(http://baeldung.com)

Custom Thread Pools In Java 8 Parallel Streams

Last modified: August 27, 2017

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

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

1. Overview

Java 8 introduced the concept of S*treams* as an efficient way of carrying out bulk operations on data. And parallel *Streams* can be obtained in environments that support concurrency.

These streams can come with improved performance – at the cost of multi-threading overhead.

In this quick tutorial, we'll look at **one of the biggest limitations of *Stream* API** and see how to make a parallel stream work with a custom *ThreadPool* instance.

2. Parallel *Stream*

Let's start with a simple example – calling the *parallelStream* method on any of the *Collection* types – which will return a possibly parallel *Stream*:

```
@Test
public void givenList_whenCallingParallelStream_shouldBeParallelStream(){
    List<Long> aList = new ArrayList<>();
    Stream<Long> parallelStream = aList.parallelStream();

    assertTrue(parallelStream.isParallel());
}
```

The default processing that occurs in such a *Stream* uses the *ForkJoinPool.commonPool(),* **a *Thread Pool* shared by the entire application.**

3. Custom *Thread Pool*

We can actually pass a custom *ThreadPool* when processing the *stream*.

The following example lets have a parallel *Stream* use a custom *Thread Pool* to calculate the sum of long values from 1 to 1,000,000, inclusive:

```
                                nmedInParallel_shouldBeEqualToExpectedTotal
                                utionException {




                                angeClosed(firstNum, lastNum).boxed()


                                new ForkJoinPool(4);
    long actualTotal = customThreadPool.submit(
      () -> aList.parallelStream().reduce(0L, Long::sum)).get();

    assertEquals((lastNum + firstNum) * lastNum / 2, actualTotal);
}
```

We used the *ForkJoinPool* constructor with a parallelism level of 4. Some experimentation is required to determine the optimal value for different environments, but a good rule of thumb is simply choosing the number based on how many cores your CPU has.

Next, we processed the content of the parallel *Stream*, summing them up in the *reduce* call.

This simple example may not demonstrate the full usefulness of using a custom *Thread Pool*, but the benefits become obvious in situations where we do not want to tie-up the common *Thread Pool* with long-running tasks (e.g. processing data from a network source), or the common *Thread Pool* is being used by other components within the application.

4. Conclusion

We have briefly looked at how to run a parallel *Stream* using a custom *Thread Pool*. In the right environment and with the proper use of the parallelism level, performance gains can be had in certain situations.

The complete code samples referenced in this article can be found over on Github (https://github.com/eugenp/tutorials/tree/master/core-java-concurrency).