159.341 Programming Languages, Algorithms & Concurrency

Java Concurrency - Part 2

Daniel Playne
d.p.playne@massey.ac.nz

Reminder

- Java threads - `Runnable`
- Synchronisation - `synchronized`
- `wait/notify`

Java Concurrency - `java.util.concurrent`

Java provides a library of data structures and classes to help develop programs using concurrency.

One of the interfaces defined by `java.util.concurrent` is the `Executor` interface which will take a `Runnable` and execute it "at some time in the future".

```
public interface Executor {
   void execute(Runnable command);
}
```

Java Concurrency - Executors

Implementations of this interface have a large degree of flexibility for how they implement this interface.

Different `Executors` could:

- launch a new thread to execute the command.
- use a thread from a pool of threads to execute it.
- or even execute it in the calling thread.

Java Concurrency - Executors

One useful implementation of `Executor` provided by Java is named `ThreadPoolExecutor`.

This implementation maintains a pool of threads which it can use to execute `Runnable` objects passed to `execute()`.

When every thread in the pool is assigned a task, any new `Runnable` objects will be added to a work queue and executed once a thread becomes available.

Java Concurrency - Executors

`ThreadPoolExecutor` constructor allows the executor to be configured:

- `corePoolSize` - number of threads the executor should keep alive.
- `maximumPoolSize` - maximum number of threads the executor may created.
- `keepAliveTime` - time that an idle, non-core thread is kept alive.
- `unit` - Time unit for `keepAliveTime`.

...

Java Concurrency - Executors

`ThreadPoolExecutor` constructor allows the executor to be configured:

- `workQueue` - queue that holds `Runnable` objects before they are started.
- `threadFactory` - interface used to create new threads.
- `handler` - controls what happens if the `workQueue` and `maximumPoolSize` are exceeded.

Java Concurrency - Executors

```
// WorkQueue
BlockingQueue<Runnable> queue = new LinkedBlockingQueue<Runnable>(10);

// ThreadFactory
ThreadFactory factory = new ThreadFactory() {
   private int mId = 0;
   public Thread newThread(Runnable r) {
      int id;
      synchronized(this) {id = mId++;}
      System.out.println("Creating new thread " + id + ".");
      return new Thread(r, "Thread " + id);
   }
};

// Handler
RejectedExecutionHandler handler = new RejectedExecutionHandler() {
   public void rejectedExecution(Runnable r, ThreadPoolExecutor executor) {
      System.out.println("Rejected task:  " + ((Example08Executor)r).getId());
   }
};

// Create Executor
ThreadPoolExecutor executor = new ThreadPoolExecutor(5, 10, 2, TimeUnit.SECONDS,
                                                     queue, factory, handler);
```

Java Concurrency - Executors

Simple `Runnable` object that just out some messages and sleeps for five seconds.

```
private int mTaskId;

public Example08Executor(int id) {
   mTaskId = id;
}

public void run() {
   System.out.println("Starting task:  " + mTaskId);

   try {
      Thread.sleep(5000);
   } catch(InterruptedException exception) {}
   System.out.println("Completed task: " + mTaskId);
}
```

Java Concurrency - Executors

Launching 25 tasks will mean 10 can be executed, 10 can be stored in the workQueue and another 5 will be rejected.

```
for(int i = 0; i < 25; i++) {
   executor.execute(new Example08Executor(i));
}

executor.shutdown();

executor.awaitTermination(500, TimeUnit.SECONDS);
```

Java Concurrency - ExecutorService

`Executor` is a very simple interface that only requires that the `Runnable` tasks be executed (or possibly rejected).

Java provides another interface called `ExecutorService` which provides some additional functionality including the ability to query the result of executing a task.

`Runnable` objects have `run()` function that returns `void`, however there is another interface `Callable` that we can use instead.

Java Concurrency - Callable

`Callable<T>` is a very simple interface that defines a single method:

```
T call()
```

An implementation of `Callable<T>` can be used to represent a task that returns a value of type `T`.

Java Concurrency - Future

This leads to the problem of how to retrieve the result of executing a `Callable` object.

The submitter of the task does not know when the task is executed or how long it is going to run for.

To retrieve the result of a task (or wait until it is available) we can use a `Future`.

Java Concurrency - Future

A `Future` represents the result of a task that may not have been computed yet.

When a thread submits a `Callable` object it can receive a `Future` object as a result.

It can then query this `Future` object to determine whether the task is completed and retrieve the result.

Java Concurrency - Future

`Future<T>` is an interface that provides the following methods

- `void cancel(bool)` - attempts to cancel a task.
- `T get()` - returns the result (waits if necessary).
- `T get(long, TimeUnit)` - same as `get()` but has a maximum time it waits.
- `bool isCancelled()` - returns `true` if task was cancelled.
- `bool isDone()` - returns `true` if task completed.

Java Concurrency - Callable

Simple `Callable` object that returns a `String` message.

```
public class Example09Future implements Callable<String> {
   ...
   private int mTaskId;

   public Example09Future(int id) {
      mTaskId = id;
   }

   public int getId() { return mTaskId; }

   public String call() {
      try {
         Thread.sleep(mTaskId * 1000);
      } catch(InterruptedException execption) {}
      return "Hello from task " + mTaskId;
   }
}
```

Java Concurrency - Futures

Submitting `Callable` objects and retrieving their results through `Futures`.

```
ExecutorService executor = Executors.newFixedThreadPool(4);

List<Future<String>> futures = new ArrayList<Future<String>>(5);

for(int i = 0; i < 5; i++) {
   futures.add(executor.submit(new Example09Future(i)));
}

for(int i = 0; i < 5; i++) {
   try {
      String result = futures.get(i).get();
      System.out.println(result);
   } catch(InterruptedException exception) {
   } catch(ExecutionException exception){}
}
```

Futures

Futures are not novel to Java and were described long before Java existed (early ideas of futures can be traced back to Algol from 1961).

Some languages also use the terms - ***promise***, ***delay*** or ***deferred*** to refer to similar constructs.

In some languages a ***promise*** is slightly different from a ***future***.

Futures

The basic idea is that a future represents a placeholder for a value which will eventually become available.

They are useful for a very common use-case in concurrent applications, a main thread need to compute several expensive functions and wants to delegate them to other threads.

Futures provide a convenient way for the main thread to then retrieve the results of those computations.

Futures

The functionality of futures would not be particularly hard to implement using other synchronisation methods.

However, a well-tested implementation is less likely to lead to concurrency errors that repeatedly writing hand-crafted implementations.

Java Concurrency - Collections

Java provides a large set of data structures which is extended by the `java.util.concurrent` to include thread-safe collections that can be accessed by multiple threads.

Saying a collection is ***thread-safe*** means that the internal state of a class and the returned values from methods will be correct when access concurrently by multiple threads.

It doesn't mean that you don't have to think about concurrent access anymore.

Java Concurrency - Collections

For example, a Java `Vector` is thread-safe (as opposed to `ArrayList` which is not).

The methods in `Vector` are synchronized to ensure that invalid results are not returned or the internal structure corrupted.

A non-synchronized vector could be corrupted if two threads decide they need to resize the vector at the same time.

Java Concurrency - Collections

Although a `Vector` is thread-safe and each method is synchronized, you do still need to consider race-conditions.

```
Vector<Integer> v = ...

if(v.contains(10) == false) {
   v.add(10);
}
```

Synchronization on the `contains` and `add` methods doesn't prevent two threads both adding 10 to the vector.

Java Concurrency - Collections

Iterators for concurrent Java collections are *fail-fast* - if a vector is structurally modified (in any way other than with that iterator) then any existing iterators will throw a `ConcurrentModificationException`.

This behaviour is intended to fail quickly rather than allowing a potentially corrupt iterator to still be used (even though in some cases it may still be valid).

Summary

- `Executor` / `ExecutorService`
- `Runnable` / `Callable`
- `Future`
- Concurrent collections