РАЗРАБОТКА ВЕБ-САЙТОВ*, JAVASCRIPT*, БЛОГ КОМПАНИИ RUVDS.COM

Как работает JS: управление памятью, четыре вида утечек памяти и борьба с ними

ПЕРЕВОД

ru_vds 18 сентября 2017 в 13:56 30,3k

Оригинал: Alexander Zlatkov

→ Часть 1: Как работает JS: обзор движка, механизмов времени выполнения, стека вызовов
→ Часть 2: Как работает JS: о внутреннем устройстве V8 и оптимизации кода

В третьем материале из серии, которая посвящена особенностям работы JavaScript, мы поговорим о памяти. Эта тема крайне важна, однако, разработчики нередко игнорируют её. В основе этой ситуации лежат разные причины, среди которых — всё возрастающая сложность современных языков программирования и прогресс в развитии средств автоматического управления памятью. Помимо рассказа о модели памяти JS, мы поделимся с вами несколькими советами, направленными на борьбу с утечками памяти.

По словам автора статьи, в компании SessionStack используют приёмы предотвращения утечек памяти для того, чтобы не допустить неоправданно высокого потребления памяти в веб-приложениях, в которые интегрированы их разработки.

Обзор

Некоторые языки, такие как C, обладают низкоуровневыми инструментами для управления памятью, такими, как `malloc()` и `free()`. Эти базовые функции используются разработчиками при взаимодействии с операционной системой для явного выделения и освобождения памяти.

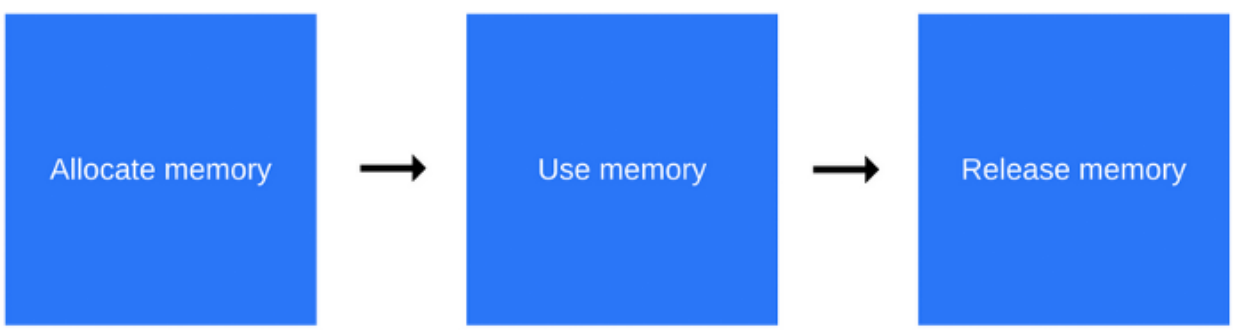
В то же время, JavaScript выделяет память, когда нечто (объекты, строки, и так далее) создаётся, и «автоматически», когда созданное больше не используется, освобождает её в ходе

процесса, называемого *сборкой мусора*. Эта вроде бы «автоматическая» природа освобождения ресурсов является источником путаницы и даёт разработчикам, использующим JavaScript (и другие высокоуровневые языки) ложное ощущение того, что они могут совершенно не заботиться об управлении памятью. **Это — большая ошибка**.

Даже программируя на высокоуровневом языке, разработчики должны понимать принципы, или по крайней мере владеть основами управления памятью. Иногда в системе автоматического управления памятью возникают проблемы (вроде ошибок, ограничений в реализации сборщика мусора и так далее), природу которых разработчики должны понимать для того, чтобы правильно их устранять (или хотя бы находить верные способы их обхода, требующие минимальных дополнительных усилий и не слишком больших объёмов вспомогательного кода).

Жизненный цикл памяти

Вне зависимости от языка программирования, жизненный цикл памяти практически всегда выглядит одинаково:



Жизненный цикл памяти: выделение, использование, освобождение

- **Выделение памяти** — память выделяется операционной системой, что позволяет программе использовать предоставленные в её распоряжение ресурсы. В низкоуровневых языках (таких, как C), это явная операция, которую необходимо производить разработчику. В высокоуровневых языках, однако, эта задача решается автоматически.
- **Использование памяти** — это то время, когда программа выполняет какие-либо операции с выделенной ранее памятью. На этом этапе, при обращении к переменным, производятся операции чтения и записи.
- **Освобождение памяти** — на данном этапе жизненного цикла памяти производится освобождение памяти, которая больше не нужна программе, то есть — возврат её системе. Как и в случае с **выделением памяти**, освобождение — явная операция в низкоуровневых языках.

Напомним, что в первом материале этого цикла можно почитать о стеке вызовов и о куче.

Что такое память?

Прежде чем рассматривать вопросы работы с памятью в JavaScript, поговорим, в двух словах, о том, что такое память.

На аппаратном уровне компьютерная память состоит из

множества триггеров. Каждый триггер состоит из нескольких транзисторов, он способен хранить один бит данных. Каждый из триггеров имеет уникальный адрес, поэтому их содержимое можно считывать и перезаписывать. Таким образом, концептуально, мы можем воспринимать компьютерную память как огромный массив битов, которые можно считывать и записывать.

Однако, программисты — люди, а не компьютеры, оперировать отдельными битами им не особенно удобно. Поэтому биты принято организовывать в более крупные структуры, которые можно представлять в виде чисел. 8 бит формируют 1 байт. Помимо байтов здесь в ходу такое понятие, как слова (иногда — длиной 16 битов, иногда — 32).

В памяти хранится много всего:

1. Все значения переменных и другие данные, используемые программами.
2. Код программ, в том числе — код операционной системы.

Компилятор и операционная система совместно выполняют основной объём работ по управлению памятью без непосредственного участия программиста, однако, мы рекомендуем разработчикам поинтересоваться тем, что происходит в недрах механизмов управления памятью.

Когда код компилируют, компилятор может исследовать примитивные типы данных и заранее вычислить необходимый для работы с ними объём памяти. Требуемый объём памяти затем

выделяется программе в пространстве стека вызовов. Пространство, в котором выделяется место под переменные, называется стековым пространством, так как, когда вызываются функции, выделенная им память размещается в верхней части стека. При возврате из функций, они удаляются из стека в порядке LIFO (последним пришёл — первым вышел, Last In First Out). Например, рассмотрим следующие объявления переменных:

```
int n; // 4 байта
int x[4]; // массив из 4-х элементов по 4 байта каждый
double m; // 8 байтов
```

Компилятор, просмотрев данный фрагмент кода (абстрагируемся тут от всего, кроме размеров самих данных), может немедленно выяснить, что для хранения переменных понадобится 4 + 4 × 4 + 8 = 28 байт.

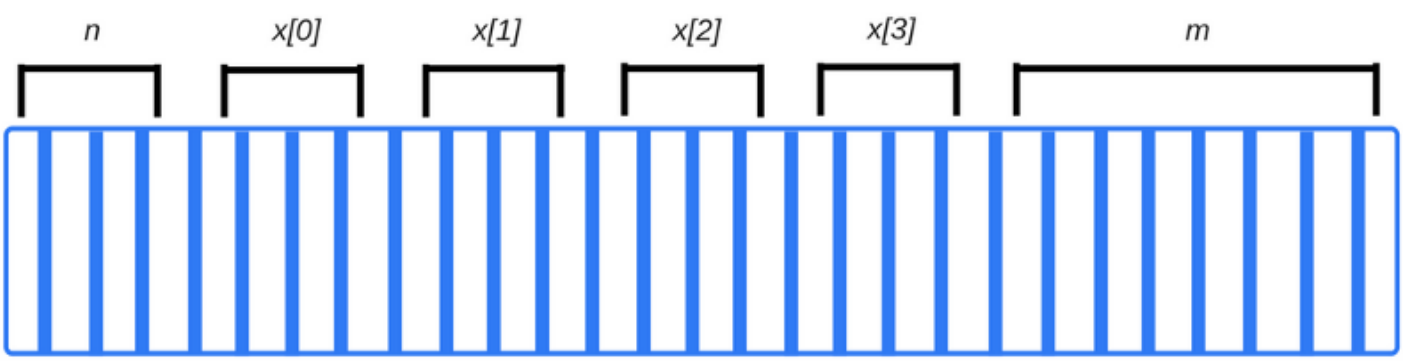
Надо отметить, что приведённые размеры целочисленных переменных и чисел с двойной точностью отражают современное состояние дел. Примерно 20 лет назад целые числа обычно представляли в виде 2-х байтовых конструкций, для чисел двойной точности использовали 4 байта. Код не должен зависеть от байтовых размеров базовых типов данных.

Компилятор сгенерирует код, который будет взаимодействовать с операционной системой, запрашивая необходимое число байтов в стеке для хранения переменных.

В вышеприведённом примере компилятору известны адреса

участков памяти, где хранится каждая переменная. На самом деле, если мы используем в коде имя переменной `n`, оно преобразуется во внутреннее представление, которое выглядит примерно так: «адрес памяти 4127963».

Обратите внимание на то, что если мы попытаемся обратиться к элементу массива из нашего примера, использовав конструкцию `x[4]`, мы, на самом деле, обратимся к данным, которые соответствуют переменной `m`. Так происходит из-за того, что элемента массива с индексом 4 не существует, запись вида `x[4]` укажет на область памяти, которая на 4 байта дальше, чем тот участок памяти, который выделен для последнего из элементов массива — `x[3]`. Попытка обращения к `x[4]` может закончится чтением (или перезаписью) некоторых битов переменной `m`. Подобное, практически гарантированно, приведёт к нежелательным последствиям в ходе выполнения программы.



Расположение переменных в памяти

Когда функция вызывает другую функцию, каждой них достаётся собственный участок стека. Здесь хранятся все их локальные переменные и указатель команд, который хранит данные о том,

где был выполнен вызов. Когда функция завершает работу, блоки памяти, занятые ей, снова делаются доступными для других целей.

Динамическое выделение памяти

К сожалению, всё усложняется, когда мы, во время компиляции, не знаем, сколько памяти понадобится для переменной. Представьте себе, что мы хотим сделать примерно следующее:

```
int n = readInput(); // прочесть данные, введённые пользователем
...
// создать массив с n элементами
```

В подобной ситуации компилятор не знает, сколько памяти понадобится для хранения массива, так как размер массива определяет значение, которое введёт пользователь.

В результате компилятор не сможет зарезервировать память для переменной в стеке. Вместо этого нашей программе придётся явно запросить у операционной системы нужное количество памяти во время её выполнения. Эта память выделяется в так называемой **куче**. В следующей таблице приведены основные различия между статическим и динамическим выделением памяти.

Разница между статическим и динамическим выделением памяти

Статическое выделение памяти	Динамическое выделение памяти
Объём должен быть известен во время компиляции.	Объём может быть неизвестен во время компиляции.
Производится во время компиляции программы.	Производится во время

	выполнения программы.
Память выделяется в стеке.	Память выделяется в куче
Порядок выделения памяти FILO (первым вошёл — последним вышел, First In Last Out)	Определённого порядка выделения памяти нет.

Для того, чтобы полностью понять, как работает динамическое выделение памяти, нам понадобится подробно обсудить концепцию **указателей**, но так мы слишком сильно отклонимся от нашей основной темы. Если вам это интересно — дайте знать автору этого материала, и, возможно, эта тема будет поднята в дальнейших публикациях.

Выделение памяти в JavaScript

Сейчас мы поговорим о том, как первый шаг жизненного цикла памяти (выделение) реализуется в JavaScript.

JavaScript освобождает разработчика от ответственности за управление выделением памяти. JS делает это самостоятельно, вместе с объявлением переменных.

```
var n = 374; // выделение памяти для числа
var s = 'sessionstack'; // выделение памяти для строки
var o = {
  a: 1,
  b: null
}; // выделение памяти для объекта и содержащихся в нём значений
var a = [1, null, 'str'];  // выделение памяти для массива
                  // и содержащихся в нём значений (похоже на работу
с объектом)
function f(a) {
  return a + 3;
} // выделение памяти для функции (она является вызываемым объектом)
// объявление функционального выражения также приводит к выделению
памяти под объект
someElement.addEventListener('click', function() {
```

```
  someElement.style.backgroundColor = 'blue';
}, false);
```

Вызовы некоторых функций также приводят к выделению памяти под объект:

```
var d = new Date(); // выделение памяти под объект типа Date
var e = document.createElement('div'); // выделение памяти для
элемента DOM
```

Вызовы методов тоже могут приводить к выделению памяти под новые значения или объекты:

```
var s1 = 'sessionstack';
var s2 = s1.substr(0, 3); // s2 - это новая строка
// Так как строки неизменяемы,
// JavaScript может решить не выделять память,
// а просто сохранить диапазон [0, 3].
var a1 = ['str1', 'str2'];
var a2 = ['str3', 'str4'];
var a3 = a1.concat(a2);
// новый массив с 4 элементами является результатом
// конкатенации элементов a1 и a2
```

Использование памяти в JavaScript

Использование выделенной памяти в JavaScript, как правило, означает её чтение и запись.

Это может быть сделано путём чтения или записи значения переменной или свойства объекта, или даже при передаче аргумента функции.

Освобождение памяти, которая больше не нужна

Большинство проблем с управлением памятью возникает на этой стадии.

Самой сложной задачей является выяснение того, когда выделенная память больше не нужна программе. Для этого часто требуется, чтобы разработчик определил, где в программе некий фрагмент памяти больше не нужен и освободил бы его.

Высокоуровневые языки имеют встроенную подсистему, которая называется **сборщиком мусора**. Роль этой подсистемы заключается в отслеживании операций выделения памяти и использование её для того, чтобы узнать, когда фрагмент выделенной памяти больше не нужен. Если это так, сборщик мусора может автоматически освободить этот фрагмент.

К сожалению, этот процесс не отличается абсолютной точностью, так как общая проблема выяснения того, нужен или нет некий фрагмент памяти, неразрешима (не поддаётся алгоритмическому решению).

Большинство сборщиков мусора работают, собирая память, к которой нельзя обратится, то есть такую, все переменные, указывающую на которую, недоступны. Это, однако, слишком смелое предположение о возможности освобождения памяти, так как в любое время некая область памяти может иметь переменные, указывающие на неё в некоей области видимости, хотя с этой областью памяти никогда уже не будут работать в

программе.

Сборка мусора

Основная концепция, на которую полагаются алгоритмы сборки мусора — это концепция **ссылок**.

В контексте управления памятью, объект ссылается на другой объект, если первый, явно или неявно, имеет доступ к последнему. Например, объект JavaScript имеет ссылку на собственный прототип (**явная ссылка**) и на значения свойств прототипа (**неявная ссылка**).

Здесь идея «объекта» расширяется до чего-то большего, нежели обычный JS-объект, сюда включаются, кроме того, функциональные области видимости (или глобальную лексическую область видимости).

Принцип лексической области видимости позволяет задать правила разрешения имён переменных во вложенных функциях. А именно, вложенные функции содержат область видимости родительской функции даже если осуществлён возврат из родительской функции.

Сборка мусора, основанная на подсчёте ссылок

Это — самый простой алгоритм сборки мусора. Объект считается пригодным для уничтожения, если на него не указывает **ни одна** ссылка.

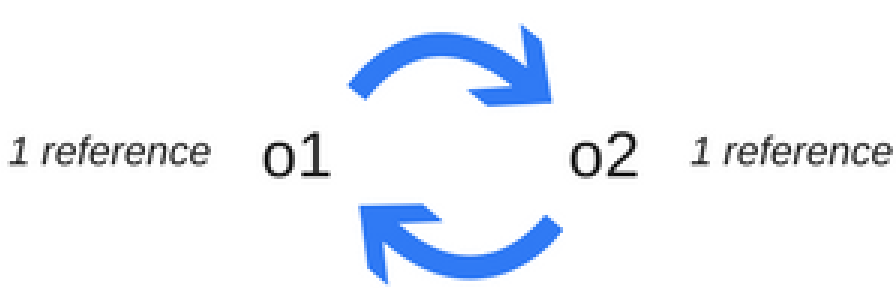
Взгляните на следующий код:

```
var o1 = {
  o2: {
    x: 1
  }
};
// Созданы 2 объекта.
// На объект o2 есть ссылка в объекте o1, как на одно из его
свойств.
// На данном этапе ни один объект не может быть уничтожен сборщиком
мусора.

var o3 = o1; // Переменная o3 - это вторая сущность, которая
          // имеет ссылку на объект, на который указывает переменная
o1.

o1 = 1;      // Теперь на объект, на который изначально ссылалась
переменная o1, есть лишь
            // одна ссылка, представленная переменной o3

var o4 = o3.o2; // Ссылка на свойство o2 объекта.
                // Теперь на этот объект есть 2 ссылки. Одна - как
на свойство
                // другого объекта.
                // Вторая - в виде переменной o4

o3 = '374'; // Теперь на объект, на который изначально ссылалась
переменная o1,
            // нет ни одной ссылки.
            // Он может быть уничтожен сборщиком мусора.
            // Однако, на его свойство o2 всё ещё ссылается
            // переменная o4. В результате память, занимаемая этим
объектом,
            // не может быть освобождена.

o4 = null; // На свойство o2 объекта, изначально записанного в
переменную o1,
           // теперь нет ссылок, значит
           // объект может быть уничтожен сборщиком мусора.
```

Циклические ссылки — источник проблем

Когда в дело вступают циклические ссылки, проявляются некоторые ограничения вышеописанной модели поиска ненужных объектов. В следующем примере создаются два объекта, ссылающиеся друг на друга. Образуется циклическая ссылка. Объекты окажутся вне досягаемости после вызова функции, то есть они бесполезны, и память, которую они занимают, могла бы быть освобождена. Однако, алгоритм подсчёта ссылок считает, что так как на каждый из двух объектов есть хотя бы одна ссылка, ни один из них нельзя уничтожить.

```
function f() {
  var o1 = {};
  var o2 = {};
  o1.p = o2; // o1 ссылается на o2
  o2.p = o1; // o2 ссылается на o1. Получается циклическая ссылка.
}

f();
```

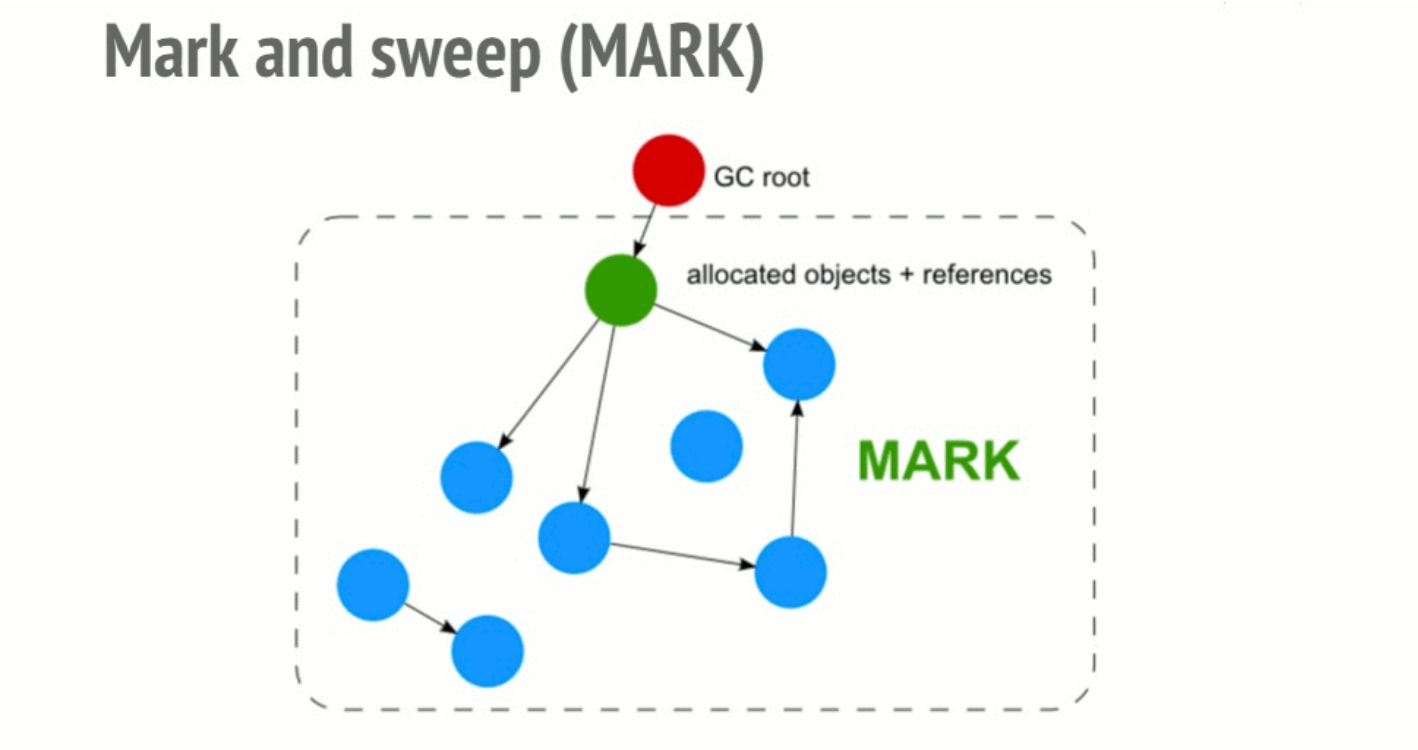


Циклическая ссылка

Алгоритм «пометь и выброси»

Для того, чтобы принять решение о том, нужно ли сохранить некий объект, алгоритм «пометь и выброси» (mark and sweep) определяет досягаемость объекта.
Алгоритм состоит из следующих шагов:

- Сборщик мусора строит список «корневых объектов». Такие объекты обычно являются глобальными переменными, ссылки на которые имеются в коде. В JavaScript примером глобальной переменной, которая может играть роль корневого объекта, является объект `window`.
- Все корневые объекты просматриваются и помечаются как активные (то есть, это не «мусор»). Также, рекурсивно, просматриваются все дочерние объекты. Всё, доступ к чему можно получить из корневых объектов, «мусором» не считается.
- Все участки памяти, не помеченные как активные, могут быть признаны подходящими для обработки сборщиком мусора, который теперь может освободить эту память и вернуть её операционной системе.



Визуализация алгоритма «пометь и выброси»

Этот алгоритм лучше предыдущего, так как ситуация «на объект нет ссылок» ведёт к тому, что объект оказывается недостижимым. Обратное утверждение, как было продемонстрировано в разделе о циклических ссылках, не верно.

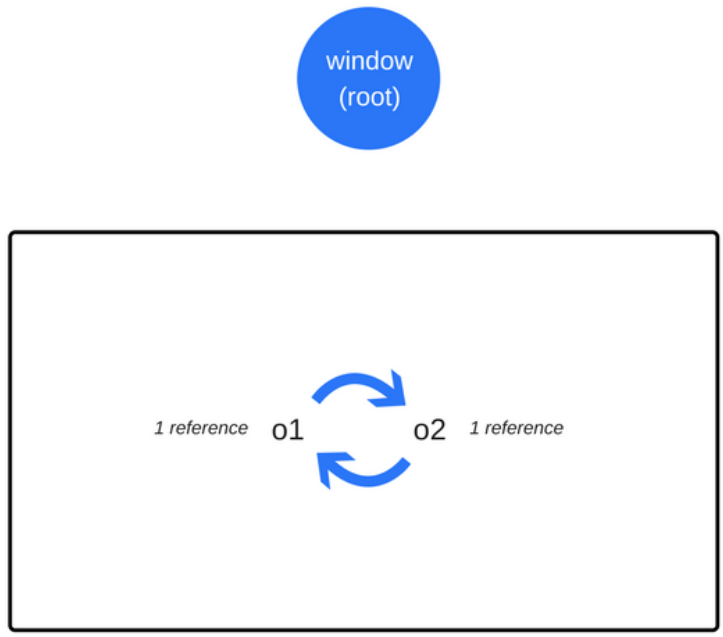
С 2012-го года все современные браузеры оснащают сборщиками мусора, в основу которых положен алгоритм «пометь и выброси». За последние годы все усовершенствования, сделанные в сфере сборки мусора в JavaScript (это — генеалогическая, инкрементальная, конкурентная, параллельная сборка мусора), являются усовершенствованиями данного алгоритма, не меняя его основных принципов, которые заключаются в определении достижимости объекта.

В этом материале вы можете найти подробности о

рассматриваемой здесь модели сборки мусора.

Решение проблемы циклических ссылок

В первом из приведённых выше примеров, после возврата из вызванной функции на два объекта больше не ссылается что-то, к чему можно обратиться из области видимости глобального объекта. Следовательно, сборщик мусора сочтёт их недостижимыми.



Циклические ссылки не мешают сборке мусора

Несмотря на то, что объекты ссылаются друг на друга, к ним нельзя получить доступ из корневого объекта.

Парадоксальное поведение сборщиков мусора

Хотя сборщики мусора удобны, при их использовании приходится

идти на определённые компромиссы. Один из них — недетерминированность. Другими словами, сборщики мусора непредсказуемы. Нельзя точно сказать, когда будет выполнена сборка мусора. Это означает, что в некоторых случаях программы используют больше памяти, чем им на самом деле нужно.

В других случаях короткие паузы, вызванные сборкой мусора, могут оказаться заметными в требовательных к производительности приложениях. Хотя непредсказуемость означает, что нельзя точно знать, когда будет произведена сборка мусора, большинство сборщиков мусора используют один и тот же шаблон выполнения операций освобождения памяти. А именно, делают они это при выделении памяти. Если память не выделяется, большинство сборщиков мусора не предпринимают активных действий. Рассмотрим следующий сценарий:

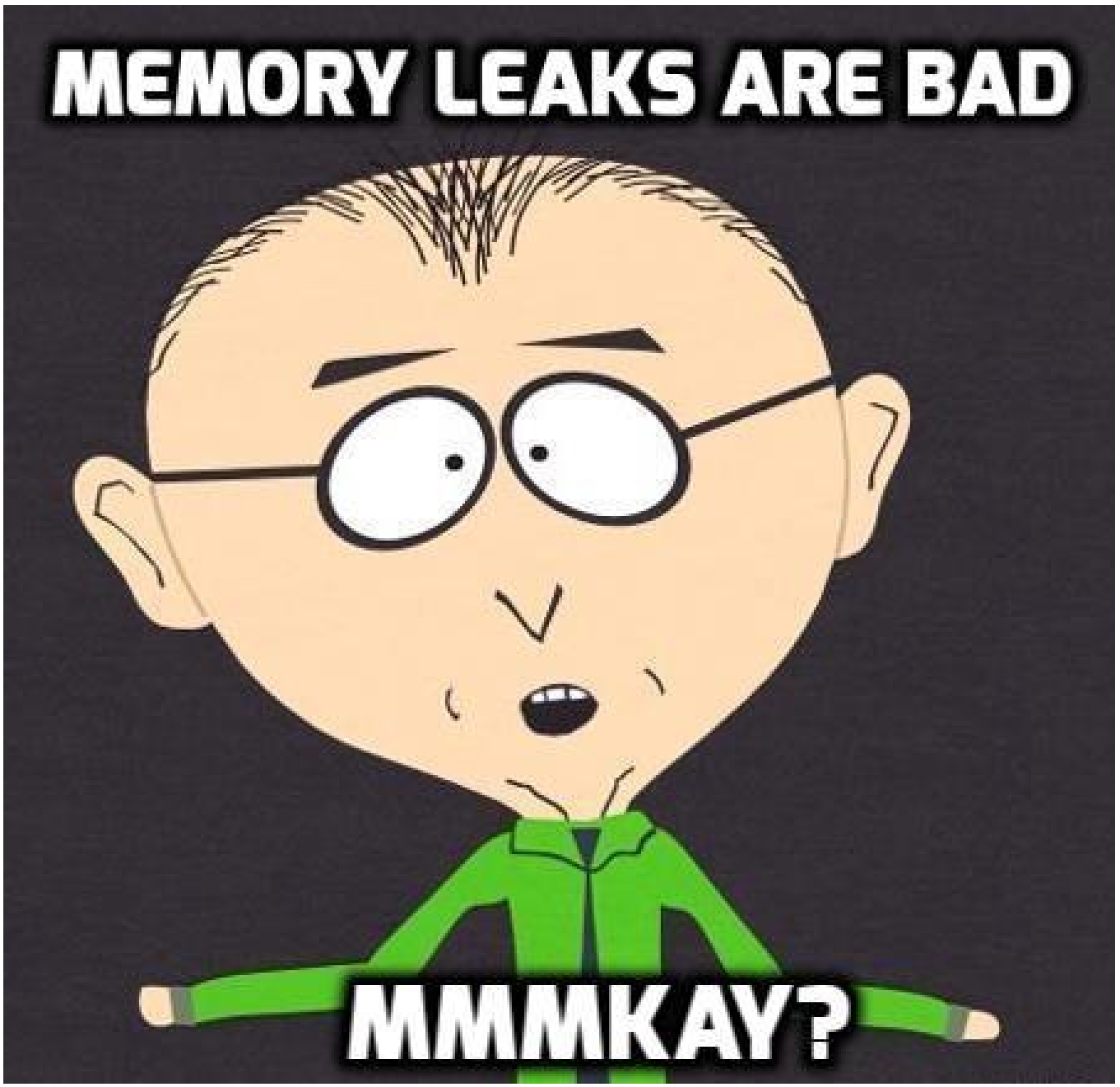
1. Было произведено несколько операций, в результате которых выделен значительный объём памяти.
2. Большинство элементов, для которых выделялась память (или все они) были помечены как недостижимые. Скажем, это может быть что-то вроде записи `null` в переменную, которая ранее ссылалась на кэш, который больше не нужен.
3. Больше память не выделялась.

При таком сценарии большинство сборщиков мусора не будет выполнять операции по освобождению памяти. Другими словами, даже хотя участки памяти могут быть освобождены, сборщик мусора их освобождать не будет. Такую ситуацию ещё нельзя назвать утечкой памяти, но она приводит к тому, что программа

использует больше памяти, чем ей нужно для обычной работы.

Что такое утечки памяти?

В двух словах, утечки памяти можно определить как фрагменты памяти, которые больше не нужны приложению, но по какой-то причине не возвращённые операционной системе или в пул свободной памяти.



Языки программирования используют разные способы управления памятью. Однако, проблема точного определения того, используется ли на самом деле некий участок памяти или нет, как уже было сказано, неразрешима. Другими словами, только разработчик знает, можно или нет вернуть операционной системе некую область памяти.

Определённые языки программирования предоставляют разработчику вспомогательные средства для управления памятью. Другие языки ожидают от программиста явных указаний касательно используемых и неиспользуемых участков памяти. Подробнее об этом можно почитать в материалах о ручном и автоматическом управлении памятью.

Рассмотрим четыре распространённых типа утечек памяти в JavaScript.

Утечки памяти в JavaScript и борьба с ними

Глобальные переменные

В JavaScript используется интересный подход к работе с необъявленными переменными. Обращение к такой переменной создаёт новую переменную в глобальном объекте. В случае с браузерами, глобальным объектом является `window`. Рассмотрим такую конструкцию:

```
function foo(arg) {
    bar = "some text";
```

```
}
```

Она эквивалентна следующему коду:

```
function foo(arg) {
    window.bar = "some text";
}
```

Если переменную `bar` планируется использовать только внутри области видимости функции `foo`, и при её объявлении забыли о ключевом слове `var`, будет случайно создана глобальная переменная.

В этом примере утечка памяти, выделенной под простую строку, большого вреда не принесёт, но всё может быть гораздо хуже.

Другая ситуация, в которой может появиться случайно созданная глобальная переменная, может возникнуть при неправильной работе с ключевым словом `this`:

```
function foo() {
    this.var1 = "potential accidental global";
}
// Функция вызывается сама по себе, при этом this указывает на глобальный объект (window),
// this не равно undefined, или, как при вызове конструктора, не указывает на новый объект
foo();
```

Для того, чтобы избежать подобных ошибок, можно добавить оператор `"use strict";` в начало JS-файла. Это включит так

называемый строгий режим, в котором запрещено создание глобальных переменных вышеописанными способами. Подробнее о строгом режиме можно почитать здесь.

Даже если говорить о вполне безобидных глобальных переменных, созданных осознанно, во многих программах их слишком много. Они, по определению, не подвергаются сборке мусора (если только в такую переменную не записать `null` или какое-то другое значение). В частности, стоит обратить пристальное внимание на глобальные переменные, которые используются для временного хранения и обработки больших объёмов данных. Если вы вынуждены использовать глобальную переменную для хранения большого объёма данных, не забудьте записать в неё `null` или что-то другое, нужное для дальнейшей работы, после того, как она сыграет свою роль в обработке большого объёма данных.

Таймеры или забытые коллбэки

В JS-программах использование функции `setInterval` — обычное явление.

Большинство библиотек, которые дают возможность работать с обозревателями и другими механизмами, принимающими коллбэки, заботятся о том, чтобы сделать недоступными ссылки на эти коллбэки после того, как экземпляры объектов, которым они переданы, становятся недоступными. Однако, в случае с `setInterval` весьма распространён следующий шаблон:

```
var serverData = loadData();
setInterval(function() {
    var renderer = document.getElementById('renderer');
    if(renderer) {
        renderer.innerHTML = JSON.stringify(serverData);
    }
}, 5000); //Это будет вызываться примерно каждые 5 секунд.
```

В этом примере показано, что может происходить с таймерами, которые создают ссылки на узлы DOM или на данные, которые в определённый момент больше не нужны.

Объект, представленный переменной `renderer`, может быть, в будущем, удалён, что сделает весь блок кода внутри обработчика события срабатывания таймера ненужным. Однако, обработчик нельзя уничтожить, освободив занимаемую им память, так как таймер всё ещё активен. Таймер, для очистки памяти, надо остановить. Если сам таймер не может быть подвергнут операции сборки мусора, это будет касаться и зависимых от него объектов. Это означает, что память, занятую переменной `serverData`, которая, надо полагать, хранит немалый объём данных, так же нельзя очистить.

В случае с обозревателями, важно использовать явные команды для их удаления после того, как они больше не нужны (или после того, как окажутся недоступными связанные объекты).

Раньше это было особенно важно, так как определённые браузеры (старый добрый IE6, например) были неспособны нормально обрабатывать циклические ссылки. В наши дни большинство браузеров уничтожают обработчики обозревателей после того, как

объекты обозревателей оказываются недоступными, даже если прослушиватели событий не были явным образом удалены. Однако, рекомендуется явно удалять эти обозреватели до уничтожения объекта. Например:

```
var element = document.getElementById('launch-button');
var counter = 0;
function onClick(event) {
   counter++;
   element.innerHtml = 'text ' + counter;
}
element.addEventListener('click', onClick);
// Сделать что-нибудь
element.removeEventListener('click', onClick);
element.parentNode.removeChild(element);
// Теперь, когда элемент выходит за пределы области видимости,
// память, занятая обоими элементами и обработчиком onClick будет
освобождена даже в старых браузерах,
// которые не способны нормально обрабатывать ситуации с
циклическими ссылками.
```

В наши дни браузеры (в том числе Internet Explorer и Microsoft Edge) используют современные алгоритмы сборки мусора, которые выявляют циклические ссылки и работают с соответствующими объектами правильно. Другими словами, сейчас нет острой необходимости в использовании метода `removeEventListener` перед тем, как узел будет сделан недоступным.

Фреймворки и библиотеки, такие, как jQuery, удаляют прослушиватели перед уничтожением узлов (при использовании для выполнения этой операции собственных API). Всё это поддерживается внутренними механизмами библиотек, которые, кроме того, контролируют отсутствие утечек памяти даже если код

работает в не самых благополучных браузерах, таких как уже упомянутый выше IE 6.

Замыкания

Одна из важных и широко используемых возможностей JavaScript — замыкания. Это — внутренняя функция, у которой есть доступ к переменным, объявленным во внешней по отношению к ней функции. Особенности реализации среды выполнения JavaScript делают возможной утечку памяти в следующем сценарии:

```
var theThing = null;
var replaceThing = function () {
  var originalThing = theThing;
  var unused = function () {
    if (originalThing) // ссылка на originalThing
      console.log("hi");
  };
  theThing = {
    longStr: new Array(1000000).join('*'),
    someMethod: function () {
      console.log("message");
    }
  };
};
setInterval(replaceThing, 1000);
```

Самое важное в этом фрагменте кода то, что каждый раз при вызове `replaceThing`, в `theThing` записывается ссылка на новый объект, который содержит большой массив и новое замыкание (`someMethod`). В то же время, переменная `unused` хранит замыкание, которое имеет ссылку на `originalThing` (она ссылается на то, на что ссылалась переменная `theThing` из предыдущего вызова `replaceThing`). Во всём этом уже можно

запутаться, не так ли? Самое важное тут то, что **когда создаётся область видимости для замыканий, которые находятся в одной и той же родительской области видимости, эта область видимости используется ими совместно**.

В данном случае в области видимости, созданной для замыкания `someMethod`, имеется также и переменная `unused`. Эта переменная ссылается на `originalThing`. Несмотря на то, что `unused` не используется, `someMethod` может быть вызван через `theThing` за пределами области видимости `replaceThing` (то есть — из глобальной области видимости). И, так как `someMethod` и `unused` находятся в одной и той же области видимости, ссылка на `originalThing`, записанная в `unused`, приводит к тому, что эта переменная оказывается активной (это — общая для двух замыканий область видимости). Это не даёт нормально работать сборщику мусора.

Если вышеприведённый фрагмент кода некоторое время поработает, можно заметить постоянное увеличение потребления им памяти. При запуске сборщика мусора память не освобождается. В целом оказывается, что создаётся связанный список замыканий (корень которого представлен переменной `theThing`), и каждая из областей видимости этих замыканий имеет непрямую ссылку на большой массив, что приводит к значительной утечке памяти.

Эту проблему обнаружила команда Meteor, у них есть отличная

статья, в которой всё это подробно описано.

Ссылки на объекты DOM за пределами дерева DOM

Иногда может оказаться полезным хранить ссылки на узлы DOM в неких структурах данных. Например, предположим, что нужно быстро обновить содержимое нескольких строк в таблице. В подобной ситуации имеет смысл сохранить ссылки на эти строки в словаре или в массиве. В подобных ситуациях система хранит две ссылки на элемент DOM: одну из них в дереве DOM, вторую — в словаре. Если настанет время, когда разработчик решит удалить эти строки, нужно позаботиться об обеих ссылках.

```
var elements = {
    button: document.getElementById('button'),
    image: document.getElementById('image')
};
function doStuff() {
    image.src = 'http://example.com/image_name.png';
}
function removeImage() {
    // Изображение является прямым потомком элемента body.
    document.body.removeChild(document.getElementById('image'));
    // В данный момент у нас есть ссылка на #button в
    // глобальном объекте elements. Другими словами, элемент button
    // всё ещё хранится в памяти, она не может быть очищена
сборщиком мусора.
}
```

Есть ещё одно соображение, которое нужно принимать во внимание при создании ссылок на внутренние элементы дерева DOM или на его концевые вершины.

Предположим, мы храним ссылку на конкретную ячейку таблицы

(тег <td>) в JS-коде. Через некоторое время решено убрать таблицу из DOM, но сохранить ссылку на эту ячейку. Чисто интуитивно можно предположить, что сборщик мусора освободит всю память, выделенную под таблицу, за исключением памяти, выделенной под ячейку, на которую у нас есть ссылка В реальности же всё не так. Ячейка является узлом-потомком таблицы. Потомки хранят ссылки на родительские объекты. Таким образом, наличие ссылки на ячейку таблицы в коде приводит к тому, что в памяти остаётся вся таблица. Учитывайте эту особенность, храня ссылки на элементы DOM в программах.

Итоги

Мы поговорили об управлении памятью, в частности, особое внимание уделили процессу её освобождения, так называемой сборке мусора. Именно на данном этапе жизненного цикла памяти возникают проблемы, которые выражаются в том, что память, занятая ненужными объектами, не может быть очищена. Как бы хорошо в современных реализациях JS-движков ни работали алгоритмы сборки мусора, программисту стоит помнить о том, что и на нём лежит определённая доля ответственности за рациональное использование памяти.

Уважаемые читатели! Сталкивались ли вы с утечками памяти в JavaScript-программах? Если да — расскажите пожалуйста, как это было, и как вы с этим справились.

Поделиться:

Сохранить:

Комментарии (10)

Похожие публикации

Может ли в JavaScript конструкция (a==1 && a==2 && a==3) оказаться равной true?

95

ПЕРЕВОД

ru_vds • 25 января в 16:08

JavaScript: путь к ясности кода

11

ПЕРЕВОД

ru_vds • 15 ноября 2017 в 13:01

ArrayBuffer и SharedArrayBuffer в JavaScript, часть 1: краткий курс по управлению памятью

12

ПЕРЕВОД

ru_vds • 21 июня 2017 в 16:24

Популярное за сутки

Наташа — библиотека для извлечения структурированной информации из текстов на русском языке

14

alexkuku • вчера в 16:12

Unit-тестирование скриншотами: преодолеваем звуковой барьер. Расшифровка доклада

4

lahmatiy • вчера в 13:05

Люди не хотят чего-то действительно нового — они хотят привычное, но сделанное иначе

25

ПЕРЕВОД

Smileek • вчера в 10:32

Руководство по SEO JavaScript-сайтов. Часть 2. Проблемы, эксперименты и рекомендации

2

ПЕРЕВОД

ru_vds • вчера в 12:04

Как адаптировать игру на Unity под iPhone X к апрелю

0

P1CACHU • вчера в 16:13

Лучшее на Geektimes

Стивен Хокинг, автор «Краткой истории времени», умер на 77 году жизни

33

HostingManager • вчера в 13:49

Обзор рынка моноколес 2018

70

lozga • вчера в 06:58

«Битва за Telegram»: 35 пользователей подали в суд на ФСБ

40

alizar • вчера в 15:14

Стивен Хокинг и его работа — что дал ученый человечеству?

8

marks • вчера в 14:46

Sunlike — светодиодный свет нового поколения

17

AlexeyNadezhin • вчера в 20:32