java.util.concurrent Package

Last Updated : 14 Mar, 2023

Java Concurrency package covers **concurrency, multithreading**, and **parallelism** on the Java platform. Concurrency is the ability to run several or multi programs or applications in parallel. The **backbone** of Java concurrency is threads (a lightweight process, which has its own files and stacks and can access the shared data from other threads in the same process). The throughput and the interactivity of the program can be improved by performing time-consuming tasks asynchronously or in parallel. **Java 5** added a new package to the java platform ⇀ java.util.concurrent package. This package has a set of classes and interfaces that helps in developing concurrent applications (multithreading) in java. Before this package, one needs to make the utility classes of their need on their own.

Main Components/Utilities of Concurrent Package

1. Executor
2. ExecutorService
3. ScheduledExecutorService
4. Future
5. CountDownLatch
6. CyclicBarrier
7. Semaphore
8. ThreadFactory
9. BlockingQueue
10. DelayQueue
11. Lock
12. Phaser

Now let us discuss some of the most useful utilities from this package that are as follows:

A. Executor

Executor is a set of interfaces that represents an object whose implementation executes tasks. It depends on the implementation whether the task should be run on a new thread or on a current thread. Hence, we can decouple the task execution flow from the actual task execution mechanism, using this interface. **The executor does not require the task execution to be asynchronous.** The simplest of all is the executable interface.

```
public interface Executor {

    void execute( Runnable command );

}
```

In order to create an **executer instance**, we need to create an invoker.

```
public class Invoker implements Executor {
```

```
  @Override
  public void execute(Runnable r) {
     r.run();
  }
}
```

Now, for the **execution of the task**, we can use this invoker.

```
public void execute() {
  Executor exe = new Invoker();
  exe.execute( () -> {
     // task to be performed
  });
}
```

If the executor can't accept the task to be executed, it will throw **RejectedExecutionException**.

B.ExecutorService

ExecutorService is an interface and only forces the underlying implementation to implement **execute()** method. It extends the Executor interface and adds a series of methods that execute threads that return a value. The methods to shut the thread pool as well as the ability to implement for the result of the execution of the task. We need to create Runnable target to use the ExecutorService.

```
public class Task implements Runnable {
  @Override
  public void run() {

     // task details
  }
}
```

Now, we can create an object/instance of this class and assign the task. We need to specify the thread-pool size while creating an instance.

```
// 20 is the thread pool size
ExecutorService exec = Executors.newFixedThreadPool(20);
```

For the creation of a single-threaded ExecutorService instance, we can use **newSingleThreadExecuter(ThreadFactory threadfactory)** for creating the instance. After the executor is created, we can submit the task.

```
public void execute() {
  executor.submit(new Task());
```

```
}
```

Also, we can create a Runnable instance for the **submission of tasks.**

```
executor.submit(() -> {
  new Task();
});
```

Two out-of-the-box termination methods are listed as follows:

1. **shutdown():** It waits till all the submitted tasks execution gets finished.
2. **shutdownNow():** It immediately terminates all the executing/pending tasks.

There is one more method that is **awaitTermination()** which forcefully blocks until all tasks have completed execution after a shutdown event-triggered or execution-timeout occurred, or the execution thread itself is interrupted.

```
try {
  exec.awaitTermination( 50l, TimeUnit.NANOSECONDS );
} catch (InterruptedException e) {
  e.printStackTrace();
}
```

C. ScheduledExecutorService

It is similar to ExecutorService. The difference is that this interface can perform tasks periodically. Both Runnable and Callable function is used to define the task.

```
public void execute() {
  ScheduledExecutorService execServ
    = Executors.newSingleThreadScheduledExecutor();

  Future<String> future = execServ.schedule(() -> {
     // ..
     return "Hello world";
  }, 1, TimeUnit.SECONDS);

  ScheduledFuture<?> scheduledFuture = execServ.schedule(() -> {
     // ..
  }, 1, TimeUnit.SECONDS);

  executorService.shutdown();
}
```

ScheduledExecutorService can also define a task after some fixed delay.

```
executorService.scheduleAtFixedRate(() -> {
  // ..
}, 1, 20, TimeUnit.SECONDS);


executorService.scheduleWithFixedDelay(() -> {
  // ..
}, 1, 20, TimeUnit.SECONDS);
```

Here,

- **scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit):** This method creates and executes a periodic action that is first invoked after the initial delay and subsequently with the given period until the service instance shutdowns.
- **scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit):** This method creates and executes a periodic action that is invoked firstly after the provided initial delay and repeatedly with the given delay between the termination of the executing one and the invocation f the next one.

D. Future

It represents the result of an **asynchronous operation.** The methods in it check if the asynchronous operation is completed or not, get the completed result, etc.
The **cancel(boolean isInterruptRunning)** API cancels the operation and releases the executing thread. On the value of isInterruptRunning value being true, the thread executing the task will be terminated instantly. Otherwise, all the in-progress tasks get completed.
Code snippet creates an **instance of Future.**

```
public void invoke() {
  ExecutorService executorService = Executors.newFixedThreadPool(20);


  Future<String> future = executorService.submit(() -> {
    // ...
    Thread.sleep(10000l);
    return "Hello";
  });
}
```

The code to check if the **result of the future** is ready or not and fetches the data when the computation is done.

```
if (future.isDone() && !future.isCancelled()) {
```

```
  try {
     str = future.get();
  } catch (InterruptedException | ExecutionException e) {
     e.printStackTrace();
  }
}
```

Timeout specification for a given operation. If the time taken is more than this time, then **TimeoutException** is thrown.

```
try {
  future.get(20, TimeUnit.SECONDS);
} catch (InterruptedException | ExecutionException | TimeoutException e) {
  e.printStackTrace();
}
```

E. CountDownLatch

It is a utility class that blocks a **set of threads** until some operations get completed. A CountDownLatch is initialized with a counter(which is of Integer type). This counter decrements as the execution of the dependent threads get completed. But once the counter decrements to zero, other threads get released.

F. CyclicBarrier

CyclicBarrier is almost the same as CountDownLatch except that we can reuse it. It allows multiple threads to wait for each other using **await()** before invoking the final task and this feature is not in CountDownLatch.

We are required to create an instance of Runnable Task to **initiate the barrier condition.**

```
public class Task implements Runnable {

  private CyclicBarrier barrier;

  public Task(CyclicBarrier barrier) {
     this.barrier = barrier;
  }

  @Override
  public void run() {
     try {
        LOG.info(Thread.currentThread().getName() +
```

```
            " is waiting");
            barrier.await();
            LOG.info(Thread.currentThread().getName() +
              " is released");
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }


}
```

Now, invoking a few threads to **race the barrier condition:**

```
public void start() {


    CyclicBarrier cyclicBarrier = new CyclicBarrier(3, () -> {
        // ..
        LOG.info("All previous tasks completed");
    });


    Thread t11 = new Thread(new Task(cyclicBarrier), "T11");
    Thread t12 = new Thread(new Task(cyclicBarrier), "T12");
    Thread t13 = new Thread(new Task(cyclicBarrier), "T13");


    if (!cyclicBarrier.isBroken()) {
        t11.start();
        t12.start();
        t13.start();
    }
}
```

In the above code, the **isBroken()** method checks if any of the threads got interrupted during the execution time.

G. Semaphore

It is used for **blocking** thread-level access to some part of the logical or physical resource. Semaphore contains a set of permits. Wherever the thread tries to enter the code part of a critical section, semaphore gives the permission whether the

permit is available or not, which means the critical section is available or not. If the permit is not available, then the thread cannot enter the critical section.
It is basically a variable named counter which maintains the count of entering and leaving threads from the critical section. When the executing thread releases the critical section, the counter increases.

Below code is used for the implementation of Semaphore:

```
static Semaphore semaphore = new Semaphore(20);


public void execute() throws InterruptedException {


  LOG.info("Available : " + semaphore.availablePermits());
  LOG.info("No. of threads waiting to acquire: " +
    semaphore.getQueueLength());


  if (semaphore.tryAcquire()) {
     try {
        //
     }
     finally {
        semaphore.release();
     }
  }


}
```

Semaphores can be used to implement a **Mutex-like** data structure.

H. ThreadFactory

It acts as a thread pool which **creates a new thread** on demand. ThreadFactory can be defined as:

```
public class GFGThreadFactory implements ThreadFactory {
  private int threadId;
  private String name;


  public GFGThreadFactory(String name) {
     threadId = 1;
     this.name = name;
```

```
    }


    @Override
    public Thread newThread(Runnable r) {
        Thread t = new Thread(r, name + "-Thread_" + threadId);
        LOG.info("created new thread with id : " + threadId +
            " and name : " + t.getName());
        threadId++;
        return t;
    }
}
```

I. BlockingQueue

BlockingQueue interface supports flow control (in addition to queue) by introducing blocking if either BlockingQueue is full or empty. A thread trying to enqueue an element in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more elements or clearing the queue completely. Similarly, it blocks a thread trying to delete from an empty queue until some other threads insert an item. BlockingQueue does not accept a null value. If we try to enqueue the null item, then it throws **NullPointerException**.

J: DelayQueue

DelayQueue is a specialized Priority Queue that orders elements based on their delay time. It means that only those elements can be taken from the queue whose time has expired. DelayQueue head contains the element that has expired in the least time. If no delay has expired, then there is no head and poll will return null. DelayQueue accepts only those elements that belong to a class of type Delayed. DelayQueue implements the getDelay() method to return the remaining delay time.

K: Lock

It is a utility for **blocking other threads** from accessing a certain segment of code. The difference between **Lock** and a **Synchronized block** is that we have Lock APIs **lock() and unlock()** operation in separate methods whereas a Synchronized block is fully contained in methods.

L: Phaser

It is more flexible than CountDownLatch and CyclicBarrier. Phaser is used to act as a **reusable barrier** on which the dynamic number of threads needs to wait before the execution continues. Multiple phases of execution can be coordinated by reusing the instance of a Phaser for each program phase.