

РЕАЛІЗАЦІЯ МОНІТОРИНГУ ВИТОКІВ ПАМ'ЯТІ В NODE.JS-ДОДАТКАХ З ВИКОРИСТАННЯМ HEAP-СНАПШОТІВ

Анотація

У сучасних веб-сервісах на базі Node.js проблема витоків пам'яті може призвести до деградації продуктивності та збоїв. Node.js використовує автоматичне збирання сміття, однак витік пам'яті виникає, коли блок пам'яті більше не використовується додатком, але не звільняється збирачем сміття. У роботі проаналізовано методи виявлення витоків пам'яті в Node.js-додатках. Особливу увагу приділено використанню heap-снэпшотів – знімків купи пам'яті процесу Node.js – для моніторингу та діагностики витоків. Розглянуто процес отримання heap-снэпшотів та інструменти їх аналізу за допомогою Chrome DevTools. Наведено результати аналізу даних снэпшотів для виявлення невивільнених об'єктів. Запропоновано практичні рекомендації щодо запобігання витокам пам'яті та підвищення надійності Node.js-додатків.

Ключові слова: Node.js, витік пам'яті, heap-снэпшот, моніторинг пам'яті, профілювання.

Abstract

Memory leaks in Node.js applications can lead to performance degradation and crashes. Node.js uses automatic garbage collection; however, a memory leak occurs when a block of memory that is no longer needed by the application is not released by the garbage collector. This paper analyzes methods for detecting memory leaks in Node.js applications, with a focus on using heap snapshots for monitoring and diagnostics. The process of obtaining heap snapshots and tools for their analysis using Chrome DevTools are examined. The snapshot data analysis is presented to identify objects that are not freed from memory. Practical recommendations are provided to prevent memory leaks and improve the reliability of Node.js applications.

Key words: Node.js, memory leak, heap snapshot, monitoring, profiling.

Вступ

Розробка високонавантажених Node.js-серверів потребує особливої уваги до управління пам'яттю. Витоки пам'яті непомітно накопичуються під час тривалої роботи процесу і здатні спричинити поступове зниження продуктивності, збільшення затримок обробки запитів та навіть аварійне завершення програми [1]. У довготривалих сервісах, що працюють без перезавантаження, своєчасне виявлення та усунення витоків пам'яті є критично важливим для забезпечення надійності. Витік пам'яті можна визначити як фрагмент пам'яті, який більше не використовується програмою, але залишається зайнятим у купі і недоступним для повторного використання системою. Подібні невивільнені блоки пам'яті з часом накопичуються і можуть досягти критичного обсягу, викликаючи збій у роботі Node.js-додатку. Основні причини витоків пам'яті в JavaScript включають: незвільнені глобальні змінні, «висячі» обробники подій, замикання, що утримують зайві посилання, та неочищені таймери (setTimeout, setInterval) [2]. Таким чином, проблема витоків пам'яті в Node.js є актуальною задачею, розв'язання якої підвищує стабільність і масштабованість серверних застосувань.

Методи виявлення витоків пам'яті

Існує кілька підходів до виявлення витоків пам'яті в Node.js-додатках. Основні методи включають:

- Моніторинг використання пам'яті: просте відстеження показників використання пам'яті (наприклад, через process.memoryUsage) упродовж часу. Безперервне зростання обсягу використовуваної пам'яті між циклами збирача сміття свідчить про потенційний витік. Графічне відображення таких метрик дозволяє виявити тенденцію неконтрольованого росту пам'яті.
- Інструменти спостереження за витоками: спеціалізовані модулі та утиліти для Node.js. Наприклад, модуль memwatch генерує подію «leak», якщо обсяг купи зростає протягом 5 послідовних циклів збирача сміття. Це надає можливість програмно виявляти витік та реагувати (логувати або

перезапустити процес) до того, як пам'ять скінчиться. Іншим корисним модулем є `heapdump`, що дозволяє зробити знімок купи (`heap-снaпшот`) у будь-який момент роботи додатка і згодом аналізувати його за допомогою сторонніх інструментів [3].

- Вбудований налагоджувач Node.js: Node.js містить інтегровані засоби профілювання пам'яті. Запуск програми з прапорцем `--inspect` відкриває можливість підключення Chrome DevTools для віддаленого налагодження [3]. Через вкладку Memory у DevTools можна робити знімки пам'яті, проводити профілювання виділень і відстежувати очищення пам'яті в реальному часі. Цей метод дозволяє аналізувати витoki безпосередньо на працюючому процесі, використовуючи знайомі інструменти розробника Chrome.

- Автоматизований профілюючий софт: рішення для моніторингу, які збирають історичні дані про використання пам'яті. Приклади – менеджер процесів PM2 або пакети на кшталт Clinic.js. Вони дозволяють відстежувати витoki у продакшні, генерувати звіти та сповіщення при досягненні порогів пам'яті. Хоча такі засоби не виявляють конкретне місце витoku, вони сигналізують про проблему, після чого розробник може застосувати детальніший аналіз (наприклад, через `heap-снaпшоти`).

Використання `heap-снaпшотів` у Node.js

Одним із найефективніших підходів до діагностики витоків пам'яті є аналіз `heap-снaпшотів` – повних знімків стану купи пам'яті. Node.js надає кілька способів отримання `heap-снaпшоту` процесу. Найбільш розповсюджений підхід – підключення Chrome DevTools до працюючого Node.js. Для цього додаток запускають із прапорцем `--inspect`, після чого у Chrome на сторінці `chrome://inspect` вибирають відповідний процес Node.js і відкривають DevTools. У вкладці Memory можна вручну зробити `heap-снaпшот` натисканням кнопки “Take snapshot” [1].

Альтернативно, знімок можна отримати програмно. Починаючи з Node.js v12.0.0 доступний спеціальний прапорець `--heapsnapshot-signal`, який дозволяє створити `снaпшот` за сигналом ОС (наприклад, SIGUSR2). В Node.js v11.13.0 з'явився метод `writeHeapSnapshot()` з модуля `v8`, який можна викликати у коді додатка для збереження стану купи у файл. Такий виклик доцільно інтегрувати, наприклад, у обробник спеціального HTTP-запиту або сигналу, аби знімок створювався без зупинки сервісу. [1] Для версій Node.js, що не підтримують ці механізми, можна скористатися зовнішнім модулем `heapdump`, який надає схожий функціонал.

Варто зазначити, що процес створення `heap-снaпшоту` є ресурсоємним [4]. Під час знімку виконання основного потоку Node.js призупиняється. До того ж, сам `снaпшот` тимчасово подвоює використання пам'яті (оскільки його формування відбувається у пам'яті), що на практиці може спричинити out of memory помилку у разі близького до граничного споживання ОЗП. Тому в продакшн-середовищі рекомендується знімати `heap-снaпшоти` на окремих інстансах або в моменти низького навантаження, щоб уникнути простою сервісу.

Аналіз даних `heap-снaпшотів`

Отриманий файл `heap-снaпшоту` містить детальну інформацію про всі об'єкти в купі, їхні типи та посилання між ними. Для аналізу `снaпшот` завантажується у Chrome DevTools (вкладка Memory, опція “Load snapshot”). DevTools дозволяє дослідити розподіл пам'яті за типами об'єктів, розмір кожного об'єкта та ланцюжки посилань (ретенери), через які об'єкти утримуються в пам'яті [4]. Ключовим підходом до виявлення витоків є порівняння декількох `снaпшотів`. Розробник робить початковий знімок `heap`, потім через певний час або після виконання підозрілих операцій – другий. Якщо між двома `снaпшотами` спостерігається істотне збільшення числа певних об'єктів, які не були звільнені, саме ці об'єкти можуть бути джерелом витoku. У Chrome DevTools для цього передбачено режим Comparison, який показує різницю між двома вибраними `снaпшотами` – зокрема, які об'єкти з'явилися або залишилися у пам'яті.

При аналізі даних `heap-снaпшоту` слід звертати увагу на об'єкти з великим retained size – тобто обсяг пам'яті, що звільниться при видаленні цього об'єкта разом із пов'язаними. Такі об'єкти часто знаходяться ближче до кореня графа об'єктів (глобальних контекстів) і можуть утримувати цілі структури в пам'яті. Виявивши підозрілий об'єкт, можна переглянути ретенерний ланцюжок (chain of retainers) – послідовність посилань від кореневого об'єкта до цього. Це допомагає зрозуміти, який саме код тримає посилання і не звільняє пам'ять. Наприклад, витік типово виникає, якщо розробник додає обробники

подій або таймери і не видаляє їх, коли вони більше не потрібні – у снапшоті такі об’єкти будуть накопичуватися. Аналізуючи снапшот, легко знайти множинні екземпляри функцій-слухачів або замкнені контексти, які мали б зникнути.

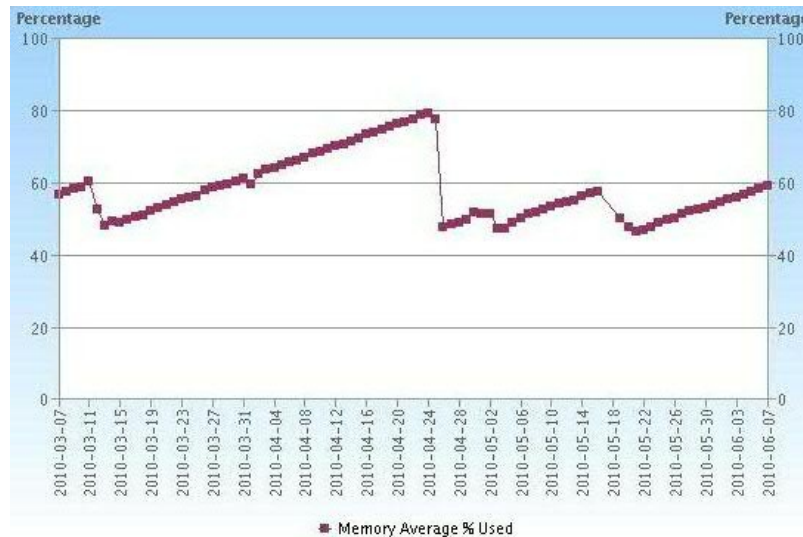


Рисунок 1. Приклад графіку використання пам'яті процесом (патерн «зубчатої пилки»)

У нормальному режимі спостерігаються періодичні спадання використання пам'яті внаслідок роботи збирача сміття (див. рис. 1). Втім, сталий тренд до збільшення рівня пам'яті, що скидається лише різким спадом (як-от після 20.04 на графіку), може свідчити про витік пам'яті. Порівнюючи hear-снапшоти до та після такого стрибка, можна визначити, які об'єкти залишилися в пам'яті – це й буде підказкою до джерела витоку. Таким чином, аналіз хронології використання пам'яті та даних снапшотів дозволяє точно локалізувати проблемні ділянки коду.

Висновок

На основі проведеного аналізу можна сформулювати рекомендації для запобігання витокам пам'яті в Node.js-додатках. По-перше, розробникам слід дотримуватись безпечних практик управління пам'яттю: видаляти непотрібні посилання на об'єкти, очищувати глобальні змінні, що більше не використовуються, своєчасно відписуватись від подій і зупиняти таймери. Це зменшує ймовірність виникнення «сміттєвих» об'єктів, недоступних для збору. По-друге, варто моніторити використання пам'яті в реальному часі на етапах тестування і у продакшні. Встановлення порогових значень споживання пам'яті та сповіщень дозволить оперативно реагувати на можливий витік до того, як він вплине на користувачів. По-третє, доцільно інтегрувати регулярне профілювання пам'яті в процес розробки: періодично знімати hear-снапшоти під час навантажувального тестування і аналізувати їх на предмет аномалій. Такий проактивний підхід допоможе виявити витоки на ранніх стадіях.

Крім цього, впровадження регулярного аналізу hear-снапшотів допомагає оптимізувати не лише використання пам'яті, а й загальну продуктивність Node.js-додатків. Зокрема, аналізуючи стан купи, можна виявити неефективні алгоритмічні рішення та зайві обчислення, які призводять до надмірного споживання ресурсів. Розуміння структури пам'яті дозволяє більш ефективно використовувати інструменти масштабування, наприклад горизонтальне масштабування серверних інстансів, що стає можливим завдяки стабільній роботі окремих екземплярів додатків. Крім того, постійний моніторинг пам'яті сприяє покращенню якості програмного коду, адже розробники починають більше уваги приділяти його оптимізації та чистоті. У результаті організація, яка дотримується цих практик, отримує переваги не лише у вигляді стабільності сервісів, але й підвищеної ефективності та знижених операційних витрат, оскільки оптимізація пам'яті дозволяє заощадити на інфраструктурних ресурсах, зокрема хмарних обчислювальних потужностях.

У висновку, застосування hear-снапшотів зарекомендувало себе як ефективний інструмент для діагностики витоків пам'яті в Node.js. Комбінація безперервного моніторингу пам'яті та детального

аналізу знімків купи дозволяє не лише знаходити “втрачений” обсяг пам’яті, але й розуміти природу витоку та місце в коді, де він виникає. Впровадження описаних підходів у практику розробки дає змогу підвищити надійність Node.js-сервісів: своєчасно виявлені та усунені витoki пам’яті запобігають збоям, забезпечують стабільну роботу серверів при тривалому навантаженні і покращують масштабованість системи. Це особливо важливо для сучасних веб-застосунків, які працюють 24/7, обслуговуючи велику кількість користувачів в реальному часі. Отже, постійний контроль за пам’яттю та використання інструментів аналізу, таких як heap-снaпшоти, є невід’ємною частиною процесу розробки продуктивних і надійних Node.js-додатків.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Node.js Documentation – Using Heap Snapshot. URL: https://nodejs-org.translate.goog/en/learn/diagnostics/memory/using-heap-snapshot?_x_tr_sl=en&_x_tr_tl=uk&_x_tr_hl=uk&_x_tr_pto=sc (дата звернення: 15.03.2025).
2. Debugging Node.js Memory Leaks: How to Detect, Solve or Avoid Them in Applications. URL: https://sematext-com.translate.goog/blog/nodejs-memory-leaks/?_x_tr_sl=en&_x_tr_tl=uk&_x_tr_hl=uk&_x_tr_pto=sc (дата звернення: 15.03.2025).
3. Debugging Memory Leaks in Node.js Applications. URL: https://www-toptal-com.translate.goog/nodejs/debugging-memory-leaks-node-js-applications?_x_tr_sl=en&_x_tr_tl=uk&_x_tr_hl=uk&_x_tr_pto=sc (дата звернення: 15.03.2025).
4. Memory Leakage Testing Using Chrome Heap Snapshot Tool. URL: https://www-pixelqa-com.translate.goog/blog/post/memory-leakage-testing-using-chrome-heap-snapshot-tool?_x_tr_sl=en&_x_tr_tl=uk&_x_tr_hl=uk&_x_tr_pto=sc (дата звернення: 15.03.2025).

Палій Олексій Миколайович — інженер програмного забезпечення; LAMPA Software, м. Вінниця, email: alexey.paliy1337@gmail.com

Oleksii Palii – Software Engineer, LAMPA Software, Vinnytsia, e-mail: alexey.paliy1337@gmail.com