جافا في الخيوط تسجيل

المجموعات، المتغيرات، مثل المشتركة الموارد مع خيوط عدة يتفاعل عندما جافا! في الخيوط تزامن موضوع في نغوص دعونا
يوفر متوقعة. غير نتائج إلى يؤدي مما البعض، بعضها مع الخيوط تتداخل حيث السباق حالات من عليك المللفات، أو
اختيار يمكنك حتى بينها وأقارن أمثلة، وأعرض منها، لكل سأشرح .Semaphore و ،Lock ،synchronized الغرض: لهذا أدوات عدة جافا
احتياجاتك. حسب الأفضل

---

1. كلمة synchronized

الأبسط، التزامن أداة إنها الوقت. نفس في طريقة أو الكود من كتلة ينفذ أن يمكن واحد خيط فقط أن synchronized لكلمة تضمن
جافا. قلب في مبنية

عملها كيفية

- لكل كائن مراقب. لديه جافا كائن كل مراقب
- القفل. عن إفراج يتم حتى الأخرى الخيوط تنتظر

مع عداد مثال synchronized:

```
class Counter {
    private int count = 0;

    public synchronized void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
            }
```

```java
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount()); // Always 2000
    }
}
```

- بدون synchronized، قد تتداخل count++ (قراءة، تعديل، كتابة)، مما يعطي نتيجة أقل من 2000.
- synchronized يقفل Counter، مما يضمن تحديثات ذرية.

كتلة synchronized: يمكنك أيضًا قفل كود محدد بدلاً من طريقة كاملة:

```java
public void increment() {
    synchronized (this) {
        count++;
    }
}
```

- استخدم كائنًا مختلفًا كقفل إذا كنت تريد تحكمًا أكثر دقة:

```java
private final Object lock = new Object();
public void increment() {
    synchronized (lock) {
        count++;
    }
}
```

المزايا والعيوب

- **المزايا**: بسيط، مبني، جيد للاستخدام الأساسي.
- **العيوب**: لا يوجد مرونة (مثل عدم القدرة على إيقاف خيط في انتظار)، يظل مقفلًا حتى يتم إفراج عنه.

---

2. واجهة Lock □□□□□.□□□□.□□□□□□□□□□.□□□□□□

توفر واجهة Lock مثل□ ReentrantLock□ أكثر من التحكم من synchronized. إنها جزء من حزمة java.util.concurrent و تم تقديمها لحل مشاكل synchronized.

الميزات الرئيسية

- □ دعوات lock() و unlock() صريحة.
- □ يدعم قفل المحاولة، الأوقات الزمنية، القابل وقفل للإيقاف.
- □ خيار العدل □الخيوط تنتظر في الترتيب□.

مثال: عداد مع ReentrantLock

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

class Counter {
    private int count = 0;
    private final Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            count++;
        } finally {
            lock.unlock(); // Always unlock in finally
        }
    }

    public int getCount() {
        return count;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter();
        Runnable task = () -> {
            for (int i = 0; i < 1000; i++) {
                counter.increment();
```

```java
            }
        };

        Thread t1 = new Thread(task);
        Thread t2 = new Thread(task);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Final count: " + counter.getCount()); // Always 2000
    }
}
```

- try-finally يضمن إفراج عن القفل حتى إذا حدث استثناء.

الميزات المتقدمة

- **قفل المحاولة:** محاولة غير مقفلة للحصول على القفل:

```java
if (lock.tryLock()) {
    try {
        count++;
    } finally {
        lock.unlock();
    }
} else {
    System.out.println("Couldn't acquire lock");
}
```

- **الوقت الزمني:** انتظر لفترة محدودة:

```java
if (lock.tryLock(1, TimeUnit.SECONDS)) { ... }
```

- **القابل للإيقاف:** السماح للخيوط في انتظار أن يتم إيقافها:

```java
lock.lockInterruptibly();
```

المزايا والعيوب

- **المزايا:** مرن، يدعم الميزات المتقدمة، التحكم الصريح.
- **العيوب:** أكثر تعقيدًا، مطلوب إفراج عن القفل يدويًا، خطر نسيان الإفراج.

3. Semaphore

يسيطر Semaphore على الوصول إلى الموارد من خلال الحفاظ على مجموعة من الإذنات. إنه مفيد للحد من التزامن لشيء ما أن يكون هناك أكثر من 5 خيوط يمكنها الوصول إلى الموارد.

كيفية عمله

- يحصل الخيوط على الإذن باستخدام acquire().
- يحرر الإذن باستخدام release().
- إذا لم يكن هناك أي إذن متاحة، تنتظر الخيوط.

مثال: الحد من الاتصالات المتزامنة بقاعدة البيانات

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

class ConnectionPool {
    private final Semaphore semaphore = new Semaphore(3); // Max 3 connections

    public void connect() {
        try {
            semaphore.acquire();
            System.out.println(Thread.currentThread().getName() + " connected");
            Thread.sleep(1000); // Simulate work
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            semaphore.release();
            System.out.println(Thread.currentThread().getName() + " disconnected");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        ConnectionPool pool = new ConnectionPool();
        Runnable task = () -> pool.connect();

        Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++) {
            threads[i] = new Thread(task, "Thread-" + i);
```

```
            threads[i].start();
        }
    }
}
```

□ الإذن. عن إفراج يتم حتى الأخرى الخيوط تنتظر الوقت؛ نفس في□الاتصال□ يمكن خيوط 3 فقط

المتقدمة الميزات

- □ العدل: new Semaphore(3, true) يضمن ترتيب □□□□.
- □ المحاولة قفل: semaphore.tryAcquire() دون يحدد قفل.

والعيوب المزايا

- □ المزايا: مثالي للمجموعات للموارد، نظام الإذنات المرن.
- □ العيوب: أكثر تعقيدًا من synchronized أو Lock، ليس للاستبعاد المتبادل البسيط.

ExecutorService مع الجمع

خيوط: مجموعة مع الأدوات هذه تعمل هكذا

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

class SharedResource {
    private int value = 0;
    private final Lock lock = new ReentrantLock();

    public void increment() {
        lock.lock();
        try {
            value++;
        } finally {
            lock.unlock();
        }
    }

    public int getValue() {
```

```java
        return value;
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        SharedResource resource = new SharedResource();
        ExecutorService executor = Executors.newFixedThreadPool(4);

        for (int i = 0; i < 10; i++) {
            executor.submit(() -> {
                for (int j = 0; j < 100; j++) {
                    resource.increment();
                }
            });
        }

        executor.shutdown();
        executor.awaitTermination(5, TimeUnit.SECONDS);
        System.out.println("Final value: " + resource.getValue()); // Always 1000
    }
}
```

المقارنة

الميزة	synchronized	Lock	Semaphore
سهولة الاستخدام	عالية	متوسطة	منخفضة
المرونة	منخفضة	عالية	متوسطة
الدقة	طريقة كتلة	صريح	بناءً على الإذن
القابل للإيقاف	لا	نعم	نعم
خيار العدل	لا	نعم	نعم
حالة الاستخدام	الاستبعاد المتبادل البسيط	التحكم المتقدم	الحد من الموارد

- **synchronized**: واحد. متغير حماية مثل بسيط متبادل استبعاد سريع،
- **Lock**: العمل عمليات مثل للإيقاف القابل أو الزمنية، الأوقات المحاولة، قفل إلى تحتاج المعقدة. العمل عمليات مثل للإيقاف القابل أو الزمنية، الأوقات المحاولة، قفل إلى تحتاج
- **Semaphore**: عدل. من الحد الإتصالات، مجموعات مثل الموارد إلى المتزامن الوصول من الحد