CALLICODER

Java Locks and Atomic Variables Tutorial

Rajeev Kumar Singh • Java • Jul 24, 2017 • 6 mins read

In multithreaded programs, access to shared variables must be synchronized in order to prevent race conditions.

In the previous tutorial, We learned how to use `synchronized` methods and `synchronized` blocks to protect concurrent access to shared variables and avoid race conditions.

Java's `synchronized` keyword internally uses the intrinsic lock associated with an object to gain exclusive access to the object's member fields.

Instead of using an intrinsic lock via the `synchronized` keyword, you can also use various Locking classes provided by Java's

CALLICODER

In this tutorial, we'll learn how to use these Locking classes provided by Java to synchronize access to shared variables.

Finally, We'll also look at a modern way of thread synchronization via various `Atomic` classes provided by Java concurrency API.

Locks

1. ReentrantLock

ReentrantLock is a mutually exclusive lock with the same behavior as the intrinsic/implicit lock accessed via the `synchronized` keyword.

ReentrantLock, as the name suggests, possesses reentrant characteristics. That means a thread that currently owns the lock can acquire it more than once without any problem.

Following is an example showing how to create a thread safe method using `ReentrantLock` -

```
import java.util.concurrent.locks.Reent

class ReentrantLockCounter {
    private final ReentrantLock lock =
```

CALLICODER

```
    // Thread Safe Increment
    public void increment() {
        lock.lock();
        try {
            count = count + 1;
        } finally {
            lock.unlock();
        }
    }
}
```

The idea is very simple - Any thread calling the `increment()` method will first acquire the lock and then increment the `count` variable. When it's done incrementing the variable, it can release the lock so that other threads waiting for the lock can acquire it.

Also, note that I've used a `try/finally` block in the above example. The finally block ensures that the lock is released even if some exception occurs.

The ReentrantLock also provides various methods for more fine-grained control -

```
import java.util.concurrent.ExecutorSer
import java.util.concurrent.Executors;
import java.util.concurrent.locks.Reent
```

CALLICODER

```
    private int count = 0;

    public int incrementAndGet() {
        // Check if the lock is current
        System.out.println("IsLocked :

        // Check if the lock is acquire
        System.out.println("IsHeldByCur

        // Try to acquire the lock
        boolean isAcquired = lock.tryLo
        System.out.println("Lock Acquir

        if(isAcquired) {
            try {
                Thread.sleep(2000);
                count = count + 1;
            } catch (InterruptedExcepti
                throw new IllegalStateE
            } finally {
                lock.unlock();
            }
        }
        return count;
    }
}

public class ReentrantLockMethodsExampl
```

CALLICODER

```
        ExecutorService executorService

        ReentrantLockMethodsCounter loc

        executorService.submit(() -> {
           System.out.println("Incremen
                   lockMethodsCounter.i
        });

        executorService.submit(() -> {
            System.out.println("Increme
                    lockMethodsCounter.
        });

        executorService.shutdown();
    }
}
```

```
# Output
IsLocked : false
IsHeldByCurrentThread : false
Lock Acquired : true

IsLocked : true
IsHeldByCurrentThread : false
Lock Acquired : false

IncrementCount (Second Thread) : 0
```

The `tryLock()` method tries to acquire the lock without pausing the thread. That is, If the thread couldn't acquire the lock because it was held by some other thread, then It returns immediately instead of waiting for the lock to be released.

You can also specify a timeout in the `tryLock()` method to wait for the lock to be available -

```
lock.tryLock(1, TimeUnit.SECONDS);
```

The thread will now pause for one second and wait for the lock to be available. If the lock couldn't be acquired within 1 second then the thread returns.

2. ReadWriteLock

ReadWriteLock consists of a pair of locks - one for read access and one for write access. The read lock may be held by multiple threads simultaneously as long as the write lock is not held by any thread.

ReadWriteLock allows for an increased level of concurrency. It performs better compared to

CALLICODER

```
import java.util.concurrent.locks.ReadW
import java.util.concurrent.locks.Reent

class ReadWriteCounter {
    ReadWriteLock lock = new ReentrantR

    private int count = 0;

    public int incrementAndGetCount() {
        lock.writeLock().lock();
        try {
            count = count + 1;
            return count;
        } finally {
            lock.writeLock().unlock();
        }
    }

    public int getCount() {
        lock.readLock().lock();
        try {
            return count;
        } finally {
            lock.readLock().unlock();
        }
    }
}
```

no thread calls `incrementAndGetCount()`. If any thread calls `incrementAndGetCount()` method and acquires the write-lock, then all the reader threads will pause their execution and wait for the writer thread to return.

Atomic Variables

Java's concurrency api defines several classes in `java.util.concurrent.atomic` package that support Atomic operations on single variables.

Atomic classes internally use compare-and-swap instructions supported by modern CPUs to achieve synchronization. These instructions are generall

Consider the following example where we use the `AtomicInteger` class to make sure that the increment to the count variable happens atomically.

Concurrency Basics

Thread and Runnable

Executor Service & Thread Pool

Callable and Future

Thread Synchronization

Locks and Atomic Variables

```
import java.util.concurrent.ExecutorSer
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.Atom

class AtomicCounter {
    private AtomicInteger count = new A
```

```
    }

    public int getCount() {
        return count.get();
    }
}

public class AtomicIntegerExample {
    public static void main(String[] ar
        ExecutorService executorService

        AtomicCounter atomicCounter = n

        for(int i = 0; i < 1000; i++) {
            executorService.submit(() -
        }

        executorService.shutdown();
        executorService.awaitTerminatio

        System.out.println("Final Count
    }
}
```

```
# Output
Final Count is : 1000
```

several threads simultaneously and be sure that the access to the count variable will be synchronized.

Following are some other atomic classes defined inside `java.util.concurrent.atomic` package. -

- AtomicBoolean
- AtomicLong
- AtomicReference

You should use these Atomic classes instead of synchronized keyword and locks whenever possible because they are faster, easier to use, readable and scalable.

Conclusion

Congratulations on finishing the last part of my Java concurrency tutorial series. In this tutorial, we learned how to use Locks and Atomic Variables for thread synchronization. You can find all the code samples used in this tutorial in my github repository.

Thank you for reading. Please ask any questions in the comment section below.

Liked the Article? Share it on Social media!