How I Test My Java Classes for Thread Safety

by Yegor Bugayenko MVB · **Apr. 03, 18 · Java Zone · Tutorial**

I touched on this problem in one of my recent webinars. Now, it's time to explain it in writing. Thread safety is an important quality of classes in languages/platforms like Java, where we frequently share objects between threads. The issues caused by lack of thread safety are very difficult to debug since they are sporadic and almost impossible to reproduce on purpose. How do you test your objects to make sure they are thread-safe? Here is how I'm doing it.

Let's say there is a simple in-memory bookshelf:

```
class Books {
  final Map<Integer, String> map =
    new ConcurrentHashMap<>();
  int add(String title) {
    final Integer next = this.map.size() + 1;
    this.map.put(next, title);
    return next;
  }
  String title(int id) {
    return this.map.get(id);
  }
}
```

First, we put a book there and the bookshelf returns its ID. Then, we can read the title of the book by its ID:

```
Books books = new Books();
String title = "Elegant Objects";
int id = books.add(title);
assert books.title(id).equals(title);
```

The class seems to be thread-safe since we are using the thread-safe `ConcurrentHashMap` instead of a more primitive and non-thread-safe `HashMap`, right? Let's try to test it:

The test passes, but it's just a one-thread test. Let's try to do the same manipulation from a few parallel threads (I'm using Hamcrest):

```
class BooksTest {
  @Test
  public void addsAndRetrieves() {
    Books books = new Books();
    int threads = 10;
    ExecutorService service =
      Executors.newFixedThreadPool(threads);
    Collection<Future<Integer>> futures =
      new ArrayList<>(threads);
    for (int t = 0; t < threads; ++t) {
      final String title = String.format("Book #%d", t);
      futures.add(service.submit(() -> books.add(title)));
    }
    Set<Integer> ids = new HashSet<>();
    for (Future<Integer> f : futures) {
      ids.add(f.get());
    }
    assertThat(ids.size(), equalTo(threads));
  }
}
```

First, I create a pool of threads via `Executors`. Then, I submit ten objects of type `Callable` via `submit()`. Each of them will add a new unique book to the bookshelf. All of them will be executed, in some unpredictable order, by some of those ten threads from the pool.

Then, I fetch the results of their executors through the list of objects of type `Future`. Finally, I calculate the amount of unique book IDs created. If the number is 10, there were no conflicts. I'm using the `Set` collection in order to make sure the list of IDs contains only unique elements.

The test passes on my laptop. However, it's not strong enough. The problem here is that it's not really testing the `Books` from multiple parallel threads. The time that passes between our calls to `submit()` is large enough to finish the execution of `books.add()`. That's why, in reality, only one thread will run at the same time. We can check that by modifying the code a bit:

```
AtomicBoolean running = new AtomicBoolean();
AtomicInteger overlaps = new AtomicInteger();
Collection<Future<Integer>> futures =
```

```
        }
        running.set(true);
        int id = books.add(title);
        running.set(false);
        return id;
      }
    )
  );
}
assertThat(overlaps.get(), greaterThan(0));
```

With this code, I'm trying to see how often threads overlap each other and do something in parallel. This never happens and `overlaps` is equal to zero. Thus, our test is not really testing anything yet. It just adds ten books to the bookshelf one by one. If I increase the number of threads to 1,000, they start to overlap sometimes. But we want them to overlap even when there's a small number of them. To solve that, we need to use `CountDownLatch`:

```
CountDownLatch latch = new CountDownLatch(1);
AtomicBoolean running = new AtomicBoolean();
AtomicInteger overlaps = new AtomicInteger();
Collection<Future<Integer>> futures =
  new ArrayList<>(threads);
for (int t = 0; t < threads; ++t) {
  final String title = String.format("Book #%d", t);
  futures.add(
    service.submit(
      () -> {
        latch.await();
        if (running.get()) {
          overlaps.incrementAndGet();
        }
        running.set(true);
        int id = books.add(title);
        running.set(false);
        return id;
      }
    )
  );
}
latch.countDown();
Set<Integer> ids = new HashSet<>();
```

And that last `assertThat()` crashes now! I'm not getting 10 book IDs, as I was before. It's 7-9, but never 10. The class, apparently, is not thread-safe!

But before we fix the class, let's make our test simpler. Let's use `RunInThreads` from Cactoos, which does exactly the same as we've done above, but under the hood:

```
class BooksTest {
  @Test
  public void addsAndRetrieves() {
    Books books = new Books();
    MatcherAssert.assertThat(
      t -> {
        String title = String.format(
          "Book #%d", t.getAndIncrement()
        );
        int id = books.add(title);
        return books.title(id).equals(title);
      },
      new RunsInThreads<>(new AtomicInteger(), 10)
    );
  }
}
```

The first argument of `assertThat()` is an instance of `Func` (a functional interface), accepting an `AtomicInteger` (the first argument of `RunsInThreads`) and returning `Boolean`. This function will be executed on 10 parallel thread, using the same latch-based approach as demonstrated above.

This `RunInThreads` seems to be compact and convenient, I'm using it in a few projects already.

By the way, in order to make `Books` thread-safe we just need to add `synchronized` to its method `add()`. Or maybe you can suggest a better solution?

P.S. I learned all this from *Java Concurrency in Practice* by Goetz et al.

Like This Article? Read More From DZone