JavaScript 内存泄露的4种方式及如何避免 - 码农网

本文将探索常见的客户端 JavaScript 内存泄露，以及如何使用 Chrome 开发工具发现问题。

简介

内存泄露是每个开发者最终都要面对的问题，它是许多问题的根源：反应迟缓，崩溃，高延迟，以及其他应用问题。

什么是内存泄露？

本质上，内存泄露可以定义为：应用程序不再需要占用内存的时候，由于某些原因，内存没有被操作系统或可用内存池回收。编程语言管理内存的方式各不相同。只有开发者最清楚哪些内存不需要了，操作系统可以回收。一些编程语言提供了语言特性，可以帮助开发者做此类事情。另一些则寄希望于开发者对内存是否需要清晰明了。

JavaScript 内存管理

JavaScript 是一种垃圾回收语言。垃圾回收语言通过周期性地检查先前分配的内存是否可达，帮助开发者管理内存。换言之，垃圾回收语言减轻了“内存仍可用”及“内存仍可达”的问题。两者的区别是微妙而重要的：仅有开发者了解哪些内存在将来仍会使用，而不可达内存通过算法确定和标记，适时被操作系统回收。

JavaScript 内存泄露

垃圾回收语言的内存泄露主因是不需要的引用。理解它之前，还需了解垃圾回收语言如何辨别内存的可达与不可达。

Mark-and-sweep

大部分垃圾回收语言用的算法称之为 Mark-and-sweep 。算法由以下几步组成：

1. 垃圾回收器创建了一个“roots”列表。Roots 通常是代码中全局变量的引用。JavaScript 中，“window”对象是一个全局变量，被当作 root 。window 对象总是存在，因此垃圾回收器可以检查它和它的所有子对象是否存在（即不是垃圾）；
2. 所有的 roots 被检查和标记为激活（即不是垃圾）。所有的子对象也被递归地检查。从 root 开始的所有对象如果是可达的，它就不被当作垃圾。
3. 所有未被标记的内存会被当做垃圾，收集器现在可以释放内存，归还给操作系统了。

现代的垃圾回收器改良了算法，但是本质是相同的：可达内存被标记，其余的被当作垃圾回收。

不需要的引用是指开发者明知内存引用不再需要，却由于某些原因，它仍被留在激活的 root 树中。在 JavaScript 中，不需要的引用是保留在代码中的变量，它不再需要，却指向一块本该被释放的内存。有些人认为这是开发者的错误。

为了理解 JavaScript 中最常见的内存泄露，我们需要了解哪种方式的引用容易被遗忘。

三种类型的常见 JavaScript 内存泄露

1：意外的全局变量

JavaScript 处理未定义变量的方式比较宽松：未定义的变量会在全局对象创建一个新变量。在浏览器中，全局对象是 `window` 。

```
function foo(arg) {
    bar = "this is a hidden global variable";
}
```

真相是：

```
function foo(arg) {
    window.bar = "this is an explicit global variable";
}
```

函数 `foo` 内部忘记使用 `var` ，意外创建了一个全局变量。此例泄露了一个简单的字符串，无伤大雅，但是有更糟的情况。

另一种意外的全局变量可能由 `this` 创建：

```
function foo() {
    this.variable = "potential accidental global";
}

// Foo 调用自己，this 指向了全局对象（window）
// 而不是 undefined
foo();
```

> 在 JavaScript 文件头部加上 `'use strict'`，可以避免此类错误发生。启用严格模式解析 JavaScript ，避免意外的全局变量。

全局变量注意事项
尽管我们讨论了一些意外的全局变量，但是仍有一些明确的全局变量产生的垃圾。它们被定义为不可回收（除非定义为空或重新分配）。尤其当全局变量用于临时存储和处理大量信息时，需要多加小心。如果必须使用全局变量存储大量数据时，确保用完以后把它设置为 null 或者重新定义。与全局变量相关的增加内存消耗的一个主因是缓存。缓存数据是为了重用，缓存必须有一个大小上限才有用。高内存消耗导致缓存突破上限，因为缓存内容无法被回收。

2：被遗忘的计时器或回调函数

在 JavaScript 中使用 `setInterval` 非常平常。一段常见的代码：

```
var someResource = getData();
setInterval(function() {
    var node = document.getElementById('Node');
    if(node) {
        // 处理 node 和 someResource
        node.innerHTML = JSON.stringify(someResource));
    }
}, 1000);
```

此例说明了什么：与节点或数据关联的计时器不再需要，`node` 对象可以删除，整个回调函数也不需要了。可是，计时器回调函数仍然没被回收（计时器停止才会被回收）。同时，`someResource` 如果存储了大量的数据，也是无法被回收的。

对于观察者的例子，一旦它们不再需要（或者关联的对象变成不可达），明确地移除它们非常重要。老的 IE 6 是无法处理循环引用的。如今，即使没有明确移除它们，一旦观察者对象变成不可达，大部分浏览器是可以回收观察者处理函数的。

观察者代码示例：

```
var element = document.getElementById('button');
function onClick(event) {
    element.innerHTML = 'text';
}

element.addEventListener('click', onClick);
```

对象观察者和循环引用注意事项
老版本的 IE 是无法检测 DOM 节点与 JavaScript 代码之间的循环引用，会导致内存泄露。如今，现代的浏览器（包括 IE 和 Microsoft Edge）使用了更先进的垃圾回收算法，已经可以正确检测和处理循环引用了。换言之，回收节点内存时，不必非要调用 `removeEventListener` 了。

3：脱离 DOM 的引用

有时，保存 DOM 节点内部数据结构很有用。假如你想快速更新表格的几行内容，把每一行 DOM 存成字典（JSON 键值对）或者数组很有意义。此时，同样的 DOM 元素存在两个引用：一个在 DOM 树中，另一个在字典中。将来你决定删除这些行时，需要把两个引用都清除。

```
var elements = {
    button: document.getElementById('button'),
    image: document.getElementById('image'),
    text: document.getElementById('text')
};

function doStuff() {
    image.src = 'http://some.url/image';
    button.click();
```

```
    console.log(text.innerHTML);
    // 更多逻辑
}

function removeButton() {
    // 按钮是 body 的后代元素
    document.body.removeChild(document.getElementById('button'));

    // 此时，仍旧存在一个全局的 #button 的引用
    // elements 字典。button 元素仍旧在内存中，不能被 GC 回收。
}
```

此外还要考虑 DOM 树内部或子节点的引用问题。假如你的 JavaScript 代码中保存了表格某一个 `<td>` 的引用。将来决定删除整个表格的时候，直觉认为 GC 会回收除了已保存的 `<td>` 以外的其它节点。实际情况并非如此：此`<td>` 是表格的子节点，子元素与父元素是引用关系。由于代码保留了 `<td>` 的引用，导致整个表格仍待在内存中。保存 DOM 元素引用的时候，要小心谨慎。

4：闭包

闭包是 JavaScript 开发的一个关键方面：匿名函数可以访问父级作用域的变量。

代码示例：

```
var theThing = null;
var replaceThing = function () {
  var originalThing = theThing;
  var unused = function () {
    if (originalThing)
      console.log("hi");
  };

  theThing = {
    longStr: new Array(1000000).join('*'),
    someMethod: function () {
      console.log(someMessage);
    }
  };
};

setInterval(replaceThing, 1000);
```

代码片段做了一件事情：每次调用 `replaceThing` ，`theThing` 得到一个包含一个大数组和一个新闭包（`someMethod`）的新对象。同时，变量 `unused` 是一个引用 `originalThing` 的闭包（先前的 `replaceThing` 又调用了 `theThing` ）。思绪混乱了吗？最重要的事情是，闭包的作用域一旦创建，它们有同样的父级作用域，作用域是共享的。`someMethod` 可以通过 `theThing` 使用，`someMethod` 与 `unused` 分享闭包作用域，尽管 `unused`从未使用，它引用的 `originalThing` 迫使

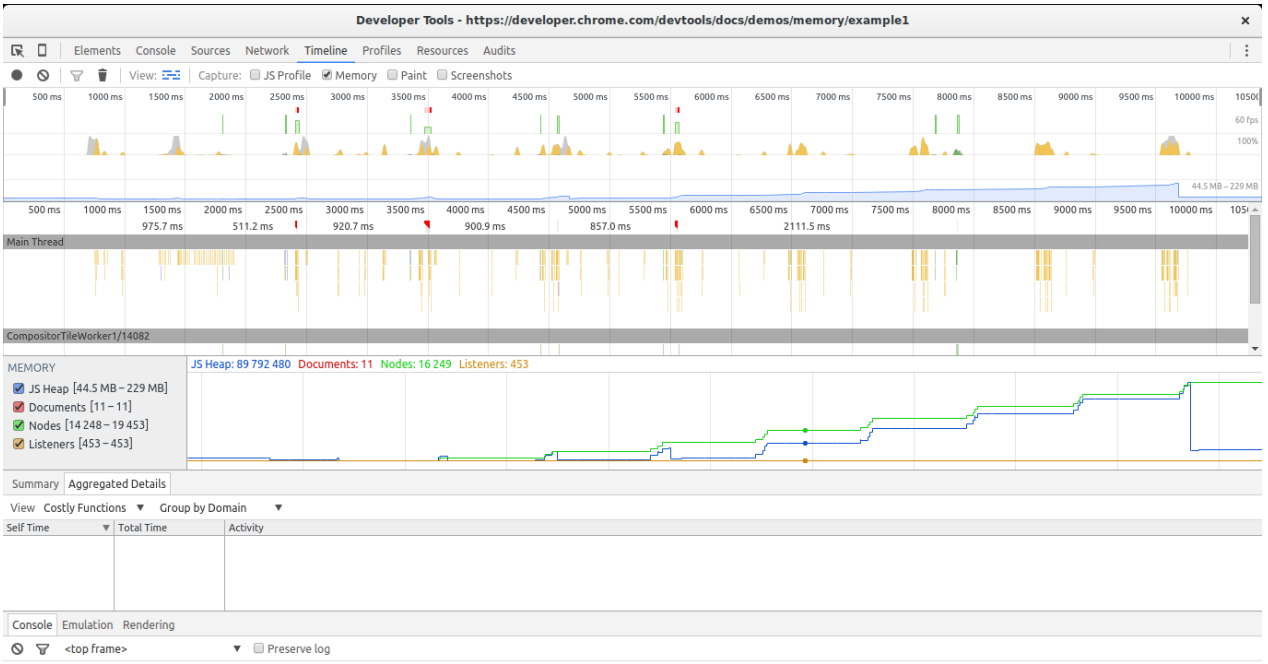
它保留在内存中（防止被回收）。当这段代码反复运行，就会看到内存占用不断上升，垃圾回收器（GC）并无法降低内存占用。本质上，闭包的链表已经创建，每一个闭包作用域携带一个指向大数组的间接的引用，造成严重的内存泄露。

> Meteor 的博文 解释了如何修复此种问题。在 `replaceThing` 的最后添加 `originalThing = null` 。

Chrome 内存剖析工具概览

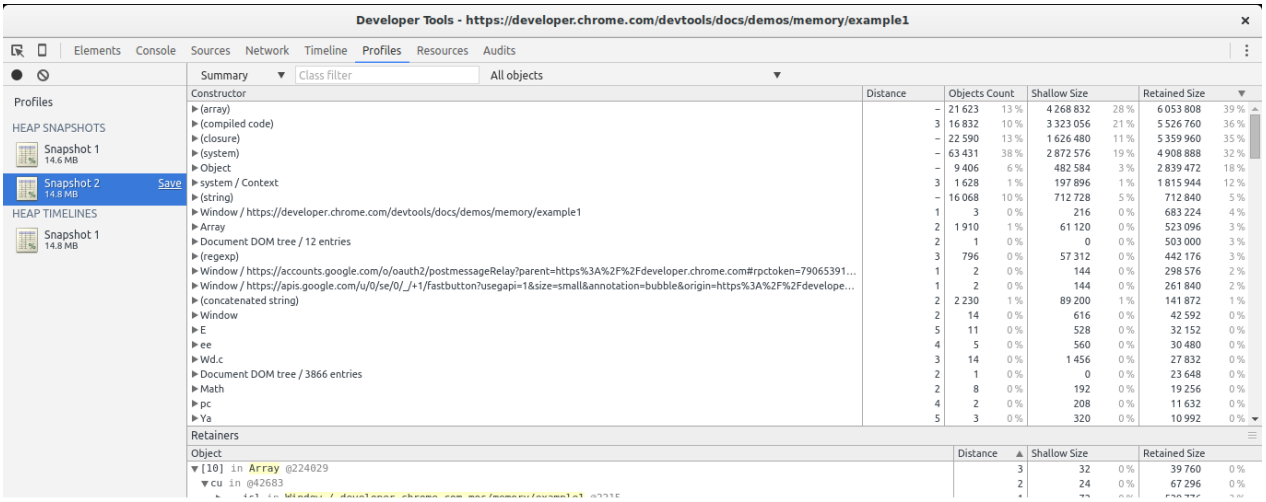
Chrome 提供了一套很棒的检测 JavaScript 内存占用的工具。与内存相关的两个重要的工具：`timeline` 和 `profiles`。

Timeline



timeline 可以检测代码中不需要的内存。在此截图中，我们可以看到潜在的泄露对象稳定的增长，数据采集快结束时，内存占用明显高于采集初期，Node（节点）的总量也很高。种种迹象表明，代码中存在 DOM 节点泄露的情况。

Profiles



Profiles 是你可以花费大量时间关注的工具，它可以保存快照，对比 JavaScript 代码内存使用的不同快照，也可以记录时间分配。每一次结果包含不同类型的列表，与内存泄露相关的有 summary（概要） 列表和 comparison（对照） 列表。

summary（概要） 列表展示了不同类型对象的分配及合计大小：shallow size（特定类型的所有对象的总大小），retained size（shallow size 加上其它与此关联的对象大小）。它还提供了一个概念，一个对象与关联的 GC root 的距离。

对比不同的快照的 comparison list 可以发现内存泄露。

实例：使用 Chrome 发现内存泄露

实质上有两种类型的泄露：周期性的内存增长导致的泄露，以及偶现的内存泄露。显而易见，周期性的内存泄露很容易发现；偶现的泄露比较棘手，一般容易被忽视，偶尔发生一次可能被认为是优化问题，周期性发生的则被认为是必须解决的 bug。

以 Chrome 文档中的代码为例：

```
var x = [];

function createSomeNodes() {
    var div,
        i = 100,
        frag = document.createDocumentFragment();

    for (;i > 0; i--) {
        div = document.createElement("div");
        div.appendChild(document.createTextNode(i + " - "+ new Date().toTimeString()));
        frag.appendChild(div);
    }
```

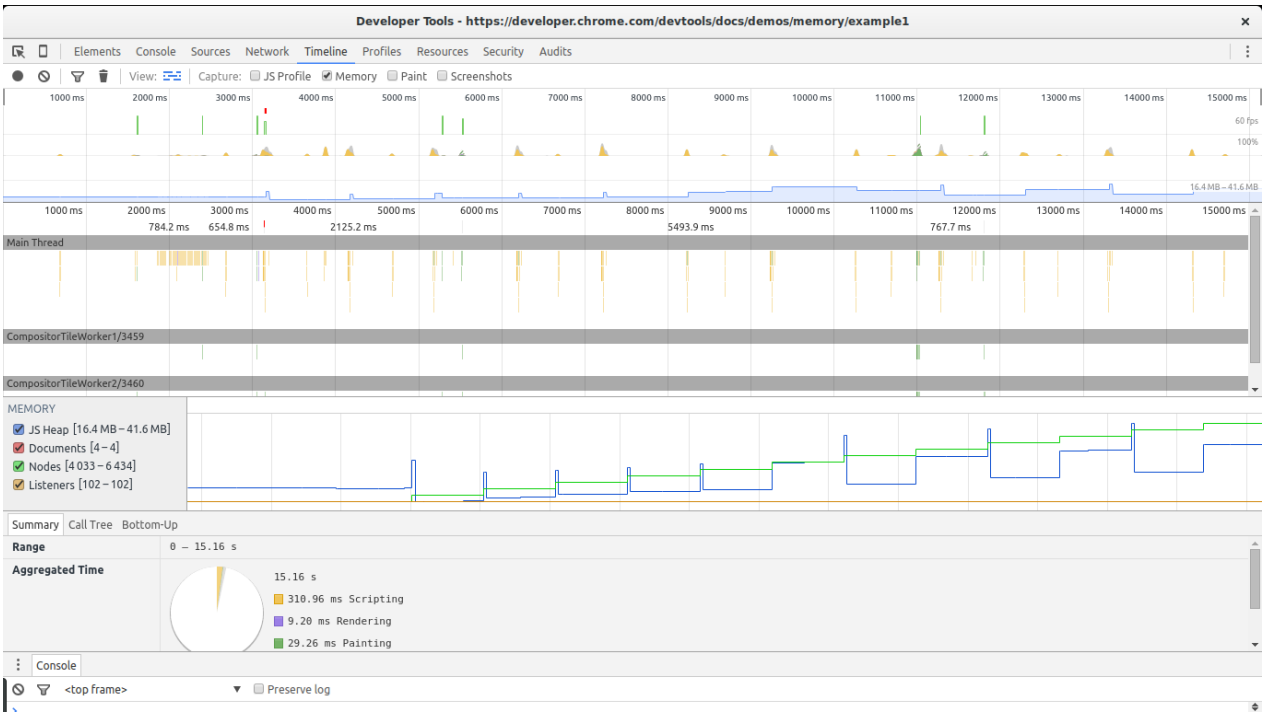
```
    document.getElementById("nodes").appendChild(frag);
}

function grow() {
    x.push(new Array(1000000).join('x'));
    createSomeNodes();
    setTimeout(grow,1000);
}
```

当 `grow` 执行的时候，开始创建 div 节点并插入到 DOM 中，并且给全局变量分配一个巨大的数组。通过以上提到的工具可以检测到内存稳定上升。

找出周期性增长的内存

timeline 标签擅长做这些。在 Chrome 中打开例子，打开 Dev Tools ，切换到 timeline，勾选 memory 并点击记录按钮，然后点击页面上的 `The Button` 按钮。过一阵停止记录看结果：



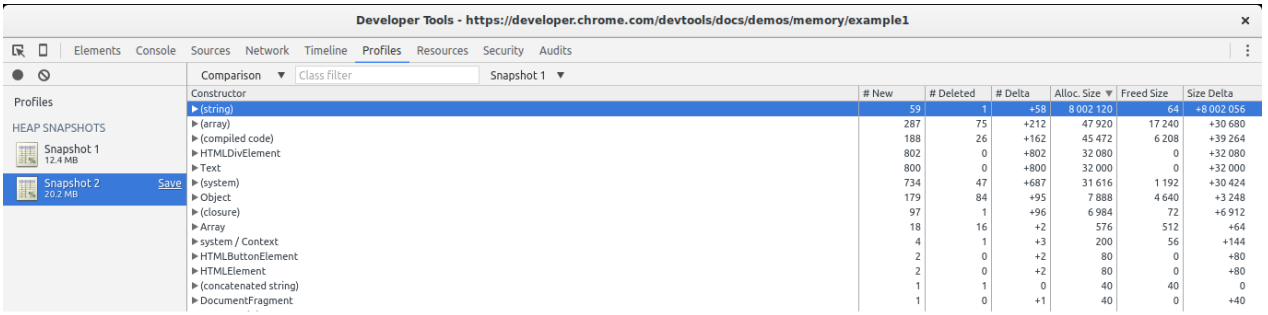
两种迹象显示出现了内存泄露，图中的 Nodes（绿线）和 JS heap（蓝线）。Nodes 稳定增长，并未下降，这是个显著的信号。

JS heap 的内存占用也是稳定增长。由于垃圾收集器的影响，并不那么容易发现。图中显示内存占用忽涨忽跌，实际上每一次下跌之后，JS heap 的大小都比原先大了。换言之，尽管垃圾收集器不断的收集内存，内存还是周期性的泄露了。

确定存在内存泄露之后，我们找找根源所在。

保存两个快照

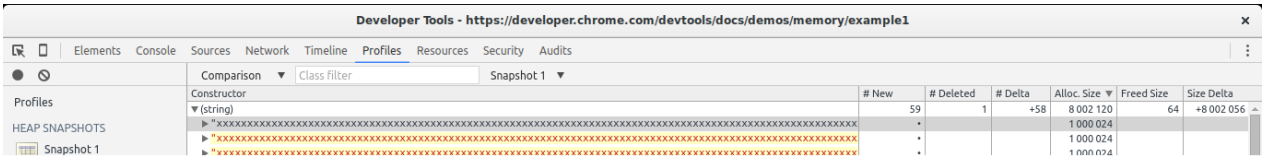
切换到 Chrome Dev Tools 的 profiles 标签，刷新页面，等页面刷新完成之后，点击 Take Heap Snapshot 保存快照作为基准。而后再次点击 `The Button` 按钮，等数秒以后，保存第二个快照。



筛选菜单选择 Summary，右侧选择 Objects allocated between Snapshot 1 and Snapshot 2，或者筛选菜单选择 Comparison ，然后可以看到一个对比列表。

此例很容易找到内存泄露，看下 `(string)` 的 `Size Delta` Constructor，8MB，58个新对象。新对象被分配，但是没有释放，占用了8MB。

如果展开 `(string)` Constructor，会看到许多单独的内存分配。选择某一个单独的分配，下面的 retainers 会吸引我们的注意。



我们已选择的分配是数组的一部分，数组关联到 `window` 对象的 `x` 变量。这里展示了从巨大对象到无法回收的 root（`window`）的完整路径。我们已经找到了潜在的泄露以及它的出处。

我们的例子还算简单，只泄露了少量的 DOM 节点，利用以上提到的快照很容易发现。对于更大型的网站，Chrome 还提供了 Record Heap Allocations 功能。

Record heap allocations 找内存泄露

回到 Chrome Dev Tools 的 profiles 标签，点击 Record Heap Allocations。工具运行的时候，注意顶部的蓝条，代表了内存分配，每一秒有大量的内存分配。运行几秒以后停止。

上图中可以看到工具的杀手锏：选择某一条时间线，可以看到这个时间段的内存分配情况。尽可能选择接近峰值的时间线，下面的列表仅显示了三种 constructor：其一是泄露最严重的`（string）`，下一个是关联的 DOM 分配，最后一个是 `Text` constructor（DOM 叶子节点包含的文本）。

从列表中选择一个 `HTMLDivElement` constructor，然后选择 `Allocation stack`。

现在知道元素被分配到哪里了吧（`grow` -> `createSomeNodes`），仔细观察一下图中的时间线，发现 `HTMLDivElement` constructor 调用了许多次，意味着内存一直被占用，无法被 GC 回收，我们知道了这些对象被分配的确切位置（`createSomeNodes`）。回到代码本身，探讨下如何修复内存泄露吧。

另一个有用的特性

在 heap allocations 的结果区域，选择 Allocation。

这个视图呈现了内存分配相关的功能列表，我们立刻看到了 `grow` 和 `createSomeNodes`。当选择 `grow` 时，看看相关的 object constructor，清楚地看到 `(string)`, `HTMLDivElement` 和 `Text` 泄露了。

结合以上提到的工具，可以轻松找到内存泄露。