Java Concurrency

ThreadPools

Stetsenko Maksym
stetsenko89@gmail.com

Prerequesites for thread pools

Thread per task is the bad way:

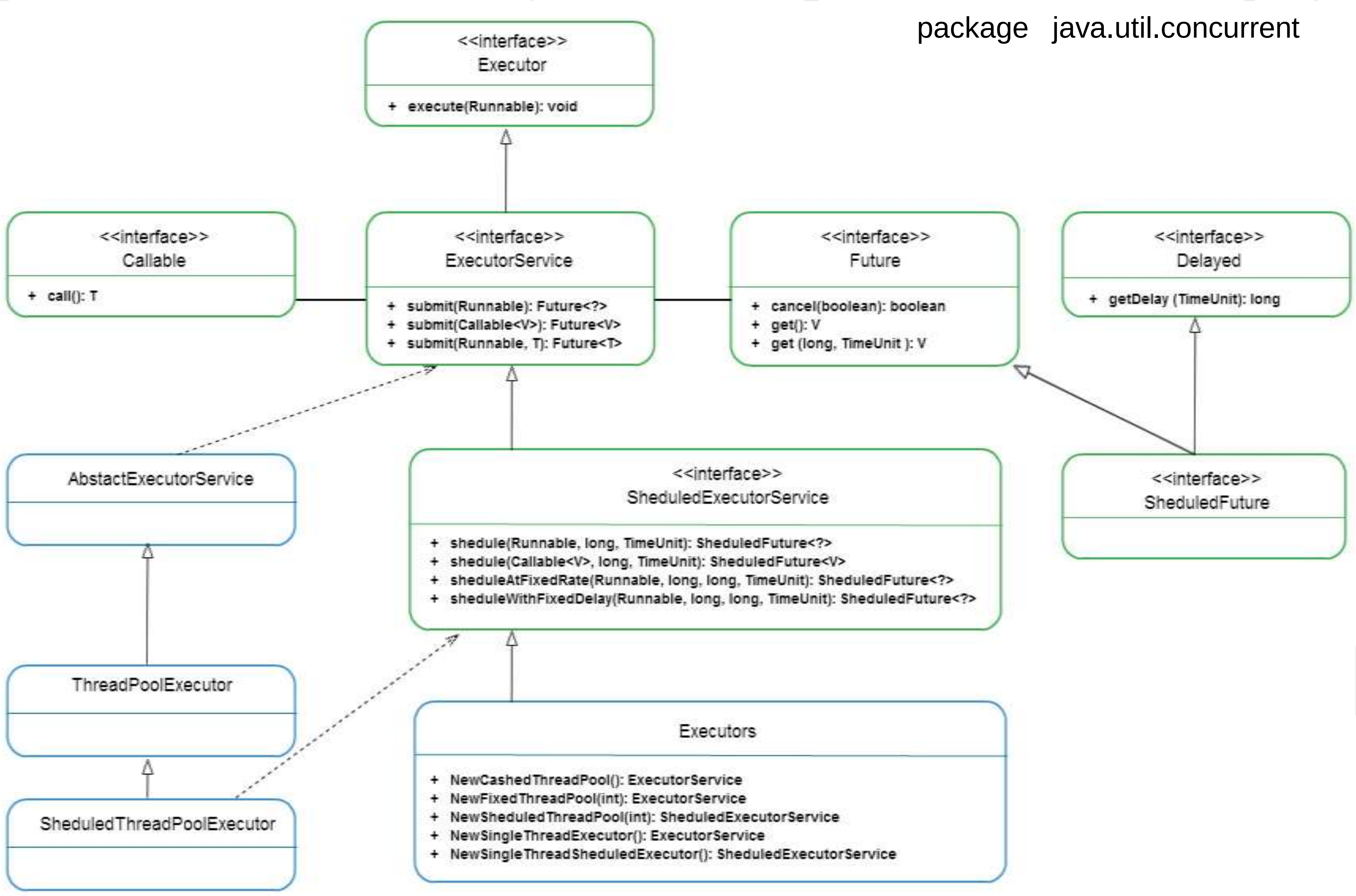
- creating a new thread for each request can consume significant computing resources.
- having many threads competing for the CPUs can impose other performance costs as well.

Reasons for using thread pools:

- gaining some performance when the threads are reused.
- better program design, letting you focus on the logic of your program.

Executor

Executor - java’s interfases for thread pool implementations.


package java.util.concurrent
<<interface>> Executor
+ execute(Runnable): void
<<interface>> Callable
+ call(): T
<<interface>> ExecutorService
+ submit(Runnable): Future<?>
+ submit(Callable<V>): Future<V>
+ submit(Runnable, T): Future<T>
<<interface>> Future
+ cancel(boolean): boolean
+ get(): V
+ get (long, TimeUnit): V
<<interface>> Delayed
+ getDelay (TimeUnit): long
AbstactExecutorService
<<interface>> SheduledExecutorService
+ shedule(Runnable, long, TimeUnit): SheduledFuture<?>
+ shedule(Callable<V>, long, TimeUnit): SheduledFuture<V>
+ sheduleAtFixedRate(Runnable, long, long, TimeUnit): SheduledFuture<?>
+ sheduleWithFixedDelay(Runnable, long, long, TimeUnit): SheduledFuture<?>
<<interface>> SheduledFuture
ThreadPoolExecutor
SheduledThreadPoolExecutor
Executors
+ NewCashedThreadPool(): ExecutorService
+ NewFixedThreadPool(int): ExecutorService
+ NewSheduledThreadPool(int): SheduledExecutorService
+ NewSingleThreadExecutor(): ExecutorService
+ NewSingleThreadSheduledExecutor(): SheduledExecutorService

ExecutorService

- Provides a means for you to manage the executor and its tasks.

- Has three states: running, shutting down and terminated.

- ExecutorServices are initially created in the running state.

- When all tasks are complete (including any waiting tasks), the executor service enters a terminated state.

Tasks execution:

Future<?> **submit(**Runnable task**)**

Future<T> **submit(**Callable<T> task**)**

Future<T> **submit(**Runnable task, T result**)**

} returns a Future object that can be used to track the progress of the task.

List<Future<T>> **invokeAll(**Collection<? extends Callable<T>> tasks**)** - returning a list of Futures holding their status and results when all complete.

T **invokeAny(**Collection<? extends Callable<T>> tasks**)** - methods execute the tasks in the given collection.

Note: *In* ***invokeAny*** *method if one task has completed, the remaining tasks are subject to cancellation.*

ExecutorService

Shutting down:

shutdown() - terminates the execution;
Any tasks that have already been sent to the executor are allowed to run, but no new tasks are accepted.
When all tasks are completed, the executor stops its thread(s).

List<Runnable> **shutdownNow**() - attempts to stop execution sooner;
All tasks that have not yet started are not run and are instead returned in a list.
Still, existing tasks continue to run: they are interrupted.

***Note**: Please remember that there's a period of time between calling the shutdown() or shutdownNow() method and when tasks executing in the executor service are all complete.*

boolean **isTerminated**() - to see if the executor service is in the terminated state.

boolean **awaitTermination**(long timeout, TimeUnit unit) - Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

ExecutorService

Using requirements

-Create the pool itself

- **Create the tasks** that the pool is to run (Runnable or Callable objects)

Thread Pool creating capabilities

```
package java.util.concurrent ;
public class ThreadPoolExecutor implements ExecutorService {

        public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit,
                                                        BlockingQueue<Runnable> workQueue);   [ such param list above mark as - //- ]

        public ThreadPoolExecutor( - // -, ThreadFactory threadFactory);

        public ThreadPoolExecutor( - // -, RejectedExecutionHandler handler);

        public ThreadPoolExecutor( - // -, ThreadFactory threadFactory, RejectedExecutionHandler handler);

}
```

ExecutorService

Using example

```
public class ThreadPoolTest {
    public static void main(String[] args) {
 ThreadPoolExecutor tpe = new ThreadPoolExecutor( 5, 10, 30L, TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
        Task[] tasks = new Task[25];
        for (int i = 0; i < tasks.length; i++) {
            tasks[i] = new Task("Task " + i);
            tpe.execute(tasks[i]);
        }
        tpe.shutdown( );
    }
}
```

```
class Task implements Runnable {

    String taskInfo;

    Task(String taskInfo){
            this.taskInfo =  taskInfo;
    }

    @Override
    public void run() {
        System.out.println(taskInfo);
        //try block omitted
            Thread.sleep(1000);
    }
}
```

java.util.concurrent.Executors

A Flexible thread pool implementations:

- Create a thread pool by calling one of the static factory methods:

newFixedThreadPool. - A fixed size thread pool creates threads as tasks are submitted, up to the maximum pool size, and then attempts to keep the pool size constant (adding new threads if a thread dies due to an unexpected exception).

newCachedThreadPool. - A cached thread pool has more flexibility to reap idle threads when the current size of the pool exceeds the demand for processing, and to add new threads when demand increases, but places no bounds on the size of the pool.

newSingleThreadExecutor. - A single threaded executor creates a single worker thread to process tasks, replacing it if it dies unexpectedly. Tasks are guaranteed to be processed sequentially according to the order imposed by the task queue (FIFO, LIFO, priority order).

newScheduledThreadPool. - A fixed-size thread pool that supports delayed and periodic task execution, similar to Timer.

Note: *The newFixedThreadPool and newCachedThreadPool factories return instances of the general purpose ThreadPoolExecutor, which can also be used directly to construct more specialized executors.*

ScheduledThreadPool example

```
public class SheduledTreadPoolExample {

    public static void main(String[] args) throws InterruptedException {

        ScheduledExecutorService scheduledThreadPool = Executors.newScheduledThreadPool(2);
        //schedule to run after sometime
        System.out.println("Current Time = " + new Date());

        Runnable runnableInst = new Runnable() {
            public void run() {

                System.out.println(Thread.currentThread().getName() + " Start. Time = " + new Date());
                try {
                    Thread.sleep(5000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + " End. Time = " + new Date());
            }
        };

        for(int i=0; i<3; i++){
            scheduledThreadPool.schedule(runnableInst , 3, TimeUnit.SECONDS);
        }

        //add some delay to let some threads spawn by scheduler
        Thread.sleep(30000);
        scheduledThreadPool.shutdown();
        while(!scheduledThreadPool.isTerminated()){//delay can be added}

        System.out.println("Finished all threads");
    }
}
```

Result

```
Current Time = Sat Feb 15 18:31:18 EET 2014
pool-1-thread-2 Start. Time = Sat Feb 15 18:31:21 EET 2014
pool-1-thread-1 Start. Time = Sat Feb 15 18:31:21 EET 2014
pool-1-thread-1 End. Time = Sat Feb 15 18:31:26 EET 2014
pool-1-thread-1 Start. Time = Sat Feb 15 18:31:26 EET 2014
pool-1-thread-2 End. Time = Sat Feb 15 18:31:26 EET 2014
pool-1-thread-1 End. Time = Sat Feb 15 18:31:31 EET 2014
Finished all threads
```

Queues and Thread Pool sizes

- The thread pool size and the queue for the tasks are two fundamental things for thread pool
- Are set in the constructor
- The size can change dynamically while the queue must remain fixed

Size

The size of the thread pool varies between a given minimum (or core) and maximum number of threads.

Queue

The queue is the data structure used to hold tasks that are awaiting execution.

- affects how certain tasks are scheduled.
- add tasks to queue through the execute() method of the thread pool, don’t call any methods on it directly
- The getQueue() method returns the queue, but you should use that for debugging purposes only;

The basic principle is that the thread pool tries to keep its minimum number of threads active.

Queues and Thread Pool sizes

APIs to deal with the **pool's size** and **queue:**

```
package java.util.concurrent;
public class ThreadPoolExecutor implements ExecutionService {

    public boolean prestartCoreThread ( );

    public int prestartAllCoreThreads ( );

    public void setMaximumPoolSize (int
    maximumPoolSize);

    public int getMaximumPoolSize ( );

    public void setCorePoolSize (int corePoolSize);

    public int getCorePoolSize ( );

    public int getPoolSize ( );

    public int getLargestPoolSize ( );

    public int getActiveCount ( );

    public BlockingQueue<Runnable> getQueue ( );

    public long getTaskCount ( );

    public long getCompletedTaskCount ( );

}
```

Queues and Thread Pool sizes

How the queue interacts with the number of threads:

1. **The thread pool is constructed** with M core threads and N maximum threads. At this point, no threads are actually created.

   prestartAllCoreThreads() - create the M core threads.

   prestartCoreThread() - preallocate one core thread.

2. **A task enters the pool** (via the thread pool's execute() method).

What happens when task enters the pool ?

**It's assumed that: M - core threads, N - maximum threads, C - an amount of created threads.*

C < M - new thread is created in the pool's attempt to reach M threads, even if some of the existing threads are idle.

M <= C <= N and **one of those threads is idle** - the task is run by an idle thread

M <= C <= N and **all the threads are busy**, then if the **task can be placed on the work queue**

$\xrightarrow{\text{yes}}$ it's put on the queue and no new thread is started

$\xrightarrow[\text{no}]{}$ the pool starts a new thread and runs the task on that thread

C == N and **all threads are busy** and if the **queue has reached its maximum size**

$\xrightarrow{\text{yes}}$ task is rejected

$\xrightarrow[\text{no}]{}$ the task is accepted and run by idle thread when all previously queued tasks have run

Queues and Thread Pool sizes

How the queue interacts with the number of threads:

3. A task completes execution.

- The thread running the task then runs the next task on the queue.

- **If no tasks are on the queue**, one of two things happens:

M < C and if a new task is queued within the timeout period

- yes → the thread runs it
- no → the thread exits, reducing the total number of threads in the pool

Note: *If the specified timeout is 0, the thread always exits, regardless of the requested minimum thread pool size.*

C <= M and timeout was 0

- yes → thread blocks indefinitely waiting for a new task to be queued
- no → thread exits

The implication: *the choice of pool size and queue are important to getting the behavior you want.*

Queues and Thread Pool sizes

Choosing queue (three choices)

- **SynchronousQueue**
  - effectively has a size of 0
  - Tasks are either run immediately (the pool has an idle thread or is below its maximum threshold) or are rejected immediately according to the saturation policy
  - an unlimited maximum number of threads prevents rejection ,but the throttling benefit is lost
  - SynchronousQueue is a practical choice only if the pool is unbounded or if rejecting excess tasks is acceptable
  - the ***newCachedThreadPool*** factory uses a SynchronousQueue

- **Unbounded queue** with an unlimited capacity, such as a ***LinkedBlockingQueue***.
  - adding a task to the queue always succeeds
  - the thread pool never creates more than M threads and never rejects a task

- **Bounded queue** with a fixed capacity, such as a ***LinkedBlockingQueue*** or an ***ArrayBlockingQueue***
  - let's suppose that the queue has a bounds of P. And tasks are becoming faster than they can be processed
  - the pool creates threads until it reaches M threads
  - if M threads created then the pool starts queueing tasks until the number of waiting tasks reaches P
  - then pool starts adding threads until it reaches N threads
  - if we reach a state where N threads are active and P tasks are queued, additional tasks are rejected
  - ***LinkedBlockingQueue*** or ***ArrayBlockingQueue*** causes tasks to be started in the order in which they arrived (FIFO)
  - ***PriorityBlockingQueue*** orders tasks according to priority. (can be defined by natural order or by a Comparator)

Rejecting tasks

Tasks are rejected by the execute() method :

- **depending on the type of queue** you use in the thread pool
- **if the queue is full**
- **if the shutdown( )** method has been **called** on the thread pool

Note: *When a task is rejected, the thread pool calls the rejected execution handler associated with the thread pool. It applies to all potential tasks. There is one rejected execution handler for the entire pool. You can write your own rejected execution handler, or you can use one of four predefined handlers*

APIs to deal with the **rejected execution handler**:

```
package java.util.concurrent;
public interface RejectedExecutionHandler {
   public void rejectedExecution(Runnable r, ThreadPoolExecutor executor);
}

package java.util.concurrent;
public class ThreadPoolExecutor implements ExecutorService {
   public void setRejectedExecutionHandler(RejectedExecutionHandler handler);
   public RejectedExecutionHandler getRejectedExecutionHandler( );
   public static class AbortPolicy implements RejectedExecutionHandler;
   public static class CallerRunsPolicy implements RejectedExecutionHandler;
   public static class DiscardPolicy implements RejectedExecutionHandler;
   public static class DiscardOldestPolicy implements RejectedExecutionHandler;
}
```

Rejecting tasks

Predefined handlers:

1. AbortPolicy

- this handler does not allow the new task to be scheduled when the queue is full or the pool has been shut down;
- in that case, the execute() method throws a **RejectedExecutionException** (is a runtime exception).
- this is the default policy for rejected tasks.

2. DiscardPolicy

- this handler silently discards the task. No exception is thrown.

3. DiscardOldestPolicy

- this handler silently discards the oldest task in the queue and then queues the new task.
- the oldest task LinkedBlockingQueue or ArrayBlockingQueue is the first in line to execute when a thread becomes idle.
- SynchronousQueue has no waiting tasks so the execute() method silently discards the submitted task.
- if the pool has been shut down, the task is silently discarded.

Rejecting tasks

Predefined handlers:

4. CallerRunsPolicy

- executes the new task independently of the thread pool if the queue is full.
- task is immediately executed by calling its run() method
- the task is silently discarded, if the pool has been shudown.
- implements a form of throttling to slow down the flow of new tasks by pushing some of the work back to the caller.

Creating a Fixed-sized Thread Pool with a Bounded Queue and the Caller-runs Saturation Policy.

```
ThreadPoolExecutor executor =  new ThreadPoolExecutor(M_THREADS, N_THREADS,
                                                      0L, TimeUnit.MILLISECONDS,
                                                      new LinkedBlockingQueue<Runnable>(CAPACITY));

executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
```

Rejecting tasks example

```
public class RejectionExample {

public static void main(String[] args) throws InterruptedException{

ThreadPoolExecutor executor =  new ThreadPoolExecutor(2, 2, 0L,
TimeUnit.MILLISECONDS, new LinkedBlockingQueue<Runnable>(2));
executor.setRejectedExecutionHandler(new PrintRejectionHadler());
            // Add some tasks
            for (int i = 0; i < 9; i++) {
              executor.execute(new WorkerTask("Task " + (i + 1)));
              // Sleep
              try {
                      Thread.sleep(500);
                  } catch (InterruptedException ex) {
                      ex.printStackTrace();
                  }
              }
          }
}
```

```
Result
Task 1 : Running; Time:1392476206s
Task 2 : Running; Time:1392476206s
Task 5 : Rejected; Time:1392476208s
Task 1 : Done; Time:1392476208s
Task 3 : Running; Time:1392476208s
Task 2 : Done; Time:1392476208s
Task 4 : Running; Time:1392476208s
Task 8 : Rejected; Time:1392476209s
Task 3 : Done; Time:1392476210s
Task 6 : Running; Time:1392476210s
Task 4 : Done; Time:1392476210s
Task 7 : Running; Time:1392476210s
Task 6 : Done; Time:1392476212s
Task 7 : Done; Time:1392476212s
```

```
 public class WorkerTask implements Runnable {
// Task name
 private String name;
 public WorkerTask(String name) {
        this.name = name;
        }
        @Override
        public void run() {
        System.out.println(name + " : Running");
         try {
             Thread.sleep(2000);
         } catch (InterruptedException e) {}
         System.out.println(name + " : Done");
         }
         @Override
          public String toString() {
               return name;
           }
           }
```

```
public class PrintRejectionHadler implements RejectedExecutionHandler {
        public PrintRejectionHadler() { }
        @Override
        public void rejectedExecution(Runnable r, ThreadPoolExecutor e) {
              System.out.println(r.toString() + " : Rejected");
        }
}
```

Rejecting tasks

One more trick

There is no predefined saturation policy to make execute block when the work queue is full.
However, the same effect can be accomplished by using a Semaphore to bound the task injection rate, as shown in listing below.
In such an approach, use an unbounded queue (there's no reason to bound both the queue size and the injection rate) and set the bound on the semaphore to be equal to the pool size plus the number of queued tasks you want to allow, since the semaphore is bounding the number of tasks both currently executing and awaiting execution.

```
public class BoundedExecutor {
private final Executor exec;
private final Semaphore semaphore;

public BoundedExecutor(Executor exec, int bound) {
        this.exec = exec;
        this.semaphore = new Semaphore(bound);
}
```

→

```
public void submitTask(final Runnable command)
throws InterruptedException {
                semaphore.acquire();
                try {
                        exec.execute(new Runnable() {
                                public void run() {
                                try {
                                        command.run();
                                } finally {
                                        semaphore.release();
                                }
                        }
                        });
                } catch (RejectedExecutionException e) {
                        semaphore.release();
                }
        }
}
```

Thread creation

To create a thread, the pool uses the currently installed thread pool factory

API to manage ThreadFactories and alive time:

```
package java.util.concurrent;
public interface ThreadFactory {
   public Thread newThread(Runnable r);
}
package java.util.concurrent;
public class ThreadPoolExecutor implements ExecutorService {
   public void setThreadactory(ThreadFactory threadFactory);
   public ThreadFactory getThreadFactory( );
   public void setKeepAliveTime(long time, TimeUnit unit);
   public long getKeepAliveTime(TimeUnit unit);
}
```

Thread factory allows to set up:

- threads with special names, priorities, daemon status, thread group
- specify an **UncaughtExceptionHandler**
- instantiate a custom Thread to debug, logging or just give meaningful names to interpret error logs

Thread creation

Executors.DefaultThreadFactory creates a thread with the following characteristics:

- New threads belong to the same thread group as the thread that created the executor.
- Security manager policy can override this and place the new thread in its own thread group
- The name of the thread reflects its pool number and its thread number within the pool.
- Within a pool, threads are numbered consecutively beginning with 1;
- Thread pools are globally assigned a pool number consecutively beginning with 1.
- The daemon status of the thread is the same as the status of the thread that created the executor.
- The priority of the thread is Thread.NORM_PRIORITY.

Executors.PrivilegedThreadFactory

- Takes advantage of security policies to grant permissions
- Creates pool threads that have the same permissions, AccessControlContext, and contextClassLoader as the thread creating the privilegedThreadFactory.
- Otherwise, threads created by the thread pool inherit permissions from whatever client happens to be calling execute or submit at the time a new thread is needed, which could cause confusing security-related exceptions.
- It’s recommended to use the privilegedThreadFactory factory method in Executors to construct your thread factory.

ThreadFactory example

```
public class CustomThreadFactory implements ThreadFactory
{
    @Override
    public Thread newThread(Runnable r)
    {
        Thread t = new Thread(r);

        //perform logging of exceptions
        t.setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler()
        {
            @Override
            public void uncaughtException(Thread t, Throwable e)
            {
                LoggerFactory.getLogger(t.getName()).error(e.getMessage(), e);
            }
        });

        return t;
    }
}
```

Extending ThreadPoolExecutor

"Hooks" for subclasses for adding logging, timing, monitoring, or statistics gathering:

package java.util.concurrent;

public class **ThreadPoolExecutor implements ExecutorService** {

protected void **beforeExecute**(Thread t, Runnable r) - is invoked prior to executing the given Runnable in the given thread

- This method is invoked by thread t that will execute task r
- May be used to re-initialize ThreadLocals
- Using for logging

Note: *To properly nest multiple overridings, subclasses should generally invoke super.beforeExecute at the end of this method*

protected void **afterExecute**(Runnable r, Throwable t) - is called whether the task completes by returning normally from run or by throwing an *Exception*

- If the task completes with an Error, afterExecute is not called
- If *beforeExecute* throws a *RuntimeException*, the task is not executed and *afterExecute* is not called
- When actions are enclosed in tasks (such as FutureTask) either explicitly or via methods such as submit, these task objects catch and maintain computational exceptions, and so they do not cause abrupt termination, and the internal exceptions are not passed to this method.

Note: *To properly nest multiple overridings, subclasses should generally invoke super.afterExecute at the beginning of this method.*

protected void **terminated**() - is called when the thread pool completes the shutdown process, after all tasks have finished and all worker threads have shut down.

- It can be used to release resources allocated by the *Executor* during its lifecycle
- Perform notification or logging or finalize statistics gathering.

Note: *To properly nest multiple overridings, subclasses should generally invoke super.terminated within this method*

Callable tasks and Future results

Prerequisites of Callable task**:**

- Runnable is a fairly limiting abstraction
- run() cannot return a value or throw checked exceptions
- Although it can have side effects such as writing to a log file or placing a result in a shared data structure

Java defines **callable task** to provide **more control over tasks**

```
package java.util.concurrent;
public interface Callable<V> {
          public <V> call( ) throws Execption;
}
```

Note: *to express a non-value-returning task with Callable, use Callable<Void>.*

Callable features:

- Can return a result or throw a checked exception
- Callable objects are used only by executor services (not simple executors)
- The services operate on callable objects by invoking their call() method and keeping track of the results of those calls.

Callable tasks and Future results

When executor service runs a callable object, the service returns a **Future** object that **allows you to:**

- retrieve those results
- monitor the status of the task
- cancel the task

Once a Future task is completed, it stays in that state forever.

The Future interface :

```
public interface Future<V> {
        V get( ) throws InterruptedException, ExecutionException;
        V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException;
        boolean isDone( );
        boolean cancel(boolean mayInterruptIfRunning);
        boolean isCancelled();
}
```

get() - returns the results of its corresponding call( ) method. The get() method blocks until the call() method has returned (or until the optional timeout has expired). The exception handling code surrounding Future.get deals with two possible problems: that the task encountered an Exception, or the thread calling get was interrupted before the results were available.

The **behaviour of *get*** varies **depending on the task state:**

running - it blocks until the task completes.

completed - returns immediately or throws an Exception.

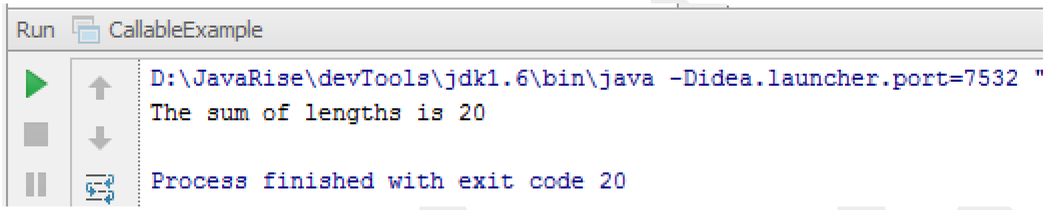
- if the task completes by throwing an exception, *get* rethrows it wrapped in an *ExecutionException*.
- if the current thread was cancelled or interrupted while waiting, get throws *CancellationException*

Callable-Future example

```
public class CallableExample {
 public static void main(String args[]) throws Exception {
        ExecutorService pool = Executors.newFixedThreadPool(3);
        Set<Future<Integer>> set = new HashSet<Future<Integer>>();
        for (String word: new String[]{"first","second","third","n-th"}) {
          Callable<Integer> callable = new WordLengthCallable(word);
            Future<Integer> future = pool.submit(callable);
            set.add(future);
        }
        int sum = 0;
        for (Future<Integer> future : set) {
            sum += future.get();
        }
        System.out.printf("The sum of lengths is %s%n", sum);
        System.exit(sum);
    }
}
```

```
public static class WordLengthCallable implements Callable {

    private String word;
    public WordLengthCallable(String word) {
        this.word = word;
    }
    public Integer call() {
        return word.length();
    }
}
```



```
D:\JavaRise\devTools\jdk1.6\bin\java -Didea.launcher.port=7532 "
The sum of lengths is 20

Process finished with exit code 20
```

NOTE:

The only requirement of call() is the value is returned at the end of the call when the get() method of Future is later called.

Because the goal of the program is to calculate the sum of all word lengths, it doesn't matter in which order theCallable tasks finish.

The first get() call to Future will just wait for the first task in the Set to complete.

This does not block other tasks from running to completion separately. It is just waiting for that one thread or task to complete.

Callable tasks and Future results

Cancelling tasks

call() - immediately returns if the state is cancelled

cancel() - set the state to cancelled. Method **behaviour** varies **depending on one of three the task state:**

- **waiting** for execution: state is set to cancelled and the call() method is never executed.
- execution **completed**: cancel( ) method has no effect.
- **running**: if the **mayInterruptIfRunning** flag
  - **false** – the cancel() method again has no effect.
  - **true -** the thread running the callable object is interrupted

Note: *The callable object must still pay attention to interuption, periodically calling the Thread.interrupted() method to see if it should exit.*

What happens when object in a thread pool is cancelled ?

- there is no immediate effect: the object still remains queued for execution.
- when the thread pool is about to execute it checks if task has been canceled then skip
- future calls to the execute() method may still be rejected, even though cancelled objects are on the thread pool's queue: the execute() method does not check the queue for cancelled objects.

purge() - looks over the entire queue and removes any cancelled objects.

caveat: *if a second thread attempts to add something to the pool (using the execute() method) at the same time the first thread is attempting to purge the queue, the attempt to purge the queue fails and the canceled objects remain in the queue.*

remove() - immediately removes the task from the thread pool queue. Method can be used with standard runnable objects.

Callable tasks and Future results

FutureTask

public class **FutureTask**<V> **implements RunnableFuture<V>** {}

FutureTask class:

- is used internally by the executor service,
- an instance of this class is returned from the submit() method of an executor service,
- ***Runnable*** object can be associated with a future result using the ***FutureTask*** class,
- ***FutureTask*** can be constructed directly a with an embedded runnable object and to monitor the status,

A ***FutureTask*** object can hold either an embedded runnable or callable object:

```
public FutureTask(Callable<V> task);
public FutureTask(Runnable task, V result);
```

The **get()** method (**for** embeded a **callable task**) - returns whatever is returned by the call( ) method of that embedded object.

The **get()** method (**for** embeded a **runnable object**) - returns whatever object was used to construct the future task object itself.

Note*: As of Java 6,* ***ExecutorService*** *implementations can override* ***newTaskFor*** *in* ***AbstractExecutorService*** *to control instantiation of the* ***Future*** *corresponding to a submitted* ***Callable*** *or* ***Runnable****.*
The default implementation just creates a new FutureTask, as shown below:

Default Implementation of **newTaskFor in ThreadPoolExecutor** (inherited from AbstractExecutorService)**.**

```
protected <T> RunnableFuture<T> newTaskFor(Callable<T> task) {
   return new FutureTask<T>(task);
}
```

Caveats

Thread Starvation 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 will **always** deadlock.

The same thing can happen in larger thread pools if all threads are executing tasks that are blocked waiting for other tasks still on the work queue. This is called thread starvation deadlock.

Example**:** “ **Task that Deadlocks in a Single-threaded Executor** ”. ***Don't Do this.***

```
public class ThreadDeadlock {
        ExecutorService exec = Executors.newSingleThreadExecutor();

        public class RenderPageTask implements Callable<String> {
                public String call() throws Exception {
                        Future<String> header, footer;
                        header = exec.submit(new LoadFileTask("header.html"));
                        footer = exec.submit(new LoadFileTask("footer.html"));
                        String page = renderBody();

                        // Will deadlock -- task waiting for result of subtask
                        return header.get() + page + footer.get();
                }
        }
}
```

Caveats

Migrating from Single-Thread Executor

Single-threaded executors make stronger promises about concurrency than do arbitrary thread pools. They guarantee that tasks are not executed concurrently, which allows you to relax the thread safety of task code. Objects can be confined to the task thread, thus enabling tasks designed to run in that thread to access those objects without synchronization, even if those resources are not thread-safe. This forms an implicit coupling between the task and the execution policy - the tasks require their executor to be single-threaded. In this case, if you changed the Executor from a single-threaded one to a thread pool, thread safety could be lost.

ThreadLocal

Tasks that use ***ThreadLocal***. ***ThreadLocal*** allows each thread to have its own private "version" of a variable.
However, executors are free to reuse threads as they see fit. The standard Executor implementations may reap idle threads when demand is low and add new ones when demand is high, and also replace a worker thread with a fresh one if an unchecked exception is thrown from a task.

Remember: *1. ThreadLocal makes sense to use in thread pools only if the thread-local value has a lifetime that is bounded by that of a task.*

2. Thread-Local should not be used in pool threads to communicate values between tasks.

Recomendation: *Reinitialize ThreadLocal variables in protected **beforeExecute** or **afterExecute** methods*

Caveats

Long-running tasks

Thread pools can have responsiveness problems if tasks can block for extended periods of time, even if deadlock is not a possibility.
A thread pool can become clogged with long-running tasks, increasing the service time even for short tasks.
One technique that can mitigate the ill effects of long-running tasks is for tasks to use timed resource waits instead of unbounded waits.
Most blocking methods in the plaform libraries come in both untimed and timed versions, such as Thread.join, BlockingQueue.put, CountDownLatch.await, and Selector.select.
If the wait times out, you can mark the task as failed and abort it or requeue it for execution later.
This guarantees that each task eventually makes progress towards either successful or failed completion, freeing up threads for tasks that might complete more quickly.

Summary

1. ExecutorService: task execution, shutting down.

2. **Thread rool implementations from java.util.concurrent.Executors:**

- newFixedThreadPool,
- newCachedThreadPool,
- newSingleThreadExecutor,
- newScheduledThreadPool

3. Correlation and API between pool size and type of queue.

4. Thread pool reaction on entering of a new task and task completion.

5. Impact of different types of queues:

- synchronous
- bounded
- unbounded

Summary

6. Rejecting tasks and predefined handlers:

- AbortPolicy
- DiscardPolicy
- DiscardOldestPolicy
- CallerRunsPolicy

7. Thread creation and thread factories:

- Executors.DefaultThreadFactory
- Executors.PrivilegedThreadFactory
- implementing of ThreadFactory interface

8. Extending ThreadPoolExecutor:

- beforeExecute
- afterExecute
- terminated

9. Callable tasks and Future results

10. Some basic caveats

The End

References

Books

1. Java Threads, Third Edition; Scott Oaks & Henry Wong; ISBN: 978-0-596-00782-9. O'Reilly, 2004.

2 . Concurrent Programming in Java: Design Principles and Patterns (2nd edition) Doug Lea; ISBN 0-201-31009-0. AddisonWesley, 1999.

Web resources

http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html

https://blogs.oracle.com/CoreJavaTechTips/entry/get_netbeans_6

http://www.ibm.com/developerworks/library/j-jtp0730/

http://www.nolte-schamm.za.net/2011/09/java-worker-thread-pool-with-threadpoolexecutor/

http://stackoverflow.com/questions/3179733/threadfactory-usage-in-java

http://www.vogella.com/tutorials/JavaConcurrency/article.html