Thread pools

Concurrent programs generally run a **large number** of **tasks**. Creating thread on demand for each task is not a good approach in terms of **performance** and usage of the **resources** as the thread creation and threads itself is very expensive. There is also a limitation for the maximum number of threads a program can create that is couple of thousands depending on the machine.

In practice thread pool is used for **large-scale applications** that launch a lot of short-lived threads to utilize resources efficiently and increase performance.

Instead of creating new threads when new tasks arrive, a **thread pool** keeps several 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.

We 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.

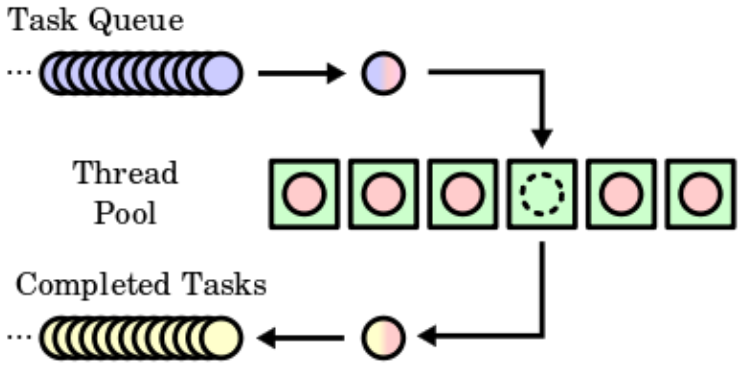
In practice, **thread pool** is used widely in **web servers** where a thread pool is used to serve client's requests. Thread pool is also used in **database applications** where a pool of threads maintaining open connections with the database.

The thread pool is a design pattern used extensively in concurrent programming as a way to avoid creating a new thread each time we want to run a new task.

Implementing a thread pool is a complex task, but we don't have to do it ourselves. The **Java Concurrency API** allows us to easily create and use thread pools without worrying about the details.

The operation of a thread pool

A **thread pool** is a **collection** of **pre-initialized threads**. Generally, the collection size is fixed, but it is not mandatory. It facilitates the execution of N number of taks using the same threads. If there 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 pickup a new task from the queue and execute it. When all tasks are completed, the threads remain active and wait for more tasks in the thread pool.



A **watcher** keeps watching the **queue** for any **new tasks**. As soon as tasks come, threads start picking up tasks and executing them again.

The Executor framework

Java introduced the concept of Executor framework to simplify the process of managing asynchronous tasks. This framework provides a way of managing the creation and execution of threads using thread pools.

The Executor framework provides an **ExecutorService interface** that defines a set of **methods** for managing the **lifecycle** of **threads** in a **thread pool**. By using an **ExecutorService**, we can create, execute, and manage threads without having to manage their lifecycle manually.

Types of thread pools

An Executor Service essentially **creates** and **manages threads** for us to run our **tasks** in **parallel.**

Java supports **five kinds** of **thread pools** that can be used for creating **executor service**:

- **Fixed Thread Pool**: We have fixed number of threads which pick up tasks assigned to it. All tasks are stored in a thread safe blocking queue. It is the best fit for most of real-life use-cases.
- **Work Stealing Pool:** Here, we define the maximum number of threads actively involved in processing tasks submitted to the executor. The queue size can grow and shrink based on the number of available tasks. It can also use multiple queues to avoid multiple threads waiting for tasks from the same queue.
- **Cached Thread Pool**: We do not have a fixed number of threads here. The blocking queue is replaced by a **synchronous queue** which only has space for one task. A new request is stored in the queue while it searches for any available thread. If no thread is available, then it'll create a new thread and add it to the pool. It also has the ability to kill threads which have been idle for more than 60 seconds. Do not use this thread pool if tasks are long-running. It can bring down the system if the number of threads exceed what the system can handle.
- **Scheduled Thread Pool**: It is used for tasks which need to be scheduled after a delay. We can configure a one-off delay, or a periodic schedule, or even a schedule with a fixed delay. A **delay queue** is used here, because of which the order of tasks is not in order.
- **Single Threaded Pool**: It is like a fixed thread pool, but the size of the blocking queue is 1. In this case if the thread is killed because of an exception, a new thread is created. It is used when we want to make sure of the order of execution (sequentially).

ExecutorService

Tasks can be submitted to an ExecutorService for execution. These **tasks** are typically **instances** of **Runnable** or **Callable**, representing units of **work** that need to be executed **asynchronously.**

We can create an executor by using one of several factory methods provided by the **Executors** utility class. Here are some of them:

- **newFixedThreadPool(int n)**: creates an executor with a fixed number of threads in the pool. This executor ensures that there are no more than n concurrent threads at any time. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread becomes available. If any thread terminates due to failure during execution, it will be replaced by a new one. The threads in the pool will exist until it is explicitly shutdown. Use this executor if you and to limit the maximum number of concurrent threads.
- **newCachedThreadPool()**: creates an expandable thread pool executor. New threads are created as needed, and previously constructed threads are reused when they are available. Idle threads are kept in the pool for one minute. This executor is suitable for applications that launch many short-lived concurrent tasks.
- **newSingleThreadExecutor()**: creates an executor that executes a single task at a time. Submitted tasks are guaranteed to execute sequentially, and no more than one task will be active at any time. Consider using this executor if you want to queue tasks to be executed in order, one after another.
- **newScheduledThreadPool(int corePoolSize)**: creates an executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute concurrently.
- **newSingleThreadScheduleExecutor()**: creates a single-threaded executor that can schedule tasks to execute after a given delay, or to execute periodically. Consider using this executor if you want to schedule tasks to execute sequentially.

You can use the factory class of Executors to create ExecutorService implementations.

Executors is a utility class that provides factory methods for creating the implementations of the interface.

```
//Executes only one thread
ExecutorService es = Executors.newSingleThreadExecutor();

//Internally manages thread pool of 2 threads
ExecutorService es = Executors.newFixedThreadPool(2);

//Internally manages thread pool of 10 threads to run scheduled tasks
ExecutorService es = Executors.newScheduledThreadPool(10);
```

We can **execute Runnable tasks** using the following **methods**:

Method	Description
void execute(Runnable task)	Executes the given command at some time in the future.
Future submit(Runnable task)	Submits a runnable task for execution and returns a Future representing that task. The Future's get() method will return null upon successful completion.
Future submit (Runnable task, T result)	Submits a runnable task for execution and returns a Future representing that task. The Future's get() method will return the given result upon successful completion.

We can **execute Callable tasks** using the following **methods**:

Method	Description
Future submit(Callable task)	Submits a value returnin task for execution and returns a Future representing the pending results of the task.
List<Future> invokeAll (Collection tasks)	Executes the given tasks returning a list of Futures holding their status and results when all complete. Notice that result is available only when all tasks are completed. Note that a completed task could have terminated either normally or by throwing an exception.
List<Future> invokeAll(Collection tasks, timeOut, timeUnit)	Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires.

There are **three** simple **steps** involved in creating a **thread pool** and running a job against it:

1. Create an ExecutorService with its own thread pool.
2. Submit Runnable or Callable tasks to the service.
3. Fetch the result of the task, if there is one.

Task queues

When tasks are submitted to **ExecutorService**, if none of the threads in pool are available to process the tasks, they get stored in a queue, below are the different queue options to choose from.

- **Unbounded Queue**: An unbounded queue, such as **LinkedBlockingQueue**, has no fixed capacity and can grow dynamically to accommodate an unlimited number of tasks. It is suitable for scenarios where the task submission rate is unpredictable or where tasks need to be queued indefinitely without the risk of rejection due to queue overflow. However, keep in mind that unbounded queues can potentially lead to memory exhaustion if tasks are submitted at a faster rate than they can be processed.
- **Bounded Queue**: A bounded queue, such as **ArrayBlockingQueue** with a specified capacity, has a fixed size limit and can only hold a finite number of tasks. It is suitable for scenarios where resource constraints or backpressure mechanisms need to be enforced to prevent excessive memory usage or system overload. Tasks may be rejected or handled according to a specified rejection policy when the queue reaches its capacity.
- **Priority Queue**: A priority queue, such as **PriorityBlockingQueue**, orders tasks based on their priority or a specified comparator. It is suitable for scenarios where tasks have different levels of importance or urgency, and higher-priority tasks need to be processed before lower-priority ones. Priority queues ensure that tasks are executed in the order of their priority, regardless of their submission order.
- **Synchronous Queue**: A synchronous queue, such as **SynchronousQueue**, is a special type of queue that enables one-to-one task handoff between producer and consumer threads. It has a capacity of zero and requires both a producer and a consumer to be available simultaneously for task exchange to occur. Synchronous queues are suitable for scenarios where strict synchronization and coordination between threads are required, such as handoff between thread pools or bounded resource access.

Shutting down a Thread pool

In general, the **ExecutorService** will not be automatically destroyed when there is no task to process. It will stay alive and wait for new work to do. To shut the service down we have the shutdown() and shutdownNow() methods.

The **shutdown()** method does not cause immediate destruction of the ExecutorService. It will make the ExecutorService stop accepting new tasks and shut down after all running threads finish their current work.

The **shutdownNow()** method tries to destroy the ExecutorService immediately, but it does not guarantee that all the running threads will be stopped at the same time.

Thread Pool examples

Let's create a task that will take 2 seconds to complete, every time.

```
import java.time.LocalDateTime;

public class TaskRunnable implements Runnable {
    private String name;
    public TaskRunnable(String name){
        this.name=name;
    }
    @Override
    public void run() {
        try {
            Thread.sleep(2000L);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println("Task["+name+"] executed on "+
LocalDateTime.now().toString());
    }
}
```

The following program creates 5 tasks and submits them to the executor queue. The executor uses a single thread to execute all tasks.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MainRunnable {
    public static void main(String[] args) {
        ExecutorService pool= Executors.newSingleThreadExecutor();
        for (int i = 0; i <5 ; i++) {
            Runnable taskRunnable=new TaskRunnable("Task"+i);
            pool.execute(taskRunnable);
        }
        pool.shutdown();
    }
}
```

The following program creates a thread pool with 3 threads where we submitted 5 tasks to the executor service. This executor service takes care of executing the submitted task by the threads from the thread pool. The executor service takes care of creating the thread pool.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class MainRunnable2 {
    public static void main(String[] args) {
        ExecutorService pool= Executors.newFixedThreadPool(3);
        for (int i = 0; i <5 ; i++) {
            Runnable taskRunnable=new TaskRunnable("Task"+i);
            pool.execute(taskRunnable);
        }
        pool.shutdown();
    }
}
```

The following program shows you how to submit a Callable task to an executor. A Callable task returns a value upon completion, and we use the Future object to obtain the value. Here's the code:

```
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.*;

public class MainCallable2 {
    public static void main(String[] args) throws ExecutionException,
InterruptedException {
        ExecutorService pool= Executors.newFixedThreadPool(3);
        Callable<String> callable = new TaskCallable();
        Set<Future<String>> set = new HashSet<Future<String>>();
        for (int i = 0; i <5 ; i++) {
            Future<String> ftr = pool.submit(callable);
            set.add(ftr);
        }
        for(Future<String> future : set){
            System.out.println(future.get());
        }
        pool.shutdown();
    }
}
```