On JavaScript Memory Leaks

Evgeniy Ilyushin, Dmitry Namiot

***Abstract*— As soon as more and more the modern applications are deployed on the web, JavaScript has become a mainstream programming environment. JavaScript applications nowadays are big programming systems. We can mention here web portals, online games, graphics, media management, and even data science. Of course, the memory management is a very important problem, especially, for the dynamic programming languages. In this paper, we provide a survey of memory leaks patterns in JavaScript.**

***Keywords*—JavaScript, memory management, memory leaks, garbage collection.**

I. Introduction

JavaScript has been around more than 20 years and nowadays is one of the most popular web development languages. It has the ability to deliver rich, dynamic web content as well as being relatively lightweight and easy to use [1]. JavaScript applications nowadays should support heavyweight web applications. We can mention here online games, graphics, media management, and even data mining. In the same time, processors on computing devices (including mobile terminals) are getting more and more elaborated [2]. It converts memory management in JavaScript into a very important issue.

Usually, in JavaScript, we do not think about memory management. Programmers can easily create and reuse objects, where JavaScript engine takes care about low-level details. But in the same time it is very to remember the use cases when the occupied memory for open tab in your browser is getting huge. Usually, it is because of memory leaks.

The central concept of JavaScript memory management is a concept of reachability. 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 [3].

There is a Garbage Collector in the JavaScript engine (in the browser), which cleans memory occupied by unreachable objects [4].

Let us see the following classical example with JavaScript closures.

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 [3].

```
function f() {
  var data = "Large piece of data  …"
;

  /* do something using data */
  function inner() {
    //  …
  }
  return inner;
}
```

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) [5].

In this paper, we would like to survey memory leaks patterns in in JavaScript, as well as memory leaks measurements.

The rest of the paper is organized as follows. In Section II, we describe memory leaks patterns. The Section III is devoted to memory leaks measurements.

II. Memory leaks patterns

In this section, we would like to discuss the typical patterns for memory leaks in JavaScript.

A. Circular references

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

Here the global variable *obj* refers to the DOM element *element* at the same time *element* refers to the global object through its *expandoProperty*.

Manuscript received Jun 15, 2015.

Evgeniy Ilyushin is a student at Lomonosov Moscow State University (e-mail: john.ilyushin@gmail.com).

Dmitry Namiot is senior scientist at Lomonosov Moscow State University (e-mail: dnamiot@gmail.com).

B. *Closures*

```
function operat(x) {
    function operatInn(y) {
        return x + y;
    } ;
    return operatInt;
}

var operat1 = operat(4);
var operat2 = operat1(3);
```

In the example, the *operat* function cannot be collected "garbage collector", as a function object is assigned to a global variable, and is still available. The *operat* function can be used through *operat(n)*.

C. *Closures and circular references*

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

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

D. *Timers*

One of the most common places associated with memory leaks is in a loop, or in *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 continue to work. An object *obj* isn't cleared, because the closure was transferred *setTimeout* 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 closure of any other function, while retaining the link.

III. Memory Leaks Measurements

Of course, we need some metrics for memory management. In this section, we would like to discuss memory leaks detection and profiling.

1. Google's Chrome Developer Tools

Basic terms

- **GC** - garbage collection
- **JSHeap** - graph of related objects
- **Shallow size** - this is the size of memory that is held by the object itself
- **Retained size** - this is the size of memory that is freed once the object itself is deleted along with its dependent objects that were made unreachable from GC roots

Prerequisites

- Should not be running other programs;
- Chromium should be started with the default settings (if you are using some experimental features, reset them to default on the page chrome: // flags);
- Need to leave open only one tab with a test site (this limitation is due to the fact that Chromium can render multiple tabs in a single process, and, accordingly, in the results of the profiling will be unnecessary objects);

Steps:

- Open "Developer tools" and select the Timeline memory view. Then select "Memory" checkbox and start recording.
- Next step, triggering events that may lead to memory leaks some number of times or reload all page for analyzing. Then run the garbage collection and stop recording.

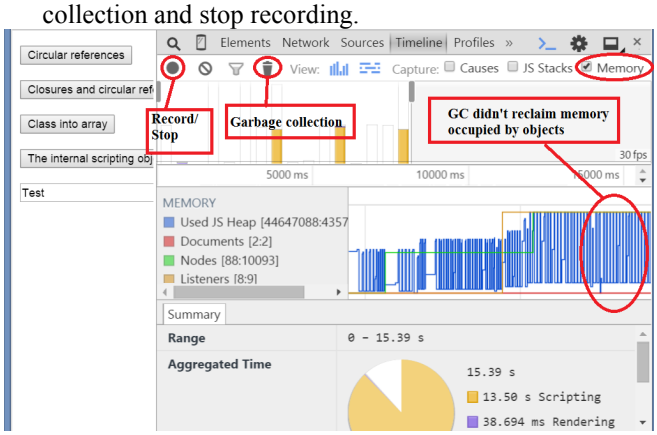

Fig. 1. Developer tools

After garbage collection, all the curves in the graph should come to make the initial state,

otherwise in the test code are present at the memory leaks (Figure 1).

- Analysis of the snapshots. For it, we need to select the Profiles view and take a snapshot. Then you need to follow the steps that lead to the memory leaks and take another snapshot. When we take a snapshot GC is running. After that, we can compare the snapshots and detect memory leaks (Fugure 2).

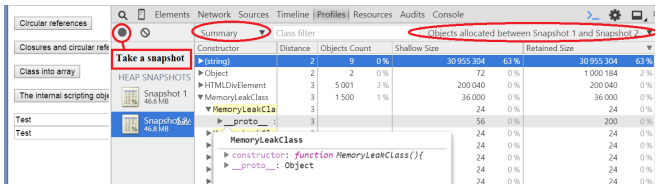

Fig. 2. Snapshots

On the figure, we can see tree of leaking objects. We skip the following objects "compiled code", "closure" and "system". If “array” and “string” have large “shallow size”, we will analyze these objects. If an object has a yellow background, it has a reference that does not allow GC to utilize the object. If an object has a red background, it is detached DOM, that has a reference from JavaScript code. If we move the cursor over an object, Chrome will display a tooltip with information about the object. The information will help us to find the object in code and fix leaks.

2. Mozilla’s Developer Tools

In order to use advanced developer tools of Firefox we need to create a debug build instead of a release build. For more on building process, see the page [6]. In our case we use a release of Firefox "Nightly".

Steps:

- Open a performance tool that gives developers a better understanding of what is happening from a performance standpoint within their applications and start recording performance.

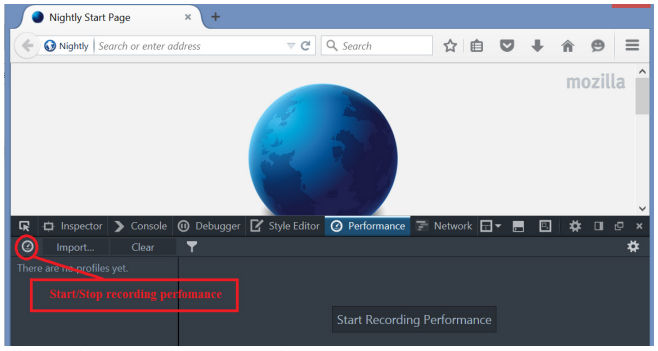

Fig. 3 FireFox performance tools

- Open an analyzed page, follow the steps that lead to the memory leaks some number of times, then stop recording.

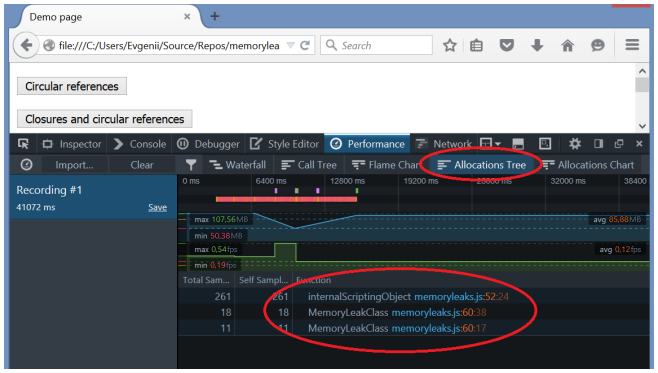

Fig. 4. Data recording

We choose "Allocations tree" view. This we can see objects, which haven't been collected GC. If we click the mouse on an object, tool will open area of JavaScript code related to the object. Also, the recording view gives developers a quick way to zoom into areas where frame rate problems are occurring (Figure 5).

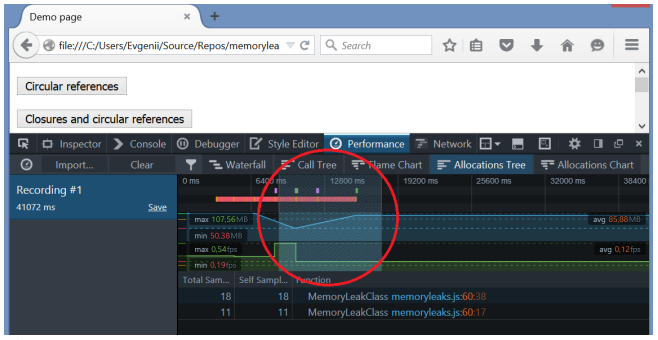

Fig.5 Frames

IV. Conclusion

In this short paper, we provide a survey for JavaScript memory leaks. As the Web programming with JavaScript continues evolving, the size and the complexity of JavaScript applications is constantly growing. Of course, the memory management is very important. For JavaScript applications it is, at the first hand, the quality of Garbage Collector. But before we start to describe Garbage Collectors, we need the whole picture for the typical memory leaks patterns.

Acknowledgment

We would like to thank Samsung Research Center in Moscow for the inspiration of this research.

References

[1] Is JavaScript The Primary Programming Language For The Enterprise? http://www.codeinstitute.net/javascript-primary-programming-language-enterprise/ Retrieved: May, 2015.

[2] Namiot, Dmitry, and Vladimir Sukhomlin. "JavaScript Concurrency Models." International Journal of Open Information Technologies 3.6 (2015): 21-24.

[3] JavaScript memory leaks http://javascript.info/tutorial/memory-leaks Retrieved: Jun, 2015

[4] Maffeis, Sergio, John C. Mitchell, and Ankur Taly. "An operational semantics for JavaScript." Programming languages and systems. Springer Berlin Heidelberg, 2008. 307-325.

[5] Heilmann, C. (2006). Beginning JavaScript with DOM scripting and Ajax: from novice to professional. Apress.

[6] FireFox build https://developer.mozilla.org/en-US/docs/Simple_Firefox_build Retrieved: Jun, 2015

Об утечках памяти в JavaScript

Евгений Ильюшин, Дмитрий Намиот

***Аннотация*— С ростом числа веб-приложений, JavaScript становится одним из основных языков программирования. Современные JavaScript приложения представляют собой достаточно большие программные системы. Мы можем упомянуть здесь веб-порталы, игры, мультимедийные приложения и даже обработку данных. Естественно, управление памятью представляет собой важную проблему, особенно в динамических языка программирования. В данной работе при приводим обзор моделей утечки памяти в JavaScript.**

***Ключевые слова*—JavaScript, управление памятью, утечки памяти, сборка мусора.**