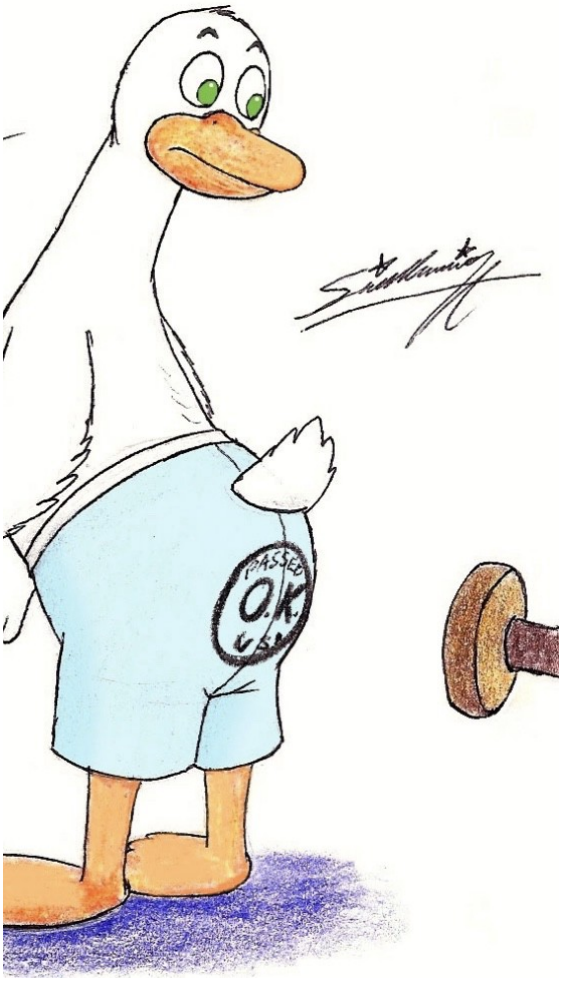
Java Concurrency through Stamped Locks

Aayush Bhatnagar
May 13, 2019 · 5 min read



Java 8 introduces the concept of **Stamped Locks**. Prior to this, we had Reentrant ReadWrite locks as the alternative to avoid full synchronisation. While Reentrant locks still remain, Stamped locks are a new addition to the java concurrency family.

Stamped Locks cannot be re-entrant, which essentially means, that if we acquire a read stamped lock, and we invoke another piece of code, we cannot acquire another write lock within its scope.

On the other hand, Reentrant locks allowed programmers to acquire multiple read locks, as long as there were no write locks. It was later discovered, that if not used

carefully – Reentrant RW locks can lead to thread starvation for the write lock under loaded conditions – which can lead to severe performance issues.

Why are they called "Stamped" ?

There are two elements of a stamped lock, namely – the lock version and locking mode. The lock version is also known as a stamp, and is essentially a value of type "long" returned every time a lock is acquired.

There are methods available which "try" to acquire a lock – used for loosely synchronised code segments, and for these methods the stamp value returned can be "zero", if the lock acquisition fails.

Example -

> *public long tryConvertToWriteLock(long stamp)*

Locking Modes

There are three locking modes available for synchronisation granularity – READ, WRITE and OPTIMISTIC READ locks.

Write Locks

The writeLock() method after acquiring the lock, returns a stamp that can be used in the method unlockWrite(long) to release the lock. When the lock is held in write mode, no read locks may be obtained, and all optimistic read validations will fail.

Read Locks

The readLock() method after acquiring the lock, returns a stamp that can be used in the method unlockRead(long) to release the lock.

Optimistic Read Locks

This locking mode can be visualised as an "advanced booking attempt" for a read lock – which may or may not be granted.

The tryOptimisticRead() method returns a stamp only if the lock is not currently held in write mode.

The validate(long) method returns true if the lock has not been acquired in write mode since obtaining a given stamp.

However, if the write lock has been acquired since – the optimistic read fails.

Example -

```
long stamp = lock.tryOptimisticRead();

if (!lock.validate(stamp)) {

stamp = lock.readLock();

try {

this.value = newValue;

} finally {

lock.unlockRead(stamp);

}

}
```

For cases where the read locks are needed for very short segments, optimistic read lock acquisition can help improve performance.

However, this has to be used with caution as the write threads can change the field values and cause logical issues in the code.

To be assured of thread safe behaviour, read locks should be used.

Conditional Locking & Conversion

Stamped Lock APIs also allow for conditional locking by using the “try” methods.

In this process, it is possible to interchange read lock stamps to writes and vice versa. The following methods are available -

```
public long tryConvertToWriteLock(long stamp)
```

If the stamp represents a write lock, then it is simply returned. If it represents a read lock, and the write lock is available, the read lock is released and a write stamp is returned. If the stamp represents an optimistic read lock, then a write stamp is returned only if immediately available. This method returns zero in all other cases.

> *public long tryConvertToReadLock(long stamp)*

If the stamp represents a write lock, releases the lock and obtains a read lock. If the stamp represents a read lock, then it is returned. If the stamp is for an optimistic read lock, then a read lock is acquired and a read stamp is returned. This method returns zero in all other cases when the lock cannot be acquired.

> *public long tryConvertToOptimisticRead(long stamp)*

If the stamp represents holding a lock, then it is released and an observation stamp is returned. If the stamp represents an optimistic read, then it is returned if its validation is successful. This method returns zero in all other cases.

Time Bound Locking

The stamped lock can also accept a guard timer – which would “wait” for a certain amount if time to acquire a lock.

This is useful for cases where we do not want all our threads to block on a code segment which is held up somewhere, for eg. Socket I/O or File I/O.

In order to guard against such situations, the following methods are provided -

public long tryReadLock(long time,TimeUnit unit) InterruptedException

Non-exclusively acquires the lock if it is available within the given time and the current thread has not been interrupted. Behavior under timeout and interruption matches that specified for method Lock.tryLock(long,TimeUnit).

Parameters:

time – the maximum time to wait for the lock

unit – the time unit of the time argument

Returns:

A stamp that can be used to unlock or convert mode, or zero if the lock is not available

Throws:

InterruptedException – if the current thread is interrupted before acquiring the lock

On the same lines the "tryWriteLock" method is also available with a similar signature -

public long tryWriteLock(long time,TimeUnit unit) throws InterruptedException

Unlocking in exception conditions

There may be situations, where we need to release the read and write locks without the need of a stamp value.

This would be needed especially on exceptions; where the stamp value may not be available.

For such cases, the "tryUnlock" method is available without the need for specifying a stamp value.

> *public boolean tryUnlockWrite()*
>
> *public boolean tryUnlockRead()*

Blocking vs Graceful Locking Behavior

If we review the stamped lock methods, it is seen that a wide variety of locking behaviour is provided.

For example, read locks can be acquired unconditionally in a blocking manner by invoking the method ***public long readLock()***

Then we have the method ***tryReadLock***, which does not block, and acquires the non-exclusive lock if it is available immediately, else returns zero.

Furthermore, we have ***optimistic locking*** where the lock stamp can be validated at a later point of time to check for the presence of a write lock before the non-exclusive read lock can be acquired.

Finally, we have the ***timed locks*** – where we can attribute a timeout value for exclusive read lock access.

This allows the developer to fine tune the locking behaviour of their application, provided that they are in full control of concurrent access semantics.

The same granularity exists for write locks, which also follow a similar method signature.

Conclusion

Developers should gradually move to stamped locks if they are using Reentrant locks.

Stamped locks need to be used with utmost care, so that the concurrency semantics of the variables is not compromised, and logical multi-threading errors are not encountered.

If you liked this article, feel free to connect with me on **LinkedIn**

Programming Java Multithreading Software Development Software