# Chapter 3. Thread Pools

## Thread Pools

The Executor Framework represents a significant shift in how developers manage threads within their applications. By introducing the Executor interface, the framework provides an abstraction layer over the raw thread management, offering developers various strategies for handling concurrent execution without the need to manually manage thread lifecycles.

In this diagram, you can see a thread pool with 6 running threads and a queue of tasks. Only 6 tasks are able to run simultaneously. When the task is completed, another one may take its place.

A diagram of a pool

Description automatically generated

1. **Task Queue**: This represents a queue of tasks waiting to be executed. Tasks are typically encapsulated in Runnable or Callable objects, which define the unit of work to be done.
2. **Thread Pool**: This is a group of pre-instantiated reusable threads. Each thread in the pool waits for a task to execute.
3. **Completed Tasks**: Once a task is executed by a thread from the pool, it moves to the completed tasks, signifying its completion.
4. **Executor**: The Executor is the conductor of this process, taking tasks from the queue and assigning them to available threads in the pool.

The diagram’s circular flow demonstrates the continuous process of task execution. A task is taken from the Task Queue, passed to the Thread Pool, and upon completion, it transitions to the Completed Tasks. A highlighted thread signifies an active execution of a task, showing the Executor’s role in facilitating this process.

## Why do we need a Thread Pool?

A thread pool is a collection of pre-initialized threads that stand ready to execute tasks. Using a thread pool offers several advantages over spawning a new thread for each task, and these benefits are particularly relevant in high-throughput or server-side programming where resource management is critical.

Here are some key reasons why we use thread pools:

1. **Resource Management**: Creating and destroying threads on the fly is resource-intensive as it involves overhead in terms of memory and time for thread creation and teardown. Thread pools help by reusing a fixed number of threads to handle tasks, thereby reducing the overhead.
2. **Control Over Thread Count**: Spawning too many threads can lead to high memory consumption and, in extreme cases, can exhaust system resources leading to poor performance or application failure. Thread pools allow you to control the maximum number of threads that your application will use, preventing resource exhaustion.
3. **Improved Performance**: Since threads are already created in a thread pool, the latency associated with thread creation does not affect task start times. Tasks can start running immediately if a thread is available, which can lead to better application performance.
4. **Task Queue Management**: Thread pools often come with a blocking queue that holds tasks waiting to be executed. This not only provides a way to manage workload but also enables smooth handling of incoming task bursts that might occur during peak loads.
5. **Reduced System Load**: By limiting the number of concurrent threads, thread pools help maintain a stable system load, preventing context switching overhead and potential CPU thrashing that can occur when too many threads are competing for CPU time.
6. **Load Smoothing**: Thread pools can help in smoothing the load on the system by queuing excessive requests when all threads are busy instead of overwhelming the system all at once.
7. **Fine-Tuned Resource Allocation**: Different types of thread pools (fixed, cached, scheduled) allow developers to choose the most appropriate type of resource allocation for their specific use case, whether it's handling a predictable load or dynamically adjusting to varying workloads.
8. **Graceful Application Shutdown**: Thread pools facilitate a graceful shutdown of applications by providing mechanisms to interrupt currently executing tasks gently, complete pending tasks, and not accept new ones.

## Executing tasks with Thread Pool

The ExecutorService is an interface in the Java Concurrency API that defines a high-level means for task execution. It can manage a pool of threads and offers various methods to coordinate their lifecycle and task submission. One such method is **execute()**, which is used to submit tasks for execution without expecting any return value.

The **execute()** method is one of the simplest ways to submit a task to an ExecutorService. This method takes a **Runnable** object, which defines the code to be executed. Here’s a look at how this method is used in practice:

*// Use of execute() method with anonymous Runnable*executorService.execute(new Runnable() {  
 public void run() {  
 System.*out*.println("Asynchronous task");  
 }  
});  
  
*// Use of execute() method with lambda expression*executorService.execute(() -> System.out.println("Asynchronous task"));

The order in which tasks are executed is not guaranteed and depends on the thread pool's state and the type of ExecutorService used.

Since **execute()** does not return a result, any exceptions thrown during task execution are not returned to the caller. Instead, they should be handled within the **Runnable** itself.

## Thread Pool types

Let’s look at how to instantiate various types of the Thread Pool.

ExecutorService executorService1 = **Executors.newSingleThreadExecutor();**

This creates an ExecutorService that manages a single thread. The single thread will execute submitted tasks sequentially, one at a time. If the thread dies due to an exception, a new one will take its place for subsequent tasks. This type of executor ensures that no two tasks are executed concurrently.

ExecutorService executorService2 = **Executors.newFixedThreadPool(10);**

This line of code creates a thread pool with a fixed number of threads (10 in this case). If all threads are actively executing tasks, new tasks will wait in a queue until a thread becomes available. This type of executor is useful when you want to limit the number of concurrent threads to a fixed number, often to limit resource usage.

ExecutorService executorService3 = **Executors.newScheduledThreadPool(10);**

This creates a thread pool that can schedule commands to run after a given delay or to execute periodically. The '10' here indicates that it can support up to 10 concurrent threads. This is particularly useful for tasks that need to occur on a schedule (e.g., periodic garbage collection, periodic updates, etc.).

ExecutorService executorService4 = **Executors.newCachedThreadPool();**

This line of code creates a thread pool that creates new threads as needed but will reuse previously constructed threads when they are available. It’s a dynamic thread pool that grows as needed. This type of executor is suitable for applications that launch many short-lived tasks.

This table demonstrates the differences between thread pool types and their typical use cases:

|  |  |  |
| --- | --- | --- |
| Thread Pool Type | Use Case | Characteristics |
| Single-Threaded Executor | Suitable for tasks that need to be executed sequentially to ensure thread safety without using synchronized or other locking mechanisms. | Tasks are executed sequentially. |
| Fixed Thread Pool | Useful when you want to limit the number of threads running in parallel, to manage resource usage. | Fixed number of threads.All threads are created upfront and reused for executing tasks. Excess tasks are queued until a thread becomes available. |
| Scheduled Thread Pool | Suitable for tasks that need to be executed periodically or after a fixed delay: timers, reminders, or scheduling tasks | Can schedule tasks to run after a given delay or to execute repeatedly with a fixed interval or delay. |
| Cached Thread Pool | Best for applications with many short-lived asynchronous tasks.Useful when tasks are lightweight and you want to start them without delay, without worrying about thread creation overhead. | Creates new threads as needed but reuses previously constructed threads when they are available.Idle threads are terminated after 60 seconds of inactivity |

## Future interface

The Future interface provides a way to represent the result of an operation that is performed asynchronously. It serves as a handle through which we can retrieve the outcome of the computation once it's complete.

The Future interface defines several methods that allow for the management and retrieval of the computation:

**isDone()**: This method returns **true** if the computation has been completed, whether it finished normally, was canceled, or terminated with an exception.

**get()**: Retrieves the result of the computation when it is done, blocking if necessary until the computation is complete.

**get(long timeout, TimeUnit unit)**: Retrieves the result of the computation if it is done within the given timeout period, otherwise, it throws a **TimeoutException**.

**cancel(boolean mayInterruptIfRunning)**: Attempts to cancel the execution of the task. The **mayInterruptIfRunning** parameter specifies whether the thread executing the task should be interrupted in an attempt to stop the task.

To submit a task that can be controlled with a Future interface, we should use Callable instead of Runnable. The Callable interface is similar to Runnable but it can return a value and throw exceptions.

The **Callable** interface represents a task that returns a result. Unlike **Runnable**, a **Callable** can return a value and can throw a checked exception. The **Callable** interface is a parameterized type, meaning you specify the type of value it returns when you declare a **Callable**.

public interface Callable<V> {  
 V call() throws Exception;  
}

**Example: submitting a callable task**

ExecutorService executor = Executors.newSingleThreadExecutor();  
  
Future<String> future = executor.submit(new Callable<String>() {  
 @Override  
 public String call() throws Exception {  
 *// Simulate long computation* Thread.*sleep*(2000);  
 return "Result of the computation";  
 }  
});  
  
executor.shutdown(); *// Always remember to shut down the executor*

This is how we can retrieve the result of the call:

try {  
 *// Blocks until the result is available or the thread is interrupted* String result = future.get();  
 System.out.println(result);  
} catch (InterruptedException | ExecutionException e) {  
 e.printStackTrace();  
}

The **get()** method blocks until the computation is finished and the result is available. If the computation throws an exception, it will be wrapped in an ExecutionException.

The **get()** method with timeout parameters is used to avoid blocking indefinitely. It allows specifying the maximum time to wait for the result.

try {  
 *// Blocks for only 1 second. After that, a TimeoutException is thrown* String result = future.get(1, TimeUnit.SECONDS);  
 System.out.println(result);  
} catch (InterruptedException | ExecutionException | TimeoutException e) {  
 e.printStackTrace();  
}

The **cancel()** method attempts to cancel the execution of the task. If the task has already been completed, has already been canceled, or could not be canceled for some other reason, this method will return false.

*// Specify if the thread should be interrupted*boolean mayInterruptIfRunning = true;

boolean isCancelled = future.cancel(mayInterruptIfRunning);  
  
if(isCancelled) {  
 System.out.println("The task was cancelled");  
} else {  
 System.out.println("The task could not be cancelled");  
}

The **isDone()** method is a non-blocking call that can be used to check if the computation is complete.

if (future.isDone()) {  
 System.out.println("The computation is done.");  
}

## Shutting down the Executor

Shutting down an executor is an important step to ensure that your Java application terminates cleanly. There are two primary methods provided by the **ExecutorService** to shut down an executor:

1. **shutdown()**: This method initiates a graceful shutdown where previously submitted tasks are executed, but no new tasks will be accepted. It does not wait for previously submitted tasks to complete execution. After calling **shutdown()**, the executor moves to a state where it will complete all existing tasks but will not accept any new ones.
2. **shutdownNow()**: This method attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution. This method does not wait for actively executing tasks to terminate. It tries to stop them immediately, typically by interrupting the threads. However, it does not guarantee that all actively executing tasks will be terminated instantly as it depends on the tasks' response to interruptions.

**Reasons to Shutdown Executors:**

* **Resource Management**: Each thread consumes system resources. Properly shutting down an executor helps to release these resources when they are no longer needed.
* **Application Termination**: To ensure that your application terminates promptly, you need to shut down all executors. If executors are left running, they may continue to keep the JVM alive since they are non-daemon threads.
* **System Stability**: By shutting down executors, you help the system to maintain stability by freeing up system resources, which can then be used for other processes.
* **Control**: Shutting down executors allows you to control when your application can accept new tasks and when it should start to wind down.

**Ways to Shutdown Executors:**

1. **Await Termination after Shutdown**: If you want to wait for the executor to finish executing all tasks before continuing with the shutdown process, you can use **awaitTermination()** after **shutdown()**. Here's an example:

executorService.shutdown(); // Disable new tasks from being submitted  
  
// Wait a while for existing tasks to terminate  
if (!executorService.awaitTermination(60, TimeUnit.SECONDS)){  
 executorService.shutdownNow(); // Cancel currently executing tasks  
}

In the code above, **shutdown()** will start the shutdown process, and **awaitTermination()** will block until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

1. **Immediate Shutdown**: If you want to stop the executor service immediately and you're not interested in waiting for tasks to finish, you can call **shutdownNow()**. This is a more abrupt way of shutting down and should be used when you need to stop all tasks right away, for example, in case of an emergency or when stopping the application is more important than completing the tasks.

## Virtual Threads

Virtual threads, introduced in Java as a part of Project Loom, aim to dramatically improve the scalability of applications by making it easier to write, maintain, and observe high-throughput concurrent applications. They represent a significant evolution in Java's concurrency model, addressing the limitations of the traditional thread model which is tightly coupled with the underlying operating system's threads. This article explores when to use virtual threads, and their use cases, and provides usage examples to illustrate their benefits and applications.

The key benefits of virtual threads include:

* **High Scalability**: Virtual threads are designed to be **lightweight**, allowing **millions of concurrent threads** to be spawned with **minimal overhead**, compared to traditional threads which are resource-intensive.
* **Simplified Concurrency Model**: They simplify the programming model for writing concurrent applications. It’s especially important as a simpler alternative to asynchronous and reactive programming, with the same or even better performance.
* **Improved Resource Utilization**: Applications can make better use of system resources by efficiently managing I/O operations, waiting tasks, and CPU-bound tasks, leading to improved throughput and performance.

**When to Use Virtual Threads:**

* **Highly Concurrent IO-bound Applications:** Virtual threads are ideal for applications that handle a high number of simultaneous IO operations, such as web servers, microservices, or asynchronous processing systems.

**Usage Examples of Virtual Threads:**

Starting a virtual thread:

Thread.startVirtualThread(() -> {  
 System.out.println("This is a virtual thread");  
});

Using Virtual Threads with Executors:

ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();  
executor.submit(() -> {  
 *// Task for virtual thread*});

### Virtual Threads implementation

This diagram illustrates the execution model of virtual threads in Java, and how they are multiplexed over a set of carrier threads.

A screenshot of a computer

Description automatically generated

**Carrier thread** can be thought of as a **regular OS thread**. The carrier thread executes runnable tasks, denoted by "R". These tasks are the virtual threads (VT1, VT2, VT3), which are lightweight threads managed by the Java Virtual Machine (JVM) rather than the operating system.

**Virtual threads can be in different states**:

* + **Runnable**: The virtual thread is actively executing on the carrier thread.
  + **Waiting**: The virtual thread is not currently executing because it's waiting for some condition to be met or for some I/O operation to complete.
  + **Blocked**: The virtual thread is prevented from running due to a lock or monitor that it's trying to acquire but can't because another thread holds the lock.

**State Transitions**. The transitions between states show how virtual threads can be suspended and resumed. When a virtual thread is waiting or blocked, the carrier thread can pick up another virtual thread and start running it. This is an efficient use of the carrier threads and allows for high concurrency, especially in I/O-bound applications.

**The key takeaways from this diagram are:**

* Virtual threads are **managed by the JVM** and are mapped onto a smaller number of carrier threads.
* Carrier threads can **execute tasks from different virtual threads**, switching between them as needed.
* This model allows a **large number of virtual threads to be used efficiently**, even if the number of physical or carrier threads is much smaller, thereby improving application scalability and performance in I/O-bound operations.

### Thread mechanisms comparison

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Virtual Threads** | **Threads (Traditional)** | **Thread Pool** |
| **Implementation** | Built on top of carrier threads, **managed by the JVM** | **OS-level threads**, directly managed by the operating system | A pool of **reusable** threads managed by the JVM |
| **Overhead** | **Low** memory and creation **overhead** | **High** memory and creation **overhead** | **Varies**, but generally lower overhead than creating new threads |
| **Scalability** | Designed to handle **millions of concurrent tasks** efficiently | **Limited scalability** due to higher resource costs | **Improves scalability over individual threads** but can be less efficient than virtual threads |
| **Use Case** | **High-throughput IO** operations, microservices, asynchronous processing | **CPU-intensive tasks**, applications with limited concurrency needs | **Repeated or similar tasks**, applications with moderate concurrency needs |
| **Lifecycle Management** | **Managed by the JVM**, less control for developers | Manual management by developers | Managed by the JVM, some control through pool parameters |
| **Suitability for IO-bound Tasks** | **Highly suitable** due to lightweight nature and efficient scheduling | Less suitable due to blocking and higher resource costs | Suitable, but efficiency depends on pool size and task nature |
| **Suitability for CPU-bound Tasks** | **Less suitable**, as they are not designed to replace computational threads | **Highly suitable** | **Suitable**, with efficiency depending on proper pool size and configuration |
| **Context Switching** | **Minimal overhead** due to managed scheduling | **Higher overhead** due to OS-level context switching | **Reduced overhead** compared to individual threads, but still present |
| **API** | New API in Project Loom (Java Virtual Thread API) | Standard Java Thread API | Executors framework in Java |