Java Thread Part 2 - 多執行緒同步/非同步操作與應用場景

第三部分：替代 synchronized 與 wait/notify/notifyAll 的機制

1. ReentrantLock

- 提供比 synchronized 更靈活的鎖機制
- 支持公平鎖和可中斷鎖
- 使用 `lock()` 和 `unlock()` 來加鎖和釋放鎖
- **ReentrantLock 比 synchronized 好的地方：**
    - 支持公平鎖，可以設定為先申請鎖的執行緒優先獲取鎖
    - 支持可中斷鎖，執行緒可以在等待鎖時被中斷
    - 提供更靈活的鎖機制，允許在不同方法中共享同一把鎖
- **Java SDK 版本**：從 Java 5 開始支援 (位於 `java.util.concurrent.locks` 套件中)
- **生活應用案例：**
    - 在銀行系統中，每個帳戶都有一個鎖，確保同一時間只有一個操作能夠進行。使用 ReentrantLock
      可以設定為公平鎖，確保先申請鎖的執行緒優先獲取鎖。

2. Condition

- 與 ReentrantLock 搭配使用，替代 wait/notify
- 提供更靈活的等待和通知機制
- 使用 `await()` 和 `signal()` 來等待和通知
- **Condition 比單純使用 ReentrantLock 好的地方：**
    - 提供多個條件變量，允許更細粒度的等待和通知控制
    - 解決了單一條件變量的局限性，可以對不同的條件進行等待和通知
- **Java SDK 版本**：從 Java 5 開始支援 (位於 `java.util.concurrent.locks` 套件中)
- **生活應用案例：**

- 在餐廳系統中，廚師只有在烹飪材料到達後才能開始煮飯。使用 Condition
  來讓廚師等待材料到達後再開始工作，確保資源的合理使用。

3. CyclicBarrier

- 允許一組執行緒互相等待，直到所有執行緒都達到一個屏障點
- 使用 `await()` 來等待其他執行緒
- **CyclicBarrier 的優點：**
  - 支持重複使用，適用於多次同步的場景
  - 可以設置屏障動作，在所有執行緒達到屏障點時執行
- **Java SDK 版本**：從 Java 5 開始支援 (位於 `java.util.concurrent` 套件中)
- **生活應用案例：**
  - 在跑步比賽中，所有選手都需要準備好並在同一時刻開始比賽。使用 CyclicBarrier 可以讓所有選手等待到達屏障點後再開始比賽。

4. Semaphore

- 控制同時訪問共享資源的執行緒數量
- 支持公平性和可中斷的獲取
- 使用 `acquire()` 和 `release()` 來獲取和釋放許可
- **Semaphore 的優點：**
  - 可以控制訪問共享資源的執行緒數量，防止資源過載
  - 支持公平性和可中斷的獲取，增強靈活性
- **模式：**
  - 公平模式（true）：保證等待時間最長的執行緒優先獲取許可，避免飢餓現象

5. CountDownLatch

- 允許一個或多個執行緒等待其他執行緒完成操作
- 使用 `await()` 等待計數器歸零，`countDown()` 進行計數
- **CountDownLatch 的優點：**

    - 簡單易用，適用於一次性的同步操作
    - 可以讓多個執行緒等待同一個條件達成，進行協作
- **Java SDK 版本**：從 Java 5 開始支援 (位於 `java.util.concurrent` 套件中)
- **生活應用案例：**
    - 在公司團建活動中，所有員工都需要完成報到才能開始活動。使用 CountDownLatch
      可以讓活動組織者等待所有員工完成報到後再開始活動。

6. CompletableFuture

- 提供非同步計算的操作
- 支持鏈式調用
- 使用 `thenApply()`, `thenAccept()`, `thenCompose()` 等方法進行非同步操作
- **CompletableFuture 的優點：**
    - 提供非同步計算操作的簡單方式
    - 支持鏈式調用，允許多個非同步操作依次執行
- **Java SDK 版本**：從 Java 8 開始支援 (位於 `java.util.concurrent` 套件中)
- **生活應用案例：**
    - 在任務管理系統中，CompletableFuture
      可以用來異步處理任務，確保不同的任務能夠高效且非同步地進行。

7. Concurrent Collections

- 提供執行緒安全的集合類，如 `ConcurrentHashMap`（分段鎖：部分無鎖操作），`ConcurrentLinkedQueue`（無鎖集合：CAS 原子操作）
- 使用 `CAS (Compare-And-Swap)`：使用硬體級別的原子操作來實現無鎖的執行緒安全操作，避免了傳統鎖的開銷，提高了性能
- **Concurrent Collections 的優點：**
    - 提供執行緒安全的集合類，簡化多執行緒環境中的數據結構操作
    - 提高多執行緒環境下的性能與效率

- **Java SDK 版本**：從 Java 5 開始支援 (位於 `java.util.concurrent` 套件中)

以下是針對每個機制的學習範例，這些範例展示了如何使用 `ReentrantLock`、`Condition`、`CyclicBarrier`、`Semaphore`、`CountDownLatch`、`CompletableFuture` 和 `Concurrent Collections`：

1. ReentrantLock 使用範例

```
import java.util.concurrent.locks.ReentrantLock;

public class BankAccount {
    private ReentrantLock lock = new ReentrantLock();
    private int balance = 100;

    public void withdraw(int amount) {
        lock.lock();
        try {
            if (balance >= amount) {
                balance -= amount;
                System.out.println("Withdrawal successful. New balance: " + balance);
            } else {
                System.out.println("Insufficient funds.");
            }
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        BankAccount account = new BankAccount();
        Runnable task = () -> account.withdraw(50);
        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
    }
}
```

2. Condition 使用範例

```
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class Restaurant {
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();
    private boolean ingredientsArrived = false;

    public void chefWait() throws InterruptedException {
        lock.lock();
        try {
            while (!ingredientsArrived) {
                System.out.println("Chef is waiting for ingredients...");
                condition.await();
            }
            System.out.println("Ingredients arrived. Chef starts cooking.");
        } finally {
            lock.unlock();
        }
    }

    public void deliverIngredients() {
```

```
        lock.lock();
        try {
            ingredientsArrived = true;
            System.out.println("Ingredients delivered.");
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Restaurant restaurant = new Restaurant();
        Thread chef = new Thread(() -> {
            try {
                restaurant.chefWait();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });
        chef.start();

        Thread.sleep(2000);  // Simulate delay in delivering ingredients
        restaurant.deliverIngredients();
    }
}
```

3. CyclicBarrier 使用範例

```
import java.util.concurrent.CyclicBarrier;

public class Race {
    private static final int RUNNERS = 3;

    public static void main(String[] args) {
        CyclicBarrier barrier = new CyclicBarrier(RUNNERS, () -> {
            System.out.println("All runners are ready. Start the race!");
        });

        for (int i = 0; i < RUNNERS; i++) {
            new Thread(new Runner(barrier)).start();
        }
    }

    static class Runner implements Runnable {
        private CyclicBarrier barrier;

        Runner(CyclicBarrier barrier) {
            this.barrier = barrier;
        }

        @Override
        public void run() {
            try {
                System.out.println(Thread.currentThread().getName() + " is ready.");
                barrier.await();
                System.out.println(Thread.currentThread().getName() + " starts running.");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
}
```

4. Semaphore 使用範例

```
import java.util.concurrent.Semaphore;

public class ParkingLot {
    private Semaphore spots = new Semaphore(3);
```

```
    public void parkCar() {
        try {
            System.out.println(Thread.currentThread().getName() + " is trying to park.");
            spots.acquire();
            System.out.println(Thread.currentThread().getName() + " parked.");
            Thread.sleep(2000);  // Simulate parking duration
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + " left the parking
spot.");
            spots.release();
        }
    }

    public static void main(String[] args) {
        ParkingLot lot = new ParkingLot();
        for (int i = 0; i < 5; i++) {
            new Thread(lot::parkCar).start();
        }
    }
}
```

5. CountDownLatch 使用範例

```
import java.util.concurrent.CountDownLatch;

public class CompanyEvent {
    private static final int EMPLOYEES = 5;
    private CountDownLatch latch = new CountDownLatch(EMPLOYEES);

    public void employeeReport() {
        System.out.println(Thread.currentThread().getName() + " has completed
registration.");
        latch.countDown();
    }

    public void startEvent() throws InterruptedException {
        latch.await();
        System.out.println("All employees have registered. Event starts!");
    }

    public static void main(String[] args) throws InterruptedException {
        CompanyEvent event = new CompanyEvent();

        for (int i = 0; i < EMPLOYEES; i++) {
            new Thread(event::employeeReport).start();
        }

        event.startEvent();
    }
}
```

6. CompletableFuture 使用範例

```
import java.util.concurrent.CompletableFuture;

public class AsyncTask {

    public static void main(String[] args) {
        CompletableFuture<Void> future = CompletableFuture.supplyAsync(() -> {
            System.out.println("Fetching data...");
            return "Data fetched";
        }).thenApply(result -> {
            System.out.println("Processing " + result);
            return result.length();
        }).thenAccept(length -> {
            System.out.println("Data length: " + length);
        });
```

```
        future.join();  // Wait for all tasks to complete
    }
}
```

7. Concurrent Collections 使用範例

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class LibrarySystem {
    private ConcurrentHashMap<String, Integer> bookInventory = new ConcurrentHashMap<>();

    public void borrowBook(String book) {
        bookInventory.compute(book, (k, v) -> (v == null || v == 0) ? 0 : v - 1);
        System.out.println(Thread.currentThread().getName() + " borrowed " + book);
    }

    public void returnBook(String book) {
        bookInventory.merge(book, 1, Integer::sum);
        System.out.println(Thread.currentThread().getName() + " returned " + book);
    }

    public static void main(String[] args) {
        LibrarySystem library = new LibrarySystem();
        library.bookInventory.put("Java Concurrency", 2);

        ExecutorService executor = Executors.newFixedThreadPool(3);
        executor.submit(() -> library.borrowBook("Java Concurrency"));
        executor.submit(() -> library.borrowBook("Java Concurrency"));
        executor.submit(() -> library.returnBook("Java Concurrency"));

        executor.shutdown();
    }
}
```

這些範例展示了每個同步/非同步機制的基本使用方法，您可以根據實際需求進行擴展和應用。

情境選擇題

1/9. 問題: 你正在設計一個銀行系統，確保在多個執行緒同時操作同一個帳戶時，帳戶餘額的計算是正確的。哪個選項最適合這個場景？

A) 使用 `ConcurrentHashMap` 來存儲帳戶餘額
B) 使用 `synchronized` 關鍵字來保護存款和取款方法
C) 使用 `CountDownLatch` 來等待所有操作完成
D) 使用 `Semaphore` 來限制同時操作帳戶的執行緒數量

答案: B) 使用 `synchronized` 關鍵字來保護存款和取款方法

原因: `synchronized` 關鍵字可以確保同一時間只有一個執行緒可以執行存款或取款操作，從而保護共享資源（帳戶餘額）的正確性。

2/9. 問題: 你正在開發一個生產者-消費者模式的應用程式，生產者產生數據並將其放入緩衝區，消費者從緩衝區中取出數據進行處理。當緩衝區為空時，消費者需要等待生產者生成數據。你應該使用哪種同步機制（複選）？

A) 使用 `CyclicBarrier` 來同步生產者和消費者
B) 使用 `wait/notify` 方法來協調生產者和消費者
C) 使用 `ReentrantLock` 來保護緩衝區
D) 使用 `CompletableFuture` 來實現非阻塞的生產者-消費者模式

答案: B) 使用 `wait/notify` 方法來協調生產者和消費者
C) 使用 `ReentrantLock` 和 `Condition` 來保護緩衝區

其他選項不適合的原因：

- A) `CyclicBarrier` 用於多個執行緒達到屏障點後一起繼續，不適用於生產者-消費者模式。
- D) `CompletableFuture` 適合異步任務的非阻塞操作，但不適用於經典的生產者-消費者模式。

3/9. 問題: 你需要設計一個高並發系統，要求可以在某些情況下中斷等待鎖的執行緒。哪個選項適合這個需求？

A) 使用 `synchronized` 關鍵字
B) 使用 `ReentrantLock` 並配合 `lockInterruptibly()` 方法
C) 使用 `CountDownLatch`
D) 使用 `CyclicBarrier`

答案: B) 使用 `ReentrantLock` 並配合 `lockInterruptibly()` 方法

原因: `ReentrantLock` 提供了 `lockInterruptibly()` 方法，允許執行緒在等待鎖的過程中響應中斷。

4/9. 問題: 你需要設計一個多執行緒的生產者-消費者系統，並且希望使用條件變量來更靈活地控制等待和通知。哪個選項最適合這個需求？

A) 使用 `wait/notify` 方法
B) 使用 `ReentrantLock` 和 `Condition`
C) 使用 `Semaphore`
D) 使用 `CyclicBarrier`

答案: B) 使用 `ReentrantLock` 和 `Condition`

原因: `Condition` 與 `ReentrantLock` 配合使用，提供比 `wait/notify` 更靈活的等待和通知機制，可以精確控制多個條件變量。

5/9. 問題: 你正在設計一個並行計算系統，要求每個階段所有參與的執行緒都必須達到屏障點才能進入下一階段。哪個選項最適合這個需求？

A) 使用 `CountDownLatch`
B) 使用 `CyclicBarrier`
C) 使用 `Semaphore`
D) 使用 `CompletableFuture`

答案: B) 使用 `CyclicBarrier`

原因: `CyclicBarrier` 適用於需要多次重用的同步操作，可以讓一組執行緒在每個階段都互相等待，直到所有執行緒都達到屏障點。

6/9. 問題: 你需要設計一個系統，限制同時訪問某資源的執行緒數量。哪個選項最適合這個需求？

A) 使用 `CountDownLatch`
B) 使用 `CyclicBarrier`
C) 使用 `Semaphore`
D) 使用 `CompletableFuture`

答案: C) 使用 `Semaphore`

原因: `Semaphore` 可以控制同時訪問共享資源的執行緒數量，防止資源過載。

7/9. 問題: 你正在設計一個系統，要求主執行緒在多個工作執行緒完成初始化工作後才繼續執行。哪個選項最適合這個需求？

A) 使用 `CyclicBarrier`
B) 使用 `CountDownLatch`
C) 使用 `Semaphore`
D) 使用 `CompletableFuture`

答案: B) 使用 `CountDownLatch`

原因: `CountDownLatch` 適用於一次性的同步操作，主執行緒可以等待多個工作執行緒完成初始化工作後再繼續執行。

8/9. 問題: 你正在設計一個在線點餐系統，要求在顧客下單後可以非阻塞地通知廚房準備餐點，並在餐點準備好後通知顧客。哪個選項最適合這個需求？

A) 使用 `CyclicBarrier`
B) 使用 `CountDownLatch`
C) 使用 `CompletableFuture`
D) 使用 `Semaphore`

答案: C) 使用 `CompletableFuture`

原因: `CompletableFuture` 支持非阻塞的異步操作，可以在顧客下單後非阻塞地通知廚房準備餐點，並在餐點準備好後通知顧客。

9/9. 問題: 你需要設計一個圖書館借書系統，確保多個讀者可以同時借閱和歸還書籍，不會發生數據競爭。哪個選項最適合這個需求？

A) 使用 `ConcurrentHashMap` 或 `ConcurrentLinkedQueue`
B) 使用 `CountDownLatch`
C) 使用 `CyclicBarrier`
D) 使用 `Semaphore`

答案: A) 使用 `ConcurrentHashMap` 或 `ConcurrentLinkedQueue`

原因: `Concurrent Collections` 提供執行緒安全的集合類，簡化多執行緒環境中的數據結構操作，適合圖書館借書系統中多個讀者同時操作的場景。

進階範例

1. **ReentrantLock** 進階範例：讀寫鎖的應用

```
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class SharedResource {
    private ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
    private int data = 0;
    public void writeData(int newData) {
        rwLock.writeLock().lock();
        try {
            System.out.println(Thread.currentThread().getName() + " is writing.");
            data = newData;
            System.out.println("Data written: " + data);
        } finally {
            rwLock.writeLock().unlock();
        }
    }
    public void readData() {
        rwLock.readLock().lock();
        try {
            System.out.println(Thread.currentThread().getName() + " is reading. Data: " + data);
        } finally {
            rwLock.readLock().unlock();
        }
    }
    public static void main(String[] args) {
```

```
        SharedResource resource = new SharedResource();

        Runnable readTask = resource::readData;
        Runnable writeTask = () -> resource.writeData((int) (Math.random() * 100));

        Thread writer = new Thread(writeTask);
        Thread reader1 = new Thread(readTask);
        Thread reader2 = new Thread(readTask);

        writer.start();
        reader1.start();
        reader2.start();
    }
}
```

解釋：這個範例展示了如何使用 `ReentrantReadWriteLock` 來實現多執行緒讀寫鎖。讀鎖允許多個執行緒同時讀取資源，而寫鎖則保證一次只有一個執行緒可以寫入資源，從而提高讀操作的性能。

2. **Condition** 進階範例：生產者/消費者模式

```
import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

import java.util.LinkedList;

import java.util.Queue;


public class ProducerConsumer {

    private Queue<Integer> queue = new LinkedList<>();

    private final int CAPACITY = 5;

    private Lock lock = new ReentrantLock();

    private Condition notFull = lock.newCondition();

    private Condition notEmpty = lock.newCondition();


    public void produce() throws InterruptedException {

        int value = 0;

        while (true) {

            lock.lock();

            try {

                while (queue.size() == CAPACITY) {

                    notFull.await();

                }

                queue.offer(value);

                System.out.println("Produced " + value);

                value++;
```

```
            notEmpty.signal();
        } finally {
            lock.unlock();
        }
    }
}
public void consume() throws InterruptedException {
    while (true) {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                notEmpty.await();
            }
            int value = queue.poll();
            System.out.println("Consumed " + value);
            notFull.signal();
        } finally {
            lock.unlock();
        }
    }
}
public static void main(String[] args) {
    ProducerConsumer pc = new ProducerConsumer();

    Thread producerThread = new Thread(() -> {
```

```
            try {
                pc.produce();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumerThread = new Thread(() -> {
            try {
                pc.consume();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        producerThread.start();
        consumerThread.start();
    }
}
```

解釋：這個範例展示了如何使用 `Condition` 來實現生產者/消費者模式。生產者在隊列已滿時等待，消費者在隊列為空時等待，通過條件變量控制生產與消費的同步操作。

3. **CyclicBarrier** 進階範例：重複使用屏障

```
import java.util.concurrent.BrokenBarrierException;

import java.util.concurrent.CyclicBarrier;



public class RelayRace {

    private static final int RUNNERS = 3;

    private CyclicBarrier barrier = new CyclicBarrier(RUNNERS, () -> System.out.println("Relay race
round complete!"));



    public void runRace() {

        for (int i = 0; i < RUNNERS; i++) {

            new Thread(new Runner(barrier)).start();

        }

    }



    static class Runner implements Runnable {

        private CyclicBarrier barrier;



        Runner(CyclicBarrier barrier) {

            this.barrier = barrier;

        }



        @Override

        public void run() {

            try {

                for (int round = 1; round <= 3; round++) {
```

```
                System.out.println(Thread.currentThread().getName() + " is ready for round " + round);
                barrier.await();
                System.out.println(Thread.currentThread().getName() + " is running in round " + round);
            }
        } catch (InterruptedException | BrokenBarrierException e) {
            Thread.currentThread().interrupt();
        }
    }
  }
  public static void main(String[] args) {
    RelayRace race = new RelayRace();
    race.runRace();
  }
}
```

解釋：這個範例展示了 `CyclicBarrier` 如何用於多次重複操作的場景，模擬了接力賽中選手在每輪比賽前等待所有人準備好後再開始。

4. **Semaphore** 進階範例：限制同時訪問的資源數量

```
import java.util.concurrent.Semaphore;

public class NetworkConnection {
   private static final int MAX_CONNECTIONS = 3;
   private Semaphore semaphore = new Semaphore(MAX_CONNECTIONS);


   public void connect() {
      try {
         semaphore.acquire();
         System.out.println(Thread.currentThread().getName() + " connected to the server.");
         Thread.sleep(2000);  // Simulate network connection duration
      } catch (InterruptedException e) {
         Thread.currentThread().interrupt();
      } finally {
         System.out.println(Thread.currentThread().getName() + " disconnected from the server.");
         semaphore.release();
      }
   }
   public static void main(String[] args) {
      NetworkConnection connection = new NetworkConnection();
      for (int i = 0; i < 5; i++) {
         new Thread(connection::connect).start();
      }
   }
}
```

解釋：這個範例展示了如何使用 `Semaphore` 限制同時連接的最大客戶端數量，適用於控制網絡連接數量或其他共享資源的使用。

5. **CountDownLatch** 進階範例：多階段同步

```
import java.util.concurrent.CountDownLatch;

public class SoftwareDeployment {
    private static final int PHASES = 3;
    private CountDownLatch latch = new CountDownLatch(PHASES);

    public void performPhase(int phase) throws InterruptedException {
        System.out.println("Phase " + phase + " in progress...");
        Thread.sleep(1000);  // Simulate time taken to complete the phase
        System.out.println("Phase " + phase + " completed.");
        latch.countDown();
    }

    public void awaitCompletion() throws InterruptedException {
        latch.await();
        System.out.println("All phases completed. Deployment successful!");
    }

    public static void main(String[] args) throws InterruptedException {
        SoftwareDeployment deployment = new SoftwareDeployment();

        for (int i = 1; i <= PHASES; i++) {
            final int phase = i;
            new Thread(() -> {
                try {
```

```
                deployment.performPhase(phase);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();
    }

    deployment.awaitCompletion();
  }
}
```

解釋：這個範例展示了如何使用 `CountDownLatch` 同步多個階段的操作，模擬了軟件部署中的多階段過程，直到所有階段完成後才繼續進行。

6. **CompletableFuture** 進階範例：並行非同步任務

```
import java.util.concurrent.CompletableFuture;
import java.util.List;
import java.util.Arrays;
public class AsyncCalculations {

    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        CompletableFuture<Void> future = CompletableFuture.allOf(
                numbers.stream()
                        .map(n -> CompletableFuture.supplyAsync(() -> calculateSquare(n)))
                        .map(f -> f.thenAccept(result -> System.out.println("Square: " + result)))
                        .toArray(CompletableFuture[]::new)
        );

        future.join();  // Wait for all tasks to complete
        System.out.println("All calculations completed.");
    }
    public static int calculateSquare(int number) {
        System.out.println("Calculating square for: " + number);
        return number * number;
    }
}
```

解釋：這個範例展示了如何使用 `CompletableFuture` 執行多個非同步任務並等待它們全部完成。這裡每個任務都是獨立計算數字的平方，並行處理後等待所有計算完成。

7. **Concurrent Collections** 進階範例：無鎖隊列操作

```
import java.util.concurrent.ConcurrentLinkedQueue;

public class HelpDesk {
    private ConcurrentLinkedQueue<String> requests = new ConcurrentLinkedQueue<>();
    public void submitRequest(String request) {
        requests.offer(request);
        System.out.println("Request submitted: " + request);
    }
    public void processRequest() {
        String request;
        while ((request = requests.poll()) != null) {
            System.out.println("Processing request: " + request);
        }
    }
    public static void main(String[] args) {
        HelpDesk helpDesk = new HelpDesk();
        // Multiple clients submitting requests
        new Thread(() -> helpDesk.submitRequest("Issue with laptop")).start();
        new Thread(() -> helpDesk.submitRequest("Cannot connect to WiFi")).start();

        // Help desk staff processing requests
        new Thread(helpDesk::processRequest).start();
    }
}
```

解釋：這個範例展示了如何使用 `ConcurrentLinkedQueue` 在多執行緒環境中無鎖地進行佇列操作，模擬了一個技術支持系統，提交請求和處理請求能夠並行進行。