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)
    {
        // □□□□
    }
}

//□□□□      -Double-check Locking□□□□□□□□            ——□□□□□□□□□□
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;
        }
    }
}

//□□    lock □□□□□□□□
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
{
    // □□□□
}
finally
{
    Monitor.Exit(_sync);
}




//□□□□
//1□□□□□□□□□□              TryEnter□
private readonly object _sync = new object();
```

```
public bool TryAccess(int timeoutMs)
{
    if (Monitor.TryEnter(_sync, timeoutMs))
    {
        try
        {
            Console.WriteLine("成功获取锁，执行操作...");
            return true;
        }
        finally
        {
            Monitor.Exit(_sync);
        }
    }
    else
    {
        Console.WriteLine("未能在指定时间内获取锁");
        return false;
    }
}
```

用途：防止因等待锁而无限阻塞。

```
// 2、线程间通信：Wait / Pulse

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

// 消费者线程
public void Consumer()
{
    Monitor.Enter(_sync);
    try
    {
        while (!_dataReady)
        {
            Console.WriteLine("消费者等待数据...");
            Monitor.Wait(_sync); // 释放锁，等待通知
```

```
        }
        Console.WriteLine("□□□□□□□□□□                ");
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}

// □□□□□□
public void Producer()
{
    Monitor.Enter(_sync);
    try
    {
        Console.WriteLine("□□□□□□□□□□                ...");
        Thread.Sleep(1000); // □□□□□□
        _dataReady = true;
        Monitor.Pulse(_sync); // □□□□□□□□
    }
    finally
    {
        Monitor.Exit(_sync);
    }
}
Wait() □□□□        while □□□□□□□□□□□□
Pulse() □□□□□□□□□              PulseAll() □□□□□□□□



//□□□□□□□□□□

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

public void UpdateValue(int newValue)
{
    Monitor.Enter(_sync);
    try
    {
        _value = newValue;
```

```
        Console.WriteLine($"□□□□□        : {_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
    {
        // □□□□□□
    }
    finally
    {
        _semaphore.Release();
    }
}

///□□□□□□□□□□□□□□

bool acquired = await _semaphore.WaitAsync(TimeSpan.FromSeconds(2));
if (acquired)
{
    try
    {
        // □□□□□□
    }
    finally
    {
        _semaphore.Release();
    }
}
else
{
    Console.WriteLine("□□□□□□□□□□□□                    ");
}
```

ReaderWriterLockSlim

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

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

```
    {
        // □□□□
    }
    finally
    {
        _rwLock.ExitReadLock();
    }
}

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

SpinLock/SpinWait

```
SpinLock spinLock = new SpinLock();
bool lockTaken = false;
try
{
    spinLock.Enter(ref lockTaken);
    // □□□□□□
}
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）。

版本号 #13
由 叶子 创建于 4 六月 2025 19:58:14
由 叶子 更新于 30 十月 2025 10:32:42