How to Handle Died Threads due to Uncaught Exceptions in Java

Abdullah Ozturk - Blog
Nov 19, 2015 · 2 min read

In concurrent applications a thread might fail and die due to uncaught runtime exceptions even without noticing since the application may continue to work. Losing one consumer thread from a thread pool can be tolerable, but losing a single dispatcher thread can degrade application workflow.

There are four alternative approaches in Java to get notified when a task fails due to an exception, so that you can log or take some recovery action.

Proactive Approach

In the solution below, *run()* method of a thread is structured with *try-catch* block and if a task throws an unchecked exception, it allows the thread to die. The replacement of this worker thread with a new thread can be done while handling the exception

```java
final class MyTask implements Runnable {
    @Override
    public void run() {
        try {
            System.out.println("My task is started running...");
            // ...
            anotherMethod();
            // ...
        } catch (Throwable t) {
            System.err.println("Uncaught exception is detected! " + t
                    + " st: " + Arrays.toString(t.getStackTrace()));
            // ... Handle the exception
        }
    }

    private void anotherMethod() {
        throw new ArithmeticException();
    }
}
```

```java

public class ProactiveHandler {
    public static void main(String[] args) {
        // Create a fixed thread pool executor
        ExecutorService threadPool = Executors.newFixedThreadPool(10);
        threadPool.execute(new MyTask());
        // ...
    }
}
```

ProactiveHandler.java hosted with ❤ by GitHub   view raw

Uncaught Exception Handler

Secondly, you can define an uncaught exception handler for the threads created by your custom thread factory, which is passed to the thread pool. When a thread exits due to an uncaught exception, the JVM reports this event to our *UncaughtExceptionHandler*, otherwise the default handler just prints the stack trace to standard error.

```java
class MyThreadFactory implements ThreadFactory {
    private static final ThreadFactory defaultFactory = Executors.defaultThreadFactory(
    private final Thread.UncaughtExceptionHandler handler;

    public MyThreadFactory(Thread.UncaughtExceptionHandler handler) {
        this.handler = handler;
    }

    @Override
    public Thread newThread(Runnable run) {
        Thread thread = defaultFactory.newThread(run);
        thread.setUncaughtExceptionHandler(handler);
        return thread;
    }
}

class MyExceptionHandler implements Thread.UncaughtExceptionHandler {
    @Override
    public void uncaughtException(Thread thread, Throwable t) {
        System.err.println("Uncaught exception is detected! " + t
                + " st: " + Arrays.toString(t.getStackTrace()));
        // ... Handle the exception
    }
}

final class MyTask implements Runnable {
    @Override
    public void run() {
        System.out.println("My task is started running...");
        // ...
        throw new ArithmeticException();
        // ...
    }
}

public class UncaughtExceptionHandler {
    public static void main(String[] args) {
        ThreadFactory factory = new MyThreadFactory(new MyExceptionHandler());
        ExecutorService threadPool = Executors.newFixedThreadPool(10, factory);
        threadPool.execute(new MyTask());
        // ...
    }
}
```

UncaughtExceptionHandler.java hosted with ❤ by GitHub view raw

However, exceptions thrown from tasks make it to the uncaught exception handler only for tasks submitted with *execute()*; for tasks submitted with *submit()* to the executor service, any thrown exception is considered to be part of the task's return status.

Thread Pool Executor Handler

If you want to be notified when a task fails due to an exception and take some task-specific recovery action, the *afterExecute()* method in *ThreadPoolExecutor* can be overridden. The solution below handles both cases you use submit() or execute() methods of the execution service.

```java
class MyThreadPoolExecutor extends ThreadPoolExecutor {
    public MyThreadPoolExecutor(int corePoolSize, int maximumPoolSize, long keepAliveTi
            TimeUnit unit, BlockingQueue<Runnable> workQueue) {
        super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
    }

    @Override
    public void afterExecute(Runnable r, Throwable t) {
        super.afterExecute(r, t);
        // If submit() method is called instead of execute()
        if (t == null && r instanceof Future<?>) {
            try {
                Object result = ((Future<?>) r).get();
            } catch (CancellationException e) {
                t = e;
            } catch (ExecutionException e) {
                t = e.getCause();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
        if (t != null) {
            // Exception occurred
            System.err.println("Uncaught exception is detected! " + t
                    + " st: " + Arrays.toString(t.getStackTrace()));
            // ... Handle the exception
            // Restart the runnable again
            execute(r);
        }
        // ... Perform cleanup actions
    }
}

final class MyTask implements Runnable {
    @Override public void run() {
        System.out.println("My task is started running...");
        // ...
        throw new ArithmeticException(); // uncatched exception
        // ...
    }
}

public class ThreadPoolExecutorHandler {
    public static void main(String[] args) {
        // Create a fixed thread pool executor
        ExecutorService threadPool = new MyThreadPoolExecutor(10, 10, 0L, TimeUnit.MILL
                new LinkedBlockingQueue<>());
        threadPool.execute(new MyTask());
        // ...
    }
}
```

ThreadPoolExecutorHandler.java hosted with ❤ by GitHub view raw

Future Get Approach

Lastly, uncaught exceptions can then be handled by blocking on the *get()* function of the *Future*, which is returned after submitting the task. If a task submitted with *submit()* terminates with an exception, it is rethrown by *Future.get()* and wrapped in an *ExecutionException*.

```java
final class MyTask implements Runnable {
```