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;
}
```

版本号 #2
由 叶子 创建于 4 六月 2025 21:22:50
由 叶子 更新于 4 六月 2025 22:07:54