Java synchronized method around parameter value

Asked 8 years, 3 months ago Modified 5 years, 10 months ago Viewed 804 times

5

Consider the following method:

```
public void upsert(int customerId, int somethingElse) {
  // some code which is prone to race conditions
}
```

I want to protect this method from race conditions, but this can only occur if two threads with the same `customerId` are calling it at the same time. If I make the whole method `synchronized` it will reduce the efficiency and it's not really needed. What I really want is to synchronize it around the `customerId`. Is this possible somehow with Java? Are there any built-in tools for that or I'd need a `Map` of `Integers` to use as locks?

Also feel free to advice if you think I'm doing something wrong here :)

Thanks!

java **concurrency** **synchronized**

Share

Improve this question

Follow

edited Sep 14, 2016 at 12:52

Nicolas Filotto

44.9k ● 11 ● 96 ● 127

asked Sep 14, 2016 at 12:50

Anton Belev

13.4k ● 23 ● 73 ● 115

1 It certainly *feels* wrong. Perhaps you should edit your question to provide more detail on *why* you need synchronization in the first place. – Sergei Tachenov Sep 14, 2016 at 12:56

3 upsert sounds like it's executing some database dml. in which case maybe you're trying to fix a transaction isolation problem with locking? if so, that would be a bad idea. – Nathan Hughes Sep 14, 2016 at 12:59

@NathanHughes I'm using an old version of Postgres where upsert is not available yet :) – Anton Belev Sep 14, 2016 at 13:11

2 Perhaps I missed the point, but why don't you just do it in a JDBC transaction and let the DBMS take care of locking for you? – Klitos Kyriacou Sep 14, 2016 at 13:30

2 Answers

Sorted by: Highest score (default)

13

The concept you're looking for is called *segmented locking* or *striped locking*. It is too wasteful to have a separate lock for each customer (locks are quite heavyweight). Instead you want to *partition* your customer ID space into a reasonable number of partitions, matching the desired degree of parallelism. Typically 8-16 would be enough, but this depends on the amount of work the method does.

This outlines a simple approach:

```
private final Object[] locks = new Object[8];

synchronized (locks[customerId % locks.length]) {
    ...implementation...
}
```

Share Improve this answer Follow

answered Sep 14, 2016 at 12:57

Marko Topolnik

200k ● 30 ● 334 ● 452

Google Guava provide special lock to support striped locking - `com.google.common.util.concurrent.Striped` – foal Aug 26, 2023 at 17:09

0

```
    private static final Set<Integer> lockedIds = new HashSet<>();

    private void lock(Integer id) throws InterruptedException {
        synchronized (lockedIds) {
            while (!lockedIds.add(id)) {
                lockedIds.wait();
            }
        }
    }

    private void unlock(Integer id) {
        synchronized (lockedIds) {
            lockedIds.remove(id);
            lockedIds.notifyAll();
        }
    }

    public void upsert(int customerId) throws InterruptedException {
        try {
            lock(customerId);

            //Put your code here.
            //For different ids it is executed in parallel.
            //For equal ids it is executed synchronously.

        } finally {
            unlock(customerId);
        }
    }
```

- **id** can be not only an 'Integer' but any class with correctly overridden 'equals' and 'hashCode' methods.
- **try-finally** - is very important - you must guarantee to unlock waiting threads after your operation even if your operation threw exception.
- It will not work if your back-end is distributed across **multiple servers/JVMs**.

answered Feb 14, 2019 at 11:03

Anton Fil

295 ● 2 ● 12