Методические указания 7

Многопоточность
Часть 2

Пулы потоков. Пакет java.util.concurrent

ExecutorService

Интерфейс java.util.concurrent.ExecutorService представляет собой механизм асинхронного выполнения, который способен выполнять задачи в фоновом режиме. Фактически, реализация ExecutorService из пакета java.util.concurrent, представляет собой реализацию пула потоков.

Вот простой пример использования ExecutorService:

```
ExecutorService executorService = Executors. newFixedThreadPool
(10 ) ; executorService.execute( new  Runnable() {     public
void  run() {
       System. out .println( "Асинхронная задача" );
   }
});
executorService.shutdown();
```

С помощью фабричного метода newFixedThreadPool() создается объект типа ExecutorService. Это пул потоков с 10 рабочими потоками, выполняющими задачи. В метод execute() передается реализация интерфейса Runnable, в которой прописана задача, передаваемая на выполнение одному из потоков ExecutorService.

Создание ExecutorService

Существует несколько типов ExecutorService, которые можно создать через класс Executors. newSingleThreadExecutor() создает пул в котором только один рабочий поток, то есть он может одновременно исполнять только одну задачу, но при каждом запуске не будет создавать новых потоков. newFixedThreadPool() создает пул с фиксированным количеством потоков, в примере ниже можно запустить одновременно выполнение не более 10 задач. newCachedThreadPool() создает пул, который может автоматически расширяться, если ему дать задачу и у него будут свободные потоки, пул отдаст задачу одному из таких потоков, если же в пуле свободных потоков нет, он создаст и запустит новый. Минусом такого подхода является то, что у cachedThreadPool нет верхней границы, и при высокой частоте появления новых задач он потенциально может создавать огромное количество потоков, пока не закончатся системные ресурсы. Ниже приведены примеры трех фабричных методов:

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
ExecutorService executorService2 = Executors.newFixedThreadPool( 10) ;
ExecutorService executorService3 = Executors.newCachedThreadPool();
```

Использование ExecutorService

Существует несколько различных способов делегирования задач для выполнения в ExecutorService:

- execute(Runnable)
- submit(Runnable)
- submit(Callable)
- invokeAny(...) ●invokeAll(...)

execute (Runnable)

Метод execute(Runnable) принимает объект java.lang.Runnable и выполняет его асинхронно. Пример:

```
ExecutorService executorService = Executors. newFixedThreadPool
(10 ) ; executorService.execute( new  Runnable() {     public
void  run() {
       System. out .println( "Асинхронная задача" );
   }
});
executorService.shutdown();
```

Получить из потока результат выполнения не получится, для этого нужно использовать интерфейс Callable, о чем пойдет речь ниже.

submit (Runnable)

Метод submit(Runnable) также принимает реализацию Runnable, но возвращает объект типа Future, который можно использовать для проверки завершенности выполнения задачи. Пример:

```
public static void  main(String[] args)  throws  Exception  {
    ExecutorService executorService = Executors.newFixedThreadPool (2 ) ;
    Future future = executorService.submit (new   Runnable() {
public void  run () {
            System . out . println ("Асинхронная задача"   );
}    });     future.get();   // вернет null если задача
завершилась корректно     executorService.shutdown(); }
```

submit (Callable)

Экземпляр Callable также позволяет дать потоку задачу, но в отличие от Runnable, его метод call() может возвращать результат. Результат Callable может быть получен через объект Future, возвращенный методом submit. Пример:

```
public static void  main(String[] args)  throws  Exception {
   ExecutorService executorService = Executors. newFixedThreadPool (2 ) ;
   Future future = executorService.submit( new
Callable(){         public  Object call()  throws
Exception {            System. out .println(
"Асинхронный вызов" );             return  "Результат
из потока" ;
       }
   });
   System. out .println( "future.get() = "  + future.get());
executorService.shutdown();
}
Результат:
Асинхронный вызов
future.get() = Результат из потока
```

shutdown() и shutdownNow()

Когда вы закончили использовать ExecutorService, его необходимо остановить, чтобы его свободные потоки прекратили свою работу. Например, если main поток завершил работу, а ExecutorService остался активным, активные потоки внутри ExecutorService не позволят JVM завершить работу приложения. Для завершения потоков внутри ExecutorService, необходимо вызвать метод shutdown(). ExecutorService не будет закрыт немедленно, но перестанет принимать новые задачи, и как только все потоки завершат текущие задачи, ExecutorService отключится.
Все задачи, отправленные в ExecutorService до shutdown(), выполняются. Если вы хотите немедленно закрыть ExecutorService, можно вызвать метод shutdownNow(), который попытается немедленно остановить все выполняемые задачи и пропустить все представленные, но не обработанные задачи.

Классы синхронизации

Semaphore

Semaphore ограничивает количество потоков при работе с ресурсами. Для этого служит счетчик. Если его значение больше нуля, то потоку разрешается доступ, а значение уменьшается. Если счетчик равен нулю, то текущий поток блокируется до освобождения ресурса. Для получения доступа используется метод **acquire()** , для освобождения – **release()** . Пример работы:

```
public class SemaphoreDemo {
    public static void main (String[]
args) {         Semaphore smp = new
Semaphore( 2) ;          for ( int i = 0;
i <         5; i++) {
             final int w = i;
new Thread(() -> {
                 try {
                    System.out.println( "Поток " + w +  " перед семафором"
);                     smp.acquire();
                    System.out.println( "Поток " + w +  " получил доступ к
ресурсу" );
                    Thread.sleep( 500) ;
                } catch (InterruptedException e) {
e.printStackTrace();
                } finally {
                    System.out.println( "Поток " + w +  " освободил
ресурс" );                         smp.release();
                }
            }).start();
        }
    }
}
```

```
Результат:
Поток 0 перед семафором
Поток 0 получил доступ к ресурсу
Поток 2 перед семафором
Поток 2 получил доступ к ресурсу
Поток 1 перед семафором
Поток 4 перед семафором
Поток 3 перед семафором
Поток 2 освободил ресурс
Поток 1 получил доступ к ресурсу
Поток 0 освободил ресурс
Поток 4 получил доступ к ресурсу
Поток 4 освободил ресурс
Поток 1 освободил ресурс
Поток 3 получил доступ к ресурсу
Поток 3 освободил ресурс
```

С помощью метода **acquire()** одновременно захватить семафор могут только два потока. Остальные становятся в очередь, пока один из «захватчиков» не освободит семафор методом **release()** .

CountDownLatch

CountDownLatch позволяет потоку ожидать завершения операций, выполняющихся в других потоках. Режим ожидания запускается методом **await()** . При создании объекта определяется количество требуемых операций, после чего уменьшается при вызове метода **countDown()** . Как только счетчик доходит до нуля, с ожидающего потока снимается блокировка.

```
public class SimpleCDL {
    public static void main (String[]
args) {        // задаем количество потоков
final int THREADS_COUNT = 6;          //
задаем значение счетчика
        final CountDownLatch cdl = new
CountDownLatch(THREADS_COUNT);         System.out.println( "Начинаем"
);
        for ( int i = 0; i < THREADS_COUNT;
i++) {                  final int w = i;
new Thread(() -> {                  try {
                    // считаем, что выполнение задачи занимает ~1 сек
Thread.sleep( 500 + ( int )(500 * Math.random()));
cdl.countDown();
                   // как только задача выполнена, уменьшаем счетчик
                    System.out.println( "Поток #" + w + " - готов"
);                 } catch (InterruptedException e) {
e.printStackTrace();
                }
            }).start();
        }
try {
            cdl.await();
// пока счетчик не приравняется нулю, будем стоять на этой строке
} catch (InterruptedException e) {          e.printStackTrace();
        }
// как только все потоки выполнили свои задачи - пишем сообщение
        System.out.println( "Работа завершена" );
    }
}
```

```
Результат:
Начинаем
Поток # 1 - готов
Поток # 0 - готов
Поток # 3 - готов
Поток # 2 - готов
Поток # 4 - готов
Поток # 5 - готов
Работа завершена
```

CyclicBarrier

Основной поток создает 6 потоков и ждет, пока каждый из них не закончит приготовление к работе.

CyclicBarrier выполняет синхронизацию заданного количества потоков в одной точке. Как только заданное количество потоков заблокировалось (вызовами метода **await()**), с них одновременно снимается блокировка.

```
public class BarrierExample { public
static void main (String[] args) {
CyclicBarrier cb = new CyclicBarrier( 3) ;
for ( int i = 0; i < 3; i++) {
final int w = i;
new Thread(() -> {
try {
System.out.println( "Поток " + w + " готовится" );
Thread.sleep( 100 + ( int ) ( 3000 * Math.random()));
System.out.println( "Поток " + w + " готов" );
cb.await();
System.out.println( "Поток " + w + " запустился" );
} catch (Exception e) {
e.printStackTrace();
}
}).start();
}
}
}
```

```
Результат:
Поток 0 готовится
Поток 1 готовится
Поток 2 готовится
Поток 2 готов
Поток 0 готов
Поток 1 готов
Поток 1 запустился
Поток 2 запустился
Поток 0 запустился
```

Потоки закончили подготовку в разное время, но стартовали вместе, так как блокировка снимается одновременно.

Lock

Интерфейс **Lock** из пакета **java.util.concurrent** – это продвинутый механизм синхронизации потоков. По гибкости он выигрывает в сравнении с блоками синхронизации. Для работы с этим интерфейсом необходимо создать объект одной из его реализаций:

```
Lock lock = new ReentrantLock();
lock.lock();
// критическая секция lock.unlock();
```

Создаем объект типа **Lock** и вызываем у него метод **lock()** – он захватывается. Если другой поток попытается вызвать этот метод у того же объекта – он будет блокирован до тех пор, пока поток, удерживающий объект **lock** , не освободит его через метод **unlock()** . Тогда этот объект смогут захватить другие потоки.

Основные отличия между Lock и синхронизированными блоками:

- Синхронизированные блоки не гарантируют, что сохранится порядок обращения потоков к критической секции;
- Нельзя выйти из синхронизированного блока по времени ожидания (**timeout**);
- Синхронизированные блоки должны полностью содержаться в одном методе. **Lock** может быть захвачен в одном методе, а освобожден в другом.

Методы интерфейса Lock:

lock()	Блокирует объект типа **Lock** , если это возможно. Если объект уже был заблокирован, то поток, вызвавший метод **lock()** , блокируется до вызова **unlock().**
tryLock()	Метод пытается заблокировать объект типа **Lock,** если это возможно. При удачном блокировании вернет «true». Если же Lock уже был заблокирован – то «false», и поток, вызвавший **tryLock()** , не будет заблокирован.
tryLock(long timeout, TimeUnit timeUnit)	Похож на **tryLock()** , но в течение заданного времени пытается захватить объект **Lock** .
unlock()	Разблокирует объект **Lock** . Вызывается только потоком, который владеет блокировкой **Lock** . При попытке других потоков обратиться к этому методу будет выдано исключение **RuntimeException** .

Интерфейс **java.util.concurrent.locks.ReadWriteLock** – это продвинутый механизм для блокировки потоков. Он позволяет множеству потоков одновременно читать данные, или только одному потоку – их записывать. Ресурс открыт для чтения множеству потоков без риска ошибок. Проблемный момент – если несколько потоков одновременно читают и записывают данные.

Правила работы ReadWriteLock

Read Lock	Если нет потоков, которые захватили объект этого типа для записи, то множество потоков могут захватить его для чтения.
Write Lock	Если нет потоков, которые захватили этот объект для записи или чтения, то только один поток может захватить его для записи.

Для работы с интерфейсом ReadWriteLock необходимо создать объект типа **ReentrantReadWriteLock**. Пример работы с ним:

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();
readWriteLock.readLock().lock();
// множество читателей может зайти в эту секцию,
// если нет записывающих потоков
readWriteLock.readLock().unlock();
readWriteLock.writeLock().lock();
// только один поток-писатель может зайти в эту секцию, // при
условии, что ни один из потоков не занимается чтением
readWriteLock.writeLock().unlock();
```

Практическое задание

1. Перенести приведенный ниже код в новый проект, где мы **организуем гонки.**

Все участники должны стартовать одновременно, несмотря на разное время подготовки. В тоннель не может одновременно заехать больше половины участников (условность).

Попробуйте все это синхронизировать.

Когда все завершат гонку, нужно выдать объявление об окончании.

Можно корректировать классы (в том числе конструктор машин) и добавлять объекты классов из пакета **util.concurrent.**

Обязательно необходимо объявить победителя гонки, он должен быть только один, и это участник первым закончивший последний этап.

```
public class MainClass { public
static final int CARS_COUNT = 4;
public static void main (String[] args)
{
        System.out.println( "ВАЖНОЕ ОБЪЯВЛЕНИЕ >>> Подготовка!!!" );
        Race race = new Race( new Road( 60) , new Tunnel(), new Road( 40)) ;
        Car[] cars = new Car[CARS_COUNT]; for ( int i =
0; i < cars.length; i++) { cars[i] = new Car(race,
20 + ( int ) (Math.random() * 10)) ;
        }
         for ( int i = 0; i < cars.length; i++)
{ new Thread(cars[i]).start();
        }
        System.out.println( "ВАЖНОЕ ОБЪЯВЛЕНИЕ >>> Гонка началась!!!" );
        System.out.println( "ВАЖНОЕ ОБЪЯВЛЕНИЕ >>> Гонка
закончилась!!!" ); } } public class Car implements Runnable {
private static int CARS_COUNT; static {
        CARS_COUNT = 0;
    }
```

```
     private Race race;
private  int speed;
private String name;
public String  getName () {
return name;
    }
     public  int  getSpeed () {
return speed;
    }      public  Car (Race race,  int
speed) {          this .race = race;
this .speed = speed;          CARS_COUNT++;
this .name =  "Участник #" + CARS_COUNT;
}
     @Override
     public  void  run () {
         try {
            System.out.println( this .name +  " готовится" );
            Thread.sleep( 500 + ( int )(Math.random() *  800)) ;
            System.out.println( this .name +  " готов" );
        }  catch (Exception e) {
            e.printStackTrace();          }          for (
int i =  0; i < race.getStages().size(); i++) {
race.getStages().get(i).go( this );          }
    } } public  abstract  class
Stage {      protected  int length;
protected String description;
public String  getDescription () {
return description;
    }
     public  abstract  void  go (Car c);
}
public  class  Road  extends  Stage {      public  Road
(int  length) {          this .length = length;
this .description =  "Дорога " + length +  " метров" ;
    }
     @Override      public
void  go (Car c) {
try {
            System.out.println(c.getName() +  " начал этап: " + description);
            Thread.sleep(length / c.getSpeed() *  1000) ;
            System.out.println(c.getName() +  " закончил этап: " +
description);          }  catch (InterruptedException e) {
e.printStackTrace();
        }
```

```
    } } public  class  Tunnel
extends  Stage {      public
Tunnel () {          this .length
=  80;
         this .description =  "Тоннель " + length +  "
метров" ;     }
     @Override
     public  void  go (Car c) {
         try {
try {
                System.out.println(c.getName() +  " готовится к этапу(ждет): "
+ description);
                System.out.println(c.getName() +  " начал этап: " +
description);
                Thread.sleep(length / c.getSpeed() *  1000)
;             }  catch (InterruptedException e) {
e.printStackTrace();
            }  finally {
                System.out.println(c.getName() +  " закончил этап:
" + description);             }
        }  catch (Exception e) {
            e.printStackTrace();
        }
    }
}
public  class  Race {
     private ArrayList<Stage> stages;      public
ArrayList<Stage>  getStages () {  return stages; }
public  Race (Stage... stages) {          this .stages =
new ArrayList<>(Arrays.asList(stages));     }
}
```

Пример выполнения кода до корректировки:

```
ВАЖНОЕ ОБЪЯВЛЕНИЕ >>> Подготовка!!! Участник #2 готовится
ВАЖНОЕ ОБЪЯВЛЕНИЕ >>> Гонка началась!!!
ВАЖНОЕ ОБЪЯВЛЕНИЕ >>> Гонка закончилась!!!
Участник #1 готовится
Участник #4 готовится
Участник #3 готовится
Участник #3 готов
Участник #3 начал этап: Дорога 60 метров
Участник #2 готов
Участник #2 начал этап: Дорога 60 метров
Участник #1 готов
Участник #1 начал этап: Дорога 60 метров
Участник #4 готов
Участник #4 начал этап: Дорога 60 метров
Участник #3 закончил этап: Дорога 60 метров
```

```
Участник #3 готовится к этапу(ждет): Тоннель 80 метров
Участник #3 начал этап: Тоннель 80 метров
Участник #2 закончил этап: Дорога 60 метров
Участник #2 готовится к этапу(ждет): Тоннель 80 метров
Участник #2 начал этап: Тоннель 80 метров
Участник #1 закончил этап: Дорога 60 метров
Участник #1 готовится к этапу(ждет): Тоннель 80 метров
Участник #1 начал этап: Тоннель 80 метров
Участник #4 закончил этап: Дорога 60 метров
Участник #4 готовится к этапу(ждет): Тоннель 80 метров
Участник #4 начал этап: Тоннель 80 метров
Участник #2 закончил этап: Тоннель 80 метров
Участник #2 начал этап: Дорога 40 метров
Участник #3 закончил этап: Тоннель 80 метров
Участник #3 начал этап: Дорога 40 метров
Участник #2 закончил этап: Дорога 40 метров
Участник #1 закончил этап: Тоннель 80 метров
Участник #1 начал этап: Дорога 40 метров
Участник #4 закончил этап: Тоннель 80 метров
Участник #4 начал этап: Дорога 40 метров
Участник #3 закончил этап: Дорога 40 метров
Участник #1 закончил этап: Дорога 40 метров
Участник #4 закончил этап: Дорога 40 метров
```

Примерный результат:

```
ВАЖНОЕ ОБЪЯВЛЕНИЕ >>> Подготовка!!!
Участник #2 готовится
Участник #1 готовится
Участник #4 готовится
Участник #3 готовится
Участник #2 готов
Участник #4 готов
Участник #1 готов
Участник #3 готов
ВАЖНОЕ ОБЪЯВЛЕНИЕ >>> Гонка началась!!!
Участник #2 начал этап: Дорога 60 метров
Участник #4 начал этап: Дорога 60 метров
Участник #3 начал этап: Дорога 60 метров
Участник #1 начал этап: Дорога 60 метров
Участник #1 закончил этап: Дорога 60 метров
Участник #3 закончил этап: Дорога 60 метров
Участник #3 готовится к этапу(ждет): Тоннель 80 метров
Участник #1 готовится к этапу(ждет): Тоннель 80 метров
Участник #1 начал этап: Тоннель 80 метров
Участник #3 начал этап: Тоннель 80 метров
Участник #4 закончил этап: Дорога 60 метров
Участник #4 готовится к этапу(ждет): Тоннель 80 метров
Участник #2 закончил этап: Дорога 60 метров
Участник #2 готовится к этапу(ждет): Тоннель 80 метров
Участник #3 закончил этап: Тоннель 80 метров
Участник #1 закончил этап: Тоннель 80 метров
Участник #2 начал этап: Тоннель 80 метров
Участник #4 начал этап: Тоннель 80 метров
Участник #3 начал этап: Дорога 40 метров
Участник #1 начал этап: Дорога 40 метров
Участник #3 закончил этап: Дорога 40 метров
Участник #3 - WIN
Участник #1 закончил этап: Дорога 40 метров
Участник #4 закончил этап: Тоннель 80 метров
Участник #4 начал этап: Дорога 40 метров
```

```
Участник #2 закончил этап: Тоннель 80 метров
```

```
Участник #2 начал этап: Дорога 40 метров
Участник #2 закончил этап: Дорога 40 метров
Участник #4 закончил этап: Дорога 40 метров ВАЖНОЕ ОБЪЯВЛЕНИЕ >>> Гонка закончилась!!!
```

Дополнительные материалы

1. Кей С. Хорстманн, Гари Корнелл Java. Библиотека профессионала. Том 1. Основы // Пер. с англ. — М.: Вильямс, 2014. — 864 с.

2. Брюс Эккель. Философия Java // 4-е изд.: Пер. с англ. — СПб.: Питер, 2016. — 1 168 с.

3. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 1 376 с.

4. Г. Шилдт. Java 8: Руководство для начинающих. // 6-е изд.: Пер. с англ. — М.: Вильямс, 2015. — 720 с.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Г. Шилдт. Java 8. Полное руководство // 9-е изд.: Пер. с англ. — М.: Вильям с , 2015. — 1 376 с.