Java Concurrency in a Nutshell: Thread Pool Overview (Part 2)

Somnath Musib
Sep 13, 2018 · 5 min read

Introduction

In the previous article, we discussed the basics of Threads and *raw* ways of creating Threads.

Java Concurrency in a Nutshell : The Basics (Part 1)
Introduction
medium.com

As discussed, most modern day applications rarely creates a thread using the discussed approach. Java language provides better alternatives with high level language APIs and concurrent utilities to write multi-threaded applications. Moreover, many applications nowadays requires not only one Thread, instead, a pool of Threads to run application with more responsiveness and high throughput. This pool of threads commonly refer as **Thread Pool** and Java language support Thread Pool with **Executor Service** APIs.

Thread Pool Photo by rawpixel on Unsplash

Executor

This is the base interface in Java's thread pool framework. An **Executor** is an object that can execute a **runnable** task. This interface provides a way to decouple task submission from the way tasks will run. So instead of creating a single thread and starting it to complete a task in following way:

```
Thread t = new Thread(new Runnable() {
@Override
public void run() {
    System.out.println("Hello World");
   }
});
t.start();
```

an **Executor** can do in following way

```
Executor executor = Executors.newFixedThreadPool(1); // Create a
thread pool with one Thread
executor.execute(new Runnable() { // Executes the task
@Override
public void run() {
  System.out.println("Hello World");
 }
});
```

Note that thread creation and other stuffs are abstracted in this API and the user only supplied task(s) to be executed to the **Executor** for execution.

Below code snippets shows some sample implementation of Executor interface instead of using fixed thread pool:

```
package com.codefountain.concurrency.threadpool;

import java.util.concurrent.Executor;

/**
 * An implementation of {@link Executor}
 *
 * @author Somnath Musib
 *
 */
public class ExecutorMain {

	public static void main(String[] args) {
		Executor directExecutor = new DirectExecutor();
		directExecutor.execute(() -> {System.out.println("Executed in direct ex
		directExecutor.execute(() -> {System.out.println("Executed in direct ex


		Executor threadPerTaskExecutor = new ThreadPerTaskExecutor();
		threadPerTaskExecutor.execute(() -> {System.out.println("Executed in th
		threadPerTaskExecutor.execute(() -> {System.out.println("Executed in th
	}


	/**
	 * An executor that executes the task on current thread
	 *
	 * @author Somnath Musib
	 *
	 */
	static class DirectExecutor implements Executor{
		@Override
		public void execute(Runnable command) {
			command.run();
		}
	}

	/**
	 * An executor that runs each task in a new thread
	 */
	static class ThreadPerTaskExecutor implements Executor{
		@Override
		public void execute(Runnable command) {
			Thread t = new Thread(command);
```

```
45                          t.start();
46                  }
47          }
48  }
```

ExecutorMain.java hosted with ❤️ by GitHub view raw

Executor Interface implementation

. . .

Executor Service

This is the base interface upon which Java's thread pool is implemented. This interface extends **Executor** interface and adds methods to manage termination and track of submitted tasks. Its **submit** method returns a **Future** object that can be used to cancel execution/wait for the submitted tasks to finish.

An executor service can be shutdown on demand. **shutdown** attempts to perform a graceful shutdown of the pool which allows all previously submitted tasks to be completed but does not accept any new task once shutdown has invoked. On the other hand, **shutdownNow** tries to forcefully shutdown the pool where it prevents all waiting tasks to start and attempts to stop currently executing tasks.

It also provides methods to perform bulk submissions of jobs.With **invokeAll** and **invokeAny**, collections of tasks can be submitted at once and Executor Service will attempt to invoke all or any task(s) respectively.

Java provides a default (abstract) implementation of this interface in **AbstractExecutorService.** This in turn is extended by **ThreadPoolExecutor** and this class provides the complete implementation of Java's thread pool.

Java provides multiple types of Thread Pool implementation based on common usage.

. . .

Future

As discussed, Java provides rich and high level API support to write highly concurrent applications. Although **runnable** interface provides a mechanism to *run* a task in a separate thread, it does little in terms of task monitoring, task control or returning

results/exception to the calling thread. **Future** in Executor Service framework partially helps in achieving above goals.

From Java documentation,

> *A `Future` represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method `get` when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the `cancel` method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled. If you would like to use a `Future` for the sake of cancellability but not provide a usable result, you can declare types of the form `Future<?>` and return `null` as a result of the underlying task*

. . .

Callable

Callable represents a task that can return a result and throws an exception back to the calling thread. It has a single no arg **call** method. A task can be defined inside the **call** method and this callable task can be submitted to an Executor Service for execution.

A callable task, once submitted to an executor service returns a **Future** for task monitoring. Callable also returns the result on completion. It throws an Exception if there is an exception in task execution.

Below code shows an example of Callable and Future:

```
1  package com.codefountain.concurrency.threadpool;
2
3  import java.util.concurrent.Callable;
4  import java.util.concurrent.ExecutionException;
5  import java.util.concurrent.ExecutorService;
6  import java.util.concurrent.Executors;
7  import java.util.concurrent.Future;
8  import java.util.concurrent.TimeUnit;
9
10 /**
11  * Example of {@link Callable} and {@link Future}
12  *
13  * @author Somnath Musib
```

```
 *
 */
public class FutureMain {

	public static void main(String[] args) {

		ExecutorService executorService = null;
		try {
			executorService = Executors.newFixedThreadPool(1);
			Future<Integer> future = executorService.submit(new FutureDemo(
			if(!future.isDone()) {
				System.out.println("|INFO| Task is still in progress.")
			}
			System.out.println("|INFO| Waiting for the computation done.");
			Integer returnValue = future.get();
			System.out.println("|INFO| Sum is "+returnValue);
		}
		catch (InterruptedException | ExecutionException e) {
			System.err.println("|ERROR| Exception occured "+e.getMessage())
		}
		finally {
			System.out.println("|INFO| Shutting down the thread pool");
			executorService.shutdown();
		}

	}


	/**
	 * A {@link Callable} task to compute sum of 1 to 10.
	 * A delay is introduce to display how main thread waits
	 * on blocking future.get() invocation
	 *
	 * @author Somnath Musib
	 *
	 */
	static class FutureDemo implements Callable<Integer>{

		@Override
		public Integer call() throws Exception {
			int sum =0;
			System.out.println("|INFO| Computation is being done by "+Threa
			for(int i=0; i<10;i++) {
				sum+=i;
				TimeUnit.SECONDS.sleep(1);
			}
			return sum;
```

```
61                  }
62
63              }
64  }
```

FutureMain.java hosted with ❤ by GitHub view raw

Callable and Future example

. . .

ThreadPoolExecutor is the main thread pool implementation and base of all types of thread pool provided in Java. Though, this class provides the full implementation for a thread pool, programmers often use more specific types of thread pools e.g. FixedThreadPool, CachedThreadPool, SingleThreadExeutor, WorkStealingPool and so on based on the task processing needs. For example, **FixedThreadPool** provides a fixed set of threads for task execution with a **LinkedBlockingQueue** to task queuing if tasks can not be picked up for immediate processing. This implementation is suitable for good resource optimization.

Before proceed further lets understand how a thread pool internally works. In Java, each thread pool has following components:

1. Core Pool Size
2. Maximum Pool Size
3. A Queue
4. A Thread Factory
5. Rejection Execution Handler

Core Pool Size

This is the minimum number of threads that should be always available in the pool. This is applicable even if there are no active task available in the pool for processing. In the even of an abnormal termination of a core pool thread, Java ensures a thread gets created and core pool size is maintained. However, core pool size threads can be terminated if **allowCoreThreadTimeOut** flag is set to true. In this case, core pool threads will be terminated once **keepAliveTime** expires. Thread pool will have no thread if pool is inactive and **keepAliveTime** expires for all core pool threads

Maximum Pool Size

This is the maximum number of threads that a pool can have and can not go beyond this value.

Queue

A queue is used internally to hold the additional tasks once all core pool threads occupied and not available for taking up new tasks. Note that a new thread is created if there are fewer no of threads running than the configured no of core poll threads and job is submitted for processing rather than queuing.

Thread Factory

A thread factory is used to create new threads in the thread pool.

Rejection Execution Handler

This handler is used to decide when thread pool is not able to accept new tasks due to different scenarios. This includes max pool size is reached or queue is full. There are different policies on how to handle tasks that can not be processed by the pool

Conclusion

This article focuses on the building blocks of Java thread pool and provides an overview. These building blocks are very fundamental to Java concurrency framework and a good understanding certainly helps on writing better multi threaded applications.