

快速开始

前置条件

- 本SDK基于Windows的UI Automation开发，所以仅支持Windows 操作系统，并且需要将项目的 `TargetFramework` 修改成:目标框架-windows,如你在.net8下使用，请将`TargetFramework`修改成 `net8.0-windows`;
- .NET Framework 4.8+ 或 .NET 6.0+ (Windows)，支持.NET的框架有:net48;net481;net6.0-windows; net7.0-windows;net8.0-windows;net9.0-windows;net10.0-windows;
- 微信 PC 客户端已安装并运行,本 SDK 基于微信 PC 客户端(版本号:3.9.12.55)的 UI 结构开发，不同版本可能存在兼容性问题。

安装

通过 NuGet 安装:

```
dotnet add package WeChatAuto.SDK
```

WeChatAuto.SDK的初始化

WeChatAuto.SDK的初始化的通用格式为:

```
WeAutomation.Initialize(...,Action<WeChatConfig> options)
```

但根据应用依赖注入引入的情况分两种情况:

- 应用没有启用依赖注入

此种情况需要自行引入`dotnet add package Microsoft.Extensions.DependencyInjection`包,这样应用也有了依赖注入容器,并且根据情况初始化WeChatConfig配置对象，如下所示:

```
// 初始化WeAutomation服务
var serviceProvider = WeAutomation.Initialize(options =>
{
    options.DebugMode = true;    //开启调试模式，调试模式会在获得焦点时边框高亮，生产环境建议关闭
    //options.EnableRecordVideo = true;    //开启录制视频功能，录制的视频会保存在项目的运行目录下的
Videos文件夹中
});

using var clientFactory = serviceProvider.GetRequiredService<WeChatClientFactory>();
```

...下面是更多代码

注意：如果应用没有启用依赖注入，clientFactory的生命周期需要自行负责，或者应用退出的时候显式`clientFactory.Dispose()`，或者将clientFactory放入`using {}`代码块中自动释放

- 应用启用了依赖注入框架

此种情况，需要在应用`Services.BuildServiceProvider()`前进行初始化,并将应用的`Services`做为初始化的第一个参数，这样本SDK就集成在应用的依赖注入框架中，如下所示：

```
using Microsoft.Extensions.Hosting;
using WeChatAuto.Services;
using WeChatAuto.Components;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

var builder = Host.CreateApplicationBuilder(args);

WeAutomation.Initialize(builder.Services, options =>
{
    //开启调试模式，调试模式会在获得焦点时边框高亮，生产环境建议关闭
    options.DebugMode = true;
    //开启录制视频功能，录制的视频会保存在项目的运行目录下的Videos文件夹中
    //options.EnableRecordVideo = true;
});

//这里注入自己的服务（或者对象），如LLM服务等
builder.Services.AddSingleton<LLMService>();

var serviceProvider = builder.Services.BuildServiceProvider();
var clientFactory = serviceProvider.GetRequiredService<WeChatClientFactory>();
...更多代码
await builder.Build().RunAsync();
...
```

此种方式的初始化无须管理clientFactory的生命周期

基本使用

示例一：给好友（或群聊昵称）发送消息：

源码请参见: 项目根目录\Examples\demo01

- 步骤一：新建项目，如下所示:

```
dotnet new console -n demo01
```

- 步骤二：将demo01.csproj项目文件的net10.0修改成net10.0-windows,如下所示:

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net10.0-windows</TargetFramework>
  <ImplicitUsings>enable</ImplicitUsings>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

- 步骤三：安装依赖

```
dotnet add package WeChatAuto.SDK
dotnet add package Microsoft.Extensions.DependencyInjection
```

- 步骤四：项目demo01的Program.cs修改成如下:

```
using Microsoft.Extensions.DependencyInjection;
using WeChatAuto.Components;
using WeChatAuto.Services;

// 初始化WeAutomation服务
var serviceProvider = WeAutomation.Initialize(options =>
{
    options.DebugMode = true;    //开启调试模式，调试模式会在获得焦点时边框高亮，生产环境建议关闭
    //options.EnableRecordVideo = true;    //开启录制视频功能，录制的视频会保存在项目的运行目录下的
Videos文件夹中
});

using var clientFactory = serviceProvider.GetRequiredService<WeChatClientFactory>();
Console.WriteLine($"当前客户端打开的微信客户端为: {string.Join(", ",
clientFactory.GetWeChatClientNames())}, 共计{clientFactory.GetWeChatClientNames().Count}个微
信客户端。");
//获取当前打开的微信客户端名称列表
var clientNames = clientFactory.GetWeChatClientNames();
//获取第一个微信客户端
var wxClient = clientFactory.GetWeChatClient(clientNames.First());
```

```
//通过微信客户端发送消息给好友昵称AI.Net, 测试时请把AI.Net修改成自己的好友昵称
wxClient?.SendWho("AI.Net", "你好, 欢迎使用AI.Net微信自动化框架!");
```

注意:

1. 本项目仅支持 Windows 系统, 请务必将项目文件的 TargetFramework 设置为 netxx.0-windows (如 net10.0-windows), 否则编译时会出现警告。后续不再赘述。
2. 如果是手动管理WeChatClientFactory,请在应用结束时运行clientFactory.Dispose(),或者像示例代码一样将代码放入using块自动释放,如果把WeChatAuto.SDK加入您的依赖注入容器, 则不存在此问题。
3. WeAutomation.Initialize()方法有两个重载, 分别适用于: 加入外部依赖注入与使用内部依赖注入。

示例二 - 演示监听好友 (或者群聊昵称) 的消息,使用消息上下文获取消息并回复,并且还演示了如何通过依赖注入获取消息上下文的注入对象,执行自己的业务逻辑:

源码请参见: [项目根目录\Examples\demo02](#)

- 前置步骤: 安装依赖

```
dotnet add package WeChatAuto.SDK
dotnet add package Microsoft.Extensions.Hosting
```

- 将项目demo02的Program.cs修改成如下

```
using Microsoft.Extensions.Hosting;
using WeChatAuto.Services;
using WeChatAuto.Components;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;

var builder = Host.CreateApplicationBuilder(args);

WeAutomation.Initialize(builder.Services, options =>
{
    //开启调试模式, 调试模式会在获得焦点时边框高亮, 生产环境建议关闭
    options.DebugMode = true;
    //开启录制视频功能, 录制的视频会保存在项目的运行目录下的Videos文件夹中
    //options.EnableRecordVideo = true;
});
```

//这里注入自己的服务（或者对象），如LLM服务等

```
builder.Services.AddSingleton<LLMService>();
```

```
var serviceProvider = builder.Services.BuildServiceProvider();
```

```
var clientFactory = serviceProvider.GetRequiredService<WeChatClientFactory>();
```

// 得到名称为"Alex"的微信客户端实例，测试时请将AI.net替换为你自己的微信昵称

```
var client = clientFactory.GetWeChatClient("Alex");
```

// 监听微信群测试11

```
await client.AddMessageListener("测试11", (messageContext) =>
```

```
{
```

```
    var index = 0;
```

//打印收到最新消息

```
    foreach (var message in messageContext.NewMessages)
```

```
    {
```

```
        index++;
```

```
        Console.WriteLine($"收到消息: {index}: {message.ToString()}");
```

```
        Console.WriteLine($"收到消息: {index}: {message.Who}: {message.MessageContent}");
```

```
    }
```

//打印收到所有消息的后十条

```
    var allMessages = messageContext.AllMessages.Skip(messageContext.AllMessages.Count  
- 10).ToList();
```

```
    index = 0;
```

```
    foreach (var message in allMessages)
```

```
    {
```

```
        index++;
```

```
        Console.WriteLine($"...收到所有消息的后10条之第{index}条:
```

```
{message.Who}: {message.MessageContent}");
```

```
        Console.WriteLine($".....详细之第{index}条: {message.ToString()}");
```

```
    }
```

//是否有人@我

```
    if (messageContext.IsBeAt())
```

```
    {
```

```
        var messageBubble = messageContext.MessageBubbleIsBeAt().FirstOrDefault();
```

```
        if (messageBubble != null)
```

```
        {
```

```
            messageContext.SendMessage("我被@了!!!! 我马上就回复你!!!! ", new
```

```
List<string> { messageBubble.Who });
```

```
        }
```

```
    else
```

```
    {
```

```
        messageContext.SendMessage("我被@了!!!! 我马上就回复你!!!! ");
```

```
    }
```

```
}
```

//是否有人引用了我的消息

```
    if (messageContext.IsBeReferenced())
```

```

{
    messageContext.SendMessage("我被引用了!!!!");
}
//是否有人拍了拍我
if (messageContext.IsBeTap())
{
    messageContext.SendMessage("我被拍一拍了[微笑]!!!!");
}
if (!messageContext.IsBeAt() && !messageContext.IsBeReferenced()
&& !messageContext.IsBeTap())
{
    //回复消息，这里可以引入大模型自动回复
    messageContext.SendMessage($"我收到了
{messageContext.NewMessages.FirstOrDefault()?.Who}的消息:
{messageContext.NewMessages.FirstOrDefault()?.MessageContent}");
}
//可以通过注入的服务容器获取你注入的服务实例，然后调用你的业务逻辑,一般都是LLM的自动回复逻辑
var llmService = messageContext.ServiceProvider.GetRequiredService<LLMService>();
llmService.DoSomething();
});

var app = builder.Build();
await app.RunAsync();

/// <summary>
/// 一个包含LLM服务的Service类，用于注入到MessageContext中
/// </summary>
public class LLMService
{
    private ILogger<LLMService> _logger;
    public LLMService(ILogger<LLMService> logger)
    {
        _logger = logger;
    }
    public void DoSomething()
    {
        _logger.LogInformation("这里是注入的服务实例，可以在这里编写你的业务逻辑 ");
    }
}

```

前置步骤跟Demo01一致,可以通过messageContext对象执行各种操作,也可以通过messageContext对象获得依赖注入容器，获取自己的对象，执行自己的业务逻辑;

示例三 - MCP Server的使用 - 以vscode为例讲解

- 进入源码的.vscode\mcp.json,修改配置如下:

```
{
  "servers": {
    "wechat_mcp_server": {
      "type": "stdio",
      "command": "dotnet",
      "args": [
        "run",
        "--project",
        "改成你的WeChatAuto.MCP.csproj的路径"
      ]
    }
  }
}
```

- 在mcp.json页面点击"Start"按钮启动mcp server
- 启动GitHub Copilot Chat,在Chat页提问: 请帮我给微信好友:AI.Net发送消息: Hello world!

虚拟码键表

键码	键名	描述	键码	键名	描述
8	Backspace	退格键	88	X	
9	Tab	Tab键	89	Y	
13	Enter	回车键	90	Z	
16	Shift	Shift键	91	Left Win	左Windows键
17	Ctrl	Ctrl键	92	Right Win	右Windows键
18	Alt	Alt键	93	Apps	菜单键
19	Pause	暂停键	96	Num 0	小键盘0键
20	Caps Lock	大小写切换键	97	Num 1	小键盘1键
27	Esc	退出键	98	Num 2	小键盘2键
32	Space	空格键	99	Num 3	小键盘3键
33	Page Up	向上翻页键	100	Num 4	小键盘4键
34	Page Down	向下翻页键	101	Num 5	小键盘5键
35	End	结尾键	102	Num 6	小键盘6键
36	Home	起始键	103	Num 7	小键盘7键
37	Left	左移键（左箭头）	104	Num 8	小键盘8键
38	Up	上移键（上箭头）	105	Num 9	小键盘9键
39	Right	右移键（右箭头）	106	Num *	小键盘*键
40	Down	下移键（下箭头）	107	Num +	小键盘+键
44	Print Screen	截屏键	108	Num Enter	小键盘回车键
45	Insert	插入键	109	Num -	小键盘-键
46	Delete	删除键	110	Num .	小键盘.键

键码	键名	描述	键码	键名	描述
48	0		111	Num /	小键盘/键
49	1		112	F1	
50	2		113	F2	
51	3		114	F3	
52	4		115	F4	
53	5		116	F5	
54	6		117	F6	
55	7		118	F7	
56	8		119	F8	
57	9		120	F9	
65	A		121	F10	
66	B		122	F11	
67	C		123	F12	
68	D		144	Num Lock	小键盘切换键
69	E		145	Scroll Lock	
70	F		160	Left Shift	左Shift键
71	G		161	Right Shift	右Shift键
72	H		162	Left Ctrl	左Ctrl键
73	I		163	Right Ctrl	右Ctrl键
74	J		164	Left Alt	左Alt键
75	K		165	Right Alt	右Alt键
76	L		186	;	::

键码	键名	描述	键码	键名	描述
77	M		187	=	=+
78	N		188	,	,<
79	O		189	-	-_
80	P		190	.	.>
81	Q		191	/	/?
82	R		192	~	
83	S		219	[[{
84	T		220	\	
85	U		221]]}]
86	V		222	'	""
87	W				

FAQ

你提，我来回答 😊

Namespace SKM

Classes

[Config](#)

参数配置类

[DpiAwareness](#)

[KMSimulatorService](#)

键鼠模拟器服务 封装skm.dll的函数，提供键鼠模拟器服务 x64环境需要复制x64\skm.dll，x86环境需要复制x86\skm.dll到当前目录，意思是：x64环境需要使用x64\skm.dll，x86环境需要使用x86\skm.dll 本服务类会自动根据当前环境复制DLL到当前目录，可以无感知使用

[SkmcCore](#)

注意: x86 和 x64 的 dll 是不同的，需要根据实际情况选择

x86 的 dll 是 x86\skm.dll，x64 的 dll 是 x64\skm.dll

使用 `KMSimulatorService` 服务类可以无感知使用，会自动根据当前环境复制 DLL 到当前目录
具体 `KMSimulatorService` 类使用请参考 [KMSimulatorService](#)