三种。

1. 系统日志

以 Microsoft 或 System 开始的日志，配置字段为 SystemLogLevel，默认级别为 Warning。

2. EFCore 日志

以 Microsoft.EntityFrameworkCore 开始的日志，配置字段为 EFCoreCommandLevel，默认级别为 Information。

3. 应用日志

除了系统日志及 EFCore 日志以外的日志。配置字段为 AppLogLevel，默认级别为 Information。

3.3.3 日志级别

日志在实际应用中是需要指定级别的。如在测试环境中希望看到更为详尽的信息，但在生产环境中只需要记录严重的错误就可以了。可以通过配置文件或 AddFilter()方法实现对日志的配置。

1. 日志级别分类

在 ASP.NET Core 中提供了 6 种级别日志，按严重性从低到高进行排序，分别为 Trace（追踪）、Debug（调试）、Information（信息）、Warning（警告）、Error（错误）和 Critical（致命）。低于设置级别的日志不会输出。

1）Trace

该级别的日志经常用来记录程序员在调试系统时出现的一些信息。该信息最为详细，其中包括一些敏感数据，在生产环境中是禁用的。

2）Debug

该级别的日志记录了在开发和调试阶段出现的信息，因该信息量大，在生产环境中应该尽量避免启用 Debug 日志。

3）Information

该级别的日志记录了应用程序运行时产生的信息，平时应用较多。

4）Warning

该级别的日志记录了在应用程序中产生的异常的或者不确定的信息，导致这些信息产生的事件通常不会使程序出错，例如文件打开错误。

5）Error

该级别的日志记录了在应用程序中执行某个操作后导致错误产生的信息，这些操作通常无法处理但不会导致整个应用程序出错。

6）Critical

该级别的日志经常用来记录需要立即处理的事件，该事件会导致应用程序崩溃，如硬盘空间不足等。

2. 日志过滤

当应用程序运行时会产生大量的日志信息，其中用户关心的信息可以进行记录和保存，对于那些不关心的信息则进行抛弃，这就要用到日志过滤功能。

1）日志配置方式

日志配置方式有两种：一种是编码方式，另一种是 JSON 文件配置方式。

（1）编码方式。

在 ConfigureLogging()方法中可以进行日志级别的设置。

```
Host.CreateDefaultBuilder(args)
.ConfigureLogging(builder =>
{
//删除所有的日志提供程序
builder.ClearProviders();
builder.AddConsole(loggerOptions => loggerOptions.IncludeScopes = true);
builder.SetMinimumLevel(LogLevel. Error);
})
```

上述代码中通过调用 ILoggingBuilder 接口的 SetMinimumLevel()方法实现了设置最低日志级别。该日志级别设置完成后所有低于该级别的日志将不会被处理和显示。例如设置的最低日志级别为 Error，那么 Warning、Information、Debug 和 Trace 级别的日志都不会被显示。

除了 SetMinimumLevel()方法能够实现最低日志级别外，AddFilter()方法也能够实现更为复杂的日志条件。下面的代码通过 ILoggerBuilder 接口设置了日志级别等于并高于 Warning 级别的日志。

```
Host.CreateDefaultBuilder(args)
    .ConfigureLogging(logging =>
        {
            logging.AddDebug().AddFilter("Microsoft", LogLevel.Warning);
        })
```

(2) JSON 文件配置方式。

默认情况下对日志的配置会通过 appsettings.json 文件来实现，该文件包含一个 Logging 节，在此节中不但可以对所有日志进行统一设置，而且还可以对每一种日志提供程序分别进行配置。

```
{
    "Logging": {
      "LogLevel": {
          "Default": "Debug",
          "Microsoft": "Error",
          "Microsoft.Hosting.Lifetime": "Debug"
      },
      "Console": {
        "LogLevel": {
            "Default": "Information"
        }
    },
    "Debug": {
        "LogLevel": {
            "Microsoft": "Debug"
        }
    },
    "EventSource": {
        "LogLevel": {
                "Default": "Warning",
                "Microsoft": "Warning",
                "Microsoft.Hosting.Lifetime": "Warning"
            }
        }
    }
}
```

在上述代码中 Logging.LogLevel 结点为全局配置项，该配置对所有日志提供者都起作用。在 LogLevel 中的字段，如 Default、Microsoft 等表示日志的类别。

- Default：如果分类日志没有在 LogLevel 中进行配置，则应用 Default 的配置。
- Microsoft：以 Microsoft 开头的所有类别的日志应用此项配置。

- Microsoft. Hosting. Lifetime：此类别较 Microsoft 更为具体，凡以 Microsoft. Hosting. Lifetime 开头的日志均应用当前配置。

2）日志过滤规则

日志的过滤规则由日志记录提供程序的全名、日志类别和日志级别这三个因素进行设置。虽然允许设置多条过滤规则，但日志框架最终只保留一个最优规则，其他都会忽略掉。

规则定义如下：首先在框架中寻找与日志记录提供程序一致的配置，如果不存在则使用 LogLevel 下的通用配置。如果有与日志提供程序一致的配置再查询该配置下是否存在最长前缀的配置与之相匹配，如果没有则应用 default 配置。如果找到了多条匹配项则只保留最后一条。如果没有选择任何规则则使用 MinimumLevel 这个配置项，该配置项的默认值为 Information。

3.3.4 日志建立接口文件

在创建日志过程中通常会用到 ILogger 和 ILoggerFactory 两个接口来进行日志文件的建立。其实现过程分别为：

- 注入 ILogger 接口，调用相应方法。
- 注入 ILoggerFactory，调用 ILoggerFactory. CreateLogger(string categoryName)方法创建 ILogger，调用相应方法。

其实现的源码如下。

```
public class MyLog
        {
            ILogger<MyLog> _log1;
            ILogger _log2;
            public MyLog(ILogger<MyLog> logger, ILoggerFactory loggerFactory)
            {
                _log1 = logger;
                _log2 = loggerFactory.CreateLogger("loggerFactoryTest"); ;
            }
            public void ShowLog()
            {
                _log1.LogInformation("Show _log1");
                _log2.LogInformation("Show _log2");
            }
        }
```

3.3.5 日志消息模板

在应用程序开发过程中，为了使得同一类的日志在输出信息格式上保持一致，可以使用日志消息模板来实现。在该模板中，通过占位符来替换不同的数据。代码如下所示。

```
string parm = "调试";
_logger.LogError("现在的时间是 {time}，日志的级别为{logger}", DateTime.Now, parm);
```

上面代码中的{time}和{logger}均为占位符，而 DateTime. Now 和 parm 是分别提供值的参数。该语句的运行结果如下。

```
现在的时间是 2022/6/5 13:58:26，日志的级别为调试
```

3.3.6 日志应用

1. 在控制台应用程序中创建日志框架

下面介绍在 ASP.NET Core 控制台应用程序中创建日志框架的过程。

【例 3-7】 创建控制台日志。

① 打开 Visual Studio 2019 应用程序，选择“创建新项目”选项，如图 3-23 所示。

图 3-23 创建新项目

② 在“创建新项目”对话框中选择“控制台应用”选项，单击“下一步”按钮，如图 3-24 所示。

图 3-24 创建控制台应用项目

③ 在“配置新项目”对话框中输入项目名称 ConsoleApp，选择项目存储位置后单击“创建”按钮，如图 3-25 所示。

图 3-25　配置新项目

④ 打开“解决方案资源管理器”，右击“依赖项”，在弹出的快捷菜单中选择“管理 NuGet 程序包”命令，如图 3-26 所示。

图 3-26　选择“管理 NuGet 程序包”命令

⑤ 在打开的“NuGet 包管理器”中分别安装 Microsoft. Extensions. Loggin 和 Microsoft. Extensions. Logging. Console，如图 3-27 所示。

⑥ 打开 Program. cs 文件并在主函数中输入如下代码。

```
var services = new ServiceCollection();
services.AddLogging(builder =>
{
    builder
        .ClearProviders()
        .AddConsole()
        .AddFilter("Microsoft", LogLevel.Debug)
        .AddFilter("System", LogLevel.Debug)
        .SetMinimumLevel(LogLevel.Debug);
});
var provider = services.BuildServiceProvider();
var logger = provider.GetRequiredService<ILogger<Program>>();
logger.LogTrace(0, "现在输出的是 logger 追踪日志信息");
logger.LogDebug(1, "现在输出的是 logger 调试日志信息");
logger.LogInformation(2, "现在输出的是 logger 常用日志信息");
```

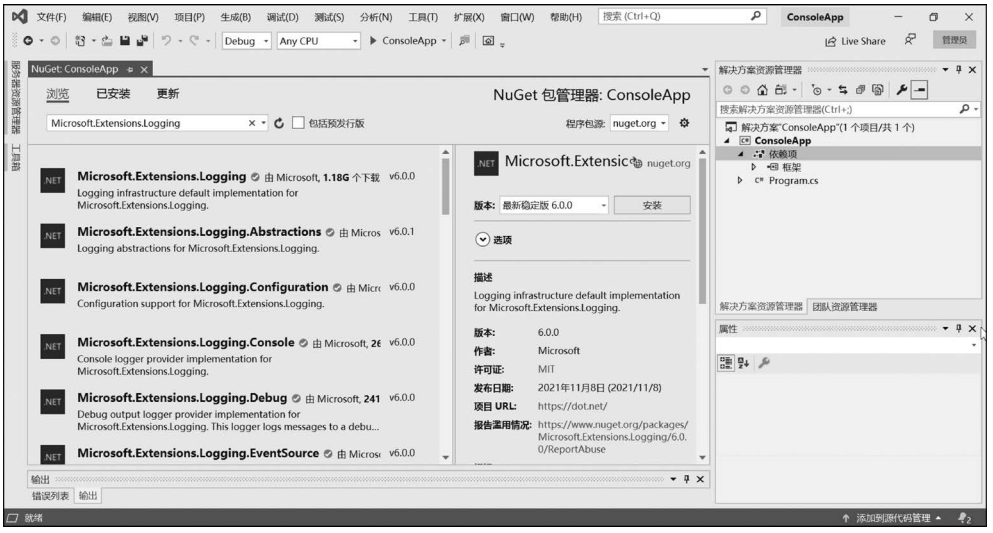

图 3-27　安装程序包

```
logger.LogWarning(3, "现在输出的是 logger 警告日志信息");
logger.LogError(4, "现在输出的是 logger 错误日志信息");
logger.LogCritical(5,"现在输出的是 logger 致命日志信息");
Console.WriteLine("控制台下的日志输出!");
```

在该文件中加入代码运行所需要的命名空间。

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

⑦ 运行该程序，显示结果如图 3-28 所示。

```
控制台下的日志输出!
dbug: ConsoleApp.Program[1]
      现在输出的是logger调试日志信息
info: ConsoleApp.Program[2]
      现在输出的是logger常用日志信息
warn: ConsoleApp.Program[3]
      现在输出的是logger警告日志信息
fail: ConsoleApp.Program[4]
      现在输出的是logger错误日志信息
crit: ConsoleApp.Program[5]
      现在输出的是logger致命日志信息

C:\temp\ConsoleApp\ConsoleApp\bin\Debug\netcoreapp3.1\ConsoleApp.exe (
进程 33612)已退出，代码为 0。
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调
试停止时自动关闭控制台”。
按任意键关闭此窗口. . .
```

图 3-28　控制台输出结果

2. 在 ASP.NET Core API 应用程序中创建日志框架

下面介绍在 ASP.NET Core API 应用程序中创建日志框架的过程。

【例 3-8】 创建 ASP.NET Core API 应用程序日志。

① 打开 Visual Studio 2019 应用程序，创建 ASP.NET Core Web 应用程序，如图 3-29 所示。

② 在“配置新项目”对话框中输入项目名称 LogWebApp，单击“创建”按钮，如图 3-30 所示。

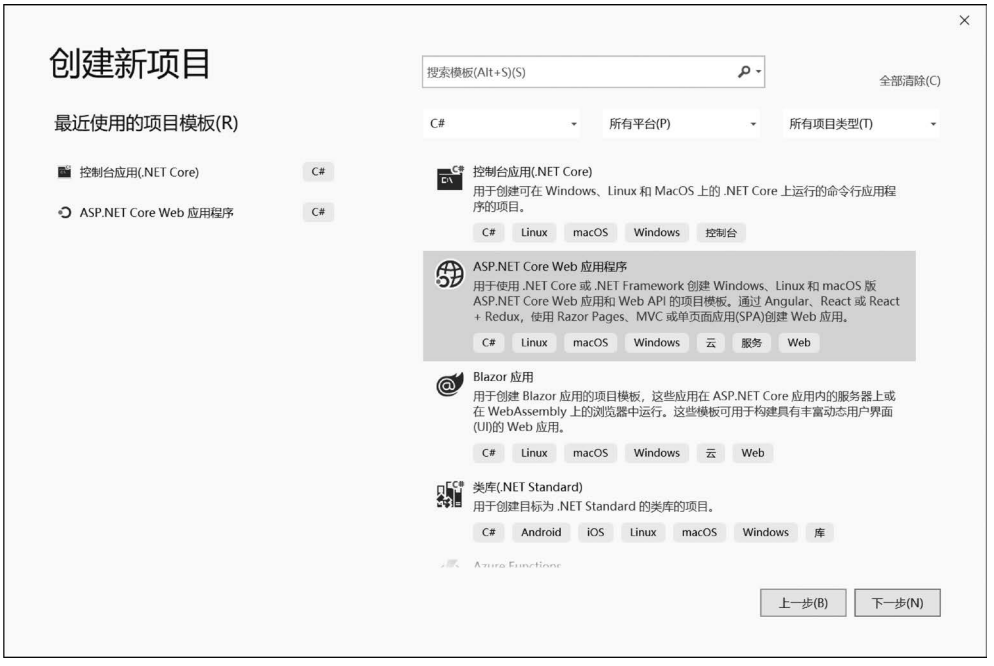

图 3-29　创建新项目

图 3-30　配置新项目

③ 在“创建新的 ASP. NET Core Web 应用程序”对话框中选择 API 项目，单击“创建”按钮，如图 3-31 所示。

④ 在新建立的 API 项目中微软提供了一个天气情况的示例。下面在该示例的基础上完成日志的建立和输出。打开 appsettings. json 文件并修改，完成日志配置。代码如下。

图 3-31　选择 API 项目

```
{
    "Logging": {                                    //日志配置结点
        "LogLevel": {                               //默认筛选器
            "Default": "Information",               //默认日志类别
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    },
    "Console": {
        "LogLevel": {
            "Default": "Error"
        }
    },
    "Debug": {                                      //针对日志提供程序 Debug 的配置
        "LogLevel": {
            "Default": "Information",               //默认日志类别
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    }
}
```

⑤ 打开 Program.cs 文件，在该文件中构建一个容器并将配置注入进容器，代码如下。

```
public class Program
{
    public static void Main(string[] args)
    {

        var configBuilder = new ConfigurationBuilder().AddJsonFile("appsettings.json").
        Build();
       CreateHostBuilder(args).Build().Run();
```

```
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
          .ConfigureLogging((hostingContext, logging) =>
          {
              logging.ClearProviders();
              logging.AddConfiguration(hostingContext.Configuration.GetSection("Debug"));
              logging.AddDebug();
          })
          .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

上述代码中使用 AddConfiguration()方法给日志添加配置,指定 appsettings.json 文件根结点下面的 Debug 结点作为日志输出的配置项。使用 AddDebug()方法将日志在调试窗口中进行显示输出。

由于 ASP.NET Core Web 框架已经自带了日志包,因此无须再引入。为该页面添加命名空间,代码如下。

```
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
```

⑥ 打开根目录下 WeatherForecast.cs 文件,该文件用来实现创建日志对象并打印日志。修改代码如下。

```
[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]
        public IEnumerable<WeatherForecast> Get()
        {
            string Message = $"当前时间:{DateTime.Now.ToLongTimeString()}";
            _logger.LogTrace(0, "现在显示的是跟踪日志信息,{Message}", Message);
            _logger.LogDebug(1, "现在显示的是调试日志信息, {Message}", Message);
            _logger.LogInformation(2, "现在显示的是常规日志信息, {Message}", Message);
            _logger.LogWarning(3, "现在显示的是警告日志信息,{Message}", Message);
            _logger.LogError(4, "现在显示的是错误日志信息, {Message}", Message);
            _logger.LogCritical(5, "现在显示的是致命日志信息, {Message}", Message);
            var rng = new Random();
```

```
            return Enumerable.Range(1, 5).Select(index => new WeatherForecast
            {
                Date = DateTime.Now.AddDays(index),
                TemperatureC = rng.Next(-20, 55),
                Summary = Summaries[rng.Next(Summaries.Length)]
            })
            .ToArray();
        }
    }
```

⑦ 运行该项目。打开调试窗口，显示结果如图 3-32 所示。

```
LogWebApp.Controllers.WeatherForecastController: Information: 现在显示的是常规日志信息, 当前时间: 12:58:43
LogWebApp.Controllers.WeatherForecastController: Warning: 现在显示的是警告日志信息,当前时间: 12:58:43
LogWebApp.Controllers.WeatherForecastController: Error: 现在显示的是错误日志信息, 当前时间: 12:58:43
LogWebApp.Controllers.WeatherForecastController: Critical: 现在显示的是致命日志信息, 当前时间: 12:58:43
```

图 3-32　调试窗口输出结果

之前在 appsettings.json 文件中配置的日志提供程序为 Debug，同时默认日志类别为 Information，因此显示的日志信息级别为 Information 及以上。

3. 第三方日志类库 NLog

NLog 是一个应用在.NET Core 项目中的日志记录类库，它的配置方式简单灵活，日志支持输出的方式有多种，如文本文件、Windows 系统日志、数据库、控制台、邮箱等。在实际使用时可以根据不同情况进行配置，使调试诊断信息按照配置好的方式发送到一个或多个输出目标中。下面介绍 NLog 的操作过程。

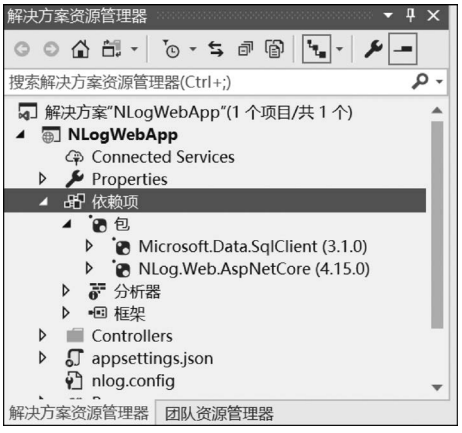

图 3-33　调试窗口输出结果

【例 3-9】 使用 NLog 将日志分别输出到文件和数据库。

① 新建 ASP.NET Core Web 应用程序，项目名称为 NLogWebApp，选择 API 项目模板。

② 安装 NLog 包。如果输出的目标是文件，则需安装 NLog.Web.AspNetCore(4.15.0)；如果输出的目标是数据库，则需安装 Microsoft.Data.SqlClient(3.1.0)，如图 3-33 所示。

③ 添加配置文件。右击项目名称 NLogWebApp，在弹出的快捷菜单中选择“添加”→“新建项”命令，在打开的“添加新项-NLogWebApp”对话框中选择“Web 配置文件”选项，并在“名称”文本框中输入 nlog.config，如图 3-34 所示。

④ 打开 nlog.config 文件，输入如下配置代码。

```
<?xml version="1.0" encoding="utf-8"?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" autoReload="true" throwExceptions="false" internalLogLevel="false" internalLogFile="NlogRecords.log">
<!-- enable asp.net core layout renderers -->
<extensions>
    <add assembly="NLog.Web.AspNetCore"/>
</extensions>
```

图 3-34　添加 nlog.config 配置文件

```
    <!--输出目标-->
    <targets>
        <!-- 1.将日志消息写入文件-->
        <target name="log_file" xsi:type="File"
            fileName="${basedir}/Logs/${shortdate}/${shortdate}.txt"
             layout="${longdate} | ${event-properties:item=EventId_Id:whenEmpty=0} |
${uppercase:${level}} | ${logger} | ${message} ${exception:format=tostring}"
            archiveFileName="${basedir}/archives/${shortdate}-{#####}.txt"
            archiveAboveSize="102400"
            archiveNumbering="Sequence"
            concurrentWrites="true"
            keepFileOpen="false" />
        <!-- 2.将日志消息写入数据库-->
        <target name= "database" xsi:type="Database" dbProvider="Microsoft.Data.SqlClient.
SqlConnection, Microsoft.Data.SqlClient"
        connectionString="data source=tjau;initial catalog=TestDB; Persist Security Info=True;
integrated security=true">
        <commandText>
        insert into NLogTable([Date],[Events],[Level],[Information]) values (getdate(), @
Events, @logLevel, @Information);
        </commandText>
        <parameter name="@Events" layout="${callsite}" />
        <parameter name="@logLevel" layout="${level}" />
        <parameter name="@Information" layout="${message}" />
    </target>
    </targets>
    <!--定义使用哪种方式输出-->
    <rules>
        <logger name="*" minlevel="Debug" writeTo="log_file" />
        <logger name="*" minlevel="Trace" writeTo="database" />
    </rules>
</nlog>
```

说明：

- 在 nlog 根结点有一个 internalLogLevel 属性，该属性的作用是如果 NLog 没有配置好或存在异常，则当 internalLogLevel 设定的值为 Trace 或 Debug 时，NLog 会产生一个内部自己的日志文件 NlogRecords.log，可以打开此文件查看具体原因，该文件通常保存在当前项目根目录，也可以通过 internalLogFile 属性自行设定保存位置。
- target 是数据输出的配置结点。在文件结点中，xsi:type="File"表示输出到文件；fileName 表示为添加的日志文件命名，文件名通常为日期；layout 表示写入日志文件的格式。在数据库结点，xsi:type="Database"表示日志输出到数据库；dbProvider 属性表示数据库适配器，数据库不同则有不同的设置值；connectionString 表示配置数据库连接字符串；commandText 表示添加到数据库中的命令脚本；parameter 表示数据库脚本的参数。
- rules 结点是各个日志记录器 Logger 的配置。name 后的属性表示记录器的名称；minlevel 表示最低匹配规则；writeTo 表示最终输出的目标。
- 在该配置文件中用到了数据表，该数据表建立脚本如下。

```
CREATE TABLE [dbo].[NLogTable](
[LogId] [int] IDENTITY(1,1) NOT NULL,
[Date] [datetime] NOT NULL,
[Events] [nvarchar](200) NULL,
[Level] [nvarchar](100) NULL,
[Information] [nvarchar](max) NULL,
) ON [PRIMARY]
```

⑤ 右击 nlog.config 文件，在弹出的快捷菜单中选择“属性”命令，将“复制到输出目录”属性值设置为“始终复制”，如图 3-35 所示。

图 3-35　设置 nlog.config 属性

⑥ 打开 Controllers 目录下的 WeatherForecastController.cs 文件，在 WeatherForecastController 类中添加如下代码。

```
private readonly Logger _log = LogManager.GetCurrentClassLogger();
```

该方法用于获取 Logger 实例。在 Get()方法中添加如下命令。

```
_log.Trace("现在输出的是追踪信息");
_log.Debug("现在输出的是调试信息");
_log.Info("现在输出的是常规信息");
_log.Warn("现在输出的是警告信息");
```

```
_log.Error("现在输出的是错误信息");
_log.Fatal("现在输出的是致命信息");
```

通过上述命令完成不同级别的日志的记录。

⑦ 运行当前项目，打开项目文件所在目录下的 bin\Debug\netcoreapp3.1\Logs 文件夹，在该文件夹中存储以日期命名的文件夹，打开该文件夹会发现有以日期命名的日志文件，打开该文件后显示的内容如图 3-36 所示。

图 3-36　日志输出到文件

打开数据库中的数据表 NLogTable 文件，显示的记录日志内容如图 3-37 所示。

	LogId	Date	Events	Level	Information
▶	1	2022-06-07 22:39:26.213	WebApplication1.Controllers.WeatherForecastController.Get	Trace	现在输出的是追踪信息
	2	2022-06-07 22:39:26.350	WebApplication1.Controllers.WeatherForecastController.Get	Debug	现在输出的是调试信息
	3	2022-06-07 22:39:26.403	WebApplication1.Controllers.WeatherForecastController.Get	Info	现在输出的是常规信息
	4	2022-06-07 22:39:26.547	WebApplication1.Controllers.WeatherForecastController.Get	Warn	现在输出的是警告信息
	5	2022-06-07 22:39:26.620	WebApplication1.Controllers.WeatherForecastController.Get	Error	现在输出的是错误信息
	6	2022-06-07 22:39:26.713	WebApplication1.Controllers.WeatherForecastController.Get	Fatal	现在输出的是致命信息
*	NULL	NULL	NULL	NULL	NULL

图 3-37　日志输出到数据库

可以看到，通过 NLog 将日志分别输出到了文本文件和数据表中。之所以二者输出的记录数不同，是因为在 nlog.config 配置文件中文件输出的级别为 Debug，而数据库输出的级别为 Trace。

小　　结

本章的主要内容包括管理软件包工具 NuGet 的使用、JavaScript 和 CSS 的部署以及日志的管理和应用。在进行 ASP.NET Core 项目的开发中经常会用到 NuGet 技术，读者能够通过 NuGet 工具完成软件包的安装、更新和卸载。JavaScript 和 CSS 在页面布局中必不可少，读者应能够熟练掌握它们的合并与部署。最后通过案例介绍了 ASP.NET Core 自身所带日志框架的应用以及第三方工具 NLog 的使用。

习　　题

一、作业题

1. 简述 Visual Studio 2019 如何管理软件包。
2. 简述 NuGet 的特性。
3. 简述如何在 ASP.NET Core Web 项目中部署 JavaScript 和 CSS。

二、上机实践题

1. 通过创建日志工厂对象的方式重新完成例 3-7 的控制台输出。

2. 根据图 3-38 所示的数据表结构，要求通过 NLog 完成日志的数据表信息输出。

列名	数据类型	允许 Null 值
Id	int	☐
LogDate	datetime	☑
LogLevel	varchar(500)	☑
LogType	varchar(500)	☑
Message	varchar(5000)	☑
MachineName	varchar(500)	☑
MachineIp	varchar(500)	☑
RequestController	varchar(500)	☑
RequestAction	varchar(500)	☑
RequestMethod	varchar(1000)	☑
RequestHeaders	varchar(5000)	☑
RequestPostBody	varchar(2000)	☑
RequestQuery	varchar(1000)	☑
RequestUrl	varchar(500)	☑

图 3-38　数据表结构