10 советов по использованию ExecutorService

Абстракция ExecutorService была представлена еще в Java 5. На дворе шел 2004 год… На секунду – сейчас Java 5 и 6 больше не поддерживаются и Java 7 готовится пополнить список. А многие Java-программисты по-прежнему не в полной мере понимают как работает ExecutorService. В вашем распоряжении множество источников, но сейчас я хотел бы рассказать о малоизвестных тонкостях и практиках по работе с ней.

**1. Именуйте пулы потоков**

Не могу не упомянуть об этом. При дампинге или во время дебаггинга можно заметить, что стандартная схема именования потока следующая: *pool-N-thread-M*, где N обозначает последовательный номер пула (каждый раз, когда вы создаете новый пул, глобальный счетчик N инкрементится), а M – порядковый номер потока в пуле. Например, *pool-2-thread-3* означает третий поток во втором пуле жизненного цикла JVM. См.: [Executors.defaultThreadFactory()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executors.html" \l "defaultThreadFactory--). Не очень информативно, не правда ли? JDK немного затрудняет правильное именование потоков, т.к. стратегия именования скрыта внутри [ThreadFactory](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadFactory.html). К счастью, Google Guava имеет встроенный класс для этого:

import com.google.common.util.concurrent.ThreadFactoryBuilder;

final ThreadFactory threadFactory = new ThreadFactoryBuilder()

.setNameFormat("Заказы-%d")

.setDaemon(true)

.build();

final ExecutorService executorService = Executors.newFixedThreadPool(10, threadFactory);

По умолчанию создаются *non-daemon*пулы потоков, решайте сами где какие уместнее.

**2. Изменяйте имена в зависимости от контекста**

Про этот трюк я узнал из статьи [«Supercharged jstack: How to Debug Your Servers at 100mph»](http://www.takipiblog.com/supercharged-jstack-how-to-debug-your-servers-at-100mph/). Раз мы знаем про имена потоков, мы можем менять их в рантайме, когда захотим! Это имеет смысл, поскольку дамп потока содержит имена классов и методов без параметров и локальных переменных. Включая некоторую важную информацию в имя потока, мы можем легко проследить какие сообщения/записи/запросы и т.п. тормозят систему или вызывают взаимную блокировку.

private void process(String messageId) {

executorService.submit(() -> {

final Thread currentThread = Thread.currentThread();

final String oldName = currentThread.getName();

currentThread.setName("Обработка-" + messageId);

try {

*//основная логика...* }

finally { currentThread.setName(oldName);

}

});

}

Внутри блока *try-finally* текущий поток называется*Обработка-ID-текущего-сообщения*, что может пригодиться при отслеживании потока сообщений в системе.

**3. Явное и безопасное завершение**

Между клиентскими потоками и пулом потоков лежит очередь заданий. Когда приложение завершает работу, вы должны побеспокоиться о двух вещах: что произойдет с заданиями, ожидающими в очереди, и как поведут себя уже выполняющиеся (об этом позже). Удивительно, но многие разработчики не закрывают пул потоков должным образом. Есть два способа: либо разрешите отработать всем задачам в очереди (*shutdown()*), либо удалите их (*shutdownNow()*) – в зависимости от конкретного случая. Например, если мы поставили в очередь набор задач и хотим вернуть управление как только все они выполнятся, используем *shutdown()*:

private void sendAllEmails(List<String> emails) throws InterruptedException {

emails.forEach(email -> executorService.submit(() -> sendEmail(email)));

executorService.shutdown();

final boolean done = executorService.awaitTermination(1, TimeUnit.MINUTES);

log.debug("Все ли письма были отправлены? {}", done);

}  
  
В этом примере мы отправляем пачку писем, каждое в виде отдельного задания для пула потоков. После постановки этих заданий в очередь мы закрываем пул, чтобы он больше не мог принять новых задач. Далее мы ждем максимум одну минуту пока все задания не будут выполнены. Однако, если какие-то задания еще не выполнены, *awaitTermination()* просто вернет *false*. Кроме того, оставшиеся задания продолжат выполняться. Знаю, хипстеры готовы пойти на:

emails.parallelStream().forEach(this::sendEmail);

Зовите меня старомодным, но мне нравится контролировать количество параллельных потоков. А альтернатива постепенному завершению *shutdown()* – это*shutdownNow()*:

final List<Runnable> rejected = executorService.shutdownNow();

log.debug("Отклоненные задачи: {}", rejected.size());

На этот раз все стоящие в очереди задачи отбрасываются и возвращаются. Уже запущенным задачам разрешено продолжить работу.

**4. Обрабатывайте прерывание потока с осторожностью**

Менее известная особенность интерфейса *Future*– возможность отмены. Далее приводится одна из моих предыдущих статей: [InterruptedException and interrupting threads explained](http://www.nurkiewicz.com/2014/05/interruptedexception-and-interrupting.html).  
Поскольку исключение [InterruptedException](http://docs.oracle.com/javase/8/docs/api/java/lang/InterruptedException.html)явно пробрасываемое (checked), никто, скорее всего, даже не задумывался о том, сколько ошибок оно подавило за все эти годы. И так как оно должно быть обработано, многие делают это неправильно или необдуманно. Давайте рассмотрим простой пример потока, который периодически делает некую очистку, а в промежутках большую часть времени спит.

class Cleaner implements Runnable {

Cleaner() {

final Thread cleanerThread = new Thread(this, "Чистильщик");

cleanerThread.start();

}

@Override

public void run() {

while(true) {

cleanUp();

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

*// TODO Auto-generated catch block*

e.printStackTrace();

}

}

}

private void cleanUp() {

*//...*

}

}

Этот код ужасен как ни крути!

1. Запуск потока из конструктора — чаще всего плохая идея. Например, некоторые фреймворки, такие как Spring, любят создавать динамические подклассы для поддержки перехвата методов. В конечном счете, мы получим два потока, запущенных из двух экземпляров.
2. Исключение *InterruptedException*проглочено, а не обработано как следует.
3. Этот класс запускает новый поток в каждом экземпляре. Вместо этого, он должен использовать [ScheduledThreadPoolExecutor](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html), который будет выдавать одни и те же потоки для многих объектов, что более надежно и эффективно.
4. Кроме того, с помощью *ScheduledThreadPoolExecutor*мы можем избежать написания циклов засыпания/работы и перейти к работе действительно по расписанию.
5. Последнее, но не менее важное. Нет никакого способа избавиться от этого потока, даже если на экземпляр Чистильщика больше никто не ссылается.

Все перечисленные проблемы важны, но подавление *InterruptedException*– наибольший грех. Прежде чем мы поймем почему, давайте подумаем для чего это исключение нужно и как мы можем использовать его преимущества, чтобы изящно прерывать потоки. Многие блокирующие операции в JDK обязывают обрабатывать *InterruptedException*, например:

* *Object.wait()*
* *Thread.sleep()*
* *Process.waitFor()*
* Множество блокирующих методов в*java.util.concurrent.\**, такие как *ExecutorService.awaitTermination(), Future.get(), BlockingQueue.take(), Semaphore.acquire() Condition.await()*и много, много других
* *SwingUtilities.invokeAndWait()*

Обратите внимание, что блокирующий ввод/вывод не пробрасывает InterruptedException (что прискорбно). Если все эти классы декларируют *InterruptedException*, вы можете быть удивлены, когда эти исключения будут брошены:

* Когда поток блокируется на каком-нибудь методе, декларирующем *InterruptedException*, и вы вызываете *Thread.interrupt()* на этом потоке, скорее всего блокирующий метод незамедлительно бросит *InterruptedException*.
* Если вы поставили задачу в очередь (*ExecutorService.submit()*) и вызвали *Future.cancel(true)* пока она еще выполняется, пул потоков постарается прервать поток, выполняющий эту задачу, эффективно завершив его.

Зная, что на самом деле представляет собой *InterruptedException*, мы сможем обработать его правильно. Если кто-то пытается прервать наш поток, и мы обнаружили это, обрабатывая *InterruptedException*, разумно будет разрешить завершить его незамедлительно, например:

class Cleaner implements Runnable, AutoCloseable {

private final Thread cleanerThread;

Cleaner() {

cleanerThread = new Thread(this, "Cleaner");

cleanerThread.start();

}

@Override

public void run() {

try { while (true) {

cleanUp();

TimeUnit.SECONDS.sleep(1);

}

} catch (InterruptedException ignored) {

log.debug("Interrupted, closing");

}

}

*//...* @Override

public void close() {

cleanerThread.interrupt();

}

}

Обратите внимание, что блок *try-catch*в данном примере окружает цикл *while*. Таким образом, если *sleep()* выбросит *InterruptedException*, мы прервем этот цикл. Вы можете возразить, что мы должны логировать стек исключения *InterruptedException*. Это зависит от ситуации. В данном случае прерывание потока является ожидаемым поведением, а не падением. В общем, на ваше усмотрение. В большинстве случаев поток прервется во время sleep() и мы быстренько завершим метод *run()* в это же время. Если вы очень осторожны, то наверняка спросите – а что будет, если поток прервется во время выполнения чистки *cleanUp()*? Зачастую вы столкнетесь с решением вручную выставить флаг, наподобие этого:

private volatile boolean stop = false;

@Override

public void run() {

while (!stop) {

cleanUp();

TimeUnit.SECONDS.sleep(1);

}

}

@Override

public void close() {

stop = true;

}

Помните, что стоп-флаг (он должен быть волатильным!) не будет прерывать блокирующие операции, мы должны дождаться пока отработает метод sleep(). С другой стороны, этот явный флаг дает нам лучший контроль, т.к. мы можем мониторить его в любое время. Оказывается, прерывание потоков работает точно так же. Если кто-то прервал поток, пока он выполнял неблокирующие вычисления (например, *cleanUp()*), такие вычисления не будут прерваны незамедлительно. Однако поток уже отмечен как прерванный, поэтому любая следующая блокирующая операция, такая как *sleep()* немедленно прервется и выбросит *InterruptedException*, поэтому мы не потеряем этот сигнал.  
  
Мы также можем воспользоваться этим фактом, если реализуем неблокирующий поток, который по-прежнему хочет использовать преимущества механизма прерывания потоков. Вместо того чтобы полагаться на *InterruptedException*, мы должны просто периодически проверять *Thread.isInterrupted():*

public void run() {

while (Thread.currentThread().isInterrupted()) {

someHeavyComputations();

}

}  
  
Как видите, если кто-то прервет наш поток, мы отменим вычисления так скоро, насколько позволят предыдущая итерация *someHeavyComputations*(). Если она выполняется очень долго или бесконечно, мы никогда не достигнем флага прерывания. Примечательно, что этот флаг не одноразовый. Мы можем вызвать *Thread.interrupted()*вместо*isInterrupted(),* что сбросит значение флага и мы сможем продолжить. Иногда вы можете захотеть проигнорировать флаг прерывания и продолжить выполнение. В этом случае *interrupted()* может пригодиться.  
  
Если вы олдскульный программист, вы наверняка помните метод *Thread.stop()*, который устарел 10 лет назад. В Java 8 были планы по его «деимплементации», но в 1.8u5 он по-прежнему с нами. Тем не менее, не используйте его и рефакторите любой код, в котором он встречается, используя *Thread.interrupt()*.  
  
Возможно, иногда вы захотите полностью проигнорировать InterruptedException. В этом случае обратите внимание на класс *Uninterruptibles*из Guava. Он содержит много методов таких как *sleepUninterruptibly()*или *awaitUninterruptibly(CountDownLatch).* Просто будьте осторожны с ними. Они не декларируют *InterruptedException*, но также полностью избавляют поток от прерывания, что довольно необычно.  
  
Итак, теперь у вас есть понимание того, почему некоторые методы бросают *InterruptedException*:

* Выброшенные *InterruptedException*должны быть адекватно обработаны в большинстве случаев.
* Подавление *InterruptedException*– зачастую плохая идея.
* Если поток был прерван во время неблокирующих вычислений. Используйте *isInterrupted*().

**5. Следите за длиной очереди и определяйте границу**

Пулы потоков неправильного размера могут привести к падению производительности, нестабильности и утечкам памяти. Если вы укажете слишком мало потоков, очередь будет расти, потребляя много памяти. С другой стороны, слишком много потоков будут замедлять всю систему из-за частых переключений контекста, что приведет к тем же симптомам. Важно сохранять глубину очереди и определять ее границы. А перегруженный пул может просто временно отказываться от новых задач.

final BlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(100);

executorService = new ThreadPoolExecutor(n, n, 0L, TimeUnit.MILLISECONDS, queue);

Вышеприведенный код эквивалентен *Executors.newFixedThreadPool(n)*, однако вместо того, чтобы использовать по умолчанию неограниченный *LinkedBlockingQueue*, мы используем *ArrayBlockingQueue*с фиксированной емкостью в 100. Это означает, что если 100 задач уже набраны, следующая задача будет отклонена с исключением *RejectedExecutionException*. Кроме того, поскольку очередь теперь доступна извне, мы можем периодически справляться о ее размере, чтобы записать в лог, отправить в JMX и т.д.

**6. Помните об обработке исключений**

Каков результат выполнения следующего кода?

executorService.submit(() -> {

System.out.println(1 / 0);

});

Я был озадачен тем, как много раз он ничего не печатал. Никаких признаков *java.lang.ArithmeticException: / by zero*, ничего. Пул потоков просто проглатывал исключение, как будто оно никогда не выбрасывалось. Если бы это был поток, созданный «с нуля», без обертки в виде пула, мог бы сработать [UncaughtExceptionHandler](https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.UncaughtExceptionHandler.html). Но с пулом потоков вы должны быть более осторожны. Если вы отправили на выполнение *Runnable*(без какого-либо результата, как выше), вы обязаны поместить все тело метода внутрь *try-catch*. Если вы помещаете в очередь Callable, удостоверьтесь, что вы всегда достаете его результат с помощью блокирующего get(), чтобы заново бросить исключение:

final Future<Integer> division = executorService.submit(() -> 1 / 0);

*//ниже будет выброшено ExecutionException, вызванное ArithmeticException*

division.get();

Примечательно, что даже в *Spring framework* допустили эту ошибку в [@Async](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/scheduling/annotation/Async.html), см.: [SPR-8995](https://jira.spring.io/browse/SPR-8995) и [SPR-12090](https://jira.spring.io/browse/SPR-12090).

**7. Следите за временем ожидания в очереди**

Мониторинг глубины рабочей очереди односторонний. При решении проблем с одиночной транзакцией/задачей, имеет смысл посмотреть сколько времени прошло между постановкой задачи и началом ее выполнения. Это время в идеале должно стремиться к нулю (когда в пуле имеется простаивающий поток), однако оно будет увеличиваться по мере постановки задач в очередь. Кроме того, если пул не имеет фиксированного числа потоков, запуск новой задачи может потребовать рождения нового потока, что тоже займет какое-то время. Чтобы четко измерять этот показатель, оберните оригинальный *ExecutorService*во что-то похожее:

public class WaitTimeMonitoringExecutorService implements ExecutorService {

private final ExecutorService target;

public WaitTimeMonitoringExecutorService(ExecutorService target) {

this.target = target;

}

@Override

public <T> Future<T> submit(Callable<T> task) {

final long startTime = System.currentTimeMillis();

return target.submit(() -> {

final long queueDuration = System.currentTimeMillis() - startTime;

log.debug("Задание {} провело в очереди {} мс", task, queueDuration);

return task.call(); } );

}

@Override

public <T> Future<T> submit(Runnable task, T result) {

return submit(() -> {

task.run();

return result;

});

}

@Override public Future<?> submit(Runnable task) {

return submit(new Callable<Void>() {

@Override

public Void call() throws Exception {

task.run();

return null;

}

});

}

*//...*

}

Это не полная реализация, но суть понятна. В момент, когда мы поставили задание в пул потоков, мы незамедлительно засекли время. Затем остановили секундомер, как только задача была извлечена и отправлена на выполнение. Не обманывайтесь близостью *startTime*и *queueDuration*в исходном коде. На самом деле эти две строки исполняются в разных потоках, в миллисекундах или даже в секундах друг от друга.

**8. Сохраняйте трассировку стека клиента**

Реактивному программированию в наши дни уделяется повышенное внимание: [Reactive manifesto](http://www.reactivemanifesto.org/), [reactive streams](http://www.reactive-streams.org/), [RxJava](https://github.com/ReactiveX/RxJava) (уже 1.0!), [Clojure agents](http://clojure.org/agents), [scala.rx](https://github.com/lihaoyi/scala.rx)… Все это выглядит здорово, но стектрейс – больше не ваш друг, он по большому счету бесполезен. Рассмотрим, к примеру, следующее исключение, возникающее во время выполнения задания в пуле потоков:

java.lang.NullPointerException: null

at com.nurkiewicz.MyTask.call(Main.java:76) ~[classes/:na]

at com.nurkiewicz.MyTask.call(Main.java:72) ~[classes/:na]

at java.util.concurrent.FutureTask.run(FutureTask.java:266) ~[na:1.8.0]

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142) ~[na:1.8.0]

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617) ~[na:1.8.0]

at java.lang.Thread.run(Thread.java:744) ~[na:1.8.0]

Мы можем легко заметить, что *MyTask*выбросило NPE в строке 76. Но мы не имеем никакого представления, кто утвердил эту задачу, поскольку стек относится только к *Thread*и *ThreadPoolExecutor*. Технически, мы можем просто перемещаться по коду в надежде найти только один участок, где выполняется постановка *MyTask*в очередь. Но без отдельных потоков (не говоря уже о событийно-ориентированном, реактивном и т.п. программировании), мы всегда видим сразу всю картину целиком. Что если мы могли бы сохранить стектрейс клиентского кода (того, что инициирует задание) и показать его, допустим, при возникновении ошибки? Идея не нова, например, [Hazelcast](http://hazelcast.com/)распространяет исключения из узла-владельца в клиентский код. Ниже приведен незамысловатый пример как сделать подобное:

public class ExecutorServiceWithClientTrace implements ExecutorService {

protected final ExecutorService target;

public ExecutorServiceWithClientTrace(ExecutorService target) {

this.target = target;

}

@Override

public <T> Future<T> submit(Callable<T> task) {

return target.submit(wrap(task, clientTrace(), Thread.currentThread().getName()));

}

private <T> Callable<T> wrap(final Callable<T> task, final Exception clientStack,

String clientThreadName) {

return () -> {

try {

return task.call();

} catch (Exception e) {

log.error("Исключение {} в задании из потока {}:", e, clientThreadName,

clientStack);

throw e;

}

};

}

private Exception clientTrace() {

return new Exception("Клиентский стектрейс");

}

@Override   
 public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks) throws

InterruptedException {

return tasks.stream().map(this::submit).collect(toList());

}

*//...*

}

В этот раз в случае неудачи, мы извлекаем полный стектрейс и название потока, где задание было поставлено в очередь. Гораздо более ценная информация по сравнению со стандартным исключением, рассмотренным ранее:  
Исключение java.lang.NullPointerException в задании из потока main:

java.lang.Exception: Клиентский стектрейс

at com.nurkiewicz.ExecutorServiceWithClientTrace.clientTrace(ExecutorServiceWithClientTrace.java:43) ~[classes/:na]

at com.nurkiewicz.ExecutorServiceWithClientTrace.submit(ExecutorServiceWithClientTrace.java:28) ~[classes/:na]

at com.nurkiewicz.Main.main(Main.java:31) ~[classes/:na]

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:1.8.0]

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62) ~[na:1.8.0]

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:1.8.0]

at java.lang.reflect.Method.invoke(Method.java:483) ~[na:1.8.0]

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:134) ~[idea\_rt.jar:na]

**9. Предпочитайте CompletableFuture**

В Java 8 был представлен более мощный класс [CompletableFuture](http://www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html). Пожалуйста, используйте его там, где это возможно. *ExecutorService*не был расширен, чтобы поддерживать эту абстракцию, так что вы должны заботиться об этом самостоятельно. Вместо:

final Future<BigDecimal> future = executorService.submit(this::calculate);

Используйте:

final CompletableFuture<BigDecimal> future = CompletableFuture.supplyAsync(this::calculate, executorService);

*CompletableFuture*расширяет *Future*, так что все работает как раньше. Но более продвинутые пользователи вашего API по-настоящему оценят расширенную функциональность, предоставляемую с помощью CompletableFuture.

**10. Синхронные очереди**

[SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html)– интересная разновидность *BlockingQueue*, которая на самом деле не совсем очередь. Это даже не структура данных как таковая. Лучше всего ее можно определить как очередь с нулевой емкостью.  
Вот что говорит *JavaDoc*:

Каждая добавляемая операция должна ожидать соответствующей операции удаления в другом потоке, и наоборот. Синхронная очередь не имеет никакой внутренней емкости, даже единичной. Вы не можете заглянуть в синхронную очередь, потому что элемент представлен только при попытке его удаления; вы не можете вставить элемент (используя любой метод), пока другой поток не удалит его: вы не можете обойти очередь потому что обходить нечего.  
Синхронные очереди похожи на «rendezvous channels», используемые в CSP и Ada.

Как все это относится к пулам потоков? Попробуем использовать *SynchronousQueue*вместе с *ThreadPoolExecutor*:

BlockingQueue<Runnable> queue = new SynchronousQueue<>();

ExecutorService executorService = new ThreadPoolExecutor(2, 2, 0L, TimeUnit.MILLISECONDS, queue);

Мы создали пул потоков с двумя потоками и *SynchronousQueue*перед этим. По сути *SynchronousQueue*— очереди с емкостью 0, поэтому такие *ExecutorService*будут только принимать новые задачи, если доступен простаивающий поток. Если все потоки заняты, новая задача будет немедленно отклонена и никогда не будет ждать очереди. Такой режим может быть полезен для незамедлительной обработки в фоновом режиме, если это возможно.  
  
Вот и все, надеюсь, вы открыли для себя как минимум одну интересную фичу!