Java and Android Concurrency

Task Execution

fausto.spoto@univr.it

git@bitbucket.org:spoto/java-and-android-concurrency.git

git@bitbucket.org:spoto/java-and-android-concurrency-examples.git

Single Thread Scheduling of Task

Many concurrent applications are organized around the idea of the execution of tasks, that is, of (likely independent) units of work

Tasks can be executed on a single thread:

```
@ThreadSafe
public class SingleThreadWebServer {
  public static void main(String[] args) throws IOException {
    try (ServerSocket socket = new ServerSocket(80)) {
      while (true) {
        Socket connection = socket.accept();
        handleRequest(connection);
      }
    }
  }
}
```

It is the simplest scheduling policy, which avoids any concurrency problem, but it is of course rather inefficient

Explicit Creation of Threads per Task

Better responsiveness can be achieved by starting a new thread for each task:

```
@ThreadSafe
public class ThreadPerTaskWebServer {
  public static void main(String[] args) throws IOException {
    try (ServerSocket socket = new ServerSocket(80)) {
      while (true) {
        Socket connection = socket.accept();
        Runnable task = () -> handleRequest(connection);
        new Thread(task).start();
      }
    }
  }
}
```

Better Responsiveness, parallel execution of CPU bound and I/O bound tasks. Tasks must be thread-safe!

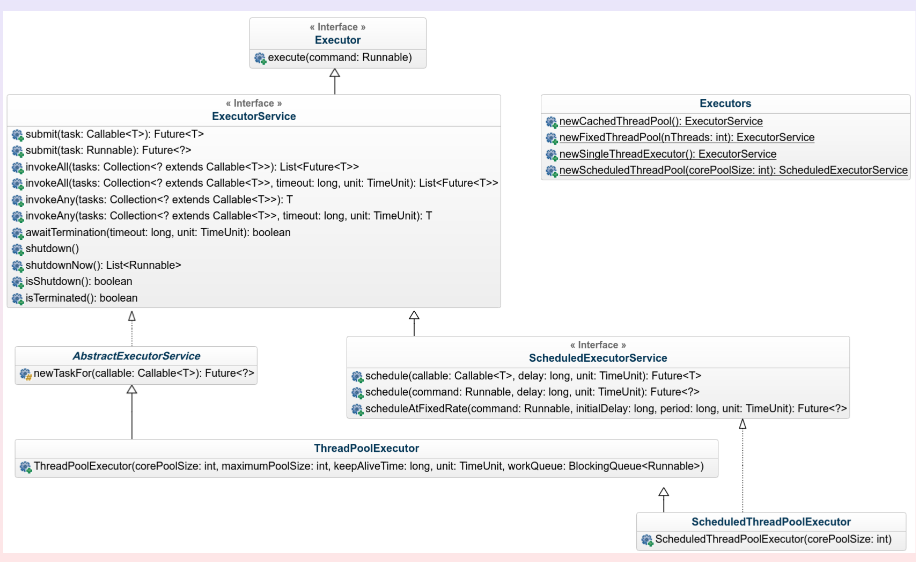
Drawbacks of Unbound Thread Creation

The implicity of `ThreadPerTaskWebServer` comes at some price:

- thread lifecycle overhead: creating and destroying threads is slow
- resource consumption: creating more runnable threads than available CPUs costs memory without any efficiency gain
- there is a limit on the number of threads. Trespassing that limit brings the application to an out-of-memory exception and the server to a denial-of-service

Up to a certain point, more threads can improve throughput, but beyond that point creating more threads just slows down the application, and creating one thread too many can cause the entire application to crash horribly

The Executor Framework



Using an Executor for a Parallel Web Service

```
@ThreadSafe
public class TaskExecutionWebServer {
  private static final int NTHREADS = 100;
  private static final Executor exec
    = Executors.newFixedThreadPool(NTHREADS);

  public static void main(String[] args) throws IOException {
    try (ServerSocket socket = new ServerSocket(80)) {
      while (true) {
        Socket connection = socket.accept();
        Runnable task = () -> handleRequest(connection);
        exec.execute(task);
      }
    }
  }
}
```

Thread Pools

Thread pools are the implementation of an executor:

- a set of threads, ready to run tasks
- a queue holding the task to process

Advantages

- threads are recycled, hence amortizing creation and teardown costs
- the right number of threads allows one to keep the processor busy without running into out of memory

Creation of Thread Pools

- by using the factory methods of `Executors`, for predefined configurations (preferred)
- by using the constructors of `ThreadPoolExecutor`, for fine-grained configuration

Predefined Thread Pool Configurations from `Executors`

`Executors.newFixedThreadPool(int nThreads)`

yields a thread pool that creates threads up to the given limit and adds a new thread if one dies

`Executors.newCachedThreadPool()`

yields an unboound thread pool, whose size grows and shrinks with the number of pending tasks

`Executors.newSingleThreadExecutor()`

yields a thread pool with a single thread, that executes tasks sequentially and replaces the only thread if it dies

`Executors.newScheduledThreadPool(int corePoolSize)`

yields a thread pool with the given number of threads, able to schedule delayed or periodic tasks

`ExecutorService`'s Shutdown

The JVM does not terminate until the last non-daemon thread has terminated

Creating a thread pool prevents JVM's termination
Thread pools must be shut down when no longer in use

An `ExecutorService` transitions through the following states (from left to right, never backwards):

$$\textit{running} \Rightarrow \textit{shutting down} \Rightarrow \textit{terminated}$$

ExecutorService's Lifecycle Methods

`void shutdown()`: *running* ⇒ *shutting down*

Graceful shutdown: wait for all pending tasks to finish but don't accept any more work. At the end, transition to the *terminated* state

`List<Runnable> shutdownNow()`: *running* ⇒ *shutting down*

Abrupt shutdown: interrupt the running tasks, wait for them to finish, don't accept any more work and return the list of pending but not yet run tasks. At the end, transition to the *terminated* state

`boolean isShutdown()`

Yields true if and only if the executor service is in the *shutting down* state

`boolean isTerminated()`

Yields true if and only if the executor service is in the *terminated* state

Wait for the Termination of an `ExecutorService`

`boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException`

Blocks and waits the given amount of time until the executor service reaches the *terminated* state

- if the *terminated* state is reached by the given timeout: yields true
- if the *terminated* state is not reached by the given timeout: yields false
- if the current (waiting) thread is interrupted before the given timeout: throws an `InterruptedException`

Rejected Tasks

What happens is a new, incoming task is submitted to an executor service in the *shutting down* or *terminated* state?

`RejectedExecutionException`

This behavior can be changed by specifying a suitable rejected execution handler

Stoppable Web Server

```
@ThreadSafe
public class LifecycleWebServer {
  private final ExecutorService exec = Executors.newCachedThreadPool();

  public void start() throws IOException {
    try (ServerSocket socket = new ServerSocket(80)) {
      while (!exec.isShutdown())
        try {
          Socket connection = socket.accept();
          exec.execute(() -> handleRequest(connection));
        }
        catch (RejectedExecutionException e) {
          log("task submission rejected", e);
        }
    }
  }

  public void stop() {
    exec.shutdown(); // graceful shutdown
  }
}
```

Delayed and Period Tasks

Do not use a `java.util.TimerTask`

The legacy class `TimerTask` has serious issues that make it deprecated nowadays (a single thread for *all* `TimerTask`s, non-replaceable)

Instead, use a `ScheduledExecutorService` and its scheduling methods

```
ScheduledExecutorService exec
          = Executors.newScheduledThreadPool(nThreads);

// run after 10 milliseconds from now
exec.schedule(task, 10L, TimeUnit.MILLISECONDS);

// run after 10 milliseconds from now and then every second
exec.scheduleAtFixedRate(task, 10L, 1000L, TimeUnit.MILLISECONDS);
```

Example: HTML Page Rendering

The sequential approach is simple and does not use any executor.
However, downloading all images might take a while, only then are images rendered

```
public class SingleThreadRenderer {
  void renderPage(CharSequence source) {
    // render the text
    renderText(source);

    // download all images
    List<ImageData> imageData = new ArrayList<>();
    for (ImageInfo imageInfo: scanForImageInfo(source))
      imageData.add(imageInfo.downloadImage());

    // render the images
    for (ImageData data: imageData)
      renderImage(data);
  }
}
```

Split Page Rendering in Two Concurrent Tasks

```
public abstract class FutureRenderer {
  private final ExecutorService executor = Executors.newCachedThreadPool();

  void renderPage(CharSequence source) {
    final List<ImageInfo> imageInfos = scanForImageInfo(source);
    Callable<List<ImageData>> task = () -> {
      List<ImageData> result = new ArrayList<>();
      for (ImageInfo imageInfo: imageInfos)
        result.add(imageInfo.downloadImage());
      return result;
    };
    Future<List<ImageData>> future = executor.submit(task);
    renderText(source); // in parallel with image downloads
    try {
      for (ImageData data: future.get())
        renderImage(data);
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
      future.cancel(true);
    } catch (ExecutionException e) {
      throw launderThrowable(e.getCause());
    }
  }
}
```

Limitations of Splitting Heterogeneous Tasks

If heterogeneous tasks require very different times, running them in parallel might not be worthwhile and complicates the code

The real performance payoff of dividing a program's workload into tasks comes when there are a large number of independent, *homogeneous* tasks that can be processed concurrently

Downloading Each Image in Parallel

A much faster and reactive page renderer would download each page in its own thread and render the image as soon as its download finished.

A completion service is a bridge between an executor and a blocking queue

- tasks submitted to the completion service are delegated to the executor
- when a task finishes, its future is put in the blocking queue

By taking futures from the queue, one gets fully computed futures, whose `get()` method will not block

Render Each Image As Soon As It Becomes Available

```
public abstract class Renderer {
  private final ExecutorService executor;

  Renderer(ExecutorService executor) {
    this.executor = executor;
  }

  void renderPage(CharSequence source) {
    List<ImageInfo> info = scanForImageInfo(source);
    CompletionService<ImageData> completionService =
      new ExecutorCompletionService<>(executor);
    for (ImageInfo imageInfo: info)
      completionService.submit(() -> imageInfo.downloadImage());
    renderText(source);
    try {
      for (int t = 0, n = info.size(); t < n; t++) {
        Future<ImageData> f = completionService.take();
        renderImage(f.get());
      }
    }
    catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    catch (ExecutionException e) { throw launderThrowable(e.getCause()); }
  }
}
```

Running a Task with a Time Limit

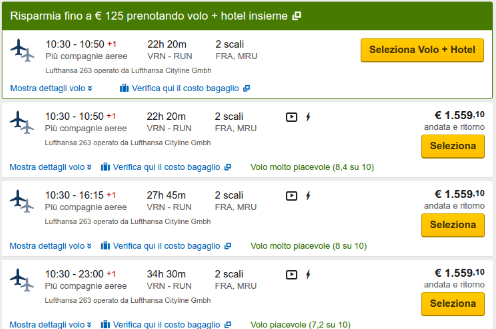
```
public class RenderWithTimeBudget {
  private static final Ad DEFAULT_AD = new Ad();
  private static final long TIME_BUDGET = 1000;
  private static final ExecutorService exec = Executors.newCachedThreadPool();

  Page renderPageWithAd() throws InterruptedException {
    long endNanos = System.nanoTime() + TIME_BUDGET;
    Future<Ad> f = exec.submit(new FetchAdTask());
    Page page = renderPageBody(); // render the page while waiting for the ad
    Ad ad;
    try {
      // Only wait for the remaining time budget
      long timeLeft = endNanos - System.nanoTime();
      ad = f.get(timeLeft, TimeUnit.NANOSECONDS);
    }
    catch (ExecutionException e) { ad = DEFAULT_AD; }
    catch (TimeoutException e) {
      ad = DEFAULT_AD;
      f.cancel(true); // useful if the task reacts to cancellation
    }
    page.setAd(ad);
    return page;
  }
}
```

Running More Tasks with a Time Limit

The typical travel reservation portal:

- it contacts many travel providers concurrently
- if they answer under the time limit, their quote is reported
- after the time limit expires, the travel provider is discarded



Asking More Travel Quotes under a Time Limit

```
public class TimeBudget {
  private static ExecutorService exec = Executors.newCachedThreadPool();

  public List<Quote> getRankedQuotes(Travel travel, Set<Company> companies,
     Comparator<Quote> ranking) throws InterruptedException {
    List<QuoteTask> tasks = new ArrayList<>();
    for (Company company: companies)
      tasks.add(new QuoteTask(company, travel));
    List<Future<Quote>> futures = exec.invokeAll(tasks, 10000, MILLISECONDS);
    List<Quote> quotes = new ArrayList<>(tasks.size());
    Iterator<QuoteTask> taskIter = tasks.iterator();
    for (Future<Quote> f: futures) {
      QuoteTask task = taskIter.next();
      try {
        quotes.add(f.get());
      } catch (ExecutionException e) {
        quotes.add(task.getFailureQuote(e.getCause()));
      } catch (CancellationException e) {
        quotes.add(task.getTimeoutQuote(e));
      }
    }
    Collections.sort(quotes, ranking);
    return quotes;
  }
}
```

Exercise: Matrix Multiplication with an Executor

Consider the parallel multiplication again. Use an executor to avoid repeated threads creation and teardown

How much faster is the resulting implementation? How much higher is the CPU utilization average?

Use the same executor also in the constructor of random matrices. Make that random construction parallel. What happens to the execution time now? Are you sharing a data structure among the threads? Is it thread-safe? Must it really be shared?