Ilyushin E.[1], Namiot D.[2]

[1] Lomonosov Moscow State University Ломоносова, Moscow, RKT program, john.ilyushin@gmail.com

[2] Lomonosov Moscow State University, Moscow, senior scientist, dnamiot@gmail.com

JAVASCRIPT MEMORY MANAGEMENT

KEYWORDS

Javascript, memory management, memory leaks.

ABSTRACT

In this paper, we describe the memory management issues in JavaScript applications. Nowadays, JavaScript has become a mainstream programming environment. Modern applications in JavaScript are complex software systems. We can mention here web portals, online games, Internet of Things (Web of Things) applications, and even data mining code. Of course, JavaScript memory management becomes a critical aspect of the development (and deployment) process. In this paper, we discuss the memory leaks patterns in JavaScript code as well as the basic issues behind garbage collection in JavaScript engines.

Introduction

Originally, JavaScript uses garbage collection for automatic memory management. But in the same time, for example, the old conception of web page life cycle (full page refresh) is no more in use. So, in the web programming, we have to deal with long-lived components and the increased complexity. So, JavaScript memory management becomes an even more critical aspect of the development (and deployment) process. The developers will need to understand and deliberately manage the individual lifecycles and memory footprints of the components in their applications. There are different components affecting JavaScript memory distribution. We can mention here DOM Elements (one of the main sources for memory management difficulties), JavaScript Objects as well client-side cache.

Usually, there are two main sources for memory-related issues in JavaScript [2]:

- Orphaned objects;
- Circular references.

Both ways are easily achievable due to “simplicity” of the language. So, it is very important for the developers to understand the background of possible memory-related issues, the associated measurements as well as the solutions which help to avoid them.

Usually, in JavaScript applications, developers do not carry about memory management. Objects could be easily created and reused, where JavaScript engine (its garbage collector) takes care about low-level details. The central concept of JavaScript memory management is a concept of reachability [3]. A distinguished set of objects are assumed to be reachable: these are known as the roots. Typically, these include all the objects referenced from anywhere in the call stack (that is, all local variables and parameters in the functions currently being invoked), and any global variables. Objects are kept in memory while they are accessible from roots through a reference or a chain of references [4]. And there is a Garbage Collector (GC) in the JavaScript engine (in the browser), which cleans memory occupied by unreachable objects [5].

Let us see the following classical example with JavaScript closures [6]. The closure makes all variables of outer functions persist while the inner function is alive. So, suppose our application creates a function and one of its variables contains a large string [4]. While the function inner stays in memory, then the variable data will hang in memory until the inner function is alive. JavaScript engine could have no idea which variables may be required by the inner function, so it keeps everything.

The next classical example is saving JavaScript data in Document Object Model (DOM) [7].

Memory leaks in JavaScript

In this section, we would like to discuss memory leaks patterns in JavaScript. Let us see the details for the above mentioned closure example.

```
function f() {
 var data="Some Large Piece of data . . . ";
 /* do something using data */
 function inner() {
```

```
  // ...
 }
 return inner;
}
```

Here the life time for function inner is unknown, so, we have to keep in memory the variable data too.

The Circular references present another classical example of memory leaks.

```
var obj;
function circular_references() {
 obj=document.getElementById("bigdata");
 document.getElementById("bigdata").expandoProperty = obj;
 obj.bigString=new Array(1000).join(new Array(2000).join("XXXXX"));
}
```

In this example, the global variable obj refers to the DOM element bigdata. At the same time, bigdata element refers to the global object through its expandoProperty.

We can see the combination of closures and circular references:

```
function  closureFunction()
{
        var leak =   document.getElementById("element");
        leak.onclick=function innerFunction(){
            alert("Hi! I will leak");};
  leak.bigString = new Array(1000).join(new Array(2000).join("XXXXX")); };
```

Here a JavaScript object leak contains a reference to a DOM object (referenced by the ID "element"). The DOM element, in turn, has a reference to the JavaScript object leak. The resulting circular reference between the JavaScript object and the DOM object causes a memory leak.

One of the most common places associated with memory leaks are setTimeout ()/setInterval () functions.

```
var obj = {
 callMeMaybe: function () {
   var myRef = this;
   var val = setTimeout(function () {
console.log('Time is running out!');
   myRef.callMeMaybe();}, 1000);
   }
};
obj.callMeMaybe();
obj = null;
```

After this section of code, timer still continues to work. An object *obj* isn't cleared, because the closure was transferred to *setTimeout* function and must be maintained for the future performance. In turn, it holds a reference to the life safety as it contains *myRef*. This would be the same if we handed the closure of any other function while retaining the link.

A rule of thumb for all JavaScript applications is obvious. Developers should avoid holding references to DOM elements they no longer need to use, unbind unneeded event listeners and analyze all use cases when storing large chunks of data they are not going to use.

Memory usage measurements

Of course, we need some metrics for memory management. In this section, we would like to discuss memory leaks detection and profiling. There are two main instruments: Google's Chrome Developer [8] and Mozilla Developer [9].

In Chrome Developer Tools, Timeline memory view and Chrome task manager can help developers identify if they are using too much memory. Memory view can track the number of live DOM nodes, documents, and JS event listeners in the inspected render process. The Object allocation tracker can help narrow down leaks by looking at JS object allocation in real-time. Developers can also use the heap profiler to take JS heap snapshots, analyze memory graphs and compare snapshots to discover what objects are not being cleaned up by garbage collection. It is illustrated in Figure 1.

And Figure 2 presents an example of the report. Red nodes (which have a darker background) do not have direct references from JavaScript to them, but are alive because they're part of a detached DOM tree. There may be a node in the tree referenced from JavaScript (maybe as a closure or variable) but is coincidentally preventing the entire DOM tree from being garbage collected.

Yellow nodes (with a yellow background) however do have direct references from JavaScript.

Firefox tool for Garbage Collection measurements looks similar (Figure 3).

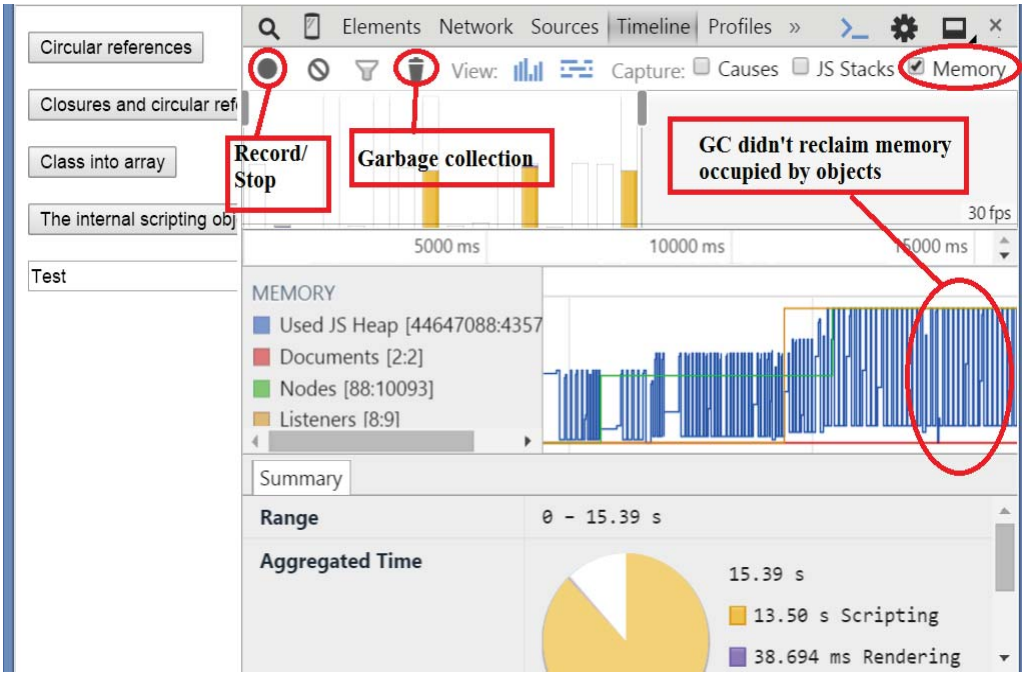

Fig. 1 Chrome Developer Tools [10]

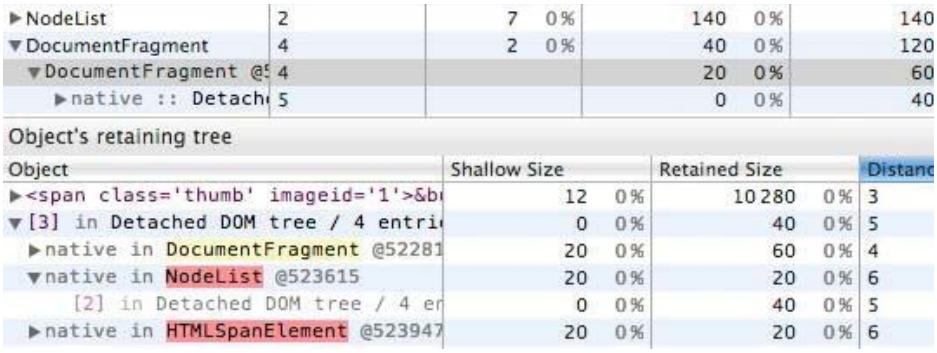

Fig. 2. Google Chrome Developer Report [10]

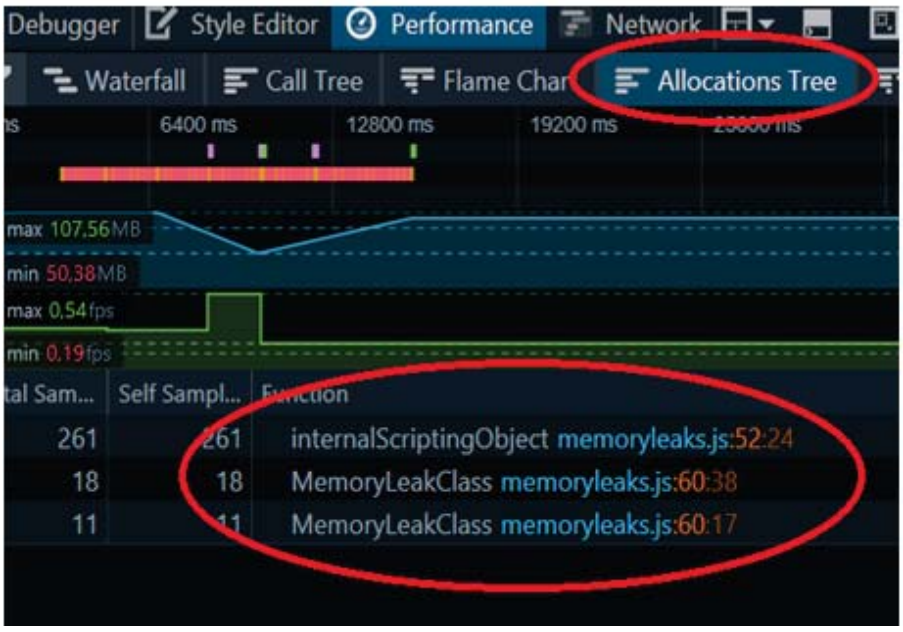

Fig. 3. Firefox memory leaks measurements [2]

Actually, Mozilla provides a list of tools for memory-related measurements [11]. Fox example, Firefox's about:memory page presents fine-grained measurements of memory usage. It is illustrated in Figure 4:

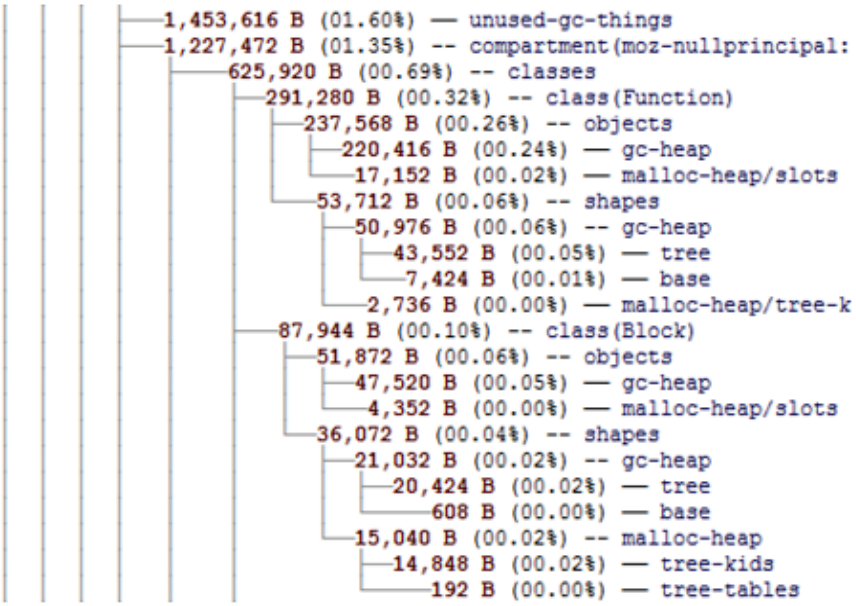

Fig 4. Memory usage in Firefox

Actually, there is a wide list of memory-related measurement tools for Firefox. On practice, most of them, probably, are unknown for developers. The above-mentioned about:memory is the easiest-to-use tool for measuring memory usage. It also lets developers do other memory-related operations like trigger GC and CC, dump GC & CC logs. This tool has got also a special "explanation" reporting – DMD. MD is a tool that identifies shortcomings in about:memory's measurements. The full list of tools (Bloatview, Refcount, GC logs, etc.) is provided in [11]. In the same time, as our experience confirms, many of the tools mentioned on page [11] are either obsolete or not supported anymore.

Garbage Collection in JavaScript

The basic algorithms for Garbage Collectors are well known and widely presented in academic papers [12-13]. Let is review some of the popular choices.

The reference counting algorithm is one of most transparent. An object is considered to be garbage when no references to that object exist. A simple expedient is to keep track in each object of the total number of references to that object. So, the implementation should add a special field to each object called a reference count. Also, every time one reference is assigned to another, the reference counts must be adjusted as above. This increases significantly the time taken by assignment statements.

With reference counting, the garbage (unused data) is easily identified. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the program. If the reference count is zero, the object is garbage. But in the same time, depends on memory allocation scheme, we will still face fragmentation problems.

As far as we understand, at this moment JavaScript engines do not use reference counting in garbage collection. It is used in PHP, for example [14]. Note, that academic papers describe some high-speed reference counting garbage collectors [15-16].

The classical problems for reference counting are so-called circular references. Nevertheless, the reference counting is an extremely useful technique for dealing with simple objects that don't refer to other objects, such as Strings. So, by our opinion, it is an open question: can we use reference counting as a part of the whole garbage collecting process? E.g., we can use it for String only (there are no circular references). Actually, for JavaScript, String objects are most used in practical applications.

The mark-and-sweep algorithm was the first garbage collection algorithm to be developed for processing cyclic data structures. Variations of the mark-and-sweep algorithm continue to be among the most commonly used garbage collection techniques. At this moment, it is utility (commodity) stuff, deployed as a part of many garbage collectors. The mark-and-sweep algorithm is called a tracing garbage collector because is traces out the entire collection of objects that are directly or indirectly accessible by

the program. The objects that a program can access directly are those objects, which are referenced by local variables on the processor stack as well as by any static variables that refer to objects. In the context of garbage collection, these variables are called the roots. An object is indirectly accessible if it is referenced by a field in some other (directly or indirectly) accessible object. Any accessible object is considered to be live. Any other objects are garbage.

The mark-and-sweep algorithm consists of two phases. In the first phase (mark), it finds and marks all accessible objects. During the second phase (sweep), the algorithm walks through the list of objects and reclaims all the dead objects. The mark-and-sweep algorithm can correctly proceed cycled references. Also, we do not need to introduce additional fields (like reference counter) for our data. The main disadvantage of the mark-and-sweep approach is the fact that that normal program execution is suspended while the garbage collection algorithm runs. It is so-called stop-the-world garbage collector.

The copying garbage collector (sometimes is called stop-and-copy and semi-space collector) starts from a set of roots and traverse all of the reachable memory-allocated objects, copying them from one-half of memory into the other half. The area of memory that we copy from is called old space and the area of memory that we copy to is called new space. When we copy the reachable data, we compact it so that it is in a contiguous chunk. This procedure lets us avoid memory fragmentation.

The mark-compact GC is some combination of copying and mark-and-sweep [17]. In the first phase (mark), it finds and marks all live objects. In the second phase (compact), the garbage collection algorithm compresses the heap by moving all the live objects into contiguous memory locations.

The generational garbage collector is based on the idea of partitioning of live objects. This partitioning procedure is based on time of memory allocation. We assume that most objects are discarded shortly after being used. So, we can deploy different GC policies to different partitions. The policy depends on objects' age. Many generational GC's are not comprehensive - they don't successfully remove all the garbage (long-lived garbage, in particular, may never get collected) [18].

All the garbage collectors in JavaScript engines we know are generational.

Interesting that all the above-mentioned implementations are stop-the-world. A sub-collector (an algorithm for garbage collecting within the generation) is still mark-and-sweep. Of course, there are many improvements for the classical mark-and-sweep. We can mention, for example, a lazy sweeping [19]. Mark-region improves the mark-sweep by dividing the heap in several regions and compacts objects to one end of the regions, and can thus reduce memory fragmentation [20]. Garbage-First (G1) works in per-region manner, marks objects and then evacuates them from current regions to new ones so that current regions can be reclaimed as a whole [21]. It is a garbage collector in Oracle JDK [22].

Mark-split removes the sweep phase from mark-sweep, and thus achieves a time complexity proportional to the size of the live data set. However, this comes with an overhead cost of maintaining a set of free memory intervals. The number of free intervals is much smaller than the number of live objects because some live objects reside adjacent to each other. It seems beneficial, in certain situations, to avoid the sweep phase at the cost of this overhead, which depends on the distribution of live objects and also highly on the data structure selected to store the free intervals. The data structure should preferably provide search for an interval at sub-linear cost, e.g., binary search trees, splay trees, or skip-lists [23].

By our opinion, there are at least two most interesting questions. At the first hand, it is not clear at this moment, why JavaScript engines do not use non-stop-the-world garbage collectors. There are concurrent and parallel implementations for mark-and-sweep, for example. They have been tested with Java, for example [24]. But we have not seen yet such implementations in connection with JavaScript. As seems to us, such a movement would be in line with the common trend to add concurrency into JavaScript [25].

The second interesting moment is the policy for running the garbage collector. Actually, it should be more complex than a simply timer-based event or percentage of free memory. It is especially true for the mobile web with relatively limited resources on mobile phones. Just think about the stop-the-world action in the middle of filling some form on the screen. In the same time, a quick stop action could be almost "invisible" during the AJAX request, when a user is waiting for the response anyway. In other words, the policy for running garbage collector should be based on the behavior and depends on the application.

Литература

1. Wagner, G., Gal, A., Wimmer, C., Eich, B., & Franz, M. (2011, June). Compartmental memory management in a modern web browser. In ACM SIGPLAN Notices (Vol. 46, No. 11, pp. 119-128). ACM.
2. Ilyushin E., Namiot D. On JavaScript Memory Leaks //International Journal of Open Information Technologies. – 2015. – T. 3. – №. 7.
3. Pienaar, J. A., & Hundt, R. (2013, February). JSWhiz: Static analysis for JavaScript memory leaks. In Code Generation and

Optimization (CGO), 2013 IEEE/ACM International Symposium on (pp. 1-11). IEEE.
4. JavaScript memory leaks http://javascript.info/tutorial/memory-leaks Retrieved: Jun, 2015.
5. Maffeis, Sergio, John C. Mitchell, and Ankur Taly. "An operational semantics for JavaScript." Programming languages and systems. Springer Berlin Heidelberg, 2008. 307-325.
6. CHEN, Y., & ZHOU, X. (2011). JavaScript Closures Research & Typical Applications. Computer Programming Skills & Maintenance, 10, 012.
7. Heilmann, C. (2006). Beginning JavaScript with DOM scripting and Ajax: from novice to professional. Apress.
8. Chrome Developer tools https://developer.chrome.com/devtools Retrieved: Jun, 2015
9. Mozilla Developer https://developer.mozilla.org/en-US/docs/Tools/Performance/Waterfall#Garbage_collection Retrieved: Jun, 2015
10. Chrome Developer Memory Profiling https://developer.chrome.com/devtools/docs/javascript-memory-profiling Retrieved: Jun, 2015
11. Mozilla Performance https://developer.mozilla.org/en-US/docs/Mozilla/Performance
12. R. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Storage Management. John Wiley & Sons Ltd, 1996.
13. Bruno R. Preiss. (2000). Data Structures and Algorithms with Object-Oriented Design Patterns in Java. John Wiley & Sons Incorporated.
14. Suzumura, T., Trent, S., Tatsubori, M., Tozawa, A., & Onodera, T. (2008, September). Performance comparison of web service engines in php, java and c. In Web Services, 2008. ICWS'08. IEEE International Conference on (pp. 385-392). IEEE.
15. Levanoni, Y., & Petrank, E. (2006). An on-the-fly reference-counting garbage collector for java. ACM Transactions on Programming Languages and Systems (TOPLAS), 28(1), 1-69.
16. Blackburn, S. M., & McKinley, K. S. (2003, October). Ulterior reference counting: Fast garbage collection without a long wait. In ACM SIGPLAN Notices (Vol. 38, No. 11, pp. 344-358). ACM.
17. Jones, R., Hosking, A., & Moss, E. (2011). The garbage collection handbook: the art of automatic memory management. Chapman & Hall/CRC.
18. Tauro, C. J., Prabhu, M. V., & Saldanha, V. J. (2012). CMS and G1 Collector in Java 7 Hotspot: Overview, Comparisons and Performance Metrics. Memory, 43(11).
19. Hughes, R.J.M.: A semi-incremental garbage collection algorithm. Software: Practice and Experience 12(11), 1081–1082 (1982)
20. Blackburn,S.M., McKinley,K.S.:Immix:Amark-region garbage collector with space efficiency, fast collection, and mutator performance. SIGPLAN Not. 43(6), 22–32 (2008)
21. Detlefs, D., Flood, C., Heller, S., Printezis, T.: Garbage-first garbage collection. In: Proceedings of the 4th ISMM, pp. 37–48. ACM (2004)
22. Garbage First in Oracle http://docs.oracle.com/javase/7/docs/technotes/guides/vm/G1.html
23. Sagonas, K., Wilhelmsson, J.: Mark and split. In: Proceedings of the 5th International Symposium on Memory Management, ISMM 2006, pp. 29–39. ACM (2006)
24. Nguyen, Nhan, Philippas Tsigas, and Håkan Sundell. "ParMarkSplit: A Parallel Mark-Split Garbage Collector Based on a Lock-Free Skip-List." Principles of Distributed Systems. Springer International Publishing, 2014. 372-387.
25. Namiot D., Sukhomlin V. JavaScript Concurrency Models //International Journal of Open Information Technologies. – 2015. – T. 3. – №. 6. – C. 21-24.