

進銷存可能會有管理庫存問題 那假設 訂單有更新產品 那 商品也異動產品 及 庫存自己也異動產品，那麼可能會有重發問題/非同步 導致資料建置有誤，該怎麼解決



為了解決進銷存系統中的並發問題和數據一致性問題，可以採取以下方法：

1. **使用事務(Transactions)：**

使用資料庫事務來確保多個操作要麼全部成功，要麼全部失敗。這樣可以避免部分操作成功，部分操作失敗的情況。

2. **悲觀鎖(Pessimistic Locking)：**

當一個操作需要更新某個數據時，對該數據加鎖，直到操作完成並提交事務。這可以防止其他操作同時訪問該數據，但會降低並發性。

3. **樂觀鎖(Optimistic Locking)：**

通過檢查數據的版本號或時間戳來確保在更新數據時沒有其他操作更改過數據。如果數據版本不匹配，則回滾並重試操作。

4. **事件驅動(Event-Driven)：**

使用事件驅動架構，當某個操作完成後，觸發相關事件，讓其他操作在事件觸發後處理數據變更。這樣可以減少耦合並提高系統的彈性。

5. **分布式鎖(Distributed Locking)：**

如果系統是分布式的，可以使用分布式鎖來確保跨節點的一致性。常見的實現有使用Redis或ZooKeeper。

以下是使用事務和樂觀鎖的範例：

## 使用事務

csharp

複製程式碼

```
public class OrderService : IOrderService
{
    private readonly IProductRepository _productRepository;
    private readonly IOrderRepository _orderRepository;
    private readonly IInventoryRepository _inventoryRepository;
    private readonly ApplicationDbContext _context;

    public OrderService(IProductRepository productRepository, IOrderRepository orderRepository, IInventoryRepository inventoryRepository, ApplicationDbContext context)
    {
        _productRepository = productRepository;
        _orderRepository = orderRepository;
        _inventoryRepository = inventoryRepository;
        _context = context;
    }

    public async Task UpdateOrderAsync(Order order)
    {
        using var transaction = await _context.Database.BeginTransactionAsync();
        try
        {
            var existingOrder = await _orderRepository.GetByIdAsync(order.Id);
            if (existingOrder == null)
            {
                throw new ArgumentException("Order not found");
            }

            await _orderRepository.UpdateAsync(order);
            await _inventoryRepository.UpdateInventoryAsync(order.ProductId, order.Quantity);

            await transaction.CommitAsync();
        }
        catch
        {
            await transaction.RollbackAsync();
            throw;
        }
    }
}
```



## 使用樂觀鎖

在你的模型中添加版本號：

csharp

複製程式碼

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
    public int Quantity { get; set; }
    public int Version { get; set; }
}
```

更新產品時檢查版本號：

csharp

複製程式碼

```
public async Task UpdateProductAsync(Product product)
{
    var existingProduct = await _context.Products.SingleOrDefaultAsync(p => p.Id == product.Id);
    if (existingProduct == null)
    {
        throw new ArgumentException("Product not found");
    }

    if (existingProduct.Version != product.Version)
    {
        throw new DbUpdateConcurrencyException("Product has been updated by another user");
    }

    existingProduct.Name = product.Name;
    existingProduct.Price = product.Price;
    existingProduct.Quantity = product.Quantity;
    existingProduct.Version++;

    _context.Products.Update(existingProduct);
    await _context.SaveChangesAsync();
}
```

## 使用事件驅動架構

定義事件：

csharp

 複製程式碼

```
public class ProductUpdatedEvent
{
    public int ProductId { get; set; }
    public int NewQuantity { get; set; }
}
```

發布事件：

csharp

 複製程式碼

```
public class EventPublisher
{
    public void Publish(ProductUpdatedEvent productUpdatedEvent)
    {
        // 發布事件邏輯
    }
}
```

訂閱事件：

csharp

 複製程式碼

```
public class InventoryService
{
    public void OnProductUpdated(ProductUpdatedEvent productUpdatedEvent)
    {
        // 處理庫存更新邏輯
    }
}
```

## 分布式鎖

使用Redis來實現分布式鎖：

csharp

複製程式碼

```
public class DistributedLockService
{
    private readonly IConnectionMultiplexer _redis;

    public DistributedLockService(IConnectionMultiplexer redis)
    {
        _redis = redis;
    }

    public async Task<bool> AcquireLockAsync(string key, TimeSpan expiration)
    {
        var db = _redis.GetDatabase();
        return await db.StringSetAsync(key, "locked", expiration, When.NotExist);
    }

    public async Task ReleaseLockAsync(string key)
    {
        var db = _redis.GetDatabase();
        await db.KeyDeleteAsync(key);
    }
}
```

在服務中使用分布式鎖：

csharp

複製程式碼

```
public async Task UpdateOrderAsync(Order order)
{
    var lockKey = $"order_lock_{order.Id}";
    if (!await _distributedLockService.AcquireLockAsync(lockKey, TimeSpan.From
    {
        throw new InvalidOperationException("Unable to acquire lock");
    }

    try
    {
        // 更新訂單和庫存的邏輯
    }
    finally
    {
        await _distributedLockService.ReleaseLockAsync(lockKey);
    }
}
```

這些方法可以幫助你確保數據的一致性，避免並發更新導致的數據錯誤。選擇適合你的系統需求的方法，並根據具體情況進行調整。

🔊 📄 ↺ 🔒 ⚙️

那鎖通常都用在服務層嗎



是的，鎖通常用在服務層。這是因為服務層負責業務邏輯和數據的一致性，確保在進行關鍵操作時，系統的狀態是一致的。鎖可以幫助確保多個併發操作不會引起數據的不一致性。

## 1. 悲觀鎖 (Pessimistic Locking)

悲觀鎖會在數據被讀取後立刻加鎖，直到事務結束才釋放鎖。這種方式適合高併發且更新頻繁的場景。

實現範例：

```
cssharp 複製程式碼

public class OrderService : IOrderService
{
    private readonly IProductRepository _productRepository;
    private readonly IOrderRepository _orderRepository;
    private readonly IInventoryRepository _inventoryRepository;
    private readonly AppDbContext _context;

    public OrderService(IProductRepository productRepository, IOrderRepository orderRepository, IInventoryRepository inventoryRepository, AppDbContext context)
    {
        _productRepository = productRepository;
        _orderRepository = orderRepository;
        _inventoryRepository = inventoryRepository;
        _context = context;
    }

    public async Task UpdateOrderAsync(Order order)
    {
        using var transaction = await _context.Database.BeginTransactionAsync();
        try
        {
            var existingOrder = await _orderRepository.GetByIdForUpdateAsync(order.Id);
            if (existingOrder == null)
            {
                throw new ArgumentException("Order not found");
            }

            // 悲觀鎖：在這裡會鎖定需要更新的數據
            await _orderRepository.UpdateAsync(order);
            await _inventoryRepository.UpdateInventoryAsync(order.ProductId, order.Quantity);

            await transaction.CommitAsync();
        }
        catch
        {
            await transaction.RollbackAsync();
            throw;
        }
    }
}
```



## 2. 樂觀鎖 (Optimistic Locking)

樂觀鎖不會在讀取數據時加鎖，而是在更新數據時檢查數據的版本號或時間戳。如果數據在讀取後被其他事務修改過，更新操作會失敗。

實現範例：

csharp

複製程式碼

```
public async Task UpdateProductAsync(Product product)
{
    var existingProduct = await _context.Products.SingleOrDefaultAsync(p => p
    if (existingProduct == null)
    {
        throw new ArgumentException("Product not found");
    }

    if (existingProduct.Version != product.Version)
    {
        throw new DbUpdateConcurrencyException("Product has been updated by a
    }

    existingProduct.Name = product.Name;
    existingProduct.Price = product.Price;
    existingProduct.Quantity = product.Quantity;
    existingProduct.Version++;

    _context.Products.Update(existingProduct);
    await _context.SaveChangesAsync();
}
```

### 3. 分布式鎖 (Distributed Locking)

分布式鎖在分佈式系統中很有用，可以確保跨多個節點的一致性。常見的實現方法包括使用Redis或ZooKeeper。

使用Redis的分布式鎖：

csharp

複製程式碼

```
public class DistributedLockService
{
    private readonly IConnectionMultiplexer _redis;

    public DistributedLockService(IConnectionMultiplexer redis)
    {
        _redis = redis;
    }

    public async Task<bool> AcquireLockAsync(string key, TimeSpan expiration)
    {
        var db = _redis.GetDatabase();
        return await db.StringSetAsync(key, "locked", expiration, When.NotExist);
    }

    public async Task ReleaseLockAsync(string key)
    {
        var db = _redis.GetDatabase();
        await db.KeyDeleteAsync(key);
    }
}
```

在服務層中使用分布式鎖：

```
cssharp 複製程式碼

public class OrderService : IOrderService
{
    private readonly IProductRepository _productRepository;
    private readonly IOrderRepository _orderRepository;
    private readonly IInventoryRepository _inventoryRepository;
    private readonly DistributedLockService _distributedLockService;

    public OrderService(IProductRepository productRepository, IOrderRepository orderRepository, IInventoryRepository inventoryRepository, DistributedLockService distributedLockService)
    {
        _productRepository = productRepository;
        _orderRepository = orderRepository;
        _inventoryRepository = inventoryRepository;
        _distributedLockService = distributedLockService;
    }

    public async Task UpdateOrderAsync(Order order)
    {
        var lockKey = $"order_lock_{order.Id}";
        if (!await _distributedLockService.AcquireLockAsync(lockKey, TimeSpan.FromSeconds(10)))
        {
            throw new InvalidOperationException("Unable to acquire lock");
        }

        try
        {
            var existingOrder = await _orderRepository.GetByIdAsync(order.Id);
            if (existingOrder == null)
            {
                throw new ArgumentException("Order not found");
            }

            await _orderRepository.UpdateAsync(order);
            await _inventoryRepository.UpdateInventoryAsync(order.ProductId, order.Quantity);
        }
        finally
        {
            await _distributedLockService.ReleaseLockAsync(lockKey);
        }
    }
}
```

這些鎖機制可以在服務層實現，確保多個操作在高併發環境下的數據一致性和完整性。選擇適合你的應用場景的鎖機制，並根據具體需求進行實施。

比如處理一筆訂單的過程發現需更新到的product 有100筆 那分布式布鎖  
該如何實現



處理一筆訂單需要更新100筆產品數據，可以使用分布式鎖來確保數據一致性。這裡有一種方法，展示了如何在分布式環境下對多個產品應用鎖，以避免數據競爭和不一致性。

## 分步驟實現方案

### 1. 獲取鎖

- 為每個需要更新的產品生成一個唯一的鎖鍵。
- 嘗試獲取所有產品的鎖。如果所有鎖都成功獲取，則進行更新；如果任何一個鎖獲取失敗，則釋放已獲取的鎖並重試。

### 2. 更新數據

- 獲取所有鎖後，開始更新產品數據和訂單數據。

### 3. 釋放鎖

- 完成數據更新後，釋放所有鎖。

以下是C#代碼範例，展示如何實現上述步驟：

csharp

複製程式碼

```
public class DistributedLockService
{
    private readonly IConnectionMultiplexer _redis;

    public DistributedLockService(IConnectionMultiplexer redis)
    {
        _redis = redis;
    }

    public async Task<bool> AcquireLockAsync(string key, TimeSpan expiration)
    {
        var db = _redis.GetDatabase();
        return await db.StringSetAsync(key, "locked", expiration, When.NotExist);
    }

    public async Task ReleaseLockAsync(string key)
    {
        var db = _redis.GetDatabase();
        await db.KeyDeleteAsync(key);
    }

    public async Task<bool> AcquireLocksAsync(IEnumerable<string> keys, TimeSpan expiration)
    {
        var tasks = keys.Select(key => AcquireLockAsync(key, expiration)).ToArray();
        var results = await Task.WhenAll(tasks);
        return results.All(result => result);
    }

    public async Task ReleaseLocksAsync(IEnumerable<string> keys)
    {
        var tasks = keys.Select(ReleaseLockAsync).ToArray();
        await Task.WhenAll(tasks);
    }
}
```

```
public class OrderService : IOrderService
{
    private readonly IProductRepository _productRepository;
    private readonly IOrderRepository _orderRepository;
    private readonly IInventoryRepository _inventoryRepository;
    private readonly DistributedLockService _distributedLockService;
    private readonly AppDbContext _context;

    public OrderService(IProductRepository productRepository, IOrderRepository orderRepository, IInventoryRepository inventoryRepository, DistributedLockService distributedLockService, AppDbContext context)
    {
        _productRepository = productRepository;
        _orderRepository = orderRepository;
        _inventoryRepository = inventoryRepository;
        _distributedLockService = distributedLockService;
        _context = context;
    }
}
```

```

public async Task UpdateOrderAsync(Order order, IEnumerable<int> productIds)
{
    var lockKeys = productIds.Select(id => $"product_lock_{id}").ToList();

    const int retryCount = 5;
    var expiration = TimeSpan.FromMinutes(1);

    for (int i = 0; i < retryCount; i++)
    {
        if (await _distributedLockService.AcquireLocksAsync(lockKeys, expiration))
        {
            try
            {
                using var transaction = await _context.Database.BeginTransactionAsync();
                try
                {
                    // 更新訂單和產品邏輯
                    await _orderRepository.UpdateAsync(order);
                    foreach (var productId in productIds)
                    {
                        var product = await _productRepository.GetByIdAsync(productId);
                        // 假設有更新產品的業務邏輯
                        await _productRepository.UpdateAsync(product);
                        await _inventoryRepository.UpdateInventoryAsync(productId, product);
                    }
                    await transaction.CommitAsync();
                }
                catch
                {
                    await transaction.RollbackAsync();
                    throw;
                }
            }
            finally
            {
                await _distributedLockService.ReleaseLocksAsync(lockKeys);
            }
            return; // 成功處理訂單
        }
        await Task.Delay(TimeSpan.FromMilliseconds(100 * (i + 1))); // 指
    }

    throw new InvalidOperationException("Unable to acquire locks after multiple attempts");
}

```

## 關鍵點

### 1. 獲取鎖：

- `AcquireLocksAsync`` 方法並行地嘗試獲取所有需要的鎖。如果所有鎖都成功獲取，返回 `true``；否則，返回 `false``。
- 使用指數退避策略進行重試，以避免因為鎖競爭導致的高頻重試。

### 2. 釋放鎖：

- 無論操作成功與否，都需要釋放已獲取的鎖。這一點在 `finally`` 塊中處理。

### 3. 數據一致性：

- 使用資料庫事務確保多個操作的原子性，如果任何操作失敗，則回滾整個事務。

這種方法可以有效處理大量產品的異動，確保數據一致性，同時最大限度減少鎖的競爭。