Using a ThreadPoolExecutor to Parallelize Independent Single-Threaded Tasks

javacodegeeks.com Eleftheria Drosopoulou December 20th, 2011 view original

The *task execution framework*, introduced in Java SE 5.0, is a giant leap forward to simplify the design and the development of multi threaded applications. The framework provides facilities to manage the concept of *task*, to manage thread life cycles and their execution policy.
In this blog post we'll describe the power, the flexibility and the simplicity of this framework showing off a simple use case.

The Basics

The executor framework introduces an interface to manage task execution: Executor. Executor is the interface you use to *submit* tasks, represented as Runnable instances. This interface also *isolates a task submission from a task execution*: executors with different execution policies all publish the same submission interface: should you change your execution policy, your submission logic wouldn't be affected by the change.

If you want to submit a Runnable instance for execution, it's as simple as:

```
Executor exec = …;
exec.execute(runnable);
```

Thread Pools

As outlined in the previous section, how the executor is going to execute your runnable isn't specified by

the Executor contract: it depends on the specific type of executor you're using. The framework provides some different types of executors, each one with a specific execution policy tailored for different use cases.

The most common type of executors you'll be dealing with are *thread pool executors*., which are instances of the ThreadPoolExecutor class (and its subclasses). Thread pool executors manage a *thread pool*, that is the pool of worker threads that's going to execute the tasks, and a *work queue*.

You surely have seen the concept of pool in other technologies. The primary advantage of using a pool is reducing the overhead of *resources creation*, reusing structures (in this case, threads) that have been *released* after use. Another implicit advantage of using a pool is the capability of *sizing your resource usage*: you can tune the thread pool sizes to achieve the load you desire, without jeopardizing system resources.

The framework provides a factory class for thread pools called Executors. Using this factory you'll be able to create thread pools of different characteristics. Often, the underlying implementation is often the same (ThreadPoolExecutor) but the factory class helps you quickly configure a thread pool without using its more complex constructor. The factory methods are:

- newFixedThreadPool: this method returns a thread pool whose maximum size is *fixed*. It will create new threads as needed up to the maximum configured size. When the number of threads hits the maximum, the thread pool will maintain the size constant.
- newCachedThreadPool: this method returns an *unbounded* thread pool, that is a thread pool without a maximum size. However, this kind of thread pool will tear down unused thread when the load reduces.
- newSingleThreadedExecutor: this method returns an executor that guarantees that tasks will be executed in a single thread.
- newScheduledThreadPool: this method returns a fixed size thread pool that supports delayed and timed task execution.

This is just the beginning. Executors also provide other facilities that are out of scope in this tutorial and that I strongly encourage you to study about:

- Life cycle management methods, declared by the ExecutorService interface (such as shutdown() and awaitTermination()).

- *Completion services* to poll for a task status and retrieve its return value, if applicable.

The ExecutorService interface is particularly important since it provides a way to *shutdown* a thread pool, which is something you almost surely want to be able to do *cleanly*. Fortunately, the ExecutorService interface is pretty simple and self-explanatory and I recommend you study its JavaDoc thoroughly.

Basically, you send a shutdown() message to an *ExecutorService*, after which it won't accept new submitted tasks, but will continue processing the already enqueued jobs. You can pool for an executor service's termination status with isTerminated(), or wait until termination using the awaitTermination(…) method. The *awaitTermination* method won't wait forever, though: you'll have to pass the maximum wait timeout as a parameter.

Warning: a source of errors and confusion is a understanding why a JVM process never exits. If you don't shutdown your executor services, thus tearing down the underlying threads, the JVM will never exit: *a JVM exits when its last non-daemon thread exits.*

Configuring a ThreadPoolExecutor

If you decide to create a ThreadPoolExecutor manually instead of using the Executors factory class, you will need to create and configure one using one of its constructors. The most extensive constructor of this class is:

```
public ThreadPoolExecutor(
int corePoolSize,
int maxPoolSize,
long keepAlive,
TimeUnit unit,
BlockingQueue<Runnable> workQueue,
RejectedExecutionHandler handler);
```

As you can see, you can configure:

- The core pool size (the size the thread pool will try to stick with).
- The maximum pool size.
- The keep alive time, which is a time after which an idle thread is eligible for being torn down.
- The work queue to hold tasks awaiting execution.
- The policy to apply when a task submission is rejected.

Limiting the Number of Queued Tasks

Limiting the number of concurrent tasks being executing, sizing your thread pool, represents a huge benefit for your application and its execution environment in terms of predictability and stability: an unbounded thread creation will eventually exhaust the runtime resources and your application might experience as a consequence, serious performance problems that may lead even to application instability.

That's a solution to just one part of the problem: you're capping the number of tasks being executed but aren't capping the number of jobs that can be submitted and enqueued for later execution. The application will experience resource shortage later, but it will eventually experience it if the submission rate consistently outgrows the execution rate.

The solution to this problem is:

- Providing a blocking queue to the executor to hold the awaiting tasks. In the case the queue fills up, the submitted task will be "rejected".
- The RejectedExecutionHandler is invoked when a task submission is rejected, and that's why the verb rejected was quoted in the previous item. You can implement you're own rejection policy or use one of the built-in policies provided by the framework.

The default rejection policies has the executor throw a RejectedExecutionException. However, other built-in policies let you:

- Discard a job silently.
- Discard the oldest job and try to resubmit the last one.
- Execute the rejected task on the caller's thread.

When and why would one use such a thread pool configuration? Let's see an example.

An Example: Parallelizing Independent Single-Threaded Tasks

Recently, I was called to solve a problem with an old job my client was running since a long time ago. Basically, the job is made up of a component that awaits for file system events on a set of directory hierarchies. Whenever an event is fired, a file must be processed. The file processing is performed by a proprietary *single threaded* process. Truth be said, by its own nature, even if I could, I don't if I could parallelize it. The arrival rate of events is very high throughout part of the day and there's no need to process file in real time, they just to get processed before the next day.

The current implementation was a mix and match of technologies, including a UNIX shell script that was responsible for scanning huge directory hierarchies to detect where changes were applied. When that implementation was put in place, the number of cores in the execution environment were two, as much. Also, the rate of events was pretty lower: nowadays they're in the order of the *millions*, for a total of between 1 and 2 terabytes of raw data to be processed.

The servers the client is running these processes nowadays are twelve core machines: a huge opportunity to parallelize those old single-threaded tasks. We've got basically all of the ingredients for the recipe, we just need to decide how to build and tune it. Some thoughts before writing any code were necessary to understand the nature of the load and these are the constraints I detected:

- A really huge number of files is to be scanned periodically: each directory contains between one and two millions of files.
- The scanning algorithm is very quick and can be parallelized.
- Processing a file will take at least 1 second, with spikes of even 2 or 3 seconds.
- When processing a file, there is no other bottleneck than CPU.
- CPU usage must be tunable, in order to use a different load profile depending on the time of the day.

I'll thus need a thread pool whose size is determined by the load profile active at the moment of invoking the process. I'm inclined to create, then, a fixed size thread pool executor configured according to the load policy. Since a processing thread is only CPU-bound, its core usage is 100% and waits on no other resources, the load policy is very easy to calculate: just take the number of core available in the processing

environment and scale it down using the load factor that's active at that moment (and check that at least one core is used in the moment of peak):

```
int cpus = Runtime.getRuntime().availableProcessors();
int maxThreads = cpus * scaleFactor;
maxThreads = (maxThreads > 0 ? maxThreads : 1);
```

Then, I need to create a ThreadPoolExecutor using a blocking queue to bound the number of submitted tasks. Why? Well: the directory scanning algorithms are very quick and will generate a huge number of files to process very quickly. How huge? It's hard to predict and its variability is pretty high. I'm not going to let the internal queue of my executor fill up indiscriminately with the objects representing my tasks (which include a pretty huge file descriptor). I'll prefer let the executor reject the files when the queue fills up.

Also, I'll use the ThreadPoolExecutor.CallerRunsPolicy as rejection policy. Why? Well, because when the queue is filled up and while the threads in the pools are busy processing the file, I'll have the thread that is submitting the task executing it. This way, the scanning stops to process a file and will resume scanning as soon as it finishes executing the current task.

Here's the code that creates the executor:

```
ExecutorService executorService =
  new ThreadPoolExecutor(
    maxThreads, // core thread pool size
    maxThreads, // maximum thread pool size
    1, // time to wait before resizing pool
    TimeUnit.MINUTES,
    new ArrayBlockingQueue<Runnable>(maxThreads, true),
    new ThreadPoolExecutor.CallerRunsPolicy());
```

The skeleton of the code is the following (greatly simplified):

```
    // scanning loop: fake scanning
    while (!dirsToProcess.isEmpty()) {
      File currentDir = dirsToProcess.pop();


      // listing children
      File[] children = currentDir.listFiles();


      // processing children
      for (final File currentFile : children) {
      // if it's a directory, defer processing
      if (currentFile.isDirectory()) {
        dirsToProcess.add(currentFile);
        continue;
      }


      executorService.submit(new Runnable() {
        @Override
        public void run() {
          try {
            // if it's a file, process it
            new ConvertTask(currentFile).perform();
          } catch (Exception ex) {
            // error management logic
          }
        }
      });
    }


    // ...

    // wait for all of the executor threads to finish
```

```
executorService.shutdown();

try {
  if (!executorService.awaitTermination(60,
TimeUnit.SECONDS)) {
    // pool didn't terminate after the first try
    executorService.shutdownNow();
  }


  if (!executorService.awaitTermination(60,
TimeUnit.SECONDS)) {
    // pool didn't terminate after the second try
  }
} catch (InterruptedException ex) {
  executorService.shutdownNow();
  Thread.currentThread().interrupt();
}
```

Conclusion

As you can see, the Java concurrency API is very easy to use, very flexible and extremely powerful. Some years ago, I would have taken much more effort to write such a simple program. This way, I could quickly solve a scalability problem caused by a legacy single threaded component in a matter of hours.

Reference: Using a ThreadPoolExecutor to Parallelize Independent Single-Threaded Tasks from our JCG partner Enrico Crisostomo at the The Grey Blog.