

Java Multithreading for Senior Engineering Interviews / ... / ThreadLocalRandom

ThreadLocalRandom

Guide to using ThreadLocalRandom with examples.

If you are interviewing, consider buying our number#1 course for <u>Java Multithreading</u> Interviews.

Overview

The class <code>java.util.concurrent.ThreadLocalRandom</code> is derived from <code>java.util.Random</code> and generates random numbers much more efficiently than <code>java.util.Random</code> in multithreaded scenarios. Interestingly, <code>Random</code> is <code>thread-safe</code> and can be used by multiple threads without malfunction, just not efficiently.

To understand why an instance of the Random class experiences overhead and contention in concurrent programs, we'll delve into the code for one of the most commonly used methods nextInt() of the Random class. The code is reproduced verbatim from the Java source code below:

```
Generates the next pseudorandom number. Subclasses should
             eral contract of {@code next} is that it returns an
 {@code 1} and {@code 32} (inclusive), then that many low-orde

    bits of the returned value will be (approximately) independently
    chosen bit values, each of which is (approximately) equally

likely to be {@code 0} or {@code 1}. The method {@code next} is
  implemented by class {@code Random} by atomically updating the seed
<pcode (seed * 0x5DEECE66DL + 0x8L) & ((1L << 48) - 1)}</pre>
 <@code (int)(seed >>> (48 - bits))}.
* This is a linear congruential pseudora
defined by D. H. Lehmer and described by Donald E. Knuth in
* <i>The Art of Computer Programming, </i> Volume 2:
* <i>Seminumerical Algorithms</i>, section 3.2.1
* Aparam bits random bits
 Greturn the next pseudorandom value from this random nu
         generator's sequence
* @since 1.1
otected int next(int bits) {
   long oldseed, nextseed;
  AtomicLong seed = this.seed;
      oldseed = seed.get();
nextseed = (oldseed * multiplier + addend) & mask;
   } while (|seed.compareAndSet(oldseed.nextseed)):
```

Examine the code above and realize the do-while loop uses the compareAndSet() method to atomically set the seed variable to a new value in its predicate. Imagine several threads invoking the next() method on a shared instance of Random, only one thread will successfully exit the loop and the rest will re-execute the loop, until all of them exit one by one. This mechanism to update the seed variable is precisely what makes the Random class inefficient for highly concurrent programs, when several threads want to generate random numbers in parallel.

The performance issues faced by Random are addressed by the ThreadLocalRandom class which is isolated in its effects to a single thread. A random number generated by one thread using ThreadLocalRandom has no bearing on random numbers generated by other threads, unlike an instance of Random that generates random numbers globally. Furthermore, ThreadLocalRandom differs from Random in that the former doesn't allow setting a seed value unlike the latter. In summary, ThreadLocalRandom is more performant than Random as it eliminates concurrent access to shared state.

The astute reader would question if maintaining a distinct Random object per thread is equivalent to using the ThreadLocalRandom class? The ThreadLocalRandom class is singleton and uses state held by the ThreadLocalRandom to use for generating random numbers and related book-keeping.

```
class Thread implements Runnable {

// The following three initially uninitialized fields are exclusively

// managed by class java.util.concurrent.ThreadlocalRandom. These

// fields are used to build the high-performance PRNGs in the

// concurrent code, and we can not risk accidental false sharing.

// Hence, the fields are isolated with @Contended.

/** The current seed for a ThreadlocalRandom */
@jdk.internal.vm.annotation.Contended("tlr")

long threadlocalRandomSeed;

/** Probe hash value; nonzero if threadlocalRandomSeed initialized */
```

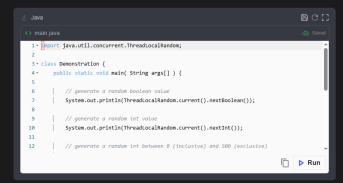
```
@jdk.internal.vm.annotation.Contended("tlr")
int threadLocalRandomProbe;

/** Secondary seed isolated from public ThreadLocalRandom sequence */
@jdk.internal.vm.annotation.Contended("tlr")
int threadLocalRandomSecondarySeed;
}
```

Each thread stores the seed itself in the field thread-localRandomSeed. As the seed is not shared among threads anymore, performance improves.

Usage

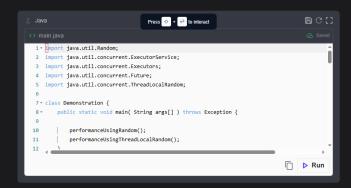
The idiomatic usage for generating random numbers takes the form of ThreadLocalRandom.current().nextInt() and is demonstrated in the widget below:



Difference between Random and ThreadLocalRandom

Consider the scenario of a single instance of Random class being shared among 5 threads. Our program has each thread generate a random integer ten thousand times. We repeat the same test using the ThreadLocalRandom class and time the execution for both scenarios in milliseconds. As expected, the test using the ThreadLocalRandom class performs better than the one using the Random class instance. Though our test is crude but it still gives us a sense of difference in performance of the two classes.

Some runs of the program may exhibit a longer execution time for ThreadLocalRandom class than the Random class. This may occur due to the widget code executing in a shared cloud environment beyond our control. However, if the reader executed the code with all aspects as constants ThreadLocalRandom would outperform Random when generating random numbers in our text code.



```
← Back lesson

✓ Mark As Completed

Next →

CountDownLatch
```

