Understanding Thread Pool

In terms of performance, creating a new thread is an expensive operation because it requires the operating system allocates resources need for the thread. Therefore, in practice thread pool is used for large-scale applications that launch a lot of short-lived threads in order to utilize resources efficiently and increase performance.

Instead of creating new threads when new tasks arrive, a thread pool keeps a number of idle threads that are ready for executing tasks as needed. After a thread completes execution of a task, it does not die. Instead it remains idle in the pool waiting to be chosen for executing new tasks.

You can limit a definite number of concurrent threads in the pool, which is useful to prevent overload. If all threads are busily executing tasks, new tasks are placed in a queue, waiting for a thread becomes available.

The Java Concurrency API supports the following types of thread pools:

- **Cached thread pool**: - keeps a number of alive threads and creates new ones as needed.
- **Fixed thread pool**: - limits the maximum number of concurrent threads. Additional tasks are waiting in a queue.
- **Single-threaded pool**:- keeps only one thread executing one task at a time.
- **Fork/Join pool**: a special thread pool that uses the Fork/Join framework to take advantages of multiple processors to perform heavy work faster by breaking the work into smaller pieces recursively.

That's basically how thread pool works. In practice, thread pool is used widely in web servers where a thread pool is used to serve client's requests. Thread pool is also used in database applications where a pool of threads maintaining open connections with the database.

Implementing a thread pool is a complex task, but you don't have to do it yourself. As the Java Concurrency API allows you to easily create and use thread pools without worrying about the details.

Why you need thread pool in Java?

Answer is usually when you develop a simple, concurrent application in Java, you create some Runnable objects and then create the corresponding Thread objects to execute them. Creating a thread in Java is an expensive operation. And if you start creating new thread instance every time to execute a task, application performance will degrade surely.

Java thread pool manages the pool of worker threads, it contains a queue that keeps tasks waiting to get executed. We can use ThreadPoolExecutor to create thread pool in Java.

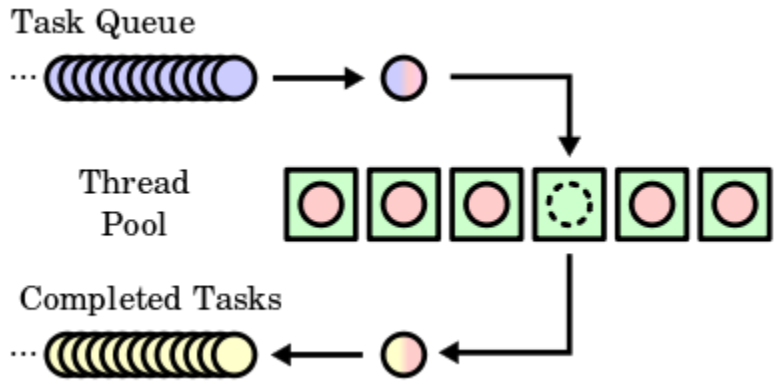
Java thread pool manages the collection of Runnable threads. The worker threads execute Runnable threads from the queue. java.util.concurrent.Executors provide factory and support methods for java.util.concurrent.Executor interface to create the thread pool in java.

Executors is a utility class that also provides useful methods to work with ExecutorService, ScheduledExecutorService, ThreadFactory, and Callable classes through various factory methods.

How thread pool works in java?

Answer: - A **thread pool is a collection of pre-initialized threads**. Generally, the size of collection is fixed, but it is not mandatory. It facilitates the execution of N number of tasks using same threads. If thread are more tasks than threads, then tasks need to wait in a queue like structure (FIFO – First in first out).

When any thread completes its execution, it can pick up a new task from queue and execute it. When all tasks are completed the threads remain active and wait for more tasks in thread pool.



Thread Pool

A watcher keeps watching queue (usually BlockingQueue) for any new tasks. As soon as tasks come, threads again start picking up tasks and execute them.

ThreadPoolExecutor class

Executors class provides simple implementation of ExecutorService using ThreadPoolExecutor, but ThreadPoolExecutor provides much more feature than that. We can specify the number of threads that will be alive when we create ThreadPoolExecutor instance, and we can limit the size of the thread pool and create our **RejectedExecutionHandler** implementation to handle the jobs that can't fit in the worker queue. Signature

public class ThreadPoolExecutor extends AbstractExecutorService

How to create ThreadPoolExecutor?

Answer: -We can create following 5 types of thread pool executors with pre-built methods in `java.util.concurrent.Executors` interface.

1. **Fixed thread pool executor** – Creates a thread pool that reuses a fixed number of threads to execute any number of tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. It is best fit for most off the real-life usecases.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10);
```

2. **Cached thread pool executor** – Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. DO NOT

use this thread pool if tasks are long running. It can bring down the system if number of threads goes beyond what system can handle.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newCachedThreadPool();
```

3. **Scheduled thread pool executor** – Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor)
Executors.newScheduledThreadPool(10);
```

4. **Single thread pool executor** – Creates single thread to execute all tasks. Ute it when you have only one task to execute.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newSingleThreadExecutor();
```

5. **Work stealing thread pool executor** – Creates a thread pool that maintains enough threads to support the given parallelism level. Here parallelism level means the maximum number of threads which will be used to execute a given task, at single point of time, in multi-processor machines.

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newWorkStealingPool(4);
```

ScheduledThreadPoolExecutor

Fixed thread pools or cached thread pools are good when you have to execute one unique task only once. When you need to execute a task, repeatedly N times, either N fixed number of times or infinitively after fixed delay, you should be using **ScheduledThreadPoolExecutor.**

`ScheduledThreadPoolExecutor` provides 4 methods which provide different capabilities to execute the tasks in repeated manner.

6. `ScheduledFuture schedule(Runnable command, long delay, TimeUnit unit)` – Creates and executes a task that becomes enabled after the given delay.
7. `ScheduledFuture schedule(Callable callable, long delay, TimeUnit unit)` – Creates and executes a `ScheduledFuture` that becomes enabled after the given delay.
8. `ScheduledFuture scheduleAtFixedRate(Runnable command, long initialDelay, long delay, TimeUnit unit)` – Creates and executes a periodic action that becomes enabled first after the given `initial` delay, and subsequently with the given `delay` period. If any execution of this task takes longer than its period, then **subsequent executions may start late, but will not concurrently execute**.
9. `ScheduledFuture scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)` – Creates and executes a periodic action that becomes enabled first after the given `initial` delay, and subsequently with the given `delay` period. No matter how much time a long running task takes, there will be a fixed `delay` time gap between two executions.

newFixedThreadPool()

NewFixedThreadPool() Method: - A fixed thread pool can be obtainted by calling the static newFixedThreadPool() method of Executors class.

Syntax

```
ExecutorService fixedPool = Executors.newFixedThreadPool(2);
```

where

- Maximum 2 threads will be active to process tasks.
- If more than 2 threads are submitted then they are held in a queue until threads become available.
- A new thread is created to take its place if a thread terminates due to failure during execution shutdown on executor is not yet called.
- Any thread exists till the pool is shutdown.

newCachedThreadPool() Method

A cached thread pool can be obtainted by calling the static newCachedThreadPool() method of Executors class.

Syntax

```
ExecutorService executor = Executors.newCachedThreadPool();
```

where

- newCachedThreadPool method creates an executor having an expandable thread pool.
- Such an executor is suitable for applications that launch many short-lived tasks.

newScheduledThreadPool() Method

A scheduled thread pool can be obtainted by calling the static newScheduledThreadPool() method of Executors class.

Syntax

```
ExecutorService executor = Executors.newScheduledThreadPool(1);
```

NewSingleThreadExecutor() Method

A single thread pool can be obtainted by calling the static newSingleThreadExecutor() method of Executors class.

Syntax

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

Where newSingleThreadExecutor method creates an executor that executes a single task at a time.

Creating a Custom Thread Pool Executor

In case you want to have more control over the behaviors of a thread pool, you can create a thread pool executor directly from the **`ThreadPoolExecutor`**class instead of the factory methods of the `Executors` utility class.

For example, the `ThreadPoolExecutor` has a general purpose constructor as follows:

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue)
```

You can tweak the parameters to suit your need, as long as you really understand what they mean:

- **`corePoolSize`**: the number of threads to keep in the pool.
- **`maximumPoolSize`**: the maximum number of threads to allow in the pool.
- **`keepAliveTime`**: if the pool currently has more than `corePoolSize` threads, excess threads will be terminated if they have been idle for more than `keepAliveTime`.
- **`unit`**: the time unit for the `keepAliveTime` argument. Can be `NANOSECONDS`, `MILLISECONDS`, `SECONDS`, `MINUTES`, `HOURS` and `DAYS`.
- **`workQueue`**: the queue used for holding tasks before they are executed. Default choices are `SynchronousQueue` for multi-threaded pools and `LinkedBlockingQueue` for single-threaded pools.

Let's see an example. The following code creates a cached thread pool that keeps minimum of 10 threads and allow maximum of 1,000 threads, and idle threads are kept in the pool for 120 seconds:

```
int corePoolSize = 10;
int maxPoolSize = 1000;
int keepAliveTime = 120;
BlockingQueue<Runnable> workQueue = new SynchronousQueue<Runnable>();

ThreadPoolExecutor pool = new ThreadPoolExecutor(corePoolSize,
                          maxPoolSize,
                          keepAliveTime,
                          TimeUnit.SECONDS,
                          workQueue);
pool.execute(new RunnableTask());
```

You can see that when `corePoolSize` = `maxPoolSize` = `1`, we have a single-threaded pool executor.

Summary

10. The `ThreadPoolExecutor` class has four different constructors but, due to their complexity, the Java concurrency API provides the `Executors` class to construct executors and other related objects. Although we can create `ThreadPoolExecutor` directly using one of its constructors, it's recommended to use the `Executors` class.
11. The cached thread pool, we have created above, creates new threads if needed to execute the new tasks, and reuses the existing ones if they have finished the execution of the task they were running, which are now available. The cached thread pool has, however, a disadvantage of constant lying threads for new tasks, so if you send too many tasks to this executor, you can overload the system. This can be overcome using fixed thread pool, which we will learn in next tutorial.
12. One critical aspect of the `ThreadPoolExecutor` class, and of the executors in general, is that you have to end it explicitly. If you don't do this, the executor will continue its execution and the program won't end. If the executor doesn't have tasks to execute, it continues waiting for new tasks and it doesn't end its execution. A Java application won't end until all its non-daemon threads finish their execution, so, if you don't terminate the executor, your application will never end.
13. To indicate to the executor that you want to finish it, you can use the `shutdown()` method of the `ThreadPoolExecutor` class. When the executor finishes the execution of all pending tasks, it finishes its execution. After you call the `shutdown()` method, if you try to send another task to the executor, it will be rejected and the executor will throw a `RejectedExecutionException` exception.
14. The `ThreadPoolExecutor` class provides a lot of methods to obtain information about its status. We used in the example the `getPoolSize()`, `getActiveCount()`, and `getCompletedTaskCount()`methods to obtain information about the size of the pool, the number of threads, and the number of completed tasks of the executor. You can also use the `getLargestPoolSize()` method that returns the maximum number of threads that has been in the pool at a time.
15. The `ThreadPoolExecutor` class also provides other methods related with the finalization of the executor. These methods are:
    - **shutdownNow()**: This method shut downs the executor immediately. It doesn't execute the pending tasks. It returns a list with all these pending tasks. The tasks that are running when you call this method continue with their execution, but the method doesn't wait for their finalization.
    - **isTerminated()**: This method returns true if you have called the `shutdown()` or `shutdownNow()` methods and the executor finishes the process of shutting it down.
    - **isShutdown()**: This method returns true if you have called the `shutdown()` method of the executor.
    - **awaitTermination(long timeout,TimeUnitunit)**: This method blocks the calling thread until the tasks of the executor have ended or the timeout

occurs. The `TimeUnit` class is an enumeration with the following constants: `DAYS`, `HOURS`, `MICROSECONDS` etc.