Concurrency II: executors

1 More concurrency

Last unit we covered the basics of concurrent programming. We used the low-level elements that we can use in Java to create programs where different tasks are being executed at the same time: threads and locks (and in particular, the `synchronized` keyword).

Although any kind of concurrent program can be created using these elements, in reality they are too basic for big programs and/or advanced tasks. It is very easy to make mistakes that will make a program run slower or —worse— block completely by using threads and locks directly. In the current context, in which multi-core computers have become the norm and applications need to be able to benefit from this inherent parallelism provided by the hardware, higher-order structures are needed to create reliable concurrent programs.

Java 5 introduced several features that aimed at covering this gap. The two most imporant, executors and concurrent collections, are explained in the following sections.

2 Executors

An `Executor` is an object that executes tasks. These tasks are defined by other objects. This is similar to what we have been doing with threads. We have created threads that ran tasks that were defined by a `Runnable` object, with a `run()` method.

The problem with this approach is that the creation of the thread, its lifetime, and the task it performs are closely linked. There is no separation between thread and task, between container and content. The thread lives for as long as the task is running, and as soon as the task finishes the thread dies. This has several disadvantages:

- Thread management is done manually and, therefore, is an error-prone process. The programmer must take care not only of the tasks that need to be performed (sometimes called the *business logic*), but also of creating the threads to run them, starting them, managing their interruptions, etc.
- Thread creation takes time. In a highly concurrent application, creation and destruction (i.e. garbage collection) of threads can result in a performance penalty.
- Thread are memory structures that use a non-trivial amount of memory. If threads are created manually one by one there is no easy and scalable way to measure the load of the application and to manage it carefully to prevent running out of resources.

In a large-scale application, it makes sense to separate thread creation and management from the rest of the application. Executors are objects that encapsulate these operations hiding them from the rest of the program.

2.1 Three types of executors

Executors are defined by three interfaces in the `java.util.concurrent` package; the package also provides some implementations for these interfaces.

Executor: This is the basic executor. It just supports launching new tasks.

ExecutorService: This is an extension of the former one, and adds features that help manage the lifecyle of the tasks to be run and also the executor itself.

ScheduledExecutorService: An extension of the former one, this interface adds methods to execute tasks in the future and at scheduled times.

As usual, variables that represent executor objects should be declared as one of these interfaces and not as one of the classes that implement them, for example:

```
Executor executor = new ThreadPoolExecutor();
```

2.1.1 Executor

The `Executor` interface defines only one method: `execute(Runnable)`. This interface is rarely used compared to the other two, but it was designed to be an easy replacement when fixing legacy code that had been implemented using pure threads and locks. If `r` is a `Runnable` object and `e` is an `Executor` object, than we can replace:

```
Thread t = new Thread(r);
t.start();
```

with

```
e.execute(r);
```

Simple, cleaner, and —more important— it opens the door to using one of the other two more powerful executors. Additionally, depending on the class that implements the `Executor`, it may be able to reuse threads from a thread pool or put the runnable tasks in a queue if the system is busy; these things are not possible when using plain threads. Thread pools and queues of tasks are two constructs that are very convenient to ease the management of threads and solve some of the problems of using plain threads as explained above. More on that below.

2.1.2 Executor Services

The `ExecutorService` interface extends the plain `Executor` with the `submit(...)` method, which is similar to `execute(Runnable)` but more versatile because it accepts both `Runnable` and `Callable` tasks (i.e. described by objects). Callable objects are similar to runnable objects but they can return a value when they finish their computation.

The `submit(...)` method returns a `Future`, which represents the result of the computation; it is called *future* because it is only known after some indeterminate time has passed since the service was called. The `Future` object can be used to retrieve the `Callable` return value, and to manage the status of both `Callable` and `Runnable` tasks.

The interface `ScheduledExecutorService` adds methods to schedule the execution of `Runnable` and `Callable` tasks at specific points in time: `schedule(...)` executes the tasks after some time delay has passed, while `scheduleAtFixedRate(...)` and `scheduleWithFixedDelay` execute specified tasks repeatedly at defined intervals.

2.2 Thread pools

The classes in `java.util.concurrent` that implement the executor interfaces described in the former section use in most cases a *thread pool*. A thread pool is a collection or worker threads to which tasks (defined by `Runnable` and `Callable` objects) can be asigned. This kind of thread exists separately from the tasks it executes and is often used to execute multiple tasks at different times.

Using thread pools reduces the overhead due to thread creation. Thread objects use a non-trivial amount of memory; in large projects allocation and disposal of a lot of thread objects can result in a noticeable performance cost. Having a pool of threads already created that can be reused results in better performance.

One common type of thread pool is the fixed thread pool. A pool of this type has a specific number of threads running at all times; if for any reason a thread terminates while it is still in use, another thread will be created to replace it. If there are more tasks to be executed than threads, tasks will be added to an internal queue. As soon as thread finishes its task and becomes available, new tasks are retrieved from the queue and assigned to the thread. In the uncommon situation in which there are more tasks than threads —or no tasks at all— then the “idle” threads will `wait()`.

Having a fixed thread pool is a way of managing the load of your application. By capping the number of parallel tasks that the system will be running at any given time (i.e. by the size of the pool), the programmer makes sure that the system will not crash unexpectdely under stress. Without a limit on the number of threads to be created, the application would create threads until it ran out of memory, and then it would crash, which is bad. By limiting the number of tasks that can be executing at the same time, and putting the new incoming tasks in a queue until there is a thread available for them, the application *degrades gracefully*; this means that it simply becomes gradually more slow in its response time, instead of becoming unresponsive suddenly and unexpectedly, and this is good.

3 Concurrent Collections

We have already used several collections from the Java Collections Framework (JCF): `List`, `Queue`, `Stack`, `Map`, `Set`... However, all these interfaces are not defined to be thread-safe, and therefore the classes that implement them (such as `ArrayList` and `HashSet`) are not expected to be. This is OK for single-threaded applications, but in concurrent applications we need to make sure that our data structures remain consistent.

We can do this by using threads and locks (i.e. `synchronized`), but the `java.util.concurrent` package also includes some nice additions to the JCF. The two more interesting ones are described below:

BlockingQueue: This is like a normal queue with two exceptions. When you try to insert an element into an already full queue, it will wait until there is space instead of failing; it can timeout if the queue remains full for too long. A similar behaviour applies when you try to retrieve an element from an empty queue: it will wait until there is something to retrieve. . . or timeout.

ConcurrentMap: This is an extension of `Map` that adds atomicity to the usual put-replace-delete operations of maps. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid external synchronization.