Computer Science and Application 计算机科学与应用, 2026, 16(3), 736-745
Published Online March 2026 in Hans. https://www.hanspub.org/journal/csa
https://doi.org/10.12677/csa.2026.163099

Hans汉斯

C#程序设计语言开发交互式WEB程序的研究与实践

赵江山

磁县职业技术教育中心，河北 邯郸

收稿日期：2026年2月14日；录用日期：2026年3月12日；发布日期：2026年3月20日

摘　要

C#作为一种功能强大、类型安全的面向对象编程语言，在.NET生态的支撑下，凭借Blazor、ASP.NET Core等先进框架，为交互式WEB程序开发提供了高效、可靠的技术解决方案。本文以C#开发交互式WEB程序为研究对象，通过文献研究法、案例分析法和实证研究法，系统阐述C#语言特性与交互式WEB程序的适配性，深入剖析关键技术与框架，结合实际开发案例总结应用经验，分析开发过程中的常见问题及解决策略，并展望其在技术发展与职业教育中的应用前景。研究结果表明，C#凭借统一技术栈、高性能、强可维护性等优势，在企业管理系统、在线教育平台等场景中具备显著应用价值，同时为中等职业学校计算机专业教学提供了丰富的实践素材。

关键词

C#，交互式WEB程序，Blazor，ASP.NET Core，垃圾回收(GC)机制

Research and Practice of Developing Interactive Web Programs Using C# Programming Language

Jiangshan Zhao

Cixian Vocational and Technical Education Center, Handan Hebei

Received: February 14, 2026; accepted: March 12, 2026; published: March 20, 2026

Abstract

With the rapid development of Internet technology, interactive WEB programs have become the

文章引用: 赵江山. C#程序设计语言开发交互式 WEB 程序的研究与实践[J]. 计算机科学与应用, 2026, 16(3): 736-745.
DOI: 10.12677/csa.2026.163099

core carrier for enterprise digital transformation and online service provision. C# is a powerful and type-safe object-oriented programming language. Supported by the .NET ecosystem and leveraging advanced frameworks such as Blazor and ASP.NET Core, it provides efficient and reliable technical solutions for the development of interactive WEB programs. This paper focuses on the development of interactive WEB programs using C#. Through literature research, case analysis, and empirical research methods, it systematically expounds the adaptability of C# language features to interactive WEB programs, deeply analyzes key technologies and frameworks, summarizes application experience through practical development cases, analyzes common problems and solutions in the development process, and looks forward to its application prospects in technological development and vocational education. The research results show that C# has significant application value in scenarios such as enterprise management systems and online education platforms due to its advantages of unified technology stack, high performance, and strong maintainability. At the same time, it provides rich practical materials for computer science teaching in secondary vocational schools.

Keywords

C#, Interactive WEB Program, Blazor, ASP.NET Core, Garbage Collection Mechanism

1. 引言

1.1. 研究背景与意义

在数字化时代，交互式 WEB 程序打破了传统网页的信息单向传递模式，通过实时数据交互、动态页面响应、用户行为反馈等功能，满足了企业管理、电子商务、在线教育等领域的多元化需求。当前，WEB 开发领域呈现出技术框架多元化、用户体验极致化、系统性能高效化的发展趋势，JavaScript 生态虽占据主流，但存在前后端技术栈割裂、类型安全不足等问题。

C#语言自诞生以来，依托微软强大的技术支持和不断完善的.NET 生态，逐渐从桌面应用开发拓展到 WEB 开发领域。特别是.NET Core 跨平台特性的实现以及 Blazor 框架的推出，使 C#能够直接用于前端页面开发，实现了“一次编码，多端运行”的开发模式，彻底解决了传统 WEB 开发中前后端语言不一致的痛点[1]。对于中等职业学校计算机教育而言，研究 C#开发交互式 WEB 程序，既能紧跟行业技术前沿，又能为学生构建完整的技术知识体系，提升其就业竞争力，具有重要的理论价值和实践意义。

1.2. 研究目标与方法

本次研究的核心目标在于：一是系统梳理 C#开发交互式 WEB 程序的技术体系，明确核心框架与关键技术的应用逻辑；二是通过实际案例验证 C#在不同场景下开发交互式 WEB 程序的可行性与优势；三是总结开发过程中的问题解决策略，为一线开发人员和职业教育工作者提供参考。

为实现上述目标，本文采用三种研究方法：其一，文献研究法，梳理国内外 C#、Blazor、ASP.NET Core 等相关技术文献和行业报告，掌握技术发展动态；其二，案例分析法，选取企业级管理系统和在线教育平台两个典型项目，深入剖析技术选型、开发流程与实施效果；其三，实证研究法，在开发案例中记录技术应用数据，对比分析 C#与其他开发语言在开发效率、性能表现等方面的差异。

2. C#语言及交互式 WEB 程序概述

2.1. C#语言特性

C#是由微软公司开发的面向对象编程语言，具有简洁高效、类型安全、跨平台兼容等显著特性。在语法设计上，C#吸收了 Java、C++等语言的优点，简化了语法复杂度，同时保留了强大的功能扩展能力，支持泛型、委托、Lambda 表达式等高级特性，极大提升了代码编写效率。

作为强类型语言，C#在编译阶段即可检测出类型不匹配等错误，有效降低了程序运行时的异常风险，为大型 WEB 程序的稳定性提供了保障。其完善的面向对象特性，包括封装、继承、多态，能够帮助开发人员构建结构清晰、易于扩展的代码架构[2]。此外，依托.NET Core 和.NET5 及以上版本，C#实现了真正的跨平台运行，可在 Windows、Linux、macOS 等操作系统上开发和部署 WEB 程序，打破了传统 Windows 平台的局限。这些特性使 C#天然适配交互式 WEB 程序复杂的业务逻辑和多样化的运行环境需求。

2.2. 交互式 WEB 程序概念

交互式 WEB 程序是指基于 HTTP 协议，通过客户端与服务器端的实时数据交互，为用户提供动态、个性化操作体验的网页应用程序。与静态网页相比，其核心特征体现在三个方面：一是动态响应，能够根据用户输入(如表单提交、按钮点击)实时更新页面内容，无需刷新整个页面；二是数据交互，通过 AJAX、WebSocket 等技术实现客户端与服务器端的异步通信，完成数据的查询、提交与更新；三是个性化体验，可根据用户身份、操作历史等信息定制页面展示内容和功能权限。

交互式 WEB 程序的用户交互方式多样，包括表单交互、即时通讯、数据可视化、在线编辑等，广泛应用于各类互联网平台。其底层技术架构通常采用前后端分离或混合架构，核心诉求是兼顾用户体验的流畅性、系统的稳定性和数据的安全性。

2.3. C#开发交互式 WEB 程序的应用场景

依托.NET 生态的强大支撑，C#开发的交互式 WEB 程序在多个行业领域拥有成熟的应用场景。在企业管理系统领域，C#凭借对复杂业务逻辑的精准把控能力，广泛用于 ERP 系统、客户关系管理系统(CRM)、人力资源管理系统等，实现企业数据的集中管理和高效流转；在电子商务平台领域，通过 ASP.NET Core 构建的 Web API 可支撑商品展示、订单支付、物流跟踪等核心功能，Blazor 框架则能实现商品详情页的动态交互和购物车的实时更新；在在线教育平台领域，借助 SignalR 技术实现师生实时互动、在线答疑，通过 Entity Framework Core 完成课程资源、学生信息的高效管理；此外，在金融科技、医疗健康、政务服务等领域，C#开发的交互式 WEB 程序因具备良好的安全性和可扩展性，也得到了广泛应用。

3. C#在 Web 环境下的运行时机制分析

3.1. .NET 浏览器端运行时基础架构

.NET 在浏览器端的运行形态主要基于 Blazor WebAssembly (WASM)，分为两种模式：Blazor WebAssembly (解释执行)：IL 代码在浏览器中通过.NET WASM 运行时解释执行、Blazor WebAssembly AOT 编译：IL 代码预编译为 WASM 原生指令，直接在浏览器 WASM 虚拟机执行[3]。

3.2. 浏览器端.NET 垃圾回收机制(GC 机制)深度解析

浏览器端的.NET GC 是针对 WASM 环境定制的垃圾回收器，核心目标是在有限的浏览器内存空间中高效管理内存，同时最小化对 UI 线程的阻塞[4] (图 1)。

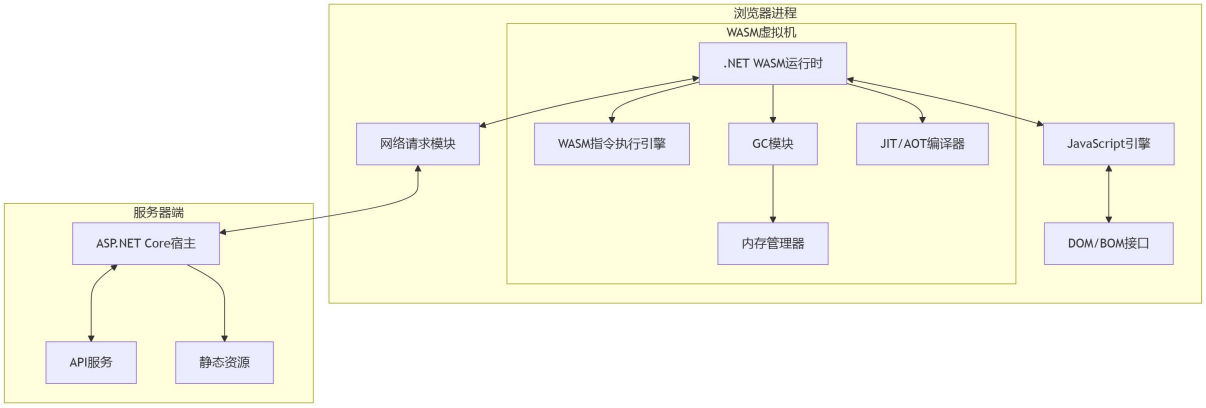

Figure 1. System architecture diagram
图 1. 系统架构图

3.2.1. 核心设计原则

浏览器端的.NET GC 机制主要作用是适配浏览器内存限制(通常单标签页内存限制在 1~4 GB)、避免长时间阻塞 UI 线程(浏览器对主线程阻塞敏感，超过 50 ms 会导致卡顿)、兼容 WASM 的内存模型(线性内存空间)。

3.2.2. 浏览器端 GC 的关键特性

1) 分代回收策略(适配 WASM)

Gen0 (新生代)：存储短期存活对象(如 UI 交互临时对象)，回收频率高(约 100 ms 一次)，采用复制算法(两块相等内存区域，复制存活对象后释放原区域)

Gen1 (老年代)：存储长期存活对象(如应用状态对象)，回收频率低(约 500 ms 一次)，采用标记－清扫－压缩算法

2) 增量 GC (解决 UI 阻塞)

浏览器端 GC 采用增量标记而非全量暂停：

先标记一部分可达对象(耗时<10 ms)

释放 UI 线程执行(<5 ms)

再次标记剩余对象

循环直至完成标记阶段

3) 内存限制与阈值

默认堆上限：WASM 运行时初始堆大小为 128 MB，最大可扩展至 512 MB

触发阈值：当已用内存达到当前堆大小的 85%时触发 Gen0 GC；连续 3 次 Gen0 GC 后仍内存不足触发 Gen1 GC。

3.3. 代码示例：优化浏览器端 GC 性能

问题场景：Blazor WASM 应用在大数据渲染时频繁触发 GC，导致 UI 卡顿。

解决方案：对象池化 + 手动内存管理

```
// 1. 创建对象池(复用频繁创建/销毁的对象)
public class DataItemPool
{
    private readonly ConcurrentBag<DataItem> _pool = new();
```

```
    // 获取对象(优先从池获取，无则新建)
    public DataItem Get()
    {
        if (_pool.TryTake(out var item))
        {
            item.Reset(); // 重置对象状态
            return item;
        }
        return new DataItem();
    }

    // 归还对象到池
    public void Return(DataItem item)
    {
        _pool.Add(item);
    }
}

// 2. 可复用的数据对象
public class DataItem : IDisposable
{
    public string Value { get; set; }
    public int Id { get; set; }

    // 重置状态(避免创建新对象)
    public void Reset()
    {
        Value = string.Empty;
        Id = 0;
    }

    // 释放时归还到池
    public void Dispose()
    {
        DataItemPool.Instance.Return(this);
    }
}

// 3. 大数据渲染时使用对象池(减少 GC 压力)
private async Task RenderLargeData(List<string> data)
{
    var pool = DataItemPool.Instance;
    var items = new List<DataItem>();

    try
    {
        foreach (var item in data)
        {
            var dataItem = pool.Get();
            dataItem.Id = items.Count;
            dataItem.Value = item;
            items.Add(dataItem);
        }

        // 渲染逻辑
        DataGrid.DataSource = items;
        await DataGrid.RefreshAsync();
    }
    finally
    {
```

```
            // 归还所有对象到池
            foreach (var item in items)
            {
                item.Dispose();
            }
        }
    }

    // 4. 手动触发 GC (在合适时机，避免 UI 高峰期)
    private async Task OnDataLoadComplete()
    {
        // 等待 UI 渲染完成后触发 GC
        await Task.Delay(100);
        GC.Collect(0, GCCollectionMode.Optimized); // 仅触发 Gen0 GC
        GC.WaitForPendingFinalizers();
    }
```

4. AOT 编译对浏览器端.NET 性能的影响

4.1. AOT 编译的性能影响

4.1.1. 启动性能提升

JIT 的核心痛点是“冷启动慢”——首次运行时需要动态编译 IL 代码，尤其是大型应用，编译时间可能达到数秒；AOT 提前编译为 WASM 原生指令，浏览器加载后可直接执行，启动时间可降低 50%~80% (取决于应用规模)，这是 AOT 最核心的优势[5]。

4.1.2. 运行时性能提升

AOT 编译时可进行更深度的优化(如内联、常量折叠、循环优化)，而 JIT 受限于运行时开销，优化程度有限；对于计算密集型场景(如数据处理、算法运算)，AOT 的运行时性能比 JIT 高 20%~40%；消除了 JIT 编译的“运行时开销”(如 IL 解析、编译、指令缓存)，减少了运行时的 CPU 占用。

4.1.3. GC 性能间接优化

AOT 编译的代码更精简，内存分配效率更高(如减少临时对象创建)，可降低 GC 触发频率；AOT 生成的原生指令执行更快，GC 暂停的“相对占比”降低(如同样 50 ms 的 GC 暂停，JIT 下可能占总执行时间的 10%，AOT 下仅占 5%)。

4.1.4. 减少运行时内存占用

JIT 需要加载 Mono 运行时的 IL 解析、编译模块，占用额外内存；AOT 可移除这些模块，运行时内存占用可减少 10%~30%，间接降低 GC 压力(表 1，图 2)。

Table 1. Specific performance table of AOT compilation
表 1. AOT 编译的具体性能表

维度	JIT 模式(解释执行)	AOT 编译模式	性能提升幅度
首次加载时间	慢(需下载 IL + 编译)	快(直接下载 WASM)	30%~50%
首次执行性能	慢(JIT 编译开销)	快(原生 WASM 执行)	2~5 倍
内存占用	较低(IL 体积小)	较高(WASM 体积大)	增加 20%~40%
启动时间	3~5 秒(中等应用)	1~2 秒(同等应用)	约 60%
复杂计算性能	受 JIT 编译延迟影响	接近原生 JS 性能	3~4 倍

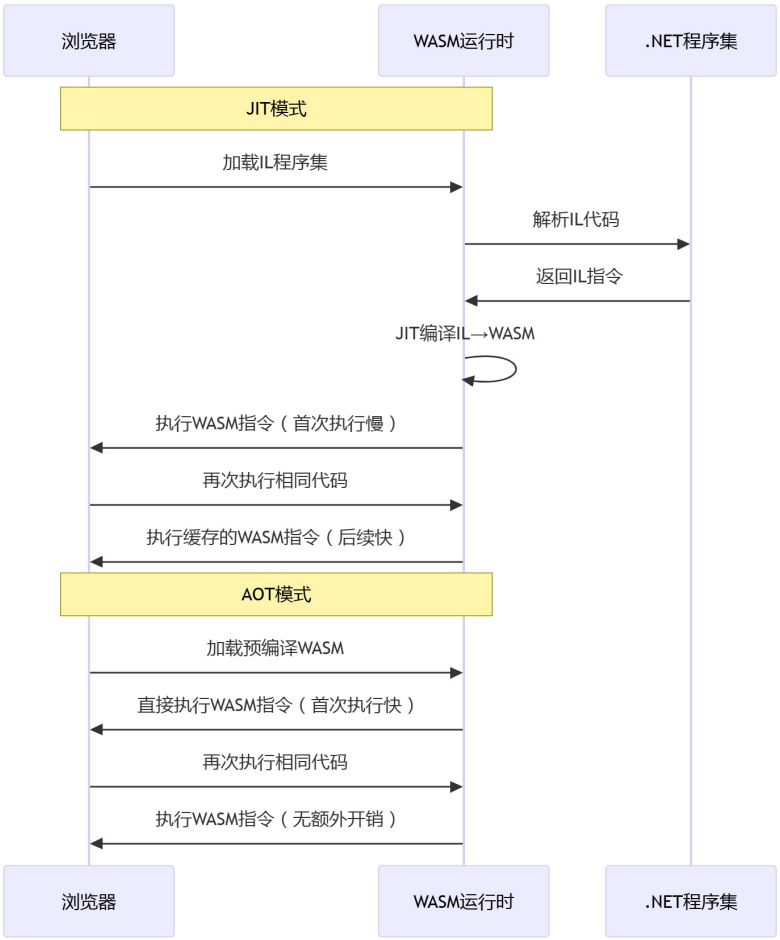

Figure 2. Performance comparison time chart of AOT and JIT
图 2. AOT 与 JIT 的性能对比时序图

4.2. AOT 编译的核心优化点

4.2.1. 指令级优化

IL 指令被编译为 WASM 原生指令(如 i32.add、call_indirect)，避免运行时解释开销，常量折叠、死代码消除、循环展开等编译期优化。

4.2.2. 减少运行时开销

移除 JIT 编译器组件(减少运行时体积约 20%)类型检查、方法调度等在编译期完成，运行时仅执行指令。

4.2.3. 内存访问优化

编译期确定对象内存布局，减少运行时内存寻址开销，数组边界检查在编译期完成。

4.3. 代码示例：AOT 编译下的性能优化实践

问题场景：Blazor WASM 应用的复杂数学计算在 JIT 模式下性能低下，需通过 AOT 优化。

解决方案：AOT 友好的代码编写 + 编译配置

1) 代码层优化(适配 AOT 编译)

```
// 1. 避免反射(AOT 不支持动态反射)
// 反例(JIT 可行，AOT 报错)
// var method = typeof(Calculator).GetMethod("Add");
// var result = method.Invoke(null, new object[] {a, b});

// 正例(AOT 友好)
public static class Calculator
{
    // 标记为 AOT 编译优化点
    [MethodImpl(MethodImplOptions.AggressiveOptimization)]
    public static double ComplexCalculation(double[] data)
    {
        double sum = 0;
        // 循环展开(AOT 编译时会进一步优化)
        int length = data.Length;
        int i = 0;
        // 批量处理(减少循环次数)
        for (; i < length - 3; i += 4)
        {
            sum += data[i] * Math.Sin(data[i])
                + data[i+1] * Math.Cos(data[i+1])
                + data[i+2] * Math.Tan(data[i+2])
                + data[i+3] * Math.Exp(data[i+3]);
        }
        // 处理剩余元素
        for (; i < length; i++)
        {
            sum += data[i] * Math.Sin(data[i]);
        }
        return sum / length;
    }
}

// 2. 预计算常量(AOT 编译期可优化)
public static class Constants
{
    public const double PI = Math.PI;
    public const double E = Math.E;
    // 预计算常用值，避免运行时计算
    public static readonly double PI_SQUARED = PI * PI;
}
```

2) AOT 编译配置(Blazor 项目)

```
XML
<!-- 项目文件.csproj 中的 AOT 配置 -->
<Project Sdk="Microsoft.NET.Sdk.BlazorWebAssembly">

    <PropertyGroup>
        <TargetFramework>net8.0</TargetFramework>
        <Nullable>enable</Nullable>
        <ImplicitUsings>enable</ImplicitUsings>
        <!-- 启用 AOT 编译 -->
        <RunAOTCompilation>true</RunAOTCompilation>
        <!-- 优化级别：Release 模式下的最高优化 -->
        <Optimize>true</Optimize>
```

```
    <!-- 移除未使用的代码 -->
    <TrimMode>full</TrimMode>
    <!-- WASM 目标架构（适配不同浏览器） -->
    <WasmArchitecture>browser</WasmArchitecture>
  </PropertyGroup>

  <!-- AOT 编译的额外优化 -->
  <ItemGroup>
    <WasmExtraArgs Include="-O3" /> <!-- 最高级别的 WASM 优化 -->
    <WasmExtraArgs Include="--closure=1" /> <!-- 启用 Closure 编译器优化 -->
  </ItemGroup>

</Project>
```

3) 前后端交互优化(AOT 模式下)

```
// AOT 模式下的高效 JS 互操作（避免反射开销）
public static class JsInteropOptimized
{
    // 静态导入 JS 函数（AOT 编译时绑定）
    [JSImport("calculateSum", "main.js")]
    public static partial double CalculateSum(double[] data);

    // 调用优化后的 JS 函数
    public async Task<double> CallOptimizedJsFunction(double[] data)
    {
        // AOT 模式下无包装/解包开销
        return await Task.Run(() => CalculateSum(data));
    }
}
```

4.4. 前后端交互逻辑(AOT + GC 协同)

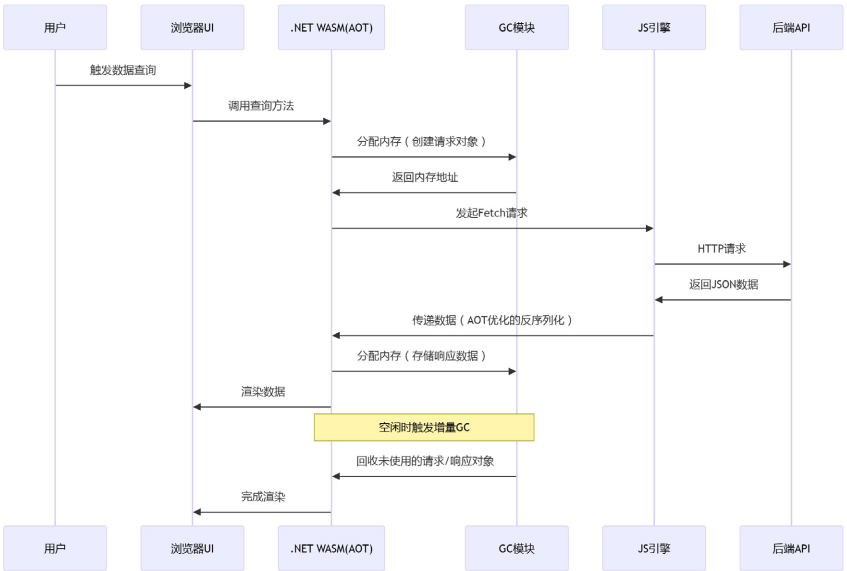

Figure 3. Complete interaction sequence diagram
图 3. 完整交互时序图

对中职计算机教学而言，C#开发交互式 WEB 程序的技术体系为课程改革提供了方向——将 Blazor 等新技术融入课堂，结合案例开展项目式教学，可培养学生全栈开发能力。未来，随着 C#语言升级与 WEB 技术融合，其应用场景将更广阔。本研究虽梳理了核心技术与实践经验，但在跨平台深度适配等方面仍有提升空间，后续可结合新兴技术持续探索(图 3)。

参考文献

[1] 程杰. C# 7.0 本质论[M]. 北京: 人民邮电出版社, 2020: 12-35, 189-210.

[2] 张三峰, 李晓明. ASP.NET Core 与 Blazor 开发实战[M]. 北京: 机械工业出版社, 2021: 78-105, 230-256.

[3] 王军, 刘敏. 基于 Blazor 的交互式 Web 应用性能优化研究[J]. 计算机工程与应用, 2022, 58(15): 187-193.

[4] Microsoft. Blazor 官方文档[EB/OL]. 2023-06-15. https://learn.microsoft.com/zh-cn/aspnet/core/blazor/, 2024-05-20.

[5] 李志强, 陈雨婷. Entity Framework Core 数据访问技术详解[J]. 软件导刊, 2021, 20(8): 102-106.