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 с передачей параметров: строки и искомого символа. 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 работы программы

Данная программа демонстрирует использование класса `CompletableFuture` в языке Java для асинхронного выполнения задачи — подсчёта количества вхождений заданного символа в строке. Код иллюстрирует основы реактивного подхода: запуск вычисления в отдельном потоке и обработку результата после его завершения без блокировки основного потока.

1.2 Основные параметры

— `input` — входная строка:

```
"Вели медведя на ратное поле. Нет не на цепи
- по собственной воле. За чужую свободу рвать
убивать."
```

— `target_char = 'e'` — символ, количество вхождений которого требуется подсчитать.

1.3 Описание функций

Метод `countLetters(String str, char ch)` Этот метод последовательно проходит по всем символам строки, считая количество совпадений с указанным символом. Алгоритм:

$$\text{count} = |\{c \in \text{str} \mid c = ch\}|$$

Реализован через цикл `for-each` по массиву символов:

```
int i = 0;
for (char c : str.toCharArray()) {
 if (c == ch) i++;
}
return i;
```

1.4 Асинхронное выполнение

а) Объявляется объект `CompletableFuture<Integer>`:

```
future = CompletableFuture.supplyAsync(() -> countLetters(input, targe
```

Здесь `supplyAsync()` запускает задачу в отдельном потоке (используя общий `ForkJoinPool.commonPool()`).

б) После запуска вычисления регистрируется обработчик завершения:

```
future.whenComplete((result, exception) -> {
 if (exception != null)
 System.out.println("Ошибка при работе: " + exception.getMessage());
 else
 System.out.println("Количество вхождений символа: " + result);
});
```

Этот метод выполняется, когда будущее (`future`) завершается:

— если задача выполнена без ошибок — печатается количество вхождений символа;

— если произошла ошибка — выводится сообщение об исключении.

в) Основной поток программы приостанавливается вызовом:

```
Thread.sleep(1000);
```

чтобы дать асинхронной задаче время завершиться до завершения метода `main()`.

1.5 Код методов класса на java

```
public static int countLetters(String str, char ch) {
 int i = 0;
 for (char c : str.toCharArray()){
  if (c == ch) i++;
 }
```

```
 return i;
}

public static void main(String[] args) {
 String input = "Вели медведя на ратное поле.
 Нет не на цепи - по собственной воле.
 За чужую свободу рвать убивать.";
 char target\_char = 'e';

 CompletableFuture<Integer> future =
 CompletableFuture.supplyAsync(() -> countLetters(input, target\_char

 future.whenComplete((result, exception) -> {
  if (exception != null) {
   System.out.println("Ошибка при работе: " + exception.getMessage())
  } else {
   System.out.println("Количество вхождений символа: " + result);
  }
 });

 try {
  Thread.sleep(1000);
 }catch (InterruptedException e) {
  e.printStackTrace();
 }
}
```

1.6 Вывод программы

При запуске на стандартной JVM выводится сообщение вида:

```
Количество вхождений символа: 14
```

(число может меняться в зависимости от регистра и текста входной строки).

1.7 Ключевые концепции

— **CompletableFuture** — класс, предоставляющий средства асинхронных вычислений с возможностью обработки результата по завершении;

— **Метод supplyAsync()** — запускает задачу в пуле потоков без блокировки основного потока;

— **Метод whenComplete()** — задаёт колбэк, который вызывается автоматически при завершении задачи;

— **Асинхронное программирование** — модель исполнения, при которой операции выполняются параллельно, а результат обрабатывается по готовности.

Заключение

Программа демонстрирует:

— создание и использование `CompletableFuture`;

— реактивную обработку результата без явного ожидания завершения потока;

— использование функционального интерфейса `Supplier<T>` для передачи логики в асинхронный поток.

Таким образом, данный пример иллюстрирует современный подход к организации асинхронных вычислений в Java без необходимости вручную управлять потоками.

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.