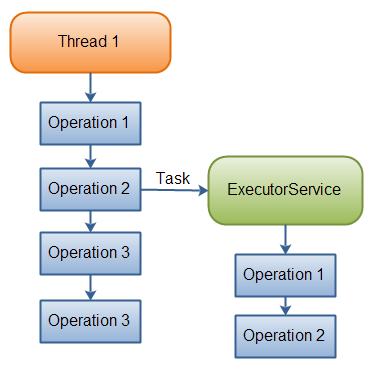
**Working with Java Executor framework in multithreaded application**

The Java platform always has a strong support for concurrent programming and multithreading. But in earlier days the support was in the form of calling native constructs itself in the application layer. The biggest disadvantage of this approach was to handle those primitive construct calls efficiently. Otherwise the application will not run properly and unexpected results will be generated.

The **Java Executor Framework** has been introduced in Java 1.5 and it is a part of java concurrency package. The Executor framework is an abstraction layer over the actual implementation of java multithreading. It is the first concurrent utility framework in java and used for standardizing invocation, scheduling, execution and control of asynchronous tasks in parallel threads. The execution rules are defined during the creation of the constructor. And then the executor runs the concurrent threads following the rules set earlier.

**Executor implementation** in java uses thread pools which consists of worker threads. The entire management of worker threads is handled by the framework. So the overhead in memory management is much reduced compared to earlier multithreading approaches.



**Figure 1**: Java Executor Service

Before java 1.5, multithreading applications were created using thread group, thread pool or custom thread pool. As a result the entire thread management was the responsibility of the programmer keeping in mind the following points.

* Thread synchronization
* Thread waiting
* Thread joining
* Thread locking
* Thread notification
* Handling dead lock

And many more concurrent execution issues that arises out of the application requirement. In real project implementation some time it is very difficult to control multithreading applications. The behaviors of the threads are also dependent on the environment where the application is deployed and running. So the same application might behave in a different way on different deployment environment. For example, the processor speed, the RAM size, the band width all has a direct impact on the multithreading application. So you have to keep all these factors in mind before creating architecture for multithreading applications.

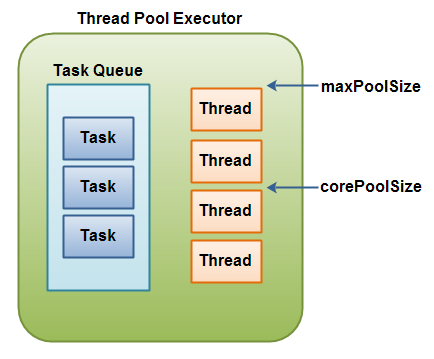
The **Java Executor framework** provides multi-threading applications an easy abstraction layer. The executor abstraction layer hides the critical parts of concurrent execution and the programmer only concentrates on the business logic implementation. In java executor framework all parallel works are considered as tasks instead of simple threads. So the application now deals simply with instances of **Runnable**(which is basically collections of tasks or parallel works) and then it is passed to an **Executor**to process. The **ExecutorService** interface extends the simplistic **Executor**interface. The **Java Executor framework** has life cycle methods to manage the entire concurrent execution flow.

The **Java Executor framework**creates tasks by using instances of Runnable or Callable. In case of Runnable, the run () method does not return a value or throw any checked exception. But Callable is a more functional version in that area. It defines a call () method that allows the return of some computed value which can be used in future processing and it also throws an exception if necessary.

The**FutureTask** class is another important component which is used to get future information about the processing. An instance of this class can wrap either a Callable or a Runnable. You can get an instance of this as the return value of submit () method of an **ExecutorService**. You can also manually wrap your task in a **FutureTask**before calling execute () method.

Following are the functional steps to implement the Java ThreadPoolExecutor.

* A **pool**of multiple threads is created.
* A **queue**is created holding all the tasks but these tasks are not yet assigned to threads from the pool.
* **Rejection handler** is used to handle the situation when one or more tasks are not able to assign in the queue. As per the default rejection policy, it will simply throw a RejectedExecutionException runtime exception, and the application can catch it or discard it.



**Figure 2**: Thread Pool Executor

The following example will show the life cycle of an executor framework. The example will cover only the basis steps and the basic interfaces. There are many more properties and subclasses available to handle different kind of applicant needs.

The following class implements Runnable and its instances will be used as a task in the next section of code.

**Listing 1**: Class implementing Runnable

public class TaskPrint implements Runnable {

private final String name;

private final int delay;

public TaskPrint(String name, int delay) {

this.name = name;

this.delay = delay;

}

public void run() {

System.out.println("Starting: " + name);

try {

Thread.sleep(delay);

} catch (InterruptedException ignored) {

}

System.out.println("Done with: " + name);

}

}

**Listing 2**: Class implementing Executor tasks

import java.util.concurrent.\*;

import java.util.Random;

public class ExecutorServiceExample {

public static void main(String args[]) {

Random random = new Random();

// Create an executor of thread pool size 3

ExecutorService executor = Executors.newFixedThreadPool(3);

// Sum up wait times to know when to shutdown

int waitTime = 600;

for (int i=0; i<10; i++) {

String name = "NamePrinter " + i;

int time = random.nextInt(500);

waitTime += time;

Runnable runner = new TaskPrint(name, time);

System.out.println("Adding: " + name + " / " + time);

executor.execute(runner);

}

try {

Thread.sleep(waitTime);

executor.shutdown();

executor.awaitTermination

(waitTime, TimeUnit.MILLISECONDS);

} catch (InterruptedException ignored) {

}

System.exit(0);

}

}

Following will describe the steps to implement java executor framework in your application.

Create an executor

First you need to create an instance of an **Executor**or **ExecutorService**. The Executor class has a number of static factory methods to create an ExecutorService depending upon the requirement of the application. Following are two main methods to create executor service.

* The **newFixedThreadPool ()**returns a ThreadPoolExecutor instance with an initialized and unbounded queue and a fixed number of threads.
* The **newCachedThreadPool ()** returns a ThreadPoolExecutor instance initialized with an unbounded queue and unbounded number of threads.

In the 1st case no extra thread is created during execution. So if there is no free thread available the task has to wait and then execute when one thread is free. In the 2nd case, existing threads are reused if available. But if no free thread is available, a new one is created and added to the pool to complete the new task. Threads that have been idle for longer than a timeout period will be removed automatically from the pool.

This is a fixed pool of 10 threads.

private static final Executor executor = Executors.newFixedThreadPool(10);

This is a cached thread pool

private static ExecutorService exec = Executors.newCachedThreadPool();

Following is an example of customized thread pool executor. The parameter values depend upon the application need. Here the core pool is having 8 threads which can run concurrently and the maximum number is 12. The queue is capable of keeping 250 tasks. Here one point should be remembered that the pool size should be kept on a higher side to accommodate all tasks. The idle time limit is kept as 5 ms.

private static final Executor executor = new ThreadPoolExecutor(5, 12, 50000L, TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>(250));

Create one or more tasks and put in the queue

After creating the executor now it’s time for creating tasks. Create one or more tasks to be performed as instances of either Runnable or Callable. In this framework, all the tasks are created and populated in a queue. After the task creation is complete the populated queue is submitted for concurrent execution.

Submit the task to the Executor

After creating the ExecutorService and proposed tasks, you need to submit the task to the executor by using either submit () or execute () method. Now as per your configuration the tasks will be picked up from the queue and run concurrently. For example if you have configured 5 concurrent executions, then 5 tasks will be picked up from the queue and run in parallel. This process will continue till all the tasks are finished from the queue.

Execute the task

Next the actual execution of the tasks will be managed by the framework. The Executor is responsible for managing the task’s execution, thread pool, synchronization and queue. If the pool has less than its configured number of minimum threads, new threads will be created as per requirement to handle queued tasks until that limit is reached. If the number is higher than the configured minimum, then the pool will not start any more threads. Instead, the task is queued until a thread is freed up to process the request. If the queue is full, then a new thread is started to handle it. But again it depends upon the type of constructor used during executor creation.

Shutdown the Executor

The termination is executed by invoking its shutdown () method. You can choose to terminate it gracefully, or abruptly.

Conclusion

From the above discussion it is very clear that the concurrency framework is very efficient and widely used in all multithreading applications. This is the first concurrency framework introduced in java. It has added a new dimension in concurrent programming. The programmers are getting a lot of flexibility using this executor framework and hence the applications are also running smoothly without any overhead of memory management synchronization etc.

Read more: <http://mrbool.com/working-with-java-executor-framework-in-multithreaded-application/27560#ixzz5T3tjGdHy>