Java Thread Pool Example using Executors and ThreadPoolExecutor

A thread pool manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed. A thread pool manages the collection of Runnable threads and worker threads execute Runnable from the queue.

java.util.concurrent.Executors provide implementation of java.util.concurrent.Executor interface to create the thread pool in java.

Let’s write a simple program to explain it’s working.

First we need to have a Runnable class.

```
package com.journaldev.threadpool;

public class WorkerThread implements Runnable {

    private String command;

    public WorkerThread(String s){
        this.command=s;
    }

    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+" Start. Command = "+c
ommand);
        processCommand();
        System.out.println(Thread.currentThread().getName()+" End.");
    }

    private void processCommand() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public String toString(){
        return this.command;
    }
}
```

Here is the test program where we are creating fixed thread pool from Executors framework.

```
package com.journaldev.threadpool;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class SimpleThreadPool {

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.execute(worker);
          }
        executor.shutdown();
        while (!executor.isTerminated()) {
        }
        System.out.println("Finished all threads");
    }

}
```

In above program, we are creating fixed size thread pool of 5 worker threads. Then we are submitting 10 jobs to this pool, since the pool size is 5, it will start working on 5 jobs and other jobs will be in wait state, as soon as one of the job is finished, another job from the wait queue will be picked up by worker thread and get’s executed.

Here is the output of the above program.

```
pool-1-thread-2 Start. Command = 1
pool-1-thread-4 Start. Command = 3
pool-1-thread-1 Start. Command = 0
pool-1-thread-3 Start. Command = 2
pool-1-thread-5 Start. Command = 4
pool-1-thread-4 End.
pool-1-thread-5 End.
pool-1-thread-1 End.
pool-1-thread-3 End.
pool-1-thread-3 Start. Command = 8
pool-1-thread-2 End.
pool-1-thread-2 Start. Command = 9
pool-1-thread-1 Start. Command = 7
pool-1-thread-5 Start. Command = 6
pool-1-thread-4 Start. Command = 5
pool-1-thread-2 End.
pool-1-thread-4 End.
pool-1-thread-3 End.
pool-1-thread-5 End.
pool-1-thread-1 End.
Finished all threads
```

The output confirms that there are five threads in the pool named from “pool-1-thread-1? to “pool-1-thread-5? and they are responsible to execute the submitted tasks to the pool.

Executors class provide simple implementation of **ExecutorService** using **ThreadPoolExecutor** but ThreadPoolExecutor provides much more feature than that. We can specify the number of threads that will be alive when we create ThreadPoolExecutor instance and we can limit the size of thread pool and create our own **RejectedExecutionHandler** implementation to handle the jobs that can’t fit in the worker queue.

Here is our custom implementation of RejectedExecutionHandler interface.

```
package com.journaldev.threadpool;

import java.util.concurrent.RejectedExecutionHandler;
import java.util.concurrent.ThreadPoolExecutor;

public class RejectedExecutionHandlerImpl implements RejectedExecutionHandler {

    @Override
    public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
        System.out.println(r.toString() + " is rejected");
    }

}
```

ThreadPoolExecutor provides several methods using which we can find out the current state of executor, pool size, active thread count and task count.

So I have a monitor thread that will print the executor information at certain time interval.

```
package com.journaldev.threadpool;

import java.util.concurrent.ThreadPoolExecutor;

public class MyMonitorThread implements Runnable
{
    private ThreadPoolExecutor executor;

    private int seconds;

    private boolean run=true;

    public MyMonitorThread(ThreadPoolExecutor executor, int delay)
    {
        this.executor = executor;
        this.seconds=delay;
    }

    public void shutdown(){
        this.run=false;
    }

    @Override
    public void run()
    {
        while(run){
                System.out.println(
                    String.format("[monitor] [%d/%d] Active: %d, Completed: %d, Ta
sk: %d, isShutdown: %s, isTerminated: %s",
                        this.executor.getPoolSize(),
                        this.executor.getCorePoolSize(),
                        this.executor.getActiveCount(),
                        this.executor.getCompletedTaskCount(),
                        this.executor.getTaskCount(),
                        this.executor.isShutdown(),
                        this.executor.isTerminated()));
                try {
                    Thread.sleep(seconds*1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
        }

    }
}
```

Here is the thread pool implementation example using ThreadPoolExecutor.

```
package com.journaldev.threadpool;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class WorkerPool {

    public static void main(String args[]) throws InterruptedException{
        //RejectedExecutionHandler implementation
        RejectedExecutionHandlerImpl rejectionHandler = new RejectedExecutionHandl
erImpl();
        //Get the ThreadFactory implementation to use
        ThreadFactory threadFactory = Executors.defaultThreadFactory();
        //creating the ThreadPoolExecutor
        ThreadPoolExecutor executorPool = new ThreadPoolExecutor(2, 4, 10, TimeUni
t.SECONDS, new ArrayBlockingQueue<Runnable>(2), threadFactory, rejectionHandler);
        //start the monitoring thread
        MyMonitorThread monitor = new MyMonitorThread(executorPool, 3);
        Thread monitorThread = new Thread(monitor);
        monitorThread.start();
        //submit work to the thread pool
        for(int i=0; i<10; i++){
            executorPool.execute(new WorkerThread("cmd"+i));
        }

        Thread.sleep(30000);
        //shut down the pool
        executorPool.shutdown();
        //shut down the monitor thread
        Thread.sleep(5000);
        monitor.shutdown();

    }
}
```

Notice that while initializing the ThreadPoolExecutor, we are keeping initial pool size as 2, maximum pool size to 4 and work queue size as 2. So if there are 4 running tasks and more tasks are submitted, the work queue will hold only 2 of them and rest of them will be handled by RejectedExecutionHandlerImpl.

Here is the output of above program that confirms above statement.

```
pool-1-thread-1 Start. Command = cmd0
pool-1-thread-4 Start. Command = cmd5
cmd6 is rejected
pool-1-thread-3 Start. Command = cmd4
pool-1-thread-2 Start. Command = cmd1
cmd7 is rejected
cmd8 is rejected
cmd9 is rejected
[monitor] [0/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated:
 false
[monitor] [4/2] Active: 4, Completed: 0, Task: 6, isShutdown: false, isTerminated:
 false
pool-1-thread-4 End.
pool-1-thread-1 End.
pool-1-thread-2 End.
pool-1-thread-3 End.
pool-1-thread-1 Start. Command = cmd3
pool-1-thread-4 Start. Command = cmd2
[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated:
 false
[monitor] [4/2] Active: 2, Completed: 4, Task: 6, isShutdown: false, isTerminated:
 false
pool-1-thread-1 End.
pool-1-thread-4 End.
[monitor] [4/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated:
 false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated:
 false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated:
 false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated:
 false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated:
 false
[monitor] [2/2] Active: 0, Completed: 6, Task: 6, isShutdown: false, isTerminated:
 false
[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated:
true
[monitor] [0/2] Active: 0, Completed: 6, Task: 6, isShutdown: true, isTerminated:
true
```

Notice the change in active, completed and total completed task count of the executor. We can invoke shutdown() method to finish execution of all the submitted tasks and terminate the thread pool.

Reference: Java Thread Pool Example using Executors and ThreadPoolExecutor from our JCG partner Pankaj Kumar at the Developer Recipes blog.