(http://baeldung.com)

Guide to ThreadLocalRandom in Java

Last modified: February 20, 2018

by baeldung (http://www.baeldung.com/author/baeldung/)

Java (http://www.baeldung.com/category/java/)

---

1. Overview

Generating random values is a very common task. This is why Java provides the *java.util.Random* class.

However, this class doesn't perform well in a multi-threaded environment.

In a simplified way, the reason for poor performance of *Random* in a multi-threaded environment is due to contention – given that multiple threads share the same *Random* instance.

To address that limitation, **Java introduced the *java.util.concurrent.ThreadLocalRandom* class in JDK 7 – for generating random numbers in a multi-threaded environment**.

Let's see how *ThreadLocalRandom* performs and how to use it in real-world applications.

2. *ThreadLocalRandom* over *Random*

***ThreadLocalRandom* is a combination of *ThreadLocal (http://www.baeldung.com/java-threadlocal)* and *Random* classes, which is isolated to the current thread.** Thus, it achieves better performance in a multithreaded environment by simply avoiding any concurrent access to the *Random* objects.

The random number obtained by one thread is not affected by the other thread, whereas *java.util.Random* provides random numbers globally.

Also, unlike *Random, ThreadLocalRandom* doesn't support setting the seed explicitly. Instead, it overrides the *setSeed(long seed)* method inherited from *Random* to always throw an *UnsupportedOperationException* if called.

Let's now take a look at some of the ways to generate random *int, long* and *double* values.

3. Generating Random Values Using *ThreadLocalRandom*

As per the Oracle documentation, **we just need to call *ThreadLocalRandom.current()* method, and it will return the instance of *ThreadLocalRandom* for the current thread**. We can then generate random values by invoking available instance methods of the class.

Let's generate a random *int* value without any bounds:

```java
int unboundedRandomValue = ThreadLocalRandom.current().nextInt());
```

Next, let's see how we can generate a random bounded *int* value, meaning a value between a given lower and upper limit.

Here's an example of generating a random *int* value between 0 and 100:

```java
int boundedRandomValue = ThreadLocalRandom.current().nextInt(0, 100);
```

Please note, 0 is the inclusive lower limit and 100 is the exclusive upper limit.

We can generate random values for *long* and *double* by invoking *nextLong()* and *nextDouble()* methods in a similar way as shown in the examples above.

Java 8 also adds the *nextGaussian()* method to generate the next normally-distributed value with a 0.0 mean and 1.0 standard deviation from the generator's sequence.

As with the *Random* class, we can also use the *doubles(), ints()* and *longs()* methods to generate streams of random values.

4. Comparing *ThreadLocalRandom* and *Random* Using JMH

Let's see how we can generate random values in a multi-threaded environment, by using the two classes, then compare their performance using JMH.

First, let's create an example where all the threads are sharing a single instance of *Random.* Here, we're submitting the task of generating a random value using the *Random* instance to an *ExecutorService:*

```java
ExecutorService executor = Executors.newWorkStealingPool();
List<Callable<Integer>> callables = new ArrayList<>();
Random random = new Random();
for (int i = 0; i < 1000; i++) {
    callables.add(() -> {
         return random.nextInt();
    });
}
executor.invokeAll(callables);
```

Let's check the performance of the code above using JMH benchmarking:

```
# Run complete. Total time: 00:00:36
Benchmark                                           Mode Cnt Score     Error
ThreadLocalRandomBenchMarker.randomValuesUsingRandom avgt 20  771.613 ± 222.220
```

Similarly, let's now use *ThreadLocalRandom* instead of the *Random* instance, which uses one instance of *ThreadLocalRandom* for each thread in the pool:

```
ExecutorService executor = Executors.newWorkStealingPool();
List<Callable<Integer>> callables = new ArrayList<>();
for (int i = 0; i < 1000; i++) {
    callables.add(() -> {
        return ThreadLocalRandom.current().nextInt();
    });
}
executor.invokeAll(callables);
```

Here's the result of using *ThreadLocalRandom:*

```
omplete. Total time: 00:00:36
rk                                                         Mode Cnt Score     Error   Un
ocalRandomBenchMarker.randomValuesUsingThreadLocalRandom avgt 20  624.911 ± 113.268 u
```

Finally, by comparing the JMH results above for both *Random* and *ThreadLocalRandom*, we can clearly see that the average time taken to generate 1000 random values using *Random* is 772 microseconds, whereas using *ThreadLocalRandom* it's around 625 microseconds.

Thus, we can conclude that ***ThreadLocalRandom* is more efficient in a highly concurrent environment**.

To learn more about **JMH**, check out our previous article here (http://www.baeldung.com/java-microbenchmark-harness).

5. Conclusion

This article illustrated the difference between *java.util.Random* and *java.util.concurrent.ThreadLocalRandom*.

We also saw the advantage of *ThreadLocalRandom* over *Random* in a multithreaded environment, as well as performance and how we can generate random values using the class.