日常随笔

- linq与 lamda
- C#系列
- DSTV-NC1解读
- EF分库、分表、分模式、多租户
- Maui学习记录
- 进程
- 进程通信
- 命令
- 高薪必备-锁
- 高薪必备-线程
- 高薪必备-大纲
- 互联网项目必备投入
- 实践经验：互联网项目起步指南
- winfrom与 wpf
- visual studio 常用开发命令
- git常规
- 生活小本本
- 专利软著系列

linq lamda

```
//                                    0     <-1
“     ”.IndexOf(“     ”, StringComparison.OrdinalIgnoreCase)
```

C#系列

判断文件是否被占用

```
public class FileStatus    {
[DllImport("kernel32.dll")]
private static extern IntPtr _lopen(string lpPathName, int iReadWrite);
[DllImport("kernel32.dll")]
private static extern bool CloseHandle(IntPtr hObject);
private const int OF_READWRITE = 2;
private const int OF_SHARE_DENY_NONE = 0x40;
private static readonly IntPtr HFILE_ERROR = new IntPtr(-1);
public static int FileIsOpen(string fileFullName)
{
if (!File.Exists(fileFullName))
{
return -1;// 不存在文件
}
  IntPtr handle = _lopen(fileFullName, OF_READWRITE | OF_SHARE_DENY_NONE);
  if (handle == HFILE_ERROR)
  {
    return 1;// 表示被占用
  }
  CloseHandle(handle);
  return 0;
}
}
```

DSTV-NC1解读

1. 代码块含义

DSTV 文件通过代码块（如 `BO`、`SI`、`AK` 等）分组描述零件的加工信息，每个代码块对应特定工艺或几何特征。以下是常见代码块的定义（引用自 DSTV 标准及 Tekla ProNest 文档）：

G代码标识	意义	描述
ST	文件块开始	整个加工文件起始
EN	文件块结束	整个加工文件结束
BO	孔打开块	定义孔加工的起始（如钻孔、冲孔）
SI	编号打开块	用于零件编号或标识信息（如零件名称、编号）
AK	外轮廓打开块	描述零件的外部轮廓（如矩形、圆形）
IK	内轮廓打开块	描述零件的内部轮廓（如开孔、凹槽）
PU	粉末打开块	与粉末切割或标记相关操作（如等离子切割）
KO	标记打开块	定义标记（如刻字、编号）的位置和内容
SC	切割（锯、切割）打开块	描述切割路径（直线、圆弧等）
TO	公差打开块	定义加工公差范围（如尺寸允许偏差）
UE	倾斜打开块	表示倾斜加工（如坡口角度）
PR	轮廓描述打开块	定义复杂轮廓的几何参数（如折线、多段曲线）
KA	弯曲打开块	描述弯曲工艺（如折弯角度、半径）

注：代码块后可附加数字（如 `BO1`、`AK2`），用于区分同一工艺下的多个操作（如多个孔或轮廓）。

2.截面类型

截面类型按照 DSTV 标准进行命名 ：

G代码标识	意义	对应截面库
I	H型钢系列	H，HP，HN，HM，HW，HT，I，BH，HI，PHI，WI
L	L型材系列	L，BL
U	U型材系列	BLU，U，C
B	板材	BL，BPL，FL，FLT，FPL，PL，PLATE，PLT，TANKO
RO	空心圆管	PIP，CFCHS，CHS，EPD，O，PD，TUBE
RU	实心圆钢	D，ELD，ROD
M	矩形管（空腹）	CFRHS，P，RHS，SHS，TUB，
C	C型材系列	CC
T	T型材系列	T，TN，TM，TW，
SO	Z 截面及所有其它类型的截面	BOX，HK，WB，BL，Z，WQ，HQ，ZZ，CW，CU，EB，BF，ESPD，SPD，EC，ED，EE，EF，EZ，EW，

3.零件坐标系

DSTV 文件中用于描述零件面的单个字母。

字母	零件面
v	前面
o	顶部
u	底部

□□	□□□
h	□□

EF分库、分表、分模式、多租户

Maui□□□□

MauiProgram□□□□

```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder.UseMauiApp<App>();
        builder.ConfigureFonts(fonts =>
        {
            fonts.AddFont("OpenSans-Regular.ttf", "OpenSansRegular");
            fonts.AddFont("OpenSans-Semibold.ttf", "OpenSansSemibold");
        });
        builder.UsePrism(prism =>
        {
            prism.ConfigureServices(service =>
            {
                RestClientOptions restClientOptions = new()
                {
                    //restClientOptions.RemoteCertificateValidationCallback += (sender, cert, chain, error) => true;
                    BaseUrl = new Uri("https://dynamicapi.lingyanspace.com"),
                    MaxRedirects = 5,
                    Timeout = TimeSpan.FromMinutes(3),
                    ThrowOnAnyError = true
                };

                RestClient restClient = new(restClientOptions);
                service.AddSingleton(restClient);
                service.AddSingleton<ICourseService, CourseService>();
            });
            prism.RegisterTypes(container =>
            {
                //□□  MainPage□□□
                container.RegisterForNavigation<MainPage>();
                container.RegisterForRegionNavigation<HomeView, HomeViewModel>();
```

```
            });
            prism.OnInitialized((container) =>
            {
                var regionManager = container.Resolve<IRegionManager>();
                regionManager.RegisterViewWithRegion("MainRegion", nameof(HomeView));
            });

            //□□□□□□
            prism.CreateWindow(navigationService =>
navigationService.NavigateAsync($"/{nameof(MainPage)}"));
        });

#if DEBUG
        builder.Logging.AddDebug();
#endif

        return builder.Build();
    }
}
```

□□□□□□

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage
    x:Class="LearnMirro.MainPage"
    xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:prism="http://prismlibrary.com"
    Title="MainPage"
    Background="#eee">
    <ContentView prism:RegionManager.RegionName="MainRegion" />
</ContentPage>
```

进程

以下是针对 **进程相关知识** 的深入讲解，涵盖从基础到高阶的编程技术，结合 **C#** 和 **WPF** 的实际应用场景，帮助你达到高阶编程水平：

进程的核心概念与底层原理
1. **进程的本质**
- **操作系统视角**：
  - **地址空间**：每个进程拥有独立的虚拟地址空间，内存隔离（防止进程间直接访问内存）。
  - **资源隔离**：文件句柄、网络端口、环境变量等资源独立管理。
  - **进程控制块（PCB）**：操作系统内核维护的进程元数据，包括：
    - 程序计数器（PC）、寄存器状态。
    - 虚拟内存映射、线程列表、打开的句柄。
    - 进程优先级、CPU 时间片、I/O 状态。

2. **进程的创建与终止**
C# 中创建进程
```csharp
using System.Diagnostics;

// 启动外部进程（如计算器）
Process calculator = new Process();
calculator.StartInfo.FileName = "calc.exe";
calculator.Start();

// 启动带参数的进程
Process notepad = new Process();
notepad.StartInfo.FileName = "notepad.exe";
notepad.StartInfo.Arguments = "test.txt";
notepad.Start();

// 同步等待进程退出
notepad.WaitForExit();
```

进程终止
```csharp
// 强制终止进程
if (notepad.ProcessId > 0)
{
    notepad.Kill();
```

```
    notepad.WaitForExit();
}
```

---

3. **□□□□□□ IPC□ **
□□□□□□□□□□□□□□□□□□□□ IPC □□□ C# □□□

**3.1 □□□ Pipe□ **
- **□□□□ **□□□□□□□□□□□
- **□□□□ **□□□□□□□□□□□□□□

**□□□□□□□□□□□□□□ **
```csharp
// □□□□□□□□□□
using System.IO.Pipes;

NamedPipeServerStream pipeServer = new NamedPipeServerStream("MyPipe",
PipeDirection.InOut);
pipeServer.WaitForConnection();
byte[] buffer = new byte[1024];
pipeServer.Read(buffer, 0, buffer.Length);
string message = Encoding.UTF8.GetString(buffer);
Console.WriteLine($"Received: {message}");

// □□□□□□□□□
NamedPipeClientStream pipeClient = new NamedPipeClientStream(".", "MyPipe",
PipeDirection.InOut);
pipeClient.Connect();
byte[] data = Encoding.UTF8.GetBytes("Hello from client!");
pipeClient.Write(data, 0, data.Length);
```

**3.2 □□□□□ Shared Memory□ **
- **□□ **□□□ `MemoryMappedFile` □□□□□□□□□□□
- **C# □□ **□
```csharp
// □□□□□□□□□
using System.IO.MemoryMappedFiles;

var mmf = MemoryMappedFile.CreateNew("MySharedMemory", 1024);
using (var writer = mmf.CreateViewStream())
{
    byte[] data = Encoding.UTF8.GetBytes("Shared Data");
    writer.Write(data, 0, data.Length);
}
```

```
// 客户端读取共享内存
using (var mmf = MemoryMappedFile.OpenExisting("MySharedMemory"))
using (var reader = mmf.CreateViewStream())
{
    byte[] buffer = new byte[1024];
    reader.Read(buffer, 0, buffer.Length);
    string message = Encoding.UTF8.GetString(buffer);
    Console.WriteLine($"Received: {message}");
}
```

3.3 Windows 特有 IPC
- **窗口消息（HWND）**: 通过 `SendMessage` 或 `PostMessage` 传递消息。
- **内存映射文件（MapViewOfFile）**: Windows API 的底层共享内存实现。
- **命名事件（Event）**: 同步进程间操作。

```csharp
// 创建命名事件（Windows API）
using System.Runtime.InteropServices;

[DllImport("kernel32.dll")]
static extern IntPtr CreateEvent(IntPtr lpEventAttributes, bool bManualReset, bool bInitialState,
string lpName);

IntPtr hEvent = CreateEvent(IntPtr.Zero, true, false, "Global\\MyEvent");
```

---

4. **进程同步与安全**
4.1 进程同步
- **互斥量（Mutex）**:
  ```csharp
  using System.Threading;

  Mutex mutex = new Mutex(false, "Global\\MyMutex");
  mutex.WaitOne(); // 获取锁
  // 临界区操作
  mutex.ReleaseMutex(); // 释放锁
  ```

- **信号量（Semaphore）**:
  ```csharp
  Semaphore semaphore = new Semaphore(2, 2, "Global\\MySemaphore"); // 最大允许 2 个进程同时访问
  semaphore.WaitOne();
  // 资源访问
```

```
  semaphore.Release();
  ```

**4.2 进程安全 **
- **权限控制 **:
  ```csharp
  // 创建受保护的命名管道
  var pipeSecurity = new PipeSecurity();
  pipeSecurity.AddAccessRule(new PipeAccessRule("Everyone", PipeAccessRights.ReadWrite,
AccessControlType.Allow));
  var pipeServer = new NamedPipeServerStream("SecurePipe", PipeDirection.InOut, 1,
PipeTransmissionMode.Byte, pipeSecurity);
  ```

---

5. **进程的高级特性 **
**5.1 进程优先级与调度 **
```csharp
// 设置进程优先级
Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.High;

// 获取进程详细信息
foreach (Process process in Process.GetProcesses())
{
    Console.WriteLine($"PID: {process.Id}, Name: {process.ProcessName}, Priority:
{process.PriorityClass}");
}
```

**5.2 进程调试与注入 **
- **调试进程 **:
  ```csharp
  Process targetProcess = Process.Start("notepad.exe");
  targetProcess.EnableDebugging = true;
  ```

- **进程注入（需管理员权限） **:
  ```csharp
  // 使用 P/Invoke 调用 Windows API 注入 DLL（示例）
  [DllImport("kernel32.dll")]
  static extern IntPtr OpenProcess(uint dwDesiredAccess, bool bInheritHandle, int dwProcessId);

  [DllImport("kernel32.dll")]
  static extern IntPtr VirtualAllocEx(IntPtr hProcess, IntPtr lpAddress, uint dwSize, uint
flAllocationType, uint flProtect);
```

```
// 具体实现需结合 WriteProcessMemory 和 CreateRemoteThread
```

---

6. **WPF 中的进程管理**
6.1 异步进程启动
```csharp
// 在 WPF 中异步启动进程
private async void StartProcessAsync()
{
    var process = new Process { StartInfo = { FileName = "notepad.exe" } };
    await Task.Run(() => process.Start());
    MessageBox.Show("进程已启动");
}
```

6.2 进程与 UI 线程交互
```csharp
// 在后台进程完成时更新 UI
private void ProcessExited(object sender, EventArgs e)
{
    Dispatcher.Invoke(() =>
    {
        ResultTextBlock.Text = "进程已退出!";
    });
}

process.Exited += ProcessExited;
```

---

7. **进程的高级应用场景**
7.1 进程沙箱化
- **隔离危险操作**:
```csharp
// 启动沙箱进程
ProcessStartInfo psi = new ProcessStartInfo("sandbox.exe");
psi.UseShellExecute = false;
psi.CreateNoWindow = true;
psi.RedirectStandardOutput = true;

Process.Start(psi);
```

7.2 进程集群与负载均衡
```csharp
// 启动多个工作进程
for (int i = 0; i < Environment.ProcessorCount; i++)
{
    Process.Start("worker.exe", $"--id={i}");
}
```

7.3 进程监控与自动化
```csharp
// 监控特定进程
private void MonitorProcess()
{
    while (true)
    {
        foreach (var process in Process.GetProcessesByName("notepad"))
        {
            if (process.Responding == false)
            {
                process.Kill();
            }
        }
        Thread.Sleep(1000);
    }
}
```

---

8. **进程的底层实现与优化**
8.1 虚拟内存管理
- **内存页大小**: 通常为 4KB。
- **优化内存使用**:
  ```csharp
  // 避免内存泄漏
  using (var mmf = MemoryMappedFile.CreateNew("MyMMF", 1024))
  {
      // 使用共享内存后自动释放
  }
  ```

8.2 进程调度优化
- **实时进程**:
  ```csharp
  Process.GetCurrentProcess().PriorityClass = ProcessPriorityClass.RealTime;
  ```

  ```

#### **8.3 进程资源监控**
```csharp
// 监控进程 CPU 和内存使用率
private void MonitorProcessResources()
{
    Process target = Process.GetProcessById(1234);
    while (!target.HasExited)
    {
        Console.WriteLine($"CPU: {target.TotalProcessorTime}, Memory: {target.WorkingSet64}
bytes");
        Thread.Sleep(1000);
    }
}
```

---

9. **高阶技巧与陷阱**
9.1 进程间死锁预防
- **避免循环依赖**: 确保 IPC 资源的单向依赖。
- **超时机制**:
  ```csharp
  // 带超时的等待事件
  if (WaitHandle.WaitAny(new[] { mutex, timeoutEvent }) == 0)
  {
      // 成功获取锁
  }
  ```

9.2 内存泄漏检测
- **使用 `System.Diagnostics`**:
  ```csharp
  var performanceCounter = new PerformanceCounter("Process", "Working Set - Private",
Process.GetCurrentProcess().ProcessName);
  Console.WriteLine($"内存使用: {performanceCounter.NextValue()} bytes");
  ```

9.3 跨平台进程管理
```csharp
// 使用跨平台库（如 System.Diagnostics.Process）
Process.Start(new ProcessStartInfo("ls", "-l") { UseShellExecute = false });
```

```

10. **实战案例: WPF 多进程架构 **

**案例: 分布式计算服务 **

```csharp
// 主进程 ( WPF UI)
public partial class MainWindow : Window
{
    private List<Process> workers = new List<Process>();

    private void StartWorkers()
    {
        for (int i = 0; i < Environment.ProcessorCount; i++)
        {
            var worker = Process.Start("Worker.exe", $"--id={i}");
            workers.Add(worker);
        }
    }

    private void StopWorkers()
    {
        foreach (var worker in workers)
        {
            worker.Kill();
        }
    }
}

// 工作进程 ( Worker.exe)
static class Program
{
    static void Main(string[] args)
    {
        // 从命令行参数获取 ID
        int workerId = int.Parse(args[0].Split('=')[1]);

        // 执行计算任务
        while (true)
        {
            // 通过管道或共享内存返回结果
        }
    }
}
```

**[illegible] **
[illegible] C# [illegible] API [illegible] Windows
[illegible]
1. **[illegible] **[illegible]
2. **[illegible] IPC**[illegible]
3. **[illegible] **[illegible]
4. **[illegible] **[illegible]

[illegible]

进程通信

父子进程匿名管道通信

父进程

```
using (var pipeServer = new AnonymousPipeServerStream(PipeDirection.Out, HandleInheritability.Inheritable))
{
    Process childProcess = new Process
    {
        StartInfo = new ProcessStartInfo
        {
            FileName = "dotnet",
            Arguments = $"run --project Hub1 --no-build -- ClientProgram {pipeServer.GetClientHandleAsString()}",
            UseShellExecute = false,
            RedirectStandardOutput = true,
            RedirectStandardError = true
        }
    };

    childProcess.Start();

    using (StreamWriter writer = new StreamWriter(pipeServer) { AutoFlush = true })
    {
        await writer.WriteLineAsync("Hello from parent!");
    }
}
```

子进程

```
 if (args.Length > 0)
{
    string pipeHandle = args[0];

    using (var pipeClient = new AnonymousPipeClientStream(PipeDirection.In, pipeHandle))
    {
        using (StreamReader reader = new StreamReader(pipeClient))
        {
            string message = await reader.ReadLineAsync();
            Console.WriteLine($"收到父进程消息            : {message}");
```

```
        }
    }
}
```

普通命名管道通信

服务端

```
Console.WriteLine("pipe管道服务已启动            ...");

var pipeServer = new NamedPipeServerStream("mypipe", PipeDirection.InOut,
NamedPipeServerStream.MaxAllowedServerInstances, PipeTransmissionMode.Byte, PipeOptions.Asynchronous);
await pipeServer.WaitForConnectionAsync();
Console.WriteLine("客户端已连接           ");

while (true)
{
    byte[] buffer = new byte[1024];
    int bytesRead = await pipeServer.ReadAsync(buffer, 0, buffer.Length);
    if (bytesRead > 0)
    {
        string message = Encoding.UTF8.GetString(buffer, 0, bytesRead);
        Console.WriteLine($"收到消息       : {message}");
    }
}
```

客户端

```
using (var pipeClient = new NamedPipeClientStream(".", "mypipe", PipeDirection.InOut,
PipeOptions.Asynchronous))
{
    await pipeClient.ConnectAsync();
    Console.WriteLine("已连接到管道服务器                ");

    while (true)
    {
        string message = Console.ReadLine();
        byte[] buffer = Encoding.UTF8.GetBytes(message);
        await pipeClient.WriteAsync(buffer, 0, buffer.Length);
        await pipeClient.FlushAsync();
    }
}
```

跨设备命名管道进程通信

```
//服务端
var pipeServer = new NamedPipeServerStream("mypipe", PipeDirection.InOut,
NamedPipeServerStream.MaxAllowedServerInstances, PipeTransmissionMode.Byte, PipeOptions.Asynchronous);
await pipeServer.WaitForConnectionAsync();
Console.WriteLine("客户端已连接          ");

while (true)
{
    byte[] buffer = new byte[1024];
    int bytesRead = await pipeServer.ReadAsync(buffer, 0, buffer.Length);
    if (bytesRead > 0)
    {
        string message = Encoding.UTF8.GetString(buffer, 0, bytesRead);
        Console.WriteLine($"收到消息       : {message}");
    }
}
//客户端
using (var pipeClient = new NamedPipeClientStream("ServerMachine", "mypipe", PipeDirection.InOut,
PipeOptions.Asynchronous))
{
    await pipeClient.ConnectAsync();
    Console.WriteLine("已连接到管道服务器              ");

    while (true)
    {
        string message = Console.ReadLine();
        byte[] buffer = Encoding.UTF8.GetBytes(message);
        await pipeClient.WriteAsync(buffer, 0, buffer.Length);
        await pipeClient.FlushAsync();
    }
}
```

PIPE管道通信库封装

```
// 管道服务器类
public class PipeServer
{
    private readonly string _pipeName;
```

```
    private readonly PipeDirection _direction;
    private readonly int _maxInstances;
    private readonly PipeTransmissionMode _transmissionMode;
    private readonly PipeOptions _options;

    public event EventHandler<string> MessageReceived;

    public PipeServer(string pipeName, PipeDirection direction, int maxInstances, PipeTransmissionMode transmissionMode, PipeOptions options)
    {
        _pipeName = pipeName;
        _direction = direction;
        _maxInstances = maxInstances;
        _transmissionMode = transmissionMode;
        _options = options;
    }

    // 启动管道服务器
    public void Start()
    {
        Task.Run(() => ListenForClients());
    }

    // 监听客户端连接
    private async Task ListenForClients()
    {
        while (true)
        {
            using (var namedPipeServerStream = new NamedPipeServerStream(_pipeName, _direction, _maxInstances, _transmissionMode, _options))
            {
                try
                {
                    await namedPipeServerStream.WaitForConnectionAsync();
                    Console.WriteLine("进程连接管道          ");

                    // 读取客户端发送的进程名称
                    byte[] nameBytes = new byte[1024];
                    int nameBytesRead = await namedPipeServerStream.ReadAsync(nameBytes, 0, nameBytes.Length);
```

```
                string processName = Encoding.UTF8.GetString(nameBytes, 0, nameBytesRead);
                Console.WriteLine($"□□□□□□□: {processName}");

                // □□□□□□□□□□□□
                while (namedPipeServerStream.IsConnected)
                {
                    byte[] bytes = new byte[1024];
                    int bytesRead = await namedPipeServerStream.ReadAsync(bytes, 0, bytes.Length);
                    if (bytesRead > 0)
                    {
                        string msg = Encoding.UTF8.GetString(bytes, 0, bytesRead);
                        Console.WriteLine($"□□: {msg}");
                        OnMessageReceived(msg);
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine($"□□: {ex.Message}");
            }
        }
    }
}

// □□□□□□□□
protected virtual void OnMessageReceived(string message)
{
    MessageReceived?.Invoke(this, message);
}
}
```

```
// □□□□□□□□□
public class PipeServerBuilder
{
    private string _pipeName;
    private PipeDirection _direction;
    private int _maxInstances;
    private PipeTransmissionMode _transmissionMode;
    private PipeOptions _options;
```

```
// 设置管道名称
public PipeServerBuilder WithPipeName(string pipeName)
{
    _pipeName = pipeName;
    return this;
}

// 设置管道方向
public PipeServerBuilder WithDirection(PipeDirection direction)
{
    _direction = direction;
    return this;
}

// 设置最大实例数
public PipeServerBuilder WithMaxInstances(int maxInstances)
{
    _maxInstances = maxInstances;
    return this;
}

// 设置传输模式
public PipeServerBuilder WithTransmissionMode(PipeTransmissionMode transmissionMode)
{
    _transmissionMode = transmissionMode;
    return this;
}

// 设置管道选项
public PipeServerBuilder WithOptions(PipeOptions options)
{
    _options = options;
    return this;
}

// 构建管道服务器
public PipeServer Build()
{
    return new PipeServer(_pipeName, _direction, _maxInstances, _transmissionMode, _options);
}
```

```
    }
```

```
public class PipeClient
{
    private readonly string _pipeName;
    private NamedPipeClientStream _namedPipeClientStream;

    public event EventHandler<string> MessageToSend;

    public PipeClient(string pipeName)
    {
        _pipeName = pipeName;
    }

    public async Task StartAsync()
    {
        string processName = Process.GetCurrentProcess().ProcessName;

        _namedPipeClientStream = new NamedPipeClientStream(".", _pipeName, PipeDirection.InOut,
PipeOptions.Asynchronous);
        try
        {
            await _namedPipeClientStream.ConnectAsync();
            Console.WriteLine("已连接到管道服务器                ");

            // 发送进程名称
            byte[] nameBytes = Encoding.UTF8.GetBytes(processName);
            await _namedPipeClientStream.WriteAsync(nameBytes, 0, nameBytes.Length);
            await _namedPipeClientStream.FlushAsync();

            // 订阅消息发送事件
            MessageToSend += async (sender, message) =>
            {
                byte[] messageBytes = Encoding.UTF8.GetBytes(message);
                await _namedPipeClientStream.WriteAsync(messageBytes, 0, messageBytes.Length);
                await _namedPipeClientStream.FlushAsync();
            };

            Console.WriteLine("客户端已启动，等待发送消息                    ...");
        }
```

```
        catch (Exception ex)
        {
            Console.WriteLine($"错误  : {ex.Message}");
        }
    }

    public void SendMessage(string message)
    {
        MessageToSend?.Invoke(this, message);
    }
}
```

```
// 创建并配置管道服务器
var pipeServer = new PipeServerBuilder()
    .WithPipeName("mypipe")
    .WithDirection(PipeDirection.InOut)
    .WithMaxInstances(NamedPipeServerStream.MaxAllowedServerInstances)
    .WithTransmissionMode(PipeTransmissionMode.Byte)
    .WithOptions(PipeOptions.Asynchronous)
    .Build();
// 启动管道服务器
pipeServer.Start();
```

```
// 创建并启动客户端
var pipeClient = new PipeClient("mypipe");
await pipeClient.StartAsync();

// 发送不同的消息
while (true)
{
    string message = Console.ReadLine();
    pipeClient.SendMessage(message);
}
```

命令

查找端口占用

```
netstat -ano | findstr :7002
```

```
taskkill /F /PID 7292  杀死进程
```

高薪必备 -锁

1、常见 “锁”及使用方式一览表

锁类型	是否支持异步	官方定义	适用场景
Lock	□ □	提供基于对象的互斥访问机制（Monitor 实现）	小范围共享资源保护
Monitor	□ □	提供与 lock 相同功能，底层机制	自定义细粒度控制
Mutex	□ □□□□□	基于操作系统内核对象的互斥锁	跨进程同步、防止程序多开
Semaphore SemaphoreSlim	□ □	控制最多允许 n 个线程同时访问	限流、异步资源池、队列控制
ReaderWriterLockSlim	□ □	允许多个读线程、一个写线程	缓存、配置中心
SpinLock SpinWait	□ □	忙等锁，适合极短等待时间	极端高性能场景
Concurrent Collections	N/A	内部已实现线程安全的集合类	多线程数据结构操作
Channel<T>	□ □	生产者消费者模型通信机制	任务队列、事件流处理

2、各种锁代码案例

Lock

```
private readonly object _sync = new object();

public void UpdateState()
{
    lock (_sync)
```

```
    {
        // 同步操作
    }
}

//高阶实践       -Double-check Locking (双重检查锁定)                ——实现线程安全的单例模式
public sealed class Singleton
{
    private static volatile Singleton _instance;
    private static readonly object _sync = new object();

    private Singleton() { }

    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_sync)
                {
                    if (_instance == null)
                    {
                        _instance = new Singleton();
                    }
                }
            }
            return _instance;
        }
    }
}

//使用    lock 实现缓存更新同步
public class CacheManager
{
    private Dictionary<string, string> _cache = new Dictionary<string, string>();
    private readonly object _sync = new object();

    public void AddOrUpdate(string key, string value)
    {
```

```
        lock (_sync)
        {
            if (_cache.ContainsKey(key))
            {
                _cache[key] = value;
            }
            else
            {
                _cache.Add(key, value);
            }
        }
    }

    public string Get(string key)
    {
        lock (_sync)
        {
            return _cache.TryGetValue(key, out var value) ? value : null;
        }
    }
}
```

Monitor

```
Monitor.Enter(_sync);
try
{
    // 同步代码
}
finally
{
    Monitor.Exit(_sync);
}



//高阶用法
//1、带超时的进入锁（              TryEnter）
private readonly object _sync = new object();
```

```
public bool TryAccess(int timeoutMs)
{
    if (Monitor.TryEnter(_sync, timeoutMs))
    {
        try
        {
            Console.WriteLine("□□□□□□□□□ ...");
            return true;
        }
        finally
        {
            Monitor.Exit(_sync);
        }
    }
    else
    {
        Console.WriteLine("□□□□□□□□□□ ");
        return false;
    }
}
```

□□□ □□□□□□□□□□□

```
// 2□□□□□□ Wait / Pulse

private readonly object _sync = new object();
private bool _dataReady = false;

// □□□□□
public void Consumer()
{
    Monitor.Enter(_sync);
    try
    {
        while (!_dataReady)
        {
            Console.WriteLine("□□□□□□□ ...");
            Monitor.Wait(_sync); // □□□□□□□□
        }
```

```
        Console.WriteLine("数据就绪，开始处理                ");
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}

// 生产者线程
public void Producer()
{
    Monitor.Enter(_sync);
    try
    {
        Console.WriteLine("生产者正在准备数据                ...");
        Thread.Sleep(1000); // 模拟耗时操作
        _dataReady = true;
        Monitor.Pulse(_sync); // 唤醒一个等待线程
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}
```

Wait() 必须放在 while 循环中，防止虚假唤醒。

Pulse() 只唤醒一个线程， PulseAll() 唤醒所有线程。

```
//自定义细粒度锁控制

private readonly object _sync = new object();
private int _value = 0;

public void UpdateValue(int newValue)
{
    Monitor.Enter(_sync);
    try
    {
        _value = newValue;
        Console.WriteLine($"值已更新为       : {_value}");
```

```
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}

public int GetValue()
{
    Monitor.Enter(_sync);
    try
    {
        return _value;
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}
```

Mutex

```
var mutex = new Mutex(false, "MyAppUniqueName");
if (mutex.WaitOne(TimeSpan.FromSeconds(3), false))
{
    try
    {
        // □□□□□□
    }
    finally
    {
        mutex.ReleaseMutex();
    }
}
```

SemaPhore/SemaPhoreSlim

```
private readonly SemaphoreSlim _semaphore = new SemaphoreSlim(3□ 5);


public async Task AccessResourceAsync()
```

```
{
    await _semaphore.WaitAsync();
    try
    {
        // 异步访问资源
    }
    finally
    {
        _semaphore.Release();
    }
}

///异步超时访问（带超时机制）

bool acquired = await _semaphore.WaitAsync(TimeSpan.FromSeconds(2));
if (acquired)
{
    try
    {
        // 成功获得资源
    }
    finally
    {
        _semaphore.Release();
    }
}
else
{
    Console.WriteLine("未能在指定时间内获取资源                    ");
}
```

ReaderWriterLockSlim

```
private readonly ReaderWriterLockSlim _rwLock = new ReaderWriterLockSlim();

public void ReadData()
{
    _rwLock.EnterReadLock();
    try
    {
```

```
        // 读取数据
    }
    finally
    {
        _rwLock.ExitReadLock();
    }
}

public void WriteData()
{
    _rwLock.EnterWriteLock();
    try
    {
        // 写入数据
    }
    finally
    {
        _rwLock.ExitWriteLock();
    }
}
```

SpinLock/SpinWait

```
SpinLock spinLock = new SpinLock();
bool lockTaken = false;
try
{
    spinLock.Enter(ref lockTaken);
    // 操作共享资源
}
finally
{
    if (lockTaken) spinLock.Exit();
}
```

Concurrent Collections

```
ConcurrentDictionary<string, string> cache = new ConcurrentDictionary<string, string>();
cache.TryAdd("key", "value");
```

```
ConcurrentQueue<int> queue = new ConcurrentQueue<int>();
queue.Enqueue(1);
```

Channel<T>

```
var channel = Channel.CreateUnbounded<int>();


// 生产者
await channel.Writer.WriteAsync(1);


// 消费者
await foreach (var item in channel.Reader.ReadAllAsync())
{
    Console.WriteLine(item);
}
```

内存屏障

核心作用：防止重排序 + 保证内存可见性

□□ □□□□□□□□□□□□□□□

现代 CPU 和编译器为了提升性能，会做两件事：

指令重排序（Instruction Reordering）：调整代码执行顺序（只要单线程结果不变）；

缓存优化：每个 CPU 核心有自己的缓存，可能不立即同步到主内存。

但在多线程环境下，这些优化会导致一个线程看不到另一个线程的最新修改！

□ □□□□□□□□

内存屏障是一种 CPU 指令，用来强制规定：屏障前后的读写操作不能跨过屏障重排序，并且确保内存操作对其他线程可见。

你可以把它想象成一道“闸门”：

闸门之前的操作必须全部完成；

之后的操作不能提前到闸门之前；

同时强制刷新缓存，让其他线程看到最新数据。

□□ 经典例子：双重检查锁定中的问题

```
_instance = new Singleton();
// 表面是一行，实际分三步：
// 1. 分配内存
// 2. 调用构造函数（初始化）
// 3. 将地址赋给 _instance
```

没有内存屏障时，CPU 可能把步骤 3 提前到步骤 2 之前（重排序）：

线程 A：先赋值 _instance = 地址（但对象还没初始化！）

线程 B：看到 _instance != null，直接使用 → 访问未初始化的对象！崩溃！

volatile 如何引入内存屏障?

在 C# 中，对 volatile 字段的读写会自动插入内存屏障：

写操作（Write）：插入释放屏障（Release Fence）→ 保证之前的写操作都完成后再写这个字段；

读操作（Read）：插入获取屏障（Acquire Fence）→ 保证之后的读操作不会提前，且能看到最新的值。

所以：

private static volatile Singleton _instance;

这行代码确保：

_instance 的赋值不会被重排序到构造函数之前；

其他线程读 _instance 时，能看到完全初始化的对象。

线程安全 VS 内存屏障：关系总结

概念	作用	层级
线程安全	保证多线程下程序行为正确	高层目标
内存屏障	防止重排序 + 保证内存可见性	底层机制之一

内存屏障是实现线程安全的底层手段之一；

但线程安全还可以通过锁（`lock`）、原子操作等其他方式实现；

`lock` 内部其实也使用了内存屏障（进入锁 = 获取屏障，退出锁 = 释放屏障）。

高薪必备 -线程

1、线程分类

类型	描述
主线程（Main Thread）	应用程序启动时的第一个线程
后台线程（Background Thread）	不阻止程序退出
前台线程（Foreground Thread）	阻止程序退出，直到完成
池化线程（Thread Pool Threads）	由 .NET 线程池管理
显式线程（new Thread()）	手动创建和控制的线程

2、线程的生命周期状态

状态	含义
Unstarted	已创建但尚未启动
Running	正在运行
WaitSleepJoin	调用了 `Wait`, `Sleep`, `Join`
Suspended	已挂起（不推荐使用）
Aborted	线程已请求终止
Stopped	线程已结束

3、线程的常用属性和方法

`IsBackground`	是否为后台线程
`Priority`	设置线程优先级（High/AboveNormal/Normal 等）
`Start()` / `Join()` / `Abort()` / `Sleep()`	控制线程行为
`CurrentThread`	获取当前线程对象
`Name`	给线程命名，方便调试

4、Thread代码案例

显式创建并控制线程

```
Thread thread = new Thread(() =>
{
    Console.WriteLine("线程开始工作          ");
    Thread.Sleep(2000);
    Console.WriteLine("线程结束       ");
});
thread.IsBackground = false; // 设置为前台线程
thread.Start();
thread.Join(); // 主线程等待子线程结束
Console.WriteLine("主线程继续         ");
```

注意: Join() 会阻塞主线程，直到该线程完成。

使用参数传递数据给线程

```
Thread thread = new Thread(obj =>
{
    string message = obj as string;
    Console.WriteLine($"收到消息       : {message}");
});
thread.Start("Hello from thread");
```

线程优先级设置（谨慎使用）

```
Thread thread = new Thread(() =>
{
    Console.WriteLine($"线程优先级         : {Thread.CurrentThread.Priority}");
});
thread.Priority = ThreadPriority.Highest;
thread.Start();
```

高优先级线程可能影响系统性能，慎用。

线程异常处理（非常重要）

```
AppDomain.CurrentDomain.UnhandledException += (sender, args) =>
{
```

```
    Console.WriteLine("未处理异常:          " + (args.ExceptionObject as Exception)?.Message);
};

Thread thread = new Thread(() =>
{
    throw new InvalidOperationException("测试异常       ");
});
thread.Start();

// 线程中抛出的异常不会自动传播到主线程!
```

线程同步与锁结合使用（ lock + Thread）

```
private static readonly object _sync = new object();
private static int _counter = 0;

Thread t1 = new Thread(() =>
{
    for (int i = 0; i < 1000; i++)
    {
        lock (_sync)
        {
            _counter++;
        }
    }
});

Thread t2 = new Thread(() =>
{
    for (int i = 0; i < 1000; i++)
    {
        lock (_sync)
        {
            _counter--;
        }
    }
});

t1.Start();
t2.Start();
```

```
t1.Join();
t2.Join();

Console.WriteLine(_counter); // 应该输出 0
```

使用 ParameterizedThreadStart 传参（带对象）

```
class Person
{
    public string Name { get; set; }
}

Thread thread = new Thread((obj) =>
{
    var person = obj as Person;
    Console.WriteLine($"你好，{person.Name}");
});
thread.Start(new Person { Name = "张三" });
```

使用 `Thread.Join(TimeSpan)` 实现超时等待

```
Thread thread = new Thread(() =>
{
    Thread.Sleep(3000);
    Console.WriteLine("线程完成");
});

thread.Start();
bool finished = thread.Join(TimeSpan.FromSeconds(2));

if (!finished)
{
    Console.WriteLine("线程仍在运行，超时");
}
```

使用 `AutoResetEvent` 或 `ManualResetEvent` 实现线程通信

```
AutoResetEvent waitHandle = new AutoResetEvent(false);
```

```
Thread worker = new Thread(() =>
{
    Console.WriteLine("等待信号       ...");
    waitHandle.WaitOne();
    Console.WriteLine("收到信号，开始工作                ");
});

worker.Start();

Thread.Sleep(2000);
waitHandle.Set(); // 发送信号



var signal = new ManualResetEvent(false);

Thread worker = new Thread(() =>
{
    Console.WriteLine("工作线程开始          ");
    Thread.Sleep(2000);
    Console.WriteLine("工作线程完成          ");
    signal.Set(); // 发送信号
});

worker.Start();
Console.WriteLine("主线程等待        ...");
signal.WaitOne(); // 等待信号
Console.WriteLine("主线程继续        ");
```

使用 `Thread.Abort()` 和 `Thread.Interrupt()` (不推荐)

```
Thread thread = new Thread(() =>
{
    try
    {
        Console.WriteLine("线程开始       ");
        Thread.Sleep(Timeout.Infinite); // 模拟长时间任务
    }
    catch (ThreadInterruptedException)
    {
        Console.WriteLine("线程被中断        ");
```

```
    }
    catch (ThreadAbortException)
    {
        Console.WriteLine("线程被终止          ");
        Thread.ResetAbort(); // 可选: 取消终止
    }
});

thread.Start();
thread.Interrupt(); // 中断     Sleep/Wait 状态
// thread.Abort(); // 强制终止 (不推荐)
```

线程本地存储（   TLS） ——ThreadStatic / AsyncLocal

```
//使用     [ThreadStatic] 存储线程私有变量
[ThreadStatic]
private static int _threadId;

Thread t1 = new Thread(() =>
{
    _threadId = 1;
    Console.WriteLine($"线程     1 ID: {_threadId}");
});

Thread t2 = new Thread(() =>
{
    _threadId = 2;
    Console.WriteLine($"线程     2 ID: {_threadId}");
});

t1.Start();
t2.Start();



//使用     AsyncLocal<T> 支持异步上下文隔离 (推荐用于                    async/await)
AsyncLocal<int> asyncLocal = new AsyncLocal<int>();

asyncLocal.Value = 1;
Console.WriteLine($"主线程值       : {asyncLocal.Value}");
```

```
Task.Run(() =>
{
    asyncLocal.Value = 2;
    Console.WriteLine($"任务线程值        : {asyncLocal.Value}");
}).Wait();
```

5、线程池（ `ThreadPool` ）

提交工作项

```
ThreadPool.QueueUserWorkItem(state =>
{
    Console.WriteLine("来自线程池的工作              ");
});
```

获取当前线程池状态

```
int workerThreads, completionPortThreads;
ThreadPool.GetAvailableThreads(out workerThreads, out completionPortThreads);
Console.WriteLine($"可用线程池线程数:              {workerThreads}");
```

自定义线程池大小（谨慎使用）

```
ThreadPool.SetMinThreads(10, 10); // 设置最小线程数
ThreadPool.SetMaxThreads(100, 100); // 设置最大线程数
```

6、Task（基于线程池的高级封装）

使用 `CancellationToken` 控制取消

```
var cts = new CancellationTokenSource();
cts.CancelAfter(TimeSpan.FromSeconds(2));

try
{
    await Task.Run(async () =>
    {
        while (!cts.Token.IsCancellationRequested)
        {
            await Task.Delay(500);
```

```
            Console.WriteLine("正在运行       ");
        }
    }, cts.Token);
}
catch (OperationCanceledException)
{
    Console.WriteLine("任务被取消         ");
}
```

使用 `ContinueWith()` 实现任务链

```
Task<int> task1 = Task.Run(() => 100);
task1.ContinueWith(prev => Console.WriteLine($"结果  : {prev.Result}"));
```

使用 `ValueTask` 减少分配（适用于高性能场景）

```
public ValueTask<int> ComputeAsync(int a, int b, CancellationToken ct = default)
{
    if (ct.IsCancellationRequested)
        return new ValueTask<int>(Task.FromCanceled<int>(ct));

    return new ValueTask<int>(a + b);
}
```

使用 `ConfigureAwait(false)` 避免上下文捕获

```
await SomeAsyncMethod().ConfigureAwait(false);
```

在库方法中推荐使用，避免在 UI 线程中死锁。

使用 `TaskCompletionSource<T>` 封装异步逻辑

```
public Task<int> DelayedResultAsync()
{
    var tcs = new TaskCompletionSource<int>();
    new Thread(() =>
    {
        Thread.Sleep(1000);
        tcs.SetResult(42);
```

```
    }).Start();
    return tcs.Task;
}
```

高薪必备 -大纲

推荐学习路径（适合准备架构师或技术负责人岗位）

阶段	学习内容
第一阶段	锁机制、线程池、TPL、Channel、PLINQ、ValueTask
第二阶段	泛型、反射、IL、Expression、AOP、DI、IoC
第三阶段	ASP.NET Core、中间件、Filter、DI、管道
第四阶段	EF Core、数据库迁移、性能优化
第五阶段	分布式系统、CAP、Saga、事件溯源
第六阶段	设计模式、DDD、CQRS、EventBus、微服务治理
第七阶段	自定义组件、源生成、AOT 编译、性能调优
第八阶段	架构设计、系统拆分、限流、熔断、日志聚合

一、性能优化与底层原理（顶级工程师必备）

- `ValueTask` vs `Task` 的区别？什么时候使用？
- `Span<T>` / `Memory<T>` 的作用及适用场景
- `ref struct`、`stackalloc`、`fixed` 的使用和限制
- 如何避免内存泄漏（如事件订阅、闭包引用等）
- `unsafe` 编程与指针操作
- `GC.Collect()` 和 `GC.WaitForPendingFinalizers()` 的用途
- `WeakReference` 的使用与应用场景
- 对象池（Object Pool）设计与实现
- 内存屏障（MemoryBarrier）与 Volatile 读写

二、.NET 运行时机制与 CLR 原理

- .NET Core 与 .NET Framework 的区别
- GC 工作原理（代龄、压缩、标记清除算法）
- JIT 编译器工作流程
- AOT 编译 vs JIT 编译的区别
- AppDomain 的作用（虽然现在少用）
- Assembly 加载机制（Load, LoadFrom, LoadFile）
- 类型加载过程（Type.Load, Type.GetType）

- 反射（Reflection）的性能问题及优化方式
- IL 中间语言基础（如 `ldloc`, `stloc`, `callvirt`）
- C# 编译器如何将代码转换为 IL

三、异步编程模型（不仅仅是 async/await）

- `async/await` 的底层原理（状态机、IAsyncStateMachine）
- `ConfigureAwait(false)` 的作用和为什么推荐在库中使用
- `SynchronizationContext` 和 `TaskScheduler` 的区别
- `ValueTaskSource` 的封装与自定义 ValueTask
- 如何实现一个“无 await”的异步方法（如基于 IAsyncResult）
- 异步本地变量：`AsyncLocal<T>` 的原理与使用陷阱
- 如何避免死锁（如 `task.Result` 在 UI 上下文中的死锁）
- `ValueTask` 是否一定比 `Task` 快？什么情况下不推荐？

四、泛型与反射进阶（高级开发必会）

- 泛型协变（in/out）的作用和限制
- 泛型约束（where T : class, new(), interface, struct 等）
- 泛型类型擦除机制（CLR 是不是真正支持泛型？）
- 泛型接口与抽象类的设计模式
- `MethodInfo.MakeGenericMethod()` 的使用
- 动态调用方法（反射性能优化技巧）
- 使用 `Expression Tree` 替代反射提高性能
- `System.Runtime.CompilerServices.Unsafe` 的用途与风险
- `MethodImpl(MethodImplOptions.AggressiveInlining)` 的作用
- `ConditionalWeakTable<TKey, TValue>` 的用途（用于扩展对象元数据）

五、集合与数据结构优化

- `ConcurrentDictionary<TKey, TValue>` 的实现原理
- `ConcurrentQueue<T>` 与 `Channel<T>` 的区别
- `SortedSet<T>`、`SortedList<TKey, TValue>`、`SortedDictionary<TKey, TValue>` 的区别
- `ArrayPool<T>` / `BufferPool` 的使用与性能优势
- `ReadOnlySpan<T>` 和 `ReadOnlyMemory<T>` 的使用场景
- 自定义高性能队列（如环形缓冲区 RingBuffer）
- 实现一个无锁队列（Lock-Free Queue）
- 数据分页与懒加载（Paging + Lazy<T>）

六、模块化架构与设计模式（大型系统必问）

- 领域驱动设计（DDD）核心概念（聚合根、值对象、仓储）

- CQRS 模式与 Event Sourcing 的区别
- Mediator 模式在项目中的实际应用
- 工厂模式 vs DI 容器
- 策略模式与命令模式的结合使用
- 单例模式的多线程安全实现（Double-check Locking）
- 装饰器模式与 AOP 实践（如日志、缓存装饰）
- 观察者模式 vs 发布-订阅模式 vs Prism 的 IEventAggregator
- 构建可插拔系统（Prism、MEF、Autofac Modules）

七、依赖注入与 IOC 容器（现代框架的核心）

- ASP.NET Core 默认容器（Microsoft.Extensions.DependencyInjection）的生命周期（Transient、Scoped、Singleton）
- 如何实现一个简易 IOC 容器（构造函数注入、属性注入）
- 服务定位器（Service Locator）模式的弊端
- 什么是 Ambient Context 模式?
- 依赖注入中循环依赖的识别与处理
- DI 与 AOP 的结合（如 Autofac + Interceptor）
- 使用 `IServiceProvider` 与 `IServiceScopeFactory` 的区别
- 在静态类中如何获取注入的服务（不推荐，但有办法）

八、跨平台与底层交互（适用于 .NET Core）

- P/Invoke 的使用与注意事项（DllImport）
- 如何调用原生 C/C++ 接口
- COM Interop（Windows 平台）
- 使用 `NativeAOT` 编译 .NET 应用
- 如何编写跨平台的 C# 代码
- `RuntimeInformation.IsOSPlatform(OSPlatform.Windows)` 的用途
- `System.Runtime.InteropServices` 的常用类
- 如何判断当前运行环境（x86/x64/ARM）

九、网络编程与通信（高并发后端开发必备）

- TCP/UDP Socket 编程与异步模型（SocketAsyncEventArgs）
- `HttpClient` 的正确使用方式（单例复用、取消令牌）
- `HttpMessageHandler` 的重用与连接池管理
- gRPC 的基本原理与 .NET 集成
- SignalR 的底层通信机制（WebSocket vs Server-Sent Events）
- 使用 `Kestrel` 实现高性能 Web 服务器
- 如何实现一个简单的 RPC 框架
- 使用 `MemoryPool<byte>` 处理大块数据传输

十、性能监控与诊断工具（排查生产问题）

- 使用 PerfView 分析线程饥饿、GC 性能瓶颈
- 使用 dotTrace / dotMemory 进行 CPU 和内存分析
- 使用 VisualVM / ANTS Memory Profiler（Java 互操作时）
- `dotnet-counters`、`dotnet-trace` 的使用
- 使用 `DiagnosticSource` 实现轻量级日志追踪
- 如何查看当前线程池状态（`ThreadPool.GetAvailableThreads`）
- 如何检测死锁（线程阻塞、同步上下文丢失）
- 使用 `MiniDump` 文件进行崩溃分析
- 使用 ETW（Event Tracing for Windows）做性能追踪

十一、单元测试与自动化测试（工程能力体现）

- 单元测试与集成测试的区别
- Mock 框架（Moq、NSubstitute）的使用
- 如何测试异步代码（包括异常抛出）
- 使用 `xUnit` / `NUnit` / `MSTest` 的差异
- 测试静态类、私有方法、密封类的技巧
- 测试依赖外部资源（数据库、API）的方法
- 使用 `TestServer` 做集成测试（ASP.NET Core）
- 如何衡量测试覆盖率（coverlet、Visual Studio、JetBrains Rider）

十二、中间件与管道模型（ASP.NET Core 核心机制）

- ASP.NET Core 的请求管道模型（Middleware Pipeline）
- 如何自定义 Middleware
- `Use`, `Run`, `Map` 的区别
- `IApplicationBuilder`、`IHostApplicationLifetime` 的使用
- 请求委托（RequestDelegate）的链式构建
- 如何实现身份验证中间件
- 使用 `PipelineBuilder` 手动构建中间件管道
- 如何在中间件中访问 DI 容器
- 使用 `HttpContext.Features` 获取底层特性

十三、配置系统与依赖管理（微服务时代重点）

- `IConfiguration` 的来源（appsettings.json、Environment Variables、Command Line）
- `IOptions<T>` 的绑定与使用（包括 Named Options）
- `IConfigureOptions<T>` 和 `IPostConfigureOptions<T>` 的区别
- 使用 `AddOptions().PostConfigure(...)` 的典型场景
- 自定义 ConfigurationProvider 的实现
- 使用 `Secret Manager` 存储敏感信息
- 热更新配置（`IOptionsMonitor<T>` vs `IOptionsSnapshot<T>`）

十四、序列化与反序列化（JSON、XML、Binary）

- `System.Text.Json` vs `Newtonsoft.Json` 的区别
- `JsonSerializer.Deserialize<T>(...)` 的性能优化技巧
- 如何自定义 JSON 转换器（JsonConverter）
- `DataContractSerializer` vs `XmlSerializer` 的选择
- `BinaryFormatter` 的安全隐患（已废弃）
- 使用 `MemoryPack` / `MessagePack` 提升序列化性能
- 使用 `System.Text.Json` 支持源生成（Source Generation）
- 使用 `Utf8JsonReader` / `JsonDocument` 做只读解析

十五、LINQ 与表达式树（底层机制）

- `IEnumerable<T>` vs `IQueryable<T>` 的区别
- LINQ 查询的延迟执行机制
- `Select` / `Where` / `GroupBy` 的底层行为
- 表达式树（Expression Tree）与动态查询构建
- 如何编译 Expression 为 Delegate
- 使用 `ExpressionVisitor` 修改表达式逻辑
- EF Core 中的表达式解析与 SQL 转换
- 使用 `AsParallel()` 实现 PLINQ
- 使用 `Partitioner` 自定义并行策略

十六、跨平台部署与发布（云原生工程师必备）

- `dotnet publish` 的参数详解（`--self-contained`, `--runtime`）
- 如何发布 AOT 编译的 .NET 应用
- 使用 `Single File` 打包的优势与限制
- `Trimming` 与 `PublishTrimmed` 的作用
- 如何减小 Docker 镜像体积
- `ReadyToRun` 编译与性能提升
- 使用 `deps.json` 控制程序集加载
- 使用 `AssemblyLoadContext` 实现热更新
- 如何在 Linux 上调试 .NET Core 程序

十七、分布式系统与微服务架构（架构师级别）

- CAP 理论与 BASE 理能力
- 微服务通信方式（REST, gRPC, Message Queue）
- 服务发现（Consul, etcd, Kubernetes）
- 分布式事务（2PC, Saga, TCC）
- 最终一致性与幂等性设计
- 使用 `Dapper` / `Entity Framework Core` 实现 Repository 模式
- 事件溯源（Event Sourcing）与快照（Snapshotting）
- 使用 `CAP` 实现最终一致性的消息队列消费
- 使用 `MassTransit` / `Rebus` / `Cap` 实现分布式任务调度
- 使用 `Serilog` 实现结构化日志记录

十八、EF Core 与数据库优化（ORM 深入）

- EF Core 的查询生命周期（Query Pipeline）
- 如何启用 `DetailedErrors` 和 `SensitiveDataLogging`
- `ChangeTracker` 的作用与关闭时机
- 显式加载（`Include`，`ThenInclude`，`Load`）
- 使用 `AsNoTracking()` 提升查询性能
- 自定义 ValueConverter 与 ValueComparer
- 使用 `FromSqlRaw` 实现复杂查询
- 使用 `Interceptors` 监控 SQL 日志
- 使用 `RelationalTypeMapping` 自定义字段映射

十九、C# 语言新特性掌握情况（C# 9/10/11/12）

- `record` 与 `class` 的区别
- `init` 属性设置器的作用
- `required` 成员（C# 11）
- `global using`、`file-scoped namespace`（C# 10）
- `with` 表达式（record 特有）
- `nint` / `Half` / `Primary Constructor`（C# 12）
- `Lambda Return Type`、`params any type`（C# 12）
- `raw string literal`（"""）的使用场景
- 使用 `source generator` 实现零运行时开销代码生成
- 使用 `partial methods` 实现编译期钩子（hook）

二十、设计能力与系统架构能力（架构师级别）

- 如何设计一个高并发系统（如订单中心）
- 如何设计一个通用权限模型（RBAC vs ABAC）
- 如何设计一个可插拔的日志系统
- 如何设计一个统一异常处理中间件
- 如何设计一个可扩展的 API 网关
- 如何设计一个高性能缓存服务（本地缓存 vs Redis）
- 如何设计一个限流服务（滑动窗口 vs 令牌桶）
- 如何设计一个分布式 ID 生成器（Snowflake vs IdGenerator）
- 如何设计一个通用后台任务队列（Channel<T> vs BackgroundService）
- 如何设计一个插件系统（AppDomain / AssemblyLoadContext）

二十一、开源生态与工具链（加分项）

- 使用 `MediatR` 实现 CQRS 模式
- 使用 `AutoMapper` 实现 DTO 到 Entity 的转换
- 使用 `FluentValidation` 实现强类型校验
- 使用 `Serilog` 实现结构化日志
- 使用 `Polly` 实现熔断、重试、降级

- 使用 `Refit` 实现声明式 REST 客户端
- 使用 `MassTransit` 或 `CAP` 实现消息队列消费
- 使用 `ImageSharp` / `SkiaSharp` 实现图像处理
- 使用 `ImageSharp.Web` 实现图片 CDN
- 使用 `IdentityServer` 实现 OAuth2 认证授权

二十二、常见开源项目理解（加分项）

开源项目	推荐掌握
Orleans	分布式 Actor 模型
Akka.NET	类似 Scala 的 Actor 模型
CAP	分布式事件总线 + 消息持久化
Entity Framework Core	ORM 深度定制
Serilog	结构化日志系统
Polly	弹性策略（重试、熔断）
MediatR	CQRS 模式
FluentValidation	声明式校验
Xunit / Moq / FluentAssertions	单元测试体系
SignalR	实时通信框架

互联网项目必备投入

作者自述

> 我是一名很普通的程序员，前端用 Vue，后端用 ASP.NET Core，PC 端开发 WinForm/WPF，移动端使用 MAUI，运维勉强算中级水平。
> 目前线上运行的项目叫做 “灵燕空间”，专注于轻量化的 HTTPS 证书管理、客户端软件升级托管、教程在线服务等模块，涵盖网页端、PC 客户端和移动端 App。

前言：别急着花钱，先理清思路！

> 如果你正准备启动一个互联网项目，请记住一句话：“前期投入要精打细算，必要支出必须砸钱，但顺序不能乱，不然分分钟亏到怀疑人生。”
>
> 作为项目的发起人，你要做好亲自扛起所有苦力工作的准备。
> 但只要按照正确的流程来走，即使中途想撤资，也能把损失降到最低。

第一步：服务器？先别买！

> 铁律 1：先不要购买服务器！
>
> - 项目初期开发和测试完全可以在本地局域网完成。
> - 一旦买了服务器，它就开始倒计时到期了。
> - 最合适的云服务器配置，只有在测试完成后才能确定。
> - 别提前超标采购，预算有限，后面还有大把地方等着你投钱呢。

> 建议做法：

- 本地开发 → 内网穿透测试 → 上线后再选云服务器
- 使用 Docker 模拟生产环境更省事

第二步：域名？也别急着注册！

> 铁律 2：前期不要买域名！
>
> - 在项目上线前，随时可能改名。
> - 域名是按年续费的，买了就不能退。
> - 如果项目变更，域名就只能被废弃。

> 建议做法：
>
> - 上线前 1个月再注册主域名（ .com/.cn/.xyz均可）
> - 选择主流平台（阿里云、腾讯云）注册即可

第三步：短信服务怎么搞？

> 个人开发者专属路径：
>
> - 运营商对个人限制严格，唯一可行方式：
> - 开发一款 App
> - 通过 App 的 ICP 备案
> - 上架任意应用商店
> - 向阿里云 /腾讯云申请短信签名

> 企业开发者路径：
>
> - 可以网站备案 or App 备案
> - 审核通过后即可开通短信服务

第四步：网站部署上线那些事儿

> 常识小课堂：
>
> **80** 端口（ **HTTP**） 和 **443** 端口（ **HTTPS**）是浏览器默认访问的两个端口。
>
> - 所以你的网址可以直接写成：
>   - `https://www.lingyanspace.com`（不用加 :443）
>   - `http://www.lingyanspace.com`（不用加 :80）
> - 如果你用了其他端口（如 7000），就必须写成：
>   - `https://www.lingyanspace.com:7000`
>
> HTTPS 是什么鬼？
>
> - □□□□□□□□□□□□□□□□□□
> - 技术上就是 SSL/TLS 加密通信，常见算法有 RSA256、ES256、ES384
> - 简单理解就是：
>   - 浏览器用公钥加密数据
>   - 服务器用私钥解密数据
>
> 推荐工具：我的站点 灵燕空间_____-HTTPS证书图形化控制台_____提供 HTTPS 证书签发、监控、部署的一站式图形化控制台，欢迎体验~

> **ICP** 备案 **&** 公安备案（中国大陆用户必看）
>
> **ICP**备案：工信部管的网站合法经营资格， 先做 **ICP** 备案 （免费）
>
> - 通过阿里云 / 腾讯云提交备
> - 通过阿里云 / 腾讯云提交备
> - 他们会帮你提交给工信部审核
> - 一般半个月出结果，听客服建议能提高成功率
>
> 公安备案：网安部门监管你的网站安全合规性 ， ICP 通过后，再做 公安备案
>
> - 登录各地公安备案系统（如北京公安网安备案系统）
> - 提交网站或 App 的主体信息
> - 审核通过后才算真正 “合法上线 ”
>
> 小贴士：

- 小公司暂时不需要办理 ICP 经营许可证（注册资本 100万 +、社保 3 人以上才需要）
- 一旦公司流水过百万，建议尽快办理 ICP 资质认定

项目开发完成后如何部署上线

1、部署前准备清单

□ □□□□□□□	包括前端 Vue、后端 ASP.NET Core、PC 客户端、移动端 App
□ □□□□□□□	Vue 构建为 dist 文件夹 tar包 ASP.NET Core 发布为 publish 文件夹 tar包
□ □□□□□□□	静态博客等等（ 2核 2G5M带宽起步） 牵扯到数据库最好加码到 4核 8G10M带宽起步
□ □□□□	如 lingyanspace.com（用于绑定访问）
□ □□ ICP 备案	若用户在中国大陆，必须备案
□ □□□□	Alibaba Cloud Linux、TencentOS、Rocky Linux、CentOS 、Windows Server

2、云厂商规则组开放端口

80、443、22（ SSH）、以及你项目实际使用的其他端口

3、安装 **dotnet**运行环境

去微软的官网: 下载 .NET 8.0 (Linux、macOS 和 Windows) | .NET

4、安装 **Nginx**（反向代理）并启动

sudo yum install -y nginx

systemd start nginx

5、上传并部署项目文件

前端 Vue 打包上传

- 将其上传到服务器目录，并且解压，例如 /var/www/html/lingyanspace-web

后端 ASP.NET Core 打包上传

- 上传至服务器路径，并且解压，例如 /var/www/html/lingyanspace-api

配置 systemd 启动后端服务（以 ASP.NET Core 为例）

- 创建服务文件: sudo nano /etc/systemd/system/lingyanspace.service
- 写入以下内容:

```
[Unit]
Description=灵燕空间 API 服务
[Service]
WorkingDirectory=/var/www/applications/lingyanspace-api
ExecStart=/usr/bin/dotnet /var/www/applications/lingyanspace-api/YourApp.dll
Restart=always
RestartSec=10
SyslogIdentifier=lingyanspace
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
[Install]
WantedBy=multi-user.target
```

- 启用并启动服务: sudo systemctl enable lingyanspace.service && sudo systemctl start lingyanspace.service
- 查看日志: journalctl -u lingyanspace.service -f

6、配置 nginx站点

- 配置站点:

```
server {
    listen 80;
    server_name www.lingyanspace.com lingyanspace.com;
    location / {
        root /var/www/html/lingyanspace-web;
        index index.html;
        try_files $uri $uri/ =404;
    }
    location /api/ {
        proxy_pass http://localhost:5000/;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

- 保存后测试并重新加载配置: sudo nginx -t && sudo nginx -s reload

7、开放防火墙 **&** 测试访问

第五步：移动端 App 上架攻略

> 国内安卓应用商店

项目	条件
ICP备案	必须有（App ICP备案）
软著 or 电子版权认证	二选一

- 电子认证：600元，10个工作日，适合快速上架推广
- 软著（计算机软件著作权）：个人可免费申请，约2个月下证；也可代办（约1000元）

注意事项：

- 不要找我代办，我不是中介
- 软著可用于政府项目申报、招投标、高校加分等多种用途

> 苹果 **App Store（iOS）**

类型	年费	特点
个人开发者	$99/年	支持一人团队，可发布App
企业开发者	$299/年	用于内部分发，不能上架App Store

温馨提示：

- 如果你的App不更新了，不续费也没关系，不会下架
- 但如果你想持续迭代，每年都要续费（美金哦！）

> **Google Play Store Android**
>
> | | | |
> |---|---|---|
> | Google Play | $25 | |
>
> - Google +
> -

ICP

> “ICP
>
> **ICP ≠ App ICP**
>
> **App ICP** “ ”

FAQ

> “ ”

lingyanspace.com
HTTPS

实践经验：互联网项目起步指南

赠言

> “前期不乱花钱，后期不怕烧钱。”——送给每一位想做产品的开发者

自我介绍

大家好，我是一名普普通通的程序员。前端用 Vue，后端玩 ASP.NET Core，PC 端折腾 WinForm/WPF，移动端最近在用 MAUI，运维水平也算“半桶水”。目前我在运营一个小项目——灵燕空间，主打 HTTPS 证书管理、客户端软件升级托管、在线教程等服务，覆盖网页、PC 客户端和移动 App。

前言：别急着花钱，先理清思路！

如果你正打算开启自己的互联网项目，送你一句肺腑之言：

> 前期投入要精打细算，必要支出必须砸钱，但顺序不能乱，不然分分钟亏到怀疑人生。

第一步：服务器？先别急！

铁律一：服务器别急着买！

项目初期开发和测试，完全可以在本地局域网搞定。服务器一买，倒计时就开始了，钱花得太早，压力也大。

建议流程：

1. 本地开发
2. 内网穿透测试
3. 上线时再选云服务器

推荐云厂商：阿里云、腾讯云、华为云等主流云厂商，活动多，性价比高。

>

> 小贴士：现在很多云厂商有“学生机”、“轻量应用服务器”等优惠，等项目成型再薅羊毛也不迟！

第二步：域名？也别着急！

铁律二：域名注册要慎重！

项目上线前，名字随时可能变。域名一买，钱就花出去了，还不能退。

建议：

- 上线前 1 个月再注册主域名（.com/.cn/.xyz 都行）
- 选阿里云、腾讯云等主流平台，靠谱
- 域名建议简短易记，避免生僻字母和数字

> 小贴士：域名注册后记得及时实名认证，否则无法解析！

第三步：短信服务怎么搞？

个人开发者专属路径

1. 先开发一款 App
2. 通过 App 做 ICP 备案
3. 上架任意应用商店
4. 再去阿里云/腾讯云申请短信签名

企业开发者路径

- 网站备案或 App 备案都可以
- 审核通过后就能开短信服务

> 小贴士：
> 短信服务审核严格，签名和模板内容要规范，涉及金融、医疗等行业需额外资质。

第四步：网站部署上线那些事儿

常识小课堂

- **80** 端口（ **HTTP**）
- **443** 端口（ **HTTPS**）

你的网站地址可以直接写成：

- `https://www.lingyanspace.com`（不用加 :443）
- `http://www.lingyanspace.com`（不用加 :80）

HTTPS 是什么？

- □□□□□□□□□□□□□□□□
- 技术上是 SSL/TLS 加密通信，常见算法有 RSA256、ES256、ES384
- 免费证书推荐： Let’s Encrypt、阿里云免费 DV 证书

> 小贴士：
> HTTPS 已成标配，微信小程序、支付宝小程序等都要求 HTTPS 接口。

ICP 备案 & 公安备案（中国大陆开发者必看）

- **ICP**备案： 工信部管，网站合法经营资格
- 公安备案： 网安部门管，网站安全合规性

> ⚠ 温馨提示：
>
> - 小公司暂时不需要办理 ICP 经营许可证（注册资本 100 万 +、社保 3 人以上才需要）。
> - 公司流水过百万，建议尽快办理 ICP 资质认定。
> - 备案期间网站需暂停访问，提前做好公告。

第五步：移动端 App 上架攻略

国内安卓应用商店

- 必须有 ICP 备案（ App ICP 备案）
- 软著 or 电子版权认证二选一
  - 电子认证： 600 元， 10 个工作日，适合快速上架
  - 软著：个人可免费申请，约 2 个月下证；也可代办（约 1000 元）

苹果 App Store（ iOS）

- 个人开发者： $99/年
- 企业开发者： $299/年
- 审核严格，建议提前准备好隐私政策、用户协议等材料

Google Play Store（ Android）

- 费用：一次性 $25
- 终身有效，支持全球发布
- 需科学上网，注意合规政策

> 小贴士：
> App 上架前务必自测隐私合规，避免因收集敏感信息被下架。

最后几个温馨提醒

- 软著认证国家免费，只是邮寄证书要快递费
- 网站 ICP 备案 ≠ App ICP 备案，两者要分别提交
- App ICP 备案类型建议选 “工具类型 ”，更容易通过
- 备案信息变更要及时更新，避免被注销

总结

> “[illegible]”

[illegible]

[illegible]

- [illegible]
- [illegible]
- [illegible]

[illegible]

[illegible] [illegible] lingyanspace.com

[illegible]

[illegible]

winfrom与 wpf

项目转换 sdk形式提示 “System.Reflection.AssemblyCompanyAttribute”特性重复

解决方案:

```
<PropertyGroup>
  <TargetFramework>net48</TargetFramework>
  <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
  <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
  <OutputType>WinExe</OutputType>
  <UseWindowsForms>true</UseWindowsForms>
  <ImportWindowsDesktopTargets>true</ImportWindowsDesktopTargets>
  //重点标记，默认会自动创建 AssemblyInfo 特性，编译不通过的原因是存在 AssemblyInfo 文件
  <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
</PropertyGroup>
```

后处理文件夹归档:

```
<Target Name="PostBuild" AfterTargets="PostBuildEvent">
  <Exec Command=":: 检查 Lib、Dll文件夹路径是否存在 &#xD;&#xA;IF NOT EXIST &quot;$(TargetDir)libs&quot;
(&#xD;&#xA;  MD &quot;$(TargetDir)libs&quot;&#xD;&#xA;)&#xD;&#xA;&#xD;&#xA;:: 将指定的 dll、xml、
pdb、config文件移动到 libs文件夹 &#xD;&#xA;move &quot;$(TargetDir)*.dll&quot;
&quot;$(TargetDir)libs&quot;&#xD;&#xA;move &quot;$(TargetDir)*.xml&quot;
&quot;$(TargetDir)libs&quot;&#xD;&#xA;move &quot;$(TargetDir)*.pdb&quot;
&quot;$(TargetDir)libs&quot;&#xD;&#xA;move &quot;$(TargetDir)*.config&quot;
&quot;$(TargetDir)libs&quot;&#xD;&#xA;&#xD;&#xA;:: 将 runtimes文件夹移动到 libs文件夹 &#xD;&#xA;move
&quot;$(TargetDir)runtimes&quot; &quot;$(TargetDir)libs&quot;&#xD;&#xA;&#xD;&#xA;:: 把主程序的相关文件从
libs转移出来 &#xD;&#xA;move &quot;$(TargetDir)libs\NLog.config&quot;
&quot;$(TargetDir)NLog.config&quot;&#xD;&#xA;move &quot;$(TargetDir)libs\$(ProjectName).exe.config&quot;
&quot;$(TargetDir)$(ProjectName).exe.config&quot;&#xD;&#xA;move
&quot;$(TargetDir)libs\$(ProjectName).exe.xml&quot;
&quot;$(TargetDir)$(ProjectName).exe.xml&quot;&#xD;&#xA;move
&quot;$(TargetDir)libs\$(ProjectName).pdb&quot; &quot;$(TargetDir)$(ProjectName).pdb&quot;" />
</Target>



//添加归档文件夹
```

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
    <startup>
        <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.8" />
    </startup>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    //□□□□□□
      <probing privatePath="libs"/>
      <dependentAssembly>
        <assemblyIdentity name="System.Runtime.CompilerServices.Unsafe"
publicKeyToken="b03f5f7f11d50a3a" culture="neutral" />
        <bindingRedirect oldVersion="0.0.0.0-6.0.0.0" newVersion="6.0.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
  <appSettings>
    <add key="HostAddress" value="https://dcc.hzchum.com" />
  </appSettings>
</configuration>
```

visual studio 常用开发命令

清理所有换成

```
C:\Users\adminpvc>dotnet nuget locals all --clear
正在清除         NuGet HTTP 缓存   : C:\Users\adminpvc\AppData\Local\NuGet\v3-cache
正在清除         NuGet 全局包文件夹          : E:\Users\adminpvc\.nuget\packages
正在清除         NuGet 临时缓存       : C:\Users\adminpvc\AppData\Local\Temp\NuGetScratch
正在清除         NuGet 插件缓存       : C:\Users\adminpvc\AppData\Local\NuGet\plugins-cache
本地资源已清除。
```

git常规

修改仓库url

```
# 查看远端地址
git remote -v
# 查看远端仓库名
git remote
# 重新设置远程仓库
git remote set-url origin https://gitee.com/xx/xx.git (新地址)
```

删除远程仓库修改url

```
# 删除远程的仓库
git remote rm origin
# 重新添加远程仓库
git remote add origin  https://gitee.com/xx/xx.git(新地址)
```

对当前仓库设置用户与邮箱

```
git config user.name "cnb"
git config user.email "cnb@example.com"   # 邮箱可按需填写，有些平台不校验
```

查看当前仓库用户与邮箱

```
git config user.name
git config user.email
```

客户端初始化

```
git init
```

```
//创建     .gitignore (如果还没有)
touch .gitignore



//添加所有文件并提交
git add .
git commit -m "初始化      Vue项目    "

//关联远程仓库
git remote add origin *****

//配置认证 (推荐使用                  Token + Credential Manager)
//如果你使用          HTTPS + Token (推荐)，首次推送时                  Git 会提示你输入用户名和密码:
//Username: 你的用户名 (如              san)
//Password: 你的访问令牌 (如                passsworortoken)
//Git 会自动保存凭据，以后无需再输。
//如果你想提前配置凭据，可以运行:

echo "protocol=https
host=cnb.cool
username=cnb
password=buuymQND8keC17" | git credential approve

//推送到远程仓库
git push -u origin master
# 强制推送 (          ⚠ 慎用，会覆盖远程)
git push -u origin main --force
```

删除本地仓库

```
//切换到其他分支
git checkout main
git branch -d feature
//强制删除
git branch -D feature
```

节点切换

```
//查看提交记录
git log --oneline --graph --decorate -n 5
```

```
//切到指定记录
git reset --hard <commit-hash>


//推荐指定远程和分支名，避免误操作            ,当远程已有新提交时会拒绝推送，防止覆盖他人代码
git push origin micro --force-with-lease
```

生活小本本

20250821 勇者不争，善者不懦

20260105 淘宝网购白天比晚上贵，杀熟

专利软著系列

统计代码行数

```
^b*[^:b#/]+.*$
```

正则全局搜索