Java synchronizing based on a parameter (named mutex/lock)

Asked 8 years, 8 months ago Active 2 years, 1 month ago Viewed 17k times

32

13

I'm looking for a way to synchronize a method based on the parameter it receives, something like this:

```
public synchronized void doSomething(name){
//some code
}
```

I want the method `doSomething` to be synchronized based on the `name` parameter like this:

Thread 1: doSomething("a");

Thread 2: doSomething("b");

Thread 3: doSomething("c");

Thread 4: doSomething("a");

Thread 1 , Thread 2 and Thread 3 will execute the code without being synchronized , but Thread 4 will wait until Thread 1 has finished the code because it has the same "a" value.

Thanks

UPDATE

Based on Tudor explanation I think I'm facing another problem: here is a sample of the new code:

```
private HashMap locks=new HashMap();
public void doSomething(String name){
    locks.put(name,new Object());
    synchronized(locks.get(name)) {
        // ...
    }
    locks.remove(name);
}
```

The reason why I don't populate the locks map is because name can have any value.

Based on the sample above , the problem can appear when adding / deleting values from the hashmap by multiple threads in the same time, since HashMap is not thread-safe.

So my question is if I make the `HashMap` a `ConcurrentHashMap` which is thread safe, will the synchronized block stop other threads from accessing locks.get(name) ??

java multithreading synchronized

Share Edit Follow Close Flag

edited Mar 28 '19 at 19:16

Triet Doan
9,053 ● 6 ● 29 ● 59

asked Sep 16 '12 at 20:20

Doua Beri
9,216 ● 15 ● 77 ● 126

Do you have a finite (and reasonably small) number of possible parameters? If so, you can use Tudor's solution and pre-populate a Map of locks. – Frank Pavageau Sep 16 '12 at 20:35

@Frank Pavageau It can work for any number of parameters, just "intern" them (not using `String.intern()` !!!!) and then synchronize on the interned representation. – gpeche Sep 16 '12 at 20:38

1 @gpeche Yes, but that's not apparent from the answer, hence my question so the answer can be refined if needed. I would have suggested a look at Guava's Striped<Lock>, then. – Frank Pavageau Sep 16 '12 at 21:05

What is the reason to synchronize in such a way? Strings are immutable and need not synchronization. Apparently there exists some set of mutable objects which you care of. Why not to synchronize on them directly? – Alexei Kaigorodov Sep 17 '12 at 3:55

So each thread will call the method with a different `name` value? Can multiple threads ever call the method with the same `name` in this scenario? – Tudor Sep 18 '12 at 13:02

8 Answers

Active Oldest Votes

19

Use a map to associate strings with lock objects:

```
Map<String, Object> locks = new HashMap<String, Object>();
locks.put("a", new Object());
locks.put("b", new Object());
// etc.
```

then:

```
public void doSomething(String name){
    synchronized(locks.get(name)) {
        // ...
    }
}
```

Share Edit Follow Flag

answered Sep 16 '12 at 20:23

Tudor
58.9k ● 12 ● 89 ● 138

1 hi. I'm thinking I have a little problem based on your example. The idea is that the name parameter can have any value so I can't create a map with a predefined keys. I was thinking to add a value on the map before the synchronized block and remove it after the synchronized block has finished. However this can lead to synch problems about adding/removing values from map. I was thinking of using ConcurrentHashMap but in this way does the synchronized block work? since ConcurrentHashMap is thread safe. Thanks – Doua Beri Sep 18 '12 at 0:10

@Doua Beri: Can you please give an example of this in the original question? Because from your explanation it's not clear how you want to synchronize multiple threads if each one of them is adding different locks. – Tudor Sep 18 '12 at 6:13

Updated the main question. Thanks – Doua Beri Sep 18 '12 at 12:47

15

TL;DR:

I use ConcurrentReferenceHashMap from the Spring Framework. Please check the code below.

Although this thread is old, it is still interesting. Therefore, I would like to share my approach with Spring Framework.

What we are trying to implement is called **named mutex/lock**. As suggested by Tudor's answer, the idea is to have a `Map` to store the lock name and the lock object. The code will look like below (I copy it from his answer):

```
Map<String, Object> locks = new HashMap<String, Object>();
locks.put("a", new Object());
locks.put("b", new Object());
```

However, this approach has 2 drawbacks:

1. The OP already pointed out the first one: how to synchronize the access to the `locks` hash map?
2. How to remove some locks which are not necessary anymore? Otherwise, the `locks` hash map will keep growing.

The first problem can be solved by using ConcurrentHashMap. For the second problem, we have 2 options: manually check and remove locks from the map, or somehow let the garbage collector knows which locks are no longer used and the GC will remove them. I will go with the second way.

When we use `HashMap`, or `ConcurrentHashMap`, it creates strong references. To implement the solution discussed above, weak references should be used instead (to understand what is a strong/weak reference, please refer to this article or this post).

So, I use ConcurrentReferenceHashMap from the Spring Framework. As described in the documentation:

> A `ConcurrentHashMap` that uses soft or weak references for both keys and values.
>
> This class can be used as an alternative to `Collections.synchronizedMap(new WeakHashMap<K, Reference<V>>())` in order to support better performance when accessed concurrently. This implementation follows the same design constraints as `ConcurrentHashMap` with the exception that null values and null keys are supported.

Here is my code. The `MutexFactory` manages all the locks with `<K>` is the type of the key.

```
@Component
public class MutexFactory<K> {

    private ConcurrentReferenceHashMap<K, Object> map;

    public MutexFactory() {
        this.map = new ConcurrentReferenceHashMap<>();
    }

    public Object getMutex(K key) {
        return this.map.compute(key, (k, v) -> v == null ? new Object() : v);
    }
}
```

Usage:

```
@Autowired
private MutexFactory<String> mutexFactory;

public void doSomething(String name){
    synchronized(mutexFactory.getMutex(name)) {
        // ...
    }
}
```

Unit test (this test uses the awaitility library for some methods, e.g. `await()`, `atMost()`, `until()`):

```
public class MutexFactoryTests {
    private final int THREAD_COUNT = 16;

    @Test
    public void singleKeyTest() {
        MutexFactory<String> mutexFactory = new MutexFactory<>();
        String id = UUID.randomUUID().toString();
        final int[] count = {0};

        IntStream.range(0, THREAD_COUNT)
                .parallel()
                .forEach(i -> {
                    synchronized (mutexFactory.getMutex(id)) {
                        count[0]++;
                    }
                });
        await().atMost(5, TimeUnit.SECONDS)
                .until(() -> count[0] == THREAD_COUNT);
        Assert.assertEquals(count[0], THREAD_COUNT);
    }
}
```

Share Edit Follow Flag

edited Mar 28 '19 at 19:21

answered Mar 28 '19 at 19:15

Triet Doan
9,053 ● 6 ● 29 ● 59

13

The answer of Tudor is fine, but it's static and not scalable. My solution is dynamic and scalable, but it goes with increased complexity in the implementation. The outside world can use this class just like using a `Lock`, as this class implements the interface. You get an instance of a parameterized lock by the factory method `getCanonicalParameterLock`.

```
package lock;

import java.lang.ref.Reference;
import java.lang.ref.WeakReference;
import java.util.Map;
import java.util.WeakHashMap;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public final class ParameterLock implements Lock {

    /** Holds a WeakKeyLockPair for each parameter. The mapping may be deleted upon
garbage collection
     * if the canonical key is not strongly referenced anymore (by the threads using
the Lock). */
    private static final Map<Object, WeakKeyLockPair> locks = new WeakHashMap<>();

    private final Object key;
    private final Lock lock;

    private ParameterLock (Object key, Lock lock) {
        this.key = key;
        this.lock = lock;
    }

    private static final class WeakKeyLockPair {
        /** The weakly-referenced parameter. If it were strongly referenced, the
entries of
         * the lock Map would never be garbage collected, causing a memory leak. */
        private final Reference<Object> param;
```

```
        /** The actual lock object on which threads will synchronize. */
        private final Lock lock;

        private WeakKeyLockPair (Object param, Lock lock) {
            this.param = new WeakReference<>(param);
            this.lock = lock;
        }
    }

    public static Lock getCanonicalParameterLock (Object param) {
        Object canonical = null;
        Lock lock = null;

        synchronized (locks) {
            WeakKeyLockPair pair = locks.get(param);
            if (pair != null) {
                canonical = pair.param.get(); // could return null!
            }
            if (canonical == null) { // no such entry or the reference was cleared in
the meantime
                canonical = param; // the first thread (the current thread) delivers
the new canonical key
                pair = new WeakKeyLockPair(canonical, new ReentrantLock());
                locks.put(canonical, pair);
            }
        }

        // the canonical key is strongly referenced now...
        lock = locks.get(canonical).lock; // ...so this is guaranteed not to return
null
        // ... but the key must be kept strongly referenced after this method returns,
        // so wrap it in the Lock implementation, which a thread of course needs
        // to be able to synchronize. This enforces a thread to have a strong reference
        // to the key, while it isn't aware of it (as this method declares to return a
        // Lock rather than a ParameterLock).
        return new ParameterLock(canonical, lock);
    }

    @Override
    public void lock() {
        lock.lock();
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {
        lock.lockInterruptibly();
    }

    @Override
    public boolean tryLock() {
        return lock.tryLock();
    }

    @Override
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException {
        return lock.tryLock(time, unit);
    }

    @Override
    public void unlock() {
        lock.unlock();
    }

    @Override
    public Condition newCondition() {
        return lock.newCondition();
    }
}
```

Of course you'd need a canonical key for a given parameter, otherwise threads would not be synchronized as they would be using a different Lock.
Canonicalization is the equivalent of the internalization of Strings in Tudor's solution. Where `String.intern()` is itself thread-safe, my 'canonical pool' is not, so I need extra synchronization on the WeakHashMap.

This solution works for any type of Object. However, make sure to implement `equals` and `hashCode` correctly in custom classes, because if not, threading issues will arise as multiple threads could be using different Lock objects to synchronize on!

The choice for a WeakHashMap is explained by the ease of memory management it brings. How else could one know that no thread is using a particular Lock anymore? And if this could be known, how could you safely delete the entry out of the Map? You would need to synchronize upon deletion, because you have a race condition between an arriving thread wanting to use the Lock, and the action of deleting the Lock from the Map. All these things are just solved by using weak references, so the VM does the work for you, and this simplifies the implementation a lot. If you inspected the API of WeakReference, you would find that relying on weak references is thread-safe.

Now inspect this test program (you need to run it from inside the ParameterLock class, due to private visibility of some fields):

```
public static void main(String[] args) {
    Runnable run1 = new Runnable() {

        @Override
        public void run() {
            sync(new Integer(5));
            System.gc();
        }
    };
    Runnable run2 = new Runnable() {

        @Override
        public void run() {
            sync(new Integer(5));
            System.gc();
        }
    };
    Thread t1 = new Thread(run1);
    Thread t2 = new Thread(run2);

    t1.start();
    t2.start();

    try {
        t1.join();
        t2.join();
        while (locks.size() != 0) {
            System.gc();
            System.out.println(locks);
        }
        System.out.println("FINISHED!");
    } catch (InterruptedException ex) {
        // those threads won't be interrupted
    }
}

private static void sync (Object param) {
    Lock lock = ParameterLock.getCanonicalParameterLock(param);
    lock.lock();
    try {
        System.out.println("Thread="+Thread.currentThread().getName()+", lock=" +
((ParameterLock) lock).lock);
        // do some work while having the lock
    } finally {
        lock.unlock();
    }
}
```

Chances are very high that you would see that both threads are using the same lock object, and so they are synchronized. Example output:

```
Thread=Thread-0, lock=java.util.concurrent.locks.ReentrantLock@8965fb[Locked by thread
Thread-0]
Thread=Thread-1, lock=java.util.concurrent.locks.ReentrantLock@8965fb[Locked by thread
Thread-1]
FINISHED!
```

However, with some chance it might be that the 2 threads do not overlap in execution, and therefore it is not required that they use the same lock. You could easily enforce this behavior in debugging mode by setting breakpoints at the right locations, forcing the first or second thread to stop wherever necessary. You will also notice that after the Garbage Collection on the main thread, the WeakHashMap will be cleared, which is of course correct, as the main thread waited for both worker threads to finish their job by calling `Thread.join()` before calling the garbage collector. This indeed means that no strong reference to the (Parameter)Lock can exist anymore inside a worker thread, so the reference can be cleared from the weak hashmap. If another thread now wants to synchronize on the same parameter, a new Lock will be created in the synchronized part in `getCanonicalParameterLock`.

Now repeat the test with any pair that has the same canonical representation (= they are equal, so `a.equals(b)`), and see that it still works:

```
sync("a");
sync(new String("a"))

sync(new Boolean(true));
sync(new Boolean(true));
```

etc.

Basically, this class offers you the following functionality:

- Parameterized synchronization
- Encapsulated memory management
- The ability to work with any type of object (under the condition that `equals` and `hashCode` is implemented properly)
- Implements the Lock interface

This Lock implementation has been tested by modifying an ArrayList concurrently with 10 threads iterating 1000 times, doing this: adding 2 items, then deleting the last found list entry by iterating the full list. A lock is requested per iteration, so in total 10*1000 locks will be requested. No ConcurrentModificationException was thrown, and after all worker threads have finished the total amount of items was 10*1000. On every single modification, a lock was requested by calling `ParameterLock.getCanonicalParameterLock(new String("a"))`, so a new parameter object is used to test the correctness of the canonicalization.

Please note that you shouldn't be using String literals and primitive types for parameters. As String literals are automatically interned, they always have a strong reference, and so if the first thread arrives with a String literal for its parameter then the lock pool will never be freed from the entry, which is a memory leak. The same story goes for autoboxing primitives: e.g. Integer has a caching mechanism that will reuse existing Integer objects during the process of autoboxing, also causing a strong reference to exist. Addressing this, however, this is a different story.

Share Edit Follow Flag

edited Mar 15 '16 at 7:57

answered Feb 5 '15 at 15:29

Timmos

3,032 ● 2 ● 27 ● 39

1

I've found a proper answer through another stackoverflow question: How to acquire a lock by a key

I copied the answer here:

Guava has something like this being released in 13.0; you can get it out of HEAD if you like.

Striped more or less allocates a specific number of locks, and then assigns strings to locks based on their hash code. The API looks more or less like

```
Striped<Lock> locks = Striped.lock(stripes);
Lock l = locks.get(string);
l.lock();
try {
  // do stuff
} finally {
  l.unlock();
}
```

More or less, the controllable number of stripes lets you trade concurrency against memory usage, because allocating a full lock for each string key can get expensive; essentially, you only get lock contention when you get hash collisions, which are (predictably) rare.

Share Edit Follow Flag

answered Jul 28 '17 at 11:29

Thomas Stubbe

1,537 ● 4 ● 19 ● 32

1

I've created a tokenProvider based on the IdMutexProvider of McDowell. The manager uses a `WeakHashMap` which takes care of cleaning up unused locks.

You could find my implementation here.

Share Edit Follow Flag

edited May 23 '17 at 11:48

Community ♦

1 ● 1

answered Feb 26 '15 at 10:23

R. Oosterholt

6,435 ● 2 ● 44 ● 70

1

Check out this framework. Seems you're looking for something like this.

```
public class WeatherServiceProxy {
...
private final KeyLockManager lockManager = KeyLockManagers.newManager();

public void updateWeatherData(String cityName, Date samplingTime, float temperature) {
        lockManager.executeLocked(cityName, new LockCallback() {
                public void doInLock() {
                        delegate.updateWeatherData(cityName, samplingTime,
temperature);
                }
        });
}
```

https://code.google.com/p/jkeylockmanager/

Share Edit Follow Flag

answered Jul 16 '14 at 12:39

Njax3SmmM2x2a0Zf7Hpd

1,276 ● 4 ● 22 ● 42

What if I have to return something from the method. – pavan Apr 20 '15 at 8:41

I have a much simpler, scalable implementation akin to @timmons post taking advantage of guavas `LoadingCache` with `weakValues`. You will want to read

0

the help files on "equality" to understand the suggestion I have made.

Define the following weakValued cache.

```
private final LoadingCache<String,String> syncStrings =
CacheBuilder.newBuilder().weakValues().build(new CacheLoader<String, String>() {
    public String load(String x) throws ExecutionException {
        return new String(x);
    }
});

public void doSomething(String x) {
      x = syncStrings.get(x);
      synchronized(x) {
          ..... // whatever it is you want to do
      }
}
```

Now! As a result of the JVM, we do not have to worry that the cache is growing too large, it only holds the cached strings as long as necessary and the garbage manager/guava does the heavy lifting.

Share Edit Follow Flag

edited Oct 21 '17 at 7:31
wp78de
16k ● 6 ● 33 ● 55

answered Oct 21 '17 at 6:49
Tom Petrillo
1 ● 2

0

I've used a cache to store lock objects. The my cache will expire objects after a period, which really only needs to be longer that the time it takes the synchronized process to run

`

```
import com.google.common.cache.Cache;
import com.google.common.cache.CacheBuilder;

...

private final Cache<String, Object> mediapackageLockCache =
CacheBuilder.newBuilder().expireAfterWrite(DEFAULT_CACHE_EXPIRE,
TimeUnit.SECONDS).build();

...

public void doSomething(foo) {
    Object lock = mediapackageLockCache.getIfPresent(foo.toSting());
    if (lock == null) {
        lock = new Object();
        mediapackageLockCache.put(foo.toString(), lock);
    }

    synchronized(lock) {
        // execute code on foo
        ...
    }
}
```

`

Share Edit Follow Flag

answered Oct 6 '16 at 8:59
JamesP
472 ● 4 ● 10