Мирослав Поліщук Ігорович

Кафедра Автоматики та Управління в Технічних Системах , Національний технічний університет України "Київський політехнічний інститут імені Ігоря Сікорського ", УКРАЇНА, м.Київ, пр-т Перемоги, 37

Управління пам'яттю в JavaScript

Більшість розробників рідко замислюються про те, як реалізовано управління пам'яттю в JavaScript. Движок зазвичай робить все за програміста, так що останньому немає сенсу міркувати про принципи функціонування механізму управлінням пам'яттю. Але рано чи пізно розробникам все ж доводиться розбиратися з проблемами, пов'язаними з пам'яттю - наприклад, витоками. Ну а розібратися з ними вийде лише тоді, коли є розуміння механізму виділення пам'яті.

Життєвий цикл пам'яті

При створенні функцій, змінних в JavaScript движок виділяє певний обсяг пам'яті. Потім він звільняє її - після того, як пам'ять вже не потрібно. Власне, виділенням пам'яті можна назвати процес резервування певного обсягу пам'яті. Ну а звільнення її - це повернення резерву системі. Використовувати її можна повторно скільки завгодно раз. Коли виконується оголошення змінної або створюється функція, то пам'ять проходить наступний цикл:

Allocate - виділення пам'яті, що робить движок. Він виділяє пам'ять, яка потрібна для створеного об'єкта.

Use - використання пам'яті. За цей момент відповідає розробник, прописуючи в коді читання і запис в пам'ять.

Release - звільнення пам'яті. Тут знову настає «зона відповідальності» JavaScript. Після того, як резерв вивільнено, пам'ять можна використовувати і для інших цілей.

«Об'єкти» в контексті управління пам'яттю мають на увазі не тільки об'єкти JS, але також функції і області дії.

Стек пам'яті і купа

В цілому, все начебто зрозуміло - JavaScript виділяє пам'ять під все, що розробник задає в коді, а потім, коли всі операції виконані, пам'ять звільняється. Але де зберігаються дані? Є два варіанти - в стеку (stack) пам'яті і в купі (heap). Що перше, що друге - назва структур даних, які використовуються движком для різних цілей.

Стек (stack) - це статична виділення пам'яті. Визначення стека відомо багатьом. Це структура даних, яка використовується для зберігання статичних даних, їх розмір завжди відомий під час компіляції[2]. В JS сюди включили примітивні значення, наприклад string, number, boolean, undefined і null, а також посилання на функції і об'єкти. Движок «розуміє», що розмір даних не змінюється, тому виділяє фіксований обсяг пам'яті для кожного із значень. Процес виділення пам'яті до виконання називається статичним виділенням пам'яті (static memory allocation). Так як браузер виділяє пам'ять заздалегідь для кожного типу даних є обмеження на розмір даних, які можна туди покласти.

Купа (heap) - динамічне виділення пам'яті. Що стосується купи, то вона знайома багатьом не гірше, ніж стек. Використовується вона для зберігання об'єктів і функцій. Але на відміну від стека движок не може «знати», який обсяг пам'яті необхідний для того або іншого об'єкта, тому пам'ять виділяється в міру необхідності. І цей спосіб виділення пам'яті називається «динамічним» (dynamic memory allocation).

Посилання в JavaScript

Що стосується стека, то на нього вказують всі змінні. Якщо ж значення не є примітивним, в стеку міститься посилання на об'єкт з купи. У ній немає певного порядку, а значить, посилання на потрібну область пам'яті зберігається в стеку. У такій ситуації об'єкт в купі схожий на будівлю, а ось посилання - це його адресу. JS зберігає об'єкти і функції в купі. А ось примітивні значення і посилання - в стеку.

Прибирання сміття. Тепер саме час повернутися до життєвого циклу пам'яті, а саме - її звільнення. Движок JavaScript відповідає не тільки за виділення пам'яті, але і за звільнення. При цьому пам'ять системі повертає збирач сміття (garbage collector). І як тільки движок «бачить», що в змінної або функції вже немає необхідності, виконується звільнення пам'яті. Але тут криється ключова проблема. Справа в тому, що вирішити, потрібна певна область пам'яті чи ні, не можна. Немає настільки точних алгоритмів, які в режимі реального часу звільняють пам'ять. Правда, є просто добре працюють алгоритми, які дозволяють робити це. Вони не ідеальні, але все ж набагато краще, ніж багато інших. Нижче - розповідь про збірку сміття, яка заснована на підрахунку посилань, а також про «алгоритмі позначок».

Алгоритм позначок. Тут на допомогу приходить інший алгоритм, який називається методом mark and sweep (Позначай і вимітай). Цей алгоритм не вважає посилання, а визначає, чи можна отримати доступ до різних об'єктів за допомогою кореневого об'єкта. У браузері це window, а в Node.js - global. Якщо об'єкт недоступний, то алгоритм позначає його, після чого прибирає. Кореневі об'єкти при цьому ніколи не знищуються. Проблема циклічних посилань тут не актуальна - алгоритм дозволяє зрозуміти, що ні dad, ні son вже недоступні, тому

їх можна «вимести», а пам'ять - повернути системі. З 2012 року абсолютно всі браузери оснащуються збирачами сміття, які працюють саме за методом mark and sweep.

Не обійшлося без недоліків і тут

Можна було б подумати, що все відмінно, і тепер можна забути про управління пам'яттю, доручивши все алгоритму. Але це не зовсім так.

Використання великого обсягу пам'яті. Через те, що алгоритми не вміють визначати, коли саме пам'ять стає непотрібною, додатки на JavaScript можуть використовувати більший обсяг пам'яті, ніж потрібно. І лише збирач може вирішити, звільняти чи ні виділену пам'ять.

Краще JavaScript з керуванням пам'яттю справляються низькорівневі мови. Але і тут є свої недоліки, що потрібно мати на увазі. Зокрема, JS не дає інструментів управління пам'яттю, на відміну від низькорівневі мов, в яких програміст «вручну» займається виділенням і звільненням пам'яті[3].

Продуктивність .Пам'ять не очищується кожен новий момент часу. Звільнення виконується з певною періодичністю. Але розробники не можуть знати, коли саме запускаються ці процеси. Тому в деяких випадках збірка сміття може негативно відбиватися на продуктивності, оскільки алгоритму для роботи потрібні певні ресурси. Правда, ситуація рідко стає прямо зовсім вже некерованою. Найчастіше наслідки цього мікроскопічні.

Витоки пам'яті

У розробці витік пам'яті - одне з найбільш неприємних явищ. Але якщо знати всі найпоширеніші види витоків, то обійти проблему можна без особливих зусиль. Витоку пам'яті найчастіше трапляються через зберігання даних в глобальних змінних. Додаток використовує більший обсяг пам'яті, ніж належить і в тому випадку, якщо забути про таймерах і коллбеках[1]. Головна проблема - односторінкові додатки (SPA), а також динамічне додавання коллбеков і обробників подій. Забуті DOM елементи в змінних. При видаленні будь-якого з елементів вище варто подбати і про його видаленні з масиву. В іншому випадку збирач сміття не стане його видаляти автоматично.

Висновки

Знання загальних принципів виділення пам'яті важливі практично з самого початку кар'єри, тому що більшу популярність зараз отримали веб-додатки (в минулому їх називали SPA - Single Page Applications). Основний ключовою особливістю цих додатків є те, що вони "живуть в часі" - при переході між сторінками не відбувається повного скидання стану (технічно сторінка не перезавантажується, а змінюється на льоту), тому витоку пам'яті накопичуються, що може призводити до загальмування роботи вкладки,

браузера і комп'ютера користувача. Друга не менш важлива особливість - рендеринг даних відбувається на клієнті. Тому при великій кількості елементів на сторінці або частих змінах даних ми можемо відчувати великі просадки по продуктивності.

Ми як розробники повинні стежити за тим, як ми виділяємо пам'ять при часто повторюваних операціях (рендеринг компонентів, обхід циклів, оголошення змінних в обробниках подій). Тому що, якщо ми занадто часто будемо оголошувати змінні у нас буде постійно виділятися нова пам'ять під зберігання їх значень, додаток буде "пухнути" в пам'яті і як наслідок - буде частіше спрацьовувати garbage collector.

Через це ми будемо відчувати постійні мікрофрізи (js однопоточен, тому всі процеси заблокують на час роботи garbage collector'a). Це сильно впливає на якість користувацького досвіду, і погіршує якість додатків.

Література

[1] Збірка сміття [Електронний ресурс]. – 5. – Режим доступу до ресурсу: https://learn.javascript.ru/garbage-collection.

[2] Implementing Heaps in JavaScript [Електронний ресурс] – Режим доступу до ресурсу: https://blog.bitsrc.io/implementing-heaps-in-javascript-c3fbf1cb2e65.

[3] How JavaScript works: memory management + how to handle 4 common memory leaks [Електронний ресурс] – Режим доступу до ресурсу: https://blog.sessionstack.com/how-javascript-works-memory-management-how-to-handle-4-common-memory-leaks-3f28b94cfbec