7 Techniques for thread-safe classes

January 11, 2018

Category: Programming concurrent Java (/categories/multithreaded_programming)

7 Techniques for thread-safe classes

Almost every Java application uses threads. A web server like Tomcat process each request in a separate worker thread, fat clients process long-running requests in dedicated worker threads, and even batch processes use the java.util.concurrent.ForkJoinPool to improve performance.

It is, therefore, necessary to write classes in a thread-safe way, which can be achieved by one of the following techniques:

No state

When multiple threads access the same instance or static variable you must somehow coordinate the access to this variable. The easiest way to do this is simply by avoiding instance or static variables. Methods in classes without instance variables do only use local variables and method arguments. The following example shows such a method which is part of the class java.lang.Math:

```
1  public static int subtractExact(int x, int y) {
2    int r = x - y;
3    if (((x ^ y) & (x ^ r)) < 0) {
4        throw new ArithmeticException("integer overflow");
5    }
6    return r;
7  }
```

No shared state

If you can not avoid state do not share the state. The state should only be owned by a single thread. An example of this technique is the event processing thread of the SWT or Swing graphical user interface frameworks (http://flylib.com/books/en/2.558.1/why_are_guis_single_threaded_.html).

You can achieve thread-local instance variables by extending the thread class and adding an instance variable. In the following example, the field pool and workQueue are local to a single worker thread.

```
package java.util.concurrent;
public class ForkJoinWorkerThread extends Thread {
    final ForkJoinPool pool;
    final ForkJoinPool.WorkQueue workQueue;
}
```

The other way to achieve thread-local variables is to use the class java.lang.ThreadLocal for the fields you want to make thread-local. Here is an example of an instance variable using java.lang.ThreadLocal:

```
public class CallbackState {
public static final ThreadLocal<CallbackStatePerThread> callbackStatePerT
    new ThreadLocal<CallbackStatePerThread>()
    {
       @Override
         protected CallbackStatePerThread  initialValue()
       {
        return getOrCreateCallbackStatePerThread();
       }
    };
}
```

You wrap the type of your instance variable inside the java.lang.ThreadLocal. You can provide an initial value for your java.lang.ThreadLocal through the method initialValue().

The following shows how to use the instance variable:

```
CallbackStatePerThread callbackStatePerThread = CallbackState.callbackStat
```

Through calling the method get() you receive the object associated with the current thread.

Since in application servers a pool of many threads is used to process requests, java.lang.ThreadLocal leads to a high memory consumption in this environment. java.lang.ThreadLocal is therefore not recommended for classes executed by the request processing threads of an application server.

Message passing

If you do not share state using the above techniques you need a way for the threads to communicate. A technique to do this is by passing messages between threads. You can implement message passing using a concurrent queue from the package java.util.concurrent. Or, better yet, use a framework like Akka (https://akka.io/), a framework for actor style concurrency. The following example shows how to send a message with Akka:

```
target.tell(message, getSelf());
```

and receive a message:

```
@Override
public Receive createReceive() {
   return receiveBuilder()
      .match(String.class, s -> System.out.println(s.toLowerCase()))
      .build();
}
```

Immutable state

To avoid the problem that the sending thread changes the message during the message is read by another thread, messages should be immutable. The Akka framework, therefore, has the convention that all messages have to be immutable (https://doc.akka.io/docs/akka/2.5.5/java/actors.html#messages-and-immutability)

When you implement an immutable class you should declare its fields as final. This not only makes sure that the compiler can check that the fields are in fact immutable but also makes them correctly initialized even when they are incorrect published. (https://shipilev.net/blog/2014/jmm-pragmatics/#_part_v_finals) Here is an example of a final instance variable:

```
public class ExampleFinalField
{
  private final int finalField;
  public ExampleFinalField(int value)
  {
   this.finalField = value;
  }
}
```

final is a field modifier. It makes the field immutable not the object the field references to. So the type of the final field should be a primitive type like in the example or also an immutable class.

Use the data structures from java.util.concurrent

Message passing uses concurrent queues for the communication between threads. Concurrent Queues are one of the data structures provided in the package java.util.concurrent. This package provides classes for concurrent maps, queues, dequeues, sets and lists. Those data structures are highly optimized and tested for thread safety.

Synchronized blocks

If you can not use one of the above techniques use synchronized locks. By putting a block inside a synchronized block you make sure that only one thread at a time can execute this section.

```
synchronized(lock)
{
 i++;
}
```

Beware that when you use multiple nested synchronize blocks you risk deadlocks. A deadlock happens when two threads are trying to acquire a lock held by the other thread. (http://vmlens.com/articles/detect_deadlocks/)

Volatile fields

Normal, nonvolatile fields, can be cached in registers or caches. Through the declaration of a variable as volatile, you tell the JVM and the compiler to always return the latest written value. This not only applies to the variable itself but to all values written by the thread which has written to the volatile field. The following shows an example of a volatile instance variable:

```
public class ExampleVolatileField
{
  private volatile int  volatileField;
}
```

You can use volatile fields if the writes do not depend on the current value. Or if you can make sure that only one thread at a time can update the field. (http://vmlens.com/articles/3_tips_volatile_fields/)

volatile is a field modifier. It makes the field itself volatile not the object it references. In case of an array you need to use java.util.concurrent.atomic.AtomicReferenceArray to access the array elements in a volatile way. See the race condition in org.springframework.util.ConcurrentReferenceHashMap (http://vmlens.com/articles/org_springframework_util_ConcurrentReferenceHashMap_is_not_thread_safe/) as an example of this error.

Even more techniques

I excluded the following more advanced techniques from this list: Atomic updates, a technique in which you call atomic instructions like compare and set provided by the CPU, java.util.concurrent.locks.ReentrantLock, a lock implementation which provides more flexibility than synchronized blocks, java.util.concurrent.locks.ReentrantReadWriteLock, a lock implementation in which reads do not block reads and java.util.concurrent.locks.StampedLock a nonreeantrant Read-Write lock with the possibility to optimistically read values.

Conclusion

The best way to achieve thread safety is to avoid shared state. For the state, you need to share you can either use message parsing together with immutable classes or the concurrent data structures together with synchronized blocks and volatile fields.