УДК 004.438JAVA DOI: 10.31673/2412-9070.2023.051519

К. П. СТОРЧАК, доктор техн. наук. професор;
О. М. ТКАЛЕНКО, канд. техн. наук, доцент;
Ю. О. БУРДА, студент;
Д. О. ГАРНИК, студент,
Державний університет інформаційно-комунікаційних технологій, Київ

ДОСЛІДЖЕННЯ ПРОБЛЕМИ ОПТИМІЗАЦІЇ ПРОДУКТИВНОСТІ В МОВІ ПРОГРАМУВАННЯ JAVA

Мова програмування Java, відома і широко використовувана, надає розробникам потужну платформу для створення програм, які можуть масштабуватися і швидко реагувати на вимоги користувачів. Однак для використання повного потенціалу Java слід докладніше дізнатися про можливості оптимізації. У запропонованій статті досліджено принципи, техніки та найкращі практики, які допомагають розробникам Java створювати програми, що відзначаються швидкістю, ефективністю та реагуванням. Також розглянуто інструменти та стратегії, які допомагають виявляти проблемні місця, налаштовувати код і поліпшувати загальну продуктивність Java-застосунків.

Ключові слова: Java; алгоритм; оптимізація; вебзастосунки; мови програмування.

Вступ

Оптимізація продуктивності в мові програмування Java є фундаментальною проблемою для розробників, які прагнуть створювати високопродуктивні та реактивні застосунки. Java відома своєю платформною незалежністю та універсальністю, надає розробникам потужну платформу для створення надійних програмних рішень. Однак досягнення оптимальної продуктивності в застосунках Java потребує глибокого розуміння нюансів мови та використання різних технік оптимізації.

У цій статті ми розглянемо ключові стратегії, найкращі практики та інструменти, які допоможуть розкрити повний потенціал Java, забезпечуючи ефективну роботу застосунків та відповідність вимогам сучасних обчислювальних середовищ.

Оптимізація продуктивності застосунків — це ієрархічний проєкт або метод, який вимагає високого рівня технологічної експертизи від інженерів. Основна структура має у своєму складі не лише код застосунку, а й операційну систему, зберігання, мережу, файлову систему, а також контейнер або віртуальну машину. Тому коли онлайн-застосунок має проблеми з продуктивністю, нам потрібно розглядати багато різних факторів і складнощів [1].

Водночас крім питань із продуктивністю, спричинених проблемами на низькому рівні коду, багато проблем із продуктивністю також глибоко містяться в самому застосунку і важко відшукуються. Щоб їх розв'язати, нам потрібно мати робоче знання про підмодулі, фреймворки та компоненти, які використовуються застосунком, а також деякі поширені інструменти для оптимізації продуктивності.

Основна частина

Як було вже зазначено, Java — це потужна мова програмування, яка дає змогу розробникам створювати надійні та масштабовані застосунки. Однак навіть досвідчені розробники можуть зробити програмний вибір, який призведе до проблем із продуктивністю. Ось деякі з найпоширеніших помилок, які можуть впливати на продуктивність застосунків Java.

Створення об'єктів. Створення занадто багатьох об'єктів може призвести до значного зниження продуктивності, оскільки формування об'єктів та їх збирання сміттям можуть бути витратними процесами на вашій платформі Java.

Операції з рядками. Рядки є незмінними в Java, а отже, кожен раз, коли ви змінюєте рядок, створюється новий об'єкт рядка. Це може швидко призвести до зупинок у продуктивності особливо при роботі з великими обсягами даних. Повторне конкатенування рядків за допомогою оператора «+» — це загальний приклад неефективного використання операцій над рядками.

Цикли. Цикли є важливою частиною програмування на Java, але погано оптимізовані цикли можуть значно впливати на продуктивність. Однією з поширених помилок є ітерація за колекціями через цикл «for-each», який створює об'єкт ітератору позаду сцени, що призводить до непотрібного розроблення об'єктів.

Регулярні вирази. Припустимо, ми виводимо функцію пошуку, яка використовує регулярний вираз для відповідності шаблонам у великому наборі даних. Складні регулярні вирази можуть бути витратними з погляду ресурсів і впливати на продуктивність, особливо якщо набір даних великий.

© К. П. Сторчак, О. М. Ткаленко, Ю. О. Бурда, Д. О. Гарник, 2023

Перш ніж намагатися здійснювати мікрооптимізацію конкретного шляху коду, варто подумати про поточний вибір коду.

Іноді фундаментальний підхід може бути неправильним, тобто навіть якщо ми прикладемо багато зусиль і зможемо зробити його на 20% швидшим за всіма можливими оптимізаціями, зміна підходу (використання кращого алгоритму) може зумовити покращення на порядок або значне збільшення продуктивності.

Це часто трапляється, коли масштаб даних, над якими ми працюємо, змінюється — досить легко написати рішення, яке діє добре зараз, але коли ми працюємо з великими обсягами реальних даних, воно починає давати збої [2].

Наприклад, припустимо, у нас є програма, яка має знайти максимальне значення у великому масиві. Один підхід може полягати в тому, щоб пройти весь масив і порівняти кожний елемент із поточним максимумом (рис. 1).

Хоча цей простий підхід працює добре для невеликих масивів, він може бути дуже повільним для дуже великих масивів, особливо якщо максимальне значення розміщено близько до кінця масиву.

```
public int findMax(int[] arr) {
    int max = 0;
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}
```

Рис. 1. Цикл проходження масиву

Кращим підходом може бути розділення масивів на менші підмасиви, визначення максимального значення в кожному підмасиві паралельно, а потім об'єднання результатів для здобуття максимального значення загалом (рис. 2) [3].

```
public int findMax(int[] arr) {
    int numThreads = Runtime.getRuntime().availableProcessors();
    int chunkSize = arr.length / numThreads;
    int[] maxValues = new int[numThreads];
    Thread[] threads = new Thread[numThreads];

    for (int i = 0; i < numThreads; i++) {
        int start = i * chunkSize;
        int end = (i == numThreads - 1) ? arr.length : (i + 1) * chunkSize;
        threads[i] = new Thread(() -> {
            int max = 0;
            for (int j = start; j < end; j++) {
                if (arr[j] > max) {
                    max = arr[j];
                }
            }
            maxValues[i] = max;
        });
        threads[i].start();
    }

    for (int i = 0; i < numThreads; i++) {
        try {
            threads[i].join();
        } catch (InterruptedException e) {
            // handle exception
        }
    }

    int max = 0;
    for (int i = 0; i < numThreads; i++) {
        if (maxValues[i] > max) {
            max = maxValues[i];
        }
    }
    return max;
}
```

Рис. 2. Розділення масиву на менші підмасиви (оптимальний підхід)

Цей код, зважаючи на кількість доступних процесорів, спочатку розділяє масив на менші підмасиви, а потім створює окремий потік для пошуку максимального значення в кожному підмасиві. Максимальні значення для кожного підмасиву зберігаються в масиві, а далі об'єднуються, щоб знайти максимальне значення загалом [4].

Цей підхід може бути набагато швидшим для великих масивів, оскільки він використовує паралельне оброблення для швидшого пошуку максимального значення. Цей нескладний приклад показує, що, просто переглянувши свій підхід до проблеми, ми часто можемо знайти більш ефективні рішення, які не потребують такої інтенсивної оптимізації коду.

Потоки (*Streams*) — це чудове доповнення до мови програмування Java, яке дає нам змогу легко переносити складні патерни з циклів «for» у загальні, більш повторно використовувані блоки коду з гарантіями сталості. Проте цей зручний підхід не є безкоштовним. Використання потоків пов'язане із втратою продуктивності.

Зазвичай ця втрата продуктивності не є дуже великою — в кращому разі ми виграємо кілька відсотків у швидкості, а в гіршому разі для загальних операцій швидкість може бути на 10-30% повільнішою. Про це варто знати. У 99% випадків втрата продуктивності від використання потоків компенсується поліпшеною зрозумілістю коду. Проте для тих 1% випадків, коли ми використовуємо потік всередині активного циклу, потрібно бути обережними щодо компромісу між продуктивністю та читабельністю.

Це особливо стосується дуже продуктивних застосунків. Збільшене споживання пам'яті та виділення з потокового API згідно з [5] може спричинити достатньо великий надмірний тиск на пам'ять, що потребує більш частого запуску збирача сміття (GC) й істотно впливає на продуктивність самого застосунку.

Паралельні потоки (*Parallel streams*) — це інша річ. Незважаючи на їхню легкість використання, їх слід застосовувати тільки в конкретних сценаріях і лише після профілювання паралельних і послідовних операцій для підтвердження, що паралельна операція дійсно швидша. На менших наборах даних (вартість обчислень у потоці визначає, що є меншим набором даних) вартість поділу роботи, її розкладання на інші потоки та знову її збирання, коли потік було оброблено, переважатиме прискорення від паралельних обчислень (рис. 3) [6].

Також потрібно брати до уваги середовище виконання нашого коду. Якщо він працює вже в сильно паралелізованому середовищі (наприклад, на вебсайті), то, ймовірно, ми навіть не отримаємо прискорення від паралельного виконання потоку. Навіть навантаженням це може бути гірше, ніж виконання без паралелізації. Через паралельну природу робочого навантаження, імовірно, запуск застосунку вже використовує максимально можливу кількість доступних ядер ЦП, що означає, що ми сплачуємо вартість розбиття даних без вигоди від збільшення обчислювальної потужності.

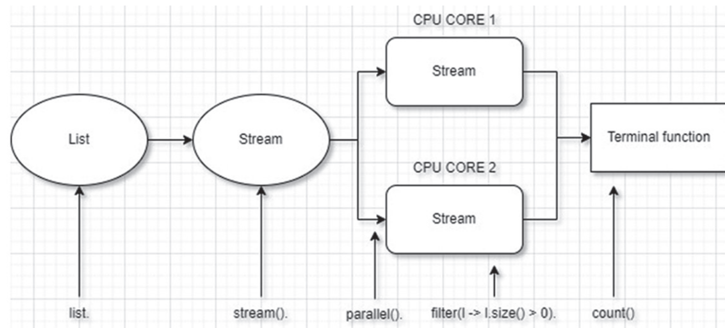

Рис. 3. Діаграма Parallel streams

Ось приклади розроблених бенчмарків. Тестовий список (testList) — це масив із 100 000 елементів, які є числами від 1 до 100 000, перетвореними в рядки і перемішаними (рис. 4).

```
// ~1,500 op/s
no usages  new *
public void testStream(ArrayList<String> state) {
    List<String> collect = state
            .stream()
            .filter(s -> s.length() > 5)
            .map(s -> "Value: " + s)
            .sorted(String::compareTo)
            .toList();
}

// ~1,500 op/s
no usages  new *
public void testFor(ArrayList<String> state) {
    ArrayList<String> results = new ArrayList<>();

    for (int i = 0;i < state.size();i++) {
        String s = state.get(i);

        if (s.length() > 5) {
            results.add("Value: " + s);
        }
    }  You, 2 minutes ago • Uncommitted changes

    results.sort(String::compareTo);
}

// ~8,000 op/s
// Note, with an array size of 10,000 and more variable load on my CPU this was 1/3rd as fast as testStream
no usages  new *
public void testStreamParrallel(ArrayList<String> state) {
    List<String> collect = state
            .stream()
            .parallel()
            .filter(s -> s.length() > 5)
            .map(s -> "Value: " + s)
            .sorted(String::compareTo)
            .toList();
}
```

Рис. 4. Тестовий список

Підсумовуючи, доходимо висновку, що потоки (Streams) здебільшого є великим досягненням для підтримання коду та його зрозумілості з незначним впливом на продуктивність. Проте варто бути обізнаним із накладними витратами для рідкісних випадків, коли потрібно видобути додаткову продуктивність із жорсткого циклу.

Якщо говорити про StringBuilder, то за рекомендаціями, які можна побачити в інтернеті, його використання за межами циклу здавалося б, мало би бути логічним. Але тестування показало, що насправді це було втричі повільніше, ніж використання «+=» або використання StringBuilder — навіть коли це не в циклі. До того ж, якщо використання «+=» у цьому контексті передається у виклики StringBuilder за допомогою javac, він все одно на вигляд набагато швидший, ніж пряме використання StringBuilder (рис. 5).

```
// ~20,000,000 operations p/s
no usages   new *
public String stringAppend() {
    String s = "foo";
    s += ", bar";
    s += ", baz";
    s += ", qux";
    s += ", bar";
    s += ", bar";
    s += ", bar";
    s += ", bar";
    s += ", bar";
    s += ", bar";
    s += ", baz";
    s += ", qux";
    s += ", baz";
    s += ", qux";
    s += ", baz";
    s += ", qux";
    s += ", baz";
    s += ", qux";
    s += ", baz";
    s += ", qux";
    s += ", baz";
    s += ", qux";

    return s;
}
```

```
// ~7,000,000 operations p/s
no usages   new *
public String stringAppendBuilder() {
    StringBuilder sb = new StringBuilder();
    sb.append("foo");
    sb.append(", bar");
    sb.append(", bar");
    sb.append(", baz");
    sb.append(", qux");
    sb.append(", baz");
    sb.append(", qux");
    sb.append(", baz");
    sb.append(", qux");
    sb.append(", baz");
    sb.append(", qux");
    sb.append(", baz");
    sb.append(", qux");
    sb.append(", baz");
    sb.append(", qux");
    sb.append(", baz");
    sb.append(", qux");
    sb.append(", baz");
    sb.append(", qux");
    sb.append(", baz");
    sb.append(", qux");

    return sb.toString();
}   You, 5 minutes ago · Uncommitted changes
```

Рис. 5. Порівняння StringBuilder та String

Створення рядків має певну накладну витрату, а отже, по змозі, його потрібно в циклах уникати. Цього легко досягти за допомогою класу StringBuilder всередині циклу. З простим прикладом додавання результати продуктивності значно різняться (рис. 6) [7].

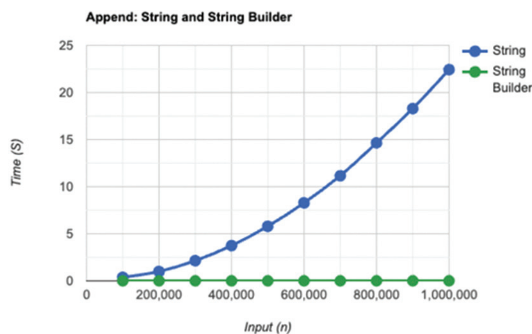

Рис. 6. Оцінювання продуктивності String та StringBuilder

Висновки

Незважаючи на те, що Java надає великий набір можливостей та API, розробники мають бути обережними з продуктивністю в процесі проєктування та реалізації свого коду. Дотримуючись порад та найкращих практик, наведених у цій статті, розробники можуть оптимізувати свій код на Java та уникнути загальних пасток, які здатні несприятливо впливати на продуктивність. Від використання стека та примітивів до уникання надмірного створення об'єктів та використання паралелізму — ці поради можуть допомогти писати більш ефективні та продуктивні застосунки на Java. Пам'ятаючи про продуктивність та дотримуючись цих рекомендацій, розробники можуть забезпечити більшу функціональність та продуктивність своїх Java-застосунків.

Список використаної літератури

1. ***Oaks S.*** *Java Performance: 2nd Edition. Addison-Wesley, O'Reilly Media, Inc. 2020. 714 p.*

2. ***Weisfeld M.*** *The Object-Oriented Thought Process: 5th Edition. Addison-Wesley Professional, 2019. 323 p.*

3. ***Пошук,*** *обробка та аналіз інформації»: навч. посіб. / К. П. Сторчак, О. М. Ткаленко, О. В. Полоневич [та ін.]. Київ: ДУТ, 2018. 127 с.*
4. ***Li Y., Jiang Z.*** *Assessing and optimizing the performance impact of the just-in-time configuration parameters — a case study on PyPy // Empirical Software Engineering. 2019. 24:4. P. 2323–2363.*
5. ***URL:*** *stackoverflow*
6. ***URL:*** *javarevisited*
7. ***URL:*** *dev.to*

K. Storchak, O. Tkalenko, Y. Burda, D. Harnyk

RESEARCH ON THE PERFORMANCE OPTIMIZATION ISSUE IN THE JAVA PROGRAMMING LANGUAGE

The Java programming language, renowned and widely used, provides developers with a powerful platform for creating programs that can scale and respond quickly to user demands. However, to fully harness Java's potential, it is necessary to delve deeper into optimization possibilities. This article explores the principles, techniques, and best practices that assist Java developers in crafting programs distinguished by speed, efficiency, and responsiveness. Additionally, it examines tools and strategies that aid in identifying problem areas, fine-tuning code, and enhancing the overall performance of Java applications.

A fundamental starting point in optimizing Java applications is profiling and analysis. Profiling tools like VisualVM, YourKit, or built-in options like jVisualVM are indispensable for identifying performance bottlenecks. By uncovering areas in need of improvement, these tools facilitate the fine-tuning of applications

Efficient memory management plays a critical role in Java optimization. Proper utilization of Java's Garbage Collection mechanisms is essential. Different GC algorithms should be explored, and heap sizes tailored to the specific needs of the application. This ensures that memory is allocated and deallocated efficiently, minimizing the risk of memory leaks. Java's support for multithreading is a powerful resource for enhancing performance.

In conclusion, Java is a powerful programming language, and optimizing Java applications is a continuous process. By following these principles, techniques, and best practices, developers can create high-performance applications that respond quickly to user demands. Regular profiling, analysis, and testing are essential to maintaining and improving the efficiency and responsiveness of Java applications. Remember that performance optimization is an ongoing journey, not a one-time task.

Keywords: Java; algorithm; optimization; web applications; programming languages.