Applying Thread Pools

Pieter van den Hombergh

Fontys Hogeschool voor Techniek en Logistiek

March 18, 2018
Couplings between tasks and execution policies
   Starvation
   Deadlock

Configuration and extension
   Configuring ThreadPoolExecutor
   Saturarion policies
   Pool customization after construction
   Extending ThreadPoolExecutor

Parallelizing recursive algorithms
   Simple iterations
   Recursion, separating path to work and work
   Collection time

Summary
   Chap 8 summary
Tasks and execution policies (We lied)

- In chapter 6 we said that the Executor framework decouples task submission from execution.
- This a bit of an overstatement. Not all tasks can be made compatible with all executions policies.
- So there is some kind of an explicit coupling. We lied 😊.
Tasks that require specific policies

Dependent Tasks If a task depends on the results of another, your policy must match that.

Task that exploit thread confinement If your task relies on some kind of locality like `ThreadLocal` if (some of) the member types used are not thread safe.

Response time sensitive tasks If you have e.g. a GUI application, the responsiveness requirements may dictate a policy and maybe a priority too.

Task use `ThreadLocal` Executors are allowed to reuse existing threads (those in a pool for instance). This may be in the way if your application depends on `ThreadLocal` as a construction element.
Thread starvation

If tasks depending (blocking waits) on other tasks execute in a threadpool, they can deadlock. For instance (book has more examples) all threads are busy or occupied and are waiting for a task still on the work queue.

Beware of the Big Famine

- Whenever you submit tasks to an executor that depend on each other results, be aware of the possibility of thread starvation deadlock.
- Document constraints on pool size or configuration.
ThreadDeadlock

```java
public class ThreadDeadlock {
    ExecutorService exec = Executors.newSingleThreadExecutor();

    public class RenderPageTask implements Callable<String> {
        public String call() throws Exception {
            Future<String> header, footer;
            header = exec.submit(new LoadFileTask("header.html"));
            footer = exec.submit(new LoadFileTask("footer.html"));
            String page = renderBody();
            // Will deadlock — task waiting for result of subtask
            return header.get() + page + footer.get();
        }
    }
}
```

⚠️ It is very easy to force deadlocks, but they are not always that obvious.
Size matters

The ideal size of a thread pool depends on the type of tasks and the type of the deployment system. Threadpool sizes should rarely be hard-coded. Sizing thread pools is no rocket science, it is more like avoiding the extremes too small and too big.

\[ N_{cpu} = \text{number of CPUs} \]  
\[ U_{cpu} = \text{target CPU utilization, } 0 \leq U_{cpu} \leq 1 \]  
\[ \frac{W}{C} = \text{ratio of wait time to compute time} \]  
\[ N_{threads} = N_{cpu} \times U_{cpu} \times \left( 1 + \frac{W}{C} \right) \]
Let’s have a look on what we have got:

```java
class NrCpus {
    public static void main(String[] args) {
        int ncpus = Runtime.getRuntime().availableProcessors();
        System.out.println("number_of_cpus="+ncpus);
    }
}
```
Configuration parameters

The parameters to the `ThreadPoolExecutor` are:

- `int` `corePoolSize`
- `int` `maximumPoolSize`
- `long` `keepAliveTime (in TimeUnit)`
- `BlockingQueue<Runnable>` `workQueue`
- `ThreadFactory` `threadFactory`
- `RejectExecutionHandler` `handler`
Configuration parameters

`newFixedThreadPool` uses these to set `corePoolSize` and `maximumPoolSize` to the same value. This may have the effect of an indefinite timeout on task start.

`newCachedThreadPool` sets the maximum pool size to virtually infinite and a lifetime of one minute, so the pool will expand and contract on demand.

Note that various parameters can be set after construction of the `ThreadPoolExecutor`. 
What to do with excess work?

If the pool size is bound, you will need a non trivial (size > 1) queue. The gain is that the memory footprint of a queue-element is certainly less then that of a thread. However, we are still using resources, so some kind of limit to that queue is in order.

There are three types of `BlockingQueue` that you can choose from to manage the work:

- a **unbounded** queue, which will still eat all your memory, given a chance,
- a **bounded** queue which must be accompagnied by some saturation policy and
- a **synchronous** queue, which does not store anything but is use for a rendez-vous type of handoff.
Managing work

Using a bounded queue is more stable, as the queue will not eat all the memory, but you still need some policy to handle saturation. One is to simply drop new requests on the floor (or your tie), as long as the queue is full.

With a bounded queue, `queue` size and `pool` size must be tuned interdependently. For instance a large queue and a small pool can help reduce memory usage, cpu usage, context switching, all at the cost of a potential constraint on throughput.
Are you being served?

For very large (or unbound) thread pools you need no *real* queue, but you can simply hand of the task (using a `SynchronousQueue`) to an already waiting thread.

The Executor has always a worker thread up its sleeve\(^a\), waiting for new work. This direct hand off is more efficient, queueing wise, than a real queue.

⚠️ Synchronous queue implies that you are willing to have an unbound thread pool or that you accept rejecting excess tasks.

\(^a\)unless out of memory or better: reaches the *large* bound
Queue choice

- **newCachedThreadPool** factory is a good default choice for an executor, as it has better queueing performance than the fixed thread pool, certainly with help of the *nonblocking* variant of **SynchronousQueue** in java6.

- Choose a fixed size if you need to limit the number of concurrent tasks for resource reasons, such as in a webserver in which you want to prevent overload vulnerability.
When work piles up.

Bounding something makes it have a limit: stop, I am full.
Saying no can be done in various ways:

- **AbortPolicy**, the default,
- **CallerRunsPolicy**, also known as *do it yourselves*,
- **DiscardPolicy**, that is dropping the workorder on the floor and
- **DiscardOldest** throws away the oldest order in the queue, hoping no-one is interested anymore.
Why not do it your selves

The CallerRunsPolicy says that the calling thread will execute the task itselfs, that is: do your own work.

This will keep the caller busy for some time and will prevent him from submitting new tasks in the meanwhile.

Eventually the overload is gradually pushed outward, for instance into the TCP stack and eventually the client.

This enables more gracefull degradation under load.
If blocking is needed

Listing 8.4 in the book presents a Bounded executor using a semaphore, might you need one.

```java
public void submitTask(final Runnable command)
    throws InterruptedException {
    semaphore.acquire();
    try {
        exec.execute(new Runnable() {
            public void run() {
                try {
                    command.run();
                } finally {
                    semaphore.release();
                }
            }
        });
    } catch (RejectedExecutionException e) {
        semaphore.release();
    }
}
```

QUIZ: What pattern is used in `submitTask` in this listing?
Thread factories

Being able to specify the way you want your threads fabricated gives opportunities like enabling naming, logging, maintaining statistics and specify the UncaughtExceptionHandler.

Also specialities as security policies can be handed over in an elegant way, using a custom Thread factory.
tuning on the road

Most ThreadPoolExecutor options can be modified after construction. This allows tuning and tweaking depended on the load for instance, without restarting the application.

You can find a reason why you want to forbid tuning while in use.

Sometimes this is not what you (or the framework) wants. For that, you can use the `unconfigurableExecutorService` wrapper factory method.
We have seen *Hook* before as a method to open up a class for extension. (Quiz anyone?)

ThreadPoolExecutor provides three hook-methods.

- **beforeExecute()** is called by the *worker* thread before it starts the task.
- **afterExecute()** is called by the *worker* thread after completing a task, either normally by returning from run or by an exception.
- **terminated()** is called after all is done, that is all tasks are done and all workers have shut down.
Hook example Executor listing 8.9

```java
private final ThreadLocal<Long> startTime = new ThreadLocal<Long>();
private final Logger log = Logger.getLogger("TimingThreadPool");
private final AtomicLong numTasks = new AtomicLong();
private final AtomicLong totalTime = new AtomicLong();

protected void beforeExecute(Thread t, Runnable r) {
    super.beforeExecute(t, r);
    log.fine(String.format("Thread%s:start%s", t, r));
    startTime.set(System.nanoTime());
}

protected void afterExecute(Runnable r, Throwable t) {
    try {
        long endTime = System.nanoTime();
        long taskTime = endTime - startTime.get();
        numTasks.incrementAndGet();
        totalTime.addAndGet(taskTime);
        log.fine(String.format("Thread%s:end%s,time=%dns", t, r, taskTime));
    } finally {
        super.afterExecute(r, t);
    }
}

protected void terminated() {
    try {
        log.info(String.format("Terminated:avg time=%dns",
            totalTime.get() / numTasks.get()));
    } finally {
        super.terminated();
    }
}
```
Simple iterations

Transforming sequential into parallel

```java
void processSequential(List<Element> elements) {
    for (Element e : elements)
        process(e);
}

void processInParallel(Executor exec, List<Element> elements) {
    for (final Element e : elements)
        exec.execute(new Runnable() {
            public void run() {
                process(e);
            }
        });
}
```

When to parallelize

If each iteration is independent of the others, the work in an iteration can be done in parallel to the others. But mind the economics of managing tasks versus work per iteration.
Applying parallelization to recursion

If the tree traversal (the tree being the potentially shared data structure) can be separated from the work per iteration step, here you can use parallelization too.

```java
public <T> void sequentialRecursive(List<Node<T>> nodes, Collection<T> results) {
    for (Node<T> n : nodes) {
        results.add(n.compute());
        sequentialRecursive(n.getChildren(), results);
    }
}

public <T> void parallelRecursive(final Executor exec, List<Node<T>> nodes, final Collection<T> results) {
    for (final Node<T> n : nodes) {
        exec.execute(new Runnable() {
            public void run() {
                results.add(n.compute());
            }
        });
        parallelRecursive(exec, n.getChildren(), results);
    }
}
```
Collecting the parallel results

You must of course harvest the results. Like so:

```java
public <T> Collection<T> getParallelResults(List<Node<T>> nodes)
    throws InterruptedException {
    ExecutorService exec = Executors.newCachedThreadPool();
    Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();
    parallelRecursive(exec, nodes, resultQueue);
    exec.shutdown();
    exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
    return resultQueue;
}
```

Studying the puzzleSolver (section 8.5.1) is left as an exercise.
Summary

Not only is the executor framework powerful and flexible, it can be tuned on the aspects of

- creation and teardown policies,
- queue handling
- excess task handling

but it also provides hooks for extension. Think it as an example of the open closed principle.

Note however, that you must understand the effects the various tuning parameters and task requirements have on each other.

Also note that puzzles are fun things.