Multi threading, Concurrency, Executor Framework

This repository covers

- Java executor framework and different thread pools
- their key differences
- real world use cases
- code examples

Java Executor Framework

The Executor framework in Java provides several useful interfaces and classes for managing threads:

Interface/Class	Description
Executor Interface	The base interface for all executors that decouples task submission from the mechanics of task execution.
ExecutorService Interface	Extends Executor and adds methods for lifecycle management, such as shutdown and await termination.
ThreadPoolExecutor	A flexible and configurable implementation of ExecutorService for managing a pool of worker threads.
ScheduledExecutorService	Extends ExecutorService and supports tasks that are executed after a delay or periodically.
Executors Utility Class	A helper class to create commonly used ExecutorService implementations like fixed-size thread pools, single-thread executors, etc.

Key Types of Executor Services

Type	Description
FixedThreadPool	`Executors.newFixedThreadPool(int nThreads)` This type of thread pool has a fixed number of threads that will be reused to execute tasks. It is typically used when you know the number of concurrent tasks and want to limit the number of threads.

CachedThreadPool	`Executors.newCachedThreadPool()` A cached thread pool creates new threads as needed, but will reuse previously constructed threads when they are available. It’s useful when you expect a large number of short-lived tasks.
SingleThreadExecutor	`Executors.newSingleThreadExecutor()` A single-threaded executor that ensures that tasks are executed sequentially in a single worker thread. It’s ideal when tasks need to be processed in a strict order, one at a time.
ScheduledThreadPoolExecutor	`Executors.newScheduledThreadPool(int corePoolSize)` Used for scheduling tasks with fixed-rate or fixed-delay execution policies. This is helpful for recurring tasks like periodic monitoring or maintenance tasks.
WorkStealingPool	`Executors.newWorkStealingPool()` This is a special type of pool designed to handle workloads efficiently in a parallel and scalable manner by distributing tasks dynamically among threads.

Executor Services Use Cases

Type	Use Case
FixedThreadPool	**Use case**: Web Server Request Handling **Scenario**: Imagine you’re building a web server where you want to process a set of incoming requests concurrently but limit the number of concurrent threads that can be used at any time to avoid overloading the system. A fixed-size thread pool allows you to ensure that only a fixed number of threads are used, which is crucial for controlling resource consumption.

CachedThreadPool	**Use case**: Handling Short-Lived Tasks in Burst Traffic **Scenario**: You run a system that processes short-lived tasks like image compression, video processing, or analytics that can vary in frequency. During peak periods, such as when many users are uploading content, you need to dynamically create threads to handle bursts of tasks without waiting for idle threads. A cached thread pool is useful here because it creates new threads as needed and reuses idle threads.
SingleThreadExecutor	**Use case**: Sequential Task Execution **Scenario**: A logging system in a server or an application that writes log entries to a single log file. The system needs to ensure that the log entries are written in order, one after the other, without concurrent writes.
ScheduledThreadPoolExecutor	**Use case**: Periodic Task Scheduling **Scenario**: You might need to run tasks at regular intervals or with a delay, like backing up data every night or cleaning up expired sessions from a database. The ScheduledThreadPoolExecutor is well-suited for this kind of periodic task scheduling.
WorkStealingPool	**Use case**: Parallelizing Independent Tasks **Scenario**: Imagine you're running a data analysis system that splits a large dataset into multiple smaller tasks. These tasks might take varying amounts of time to complete, and you want to ensure that all available threads are utilized as efficiently as possible by dynamically redistributing tasks to threads that are idle. **Example**: An image processing pipeline that processes different sections of images. Each section may vary in size or complexity, and you want the tasks to be handled as quickly as possible by "stealing" tasks from other threads that are idle.

Executor Services Use Cases - eCommerce Platform

Type	Use Case
FixedThreadPool	**Use case**: Order Processing and Inventory Management **Scenario**: When customers place orders, the system needs to perform multiple tasks like inventory checks, payment processing, shipping arrangements, and order confirmation. A fixed-size thread pool is ideal for managing these tasks, ensuring that a consistent number of threads are available to process orders without overwhelming the system.
CachedThreadPool	**Use case**: Handling Customer Queries during High Traffic **Scenario**: During high traffic periods like holiday sales, the platform may need to process a large number of customer queries related to product availability, shipping status, and order issues. Since queries come intermittently and vary in frequency, a CachedThreadPool allows dynamic creation of threads to handle queries on-demand, while reusing threads when available.
SingleThreadExecutor	**Use case**: Order Transaction Logging **Scenario**: Certain tasks need to be executed in a strict sequential order to maintain consistency, like logging financial transactions or recording audit trails. Using a SingleThreadExecutor ensures that logs are written in order without concurrency issues. **Example**: Writing transactional logs for every order placed. This ensures that logs are written in sequence, preventing race conditions and ensuring accurate records.
ScheduledThreadPoolExecutor	**Use case**: Promotional Discount Application **Scenario**: In an eCommerce platform like Amazon, you may want to apply promotional discounts at specific intervals or scheduled times, for example, applying a flash sale discount at a specific time or running or running a task every hour to update product prices based on market

	trends.
WorkStealingPool	**Use case**: Parallel Product Search **Scenario**: In an eCommerce platform, product searches are often computationally intensive, especially when customers filter products based on multiple attributes (size, color, price range, etc.). A WorkStealingPool helps distribute these search tasks efficiently among available threads, ensuring fast response times even during peak traffic. **Example**: When a customer searches for a product, the search request is split into smaller sub-tasks (e.g., querying different product categories, sorting results by price, etc.), and these sub-tasks can be processed in parallel by the work-stealing pool, which optimizes thread utilization.

CachedThreadPool vs WorkStealingPool

Feature	CachedThreadPool	WorkStealingPool
Thread Count	There is no fixed number of threads in a cached thread pool. It can grow indefinitely depending on the number of tasks. Threads are created and destroyed dynamically.	Starts with threads equal to the number of available processors (usually `Runtime.getRuntime().availableProcessors()`) and can dynamically increase if needed.
Idle Threads	Threads that are idle for more than **60 seconds** are terminated.	Threads can be idle, but the pool will typically not terminate them. They may steal work from other threads if idle.
Work Stealing	No work stealing. Threads handle tasks assigned to them.	**Work Stealing**: Threads that finish their tasks can “steal” tasks from other threads that are still busy, optimizing work distribution.
Thread Reuse	Threads are reused if idle. If no threads are idle, new threads are created to handle tasks.	Threads are used for parallel task execution and work-stealing. It can dynamically scale, but the number of threads is typically constrained to the number of available processors.

Best For	Short-lived tasks with variable arrival times, such as handling HTTP requests.	Long-running, parallelizable tasks, such as computational tasks or divide-and-conquer algorithms.
Efficiency	Suitable for tasks that arrive at unpredictable rates but do not need to be divided into smaller pieces.	Best suited for tasks that can be divided into smaller sub-tasks, where work stealing helps to balance the load efficiently.
Example Use Cases	Web servers, Dynamic task processing like file uploads, database queries.	Parallel computation tasks (e.g., large-scale data processing, matrix multiplication), Product search in eCommerce platforms (parallel searches by category).

Customizing `ThreadPoolExecutor`

We can directly use the ThreadPoolExecutor class to have finer control over the behavior of your thread pool, such as the core pool size, maximum pool size, keep-alive time, and blocking queue.

```
ThreadPoolExecutor executor = new ThreadPoolExecutor(
        2,  // corePoolSize
        4,  // maximumPoolSize
        60L, // keep-alive time
        TimeUnit.SECONDS,
        new LinkedBlockingQueue<>(10));  // Task queue with a capacity of 10

for (int i = 0; i < 20; i++) {
    executor.submit(() -> {
        System.out.println(Thread.currentThread().getName() + " is processing a task");
    });
}
executor.shutdown();
```

Repo link here - https://github.com/pavanuppuluri/Multi-Threading-Concurrency-Executor-Framework

Thank you,
Happy coding !!!