****

**软件开发环境配置说明**

题 目： 高校学生档案管理系统

学 院： 计算机科学与工程学院

专 业： 计算机科学与技术

班 级： 2018 级卓越工程师班

学生姓名： 熊文婷

于泽浩

贾傲羊

杨子豪

指导教师： 代祖华

目 录

[第一部分 开发环境 1](#_Toc8051)

[（一）ASP.NET Core环境设置 1](#_Toc9509)

[（二）blazor wass项目配置说明 3](#_Toc3330)

[（三）ASP.NET Core 托管部署 6](#_Toc13634)

[（四）多个Blazor WebAssembly应用的静态资产和类库 11](#_Toc31981)

[第二部分 独立部署 12](#_Toc1244)

[（一）Azure 应用服务 12](#_Toc12204)

[（二）IIS 12](#_Toc2325)

[（三）web.config 12](#_Toc19004)

[（四）使用自定义 web.config 13](#_Toc19921)

[（五）Brotli 和 Gzip 压缩 14](#_Toc8294)

[（六）Azure 存储 15](#_Toc14687)

[（七）Nginx 15](#_Toc568)

[（八）Apache 16](#_Toc23181)

[（九）GitHub 18](#_Toc11874)

[第三部分 主机配置值 19](#_Toc31766)

[（一）内容根 19](#_Toc22609)

[（二）基路径 19](#_Toc20686)

[（三）URL 20](#_Toc26681)

[（四）配置裁边器 20](#_Toc26115)

[（五）更改 DLL 文件的文件扩展名 20](#_Toc3190)

[第四部分 解决完整性检查失败 23](#_Toc13786)

[（一）诊断完整性问题 23](#_Toc12007)

[（二）禁用非PWA应用的完整性检查 25](#_Toc16829)

[（三）禁用PWA的完整性检查 26](#_Toc13877)

# 第一部分 开发环境

Windows10 21h1 WorkStation

.net5.0

Blazor

AntDesign Pro

Visual studio 2019 Enterprice

## （一）ASP.NET Core环境设置

1.如何设置用于.NetCore应用程序开发的开发机器

2.安装SDK和IDE

3.验证安装

**开发和.NET Core应用程序所需的工具和软件：**

1.设备：（Windows、Mac、Linux）

2.IDE：Visual Studio、Visual Code  
3.NetSDK：软件开发工具包，有助于开发和运行系统中的应用程序。

**如何为构建 .NET Core/ASP.NET Core应用程序准备开发环境？**

NET Core可以通过两种方式安装：  
1.通过安装VisualStudio 2017/2019  
2.通过安装 .NET Core SDK

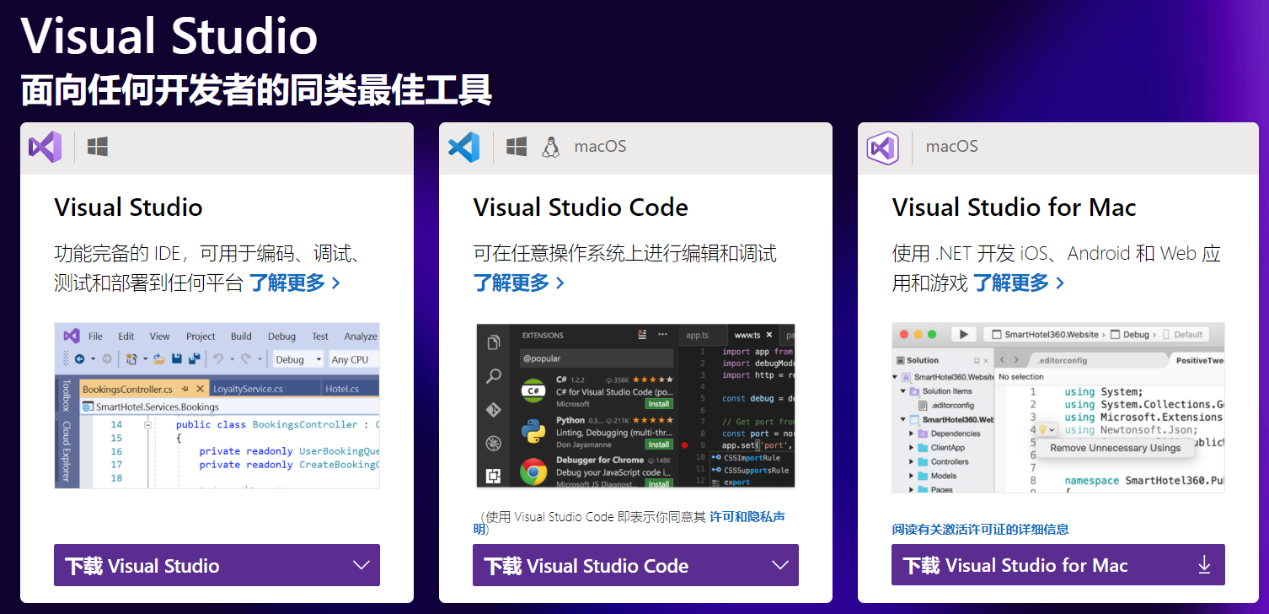
**安装VisualStudio 2017/2019：**

目前，.NETCore2.1 和 .NETCore3.1 具有长期支持（LTS）。Visual Studio 2017 支持 .NET Core 2.1，而 Visual Studio 2019 支持这两个版本。

可以使用喜欢的IDE版本，如 Visual Studio 2017/2019、Visual Studio Code等来开发、还原、构建和运行 .NET Core 应用程序。

在下面的示例中, 使用的是Windows操作系统。因此，我将使用 Visual Studio作为我的IDE，或者您可以说集成开发环境 （IDE） 用于开发ASP.NET应用程序。  
 如果开发 PC 上没有 Visual Studio，则建议安装最新的 Visual Studio 2019。如果已经拥有 Visual Studio 2017 或 2019，则您已经安装了 .NET Core SDK 2.1。

下载并且安装VisualStudio

根据操作系统下载并安装 Visual Studio 2019。根据您的许可证选择适当的版本。Visual Studio 社区版对学生、开源贡献者和个人免费。  
[](https://img2020.cnblogs.com/blog/1161656/202009/1161656-20200922095427695-424052598.png)

**图1-1 Visual Stdio下载页面图**

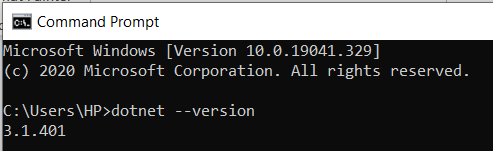
然后再根据您的操作系统, 下载所需要的VisualStudio, 然后下载并且安装它。请注意, 这里需要勾选.NetCore跨平台开发,如下所示。  
[](https://img2020.cnblogs.com/blog/1161656/202009/1161656-20200922095813161-1695481480.png)

**图1-2 Visual Stdio安装页面**

**.NET Core SDK组件**

在VisualStudio安装程序中, 选择到单个组件, 即可看到多个.NET CoreSDK版本, 即可根据你的实际需求进行安装。  
[](https://img2020.cnblogs.com/blog/1161656/202009/1161656-20200922100350511-1619178243.png)

**图1-3 Visual Stdio配置图**

安装完成之后, 你可以打开CMD窗口, 进行验证安装是否完成,如下所示。  
[](https://img2020.cnblogs.com/blog/1161656/202009/1161656-20200922100050973-1547347993.png)

**图1-4 Visual Stdio下载安装成功图**

## （二）blazor wass项目配置说明

利用 [Blazor WebAssembly 托管模型](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0" \l "blazor-webassembly)：

1.将 Blazor 应用、其依赖项及 .NET 运行时并行下载到浏览器。

2.应用将在浏览器线程中直接执行。

**自定义加载启动资源的方式：**

自定义如何使用 loadBootResource API 加载启动资源。 有关详细信息，请参阅 [ASP.NET Core Blazor 启动](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/fundamentals/startup?view=aspnetcore-5.0" \l "load-boot-resources)。

压缩发布 Blazor WebAssembly 应用时，将在发布过程中对输出内容进行静态压缩，从而减小应用的大小，并免去运行时压缩的开销。 使用以下压缩算法：

1.[Brotli](https://tools.ietf.org/html/rfc7932)（级别最高）

2.[Gzip](https://tools.ietf.org/html/rfc1952)

Blazor 依赖于主机提供适当的压缩文件。 使用 ASP.NET Core 托管项目时，主机项目能够执行内容协商并提供静态压缩文件。 托管 Blazor WebAssembly 独立应用时，可能需要额外的工作来确保提供静态压缩文件：

1.有关 IIS web.config 压缩配置，请参阅 [IIS：Brotli 和 Gzip 压缩](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/host-and-deploy/webassembly?view=aspnetcore-5.0" \l "brotli-and-gzip-compression) 部分。

2.如果在不支持静态压缩文件内容协商的静态托管解决方案（例如 GitHub 页面）上进行托管，请考虑配置应用以提取和解码 Brotli 压缩文件：

（1）从 [google/brotli GitHub repository](https://github.com/google/brotli) 中获取 JavaScript Brotli 解码器。 解码器文件被命名为 decode.js，并且位于存储库的 [js 文件夹](https://github.com/google/brotli/tree/master/js)中。

（2）备注[google/brotli GitHub 存储库](https://github.com/google/brotli)中的缩小版 decode.js 脚本 (decode.min.js) 中存在回归。

在问题 [TypeError in decode.min.js (google/brotli #881)](https://github.com/google/brotli/issues/881) 得到解决之前，采取以下方法之一：

1.暂时使用脚本的未缩小版本。

2.使用与 ASP.NET Core 兼容的第三方缩小工具在生成时自动缩小脚本。

3.使用 [npm 包](https://www.npmjs.com/package/brotli)。

本部分中的示例代码使用脚本的“未缩小”版本 (decode.js)：

1.更新应用以使用解码器。 将 wwwroot/index.html 中结束 <body> 标记内的标记更改为以下内容：

<script src="decode.js"></script>

<script src="\_framework/blazor.webassembly.js" autostart="false"></script>

<script>

Blazor.start({

loadBootResource: function (type, name, defaultUri, integrity) {

if (type !== 'dotnetjs' && location.hostname !== 'localhost') {

return (async function () {

const response = await fetch(defaultUri + '.br', { cache: 'no-cache' });

if (!response.ok) {

throw new Error(response.statusText);

}

const originalResponseBuffer = await response.arrayBuffer();

const originalResponseArray = new Int8Array(originalResponseBuffer);

const decompressedResponseArray = BrotliDecode(originalResponseArray);

const contentType = type ===

'dotnetwasm' ? 'application/wasm' : 'application/octet-stream';

return new Response(decompressedResponseArray,

{ headers: { 'content-type': contentType } });

})();

}

}

});

</script>

若要禁用压缩，请将 BlazorEnableCompression MSBuild 属性添加到应用的项目文件，并将值设置为 false：

<PropertyGroup>

<BlazorEnableCompression>false</BlazorEnableCompression>

</PropertyGroup>

可在命令行界面中使用以下语法将 BlazorEnableCompression 属性传递给 [dotnet publish](https://docs.microsoft.com/zh-cn/dotnet/core/tools/dotnet-publish) 命令：

dotnet publish -p:BlazorEnableCompression=false

重写 URL，以实现正确路由，在 Blazor WebAssembly 应用中路由对页组件的请求不如在 Blazor Server 托管应用中路由请求直接。 假设有一个具有两个组件的 Blazor WebAssembly 应用：

（1）Main.razor：在应用的根目录处加载，并包含指向 About 组件 (href="About") 的链接。

（2）About.razor：About 组件。

使用浏览器的地址栏（例如，https://www.contoso.com/）请求应用的默认文档：

1.浏览器发出请求。

2.返回默认页，通常为 index.html。

3.index.html 启动应用。

4.Blazor 的路由器进行加载，然后呈现 Razor Main 组件。

在 Main 页中，选择指向 About 组件的链接适用于客户端，因为 Blazor 路由器阻止浏览器在 Internet 上发出请求，针对 About 转到 www.contoso.com，并为呈现的 About 组件本身提供服务。 针对 Blazor WebAssembly 应用中的内部终结点的所有请求，工作原理都相同：这些请求不会触发对 Internet 上的服务器托管资源的基于浏览器的请求。 路由器将在内部处理请求。

如果针对 www.contoso.com/About 使用浏览器的地址栏发出请求，则请求会失败。应用的 Internet 主机上不存在此类资源，所以返回的是“404-找不到”响应。

由于浏览器针对客户端页面请求基于 Internet 的主机，因此 Web 服务器和托管服务必须将对服务器上非物理方式资源的所有请求重写为 index.html 页。 如果返回 index.html，应用的 Blazor 路由器将接管工作并使用正确的资源响应。

部署到 IIS 服务器时，可以将 URL 重写模块与应用的已发布 web.config 文件一起使用。 有关详细信息，请参阅 [IIS](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/host-and-deploy/webassembly?view=aspnetcore-5.0" \l "iis) 部分。

## （三）ASP.NET Core 托管部署

托管部署通过在 Web 服务器上运行的 [ASP.NET Core](https://docs.microsoft.com/zh-cn/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-5.0) 应用为浏览器提供 Blazor WebAssembly 应用。

客户端Blazor WebAssembly应用与服务器应用的其他任何静态Web资产一起发布到服务器应用的/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot 文件夹。 这两个应用一起部署。 需要能够托管 ASP.NET Core 应用的 Web 服务器。 对于托管部署，Visual Studio 会在选择 Hosted 选项（使用 dotnet new 命令时为 -ho|--hosted）的情况下，包含 Blazor WebAssembly 应用项目模板（使用 [dotnet new](https://docs.microsoft.com/zh-cn/dotnet/core/tools/dotnet-new) 命令时为 blazorwasm 模板）。

应用配置托管的 Blazor 解决方案可为多个 Blazor WebAssembly 应用提供服务。

 但多个客户端应用在托管的 Blazor WebAssembly 程序方案中运行时，不需要使用 Visual Studio 和 Visual Studio 解决方案。 如果你未在使用 Visual Studio，则忽略 {SOLUTION NAME}.sln 文件以及为 Visual Studio 创建的任何其他文件。

如下示例中：

1.初始（第一个）客户端应用是通过 Blazor WebAssembly 项目模板创建的解决方案的默认客户端项目。 第一个客户端应用可在浏览器中通过端口 5001 上或主机为 firstapp.com 的 URL /FirstApp 访问。

2.向解决方案添加第二个客户端应用：SecondBlazorApp.Client。 第二个客户端应用可在浏览器中通过端口 5002 上或主机为 secondapp.com 的 URL/SecondApp 访问。

使用现有的托管 Blazor 解决方案，或者从 Blazor 托管的项目模板创建一个新解决方案：

1.在客户端应用的项目文件中，在 <PropertyGroup> 中添加一个值为 FirstApp 的 <StaticWebAssetBasePath> 属性，以设置项目静态资产的基路径：

<PropertyGroup>

...

<StaticWebAssetBasePath>FirstApp</StaticWebAssetBasePath>

</PropertyGroup>

2.向解决方案添加第二个客户端应用：

（1）向解决方案的文件夹添加一个名为 SecondClient 的文件夹。 在添加 SecondClient 文件夹后，通过项目模板创建的解决方案文件夹包含以下解决方案文件和文件夹：

① Client（文件夹）

② SecondClient（文件夹）

③ Server（文件夹）

④ Shared（文件夹）

⑤ {SOLUTION NAME}.sln（文件）

占位符 {SOLUTION NAME} 是解决方案的名称。

（2）在 Blazor WebAssembly 项目模板的 SecondClient 文件夹中创建一个名为 SecondBlazorApp.Client 的 Blazor WebAssembly 应用。

（3）在SecondBlazorApp.Client 应用的项目文件中，执行以下操作：

①向<PropertyGroup>添加一个值为 SecondApp 的 <StaticWebAssetBasePath> 属性：

<PropertyGroup>

...

<StaticWebAssetBasePath>SecondApp</StaticWebAssetBasePath>

</PropertyGroup>

② 向 Shared 项目添加一个项目引用：

<ItemGroup>

<ProjectReference Include="..\Shared\{SOLUTION NAME}.Shared.csproj" />

</ItemGroup>

占位符 {SOLUTION NAME} 是解决方案的名称

1.在服务器应用的项目文件中，为新增的 SecondBlazorApp.Client 客户端应用创建一个项目引用：

<ItemGroup>

<ProjectReference Include="..\Client\{SOLUTION NAME}.Client.csproj" />

<ProjectReference Include="..\SecondClient\SecondBlazorApp.Client.csproj" />

<ProjectReference Include="..\Shared\{SOLUTION NAME}.Shared.csproj" />

</ItemGroup>

2.在服务器应用的 Properties/launchSettings.json 文件中，配置 Kestrel 配置文件 ({SOLUTION NAME}.Server) 的 applicationUrl，以访问位于端口 5001 和 5002 的客户端应用：

"applicationUrl": "https://localhost:5001;https://localhost:5002",

3.在服务器应用的 Startup.Configure 方法 (Startup.cs) 中，删除在调用 [UseHttpsRedirection](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.aspnetcore.builder.httpspolicybuilderextensions.usehttpsredirection) 后显示的以下行：

app.UseBlazorFrameworkFiles();

app.UseStaticFiles();

app.UseRouting();

app.UseEndpoints(endpoints =>

{

endpoints.MapRazorPages();

endpoints.MapControllers();

endpoints.MapFallbackToFile("index.html");

});

添加将请求映射到客户端应用的中间件。 下面的示例将中间件配置为在以下情况下运行：

（1）原始客户端应用的请求端口为 5001，或新增的客户端应用的请求端口为 5002。

（2）原始客户端应用的请求主机为 firstapp.com，或新增的客户端应用的请求主机为 secondapp.com。

 示例需要其他配置以实现以下目的：

① 访问示例主机域 firstapp.com 和 secondapp.com 上的应用。

② 获取客户端应用的证书以启用 TLS 安全性 (HTTPS)。

将以下代码放在先前已删除行的位置：

app.MapWhen(ctx => ctx.Request.Host.Port == 5001 ||

ctx.Request.Host.Equals("firstapp.com"), first =>

{

first.Use((ctx, nxt) =>

{

ctx.Request.Path = "/FirstApp" + ctx.Request.Path;

return nxt();

});

first.UseBlazorFrameworkFiles("/FirstApp");

first.UseStaticFiles();

first.UseStaticFiles("/FirstApp");

first.UseRouting();

first.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

endpoints.MapFallbackToFile("/FirstApp/{\*path:nonfile}",

"FirstApp/index.html");

});

});

app.MapWhen(ctx => ctx.Request.Host.Port == 5002 ||

ctx.Request.Host.Equals("secondapp.com"), second =>

{

second.Use((ctx, nxt) =>

{

ctx.Request.Path = "/SecondApp" + ctx.Request.Path;

return nxt();

});

second.UseBlazorFrameworkFiles("/SecondApp");

second.UseStaticFiles();

second.UseStaticFiles("/SecondApp");

second.UseRouting();

second.UseEndpoints(endpoints =>

{

endpoints.MapControllers();

endpoints.MapFallbackToFile("/SecondApp/{\*path:nonfile}",

"SecondApp/index.html");

});

});

1.在服务器应用的天气预报控制器 (Controllers/WeatherForecastController.cs) 中，将到 WeatherForecastController 的现有路由 ([Route("[controller]")]) 替换为以下路由：

[Route("FirstApp/[controller]")]

[Route("SecondApp/[controller]")]

先前添加到服务器应用的 Startup.Configure 方法的中间件会将向 /WeatherForecast 发出的传入请求修改为 /FirstApp/WeatherForecast 或 /SecondApp/WeatherForecast，具体取决于端口 (5001/5002) 或域 (firstapp.com/secondapp.com)。 为了将天气数据从服务器应用返回到客户端应用，需要前面的控制器路由。

## （四）多个Blazor WebAssembly应用的静态资产和类库

使用以下方法来引用静态资产：

1.如果资产位于客户端应用的 wwwroot 文件夹中，请正常提供路径：

<img alt="..." src="/{PATH AND FILE NAME}" />

{PATH AND FILE NAME} 占位符是 wwwroot 下的路径和文件名。

2.如果资产位于 [Razor 类库 (RCL)](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/components/class-libraries?view=aspnetcore-5.0) 的 wwwroot 文件夹中，请按照 [ASP.NET Core 的类库中的可重用 Razor UI](https://docs.microsoft.com/zh-cn/aspnet/core/razor-pages/ui-class?view=aspnetcore-5.0" \l "consume-content-from-a-referenced-rcl) 中的指导，在客户端应用中引用静态资产：

<img alt="..." src="\_content/{LIBRARY NAME}/{PATH AND FILE NAME}" />

占位符 {LIBRARY NAME} 是库的名称。 {PATH AND FILE NAME} 占位符是 wwwroot 下的路径和文件名。

# 第二部分 独立部署

独立部署将 Blazor WebAssembly 应用作为客户端直接请求的一组静态文件提供。 任何静态文件服务器均可提供 Blazor 应用。

独立部署资产将发布到/bin/Release/{TARGET FRAMEWORK}/publish/wwwroot 文件夹。

## （一）Azure 应用服务

可以将 Blazor WebAssembly 应用部署到 Windows 上的 Azure 应用服务，该服务在 [IIS](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/host-and-deploy/webassembly?view=aspnetcore-5.0" \l "iis) 上托管应用。

目前不支持将独立的 Blazor WebAssembly 应用部署到适用于 Linux 的 Azure 应用服务。 目前无法提供用于托管应用的 Linux 服务器映像。 正在进行此工作以支持此场景。

## （二）IIS

IIS 是适用于 Blazor 应用的强大静态文件服务器。 要配置 IIS 以托管 Blazor，请参阅[在 IIS 上生成静态网站](https://docs.microsoft.com/zh-cn/iis/manage/creating-websites/scenario-build-a-static-website-on-iis)。

/bin/Release/{TARGET FRAMEWORK}/publish 文件夹中已创建发布的资产。 在 Web 服务器或托管服务上托管 publish 文件夹的内容。

## （三）web.config

发布 Blazor 项目时，将使用以下 IIS 配置创建 web.config 文件：

1.对以下文件扩展名设置 MIME 类型：

（1）.dll: application/octet-stream

（2）.json: application/json

（3）.wasm: application/wasm

（4）.woff: application/font-woff

（5）.woff2: application/font-woff

2.对以下 MIME 类型启用 HTTP 压缩：

（1）application/octet-stream

（2）application/wasm

3.建立 URL 重写模块规则：

（1）提供应用的静态资产所驻留的子目录 (wwwroot/{PATH REQUESTED})。

（2）创建 SPA 回退路由，以便非文件资产请求能够重定向到应用的静态资产文件夹中的默认文档 (wwwroot/index.html)。

## （四）使用自定义 web.config

要使用自定义 web.config 文件，请将自定义 web.config 文件置于项目文件夹的根目录下。 将项目配置为使用应用项目文件中的 PublishIISAssets 发布特定于 IIS 的资源，并发布项目：

<PropertyGroup>

<PublishIISAssets>true</PublishIISAssets>

</PropertyGroup>

**安装 URL 重写模块**

重写 URL 必须使用 [URL 重写模块](https://www.iis.net/downloads/microsoft/url-rewrite)。 此模块默认不安装，且不适用于安装为 Web 服务器 (IIS) 角色服务功能。 必须从 IIS 网站下载该模块。 使用 Web 平台安装程序安装模块：

1.以本地方式导航到 [URL 重写模块下载页](https://www.iis.net/downloads/microsoft/url-rewrite" \l "additionalDownloads)。 对于英语版本，请选择“WebPI”以下载 WebPI 安装程序。 对于其他语言，请选择适当的服务器体系结构 (x86/x64) 下载安装程序。

2.将安装程序复制到服务器。 运行安装程序。 选择“安装”按钮，并接受许可条款。 安装完成后无需重启服务器。

**配置网站**

将网站的物理路径设置为应用的文件夹。 该文件夹包含：

1.web.config 文件，IIS 使用该文件配置网站，包括所需的重定向规则和文件内容类型。

2.应用的静态资产文件夹。

作为 IIS 子应用托管，如果独立应用作为 IIS 子应用托管，请执行下列任一操作：

1.禁用继承的 ASP.NET Core 模块处理程序，通过向文件添加 <handlers> 部分，删除 Blazor 应用的已发布 web.config 文件中的处理程序：

<handlers>

<remove name="aspNetCore" />

</handlers>

2.通过使用 <location> 元素并且将 inheritInChildApplications 设置为 false，禁止继承根（父级）应用的 <system.webServer> 部分：

<hxml version="1.0" encoding="utf-8"?>

<configuration>

<location path="." inheritInChildApplications="false">

<system.webServer>

<handlers>

<add name="aspNetCore" ... />

</handlers>

<aspNetCore ... />

</system.webServer>

</location>

</configuration>

除[配置应用的基路径](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/host-and-deploy/?view=aspnetcore-5.0" \l "app-base-path)外，还需删除处理程序或禁用继承。 在 IIS 中配置子应用时，在应用的 index.html 文件中将应用基路径设置为 IIS 别名。

## （五）Brotli 和 Gzip 压缩

托管的 Blazor 应用使用默认的 ASP.NET Core 应用 web.config 文件，而不使用本部分中所链接的文件。

通过 web.config 可将 IIS 配置为提供独立 Blazor WebAssembly 应用的 Brotli 或 Gzip 压缩的 Blazor 资产。 若要查看示例配置文件，请参阅 [web.config](https://github.com/dotnet/AspNetCore.Docs/blob/main/aspnetcore/blazor/host-and-deploy/webassembly/_samples/web.config?raw=true)。

在以下情况下，可能需要进一步配置示例web.config 文件：

1.应用的规范具有以下任意一个要求：

2.提供不是由示例 web.config 文件配置的压缩文件，采用非压缩格式提供由示例 web.config 文件配置的压缩文件。

3.服务器的 IIS 配置（例如 applicationHost.config）提供了服务器级 IIS 默认值。 根据服务器级别配置，应用可能要求 IIS 配置不同于示例 web.config 文件所包含的配置。

## （六）Azure 存储

[Azure 存储](https://docs.microsoft.com/zh-cn/azure/storage/)静态文件承载允许无服务器的 Blazor 应用承载。 支持自定义域名、Azure 内容分发网络 (CDN) 以及 HTTPS。

为存储帐户上的静态网站承载启用 blob 服务时：

1.设置“索引文档名称”到 index.html。

2.设置“错误文档路径”到 index.html。 Razor 组件和其他非文件终结点不会驻留在由 blob 服务存储的静态内容中的物理路径中。 当收到 Blazor 路由器应处理的对这些资源之一的请求时，由 blob 服务生成的“404 - 未找到”错误会将此请求路由到“错误文档路径”。 返回 index.html blob，Blazor 路由器会加载并处理此路径。

如果由于文件的 Content-Type 标头中的 MIME 类型不正确，导致在运行时未加载文件，请执行以下任一操作：

1.配置工具，用于在部署文件时设置正确的 MIME 类型（Content-Type 标头）。

2.在部署应用后更改文件的 MIME 类型（Content-Type 标头）。

在每个文件的存储资源管理器（Azure 门户）中，执行以下操作：

1.右键单击该文件并选择“属性”。

2.设置“ContentType”并选择“保存”按钮 。

## （七）Nginx

以下 nginx.conf 文件已简化，以展示如何配置 Nginx，以便每当在磁盘上找不到相应文件时，就发送 index.html 文件。

events { }

http {

server {

listen 80;

location / {

root /usr/share/nginx/html;

try\_files $uri $uri/ /index.html =404;

}

}

}

使用 [limit\_req](https://nginx.org/docs/http/ngx_http_limit_req_module.html" \l "limit_req) 设置 [NGINX 突发速率限制](https://www.nginx.com/blog/rate-limiting-nginx/" \l "bursts)时，Blazor WebAssembly 应用可能需要一个较大的 burst 参数值来容纳应用发出的请求数。 首先，将值设置为不低于 60：

http {

server {

...

location / {

...

limit\_req zone=one burst=60 nodelay;

}

}

}

如果浏览器开发人员工具或网络流量工具指示请求收到“503 - 服务不可用”状态代码，则将该值调高。

有关生产 Nginx Web 服务器配置的详细信息，请参阅 [Creating NGINX Plus and NGINX Configuration Files](https://docs.nginx.com/nginx/admin-guide/basic-functionality/managing-configuration-files/)（创建 NGINX 增强版和 NGINX 配置文件）。

## （八）Apache

若要将 Blazor WebAssembly 应用部署到 CentOS 7 或更高版本，请执行以下操作：

1.创建 Apache 配置文件。 下面的示例展示了一个简化的配置文件 (blazorapp.config)：

<VirtualHost \*:80>

ServerName www.example.com

ServerAlias \*.example.com

DocumentRoot "/var/www/blazorapp"

ErrorDocument 404 /index.html

AddType application/wasm .wasm

AddType application/octet-stream .dll

<Directory "/var/www/blazorapp">

Options -Indexes

AllowOverride None

</Directory>

<IfModule mod\_deflate.c>

AddOutputFilterByType DEFLATE text/css

AddOutputFilterByType DEFLATE application/javascript

AddOutputFilterByType DEFLATE text/html

AddOutputFilterByType DEFLATE application/octet-stream

AddOutputFilterByType DEFLATE application/wasm

<IfModule mod\_setenvif.c>

BrowserMatch ^Mozilla/4 gzip-only-text/html

BrowserMatch ^Mozilla/4.0[678] no-gzip

BrowserMatch bMSIE !no-gzip !gzip-only-text/html

</IfModule>

</IfModule>

ErrorLog /var/log/httpd/blazorapp-error.log

CustomLog /var/log/httpd/blazorapp-access.log common

</VirtualHost>

2.将 Apache 配置文件放入 /etc/httpd/conf.d/ 目录（这是 CentOS 7 中的默认 Apache 配置目录）。

3.将应用的文件放入 /var/www/blazorapp 目录（配置文件中特定于 DocumentRoot 的位置）。

4.重启 Apache 服务。

## （九）GitHub

要处理 URL 重写，请使用脚本添加 wwwroot/404.html 文件，该脚本可处理到 index.html 页的重定向请求。

如果使用项目站点而非组织站点，请在 wwwroot/index.html 中更新 <base> 标记。 将 href 属性值设置为，包含尾部斜杠的 GitHub 存储库名称（例如，/my-repository/）。

在[SteveSandersonMS/BlazorOnGitHubPages GitHub 存储库](https://github.com/SteveSandersonMS/BlazorOnGitHubPages)中，将在发布时通过 [.github/workflows/main.yml 配置文件](https://github.com/SteveSandersonMS/BlazorOnGitHubPages/blob/master/.github/workflows/main.yml)更新基本 href。

[SteveSandersonMS/BlazorOnGitHubPages GitHub 存储库](https://github.com/SteveSandersonMS/BlazorOnGitHubPages)不归 .NET Foundation 或 Microsoft 所有，也不由它们提供维护和支持。

# 第三部分 主机配置值

在开发环境中，[Blazor WebAssembly 应用](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/hosting-models?view=aspnetcore-5.0" \l "blazor-webassembly)可以在运行时接受以下主机配置值作为命令行参数。

## （一）内容根

--contentroot 参数设置包含应用内容文件的目录的绝对路径（[内容根目录](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/?view=aspnetcore-5.0" \l "content-root)）。 在下面的示例中，/content-root-path 是应用的内容根路径。

1.以本地方式在命令提示符下运行应用时传递该参数。 在应用的目录中，执行以下操作：

dotnet run --contentroot=/content-root-path

2.在 IIS Express 配置文件中，向应用的 launchSettings.json 文件添加条目。 如果使用 Visual Studio 调试器并在命令提示符中运行 dotnet run 来运行应用，使用的是此设置。

"commandLineArgs": "--contentroot=/content-root-path"

3.在 Visual Studio 中，在“属性” > “调试” > “应用程序参数”中指定参数 。 在 Visual Studio 属性页中设置参数可将参数添加到 launchSettings.json 文件。

--contentroot=/content-root-path

## （二）基路径

--pathbase 参数可设置使用非根相对 URL 路径本地运行的应用的应用基路径（将 <base> 标记 href 针对暂存和生产设置为 / 之外的路径）。 在下面的示例中，/relative-URL-path 是应用的基路径。 有关详细信息，请参阅[应用基路径](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/host-and-deploy/?view=aspnetcore-5.0" \l "app-base-path)。

不同于向 href 标记的 <base> 提供的路径，传递 --pathbase 参数值时不包括尾部反斜杠 (/)。 如果在 <base> 标记中以 <base href="/CoolApp/"> 形式（包括尾部反斜杠）提供应用基路径，则以 --pathbase=/CoolApp 形式（无尾部反斜杠）传递命令行参数值。

1.以本地方式在命令提示符下运行应用时传递该参数。 在应用的目录中，执行以下操作：

dotnet run --pathbase=/relative-URL-path

2.在 IIS Express 配置文件中，向应用的 launchSettings.json 文件添加条目。 如果使用 Visual Studio 调试器并在命令提示符中运行 dotnet run 来运行应用，使用的是此设置。

"commandLineArgs": "--pathbase=/relative-URL-path"

3.在 Visual Studio 中，在“属性” > “调试” > “应用程序参数”中指定参数 。 在 Visual Studio 属性页中设置参数可将参数添加到 launchSettings.json 文件。

--pathbase=/relative-URL-path

## （三）URL

--urls 参数设置 IP 地址或主机地址，其中包含侦听请求的端口和协议。

1.以本地方式在命令提示符下运行应用时传递该参数。 在应用的目录中，执行以下操作：

dotnet run --urls=http://127.0.0.1:0

2.在 IIS Express 配置文件中，向应用的 launchSettings.json 文件添加条目。 如果使用 Visual Studio 调试器并在命令提示符中运行 dotnet run 来运行应用，使用的是此设置。

"commandLineArgs": "--urls=http://127.0.0.1:0"

3.在 Visual Studio 中，在“属性” > “调试” > “应用程序参数”中指定参数 。 在 Visual Studio 属性页中设置参数可将参数添加到 launchSettings.json 文件。

--urls=http://127.0.0.1:0

## （四）配置裁边器

Blazor 对每个发布版本执行中间语言 (IL) 剪裁，以从输出程序集中删除不必要的 IL。有关详细信息，请参阅 [配置适用于 ASP.NET Core Blazor 的裁边器](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/host-and-deploy/configure-trimmer?view=aspnetcore-5.0)。

## （五）更改 DLL 文件的文件扩展名

如果需要更改应用的已发布 .dll 文件的文件扩展名，请按照本部分中的指导进行操作。

发布应用后，使用 shell 脚本或 DevOps 生成管道将 .dll 文件重命名，以使用其他文件扩展名。 将 .dll 文件的目标位置设为应用的已发布输出的wwwroot目录中（例如{CONTENT ROOT}/bin/Release/netstandard2.1/publish/wwwroot）。

在下面的示例中，重命名 .dll 文件，以使用 .bin 文件扩展名。

**在 Windows 上：**

dir .\\_framework\\_bin | rename-item -NewName { $\_.name -replace ".dll\b",".bin" }

((Get-Content .\\_framework\blazor.boot.json -Raw) -replace '.dll"','.bin"') | Set-Content .\\_framework\blazor.boot.json

如果服务工作进程资产也在使用中，请添加以下命令：

((Get-Content .\service-worker-assets.js -Raw) -replace '.dll"','.bin"') | Set-Content .\service-worker-assets.js

**在 Linux 或 macOS 上：**

for f in \_framework/\_bin/\*; do mv "$f" "`echo $f | sed -e 's/\.dll/.bin/g'`"; done sed -i 's/\.dll"/.bin"/g' \_framework/blazor.boot.json

如果服务工作进程资产也在使用中，请添加以下命令：

sed -i 's/\.dll"/.bin"/g' service-worker-assets.js

要使用不同于 .bin 的其他文件扩展名，请在前面的命令中替换 .bin。

若要处理压缩的 blazor.boot.json.gz 和 blazor.boot.json.br 文件，请采用以下方法之一：

1.删除压缩的 blazor.boot.json.gz 和 blazor.boot.json.br 文件。此方法禁用压缩。

2.重新压缩更新后的 blazor.boot.json 文件。

上述指导也适用于正在使用服务工作进程资产的情况。 删除或重新压缩 wwwroot/service-worker-assets.js.br 和 wwwroot/service-worker-assets.js.gz。 否则，浏览器中的文件完整性检查将失败。

以下 Windows 示例使用项目根目录中的 PowerShell 脚本。

ChangeDLLExtensions.ps1::

PowerShell复制

param([string]$filepath,[string]$tfm)

dir $filepath\bin\Release\$tfm\wwwroot\\_framework\\_bin | rename-item -NewName { $\_.name -replace ".dll\b",".bin" }

((Get-Content $filepath\bin\Release\$tfm\wwwroot\\_framework\blazor.boot.json -Raw) -replace '.dll"','.bin"') | Set-Content $filepath\bin\Release\$tfm\wwwroot\\_framework\blazor.boot.json

Remove-Item $filepath\bin\Release\$tfm\wwwroot\\_framework\blazor.boot.json.gz

如果服务工作进程资产也在使用中，请添加以下命令：

((Get-Content $filepath\bin\Release\$tfm\wwwroot\service-worker-assets.js -Raw) -replace '.dll"','.bin"')|Set-Content $filepath\bin\Release\$tfm\wwwroot\service-worker-assets.js

在项目文件中，在发布应用后运行脚本：

<Target Name="ChangeDLLFileExtensions" AfterTargets="Publish" Condition="'$(Configuration)'=='Release'">

<Exec Command="powershell.exe -command &quot;&amp; { .\ChangeDLLExtensions.ps1 '$(SolutionDir)' '$(TargetFramework)'}&quot;" />

</Target>

# 第四部分 解决完整性检查失败

当 Blazor WebAssembly 下载应用的启动文件时，它会指示浏览器对响应执行完整性检查。 它使用 blazor.boot.json 文件中的信息为 .dll、.wasm 和其他文件指定预期的 SHA-256 哈希值。 这对以下原因有所帮助：

1.它可确保不会出现加载不一致文件集的风险，例如，在用户正在下载应用程序文件时将新部署应用到 Web 服务器的情况。 不一致的文件可能导致未定义的行为。

2.它可确保用户的浏览器从不缓存不一致或无效的响应，这些响应可能会阻止他们启动应用（即使他们手动刷新了页面也是如此）。

3.它可以安全地缓存响应，甚至无需检查服务器端更改，直到预期的 SHA-256 哈希本身发生更改，因此，后续页面加载需要较少的请求即可快速完成。

如果Web 服务器返回的响应与预期的 SHA-256 哈希不匹配，将看到类似于以下内容的错误显示在浏览器的开发人员控制台中：

未能使用计算出的SHA-256完整性“IIa70iwvmEg5WiDV17OpQ5eCztNYqL186J56852RpJY=”在资源“https://myapp.example.com/\_framework/MyBlazor App.dll”的“integrity”属性中找到有效摘要。 资源已被阻止。

在大多数情况下，这不是完整性检查本身的问题。 相反，它表示存在其他问题，并且完整性检查会警告其他问题。

## （一）诊断完整性问题

生成应用时，生成的 blazor.boot.json 清单将描述生成输出生成时启动资源（例如，.dll、.wasm 和其他文件）的 SHA-256 哈希。 只要 blazor.boot.json 中的 SHA-256 哈希与传递到浏览器的文件相匹配，完整性检查就会通过。

**此失败的常见原因包括：**

1.Web 服务器的响应是一个错误（例如，“404 - 找不到”或“500 - 内部服务器错误”），而不是浏览器所请求的文件。 浏览器会将其报告为完整性检查失败，而不是响应失败。

2.在文件生成和传递到浏览器之间已更改文件的内容。 下面可能会发生这种情况：

（1）你或生成工具手动修改生成输出的情况。

（2）部署过程的某个方面修改了文件的情况。 例如，在使用基于 Git 的部署机制时，请记住，如果你在 Windows 上提交文件并在 Linux 上检查它们，则 Git 会以透明方式将 Windows 样式的行尾转换为 Unix 样式的行尾。 更改文件行尾将更改 SHA-256 哈希。 若要避免此问题，请考虑[使用 .gitattributes 将生成项目视为 binary 文件](https://git-scm.com/book/en/v2/Customizing-Git-Git-Attributes)。

（3）Web 服务器在提供文件内容的过程中对其进行修改。 例如，某些内容分发网络 (CDN) 会自动尝试[缩小](https://docs.microsoft.com/zh-cn/aspnet/core/client-side/bundling-and-minification?view=aspnetcore-5.0" \l "minification) HTML，从而对其进行修改。 可能需要禁用此类功能。

若要诊断哪些功能适用，请执行执行操作：

1.通过读取错误消息来记下哪个文件触发错误。

2.打开浏览器的开发人员工具，然后在“网络”选项卡中查找。如有必要，请重新加载页面以查看请求和响应的列表。 在该列表中查找触发错误的文件。

3.检查响应中的 HTTP 状态代码。 如果服务器返回除“200 - 正常”（或其他 2xx 状态代码）以外的任何内容，则需要诊断服务器端问题。 例如，状态代码 403 表示存在授权问题，而状态代码 500 表示服务器以未指定的方式失败。 请参阅服务器端日志以诊断和修复应用。

4.如果资源的状态代码为“200 - 正常”，请在浏览器的开发人员工具中查看响应内容，并检查内容是否与预期的数据匹配。 例如，常见问题是错误配置了路由，因此请求甚至返回其他文件的 index.html 数据。 请确保对 .wasm 请求的响应是 WebAssembly 二进制文件，对 .dll 请求的响应是 .NET 程序集二进制文件。 如果不是，则需要诊断服务器端路由问题。

5.搜索以验证应用的已发布和已部署输出，并提供[完整性 PowerShell 脚本故障排除](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/host-and-deploy/webassembly?view=aspnetcore-5.0" \l "troubleshoot-integrity-powershell-script)。

如果确认服务器返回看似正确的数据，则必须在生成文件和传递文件之间修改内容。 若要对此进行调查，请执行以下操作：

1.如果在生成文件后修改文件，请检查生成工具链和部署机制。 例如，在 Git 转换文件行尾时，如前所述。

2.如设置为动态修改响应（例如，尝试缩小 HTML），请检查 Web 服务器或 CDN 配置。 Web 服务器可以实现 HTTP 压缩（例如，返回 content-encoding: br 或 content-encoding: gzip），因为这不会影响解压缩后的结果。 但是，Web 服务器不可以修改未压缩的数据。

完整性 PowerShell 脚本故障排除使用 [integrity.ps1](https://github.com/dotnet/AspNetCore.Docs/blob/main/aspnetcore/blazor/host-and-deploy/webassembly/_samples/integrity.ps1?raw=true) PowerShell 脚本来验证已发布和已部署的 Blazor 应用。 当应用出现 Blazor 框架无法识别的完整性问题时，该脚本将作为起点提供。 您的应用可能需要自定义脚本。

此脚本将检查 publish 文件夹中的文件，并从部署的应用中下载这些文件，以检测包含完整性哈希的不同清单中的问题。 这些检查应检测最常见的问题：

1.修改了已发布的输出中的文件，但未实现它。

2.应用未正确部署到部署目标，或者在部署目标的环境中发生了更改。

3.部署的应用与发布应用的输出之间存在差异。

在 PowerShell 命令行中使用以下命令调用脚本：

.\integrity.ps1 {BASE URL} {PUBLISH OUTPUT FOLDER}

占位符：

1.{BASE URL}：已部署的应用的 URL。

2.{PUBLISH OUTPUT FOLDER}：应用的 publish 文件夹的路径，或已发布的用于部署应用的位置。

若要将 dotnet/AspNetCore.Docs GitHub 存储库克隆到使用 [Bitdefender](https://www.bitdefender.com/) 病毒扫描程序的系统，请向 integrity.ps1 脚本的 Bitdefender 添加例外。 在克隆存储库之前，请将例外添加到 Bitdefender，以避免病毒扫描程序隔离该脚本。 以下示例是 Windows 系统上已克隆的存储库的脚本的典型路径。 请根据需要调整路径。 占位符 {USER} 是用户的路径段。

C:\Users\{USER}\Documents\GitHub\AspNetCore.Docs\aspnetcore\blazor\host-and-deploy\webassembly\\_samples\integrity.ps1

## （二）禁用非PWA应用的完整性检查

在大多数情况下，不要禁用完整性检查。 禁用完整性检查并不能解决导致意外响应的根本问题，并且会导致丢失前面列出的权益。

在某些情况下，Web 服务器无法用于返回一致的响应，但别无选择，只能禁用完整性检查。 若要禁用完整性检查，请将以下内容添加到 Blazor WebAssembly 项目的 .csproj 文件中的属性组：

<BlazorCacheBootResources>false</BlazorCacheBootResources>

BlazorCacheBootResources 还会根据 SHA-256 哈希禁用 Blazor 缓存 .dll、.wasm 和其他文件的默认行为，因为属性指示无法依靠 SHA-256 哈希来确保正确性。 即使有此设置，浏览器的普通 HTTP 缓存仍可能会缓存这些文件，但是否发生这种情况取决于你的 Web 服务器配置和它所提供的 cache-control 标头。

BlazorCacheBootResources 属性不会禁用[渐进式 Web 应用程序 (PWA)](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/progressive-web-app?view=aspnetcore-5.0) 的完整性检查。 有关 PWA 的相关指南，请参阅[禁用 PWA 的完整性检查](https://docs.microsoft.com/zh-cn/aspnet/core/blazor/host-and-deploy/webassembly?view=aspnetcore-5.0" \l "disable-integrity-checking-for-pwas)部分。

## （三）禁用PWA的完整性检查

Blazor 的渐进式 Web 应用程序 (PWA) 模板包含建议的 service-worker.published.js 文件，该文件负责获取和存储应用程序文件以供脱机使用。 这是普通应用启动机制的独立进程，具有其自己单独的完整性检查逻辑。

在 service-worker.published.js 文件中，出现以下行：

.map(asset => new Request(asset.url, { integrity: asset.hash }));

若要禁用完整性检查，请通过将该行更改为以下行来删除 integrity 参数：

.map(asset => new Request(asset.url));

同样，禁用完整性检查意味着会丢失完整性检查提供的安全保证。 例如，如果用户的浏览器在你部署新版本的那一刻缓存应用，则会存在风险，它可能会缓存旧部署中的某些文件和新部署中的某些文件。 如果发生这种情况，则在部署进一步更新之前，应用程序会在中断状态下停滞。