The `ReentrantReadWriteLock` and its drawbacks

Before Java 8 we had a `ReentrantReadWriteLock` class that was used for reading and writing data in a thread-safe manner.

Here are a few of the important points about `ReentrantReadWriteLock`:

1. Multiple threads can acquire a read lock simultaneously.
2. Only one thread can acquire a write lock.
3. If a thread wants to acquire a write lock and there are some threads that have read lock, the thread will wait until all the threads release the read lock.

There are a few problems with using the `ReentrantReadWriteLock` class:

1. It can lead to starvation.
2. Sometimes it can be significantly slower than other synchronizers.

The improvements provided by `StampedLock`

To overcome these disadvantages, `StampedLock` is added. Apart from providing separate read and write locks, also has a feature for optimistic locking for reading operations.

`StampedLock` also provides a method to upgrade read lock to write lock, which is not in `ReentrantReadWriteLock` in Java.

The `StampedLock` class provides three locking modes:

1. Read
2. Write
3. Optimistic read

Let's look at a basic example of `StampedLock`. In the below example we have used a few operations that are available in the `StampedLock` class.

a) `readLock()` - This method is used to acquire the read lock. This method returns a stamp that should be used while releasing the lock.

b) `unlockRead(long stamp)` - This method is used to release the read lock. This method takes a stamp as an input. If the stamp provided does not match, `IllegalStateExceptio`n is thrown.

a) `writeLock()` - This method is used to acquire the write lock. This method returns a stamp that should be used while releasing the lock.

b) `unlockWrite(long stamp)` - This method is used to release the write lock. This method takes a stamp as an input. If the stamp provided does not match then IllegalStateException is thrown.

```
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.locks.StampedLock;

public class StampedLockDemo {

    static Map<String, Integer> data = new HashMap<>();
    static StampedLock lock = new StampedLock();

    // Method to read data from the Map.
    public static Integer readDataFromMap(String key) {

        long stamp = lock.readLock();
        try {
            return data.get(key);
        } finally {
            lock.unlockRead(stamp);
        }
    }

    // Method to write data to the Map.
    public static void writeDataToMap(String key, Integer value) {
        long stamp = lock.writeLock();
        try {
            data.put(key, value);
        } finally {
            lock.unlockWrite(stamp);
        }
```

Run Save Reset

Non blocking lock methods

The `readLock()` and `writeLock()` methods discussed above are blocking methods. This means that if a thread `t1` tries to acquire a read lock and some other thread, like `t2` has already acquired a write lock, the thread `t1` will block.

If we want, our thread should not block. We can use one of the following methods:

1. `tryReadLock()` - Acquire the lock if it is immediately available otherwise don't block.
2. `tryWriteLock()` - Acquire the lock if it is immediately available otherwise don't block.
3. `tryReadLock(long time, TimeUnit unit)` - Try to acquire the lock till the provided time limit.
4. `tryWriteLock(long time, TimeUnit unit)` - Try to acquire the lock until the provided time limit.

Let's look at an example of this.

```
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.locks.StampedLock;

public class StampedLockDemo {

    static Map<String, Integer> data = new HashMap<>();
    static StampedLock lock = new StampedLock();

    //Method to read data from the Map. Since we are using tryReadLock(), the thread will not block.
    public static Integer readDataFromMap(String key) {

        long stamp = lock.tryReadLock();
        int result = 0;
        if(stamp != 0L){
            try {
                result = data.get(key);
            } finally {
                lock.unlockRead(stamp);
            }
        }
        return result;
    }

    //Method to write data to the Map. Since we are using tryWriteLock(), the thread will not block.
    public static void writeDataToMap(String key, Integer value) {
        long stamp = lock.tryWriteLock();
        if(stamp != 0L){
```

Run Save Reset

Optimized reading

Acquiring and releasing a lock is a costly process and can lead to starvation.

Suppose we have a use case where data is read frequently but rarely updated. In this case, it is not advisable to get a read lock every time we are reading.

In such situations, we can use `tryOptimisticRead()` for our reading operations. Here is how `tryOptimisticRead()` works.

Suppose thread `t1` tries to get an optimistic lock

1. If some other thread has already acquired a write lock, thread `t1` returns. It is not able to acquire the lock.
2. If some other thread has already acquired a read lock then `tryOptimisticRead()` returns an **observation stamp**.

> Please note that we have not acquired a lock. We have just received an observation stamp.

3. Now, thread `t1` will completes the reading, and then it calls the `validate(long stamp)` method. This method tells if a write operation after the observation stamp was generated.
4. If the validation is successful, it means we have the most recent data and we are good.
5. If the validation is not successful, it means that we may not have the most recent data and we need to do something else.

So, this is the whole concept of optimistic locking. The benefit of optimizing locking is that we are not actually acquiring the lock so it is a cheap operation.

```
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.locks.StampedLock;

public class StampedLockDemo {

    static Map<String, Integer> data = new HashMap<>();
    static StampedLock lock = new StampedLock();

    public static Integer readDataFromMap(String key) {
        int result = 0;
        long stamp = lock.tryOptimisticRead();

        if(stamp != 0L){
            result = data.get(key);
        }

        if (!lock.validate(stamp)) {
        // This means that the data was modified after we called optimistic read.
        // Do extra work here to get the latest data.
        }
        return result;
    }

    public static void writeDataToMap(String key, Integer value) {
        long stamp = lock.tryWriteLock();
        if(stamp != 0L){
            try {
```

Run Save Reset

Converting lock modes

In the `StampedLock` class, it is possible to convert one lock mode to another, i.e., we can convert a read lock to a write lock and vice versa.

We can convert the locks' modes using the following methods:

1. `tryConvertToWriteLock(long stamp)`

- If the lock we are trying to convert is already a write lock, then return the lock.
- If the lock we are trying to convert is a read lock and a write lock is available then return the write lock and release the read lock.
- If the lock we are trying to convert is an optimistic read lock, then return the write lock if available.
- Return zero.

2. `tryConvertToReadLock(long stamp)`

- If the lock we are trying to convert is already a read lock then return the lock.
- If the lock we are trying to convert is a write lock then return the read lock and release write lock.
- If the lock we are trying to convert is an optimistic read lock, and then return the read lock if it is available.
- Return zero.

3. `tryConvertToOptimisticRead(long stamp)`

- If the stamp represents an optimistic read lock, then return it if it is validated.
- If the stamp represents a lock then release the lock and return an observation stamp.
- Return zero.