Многопоточное программирование в Java 8. Часть первая. Параллельное выполнение кода с помощью потоков

Рассказывает Бенджамин Винтерберг (http://winterbe.com), Software Engineer

Добро пожаловать в первую часть руководства по параллельному программированию (https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB% в Java 8. В этой части мы на простых примерах рассмотрим, как выполнять код параллельно с помощью потоков, задач и сервисов исполнителей.

Впервые Concurrency API (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html) был представлен вместе с выходом Java 5 и с тех пор постоянно развивался с каждой новой версией Java. Большую часть примеров можно реализовать на более старых версиях, однако в этой статье я собираюсь использовать лямбда-выражения. Если вы все еще не знакомы с нововведениями Java 8, рекомендую посмотреть мое руководство (http://winterbe.com/posts/2014/03/16/java-8-tutorial/).

Потоки и задачи

Все современные операционные системы поддерживают параллельное выполнение кода с помощью процессов (https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8 и потоков (https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%82%D0%BE%D0%BA_%D0%B2%D1%8B%D0%BF Процесс — это экземпляр программы, который запускается независимо от остальных. Например, когда вы запускаете программу на Java, ОС создает новый процесс, который работает параллельно другим. Внутри процессов мы можем использовать потоки, тем самым выжав из процессора максимум возможностей.

Потоки (https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html) *(threads)* в Java поддерживаются начиная с JDK 1.0. Прежде чем запустить поток, ему надо предоставить участок кода, который обычно называется «задачей» *(task)*. Это делается через реализацию интерфейса `Runnable`, у которого есть только один метод без аргументов, возвращающий `void` — `run()`. Вот пример того, как это работает:

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();

System.out.println("Done!");
```

Поскольку интерфейс `Runnable` функциональный, мы можем использовать лямбда-выражения, которые появились в Java 8. В примере мы создаем задачу, которая выводит имя текущего потока на консоль, и запускаем ее сначала в главном потоке, а затем — в отдельном.

Результат выполнения этого кода может выглядеть так:

```
Hello main
Hello Thread-0
Done!
```

или так:

```
Hello main
Done!
Hello Thread-0
```

Из-за параллельного выполнения мы не можем сказать, будет наш поток запущен до или после вывода «Done!» на экран. Эта особенность делает параллельное программирование сложной задачей в больших приложениях.

Потоки могут быть приостановлены на некоторое время. Это весьма полезно, если мы хотим сэмулировать долго выполняющуюся задачу. Например, так:

```
Runnable runnable = () -> {
    try {
        String name = Thread.currentThread().getName();
        System.out.println("Foo " + name);
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Bar " + name);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Thread thread = new Thread(runnable);
thread.start();
```

Когда вы запустите этот код, вы увидите секундную задержку между выводом первой и второй строки на экран. `TimeUnit` — полезный класс для работы с единицами времени, но то же самое можно сделать с помощью `Thread.sleep(1000)`.

Работать с потоками напрямую неудобно и чревато ошибками. Поэтому в 2004 году в Java 5 добавили Concurrency API. Он находится в пакете `java.util.concurrent` и содержит большое количество полезных классов и методов для многопоточного программирования. С тех пор Concurrency API непрерывно развивался и развивается.

Давайте теперь подробнее рассмотрим одну из самых важных частей Concurrency API — сервис исполнителей *(executor services)*.

Исполнители

Concurrency API вводит понятие сервиса-исполнителя *(ExecutorService)* — высокоуровневую замену работе с потоками напрямую. Исполнители выполняют задачи асинхронно и обычно используют пул потоков, так что нам не надо создавать их вручную. Все потоки из пула будут использованы повторно после выполнения задачи, а значит, мы можем создать в приложении столько задач, сколько хотим, используя один исполнитель.

Вот как будет выглядеть наш первый пример с использованием исполнителя:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

// => Hello pool-1-thread-1
```

Класс `Executors` предоставляет удобные методы-фабрики для создания различных сервисов исполнителей. В данном случае мы использовали исполнитель с одним потоком.

Результат выглядит так же, как в прошлый раз. Но у этого кода есть важное отличие — он никогда не остановится. Работу исполнителей надо завершать явно. Для этого в интерфейсе `ExecutorService` есть два метода: `shutdown()`, который ждет завершения запущенных задач, и `shutdownNow()`, который останавливает исполнитель немедленно.

Вот как я предпочитаю останавливать исполнителей:

```
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}
finally {
    if (!executor.isTerminated()) {
        System.err.println("cancel non-finished tasks");
    }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

Исполнитель пытается завершить работу, ожидая завершения запущенных задач в течение определенного времени (5 секунд). По истечении этого времени он останавливается, прерывая все незавершенные задачи.

Callable и Future

Кроме `Runnable`, исполнители могут принимать другой вид задач, который называется `Callable`. `Callable` — это также функциональный интерфейс, но, в отличие от `Runnable`, он может возвращать значение.

Давайте напишем задачу, которая возвращает целое число после секундной паузы:

```
Callable task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

Callable-задачи также могут быть переданы исполнителям. Но как тогда получить результат, который они возвращают? Поскольку метод `submit()` не ждет завершения задачи, исполнитель не может вернуть результат задачи напрямую. Вместо этого исполнитель возвращает специальный объект Future, у которого мы сможем запросить результат задачи.

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

После отправки задачи исполнителю мы сначала проверяем, завершено ли ее выполнение, с помощью метода `isDone()`. Поскольку задача имеет задержку в одну секунду, прежде чем вернуть число, я более чем уверен, что она еще не завершена.

Вызов метода `get()` блокирует поток и ждет завершения задачи, а затем возвращает результат ее выполнения. Теперь `future.isDone()` вернет `true`, и мы увидим на консоли следующее:

```
future done? false
future done? true
result: 123
```

Задачи жестко связаны с сервисом исполнителей, и, если вы его остановите, попытка получить результат задачи выбросит исключение:

```
executor.shutdownNow();
future.get();
```

Вы, возможно, заметили, что на этот раз мы создаем сервис немного по-другому: с помощью метода `newFixedThreadPool(1)`, который вернет исполнителя с пулом в один поток. Это эквивалентно вызову метода `newSingleThreadExecutor()`, однако мы можем изменить количество потоков в пуле.

Таймауты

Любой вызов метода `future.get()` блокирует поток до тех пор, пока задача не будет завершена. В наихудшем случае выполнение задачи не завершится никогда, блокируя ваше приложение. Избежать этого можно, передав таймаут:

```
ExecutorService executor = Executors.newFixedThreadPool(1);

Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
});

future.get(1, TimeUnit.SECONDS);
```

Выполнение этого кода вызовет `TimeoutException`:

```
Exception in thread "main" java.util.concurrent.TimeoutException
    at java.util.concurrent.FutureTask.get(FutureTask.java:205)
```

Вы уже, возможно, догадались, почему было выброшено это исключение: мы указали максимальное время ожидания выполнения задачи в одну секунду, в то время как ее выполнение занимает две.

InvokeAll

Исполнители могут принимать список задач на выполнение с помощью метода `invokeAll()`, который принимает коллекцию callable-задач и возвращает список из `Future`.

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");

executor.invokeAll(callables)
    .stream()
    .map(future -> {
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);
```

В этом примере мы использовали функциональные потоки Java 8 для обработки задач, возвращенных методом `invokeAll`. Мы прошлись по всем задачам и вывели их результат на консоль. Если вы не знакомы с потоками (*streams*) Java 8, смотрите мое руководство (http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/).

InvokeAny

Другой способ отдать на выполнение несколько задач — метод `invokeAny()`. Он работает немного по-другому: вместо возврата `Future` он блокирует поток до того, как завершится хоть одна задача, и возвращает ее результат.

Чтобы показать, как работает этот метод, создадим метод, эмулирующий поведение различных задач. Он будет возвращать `Callable`, который вернет указанную строку после необходимой задержки:

```
Callable callable(String result, long sleepSeconds) {
    return () -> {
        TimeUnit.SECONDS.sleep(sleepSeconds);
        return result;
    };
}
```

Используем этот метод, чтобы создать несколько задач с разными строками и задержками от одной до трех секунд. Отправка этих задач исполнителю через метод `invokeAny()` вернет результат задачи с наименьшей задержкой. В данном случае это «task2»:

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
    callable("task1", 2),
    callable("task2", 1),
    callable("task3", 3));

String result = executor.invokeAny(callables);
System.out.println(result);

// => task2
```

В примере выше использован еще один вид исполнителей, который создается с помощью метода `newWorkStealingPool()`. Этот метод появился в Java 8 и ведет себя не так, как другие: вместо использования фиксированного количества потоков он создает `ForkJoinPool` (https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html) с определенным параллелизмом *(parallelism size)*, по умолчанию равным количеству ядер машины.

`ForkJoinPool` впервые появился в Java 7, и мы рассмотрим его подробнее в следующих частях нашего руководства. А теперь давайте посмотрим на исполнители с планировщиком *(scheduled executors)*.

Исполнители с планировщиком

Мы уже знаем, как отдать задачу исполнителю и получить ее результат. Для того, чтобы периодически запускать задачу, мы можем использовать пул потоков с планировщиком.

`ScheduledExecutorService` способен запускать задачи один или несколько раз с заданным интервалом.

Этот пример показывает, как заставить исполнитель выполнить задачу через три секунды:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());
ScheduledFuture<?> future = executor.schedule(task, 3, TimeUnit.SECONDS);

TimeUnit.MILLISECONDS.sleep(1337);

long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
System.out.printf("Remaining Delay: %sms", remainingDelay);
```

Когда мы передаем задачу планировщику, он возвращает особый тип `Future` — `ScheduledFuture`, который предоставляет метод `getDelay()` для получения оставшегося до запуска времени.

У исполнителя с планировщиком есть два метода для установки задач: `scheduleAtFixedRate()` и `scheduleWithFixedDelay()`. Первый устанавливает задачи с определенным интервалом, например, в одну секунду:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());

int initialDelay = 0;
int period = 1;
executor.scheduleAtFixedRate(task, initialDelay, period, TimeUnit.SECONDS);
```

Кроме того, он принимает начальную задержку, которая определяет время до первого запуска.

Обратите внимание, что метод `scheduleAtFixedRate()` не берет в расчет время выполнения задачи. Так, если вы поставите задачу, которая выполняется две секунды, с интервалом в одну, пул потоков рано или поздно переполнится.

В этом случае необходимо использовать метод `scheduleWithFixedDelay()`. Он работает примерно так же, как и предыдущий, но указанный интервал будет отсчитываться от времени завершения предыдущей задачи.

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        System.out.println("Scheduling: " + System.nanoTime());
    }
    catch (InterruptedException e) {
        System.err.println("task interrupted");
    }
};

executor.scheduleWithFixedDelay(task, 0, 1, TimeUnit.SECONDS);
```

В этом примере мы ставим задачу с задержкой в одну секунду между окончанием выполнения задачи и началом следующей. Начальной задержки нет, и каждая задача выполняется две секунды. Так, задачи будут запускаться на 0, 3, 6, 9 и т. д. секунде. Как видите, метод `scheduleWithFixedDelay()` весьма полезен, если мы не можем заранее сказать, сколько будет выполняться задача.