aboba

Содержание

Введение

В современных вычислительных системах всё большее значение приобретают параллельные и асинхронные методы обработки данных [1]. Использование многопоточности позволяет выполнять независимые задачи одновременно, повышая общую эффективность программы и снижая время отклика.

Java предоставляет мощные средства для организации параллельных вычислений, включая такие высокоуровневые компоненты, как `ExecutorService`, `Callable` и `Future` [2]. Данные инструменты позволяют запускать задачи в отдельных потоках, получать результаты их выполнения и контролировать процесс работы без необходимости вручную управлять потоками.

В данной работе демонстрируется пример вычислительной задачи, выполняемой асинхронно: генерация массива случайных чисел и проверка наличия заданного элемента. Подобный подход может применяться для выполнения тяжёлых вычислений, анализа данных, поиска решений в больших пространствах или обработки потоков информации, не блокируя основной поток программы.

1 JAVA код

1.1 Описание программы

Программа реализует следующий алгоритм:

а) Задаются размер массива `size` и искомое значение `target`.

б) Определяется функциональный объект `BiPredicate`, который принимает на вход размер массива и число, а возвращает логическое значение, показывающее, содержится ли данное число в массиве.

в) Внутри предиката создаётся массив случайных чисел от 0 до 99.

г) С помощью стримов Java производится проверка наличия числа `target`.

д) Задача (в виде `Callable`) передаётся в `ExecutorService`, который создаёт отдельный поток для выполнения проверки.

е) Основной поток ожидает завершения задачи, периодически выводя символ `.`, имитируя «ожидание» результата.

ж) После завершения задачи результат извлекается через объект `Future` и выводится в консоль.

1.2 Текст программы

```
1   public class second_lab {
2       public static void main(String[] args) throws Exception {
3           int size = 100;
4           int target = 42;
5
6           BiPredicate<Integer, Integer> containsNumber = (n, x) -> {
7               Random rand = new Random();
8               int[] arr = new int[n];
9
10              for (int i = 0; i < n; i++) {
11                  arr[i] = rand.nextInt(100);
12              }
13
14              System.out.println("Generated array: " +
                    Arrays.toString(arr));
15              return Arrays.stream(arr).anyMatch(i -> i == x);
16          };
17
18          ExecutorService executor = Executors.newSingleThreadExecutor();
```

```
19
20              Callable<Boolean> task = () -> containsNumber.test(size,
                    target);
21              Future<Boolean> future = executor.submit(task);
22
23              while (!future.isDone()) {
24                  System.out.println('.');
25                  Thread.sleep(100);
26              }
27
28              boolean result = future.get();
29
30              executor.shutdown();
31
32              System.out.println("\nNumber " + target+ (result ? " is in" :
                    " is not in") + " in array");
33          }
34      }
```

1.3 Результаты выполнения

При запуске программы создаётся отдельный поток, который выполняет задачу генерации массива и поиска числа. Основной поток при этом остаётся активным и может выполнять другие действия — в примере он выводит в консоль точки, показывая процесс ожидания результата.

Пример вывода:

```
Сгенерированный массив: [13, 42, 56, 7, 88, ...]
...
Число 42 найдено в массиве
```

1.4 Преимущества подобного подхода

— **Асинхронность:** выполнение длительных операций без блокировки основного потока.

— **Масштабируемость:** возможность использовать несколько потоков и пул задач.

— **Безопасность:** контроль за завершением задач и корректным освобождением ресурсов.

— **Гибкость:** возможность интеграции с другими потоковыми и реактивными моделями (например, `CompletableFuture` или `Reactive Streams`).

Заключение

Демонстрируемая программа показывает, как можно использовать средства многопоточности Java для реализации простых вычислительных задач, не перегружая основной поток исполнения. Подобный метод организации кода особенно эффективен в системах, где требуется обрабатывать большое количество независимых событий или данных — например, при моделировании, статистическом анализе или асинхронных вычислениях.

Таким образом, применение `ExecutorService`, `Callable` и `Future` обеспечивает удобный и надёжный способ организации параллельных вычислений, открывая широкие возможности для масштабируемых приложений.

Список использованных источников

1. *Зубкова, В. В.* Анализ актуальности закона Мура / В. В. Зубкова // *Перспективы развития информационных технологий*. — 2014. — no. 21. — Дата обращения: 06.10.2025. URL: https://cyberleninka.ru/article/n/analiz-aktualnosti-zakona-mura.

2. *Сьерра, К.* Изучаем Java: [пер. с англ.] / К. Сьерра, Б. Бейтс. Мировой компьютерный бестселлер. — Эксмо, 2012. URL: https://books.google.ru/books?id=qEZ1kQEACAAJ.