ThreadPoolExecutor

Go through the most comprehensive and complete guide to working with the ThreadPoolExecutor class.

Overview

In general, a thread pool is a group of threads instantiated and kept alive to execute submitted tasks. Thread pools can achieve better performance and throughput than creating an individual thread per task by circumventing the overhead associated with thread creation and destruction. Additionally, system resources can be better managed using a thread pool, which allows us to limit the number of threads in the system.

Generally the use of the `ThreadPoolExecutor` class is discouraged in the favor of thread pools that can be instantiated using the `Executors` factory methods. These thread pools come with pre-configured settings that are commonly used in most scenarios, however, the `ThreadPoolExecutor` comes with several knobs and parameters that can be fine-tuned to suit unusual use-cases. Before we delve into the `ThreadPoolExecutor` class, we'll list some of the thread pools provided by the `Executors` factory methods:

- `Executors.newCachedThreadPool()` - (unbounded thread pool, with automatic thread reclamation)
- `Executors.newFixedThreadPool(int)` (fixed size thread pool)

pools that can be instantiated using the `Executors` factory methods. These thread pools come with pre-configured settings that are commonly used in most scenarios, however, the `ThreadPoolExecutor` comes with several knobs and parameters that can be fine-tuned to suit unusual use-cases. Before we delve into the `ThreadPoolExecutor` class, we'll list some of the thread pools provided by the `Executors` factory methods:

- `Executors.newCachedThreadPool()` - (unbounded thread pool, with automatic thread reclamation)
- `Executors.newFixedThreadPool(int)` (fixed size thread pool)
- `Executors.newSingleThreadExecutor()` (single background thread)
- `Executors.newScheduledThreadPool(int)` (fixed size thread pool supporting delayed and periodic task execution.)

Example

Let's consider the constructor that takes-in the most arguments to instantiate the `ThreadPoolExecutor` class:

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

We'll shortly study how each of these arguments impact the behavior of the executor. First, we'll start with a simple example program that demonstrates the use of the `ThreadPoolExecutor` class:

`corePoolSize` and `maximumPoolSize`

The arguments `corePoolSize` and the `maximumPoolSize` together determine the number of threads that get created in the pool. The workflow is as follows:

- When the pool has less than `corePoolSize` threads and a new task arrives, a new thread is instantiated even if other threads in the pool are idle

thread is instantiated even if other threads in the pool are idle.

- When the pool has more than `corePoolSize` threads but less than `maximumPoolSize` threads then a new thread is only created if the queue that holds the submitted tasks is full.
- The maximum number of threads that can be created is capped by `maximumPoolSize`.

Both `corePoolSize` and `maximumPoolSize` can be changed dynamically after construction of the pool instance. Note that a newly instantiated pool creates core threads only when tasks start arriving in the queue. However, this behavior can be tweaked by invoking one of the `prestartCoreThread()` or `prestartAllCoreThreads()` methods, which is a good idea when creating a pool with a non-empty queue.

Setting `corePoolSize` equal to `maximumPoolSize`

If we set `corePoolSize` equal to `maximumPoolSize` we effectively create a fixed size thread pool.

Setting `maximumPoolSize` to an arbitrary high value

Setting `maximumPoolSize` to an unbounded value such as `Integer.MAX_VALUE` allows the pool to accommodate an arbitrary number of concurrent tasks.

Keep-alive

A thread pool will eliminate threads in excess of `corePoolSize` after `keepAliveTime` has elapsed. The `unit` argument specifies the `TimeUnit` for the passed-in value of `keepAliveTime`, which. can be milliseconds, minutes, hours etc.

ThreadFactory

The pool creates new threads using a `ThreadFactory`. The user has the choice to pass-in a factory of her own choice or let the `ThreadPoolExecutor` class choose the default. Usually, you would pass-in a thread factory argument if you want to change the thread name, thread group, priority, daemon status etc.

Queuing

The `ThreadPoolExecutor` takes in a `BlockingQueue` as a parameter in its constructor. The queue is used to hold tasks submitted to the executor. The queue works in tandem with the pool's thread size.

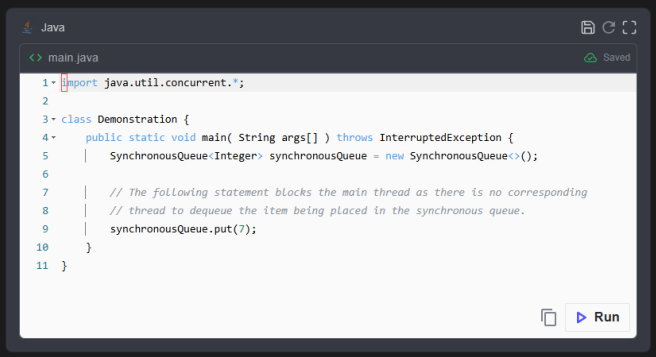
- If fewer than `corePoolSize` threads are running when a new task is submitted, the executor prefers adding a new thread rather than queuing the task. Remember:
- If `corePoolSize` or more threads are running, the executor prefers queuing the task than creating a new thread.
- If the queue is full and creating a new thread would exceed `maximumPoolSize` the submitted task is rejected by the executor. We'll shortly explain the various policies that govern task rejection.

Queuing Strategies

The choice of the queue we pass-in determines the queuing strategy for the executor. The queuing strategies are:

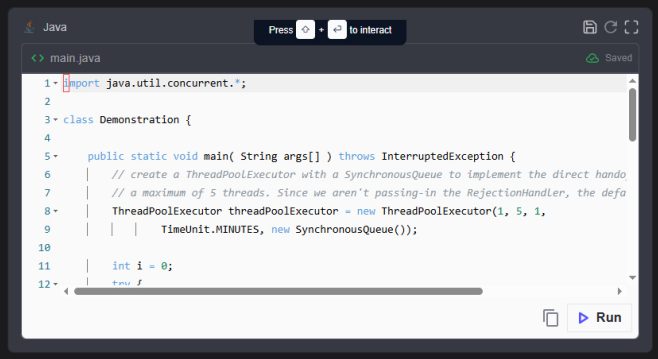
Direct handoffs

Direct handoff design involves an object running in one thread syncing up with an object running in another thread to hand off a piece of information, event or task. The `SynchronousQueue` class can be used for implementing the direct handoff strategy. The `SynchronousQueue` doesn't have an internal capacity (not even 1) and an item can only be inserted in the queue if another thread is simultaneously removing it. The widget below demonstrates that if we add an item to a `SynchronousQueue` without another thread removing the item, the thread making the insert simply blocks. The execution for the widget below times out as the main thread gets blocked.



Given this behavior it is obvious that if the `ThreadPoolExecutor` is initialized with

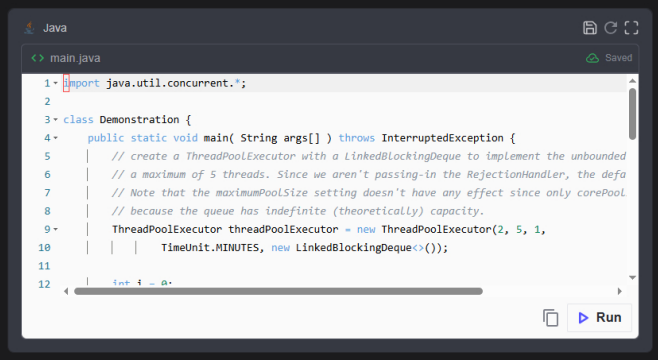
`SynchronousQueue`, each new task submitted to the `ThreadPoolExecutor` is handed off by the queue to one of the pool threads for execution, i.e. the queue doesn't hold any tasks. However, if tasks submitted exceed `maximumPoolSize` the queue doesn't hold any tasks and if no free threads are available then the submitted tasks are rejected. Consider the example program below, where we set the `maximumPoolSize` to 5 and then attempt to submit 50 tasks. Each task sleeps for 1 second so the entire pool is hogged after 5 tasks are submitted. The 6th task when submitted has the executor throw the `RejectedExecutionException` to indicate that the task can't be accepted for execution by the thread pool. The tasks can't be queued by the `SynchronousQueue` and if no free thread is available a new one must be created but if the number of threads has already reached the maximum allowed number then the task is rejected.



Folks with background in CSP or ADA would find the `SynchronousQueue` similar to *rendezvous channels*. Using the direct handoff strategy requires that the maximum allowed threads for a thread pool should be unbounded e.g. `Integer.MAX_VALUE` to avoid tasks being rejected. However, this setting entails that the number of threads in the system can grow to be very large if tasks are submitted at a rate faster than they can be processed at. Direct handoff policy is useful when handling sets of requests that might have internal dependencies as lockups are avoided.

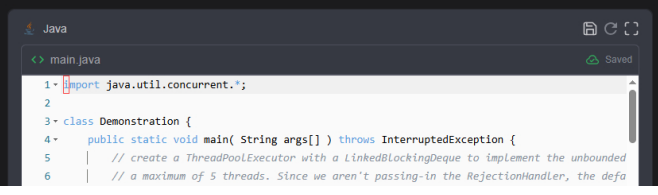
Unbounded queues

If we use a queue such as the `LinkedBlockingQueue` without a predefined capacity, the queue can arbitrarily grow in size. The consequence is that tasks get added to the queue if all the `corePoolSize` threads are busy. Interestingly, the `maximumPoolSize` setting takes no effect and only `corePoolSize` threads are ever created. Submitted tasks sit in the queue waiting for execution. Using this strategy we can see the queue size grow indefinitely in contrast to the direct handoff approach in which the number of threads can grow indefinitely. Consider the program below that uses the `LinkedBlockingQueue` without a defined capacity and only 5 tasks the same as the `corePoolSize` execute at any time. The rest pile-up in the queue.



The output of the above program is interesting to observe. Note, that only two threads, which is also the `corePoolSize`, ever execute the submitted tasks. At a time, only two tasks are executed while the rest queue-up and the queue grows without any bounds.

We can also define a capacity when passing in the `LinkedBlockingQueue`. In that scenario the executor can reject newly submitted tasks if the queue has reached capacity and `maximumPoolSize` threads have been created and are busy executing other tasks. Note that with a defined capacity queue the setting `maximumPoolSize` becomes effective.



```
    // The queue has a defined capacity so the setting maximumPoolSize does take effect.
    ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(2, 5, 1,
            TimeUnit.MINUTES, new LinkedBlockingDeque<>(5));

    int i = 0;
    try {
```

Run

In the above program the 11th task gets rejected and the first 10 get executed. This makes sense because the thread pool has a maximum of 5 threads, all of which start working on the first five submitted tasks. Thereafter the next 5 threads get queued-up in the queue which has a maximum capacity of 5. When the 11th task gets submitted, all five tasks are busy executing the first 5 tasks and the queue is full, therefore the 11th task is rejected.

Bounded queues

The astute reader can deduce from the previous discussion that a tradeoff between the maximum threads and queue sizes exists. Constraining one allows the other to grow unbounded. The middle of the spectrum is to define a queue with a certain capacity and also set a limit on the maximum number of threads. Using a bounded queue with a finite maximum pool size helps prevent resource exhaustion in the system. However, using large queue sizes and small pools minimizes CPU usage, OS resources, and context-switching overhead, but can lead to artificially low throughput. In systems where threads occasionally block for I/O, a system may be able to schedule time for more threads than you otherwise allow. Using smaller queues generally requires larger pool sizes, which keeps CPUs busier but may encounter unacceptable scheduling overhead, which also decreases throughput.

Thus choosing a queuing strategy will involve looking at the particular characteristics of your system and picking an option suitable for your use case.

Queue Manipulation

We can access the queue using the `getQueue()` method, however, use of this method for purposes other than debugging and monitoring is discouraged. Two other methods, namely, `remove(Runnable)` and `purge()` are available to assist in storage reclamation when large numbers of queued tasks become cancelled.

Task Rejection

If the executor becomes overwhelmed with tasks, it can reject newly submitted tasks. This occurs when the executor has a defined maximum pool size and a defined queue capacity and both resources hit their limits. Tasks can also be rejected when they are submitted to an executor that has already been shutdown. There are four different policies that can be supplied to the executor to determine the course of action when tasks can't be accepted any more. These policies are represented by four classes that extend the `RejectedExecutionHandler` class. The executor invokes the `rejectedExecution()` method of the supplied `RejectedExecutionHandler` when a task is intended for rejection. We discuss them below:

`ThreadPoolExecutor.AbortPolicy`

The abort policy simply throws the runtime `RejectedExecutionException` when a task can't be accepted. The previous widget demonstrates the use of the `ThreadPoolExecutor.AbortPolicy` when tasks get rejected if they can't be accommodated by the thread pool.

`ThreadPoolExecutor.CallerRunsPolicy`

According to this policy the thread invoking the `execute()` method of the executor itself runs the task. This mechanism serves to throttle the rate at which tasks are submitted as the submitting threads themselves end up executing the tasks they submit.

The previous program is reproduced with the change of the `RejectionHandler` to `ThreadPoolExecutor.CallerRunsPolicy`:

Java

main.java

Saved

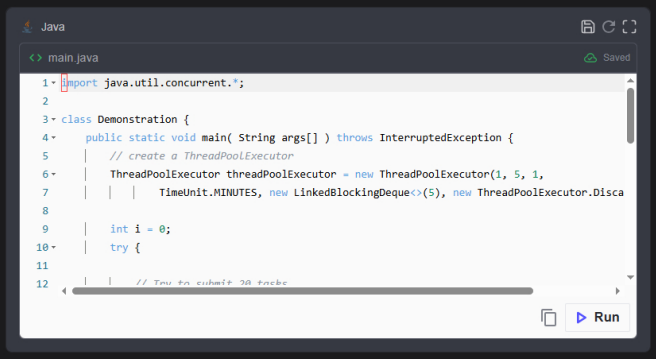
```
import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] )  throws InterruptedException {
        // create a ThreadPoolExecutor
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(1, 5, 1,
                TimeUnit.MINUTES, new LinkedBlockingDeque<>(5), new ThreadPoolExecutor.Calle

        int i = 0;
        try {

            // Try to submit 20 tasks
```

Run

Notice that in the output of the above program, when the thread pool can't accept any more tasks, the main thread that is submitting the tasks, is itself pulled-in to execute the submitted task. Consequently, the submission of new tasks slows down as the main thread now executes the task itself.

`ThreadPoolExecutor.DiscardPolicy`

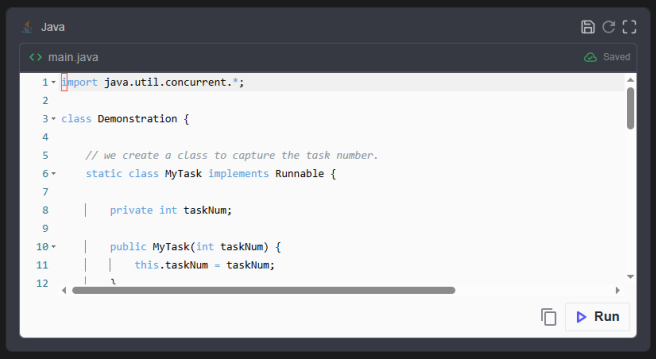
A task that can't be executed is simply dropped. In the program below, the rejection policy has been changed to `ThreadPoolExecutor.DiscardPolicy`:



`ThreadPoolExecutor.DiscardOldestPolicy`

When a task can't be accepted for execution, this policy causes the oldest unhandled request/task to be discarded and then the execution is retried for the just submitted task. In case the executor is shutdown then the task is simply discarded.

To demonstrate the `ThreadPoolExecutor.DiscardOldestPolicy` we'll create a class `MyTask` that extends `Runnable` to capture the order in which tasks are submitted to the thread. The program is similar to the previous examples and changes the rejection policy to `ThreadPoolExecutor.DiscardOldestPolicy`.



If you examine the output of the program above, you'll notice that the tasks that get executed aren't numbered sequentially. As threads become busy and the queue fills-up, the policy dictates to discard the oldest submitted task in the queue. Note that the last 5 tasks numbered 20, 19, 18, 17 and 16 are always executed as they are submitted in the end and there are no task submissions after them that may cause them to be discarded.

Shutting down

The `ThreadPoolExecutor` can be shutdown by invoking the `shutdown()` method. A pool that is no longer referenced and has no remaining threads will shutdown automatically. In case `shutdown()` isn't invoked then the configuration must make sure that unused threads eventually die by setting the `corePoolSize` to zero and choosing an appropriate `keepAliveTime` value. Another option if `corePoolSize` is set to a non-zero is to use the `allowCoreThreadTimeOut(boolean)` method to have the time out policy apply to both core and non-core threads.

Hooks

The `ThreadPoolExecutor` class also exposes protected overridable methods that derived classes can override. For instance: The `beforeExecute(Thread, Runnable)` and `afterExecute(Runnable, Throwable)` methods are called are called before and after execution of each task. These can be used to manipulate the execution environment; for example, reinitializing ThreadLocals, gathering statistics, or adding log entries etc. Similarly, the method `terminated()` can be overridden to perform any special processing that needs to be done once the Executor has fully terminated. Note that if any of the `BlockingQueue` methods, callbacks or hooks throw an exception, threads in the pool may fail, terminate abruptly and possibly get replaced.

Conclusion

In summary, for the vast majority of use-cases the `Executors` factory method should be used for instantiating thread pools, however, there may be instances where fine-grained control is desired and the `ThreadPoolExecutor` class is a good fit for such scenarios.

← Back lesson

An Example: Timer vs ScheduledThreadPool

Completed Next →

Callable Interface