Implicit couplings between tasks and execution policies

Executor framework decouples task submission from task execution. But there are many tasks that require specific execution policies:

- **Dependent tasks** create constraints on the execution policy that must be carefully managed to avoid liveness problems.
- **Tasks that exploit thread confinement and work in single-threaded executors**. Single-threaded pools make stronger promises about concurency. They guarantee tasks are not executed concurrently, which allows to relax thread safety. Objects can be confined to the thread and accessed without syncrhonization. This forms an implicit coupling between the task and execution policy.
- **Response-time-sensitive tasks**. GUI applications are sensitive to response time. Implicit coupling is the restriction not to submit long running tasks or many tasks to executor.
- **Tasks using ThreadLocal**—threads can be reused between tasks. Hence **ThreadLocal** can be used in thread pools only when lifetime of an object is restricted to a single task. Should not be used in pool threads to communicate between threads.

Thread pools work best when tasks are homogeneous and independent

- mixing long-running and short-running tasks risks clogging the pool
- submitting tasks that depend on each other risks deadlock

Some tasks require or preclude a specific execution policy

- tasks that depend on other tasks require that the thread pool be large enough that tasks are never queued or rejected
- tasks that exploit thread confinement require serial execution

Thread starfvation deadlock

If tasks that depend on other tasks execute in a thread pool, they can deadlock

- in a single-threaded executor a task that submits another task to the same executor and waits for its result always deadlocks
- same thing can happen in larger pools if all threads are executing tasks that are blocked waiting for other tasks still on the work queue

In addition to any explicit bounds on the size of a thread pool, there may also be implicit limits because of constraints on other resource. If all tasks need a JDBC connection and there are only 10 available then it is as if thread pool had size of 10.

Long running tasks

Even if deadlock is not a possibility thread pool can be clogged with long-running tasks, increasing service time for short tasks. The responsiveness may suffer.

- use timed versions of methods
- tune the size of the pool

Sizing thread pools

Common rules:

- Sizes should be provided by a configuration mechanism or computed dynamically by consulting **Runtime.getRuntime().availableProcessors**.
- If a thread pool size is too big, threads compete for scarce CPU and memory resources.
- If a thread pool size is too small, throughput suffers as processors go unused despite available work.
- Understand your computing environment (how many processors? memory?)
- Understand the nature of your tasks (mostly computation, IO, or both?), do they need a scarce resource (JDBC connection)? Different categories of tasks with differents behaviours consider using multiple threads where each can be configured accordingly to its workload.

How to compute thread pool size:

- For compute-intensive tasks, an Ncpu processor achieves optimum utilization with a thread pool of **Nthreads = Ncpu+1** threads (even compute-intensive threads ocasionally take a page fault or pause for some other reason, and extra runnable thread prevents CPU cycles from going unused)
- For IO intensive or any other blocking tasks: **Nthreads = Ncpu** Ucpu / C*, where **C in [0; 1]** is the ratio of computing time, **Ucpu in [0; 1]** is the desired load.
- **Ncpu = Runtime.getRuntime().availableProcessors()**
- Other resources can contribute to sizing constraints are memory, file handles, socket handles, database connections. Calculate how much each task requires, divide total amount available by that number. This will an upper bound on the pool size.

Configuring ThreadPoolExecutor

ThreadPoolExecutor provides the base imlementation for the executors returned by the **newCachedThreadPool**, **newFixedThreadPool**, **newScheduled ThreadExecutor** factories in **Executors**.
If the defaul execution policy does not meet your needs, you can instantiate a **ThreadPoolExecutor** through its constructor and customize it as you see fit.

Thread creation and teardown

Parameters:

- core pool size — the target size, the implementation attempts to maintain the pool at this size even when there are no tasks to execute and will not create more threads than this unless the work queue is full

- maximum pool size — the upper bound on how many pool threads can be active at once
- keep-alive time — a thread that has been idle for longer than keep-alive time becomes a candidate for reaping and can be terminated if the current pool size exceeds the core size

Notes:

- When a **ThreadPoolExecutor** is initially created, the core threads are not started immediately, but instead as tasks are submitted, unless you call **prestartAllCoreThreads**.
- Developers can be tempted to set the core size to zero so that the worker threads will eventually be torn down and therefore won't prevent the JVM from exiting. This can cause strange-seeming behavior in thread pools that don't use a **SynchronousQueue** for their work queue (as **newCachedThreadPool** does). If the pool is already at its core size **ThreadPoolExecutor** creates a new thread only if the work queue is full. So tasks submitted to a thread pool of zero core size will not execute until the queue fills up, which may not be what is desired. Use **allowCoreThreadTimeOut** with core size set to zero if you want a bounded thread pool and have all the threads torn down when there is no work to do.

Implementations:

- the **newFixedThreadPool** factory sets both the core pool size and the maximum pool size to the requested poo size, creating the effect of infinite timeout
- the **newCachedPool** factory sets the maximum pool size to Integer.MAX_VALUE and the core pool size to zero with a timeout of one minute, creating the effect of an infinitely expandable thread pool that will contract again when demand decreases

Managing queued tasks

Unbounded thread creation may lead to instability. Bounded thread pools limit the number of tasks that can be executed concurrently. Representin tasks as Runnable objects is cheaper than with a thread. However, this is only a parial solution

- if tasks keep up coming faster than they can be processed the system can run out of memory
- response time will get progressively worse as the task queue grows

ThreadPoolExecutor allows you to supply a **BlockingQueue** to hold tasks awaiting execution. There are three basic approaches to task queueing:

- unbounded queue, tasks will queue up if all workers are busy, but the queue could grow without bound if the tasks keep arriving faster then they are executed. Unbounded **LinkedBlockingQueue** is the default for **newFixedThreadPool** and **newSingleThreadExecutor**

- bounded queue, tasks queue up until the queue is full, raising a question of what to do with new tasks when the queue is full. ArrayBlockingQueue, **LinkedBlockingQueue**, **PriorityBlockingQueue**.
- synchronous handout are acceptable if the thread pool is very large or unbounded, or if rejecting tasks is acceptabke. Tasks are handed off to threads waiting on a queue, new threads are created if current thread pool is less than the maximum, otherwise tasks are rejected according to the saturation policy. The **newCachedThreadPool** factory uses this approach.

Also, bounded thread pools or queues are good only when tasks are independent. They are not suitable when tasks depend on each other, because bounded implementations can cause starvation deadlock. Use

- **cachedThreadPool** where both the queue and the pool are unbounded
- or use bounded thread pool, a **SynchronousQueue** as the work queue and the **caller-runs** saturaiion policy.

Some considerations

- fixed thread pool is a good choice when you need to limit the number of concurrent tasks for resource-management purposes (server application handling requests)
- the **newCachedThreadPool** factory provides better queueing performance than a fixed thread pool (non-blocking algorithm for **SyncrhonousQueue**)

Saturation policies

Saturation policy comes into play when:

- the work queue fills up
- a task is submitted to a shutdown **Executor**

Saturation policy of **ThreadPoolExecutor** can be modified with **setRejectedExecutionHandler**.

Policies:

- the **abort** (default) policy causes execute to throw the unchecked **RejectedExecutionException**
- the **discard** policy silently discards the newly submitted task if it cannot be queued for execution
- the **discard-oldest** policy discards the task that would otherwise be executed next and tries to resubmit the new task. (for priority queues discards a task with the highest priority, not a good choice)
- the **caller-runs** policy neither discards nor throws an exception, but instead tries to slow down the flow of the new tasks by pushing some of the work back to the caller. It executes the newly submitted task not in a thread pool but in the thread that calls **execute**. The main thread would also not be accepting during this time, so incoming requests would queue up in the TCP layer instead of in the application. The TCP

layer will start discarding connection requests as well. As the server becomes overloaded, the overload is gradually pushed outward — from the pool threads to the work queue to the application to the TCP layer and eventually to the client — enabling more graceful degradation under load.

- no saturation policy to block **execute** when the work queue is full. The same effect may be accomplished by using a semaphore to bound the task injection rate: use an unbounded queue, set the bound on the semaphore to be equal to the thread pool size plus the number of queued tasks you want to allow.

Thread factories

ThreadPool has a single method **createPool** that is called when the pool wants to create a new thread.
Some reasons to define thread factory:

- specify **UncaughtExceptionHandler**
- instantiate an instance of custom **Thread** class

Customizing ThreadPoolExecutor after construction

Most of the options passed to the **ThreadPoolExecutor** constructor can be modified via setters. If executor was created with one of factory methods in **Executors** then you can cast the result to **ThreadPoolExecutor** to access these setters.

Executors includes a factory method **unconfigurableExecutorService**, it wraps one with only the methods of **ExecutorService** so it cannot be further configured. **newSingleThreadExecutor** returns an **ExecutorService** wrapped in this manner rather than a raw **ThreadPoolExecutor** (it promises not to execute threads concurrently; if some misguided code was to increase the pool size on a single-treaded executor, it would undermine the indended execution semantics).

Extending ThreadPoolExecutor

ThreadPoolExecutor is designed for extension, it provides several hooks for subclasses to override:

- **beforeExecute**, **afterExecute** — called in the thread that executes the task, can be used for adding logging, timing, monitoring, statistics gathering; afterExecute executes if the task completes normally or throws **Exception** (if it throws Error then the **afterExecute** is not invoked). If **beforeExecute** throws an **RuntimeException**, the task and **afterExecute** are not executed at all.
- **terminated** hook is called when the thread pool completes the shutdown process, after all tasks have finished and all worker threads have shut down. Can be used to release resources allocated by **Executor** during lifetime, perform notification, logging, statistics gathering.

Parallelizing Recursive Algorithms

Sequential loop iterations are suitable for parallelization when **each iteration is independent of the others** and **the work done in each iteration of the loop body is significant enough** to offset the cost of managing a new task. It is possible then to shutdown the executor and **awaitTermination** to wait for tasks completion. (Graph in-depth traversion is an example).

- parallel implementation will probably not consume the stack
- if no solution is found parallel implementation must terminate successfully (beware of this case)
- restrain the depth of search or the wait time