# Лабораторная работа №2

## *ExecutorService в Java*

### 1. Введение

ExecutorService – это интерфейс Java, который предоставляет удобный способ управления потоками исполнения. Он позволяет создавать пул потоков исполнения и выполнять задачи в этих потоках. Существует несколько ключевых реализаций ExecutorService:

ThreadPoolExecutor
ScheduledThreadPoolExecutor
CachedThreadPoolExecutor
ForkJoinPool

Рассмотрим каждый из них детально с примером их использования.

### 1.1 ThreadPoolExecutor

ThreadPoolExecutor – это реализация интерфейса ExecutorService в Java, который предоставляет пул потоков исполнения для выполнения задач в фоновом режиме.

ThreadPoolExecutor позволяет создать и настроить пул потоков исполнения с определенным количеством потоков, а также управлять очередью задач. Он может быть использован для выполнения задач в фоновом режиме, что может улучшить производительность и реактивность вашего приложения.

ThreadPoolExecutor управляет пулом потоков исполнения, которые используются для выполнения задач. При создании ThreadPoolExecutor вы указываете минимальное и максимальное количество потоков, которые могут быть использованы для выполнения задач, а также время жизни потока.

ThreadPoolExecutor хранит очередь задач, которые ожидают выполнения в пуле потоков исполнения. По умолчанию используется очередь с ограниченной ёмкостью, но вы можете создать свою собственную реализацию очереди задач, если требуется.

ThreadPoolExecutor определяет способ обработки задач, которые не могут быть выполнены из-за переполнения очереди или ошибки в выполнении задачи. По умолчанию используется обработчик, который выбрасывает исключение, но вы можете создать свой собственный обработчик задач.

ThreadPoolExecutor имеет свой жизненный цикл, который может быть управляем через методы start(), shutdown() и awaitTermination(). Вы можете использовать эти методы для запуска и остановки ThreadPoolExecutor и ожидания завершения выполнения всех задач.

Один из наиболее ярких примеров использования ThreadPoolExecutor – это выполнение параллельной обработки изображений. Предположим, у вас есть большой набор изображений, которые нужно обработать, например, изменить их размер, добавить водяной знак и сохранить в другой формат.

```
public class ImageProcessor {
    private final ThreadPoolExecutor executor;

    public ImageProcessor() {
        // Создаем ThreadPoolExecutor с фиксированным количеством потоков
        this.executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(4);
    }

    public void processImages(List<File> imageFiles) {
        // Добавляем задачи обработки каждого изображения в ThreadPoolExecutor
        for (File imageFile : imageFiles) {
```

```
            executor.execute(new ImageTask(imageFile));
        }

        // Останавливаем выполнение задач и закрываем ThreadPoolExecutor
        executor.shutdown();
    }

    private static class ImageTask implements Runnable {
        private final File imageFile;

        public ImageTask(File imageFile) {
            this.imageFile = imageFile;
        }

        @Override
        public void run() {
            // Код для обработки изображения
            System.out.println("Обработка изображения: " + imageFile.getName());
        }
    }
}
```

Пример класса-клиента:

```
public class ImageProcessorDemo {
    public static void main(String[] args) {
        // Создаем ImageProcessor
        ImageProcessor imageProcessor = new ImageProcessor();
        // Создаем список файлов изображений, которые нужно обработать
        List<File> imageFiles = new ArrayList<>();
        imageFiles.add(new File("/path/to/image1.jpg"));
        imageFiles.add(new File("/path/to/image2.jpg"));
        imageFiles.add(new File("/path/to/image3.jpg"));
        imageFiles.add(new File("/path/to/image4.jpg"));
        // Обрабатываем изображения
        imageProcessor.processImages(imageFiles);
    }
}
```

### 1.2 ScheduledThreadPoolExecutor

ScheduledThreadPoolExecutor – это реализация интерфейса ScheduledExecutorService в Java, который предоставляет пул потоков исполнения для выполнения задач в определенный момент времени или с определенной периодичностью.

ScheduledThreadPoolExecutor позволяет создать и настроить пул потоков исполнения с определенным количеством потоков, а также управлять очередью задач. Он может быть использован для выполнения регулярных задач в фоновом режиме, что может улучшить производительность и реактивность вашего приложения.

ScheduledThreadPoolExecutor управляет пулом потоков исполнения, которые используются для выполнения задач. При создании ScheduledThreadPoolExecutor вы указываете минимальное и максимальное количество потоков, которые могут быть использованы для выполнения задач, а также время жизни потока.

ScheduledThreadPoolExecutor хранит очередь задач, которые ожидают выполнения в пуле потоков исполнения. По умолчанию используется очередь с ограниченной ёмкостью, но вы можете создать свою собственную реализацию очереди задач, если требуется.

ScheduledThreadPoolExecutor определяет способ обработки задач, которые не могут

быть выполнены из-за переполнения очереди или ошибки в выполнении задачи. По умолчанию используется обработчик, который выбрасывает исключение, но вы можете создать свой собственный обработчик задач.

ScheduledThreadPoolExecutor имеет свой жизненный цикл, который может быть управляем через методы start(), shutdown() и awaitTermination(). Вы можете использовать эти методы для запуска и остановки ScheduledThreadPoolExecutor и ожидания завершения выполнения всех задач.

ScheduledThreadPoolExecutor предоставляет методы для выполнения регулярных задач в определенный момент времени или с определенной периодичностью. Вы можете использовать методы schedule() и scheduleAtFixedRate() для запуска задач в определенное время и с определенной периодичностью соответственно.

Один из наиболее ярких примеров использования ScheduledThreadPoolExecutor – это выполнение регулярных задач в определенный промежуток времени. Рассмотрим пример резервного копирования файлов с определенной периодичностью.

Вот пример кода, который использует ScheduledThreadPoolExecutor для выполнения регулярных задач резервного копирования файлов:

```
public class BackupService {
    private final ScheduledExecutorService scheduler;

    public BackupService() {
        // Создаем ScheduledThreadPoolExecutor с 1 потоком
        this.scheduler = Executors.newScheduledThreadPool(1);
    }

    public void startBackup(String sourceDir, String destDir, long period, TimeUnit timeUnit) {
        // Создаем новую задачу BackupTask
        BackupTask task = new BackupTask(sourceDir, destDir);

        // Запускаем задачу каждый период времени с использованием ScheduledThreadPoolExecutor
        scheduler.scheduleAtFixedRate(task, 0, period, timeUnit);
    }

    public void stopBackup() {
        // Останавливаем выполнение задач и закрываем пул потоков исполнения
        scheduler.shutdown();
    }

    private static class BackupTask implements Runnable {
        private final String sourceDir;
        private final String destDir;

        public BackupTask(String sourceDir, String destDir) {
            this.sourceDir = sourceDir;
            this.destDir = destDir;
        }

        @Override
        public void run() {
            // Код для резервного копирования файлов
            System.out.println("Резервное копирование файлов...");
        }
    }
}
```

Пример класса-клиента:

```
public class BackupServiceDemo {
    public static void main(String[] args) {
        // Создаем BackupService
        BackupService backupService = new BackupService();

        // Запускаем резервное копирование каждые 5 секунд
        backupService.startBackup("/path/to/source", "/path/to/destination", 5, TimeUnit.SECONDS);

        // Ждем 20 секунд
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Останавливаем выполнение резервного копирования
        backupService.stopBackup();
    }
}
```

### 1.3 CachedThreadPoolExecutor

CachedThreadPoolExecutor – это реализация интерфейса ExecutorService в Java, который предоставляет пул потоков исполнения для выполнения задач в фоновом режиме.

CachedThreadPoolExecutor автоматически масштабирует количество потоков исполнения в зависимости от количества задач, которые нужно выполнить. Если задачи поступают слишком быстро и текущее количество потоков исполнения не может справиться с ними, CachedThreadPoolExecutor создает новый поток исполнения, чтобы обеспечить выполнение задачи. Если задачи не поступают достаточно быстро и текущее количество потоков исполнения избыточно, CachedThreadPoolExecutor автоматически удаляет потоки исполнения, чтобы уменьшить нагрузку на систему.

CachedThreadPoolExecutor управляет пулом потоков исполнения, которые используются для выполнения задач. При создании CachedThreadPoolExecutor не нужно указывать количество потоков, так как он автоматически масштабируется в зависимости от количества задач.

CachedThreadPoolExecutor не хранит очередь задач, которые ожидают выполнения в пуле потоков исполнения. Вместо этого он немедленно создает новые потоки исполнения для каждой задачи, если текущее количество потоков исполнения не может справиться с нагрузкой.

CachedThreadPoolExecutor определяет способ обработки задач, которые не могут быть выполнены из-за ошибки в выполнении задачи. По умолчанию используется обработчик, который выбрасывает исключение, но вы можете создать свой собственный обработчик задач.

CachedThreadPoolExecutor имеет свой жизненный цикл, который может быть управляем через методы start(), shutdown() и awaitTermination(). Вы можете использовать эти методы для запуска и остановки CachedThreadPoolExecutor и ожидания завершения выполнения всех задач.

Рассмотрим пример, в котором мы будем обрабатывать множество задач с помощью CachedThreadPoolExecutor. Предположим, у нас есть список файлов, которые нужно скопировать с одного места на другое.

```
public class FileCopyManager {
    private final ExecutorService executor;
```

```
    public FileCopyManager() {
        this.executor = Executors.newCachedThreadPool();
    }

    public void copyFiles(List<File> sourceFiles, List<File> destFiles) {
        for (int i = 0; i < sourceFiles.size(); i++) {
            FileCopyTask task = new FileCopyTask(sourceFiles.get(i), destFiles.get(i));
            executor.execute(task);
        }
        executor.shutdown();
    }

    private static class FileCopyTask implements Runnable {
        private final File sourceFile;
        private final File destFile;

        public FileCopyTask(File sourceFile, File destFile) {
            this.sourceFile = sourceFile;
            this.destFile = destFile;
        }

        @Override
        public void run() {
            // Копирование файла
            System.out.println("Копирование файла " + sourceFile.getName() + " в " + destFile.getName());
        }
    }
}
```

Пример класса-клиента:

```
public class FileCopyManagerDemo {
    public static void main(String[] args) {
        List<File> sourceFiles = Arrays.asList(new File("source/file1.txt"), new File("source/file2.txt"), new File("source/file3.txt"));
        List<File> destFiles = Arrays.asList(new File("destination/file1.txt"), new File("destination/file2.txt"), new File("destination/file3.txt"));

        FileCopyManager copyManager = new FileCopyManager();
        copyManager.copyFiles(sourceFiles, destFiles);
    }
}
```

## 1.4 ForkJoinPool

ForkJoinPool – это реализация ExecutorService в Java, которая используется для параллельного выполнения задач. Эти задачи могут быть разбиты на более мелкие подзадачи. Она позволяет использовать принцип “разделяй и властвуй” для более эффективного использования многопроцессорных и многопоточных систем.

ForkJoinPool управляет пулом потоков исполнения, которые используются для выполнения задач. Количество потоков исполнения в пуле задается при создании экземпляра ForkJoinPool. Каждый поток в пуле имеет свой собственный стек вызовов, что позволяет ForkJoinPool улучшить производительность в случае, когда задачи могут быть разбиты на более мелкие подзадачи.

ForkJoinPool поддерживает работу с задачами типа RecursiveAction и RecursiveTask, которые представляют собой рекурсивно делимые задачи без возвращаемого значения и с

возвращаемым значением соответственно. Когда ForkJoinPool получает задачу типа RecursiveTask, он разбивает ее на более мелкие подзадачи, выполняет их параллельно в разных потоках и объединяет результаты выполнения в единую итоговую задачу.

ForkJoinPool имеет механизм обработки ошибок, который позволяет определить, как следует обрабатывать ошибки, возникающие во время выполнения задач. По умолчанию, если задача вызывает исключение, то ForkJoinPool прерывает выполнение всех задач и выбрасывает исключение ForkJoinTask.ExceptionalCompletion. Однако, этот механизм может быть настроен на более гибкую обработку ошибок.

ForkJoinPool имеет свой жизненный цикл, который может быть управляем через методы start(), shutdown() и awaitTermination(). Вы можете использовать эти методы для запуска и остановки ForkJoinPool и ожидания завершения выполнения всех задач.

Одним из ярких примеров использования ForkJoinPool является задача вычисления факториала числа с помощью рекурсии.

Вот пример кода, который использует ForkJoinPool для вычисления факториала числа:

```
public class FactorialCalculator extends RecursiveTask<Integer> {
   private final int number;
   public FactorialCalculator(int number) {
      this.number = number;
   }
   @Override
   protected Integer compute() {
      if (number <= 1) {
         return 1;
      } else {
         FactorialCalculator subTask = new FactorialCalculator(number - 1);
         subTask.fork();
         return number * subTask.join();
      }
   }
   public static void main(String[] args) {
      ForkJoinPool pool = ForkJoinPool.commonPool();
      FactorialCalculator task = new FactorialCalculator(10);
      int result = pool.invoke(task);
      System.out.println(result);
   }
}
```

Здесь мы создаем класс FactorialCalculator, который наследуется от класса RecursiveTask. Метод compute() переопределен для вычисления факториала числа.

Внутри метода compute() мы проверяем, является ли число меньше или равным 1. Если да, то возвращаем 1. В противном случае мы создаем новый экземпляр FactorialCalculator с числом, на 1 меньше, чем исходное число. Затем мы запускаем эту подзадачу с помощью метода fork() и ждем его завершения с помощью метода join(). Затем мы возвращаем результат умножения числа на результат выполнения подзадачи.

В методе main() мы создаем ForkJoinPool и вызываем метод invoke() с нашей задачей FactorialCalculator. Этот метод блокируется до тех пор, пока задача не завершится, и возвращает результат.

## Заключение

Некоторые из преимуществ использования ExecutorService включают в себя:

– Повторное использование потоков в пуле, что может снизить накладные расходы по сравнению с созданием новых потоков для каждой задачи.

– Ограничение количества потоков, используемых для группы задач, что позволяет

избежать нехватки ресурсов и повысить общую производительность системы.

– Управление рабочими очередями для управления потоком задач, что может уменьшить конкуренцию и повысить скорость реагирования.

В Java интерфейс ExecutorService имеет несколько реализаций, включая ThreadPoolExecutor, ScheduledThreadPoolExecutor и ForkJoinPool.

## 2. Порядок выполнения работы

В соответствии с **вариантом** выполните следующее основное задание:

1. Создайте новый проект в среде программирования.
2. Реализуйте с помощью ThreadPoolExecutor выполнение параллельной обработки изображений. Например, изменить их размер, добавить водяной знак или сохранить в другой формат.
3. Реализуйте возможность резервного копирования файлов с определенной периодичностью с помощью ScheduledThreadPoolExecutor.
4. Реализуйте копирование группы файлов с помощью CachedThreadPoolExecutor.
5. Реализуйте рекурсивное вычисление суммы ряда с помощью ForkJoinPool.
6. Подготовьте отчет о выполнении лабораторной работы:

Для успешной сдачи лабораторной работы необходимо:

1. представить преподавателю отлаженный код программы;
2. подготовить отчет по работе.

## 3. Порядок оформления отчета

Отчет о выполнении лабораторной работы должен содержать:

1) титульный лист;
2) задание;
3) текст программы;
4) результаты работы программы.