FineUICore 技术白皮书

全球首创！在 ASP.NET Core 中运行 WebForms 业务代码。

三生石上

合肥三生石上软件有限公司

1. 目录

2. 先说结论

我们为 ASP.NET Core 带来了全新的 WebForms 开发模式，可以让 20 年前的 WebForms 业务代码在最新的 ASP.NET Core 框架中运行，代码相似度 99%！

一图胜万言！

3. 为什么要升级到 ASP.NET Core?

将十几年依赖于 WebForms 和 .Net Framework 的项目移植到 ASP.NET Core 将是一项艰巨的任务，特别是对于企业管理系统而言，数百个页面可不是闹着玩的。

经典 WebForms 已经不再更新

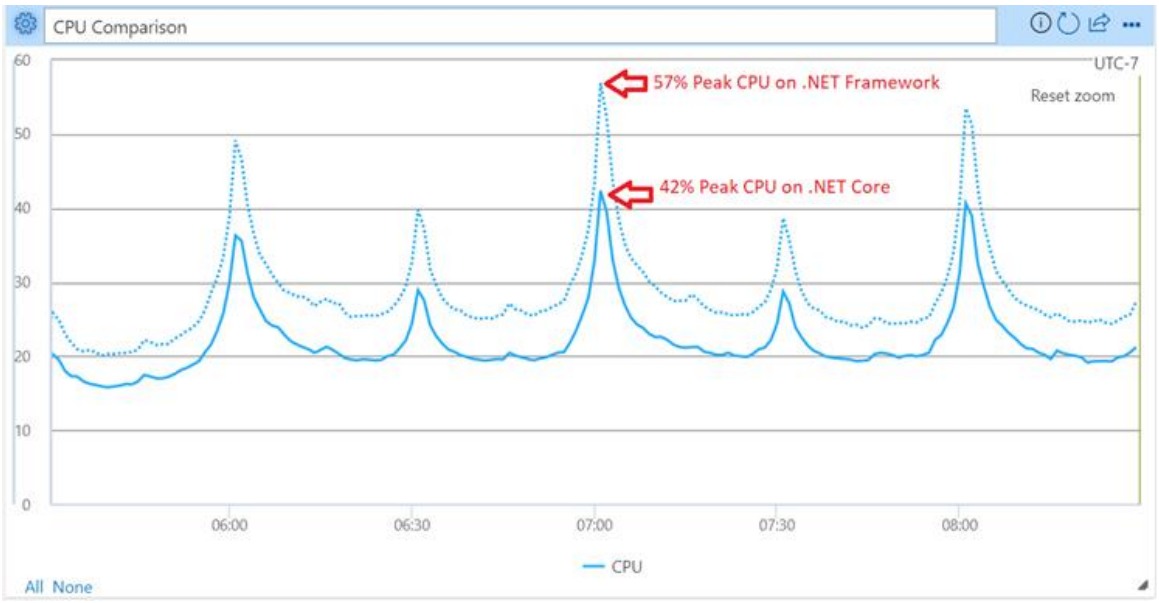
为什么要迁移到 ASP.NET Core?

虽然 ASP.NET Core 非常优秀，但最根本的问题却是 **WebForms 已经不再更新。**

随着时间的推移，WebForms 项目将面临越来越多的安全风险，因此容易受到攻击，维护成本也会越来越高，因为想找到一个熟悉过时技术的开发人员也会越来越难。及时将自己的项目升级到最新的技术是减少系统风险的不二法门。

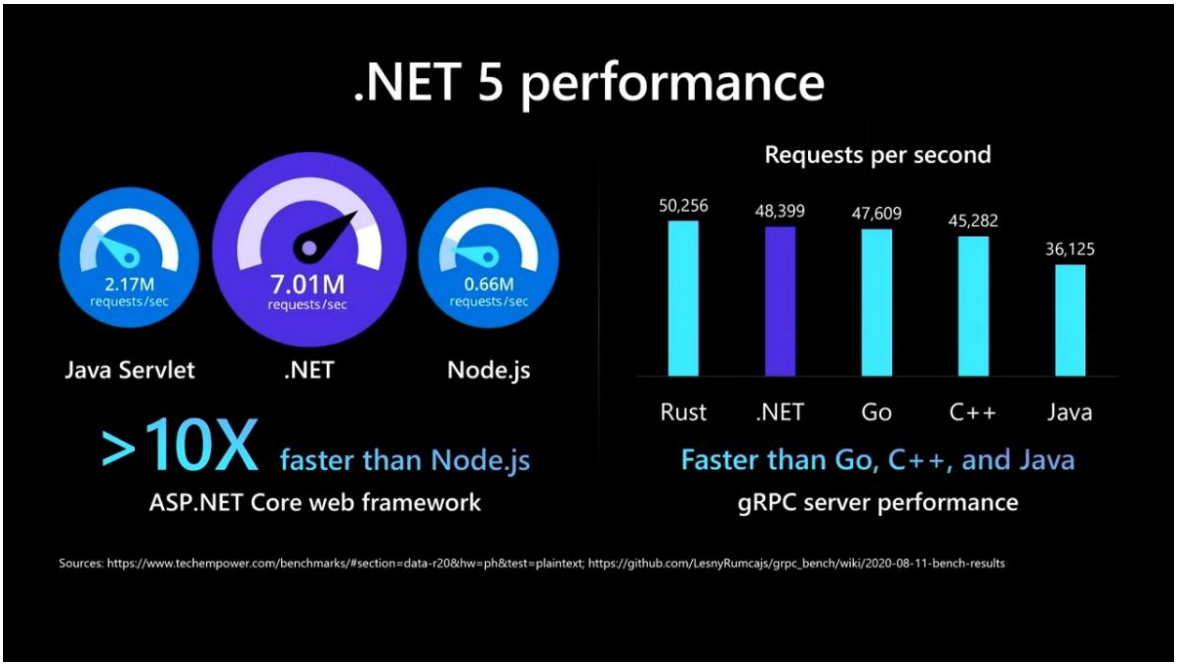
ASP.NET Core 的性能好是公认的

值得一提的是，ASP.NET Core 性能好是公认的，有报道称 Microsoft Teams 从 .NET Framework 4.6.2 迁移到 .NET Core 3.1，CPU 性能提升 25%。



1. Microsoft Teams' journey to .NET Core | .NET
2. OneService Journey to .NET 6 - .NET Blog

另有报道，ASP.NET Core 性能已经 10 倍于 Node.js，甚至比 Go, C++, Java 都要快。



How fast is ASP.NET Core?

小结

总的来说，ASP.NET Core 足够优秀来支撑这次升级：

1. ASP.NET Core 开源免费（MIT），信创产品适用。

2. ASP.NET Core 跨平台，Linux、Windows、Mac 都可以开发和运行。

3. 可以使用最新的 C# 特性，以及最新 VS 带来的效率提升。

4. 更好的性能，意味着更快的访问速度。

5. 更好的安全性。

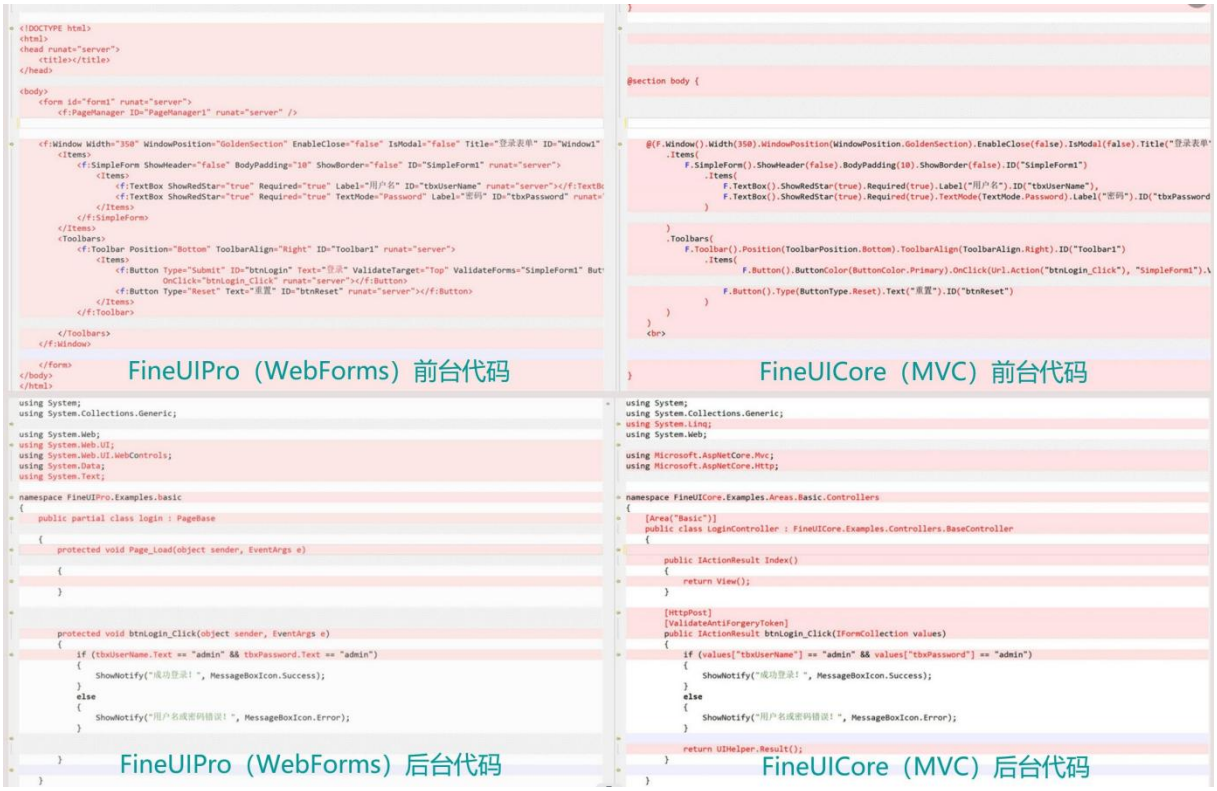
4. 简化开发工作，我们一直在努力！

为了减少大家从 WebForms 升级到最新的 ASP.NET Core 的工作量，我们一直在努力。

ASP.NET Core - MVC 开发模式

2017-12-06，我们正式发布了支持跨平台开发和部署的 FineUICore，此时只有经典的 Model-View-Controller 模式，并且前台页面是 Razor 函数的写法。如果你当时要从 FineUIPro 升级到 FineUICore，工作量还是蛮大的，来看下直观的对比。

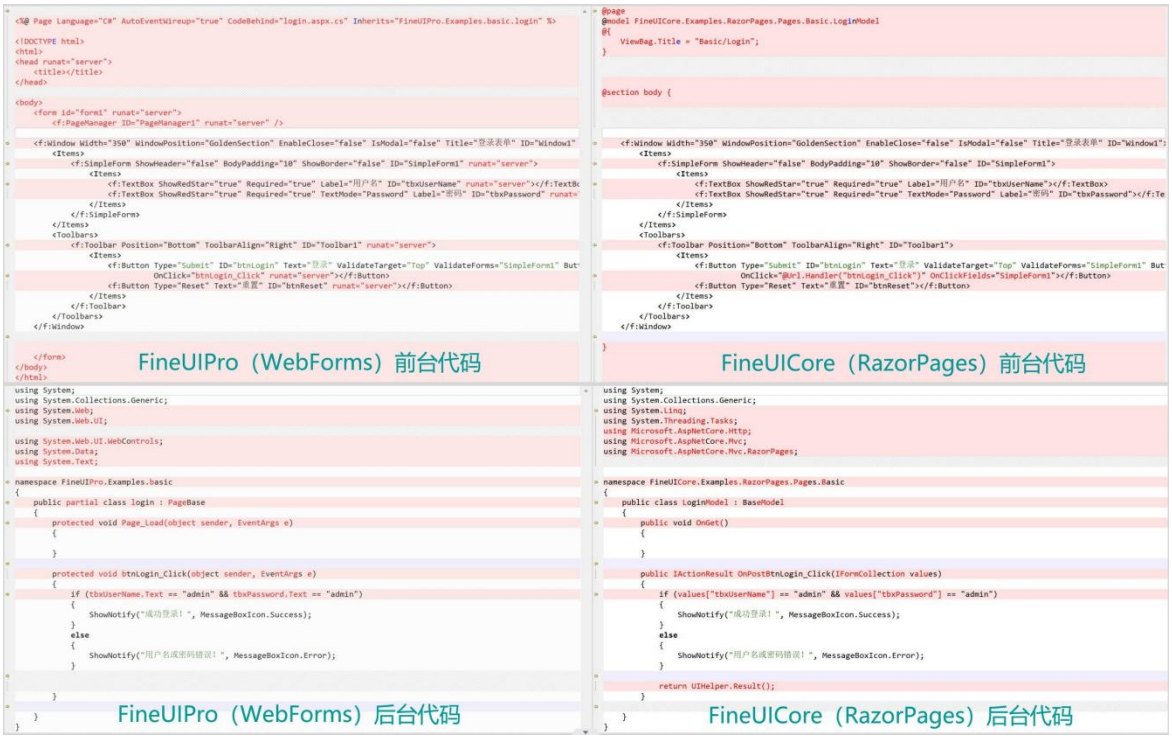
由于 ASP.NET Core Razor 视图的写法和标签的写法完全不同，所以前台代码的相似度几乎为零！仅有部分后台业务逻辑是一样的。



ASP.NET Core - RazorPages 开发模式

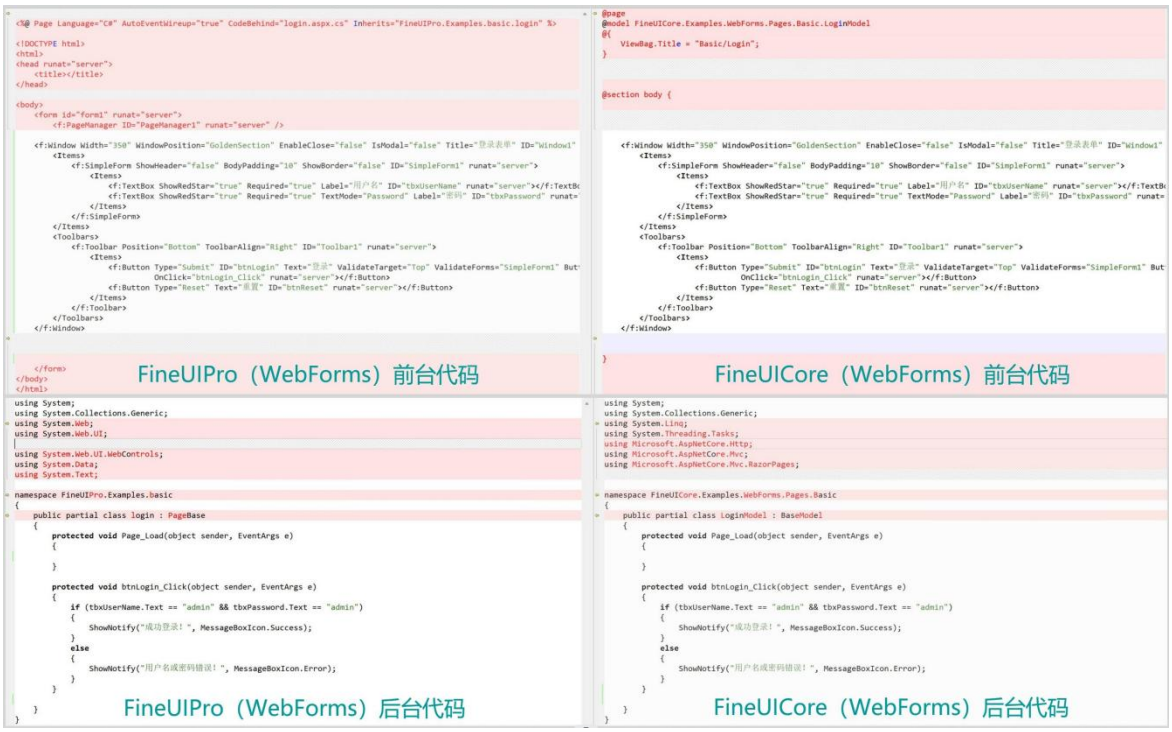
2019-06-20，我们推出了支持 Razor Pages 和 Tag Helpers 的 FineUICore，可以方便的迁移之前的 WebForms 应用，这个版本尽量保证 .cshtml 视图文件和 WebForms 的 .aspx 的一致性，可以减轻升级的工作量。

我们专门写了一篇文章详细描述升级过程，可以参考：【FineUICore】全新 ASP.NET Core，比 WebForms 还简单！ - 三生石上(FineUI 控件) - 博客园



ASP.NET Core - WebForms 开发模式

2024 年的今天，我们推出支持 WebForms 开发模式的 FineUICore，不仅可以做到前台页面的高度相似，而且后台业务代码也可以做到 99%的相似度。



小结

十几年如一日，我们初心不变，始终恪守如下三个原则，为提升大家的开发体验而不懈努力：

1. 一切为了简单。

2. 用心实现 80% 的功能。

3. 创新所以独一无二。

5. 为什么引入 WebForms 开发模式？

自从 2019 年推出支持 RazorPages 的 FineUICore 以来，我们不断收到用户反馈，吐槽 ASP.NET Core 的使用复杂，没有之前的 WebForms 好用。

我简单总结了一下，有人吐糟传递参数麻烦，还要自己写 JavaScript 代码；有人吐槽后台代码的一致；还有人搞不清楚 UIHelper 该什么时间使用，以及创建的控件和页面上的控件实例有啥关系。

初始化数据的方式不同

在 ASP.NET Core 中，我们需要在 OnGet 函数中初始化数据，然后通过 ViewData 传入视图文件：

```
public void OnGet()
{
    LoadData();
}

private void LoadData()
{
    var recordCount = DataSourceUtil.GetTotalCount();
    // 1.设置总项数（特别注意：数据库分页初始化时，一定要设置总记录数 RecordCount）
    ViewBag.Grid1RecordCount = recordCount;
    // 2.获取当前分页数据
    ViewBag.Grid1DataSource = DataSourceUtil.GetPagedDataTable(pageIndex: 0, pageSize: 5);
}
```

而在 WebForms 的 Page_Load 中，我们可以直接获取表格控件进行数据绑定：

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        BindGrid();
    }
}

private void BindGrid()
{
    // 1.设置总项数（特别注意：数据库分页一定要设置总记录数 RecordCount）
    Grid1.RecordCount = GetTotalCount();
    // 2.获取当前分页数据
    DataTable table = GetPagedDataTable(Grid1.PageIndex, Grid1.PageSize);
    // 3.绑定到 Grid
    Grid1.DataSource = table;
    Grid1.DataBind();
}
```

向后台传递数据的方式不同

在 ASP.NET Core 中，所有后台拿到的数据都需要在视图代码中通过 JavaScript 的方式获取：

```
<f:Button ID="btnSubmit" CssClass="marginr" ValidateForms="SimpleForm1" Text="登录"
    OnClick="@Url.Handler("btnSubmit_Click")"
    OnClickParameter1="@(new Parameter("userName", "F.ui.tbxUserName.getValue()"))"
    OnClickParameter2="@(new Parameter("password", "F.ui.tbxPassword.getValue()"))">
</f:Button>
```

比如这个示例向后台传递了两个参数 userName 和 password，后台通过函数参数的方式接受：

```
public IActionResult OnPostBtnSubmit_Click(string userName, string password)
{
    UIHelper.Label("labResult").Text("用户名：" + userName + " 密码：" + password);
    return UIHelper.Result();
}
```

这个示例有两个难点：

1. 代码抽象不好理解：通过 UIHelper.Label 函数拿到的控件是一个在内存中新建的实例（其目的是为了向前台输出一段改变标签控件文本的 JavaScript 脚本），和页面初始化时的那个 Label 控件没有任何关系。
2. 不小心写错参数名称的话，编译不会报错，运行时不能正确获取传入的参数值。

而在 WebForms 中，可以直接在后台获取控件的属性，无需任何特殊处理：

```
<f:Button ID="btnSubmit" CssClass="marginr" runat="server" OnClick="btnSubmit_Click"
ValidateForms="SimpleForm1" Text="登录">
</f:Button>
```

后台直接通过控件实例的属性获取，可以直接通过智能提示快速输入属性名称，而且有编译时提示：

```
protected void btnSubmit_Click(object sender, EventArgs e)
{
    labResult.Text = "用户名：" + tbxUserName.Text + " 密码：" + tbxPassword.Text;
}
```

回发时数据处理方式不同

在 ASP.NET Core 中，后台更新表格数据需要一套单独的代码（因为页面初始化时使用 ViewData 进行数据传递，所以无法和回发时的数据绑定共用一套代码）：

```
public IActionResult OnPostGrid1_PageIndexChanged(string[] Grid1_fields, int Grid1_pageIndex)
{
    var grid1 = UIHelper.Grid("Grid1");
    var recordCount = DataSourceUtil.GetTotalCount();
    // 1.设置总项数（数据库分页回发时，如果总记录数不变，可以不设置 RecordCount）
    grid1.RecordCount(recordCount);
    // 2.获取当前分页数据
    var dataSource = DataSourceUtil.GetPagedDataTable(pageIndex: Grid1_pageIndex, pageSize:
5);
    grid1.DataSource(dataSource, Grid1_fields);
    return UIHelper.Result();
}
```

而在 WebForms 中，页面回发时重新绑定表格数据和页面初始化时共用一套代码：

```
protected void Grid1_PageIndexChange(object sender, GridPageEventArgs e)
{
    BindGrid();
}
```

小结

经过前面的对比，我们能明显感觉到 WebForms 的代码更加直观，更加容易理解，并且 WebForms 的代码量更少，易于维护。

6. 全新 WebForms 开发模式（全球首创）

全球首创，实至名归

为了解决上述问题，让开发人员在享受 ASP.NET Core 免费开源跨平台速度快的优点同时，还能拥有 WebForms 比较高的开发效率，我们为 ASP.NET Core 引入了 WebForms 模式。

截止目前，能真正将 WebForms 引入 ASP.NET Core 的控件库厂商仅此一家，别无分店。我们也诚挚的邀请你来试用，相信你一定会喜欢这个全球首创的创新功能。

视图文件+页面模型文件+自动生成的设计时文件

首先从一个最简单的页面入手，我们来看下启用 WebForms 的 ASP.NET Core 到底是个什么样子?

一个简单的模拟登录页面，用户输入指定的用户名和密码之后，弹出登录成功提示框。

在 ASP.NET Core RazorPages 项目中，我们需要新建一个页面文件以及后台代码文件（或者称之为页面模型）：

Login.cshtml.designer.cs Login.cshtml.cs Login.cshtml

FineUICore.Examples.WebForms

```
@page
@model FineUICore.Examples.WebForms.Pages.Basic.LoginModel
@{
    ViewBag.Title = "Basic/Login";
}

@section body {

    <div>
        用户名：admin
        <br>
        密码：admin
        <br>
        <br>
        <br>
        注意：在任意输入框内按回车键都会触发表单的提交（相当于点击【登陆】按钮）。
    </div>

    <f:Window Width="350" WindowPosition="GoldenSection" EnableClose="false" IsModal="false" Title="登录表单" ID="Window1">
        <Items>
            <f:SimpleForm ShowHeader="false" BodyPadding="10" ShowBorder="false" ID="SimpleForm1">
                <Items>
                    <f:TextBox ShowRedStar="true" Required="true" Label="用户名" ID="tbxUserName"></f:TextBox>
                    <f:TextBox ShowRedStar="true" Required="true" TextMode="Password" Label="密码" ID="tbxPassword"></f:TextBox>
                </Items>
            </f:SimpleForm>
        </Items>
        <Toolbars>
            <f:Toolbar Position="Bottom" ToolbarAlign="Right" ID="Toolbar1">
                <Items>
                    <f:Button ButtonColor="Primary" Type="Submit" ID="btnLogin" Text="登录"
                              ValidateTarget="Top" ValidateForms="SimpleForm1" OnClick="btnLogin_Click"></f:Button>
                    <f:Button Type="Reset" Text="重置" ID="btnReset"></f:Button>
                </Items>
            </f:Toolbar>
        </Toolbars>
    </f:Window>

}
```

Login.cshtml.designer.cs Login.cshtml.cs Login.cshtml

FineUICore.Examples.WebForms FineUICore.Examples.WebForms.Pages.Basic.LoginMo

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;

namespace FineUICore.Examples.WebForms.Pages.Basic
{
    public partial class LoginModel : BaseModel
    {
        protected void Page_Load(object sender, EventArgs e)
        {

        }

        protected void btnLogin_Click(object sender, EventArgs e)
        {
            if (tbxUserName.Text == "admin" && tbxPassword.Text == "admin")
            {
                ShowNotify("成功登录！", MessageBoxIcon.Success);
            }
            else
            {
                ShowNotify("用户名或密码错误！", MessageBoxIcon.Error);
            }
        }
    }
}
```

注意，在登录按钮的点击事件中，可以直接读取输入框的 tbxUserName 的 Text 属性，这个就是

FineUICore 黑魔法，我们会将控件的一些关键属性回发到后台，并自动绑定到相应的控件实例。

而这个控件实例 (tbxUserName) 是在一个名为 Login.cshtml.designer.cs 文件中声明的，FineUICore 会在页面回发时自动初始化这个实例，并绑定关键属性值。

Login.cshtml.designer.cs | Login.cshtml.cs | Login.cshtml

FineUICore.Examples.WebForms | FineUICore.Examples.WebForms.Pages.Basic.Lc

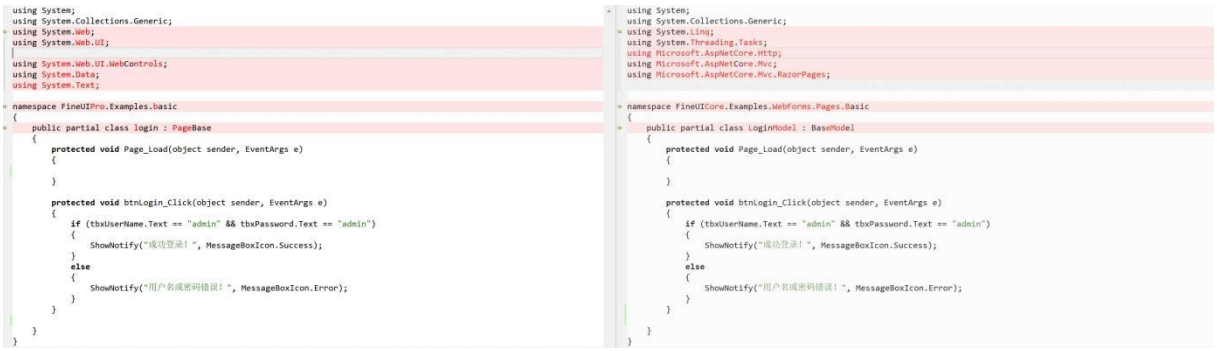
```
1   //------------------------------------------------------------------------------
2   // <auto-generated>
3   //     This code was generated by FineUICore.
4   //
5   //     Changes to this file may cause incorrect behavior and will be lost if
6   //     the code is regenerated.
7   // </auto-generated>
8   //------------------------------------------------------------------------------
9
10  namespace FineUICore.Examples.WebForms.Pages.Basic
11  {
12      public partial class LoginModel
13      {
14          protected FineUICore.Window Window1;
15          protected FineUICore.SimpleForm SimpleForm1;
16          protected FineUICore.TextBox tbxUserName;
17          protected FineUICore.TextBox tbxPassword;
18          protected FineUICore.Toolbar Toolbar1;
19          protected FineUICore.Button btnLogin;
20          protected FineUICore.Button btnReset;
21      }
22  }
23
```

注：我们会提供一个 Visual Studio 插件自动生成这个文件，无需开发人员手工编写。

小结

如果上述代码让你想起了 20 年前的 WebForms，那就对了。业务代码 99%的相似度是实打实的，这也就为经典 WebForms 的项目迁移到最新的 ASP.NET Core 奠定了扎实的基础。

让我们用工具对比下实现相同功能的经典 WebForms 和 FineUICore（开启 WebForms 模式）代码。



7. 哪些所谓的 WebForms 缺点怎么办?

WebForms 的缺点已经不复存在!

20 年前大家所诟病的 WebForms 的缺点之一（网络传输量大）已经不复存在，而 WebForms 的快速开发特性（Rapid Application Development - RAD）却越来越重要。

有报告显示，今天的主流网站的网页过于臃肿，以至于严重影响浏览性能，而能流畅玩手游《绝地求生》的入门级移动设备甚至难以正常加载。Wix 每个网页需要加载 21MB，Patreon 和 Threads 每个网页需要加载 13MB 的数据。臃肿的网页导致加载时间长达 33 秒，部分情况下甚至无法加载。基本上主流社交平台都存在臃肿的问题。而内容创建平台 Squarespace 和论坛 Discourse 的新版本通常比旧版本性能更差。

Site	Size		M3 Max		M1 Pro		M3/10		Tecno S8C		Itel P32	
	wire	raw	LCP*	CPU	LCP*	CPU	LCP*	CPU	LCP*	CPU	LCP*	CPU
danluu.com	6kB	18kB	50ms	20ms	50ms	30ms	0.2s	0.3s	0.4s	0.3s	0.5s	0.5s
HN	11kB	50kB	0.1s	30ms	0.1s	30ms	0.3s	0.3s	0.5s	0.5s	0.7s	0.6s
MyBB	0.1MB	0.3MB	0.3s	0.1s	0.3s	0.1s	0.6s	0.6s	0.8s	0.8s	2.1s	1.9s
phpBB	0.4MB	0.9MB	0.3s	0.1s	0.4s	0.1s	0.7s	1.1s	1.7s	1.5s	4.1s	3.9s
WordPress	1.4MB	1.7MB	0.2s	60ms	0.2s	80ms	0.7s	0.7s	1s	1.5s	1.2s	2.5s
WordPress (old)	0.3MB	1.0MB	80ms	70ms	90ms	90ms	0.4s	0.9s	0.7s	1.7s	1.1s	1.9s
XenForo	0.3MB	1.0MB	0.4s	0.1s	0.6s	0.2s	1.4s	1.5s	1.5s	1.8s	FAIL	FAIL
Ghost	0.7MB	2.4MB	0.1s	0.2s	0.2s	0.2s	1.1s	2.2s	1s	2.4s	1.1s	3.5s
vBulletin	1.2MB	3.4MB	0.5s	0.2s	0.6s	0.3s	1.1s	2.9s	4.4s	4.8s	13s	16s
Squarespace	1.9MB	7.1MB	0.1s	0.4s	0.2s	0.4s	0.7s	3.6s	14s	5.1s	16s	19s
Mastodon	3.8MB	5.3MB	0.2s	0.3s	0.2s	0.4s	1.8s	4.7s	2.0s	7.6s	FAIL	FAIL
Tumblr	3.5MB	7.1MB	0.7s	0.6s	1.1s	0.7s	1.0s	7.0s	14s	7.9s	8.7s	8.7s
Quora	0.6MB	4.9MB	0.7s	1.2s	0.8s	1.3s	2.6s	8.7s	FAIL	FAIL	19s	29s
Bluesky	4.8MB	10MB	1.0s	0.4s	1.0s	0.5s	5.1s	6.0s	8.1s	8.3s	FAIL	FAIL
Wix	7.0MB	21MB	2.4s	1.1s	2.5s	1.2s	18s	11s	5.6s	10s	FAIL	FAIL
Substack	1.3MB	4.3MB	0.4s	0.5s	0.4s	0.5s	1.5s	4.9s	14s	14s	FAIL	FAIL
Threads	9.3MB	13MB	1.5s	0.5s	1.6s	0.7s	5.1s	6.1s	6.4s	16s	28s	66s
Twitter	4.7MB	11MB	2.6s	0.9s	2.7s	1.1s	5.6s	6.6s	12s	19s	24s	43s
Shopify	3.0MB	5.5MB	0.4s	0.2s	0.4s	0.3s	0.7s	2.3s	10s	26s	FAIL	FAIL
Discourse	2.6MB	10MB	1.1s	0.5s	1.5s	0.6s	6.5s	5.9s	15s	26s	FAIL	FAIL
Patreon	4.0MB	13MB	0.6s	1.0s	1.2s	1.2s	1.2s	14s	1.7s	31s	9.1s	45s
Medium	1.2MB	3.3MB	1.4s	0.7s	1.4s	1s	2s	11s	2.8s	33s	3.2s	63s
Reddit	1.7MB	5.4MB	0.9s	0.7s	0.9s	0.9s	6.2s	12s	1.2s	∞	FAIL	FAIL

[How web bloat impacts users with slow devices]

WebForms 需要在客户端和服务端保持控件状态，所以在页面回发时，需要将页面上所有控件的状态信息一并回发，导致比较大的网络传输。20 年后的今天，随着 4G、5G 移动网络的普及，以及充足的宽带网络，这些流量已经变得不值一提。

WebForms 是划时代的技术，也可以看做是微软的低代码解决方案，只不过 20 年前出来太超前了，受制于网络传输带宽的限制，所以才为大家所诟病。现在回头看看，每次页面回发时多传输 10K 数据算个事吗？想想你刷一个抖音视频怎么说也要消耗 10M（10,240K）流量吧。而 WebForms 带来的开发效率提升，以及后期节约的维护成本，则是实实在在的好处，真金白金看得见摸得着。

实测 WebForms 的数据传输量

我猜测大家估计还是心有不甘，虽然多点数据传输能提高开发效率，减少我们写的代码量，提高可维护

性。但是成年人的世界既要、又要还要，能少传输点数据岂不是更妙。

带着这个疑问，我们来对比下 FineUICore（RazorPages）、FineUIPro（经典 WebForms）和 FineUICore（WebForms 开发模式）下传输的数据量，争取让大家用的心情舒畅。

示例一：表格的数据库分页与排序

	ASP.NET Core RazorPages	ASP.NET Core WebForms	ASP.NET 经典 WebForms
页面第一次加载	4.5	4.7	8.3
切换分页（上传）	0.5	1.2	3.2
切换分页（下载）	0.5	0.5	2.8

示例二：省市县联动

	ASP.NET Core RazorPages	ASP.NET Core WebForms	ASP.NET 经典 WebForms
页面第一次加载	4.3	4.4	5.8
选择安徽省（上传）	0.4	0.6	2.3
选择安徽省（下载）	0.8	0.8	0.9
选择合肥市（上传）	0.4	0.6	3.2
选择合肥市（下载）	0.3	0.3	0.5

示例三：树控件延迟加载

	ASP.NET Core RazorPages	ASP.NET Core WebForms	ASP.NET 经典 WebForms
页面第一次加载	3.5	3.5	3.8
点击驻马店市（上传）	0.2	0.9	1.0
点击驻马店市（下载）	0.2	0.2	0.6

注：上述表格中数字表示网络数据传输量，单位 KB。

经过上述三个页面对比，我们可以看出，经典 WebForms 不管是页面第一次加载，还是回发时上传和

下载的数据量都是最大的。

小结

1. 相比经典 WebForms，不管是页面第一次加载，还是回发时的数据传输量，ASP.NET Core（WebForms 开发模式）都是碾压级的，综合数据下载量比经典 WebForms 减少 50% 左右。

2. 与数据传输量最少的 ASP.NET Core RazorPages 相比，启用 WebForms 时，只有在页面回发时上传数据量有所增加，而页面第一次加载和回发时的下载数据量两者保持一致。

3. 不管哪种技术，上述三个示例的数据传输都是 10KB 之内，相比现在动辄 10MB（大了 1000 倍!）的数据传输，你觉得 WebForms 数据传输量大的缺点还存在吗?

8. 如何开启 WebForms 开发模式?

首先确保你使用的是 ASP.NET Core RazorPages 开发模式，只需要如下两个步骤即可在 FineUICore 项目中轻松开启 WebForms 模式。

第一步：修改 appsettings.json 配置文件

```
{
  "FineUI": {
    "EnableWebForms": true,
    "DebugMode": true,
    "Theme": "Pure_Black",
    "EnableAnimation": true,
    "MobileAdaption": true
  }
}
```

第二步：修改 Startup.cs 启动文件

在 ConfigureServices 函数中，增加 WebForms 过滤器，如下所示。

```
// FineUI 服务
services.AddFineUI(Configuration);

services.AddRazorPages().AddMvcOptions(options =>
{
    // 自定义 JSON 模型绑定（添加到最开始的位置）
    options.ModelBinderProviders.Insert(0, new FineUICore.JsonModelBinderProvider());

    // 自定义 WebForms 过滤器（仅在启用 EnableWebForms 时有效）
    options.Filters.Insert(0, new FineUICore.WebFormsFilter());

}).AddNewtonsoftJson().AddRazorRuntimeCompilation();
```

搞定!

小结

深度集成到 FineUICore 中,仅仅通过一个参数来控制是否开启 WebForms,可以对比学习 RazorPages 和 WebForms，降低了学习成本，同时也让之前购买 FineUICore 企业版的客户享受到 WebForms 带来的便利。

9. Page_Load 事件的回归

在经典 WebForms 页面中，Page_Load 事件非常重要，也是大家耳熟能详的，甚至在 20 年前 ASP.NET 1.0 发布的时候，我们就是这么写代码的。

Page_Load 事件往往伴随着对 IsPostBack 属性的判断，因为 Page_Load 事件不管是页面第一次加载，还是页面回发都会执行。因此对于哪些只需要在页面第一次加载的代码，就需要放到 !IsPostBack 的逻辑判断中。

RazorPages 中的复选框列表的初始化

示例：https://pages.fineui.com/#/Form/CheckBoxList

在 ASP.NET Core RazorPages 开发模式下，我们需要在 OnGet 中初始化数据，由于此时页面视图尚未初始化，因此我们无法知道页面视图上的任何定义。

```
public void OnGet()
{
    LoadData();
}

private void LoadData()
{
    List<TestClass> myList = new List<TestClass>();
    myList.Add(new TestClass("1", "数据绑定值 1"));
    myList.Add(new TestClass("2", "数据绑定值 2"));
    myList.Add(new TestClass("3", "数据绑定值 3"));
    myList.Add(new TestClass("4", "数据绑定值 4"));

    ViewBag.CheckBoxList2DataSource = myList;
    ViewBag.CheckBoxList2SelectedValueArray = new string[] { "1", "3" };
}
```

将准备好的数据保存在 ViewData（自定义的 ViewBag）中，然后传入视图文件，并在页面视图标签中使用这些数据。

```
<f:CheckBoxList ID="CheckBoxList2" Label="列表二（一列）" ColumnNumber="1"
                DataTextField="Name" DataValueField="Id"
                DataSource="@ViewBag.CheckBoxList2DataSource"
                SelectedValueArray="@ViewBag.CheckBoxList2SelectedValueArray">
</f:CheckBoxList>
```

WebForms 复选框列表的初始化

示例：https://forms.fineui.com/#/Form/CheckBoxList

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        LoadData();
    }
}

private void LoadData()
{
    List<TestClass> myList = new List<TestClass>();
    myList.Add(new TestClass("1", "数据绑定值 1"));
    myList.Add(new TestClass("2", "数据绑定值 2"));
    myList.Add(new TestClass("3", "数据绑定值 3"));
    myList.Add(new TestClass("4", "数据绑定值 4"));

    CheckBoxList2.DataSource = myList;
    CheckBoxList2.DataBind();

    CheckBoxList2.SelectedValueArray = new string[] { "1", "3" };
}
```

其中，IsPostBack 属性定义在页面模型基类 BaseModel.cs 中：

```
public bool IsPostBack
{
    get
    {
        return FineUICore.PageContext.IsFineUIAjaxPostBack();
    }
}
```

注意：在Page_Load事件中，页面视图已经初始化完毕，因此我们可以直接调用页面视图上的控件实例，比如这里的CheckBoxList2，对应于页面上的CheckBoxList标签定义。

```
<f:CheckBoxList ID="CheckBoxList2" Label="列表二（一列）" ColumnNumber="1"
                DataTextField="Name" DataValueField="Id">
</f:CheckBoxList>
```

小结

从上面示例中可以看出，WebForms 模式下的页面初始化更加直观，等视图文件初始化完毕后，直接获取控件实例，并设置控件属性。反过来看 RazorPages 的实现就有点繁琐了，必须通过 ViewData 进行中转，先赋值，再使用，在页面模型 OnGet 函数中无法获取视图中定义的变量。

10. 页面回发事件（PostBack）

简化页面回发事件的函数名

首先看下 RazorPages 中的按钮点击事件，：

```
<f:Button ID="btnChangeEnable" Text="启用后面的按钮"
            OnClick="@Url.Handler("btnChangeEnable_Click")" />
<f:Button ID="btnEnable" Text="禁用的按钮" OnClick="@Url.Handler("btnEnable_Click")"
            Enabled="false" />
```

对应的后台事件处理器：

```
public IActionResult OnPostBtnChangeEnable_Click()
{
    var btnEnable = UIHelper.Button("btnEnable");
    btnEnable.Enabled(true);
    btnEnable.Text("本按钮已经启用（点击弹出对话框）");
    return UIHelper.Result();
}
```

在视图文件中，定义了按钮的点击事件名为 btnChangeEnable_Click，而后台对应的事件处理器名称为 OnPostBtnChangeEnable_Click。由于前后台事件名称的不一致，导致很多开发人员将后台事件名称误写为 OnPostbtnChangeEnable_Click，导致无法进入事件处理函数。

而 WebForms 开发模式下，再看下相同的示例：

```
<f:Button ID="btnChangeEnable" Text="启用后面的按钮"
    OnClick="btnChangeEnable_Click" />
<f:Button ID="btnEnable" Text="禁用的按钮" OnClick="btnEnable_Click"
    Enabled="false" />
```

对应的后台处理函数名称和前台的定义一模一样：

```
protected void btnChangeEnable_Click(object sender, EventArgs e)
{
    btnEnable.Enabled = true;
    btnEnable.Text = "本按钮已经启用（点击弹出对话框）";
}
```

除了事件名称保持前后台一致，代码逻辑中已经完全移除 UIHelper 的调用，我们可以直接调用控件实例，修改实例属性（并非所有属性都可以在页面回发中改变，我们将这些能够在回发中改变的属性为 AJAX 属性，这个概念和经典 FineUIPro 保持一致）。

页面回发事件的参数-JavaScript 代码

在 RazorPages 中，所有后台需要的参数都需要自行传入，而获取这些参数的代码无疑是 JavaScript，

因此就需要在视图文件中混杂 JavaScript 代码。这样无形中就对开发人员提出了更高的要求，不仅要求开发人员熟悉服务器端控件的属性和方法，而且需要熟悉对应的 JavaScript 控件的方法。让代码复杂度增加，提高维护成本。

来看一个简单的示例，后台修改复选框的选中状态，在 RazorPages 中，我们需要在视图文件中自行获取复选框的当前状态，如下所示：

```
<f:CheckBox ID="CheckBox1" ShowLabel="false" Text="复选框" Checked="true">
</f:CheckBox>
<f:Button ID="btnSelectCheckBox" Text="选择/反选复选框"
    OnClick="@Url.Handler("btnSelectCheckBox_Click")"
    OnClickParameter1="@(new Parameter("isChecked",
"F.ui.CheckBox1.isChecked()"))"></f:Button>
```

注意，上述代码 F.ui.CheckBox1.isChecked() 就是一段 JavaScript 代码，用于获取复选框 CheckBox1 的当前选中状态。

对应的后台代码：

```
public IActionResult OnPostBtnSelectCheckBox_Click(bool isChecked)
{
    UIHelper.CheckBox("CheckBox1").Checked(!isChecked);
    return UIHelper.Result();
}
```

实现相同的功能，在 WebForms 开发模式下，视图代码就非常简单了：

```
<f:CheckBox ID="CheckBox1" ShowLabel="false" Text="复选框" Checked="true">
</f:CheckBox>
<f:Button ID="btnSelectCheckBox" CssClass="marginr" Text="选择/反选复选框"
    OnClick="btnSelectCheckBox_Click"></f:Button>
```

干净利落，绝不拖泥带水。后台代码也简化了很多：

```
protected void btnSelectCheckBox_Click(object sender, EventArgs e)
{
    CheckBox1.Checked = !CheckBox1.Checked;
}
```

页面回发事件的参数-OnClickFields 属性

在 RazorPages 中，如果想在后台处理器中获取某个表单控件的状态，可以通过 OnClickFields 属性传入（不同事件属性名不同，比如表格分页对应的事件参数属性名为 OnPageIndexChangedFields）。

示例：https://pages.fineui.com/#/DropDownList/DropDownList

```
<f:DropDownList ID="DropDownList1">
</f:DropDownList>
<f:Button ID="btnGetSelection" Text="获取此下拉列表的选中项"
    OnClick="@Url.Handler("btnGetSelection_Click")"
    OnClickFields="DropDownList1"></f:Button>
```

后台代码：

```
public IActionResult OnPostBtnGetSelection_Click(string DropDownList1, string
DropDownList1_text)
{
    if (!String.IsNullOrEmpty(DropDownList1))
    {
        UIHelper.Label("labResult").Text(String.Format("选中项：{0}（值：{1}）",
DropDownList1_text, DropDownList1));
    }
    else
    {
        UIHelper.Label("labResult").Text("无选中项");
    }

    return UIHelper.Result();
}
```

逻辑虽然很简单，但是开发人员有一个认知负担，不仅要通过学习和命名约定知道 DropDownList1 名称表示下拉列表的选中值，而且要知道 DropDownList1_text 表示下拉列表选中的文本，确保

DropDownList1_text 大小写完全正确才能接收到前台传入的值！

完成相同功能的 WebForms 代码，不仅简洁，而且可以充分利用 IDE 的智能提示，不需要手工输入参数名称。而且编译时也会杜绝大小写错误这样的问题。

示例：https://forms.fineui.com/#/DropDownList/DropDownList

```
<f:DropDownList ID="DropDownList1">
</f:DropDownList>
<f:Button ID="btnGetSelection" Text="获取此下拉列表的选中项"
    OnClick="btnGetSelection_Click"></f:Button>
```

后台代码：

```
protected void btnGetSelection_Click(object sender, EventArgs e)
{
    if (!String.IsNullOrEmpty(DropDownList1.Text))
    {
        labResult.Text = String.Format("选中项：{0}（值：{1}）", DropDownList1.Text,
DropDownList1.SelectedValue);
    }
    else
    {
        labResult.Text = "无选中项";
    }
}
```

小结

下面简单总结一下 WebForms 模式下回发事件和 RazorPages 中的不同之处：

1. 视图代码中无需将事件名称置于 Url.Handler()函数中。
2. 视图代码中无需编写 JavaScript 代码来获取控件状态。
3. 视图中也无需设置 OnClickFields 来向后台传递控件状态。
4. 后台事件名称和前台视图定义的事件名称完全一致。
5. 事件处理函数的返回值是 void，因此无需返回 UIHelper.Result()。

6. 事件处理函数参数和经典的 WebForms 保持一致，第一个参数是触发事件的控件实例，第二个是事件参数（比如表格分页的事件参数类型为 GridPageEventArgs）。
7. 事件处理函数中完全移除对 UIHelper 的依赖（之前需要重建控件实例，比如 UIHelper.Button("btnEnable")）。

如果你是从经典的 ASP.NET WebForms 直接学习的 FineUICore（WebForms 开发模式），忘记上面所有的不同，你只需要记着一点：FineUICore（WebForms 模式）的事件处理和经典 WebForms 的事件处理**一模一样**!

11. 共享同一套代码（第一次加载和回发请求）

为什么 RazorPages 中无法共享代码?

之前我已经提到，在 RazorPages 中无法在页面第一次请求和页面回发事件之间共享代码。原因很简单，OnGet 时页面视图尚未执行，我们只能从数据库获取数据并填充 ViewData 对象，而在页面回发事件中 ViewData 不可用，我们必须直接操作页面控件。这种割裂和不一致性带来了很多麻烦，有时我们不得不写两套代码，来实现相同的功能。

示例：https://pages.fineui.com/#/GridPaging/Database

以数据库分页为例，首先在 OnGet 中获取数据并保存到 ViewBag 中：

```
public void OnGet()
{
    LoadData();
}

private void LoadData()
{
    var recordCount = DataSourceUtil.GetTotalCount();

    // 1.设置总项数（特别注意：数据库分页初始化时，一定要设置总记录数 RecordCount）
    ViewBag.Grid1RecordCount = recordCount;

    // 2.获取当前分页数据
    ViewBag.Grid1DataSource = DataSourceUtil.GetPagedDataTable(pageIndex: 0, pageSize: 5);
}
```

然后在视图文件中使用这些数据：

```
<f:Grid ID="Grid1" ShowBorder="true" ShowHeader="true" Title="表格"
    DataIDField="Id" DataTextField="Name" EnableCheckBoxSelect="true"
    AllowPaging="true" PageSize="5" IsDatabasePaging="true"
    RecordCount="@ViewBag.Grid1RecordCount"
    DataSource="@ViewBag.Grid1DataSource"
    OnPageIndexChanged="@Url.Handler("Grid1_PageIndexChanged")"
    OnPageIndexChangedFields="Grid1">
    <Columns>
        ...
    </Columns>
</f:Grid>
```

再看下分页事件，我们使用 OnPageIndexChangedFields 属性来执行分页回发时需要向后台传递 Grid1 控件的状态。后台处理器使用 Grid1_fields 和 Grid1_pageIndex 来接收表格 Grid1 回发的状态。正如上一节我们说的那样，开发者这里会有比较强的心理负担，不仅要保证参数名称一个字符串都不要错，而且还要知道参数类型，否则无法接受前台传入的数据。

```
public IActionResult OnPostGrid1_PageIndexChanged(string[] Grid1_fields, int Grid1_pageIndex)
{
    var grid1 = UIHelper.Grid("Grid1");

    var recordCount = DataSourceUtil.GetTotalCount();

    // 1.设置总项数（数据库分页回发时，如果总记录数不变，可以不设置 RecordCount）
    grid1.RecordCount(recordCount);

    // 2.获取当前分页数据
    var dataSource = DataSourceUtil.GetPagedDataTable(pageIndex: Grid1_pageIndex, pageSize:
5);
    grid1.DataSource(dataSource, Grid1_fields);

    return UIHelper.Result();
}
```

仔细观察这段，你会发现对 GetTotalCount 和 GetPagedDataTable 的调用在 OnGet 中已经执行了一次，这里还要再重写一遍，也是无奈之举！在 OnGet 中，我们将相关数据赋值给 ViewBag，在 OnPost 中，我们将这些数据通过函数调用的形式最终输出到前台。

对于一个完美主义者，一个对代码精益求精的开发人员来说，这种在一个页面重复代码的行为是不能容忍的，而又是无可奈何的。

恼人的硬编码数字 5

细心的网友可能还会发现另一个瑕疵，那就是 pageSize 的硬编码和前台后的多处定义，简单来说，这个页面在三个地方用到了数字 5。

第一次是获取数据的时候：

```
ViewBag.Grid1DataSource = DataSourceUtil.GetPagedDataTable(pageIndex: 0, pageSize: 5);
```

第二次是视图中赋值表格的 PageSize 属性时：

```
<f:Grid …
    AllowPaging="true" PageSize="5" IsDatabasePaging="true"
```

第三次是分页回发，更新数据时：

```
var dataSource = DataSourceUtil.GetPagedDataTable(pageIndex: Grid1_pageIndex, pageSize: 5);
```

有没有解决办法，当然是有的！

我们只需要在页面模式文件中定义一个变量即可，然后把这个数字使用 ViewBag 的方式传入视图，并在三个地方使用这个变量。

```
private int pageSize = 5;
```

```
ViewBag.Grid1PageSize = pageSize;
```

```
<f:Grid …
    AllowPaging="true" PageSize="@ViewBag.Grid1PageSize" IsDatabasePaging="true"
```

功能是实现了，怎么总有高射炮打蚊子-大材小用的意思，又有点脱裤子放屁-多此一举的感觉？！

如果你也这么觉得，那说明咱们是心有灵犀，英雄所见略同，等会你就能看到 WebForms 开发模式下 FineUICore 的优雅处理方式。

WebForms 优雅的共享一套表格分页代码

实现上述相同的功能，WebForms 开发模型提供的解决方案不仅直观，简洁，代码量小，有智能提示和编译时报错，而且相当的优雅，读起来就让人心情舒畅。

下面我们就来一气哈成，从视图定义，到页面第一次加载，最后是分页回发，来看看 WebForms 的魔

力。

示例：https://forms.fineui.com/#/GridPaging/Database

```
<f:Grid ID="Grid1" ShowBorder="true" ShowHeader="true" Title="表格"
    DataIDField="Id" DataTextField="Name" EnableCheckBoxSelect="true"
    AllowPaging="true" PageSize="5" IsDatabasePaging="true"
    OnPageIndexChanged="Grid1_PageIndexChanged">
    <Columns>
        ...
    </Columns>
</f:Grid>
```

干净，纯粹，没有一堆碍眼的 ViewBag，也无需考虑回发参数。

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        LoadData();
    }
}

private void LoadData()
{
    var recordCount = DataSourceUtil.GetTotalCount();

    // 1.设置总项数（特别注意：数据库分页初始化时，一定要设置总记录数 RecordCount）
    Grid1.RecordCount = recordCount;

    // 2.获取当前分页数据
    Grid1.DataSource = DataSourceUtil.GetPagedDataTable(pageIndex: Grid1.PageIndex,
pageSize: Grid1.PageSize);
    Grid1.DataBind();
}
```

直观，易懂，直接控件属性赋值，没有中间商赚差价。

```
protected void Grid1_PageIndexChanged(object sender, GridPageEventArgs e)
{
    LoadData();
}
```

 ，这也太简单了，简单的不像话！

之前那个恼人的硬编码数字 5 已经消失不见了，我们只需要在页面标签中定义一次就行了。在 Page_Load 事件和页面回发事件中，都可以方便的使用 Grid1.PageSize 来获取这个值，是不是很简单！

WebForms 表格的分页和排序同样简洁和优雅

当表格的数据库分页和排序同时存在，我们对比看下 RazorPages 和 WebForms 的异同，大家感受一下哪种解决方案更简洁，用背景色标识不同的方案。

示例一：https://pages.fineui.com/#/Grid/SortingPagingDatabase

示例二：https://forms.fineui.com/#/Grid/SortingPagingDatabase

视图标签定义：

```
<f:Grid ID="Grid1" ShowBorder="true" ShowHeader="true" Title="表格"
EnableCheckBoxSelect="true" DataIDField="Id" DataTextField="Name"
        AllowPaging="true" PageSize="5" IsDatabasePaging="true"
         OnPageIndexChanged="@Url.Handler("Grid1_PageIndexChanged")"
         OnPageIndexChangedFields="Grid1"
        AllowSorting="true"
         SortField="@ViewBag.Grid1SortField"
         SortDirection="@ViewBag.Grid1SortDirection"
         OnSort="@Url.Handler("Grid1_Sort")"
         OnSortFields="Grid1"
        DataSource="@ViewBag.Grid1DataSource"
         RecordCount="@ViewBag.Grid1RecordCount">
    <Columns>
        ...
    </Columns>
</f:Grid>
```

```
<f:Grid ID="Grid1" ShowBorder="true" ShowHeader="true" Title="表格"
EnableCheckBoxSelect="true" DataIDField="Id" DataTextField="Name"
            AllowPaging="true" PageSize="5" IsDatabasePaging="true"
             OnPageIndexChanged="Grid1_PageIndexChanged"
            AllowSorting="true" SortField="Gender" SortDirection="ASC"
             OnSort="Grid1_Sort">
    <Columns>
        ...
    </Columns>
</f:Grid>
```

页面初始化：

```
public void OnGet()
{
    LoadData();
}

private void LoadData()
{
    string sortField = "Gender";
    string sortDirection = "ASC";

    var recordCount = DataSourceUtil.GetTotalCount();

    // 1.设置总项数（特别注意：数据库分页初始化时，一定要设置总记录数 RecordCount）
    ViewBag.Grid1RecordCount = recordCount;

    // 2.获取当前分页数据
    ViewBag.Grid1DataSource = DataSourceUtil.GetPagedDataTable(pageIndex: 0,
        pageSize: 5,
        sortField: sortField,
        sortDirection: sortDirection);

    ViewBag.Grid1SortField = sortField;
    ViewBag.Grid1SortDirection = sortDirection;
}
```

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        LoadData();
    }
}

private void LoadData()
{
    var recordCount = DataSourceUtil.GetTotalCount();

    // 1.设置总项数（特别注意：数据库分页初始化时，一定要设置总记录数 RecordCount）
    Grid1.RecordCount = recordCount;

    // 2.获取当前分页数据
    Grid1.DataSource = DataSourceUtil.GetPagedDataTable(pageIndex: Grid1.PageIndex,
        pageSize: Grid1.PageSize,
        sortField: Grid1.SortField,
        sortDirection: Grid1.SortDirection);
    Grid1.DataBind();
}
```

分页和排序事件：

```
public void Grid1PageIndexChangedOrSort(string[] Grid1_fields, int Grid1_pageIndex, string
Grid1_sortField, string Grid1_sortDirection)
{
    var grid1 = UIHelper.Grid("Grid1");

    var recordCount = DataSourceUtil.GetTotalCount();

    // 1.设置总项数（数据库分页回发时，如果总记录数不变，可以不设置 RecordCount）
    grid1.RecordCount(recordCount);

    // 2.获取当前分页数据
    var dataSource = DataSourceUtil.GetPagedDataTable(pageIndex: Grid1_pageIndex, pageSize: 5,
sortField: Grid1_sortField, sortDirection: Grid1_sortDirection);
    grid1.DataSource(dataSource, Grid1_fields);
}

public IActionResult OnPostGrid1_PageIndexChanged(string[] Grid1_fields, int Grid1_pageIndex,
string Grid1_sortField, string Grid1_sortDirection)
{
    Grid1PageIndexChangedOrSort(Grid1_fields, Grid1_pageIndex, Grid1_sortField,
Grid1_sortDirection);

    return UIHelper.Result();
}

public IActionResult OnPostGrid1_Sort(string[] Grid1_fields, int Grid1_pageIndex, string
Grid1_sortField, string Grid1_sortDirection)
{
    Grid1PageIndexChangedOrSort(Grid1_fields, Grid1_pageIndex, Grid1_sortField,
Grid1_sortDirection);

    return UIHelper.Result();
}
```

```
protected void Grid1_Sort(object sender, EventArgs e)
{
    LoadData();
}

protected void Grid1_PageIndexChanged(object sender, EventArgs e)
{
    LoadData();
}
```

如果说 WebForms 简洁的页面标签和直观的页面初始化代码还不足以动摇你的心智，那最后分页和排序事件代码，一行 LoadData()代码搞定，和 RazorPages 洋洋洒洒 13 行代码，算不算是降维打击。

WebForms 切换表格数据源

切换表格数据源，通过点击重新绑定表格按钮，在有数据和无数据之间切换。下面简单对比下 RazorPages 和 WebForms 下后台代码的实现。

示例一：https://pages.fineui.com/#/GridOther/EmptyText

```
public void OnGet()
{
    ViewBag.Grid1DataSource = DataSourceUtil.GetDataTable();
    ViewBag.Grid1DataSourceKey = "source1";
}
public IActionResult OnPostButton1_Click(string[] fields, string source)
{
    var grid1 = UIHelper.Grid("Grid1");
    if (String.IsNullOrEmpty(source) || source == "source2")
    {
        grid1.DataSource(DataSourceUtil.GetDataTable(), fields);
        grid1.Attribute("data-source-key", "source1");
    }
    else
    {
        grid1.DataSource(null);
        grid1.Attribute("data-source-key", "source2");
    }
    return UIHelper.Result();
}
```

示例二：https://forms.fineui.com/#/GridOther/EmptyText

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        LoadData();
    }
}
private void LoadData()
{
    var sourceKey = Grid1.Attributes.Value<string>("data-source-key");
    if (String.IsNullOrEmpty(sourceKey) || sourceKey == "source2")
    {
        Grid1.DataSource = DataSourceUtil.GetDataTable();
        Grid1.DataBind();
        sourceKey = "source1";
    }
    else
    {
        Grid1.DataSource = null;
        Grid1.DataBind();
        sourceKey = "source2";
    }
    Grid1.Attributes["data-source-key"] = sourceKey;
}

protected void Button1_Click(object sender, EventArgs e)
{
    LoadData();
}
```

WebForms 的后台代码，页面初始化和按钮点击事件共用 LoadData 函数，更加直观易懂。

小结

FineUICore 的 WebForms 开发模式下，Page_Load 事件在视图文件初始化完成之后调用，因此我们可以抛弃传统的 ViewData 传值方式，而是直接使用视图中定义的控件实例，可以读取控件属性（比如 Grid1.PageSize），也可以对控件属性赋值（Grid1.DataSource=table1;）。

我们也对页面回发的代码进行了改造，之前使用 UIHelper 创建一个虚拟的控件，并调用控件方法来修改控件属性（比如 UIHelper.Button("Button1").Text("按钮一")），在 WebForms 模式下，我们会重建页面视图中定义的控件（恢复其状态和关键属性），并使用赋值操作来修改控件属性（比如 Button1.Text="按钮一";）。

综上所述，页面第一次加载和页面回发时，我们都可以获取视图中定义的控件实例，也都可以通过赋值操作修改控件属性，自然而然可以共用一套代码了。这样写出来的代码不仅代码量少，而且直观、易懂，降低维护成本，何乐而不为。

12. __doPostBack 函数的回归（自定义回发）

说起__doPostBack 函数你一定不会陌生，任意打开一个经典的 WebForms 页面源代码，都能看到她的身影。

```
<body>
    <form method="post" action="./" id="form1">
<div class="aspNetHidden">
<input type="hidden" name="__EVENTTARGET" id="__EVENTTARGET" value="" />
<input type="hidden" name="__EVENTARGUMENT" id="__EVENTARGUMENT" value="" />
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE" value="/wEPDwUKLTMw
</div>

<script type="text/javascript">
//<![CDATA[
var theForm = document.forms['form1'];
if (!theForm) {
    theForm = document.form1;
}
function __doPostBack(eventTarget, eventArgument) {
    if (!theForm.onsubmit || (theForm.onsubmit() != false)) {
        theForm.__EVENTTARGET.value = eventTarget;
        theForm.__EVENTARGUMENT.value = eventArgument;
        theForm.submit();
    }
}
//]]>
</script>
```

_doPostBack 的调用与捕获

FineUICore 的 WebForms 开发模式，我们也引入了这个函数，用于自定义页面回发，两个参数的含义和传统的 WebForms 之前的一模一样：

1. eventTarget: 触发本次回发事件的控件实例（一般留空）。
2. eventArgument：回发事件参数（需要重点关注）。

由于页面的 Page_Load 事件在页面回发时也会调用，所以我们只需要在这里捕获自定义回发事件即可。

下面通过一个示例（复选框列表（最多选中一项））来看下__doPostBack 的函数调用与后台捕获。

示例：https://forms.fineui.com/#/Form/CheckBoxListRadio

```
<f:CheckBoxList ID="CheckBoxList1" ColumnNumber="2" Label="列表一">
    <Items>
        ...
    </Items>
    <Listeners>
        <f:Listener Event="change" Handler="onCheckBoxListChange" />
    </Listeners>
</f:CheckBoxList>
```

上述代码定义了一个复选框列表控件，以及客户端事件 change 的处理函数 onCheckBoxListChange：

```
<script>
// 同时只能选中一项
function onCheckBoxListChange(event, checkbox, isChecked) {
    var me = this;

    // 当前操作是：选中
    if (isChecked) {
        // 仅选中这一项
        me.setValue(checkbox.getInputValue());
    }

    // 触发后台事件
    __doPostBack('', 'CheckBoxList1_Change');
}
</script>
```

上述代码修改了复选框列表的默认行为，当点击某一个复选框使其选中时，会清空其他所有选中项，并将复选框列表的选中项设为当前选中状态的复选框（setValue(checkbox.getInputValue())）。

然后进行自定义回发，在处理后台逻辑：

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
    }
    else
    {
        string arg = GetRequestEventArgument(); // 此函数所在文件：PageBase.cs
        if (arg == "CheckBoxList1_Change")
        {
            if (CheckBoxList1.SelectedValueArray.Length > 0)
            {
                ShowNotify(String.Format("列表一的选中项：{0}", String.Join(", ",
CheckBoxList1.SelectedValueArray)));
            }
            else
            {
                ShowNotify("列表一没有选中项！");
            }
        }
    }
}
```

看似有点复杂，其实逻辑非常直观易懂，我们从上到下理一理其中的三个 if-else 判断：

- 如果是页面第一次加载：什么都不做（本页面无需初始化代码）
- 如果是页面回发，则获取回发参数（GetRequestEventArgument()）
  - 如果参数是 CheckBoxList1_Change（对应于视图中的__doPostBack 调用）
    - 如果复选框列表存在选中值：则弹出提示信息（列表一的选中项：value5）
    - 如果复选框列表没有选中值：则弹出提示信息（列表一没有选中项!）

其中 GetRequestEventArgument()是在父类中定义的公共函数，不仅可以获取单个回发参数，还可以切分以$分割的多个参数，如下所示：

```
public string GetRequestEventArgument()
{
    return Request.Form["__EVENTARGUMENT"];
}

public string[] GetRequestEventArguments()
{
    var arg = GetRequestEventArgument();
    return arg.Split("$");
}
```

自定义回发参数

简单的自定义回发参数，可以直接使用$分割的字符串，这个约定也是从经典 WebForms 借鉴而来的。

下面以自定义树节点的点击事件来演示其用法。

示例：https://forms.fineui.com/#/Tree/TreeNodeClick

```
<f:Tree ID="Tree1" IsFluid="true" ShowHeader="true" Title="树控件">
    <Nodes>
        <f:TreeNode Text="中国" Expanded="true" NodeID="china">
            ...
        </f:TreeNode>
    </Nodes>
    <Listeners>
        <f:Listener Event="nodeclick" Handler="onTree1NodeClick"></f:Listener>
    </Listeners>
</f:Tree>
```

页面视图中定义了一个树控件，客户端事件 nodeclick 以及其处理函数 onTree1NodeClick：

```
<script>
function onTree1NodeClick(event, nodeId) {
    var tree = this;
    if ($.inArray(nodeId, ["zhumadian", "suiping", "xiping"]) >= 0) {
        var nodeData = tree.getNodeData(nodeId);
        // 触发后台事件
        __doPostBack('', 'Tree1_NodeClick$' + nodeId);
    }
}
</script>
```

在上述 JavaScript 代码中，我们将树节点的点击事件限制在"zhumadian", "suiping", "xiping"三个节点，然后触发后台事件（__doPostBack），注意如何使用$符号来分割事件名称（Tree1_NodeClick）和事件参数（nodeId）。

后台的捕获代码遵循相似的逻辑：

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
    }
    else
    {
        string[] args = GetRequestEventArguments(); // 此函数所在文件：PageBase.cs
        switch (args[0])
        {
            case "Tree1_NodeClick":
                string nodeId = args[1];
                labResult.Text = String.Format("你点击了树节点：{0}（{1}）", nodeId,
Tree1.FindNode(nodeId).Text);
                break;
        }
    }
}
```

这里我们调用 GetRequestEventArguments()获取切分后的回发参数，其中第二个参数即是点击的树节点标识符。

自定义回发参数（JSON 对象）

有时我们需要向后台传递多个参数，此时将多个参数封装为一个 JSON 字符串是最简洁和稳妥的做法。下面通过一个示例来演示多参数的用法。

示例：https://forms.fineui.com/#/Grid/RowClick

```
<f:Grid ID="Grid1" ShowBorder="true" ShowHeader="true" Title="表格" ...>
    <Columns>
        ...
    </Columns>
    <Listeners>
        <f:Listener Event="rowclick" Handler="onGrid1RowClick"></f:Listener>
    </Listeners>
</f:Grid>
```

视图文件定义了一个表格控件，客户端行点击事件 rowclick 以及相应的事件处理函数 onGrid1RowClick：

```
<script>
function onGrid1RowClick(event, rowId) {
    var grid1 = F.ui.Grid1;
    var rowData = grid1.getRowData(rowId);
    var rowEl = grid1.getRowEl(rowId);
    var column = grid1.getColumn(grid1.getSelectedCell()[1]);

    // 触发后台事件
    __doPostBack('', 'Grid1_RowClick$' + encodeURIComponent(JSON.stringify({
        rowId: rowData.id,
        rowText: rowData.text,
        rowIndex: rowEl.index(),
        columnText: column.text
    })));
}
</script>
```

客户端点击事件中，我们首先通过表格 API 函数获取当前行点击的行数据，行的 DOM 节点以及所在的列，然后将这些数据封装成一个 JavaScript 对象，接着使用 JSON.stringify 将其转换为字符串，最后使用 JavaScript 原生函数 encodeURIComponent 对其进行 URL 编码（防止参数中带有特殊字符，比

如我们约定的分隔符$)。

可以打开浏览器的调试工具，查看行点击时 HTTP 的请求数据：

对$分隔符的后半部分进行解码，我们可以看出真实传递的数据：

在后台的捕获处理中，我们只需要做相应的解码，整体的代码结构和之前的非常类似：

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
    }
    else
    {
        string[] args = GetRequestEventArguments(); // 此函数所在文件：PageBase.cs
        switch (args[0])
        {
            case "Grid1_RowClick":
                var param = JObject.Parse(HttpUtility.UrlDecode(args[1]));
                ShowNotify(String.Format("你点击了第 {0} 行，行 ID：{1}，姓名：{2}，列：{3}",
                    param.Value<int>("rowIndex") + 1,
                    param.Value<string>("rowId"),
                    param.Value<string>("rowText"),
                    param.Value<string>("columnText")));
                break;
        }
    }
}
```

其中，HttpUtility.UrlDecode()函数对应于客户端的 encodeURIComponent()，而 JObject.Parse()函数用于将字符串表示的 JSON 数据为 JObject 对象，对应于客户端的 JSON.stringify()函数。

小结

我们为 FineUICore 的 WebForms 模式引入了大家所熟悉的__doPostBack 函数，在自定义回发时，只需要关注第三个参数，可以传入单个参数或者多个参数，多个参数建议先转换为 JSON 数据，不仅代码结构清晰，而且方便后台解析，唯一需要注意的是对 JSON 数据进行 URL 编码。

后台捕获只需要在 Page_Load 事件中处理即可，可以直接调用基类中的 GetRequestEventArgument()函数获取传入的回发参数。现在大家应该对!IsPostBack 的逻辑判断有了更深的理解。

13. 注册客户端脚本（RegisterStartupScript）

相信 FineUIPro 的老客户对这个函数也不陌生，用于向页面注册（或者说是执行）一段脚本，可以用于页面第一次加载，也可以用于页面回发。

在 FineUICore 中，我们同样引入了相同的函数。由于 PageContext 同样存在于 Microsoft.AspNetCore.Mvc.RazorPages 命名控件下面，为了避免命名冲突，我们在基类中引入此函数。

```
public void RegisterStartupScript(string scripts)
{
    FineUICore.PageContext.RegisterStartupScript(scripts);
}
```

页面第一次加载时注册脚本

在 Page_Load 事件中调用 RegisterStartupScript()函数，用于向页面输出一段脚本，以便客户端 JavaScript 代码调用。比如下面这个示例，在后台定义了两个变量，并输出到前台页面，方便前台 JavaScript 代码调用。

示例：https://forms.fineui.com/#/GridEditor/Dynamic

```
private int _minYear = 2014;
private int _maxYear = 2017;

private void LoadData()
{
    ...
    RegisterStartupScript(String.Format("window._MINYEAR={0};window._MAXYEAR={1};",
_minYear, _maxYear));
}
```

查看浏览器端的页面源代码，你会发现这两个变量会在自定义脚本之前定义，方便调用。

```
                id: 'labResult',
                renderTo: '#labResult_wrapper',
                encodeText: false,
                value: ''
            });
            window._MINYEAR = 2014;
            window._MAXYEAR = 2017;
        });
    </script>
    <script type="text/javascript" src="/res/js/common.js?v11.0.0"></script>
    <script src="/res/js/grid.js"></script>
    <script>

        function updateTotalCell(grid, rowId) {
            var me = grid
              , total = 0;
            for (var i = _MINYEAR; i <= _MAXYEAR; i++) {
                total += me.getCellValue(rowId, 'Year_' + i);
            }

            me.updateCellValue(rowId, 'TotalFee', total);
        }
```

页面回发时注册脚本

在页面回发时，大部分时候我们可以通过修改控件属性来改变页面上控件的状态，但并非所有控件属性都可以在后台改变。对于一些特殊情况，我们需要自行输出一段脚本，通过调用客户端控件的 API 函数来达到目的，或者调用事先写好的客户端函数。

调用客户端 API-树控件的延迟加载

示例：https://forms.fineui.com/#/Tree/TreeLazyLoad

由于无法通过改变树控件属性的方式实现这个功能，我们提供了 GetLoadDataReference()函数来生成一段延迟加载某个节点数据的脚本，并通过 RegisterStartupScript()函数将其向前台注册：

```
protected void Tree1_NodeLazyLoad(object sender, TreeNodeEventArgs e)
{
    List<TreeNode> nodes = DynamicAppendNode(e.NodeID);

    RegisterStartupScript(Tree1.GetLoadDataReference(e.NodeID, nodes));
}
```

这个事件会向前台页面注册一段 JavaScript，通过浏览器的调试工具，我们看到类似的输出：

```
F.ui.Tree1.loadData('zhumadian', F.f_treeData([{
    "f0": "遂平县（延迟加载）",
    "f2": "suiping"
}, {
    "f0": "西平县",
    "f1": 1,
    "f2": "xiping"
}]));
```

调用客户端自定义函数-禁用加载更多按钮

示例：https://forms.fineui.com/#/GridPaging/DatabaseMoreFlow

在这个例子中，用户点击“加载更多...”按钮会向表格追加一页数据，当所有数据都已经展示完毕，我们需要将此按钮禁用并修改文字为“全部加载完毕”。由于这个按钮是在页面第一次加载时通过 JavaScript 脚本创建的，所有无法在后台通过服务器端控件属性来改变，只能通过执行自定义脚本来实现。

先看下如何创建的“加载更多...”按钮：

```
<script>
F.ready(function () {
    var grid1 = F.ui.Grid1;
    var tableEl = grid1.el.find('.f-grid-bodyct .f-grid-table');
    tableEl.after('<div class="morebutton"><a href="javascript:;">加载更多...</a></div>');
    ...
});
</script>
```

在上述代码中，F.ready()回调表示页面已经渲染完毕，此时我们找到表格 Grid1 的.f-grid-table 节点，然后在这个 DOM 节点的后面追加一个链接按钮。可以在浏览器的调试工具中查看相应的 DOM 节点位置。

事先在页面视图中定义一个禁用此按钮的 JavaScript 函数 disableMoreButton：

```
<script>
var morebuttonSelector = '.f-grid-bodyct >.morebutton a';

function disableMoreButton() {
    var grid1 = F.ui.Grid1;
    // 禁用链接
    var morebuttonEl = grid1.el.find(morebuttonSelector);
    morebuttonEl.attr('disabled', true).removeAttr('href').text('全部加载完毕');
}
</script>
```

在后台的事件处理中，当发现已经是最后一页的数据时，调用前台定义的 disableMoreButton()函数来禁用此按钮：

```
private void doMoreClick()
{
    var dataIndex = Grid1.Attributes.Value<int>("data-index");
    dataIndex++;
    ...
    if (dataIndex == pageCount - 1)
    {
        RegisterStartupScript("disableMoreButton();");
    }
}
```

小结

RegisterStartupScript()函数作为 FineUICore 默认代码实现的一个有益补充，增加了代码的灵活性，也扩展了 FineUICore 的适用范围，对于一些复杂的需求或者需要和第三方 JavaScript 组件集成的场景，也能轻松应对。建议 FineUICore 的高阶用户灵活掌握，有时能实现意想不到的效果。

14. 动态创建控件

启用 WebForms 开发模式的 FineUICore 项目中，动态创建控件的代码需要放在 Page_Load 事件中，

非常直观易懂，下面通过不同的示例来演示动态创建控件的过程。

动态创建表格列

动态创建表格列的示例非常简单，新建 GridColumn 对象（可以是 RowNumberField、RenderField、RenderCheckField 等，如果是多表头，需要递归创建 GroupField 和她的 Columns 属性），并逐个添加到表格的 Columns 属性即可。

示例：https://forms.fineui.com/#/Grid/DynamicColumns

```
protected void Page_Load(object sender, EventArgs e)
{
    if(!IsPostBack)
    {
        LoadData();
    }
}

private void LoadData()
{
    Grid1.Columns.Clear();
    foreach(var column in CreateGridColumns())
    {
        Grid1.Columns.Add(column);
    }
    Grid1.DataSource = DataSourceUtil.GetDataTable();
    Grid1.DataBind();
}
```

其中 CreateGridColumns 返回一个 GridColumn 的列表对象，可以转到示例查看相关源代码。

动态创建表单字段-控件实例声明与回发事件

示例：https://forms.fineui.com/#/Form/FormDynamic

```
protected void Page_Load(object sender, EventArgs e)
{
    if(!IsPostBack)
    {
        LoadData();
    }
}

private void LoadData()
{
    var tbxUser = new FineUICore.TextBox();
    tbxUser.ID = "tbxUserName";
    ...
    FormRow1.Items.Add(tbxUser);

    var ddlGender = new FineUICore.DropDownList();
    ddlGender.ID = "ddlGender";
    ...
    // 添加后台事件处理函数
    ddlGender.Events.Add(new Event("change", "ddlGender_SelectedIndexChanged"));
    FormRow1.Items.Add(ddlGender);
}
```

这个示例动态创建了一个文本输入框和一个下拉列表控件，并为下拉列表控件绑定了一个后台事件（ddlGender_SelectedIndexChanged），用于响应下拉列表的选择项改变事件（change）。

下面看下这个后台事件处理函数：

```
protected void ddlGender_SelectedIndexChanged(object sender, EventArgs e)
{
    ShowNotify("选择的性别：" + ddlGender.Text);
}
```

上述代码有点问题，标识符为 ddlGender 的下拉列表控件是通过后台代码创建的，因此.cshtml.designer.cs 文件中没有这个控件的声明，因此上述代码会出现引用错误。

```
0 个引用
protected void ddlGender_SelectedIndexChanged(object sender, EventArgs e)
{
    ShowNotify("选择的性别: " + ddlGender.Text);
}

0 个引用
```

CS0103: 当前上下文中不存在名称"ddlGender"

显示可能的修补程序 (Alt+Enter或Ctrl+.)

对于这种错误，我们需要在页面模型文件中自定义 ddlGender 变量即可。在页面回发时，FineUICore 会先查找是否存在名为 ddlGender 的声明，并将从客户端恢复的下拉列表控件实例赋值给这个变量，然后再调用事件处理函数。

动态创建按钮与按钮菜单-递归生成多层菜单

创建多层菜单涉及到递归调用，程序稍微有点复杂，需要大家认真理解。

示例：https://forms.fineui.com/#/Toolbar/MenuDynamic

数据源为一个 menu.xml 文件：

```
<?xml version="1.0" encoding="utf-8" ?>
<menu>
  <menuItem text="中国科学技术大学" navigateurl="http://www.ustc.edu.cn/">
    <menuItem text="化学与材料科学学院" navigateurl="http://scms.ustc.edu.cn/" />
    <menuItem text="管理学院" navigateurl="http://business.ustc.edu.cn/zh_CN/" >
      <menuItem text="工商管理系" navigateurl="http://is.ustc.edu.cn/" />
      <menuItem text="统计与金融系"  navigateurl="http://stat.ustc.edu.cn/" />
    </menuItem>
  </menuItem>
</menu>
```

首先从根节点创建按钮对象：

```
private void LoadData()
{
    XmlDocument doc = LoadXml("~/wwwroot/content/toolbar/menu.xml");

    // 根节点（中国科学技术大学）
    XmlNode node = doc.DocumentElement.ChildNodes[0];
    FineUICore.Button btn = new FineUICore.Button();
    btn.Text = node.Attributes["text"].Value;
    Toolbar1.Items.Add(btn);

    ResolveMenu(btn, node.ChildNodes);
}
```

然后递归创建菜单对象：

```
private void ResolveMenu(ControlBase btn, XmlNodeList nodes)
{
    PropertyInfo menuInfo = btn.GetType().GetProperty("Menu");
    Menu menu = menuInfo.GetValue(btn, null) as Menu;

    foreach (XmlNode node in nodes)
    {
        XmlAttribute attrURL = node.Attributes["navigateurl"];
        if (attrURL != null)
        {
            FineUICore.MenuHyperLink lnk = new FineUICore.MenuHyperLink();
            lnk.Text = node.Attributes["text"].Value;
            lnk.NavigateUrl = attrURL.Value;
            lnk.Target = "_blank";
            menu.Items.Add(lnk);

            if (node.ChildNodes.Count > 0)
            {
                ResolveMenu(lnk, node.ChildNodes);
            }
        }
    }
}
```

上述代码有点小技巧，注意 ResolveMenu 的第一个参数是 FineUICore.ControlBase，而不是 FineUICore.Button 类型，这是为什么?

因为在递归时，这个函数同样需要接受菜单项 FineUICore.MenuHyperLink 类型，虽然 Button 和 MenuHyperLink 都有相同的 Menu 对象，但却是两个不同的类型，因此我们需用使用反射来获取她的 Menu 对象（GetType().GetProperty("Menu")）。

小结

动态创建控件也属于高阶话题，适合于复杂的应用需求。创建控件的过程非常直观易懂，无外乎先新建（new）一个控件实例，设置实例属性，然后将其添加到父控件的集合属性上（比如 Items、Nodes、Columns 等）。

有两点需要特别注意，一个就是如何动态创建服务器端事件（Event 对象）和客户端事件（Listener），

另一个就是如何在页面模型中声明动态创建的控件，以便在回发事件中使用。

15. DataKeyNames 和 DataKeys 属性的回归

RazorPages 中获取行信息-自定义 JavaScript 代码

FineUICore 的 RazorPages 开发模式下，如果想获取选中行信息（比如行 ID、行中某些列的值），只能通过 JavaScript 获取后传入后台。下面通过一个示例说明。

示例：https://pages.fineui.com/#/Grid/CheckAll

在视图页面中，我们定义了一个表格控件，一个按钮控件来获取表格选中行信息。

```
<f:Grid ID="Grid1" ...>
    <Columns>
        ...
    </Columns>
</f:Grid>
<f:Button Text="选中了哪些行" ID="Button1"
    OnClick="@Url.Handler("Button1_Click")"
    OnClickParameter1="@(new Parameter("selected", "getGridSelectedRows()"))">
</f:Button>
```

注意按钮的点击事件需要使用 JavaScript 代码获取选中行信息：

```
<script>
function getGridSelectedRows() {
    var result = [], grid = F.ui.Grid1;

    $.each(grid.getSelectedRows(true), function (index, item) {
        var itemArr = [];
        itemArr.push(item.id);
        itemArr.push(item.text);
        itemArr.push(item.values.Gender);
        itemArr.push(item.values.Major);

        result.push(itemArr);
    });

    return F.toJSON(result);
}
</script>
```

后台事件处理函数：

```
public IActionResult OnPostButton1_Click(JArray selected)
{
    StringBuilder sb = new StringBuilder();
    sb.Append("<table class=\"result\"><tr><th>ID</th><th>Text</th><th>性别</th><th>专业
</th></tr>");
    foreach (JArray item in selected)
    {
        sb.AppendFormat("<tr><td>{0}</td><td>{1}</td><td>{2}</td><td>{3}</td></tr>",
            item[0], item[1],
            Convert.ToInt32(item[2].ToString()) == 1 ? "男" : "女",
            item[3]);
    }
    sb.Append("</table>");
    ShowNotify("<raw>" + sb.ToString() + "</raw>", MessageBoxIcon.None);
    return UIHelper.Result();
}
```

上述代码虽然清晰直观，但美中不足的是我们需要对表格的 JS API 足够的熟悉，知道 getSelectedRows(true)返回包含行对象的选中行数组。对于数组中的每一个 item，我们还要了解 item.text 对应于表格的 DataTextField 属性，item.values.Gender 对应于表格 ColumnID 为 Gender

的列。

有没有更好的解决办法，经典的 WebForms 为我们树立了一个榜样，我们也同样为 FineUICore 的 WebForms 开发模式引入了同样的解决方案。

WebForms 中获取行信息-简单直观

实现相同的功能，在 WebForms 开发模式的 FineUICore 项目中，我们只需要定义一个 DataKeyNames 属性即可。

示例：https://forms.fineui.com/#/Grid/CheckAll

视图页面中只需要定义 DataKeyNames 属性，无需自定义 JavaScript 代码。

```
<f:Grid ID="Grid1" ...
    DataKeyNames="@(new string[] { "Id", "Name", "Gender", "Major" })">
    <Columns>
        ...
    </Columns>
</f:Grid>

<f:Button Text="选中了哪些行" ID="Button1" OnClick="Button1_Click"></f:Button>
```

后台事件处理中结合使用 SelectedRowIndexArray 和 DataKeys 就能获取选中的行信息。

```
protected void Button1_Click(object sender, EventArgs e)
{
    StringBuilder sb = new StringBuilder();
    sb.Append("<table class=\"result\"><tr><th>ID</th><th>Text</th><th>性别</th><th>专业
</th></tr>");
    foreach (int rowIndex in Grid1.SelectedRowIndexArray)
    {
        var dataKeys = Grid1.DataKeys[rowIndex];
        sb.AppendFormat("<tr><td>{0}</td><td>{1}</td><td>{2}</td><td>{3}</td></tr>",
            dataKeys[0],
            dataKeys[1],
            Convert.ToInt32(dataKeys[2].ToString()) == 1 ? "男" : "女",
            dataKeys[3]);
    }
    sb.Append("</table>");
    ShowNotify("<raw>" + sb.ToString() + "</raw>", MessageBoxIcon.None);
}
```

小结

实现相同的功能，WebForms 开发模式下的代码量少了一半左右，而且不需要自定义 JavaScript 的参与，不仅方便开发，更有利于代码维护。

可能还有那么一个小小的缺点需要注意，表格启用 DataKeyNames 属性，在页面回发时，会自动附加相关的行数据，使得网络上传负荷有所增加。正如我们前面所说，在网络带宽普遍富裕的今天，多几十 K 的网络流量不会对用户体验有任何影响，而带来的开发效率提升和维护成本的降低却是实实在在的。

FineUICore 始终和广大的开发人员站在一起，矢志不渝的为开发人员谋福利。还是那句老话：一切为了简单。

16. 下拉列表的 PersistItems 属性

在后台事件中，我们可以直接访问下拉列表控件的 Items 和 SelectedItemArray 属性，这为我们开发带来了方便。

但是也存在另外一种情况，无需在回发事件中访问 Items 属性，并且下拉列表的下拉项目比较多，如果每次回发都要维持 Items 属性，势必会增加网络传输量。我们特别为下拉列表增加了一个 PersistItems 属性，可以在不需要时关闭 Items 属性的状态维护（页面回发时）。

示例：https://forms.fineui.com/#/DropDownList/ShengShiXian

默认启用 PersistItems 属性，在页面上选中安徽省->合肥市，当前请求的上传数据量为 3.1K。

禁用下拉列表的 PersistItems 属性，重复相同的操作，当前请求的上传数据量为 0.6K。

由此可见，页面回发时减少的数据量还是很明显的。

但是在实际开发过程中，为了开发和维护方便，还是建议保持 PersistItems 的默认值（true），只有在需要项目优化的时候才考虑禁用此属性。

17. 禁用 WebForms 的例外情况

在启用 WebForms 的 FineUICore 项目中，我们是支持传统 RazorPages 写法的，无需做任何更改！这也方便我们将项目中的页面逐个转换为 WebForms 页面，而不会出现运行中断的情况。

但是对于那些确实用不上 WebForms 开发模式的页面，我们可以显式的设置 PageManager 的 EnableWebForms=false 来禁用某个页面的 WebForms 开发模式。

比如在线示例“其他->数据模型”中的页面都无需启用 WebForms 开发模式。因为在这些页面中，后台通过标识为 BindProperty 的属性来接收前台数据。

这些页面通过如下方式禁用 WebForms 开发模式，在视图文件中，找到 PageManager 实例并禁用 WebForms 即可：

```
@{
    var F = Html.F();
    // NoWebForms
    F.PageManager.EnableWebForms(false);
}
```

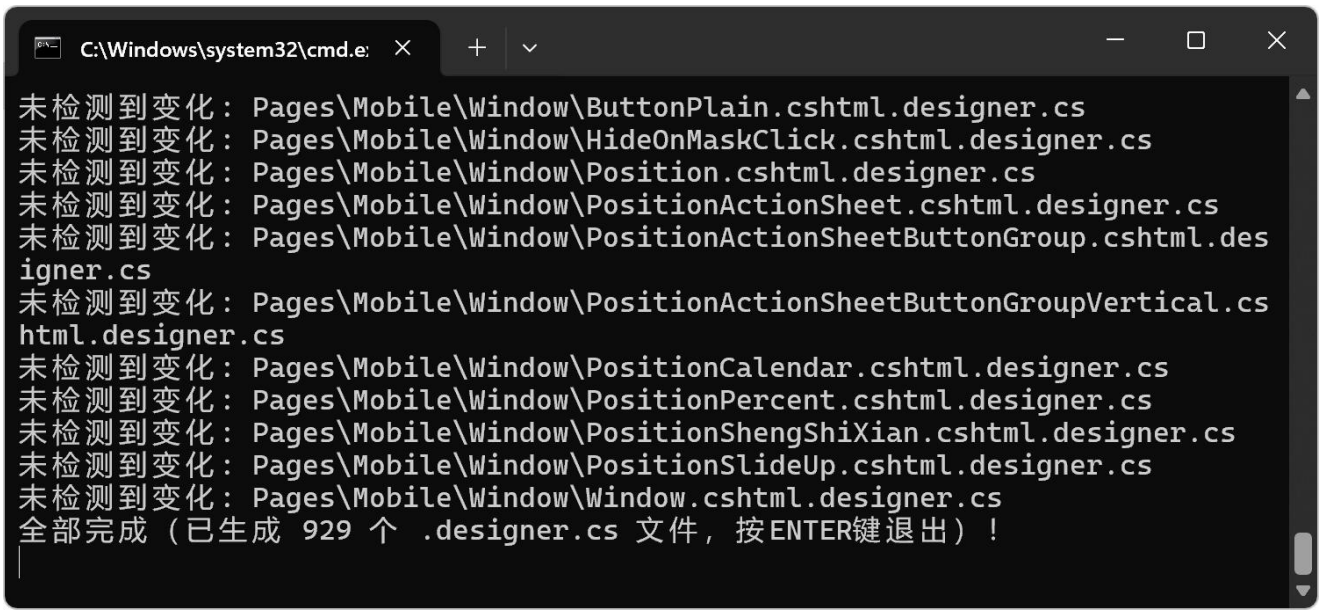
18. 设计时文件自动生成工具

在 FineUICore 的 WebForms 开发模式下，每一个页面 .cshtml 文件都对应一个 .cshtml.designer.cs 文件，按照经典 WebForms 的约定我们称之为设计时文件。

这个文件不需要自己编写，我们提供一个自动检测工具（在官网示例项目的根目录下），当检测到当前目录下.cshtml 文件改变时会自动生成设计时文件。

运行“自动生成设计时文件.exe”自动生成工具：

我们还提供了一键生成所有.cshtml 的设计时文件工具“一键生成当前目录下的全部设计时文件.bat”：



看一下生成的设计时文件：

```
namespace FineUICore.Examples.WebForms.Pages.Button
{
    public partial class ButtonModel
    {
        protected FineUICore.Button btnPrimary;
        protected FineUICore.Button btnChangeEnable;
        ...
    }
}
```

对于哪些禁用 WebForms 的页面，其实是不需要设计时文件的，我们只需要为.cshtml 文件添加“//NoWebForms”，自动化生成工具会自动跳过这些文件。

```
@{
    var F = Html.F();
    // NoWebForms
    F.PageManager.EnableWebForms(false);
}
```

19. 常用链接

官网首页：https://fineui.com/#webforms

在线演示：https://forms.fineui.com/

版本更新：https://fineui.com/versions/

经典案例：https://fineui.com/cases/

社区交流：https://fineui.com/fans/

企业版申请试用

请填写如下资料发送到邮箱：2877408506@qq.com

- 产品名称：FineUICore{WebForms 开发模式}
- 单位全称：XXX 单位
- 申请人邮箱：XXX
- 申请人 QQ：XXX
- 申请人姓名：XXX
- 申请人地址：XX 省 XX 市

20. 更新记录

本文档的更新记录：

2024-05-15 v1.1

1. 更新“禁用 WebForms 的例外情况”一节，简化代码。

2. 新增“设计时文件自动生成工具”一节。

3. 新增“常用链接”一节。

2024-05-10 v1.0

初始版本。

FineUICore 技术白皮书

全球首创！在 ASP.NET Core 中运行 WebForms 业务代码。

三生石上 2024-05-15 v1.1