Thread Subcontracting: Callables, Futures & Executors

4010-441

Principles of Concurrent System Design

Rather than managing individual threads ourselves, we would like to hand that work off to a subcontractor.

- Managing threads is error prone.
- We want controlled levels of concurrency.
- We want "fire and forget" functionality.

We will need to specify to our subcontractor the code that we want to execute.

You have already used objects of which class to specify code to execute?

What would a method to simply execute the code specified by an object of that class look like?

Suppose you want the code executed in its own thread what would the method look like?

Having a basic method to execute code is nice, but to make this really useful we need other features.

Think about the different tasks you would like to do concurrently. Some examples might be:

- **Handle requests from separate clients in the order the requests arrived**
- **Execute a large computation in parallel chunks**
- **Complete execution of a task in stages**
- **On a regular basis, perform preventative maintenance checks on our system while it continues to run**
- **…**

To handle these varied types of tasks, you may need a coordinator for the subcontractors. What are the features that you would want in a coordinator?

The Java concurrency package provides some basic building blocks to use.

- Executor – interface for executing a task
- Callable – interface to define task to execute with more features than Runnable
- Future – separates execution from getting the result of the asynchronous task execution
- ExecutorService – interface for managing task execution
- Executors – factories for instantiating execution services

Building Block #1 - Callables

- Callable is an interface like Runnable.

```
public interface Callable<V> {
   public V call()
}
```

- Differences:
  - Callables can return values.
  - Callables can propagate exceptions.
  - Callables are examples of "objects as code."
- Example: Compute statistics from data:

```
class ComplexStats<Stats> implements Callable<Stats> {
  DataSet ds ;
  public ComplexStats(DataSet ds) { this.ds = ds }
  public Stats call() {
    Stats result ;
    result = // . . . whatever based on data set . . .
    return result ;
  }
}
```

Building Block #2 - Futures

- A result of asynchronous execution (that is, the result will be available in the future).

```
public interface Future<V> {
  public boolean cancel(boolean mayInterrupt);
  public V get() ;
  public V get(long timeout, TimeUnit unit) ;
  public boolean isCancelled() ;
  public boolean isDone() ;
}
```

- Decouples the execution specified in a Callable from retrieval of its result.

Building Block #3 ExecutorService

- Extension of basic Executor
- Adds services for:
  - Submitting jobs and getting `Future` tokens.
  - Shutting down all the managed tasks.
- Interface summary:

```
public interface ExecutorService {
  <T> Future<T> submit( Callable<T> task ) ;
  <T> Future<T> submit( Runnable task, T result ) ;
      Future<?> submit( Runnable task ) ;

  void shutdown() ;                  // no more submissions
  List<Runnable> shutdownNow() ;     // halt dead in tracks
  boolean isShutdown() ;             // executor has been shutdown

  boolean isTerminated() ;           // all tasks terminated
  boolean awaitTermination() ;       // wait until all tasks complete
  . . .
}
```

Executor Factories for ExecutorService Objects

`static ExecutorService newCachedThreadPool()`

An executor that will create new threads as needed but will reuse existing ones when possible.

`static ExecutorService newFixedThreadPool(int nThreads)`

An executor that will allocate tasks from an unbounded queue to a fixed pool of size *nThreads*.

`static ScheduledExecutorService newScheduledThreadPool(int csize)`

An executor that will allocate tasks to a pool of size csize. Adds methods to run a task periodically or after a delay.

`static ExecutorService newSingleThreadExecutor()`

An executor that will allocate tasks from an unbounded queue to a single thread.

`static ExecutorService newSingleThreadScheduledExecutor()`

An executor that will allocate periodic or delayed tasks to a single thread.

The recursive quicksort algorithm is ripe for speedup through concurrency.

```
quicksort( collection ) {
   partition data around pivot with all values less/greater
   than pivot in left half, and all values greater/less than
   pivot in right half

   sort left half
   sort right half
}
```

Where will the concurrency exist?

Will you use a fixed pool or expanding pool of threads? Why?

How will you coordinate shutdown? Does it make sense to use latches or barriers? Why?

Last But Not Least: ExecutorCompletionService<V>

- Combines executor at front end with a result queue at the rear end.
- As an Executor:
  - Has a submit method
  - Can submit Callable<V>s and Runnables
  - Always get back a Future<V>
  - For Runnables, provide the V object to return
- As a completion service:
  - Provide a V take() method to retrieve next result Future<V> when a task completes.
  - Also have V poll(), with and without timeouts.
  - No guarantee the results come out in the order of submission.

Closing Thought Question

There are two ways one can invoke a method to get work done:

1. Call it directly
2. Wrap it in a Callable, submit it for execution, and retrieve the result later via a Future.

How are these two mechanisms alike?

What is the primary distinction between them?

Concurrent QuickSort

```
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;

public class Qsort {
   private static final int NTHREADS = 4;
   private static final AtomicInteger taskCount = new AtomicInteger();
   private int values[];
   private final ExecutorService es =
       Executors.newFixedThreadPool(NTHREADS) ;

   class Sorter implements Runnable { ... next slide ... }

   public Qsort(int v[]) {
      this.values = (int[]) v.clone();
   }

   public int[] doSort() throws InterruptedException {

      (void) taskCount.set(1); // about to submit the first task

      (void) es.submit(new Sorter(0, values.length - 1));

      if( ! es.awaitTermination(10L, TimeUnit.MINUTES) )
         throw new InterruptedException("Sort took too long.");

      return (int[]) values.clone();
   }
}
```

Concurrent QuickSort

```
// Use quicksort recursively to sort the array from lo to hi, inclusive.

  class Sorter implements Runnable {
     private int lo;
     private int hi;

     Sorter(int lo, int hi) {
        this.lo = lo;
        this.hi = hi;
     }

     private int partition() { ... next slide ... }

     public void run() {
        if ( lo < hi ) {
           int pivpos = partition();

           taskCount.addAndGet(2) ;  // two additional tasks

           es.submit(new Sorter(lo, pivpos - 1));
           es.submit(new Sorter(pivpos + 1, hi));
        }
        // This task is about to terminate. Turn off the lights if it
        // is the last one.
        if( taskCount.decrementAndGet() == 0 ) {
               es.shutdown();
        }
     }
  }
```

Concurrent QuickSort

```
 private int partition() {
   int i = lo - 1 ;
   int t ;
   for ( int j = lo ; j < hi ; j++ ) {
      if ( values[j] <= values[hi] ) {
         t = values[j] ;
         i++ ;
         values[j] = values[i] ;
         values[i] = t ;
      }
   }
   t = values[i + 1] ;
   values[i + 1] = values[hi] ;
   values[hi] = t ;
   return i + 1 ;
}
```