Executing Tasks in Threads

Application goals:

- Low latency
- Graceful degradation as they become overloaded

Choosing task boundaries:

- Independence — facilitates concurrency
- Small fraction of application's processing capacity

Many server applications offer a natural choice of task boundary

Approaches:

- Executing sequentially
  Pros: offers simplicity and safety advantage
  Cons: poor latency and thorughput, obviously: IO, computation, serialized requests (one at a time)
  Explicitly creating threads for tasks
  Pros
  * Improves latency — request processing is offloaded from the main loop
  * Improves throughput if blocking (IO, locking, resources availability) is presented or if server has multiple cores
  Cons
  * Task handling code must be thread safe
  * CPU overhead: creating a new thread for each request is time consuming
  * Memory consumption: if not enough cpus, thread stay idle
  * Limit on the number of available threads
  * During prototyping or testing problem of too much threads can stay undetected: put a limit

The Executor Framework

Executor framework

- Decouples task handling from it's execution: possible to use different execution policies
- Implementations add hooks for statistics gathering, application management, monitoring

Execution policiy

Execution policiy defines

- In what thread it will be executed
- In what order to exeute tasks

- How many tasks will execute concurrently
- How many tasks can be queued pending execution
- If a task should be rejected because the system is overloaded: who is the victim and how application should be notified
- Actions to be taken before and after execution of a task

Execution policy is a resource management tool, depends on

- computing resources
- quality of service requirements
- ensuring application does not fail of suffer performance problems due to resource exhaustion

Thread pool

Pool is tightly bound to working queue

- request task from queue
- execute task
- return to pool

Thread pool advantages

- Reusing threads reduces thread creation and teardown costs
- Reduces requests processing latency
- Protecting against threads competing for resources

Executors static methods

- newFixedThreadPool — creates tasks as they are submitted, up to maximum pool size, attempts to keep pool size constant
- newCachedThreadPool — reaps idle threads when current size of pool exceeds demand for processing, adds new threads when demand increases
- newSingletonThreadExecutor — single thread, tasks processed sequentially, as defined by task queue (FIFO, LIFO, priority order)
- newScheduledThreadPool — fixed size widh delayed or periodic tasks

Lifecycle

JVM can’t exit until all non-daemon threads exit. Framework needs shutdown policy.

ExecutorService extending **Executor** adds lifecycle management capabiities. It has 3 states:

- Running — initial state
- Shutting down
- Terminated

Ways to shutdown

- **shutdown** method issues a graceful shutdown: no new tasks accepted, previously submitted tasks are allowed to complete
- **shutdonNow** method issues an abrupt shutdown: attemtps to cancel outstanding tasks and does not start queued tasks

When terminating

- new submitted tasks are handled by the rejected execution handler — might discard task or make **execute** throw **RejectedExecutionException**
- once all tasks are complete, service transitions to the terminated state — use **awaitTermination** or **isTerminated** to be notified when

Delayed and periodic tasks

Timer can execute deferred and periodic tasks, but has drawbacks:

- it's single threaded, when one task takes too long, another will:
  fixed delay — skip some runs
  fixed rate — runs several time in consequence
- if a task throws unchecked exception then
  Timer thread does not catch it, this terminates thread
  Timer does not resurrect the thread, assumes it's cancelled, scheduled tasks stop running, new tasks can't be scheduled

ScheduledThreadPoolExecutor solves addresses problems.

DelayQueue, a **BlockingQueue** implementation, provides scheduling functionality of ScheduledThreadPoolExecutor

- manages a collection of **Delayed** objects,
- **Delayed** has a delay time associated with it
- **DelayQueue** lets you take only expired objects

Callable, Runnable, Future

Runnable, Callable, differences:

- returning value
- throwing an exception

Future allows:

- cancel the task
- **get** the result, behavior depends on the state:
  if the task has successfully completed it returns immediately
  if the task failed it rethrows exception wrapped in ExecutionException
  if the task was cancelled it throws CancellationException
  if the task is executing it blocks

CompletionService
ExecutorCompletionService

- submit **Callable** tasks
- **poll** or **take** to retrieve completed results packaged as Futures