Пулы потоков выполнения в Java

Шаблон Thread Pool помогает экономить ресурсы в многопоточном приложении и удерживать параллелизм в определенных пределах.

Когда мы используем пул потоков, **мы пишем наш конкурентный код в виде параллельных задач и отправляем их на выполнение экземпляру пула потоков.** Этот экземпляр управляет несколькими потоками, повторно используемыми для выполнения этих задач.

1. **Executor Framework**

- ***Интерфейс Executor*** предоставляет стандартный способ отправки задач на выполнение. Он отделяет отправку задач от их выполнения, позволяя гибко управлять потоками выполнения.
- ***Интерфейс ExecutorService*** расширяет Executor и добавляет методы для управления жизненным циклом исполнителя, такие как закрытие пула и ожидание завершения.
- ***Класс утилит Executors*** предоставляет фабричные методы для создания различных типов экземпляров ExecutorService, включая пулы фиксированных потоков, пулы кэшированных потоков и исполнители с одним потоком выполнения.

Пример использования ExecutorService:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);

// Создает пул потоков с 5 потоками

        for (int i = 0; i < 10; i++) {
            final int taskId = i;
            executor.submit(() -> {

// Отправляет задачу (Runnable) исполнителю
                System.out.println("Executing task " + taskId + " in thread: "
+ Thread.currentThread().getName());
            });
        }

        executor.shutdown(); // Инициирует упорядоченное завершение
    }
}
```

2. **Пул потоков (ThreadPoolExecutor)**

• ***Пул потоков — это набор рабочих потоков***, которые могут выполнять несколько задач. Он повторно использует существующие потоки, сокращая дополнительные затраты на создание и уничтожение потоков для каждой задачи.

• ***ThreadPoolExecutor — это*** основной ***класс*** для создания и управления пользовательскими пулами потоков. Он позволяет детально контролировать такие параметры, как размер базового пула, максимальный размер пула, время поддержания активности и тип очереди.

Пример ThreadPoolExecutor:

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ThreadPoolExample {
    public static void main(String[] args) {
        ThreadPoolExecutor executor = new ThreadPoolExecutor(
                2, // Размер CorePool
                 5, // Максимальный размер пула
                60, TimeUnit.SECONDS, // время поддержания в рабочем состоянии
                new ArrayBlockingQueue<>(10) // Очередь работ );

        for (int i = 0; i < 15; i++) {
            final int taskId = i;
            executor.submit(() -> {
                System.out.println("Выполнение задачи " + taskId + " в потоке:
" + Thread.currentThread().getName());
                try {
                    Thread.sleep(100); // Симулировать работу
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }

        executor.shutdown();
    }
}
```

3. Фреймворк Fork/Join

• Фреймворк Fork/Join — это реализация ExecutorService, оптимизированная для задач, которые можно рекурсивно разделить на более мелкие подзадачи (разделение и ограничение).

• Использует алгоритм кражи работы, в котором неактивные потоки выполнения могут «красть» задачи из очередей занятых потоков выполнения, обеспечивая эффективное использование ресурсов. • Ключевыми классами являются ForkJoinPool, RecursiveTask (для задач, возвращающих результат) и RecursiveAction (для задач, не возвращающих результат).

Пример фреймворка Fork/Join:

```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class ForkJoinExample {
    static class SumTask extends RecursiveTask<Long> {
        private final long[] array;
        private final int start;
        private final int end;
        private static final int THRESHOLD = 1000;

        public SumTask(long[] array, int start, int end) {
            this.array = массив;
            this.start = start;
            this.end = end;
        }

 @Override
```

```
        protected Long compute() {
            if (end - start <= THRESHOLD) {
                long sum = 0;
                for (int i = start; i < end; i++) {
                    sum += array[i];
                }
                return sum;
            } else {
                int mid = start + (end - start) / 2;
                SumTask leftTask = new SumTask(array, start, mid);
                SumTask rightTask = new SumTask(array, mid, end);

                leftTask.fork(); // Выполняет асинхронную подзадачу слева
                long rightResult = rightTask.compute();

// Выполняет синхронную подзадачу правильно
                long leftResult = leftTask.join();

// Ожидайте и получите результат левой подзадачи
                return leftResult + rightResult;
            }
        }
    }

    public static void main(String[] args) {
        long[] array = new long[100000];
        for (int i = 0; i < array.length; i++) {
            array[i] = i + 1;
        }

        ForkJoinPool pool = new ForkJoinPool();
        long sum = pool.invoke(new SumTask(array, 0, array.length));
        System.out.println("Общая сумма: " + sum);
        pool.shutdown();
    }
}
```

Лабораторная работа № 4

Тема работы: ***Пулы потоков выполнения в Java. Синхронизация потоков в Java.***

Цель работы:

Изучение и применение механизмов управления и синхронизации потоков выполнения в Java с использованием **пулов потоков** и **классической концепции «производитель-потребитель»** для обеспечения эффективного и безопасного параллельного выполнения процессов.

Задачи работы:

4. **Понимание концепции пула потоков** и преимуществ использования класса `ExecutorService` по сравнению с ручным созданием потоков.
5. **Реализация модели «производитель-потребитель»** с использованием механизмов синхронизации (`wait()`, `notify()`, `synchronized`, `BlockingQueue`).
6. **Применение класса `Executors`** для создания фиксированного пула потоков (`FixedThreadPool`).
7. **Анализ конкурентного поведения** и проблем синхронизации (блокировка, интерференция, одновременный доступ).

8. **Разработка практического Java-приложения,** в котором несколько производителей и потребителей совместно используют общий репозиторий.
9. **Измерение производительности** при разных размерах пула потоков и емкости хранилища.

Пример реализации:

```
import java.util.concurrent.*;
import java.util.*;
import java.util.concurrent.atomic.AtomicInteger;

public class ProducerConsumerExecutorFixed {

    private static final int BUFFER_CAPACITY = 9;
    private static final BlockingQueue<Character> buffer = new
ArrayBlockingQueue<>(BUFFER_CAPACITY);

    private static final int CONSUMER_GOAL = 13;
    private static final int PRODUCER_COUNT = 3;
    private static final int CONSUMER_COUNT = 4;

    private static final int TOTAL_OBJECTS = CONSUMER_GOAL * CONSUMER_COUNT; // 52

    // Атомные счетчики
    private static final AtomicInteger totalProduced = new AtomicInteger(0);
    private static final AtomicInteger totalConsumed = new AtomicInteger(0);
    private static final Map<Integer, AtomicInteger> consumerCounters = new
ConcurrentHashMap<>();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(PRODUCER_COUNT +
CONSUMER_COUNT);

        for (int i = 1; i <= CONSUMER_COUNT; i++) {
            consumerCounters.put(i, new AtomicInteger(0));
        }

        // Запускаем производителей
        for (int i = 1; i <= PRODUCER_COUNT; i++) {
            executor.execute(new Producer(i));
        }

        // Запускаем потребителей
        for (int i = 1; i <= CONSUMER_COUNT; i++) {
            executor.execute(new Consumer(i));
        }

        executor.shutdown();

        try {
            // Ждем, пока все потоки завершатся
            executor.awaitTermination(60, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("\n========== ИТОГОВЫЙ ОТЧЕТ ==========");
        System.out.println("Общее количество произведенных продуктов: " +
totalProduced.get());
        System.out.println("Всего потреблено: " + totalConsumed.get());
        consumerCounters.forEach((id, count) ->
```

```
                System.out.println("Потребитель " + id + ": " + count.get() + "
объектов потреблено"));

    }

    // ================================
    // КЛАСС ПРОИЗВОДИТЕЛЬ
    // ================================
    static class Producer implements Runnable {
        private final int id;
        private final Random random = new Random();
        private final char[] consonants = "BCDFGHJKLMNPQRSTVWXYZ".toCharArray();

        Producer(int id) {
            this.id = id;
        }

        @Override
        public void run() {
            try {
                while (totalProduced.get() < TOTAL_OBJECTS) {
                    char item = consonants[random.nextInt(consonants.length)];

                    if (buffer.remainingCapacity() == 0) {
                        System.out.println("⚠️  [Производитель " + id + "] Склад
заполнен, ожидайте...");
                    }

                    buffer.put(item);
                    int produced = totalProduced.incrementAndGet();

                    System.out.println("🧱 [Производитель " + id + "] произвел: " +
item +
                            " | Всего произведено: " + produced +
                            " | Текущая емкость: " + buffer.size() + "/" +
BUFFER_CAPACITY);
                 // Thread.sleep(200);

                    if (produced >= TOTAL_OBJECTS) {
                          System.out.println("Производитель  " + id + " завершил
работу");
                          break;
                    }

                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    // ================================
    // КЛАСС ПОТРЕБИТЕЛЬ
    // ================================
    static class Consumer implements Runnable {
        private final int id;

        Потребитель(int id) {
            this.id = id;
        }

        @Override
```

```
        public void run() {
            try {
                while (consumerCounters.get(id).get() < CONSUMER_GOAL) {
                    if (buffer.isEmpty()) {
                        System.out.println("⚠️  [Потребитель " + id + "] Склад пуст,
ожидайте...");
                    }

                    char item = buffer.take();
                    consumerCounters.get(id).incrementAndGet();
                    int consumed = totalConsumed.incrementAndGet();

                    System.out.println("🍽️  [Потребитель " + id + "] потреблял: " +
item +
                            " | Всего потреблено: " + consumerCounters.get(id).get()
+
                            " | В складе: " + buffer.size() +
                            " | Всего: " + consumed);

                    Thread.sleep(300);
                }

                System.out.println("✅ [Потребитель " + id + "] был удовлетворен 13
объектами!");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

Задачи для решения:

Даны X производителей, которые случайным образом производят F объектов, которые потребляются Y потребителями. Необходимо отобразить информацию о производстве и потреблении объектов, сообщения о случаях, когда «склад пуст или полон». Все операции выполняются до тех пор, пока каждый потребитель не будет удовлетворен Z объектами.

Размер склада равен D. Значения для X, Y, Z, D, F указаны в таблице 3.

Таблица 1 Варианты выполнения задачи

№	X	Y	Z	D	Тип Объекты
1	2	3	11	8	Четные числа
2	3	4	2	5	Нечетные числа
3	4	3	3	10	Вокал
4	3	2	12	11	Согласные
5	2	5	3	12	Четные числа
6	2	4	4	7	Нечетные числа
7	3	3	5	6	Вокал

8	4	2	4	5	Согласные
9	5	2	3	4	Четные числа
10	3	3	5	2	Нечетные числа
F - каждый производитель производит по 2 предмета за раз для всех вариантов					

Критерии оценки:

1. Создание и инициализация потоков для выполнения задач.
2. Выбор форм синхронизации потоков.
3. Синхронизация потоков с подходящими формами.
4. Создание графического интерфейса программы.
5. **Корректность кода** — проверка правильности кода, отсутствие ошибок в работе.
6. **Соблюдение инструкций и требований** — проверка правильности требований задачи, таких как количество потоков выполнения.
7. **Оптимизация кода** — оценка эффективности кода в использовании ресурсов и избегание кода.
8. **Соблюдение срока сдачи** — оценка баллов в зависимости от пунктуальности, сдана ли работа в установленный срок.
9. **Оценка знаний** — объяснения, данные о процессе выполнения работы, что может включать описание основных функций и использованной логики.
10. Использование студентом ИИ.

Для получения оценки 5 - 6 обязательны критерии 1,2,3,5,8,9

Для получения оценки 7-8 обязательны критерии 1, 2, 3, 4, 5, 6, 8, 9.

Для получения оценки 9-10 обязательны критерии 1-9.

Если был использован критерий 10, оценка снижается на 2 балла, только если студент разъяснил принцип работы кода. В противном случае лабораторная работа не зачитывается.

Контрольные вопросы

1. Что такое *пул потоков* и каковы его преимущества по сравнению с ручным созданием потоков?
2. Какую роль играют интерфейс `Executor` и класс `ExecutorService` в Java?
3. Какие основные методы предлагает `ExecutorService` для управления потоками?
4. В чем заключается проблема производитель-потребитель?
5. Как можно реализовать эту проблему с помощью `BlockingQueue`?
6. В чем разница между `synchronized`, `ReentrantLock` и механизмами из `java.util.concurrent`?
7. Что представляют собой методы `wait()`, `notify()`, `notifyAll()` и при каких условиях они могут быть вызваны?
8. Как правильно завершить выполнение пула потоков (`shutdown()` vs `shutdownNow()`)?
9. Что происходит, если количество производителей превышает количество потребителей?

10. Каковы основные проблемы синхронизации, которые могут возникнуть в параллельной системе?

Список рекомендуемой литературы

1. Балауреа, М. – *Параллельное и распределенное программирование на Java*, Издательство Технического университета, Кишинев, 2022.
2. Моц, И. – *Продвинутое программирование на Java. Потоки выполнения и синхронизация*, Издательство Polirom, Яссы, 2020.
3. Попа, К. – *Конкуренция и параллелизм в Java*, Издательство MatrixRom, Бухарест, 2018.
4. Официальная документация Oracle (раздел *«Конкуренция и исполняющая* среда»): https://docs.oracle.com/javase/tutorial/essential/concurrency/
5. Хорстманн К. – *Java. Библиотека профессионала. Том 2: Расширенные средства программирования*, СПб: Питер, 2021.
6. Шилдт Г. – *Java: Руководство для начинающих*, 12-е издание, Москва: Вильямс, 2022.
7. Головач С., Романчук В. – *Java. Многопоточность. Синхронизация и параллельность*, Киев, 2020.
8. Документация Oracle по многопоточности: https://docs.oracle.com/javase/tutorial/essential/concurrency/
9. Гетц, Б., Пейерлс, Т. – *Java Concurrency in Practice*, Addison-Wesley, 2006.
10. Bloch, J. – *Effective Java*, 3-е издание, Addison-Wesley, 2018.
11. Oracle Docs – *Утилиты и исполнители параллелизма*: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
12. Lea, D. – *Параллельное программирование в Java: принципы и шаблоны проектирования*, Addison-Wesley, 2-е издание, 2000.