# 一、内存泄漏（不再需要 (没有作用) 的内存数据依然被其他对象引用着。 ）

**表现：**是内存一直在缓慢增长。

例子1：

function foo() {

//创建一个临时的temp\_array

temp\_array = new Array(200000)

}

解释：当执行这段代码时，由于函数体内的对象没有被 var、let、const 这些关键字声明，那么 V8 就会使用 this.temp\_array 替换 temp\_array。 在浏览器，默认情况下，this 是指向 window 对象的，而 window 对象是常驻内存的，所 以即便 foo 函数退出了，但是 temp\_array 依然被 window 对象引用了， 所以 temp\_array 依然也会和 window 对象一样，会常驻内存。因为 temp\_array 已经是不再被 使用的对象了，但是依然被 window 对象引用了，这就造成了 temp\_array 的泄漏。

例子2：时刻警惕闭包这种情况，因为闭包会引用父级函数中定义的变量，如果引用了不被需要的变量，那么也会造成内存泄漏。

function foo(){

var temp\_object = new Object()

temp\_object.x = 1

temp\_object.y = 2

temp\_object.array = new Array(200000)

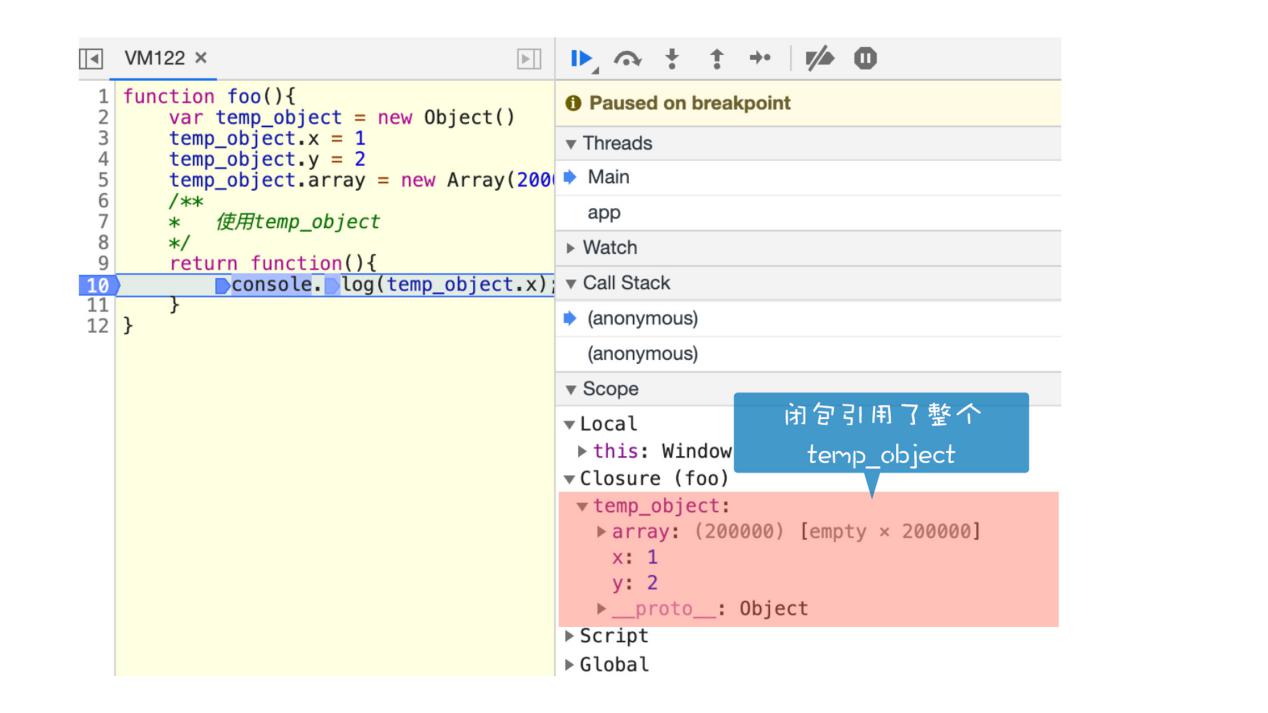
return function(){

console.log(temp\_object.x);

}

}

解释：foo 函数使用了一个局部临时变量 temp\_object，temp\_object 对象有三个属性，x、y，还有一个非常占用内存的 array 属性。最后 foo 函数返回了一个匿名函数，该匿名函数引用了 temp\_object.x。那么当调用完 foo 函数之后，由于返回的匿名函数引用了 foo 函数中的 temp\_object.x，这会造成 temp\_object 无法被销毁，即便只是引用了temp\_object.x，也会造成整个 temp\_object 对象依然保留在内存中。我们可以通过Chrome 调试工具查看下：

从上图可以看出，我们仅仅是需要 temp\_object.x 的值，V8 却保留了整个 temp\_object对象。要解决这个问题，我就需要根据实际情况，来判断闭包中返回的函数到底需要引用什么数据，不需要引用的数据就绝不引用，因为上面例子中，返回函数中只需要 temp\_object.x的值，因此我们可以这样改造下这段代码：

function foo(){

var temp\_object = new Object()

temp\_object.x = 1

temp\_object.y = 2

temp\_object.array = new Array(200000)

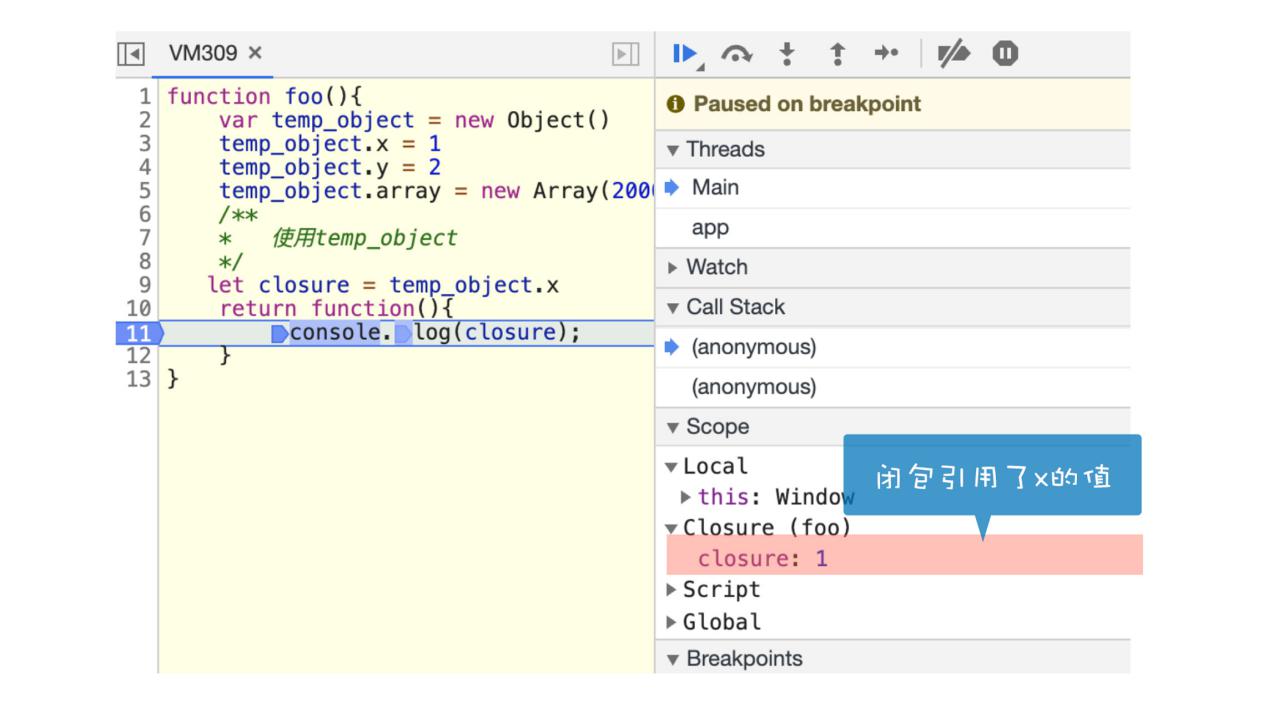
let closure = temp\_object.x10 return function(){

console.log(closure);

}

}

当再次执行这段代码时，我们就可以看到闭包引用的仅仅是一个 closure 的变量，最终如下图所示：



例子3：引用了 DOM 节点而造成的内存泄漏的问题，只有同时满足DOM 树和JavaScript 代码都不引用某个 DOM 节点，该节点才会被作为垃圾进行回收。如果某个节点已DOM 树移除，但 JavaScript 仍然引用它，我们称此节点为“detached ”。“detached ”节点是 DOM 内存泄漏的常见原因。

let detachedTree;

function create() {

var ul = document.createElement('ul');

for (var i = 0; i < 100; i++) {

var li = document.createElement('li');

ul.appendChild(li);

}

detachedTree = ul;

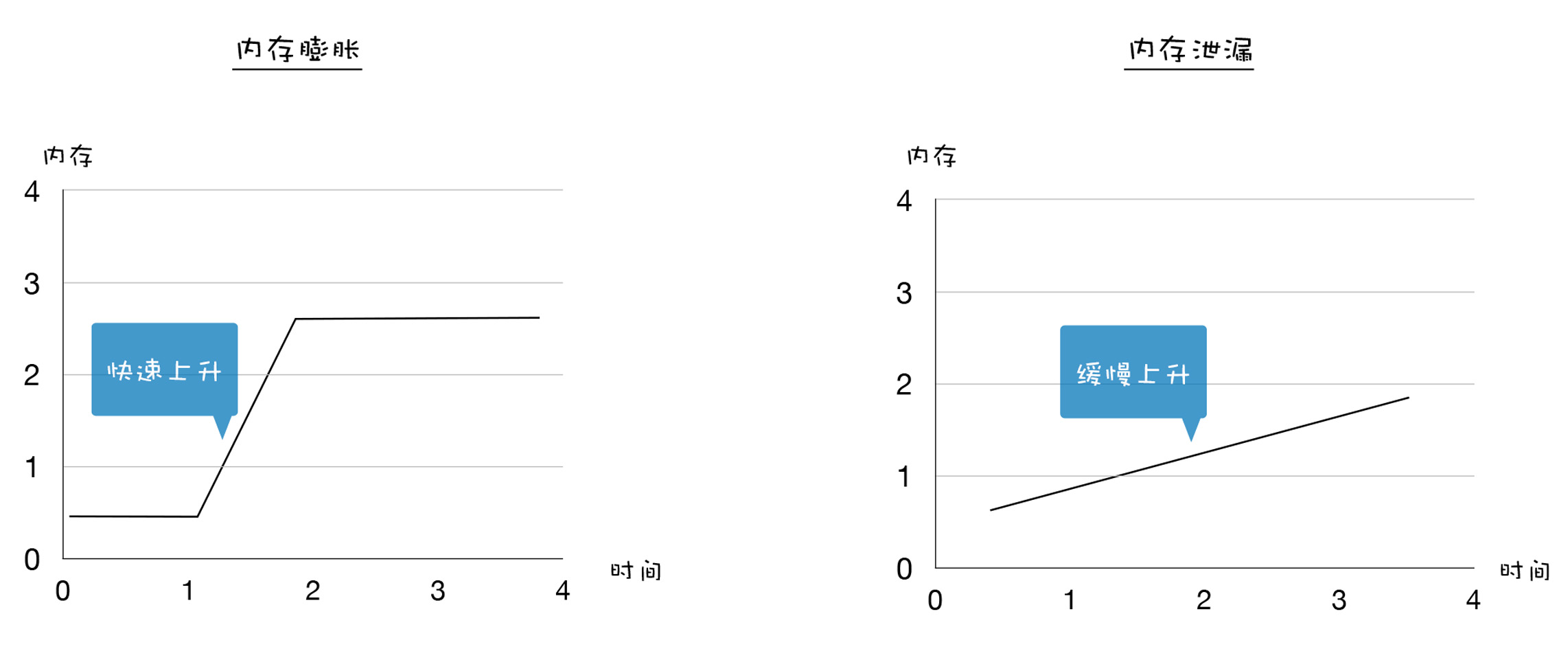
}

create()

解释：我们通过 JavaScript 创建了一些 DOM 元素，有了这些内存中的 DOM 元素，当有需要的时候，我们就快速地将这些 DOM 元素关联到 DOM 树上，一旦这些 DOM 元素从 DOM上被移除后，它们并不会立即销毁，这主要是由于 JavaScript 代码中保留了这些元素的引用，导致这些 DOM 元素依然会呆在内存中。所以在保存 DOM 元素引用的时候，我们需要非常小心谨慎。

# **二、内存膨胀（主要表现在程序员对内存管理的不科学）**

**表现：**内存在某一段时间内快速增长，然后达到一个平稳的峰值继续运行



例子1：只需要 50M 内存就可以搞定的，有些程序员却花费了 500M 内存

例子2：没有充分地利用好缓存

例子3：加载了一些不必要的资源

例子4：一次性加载了大量的资源，内存会快速达到一个峰值

**解决办法：**合理规划项目，充分利用缓存等技术来减轻项目中不必要的内存占用

**三、频繁的垃圾回收（频繁使用大的临时变量，导致了新生代空间很快被装满，从而频繁触发垃圾回收。频繁的垃圾回收操作会让你感觉到页面卡顿。）**

例子1：

function strToArray(str) {

let i = 0

const len = str.length

let arr = new Uint16Array(str.length)

for (; i < len; ++i) {

arr[i] = str.charCodeAt(i)

}

return arr;

}

function foo() {

let i = 0

let str = 'test V8 GC'

while (i++ < 1e5) {

strToArray(str);

}

}

解释：这段代码就会频繁创建临时变量，这种方式很快就会造成新生代内存内装满，从而频繁触发垃圾回收。为了解决频繁的垃圾回收的问题，你可以考虑将这些临时变量设置为全局变量。