06 JS 内存管理机制解析

更新时间：2020-03-13 10:37:02

> 与有肝胆人共事，从无字句处读书。——周恩来

JS 内存机制相关的面试问题，常常紧跟在闭包之后出现。

很多时候，面试官会抛出类似 “闭包中与内存泄漏之间的关系” 这样的问题作为引子。若是单纯的就事论事，这类问题并不算难，寥寥数行就可以回答清楚。

可惜面试官很多时候都不是为了就事论事，而是准备深挖 JS 的内存管理、垃圾回收机制、以及内存泄漏成因等等具有相当深度的问题。从我个人的经验来看，每次抛出这类问题，都可以拍死一大片功底不够扎实的同学。所以这节，我们就从根儿上刨起，把这一串的知识链路捋透彻，让面试官挖无可挖。

JS 内存生命周期

内存管理是每一种编程语言都会具备的一种基本能力。

区别在于，一些语言会将这种能力开放 —— 比如 C 语言中的 `malloc()` 和 `free()` 方法，这些方法的暴露，使得开发者能够切身感受到内存管理这件事情的存在。

而在另一些语言 —— 比如 JS 中，这种能力是被 “隐藏” 了的：JS 并没有暴露任何内存操作给开发者，而是自己默默地自动完成了所有的管理动作。这是 JS 内存管理不被大多数同学所重视的原因。

JS 的内存生命周期，和大多数程序语言一样，分为三个阶段。这三个阶段非常好理解，跟农民种地的过程差不多：

- “挖坑”—— 在内存空间这片广袤无垠的沃土里，划出自己的一亩三分地，此举称为 “分配内存”。
- “用坑”—— 往你的一亩三分地里 “种菜” ：塞入你需要存储的信息。此后你可以读取它，也可以更改它，此举称为 “内存的读与写” 操作。
- “还坑”—— 用坑一时爽，但作为好公民，咱用完这个地就得及时上交给村里。这个 “还回去” 的动作，就叫做内存的释放。

如此一来呢，想必大家都对这个 “种地” 一般的生命周期流程心知肚明了。但是，光知道流程，你还未必能种出好果子。要想奔小康，你还得知道这 “挖坑” 阶段里头的另一层门道 —— 不同的果子（数据），它喜欢不同的土壤，咱们必须从每一种 “果子” 的类型和特性出发，为它开辟不同类型的内存空间。

栈内存与堆内存

基本类型和引用类型

JS 中的数据类型，整体上来说只有两类：基本类型和引用类型。

其中基本类型包括：Sting、Number、Boolean、null、undefined、Symbol。这类型的数据最明显的特征是**大小固定、体积轻量、相对简单**，它们被放在 JS 的栈内存里存储。

而排除掉基本类型，剩下的数据类型就是**引用类型**，比如 Object、Array、Function 等等等等。这类数据**比较复杂、占用空间较大、且大小不定**，它们被放在 JS 的堆内存里存储。

图解堆与栈

大家知道，堆和栈分别是不同的数据结构。栈是线性表的一种，而堆则是树形结构。

若要从计算机理论的角度来理解堆栈内存，相对会比较生涩。其实，只要大家能理解基本类型和引用类型在内存世界里实际上是以一种什么形态存在，这块知识就没法难倒你了。要做到这一点，不必大费周章，咱们直接来看一个例子：

```
let a = 0;
let b = "Hello World"
let c = null;
let d = { name: '修言' };
let e = ['修言', '小明', 'bear'];
```

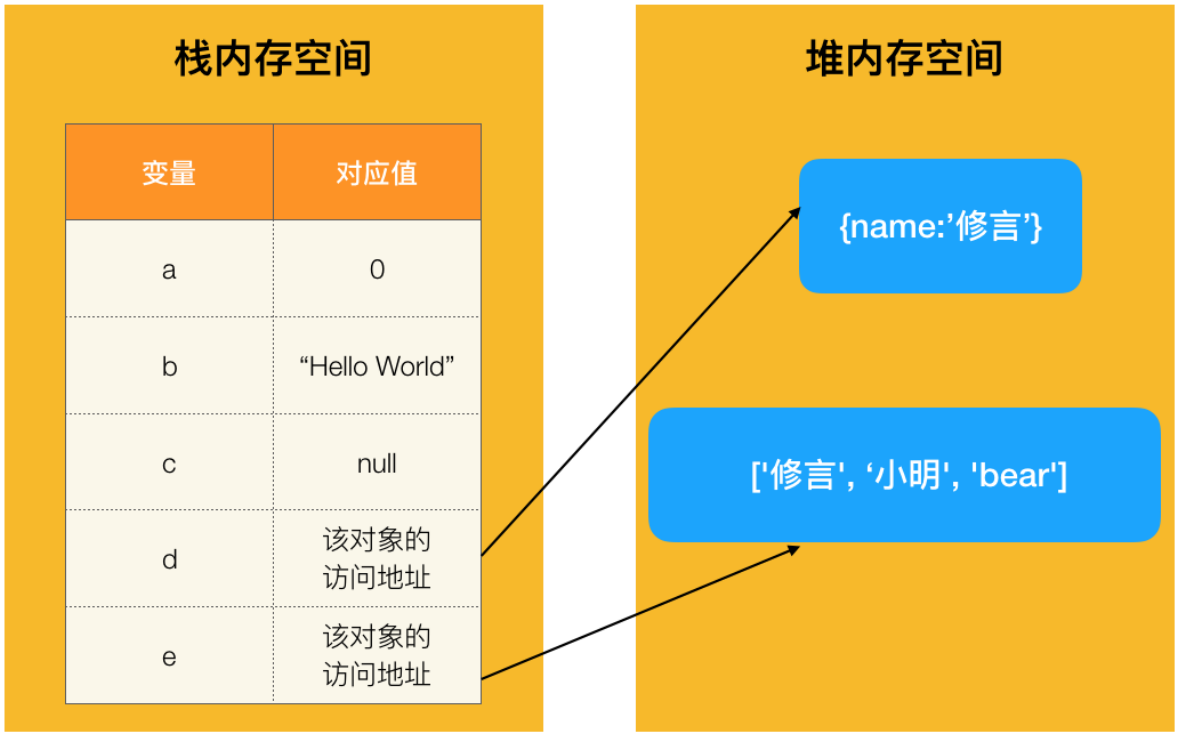
第一个问题：上述五个变量分别对应哪种存储方式？

根据我们刚刚讲过的知识，按照如下的思路，我们不难分析出答案：

```
a —— Number类型 —— 基本类型 —— 栈内存
b —— String类型 —— 基本类型 —— 栈内存
c —— null —— 基本类型 —— 栈内存
d —— Object —— 引用类型 —— 堆内存
e —— Array —— 引用类型 —— 堆内存
```

第二个问题： 上述五个变量在访问机制上有何区别？

这个问题看似在问变量，其实在问内存。这里大家需要对 JS 堆栈存储特征有所掌握。我们这里的 5 个变量，在内存中的形态如图所示：



在访问 a、b、c 三个变量时，过程非常简单：**从栈中直接获取该变量的值。**

而在访问 d 和 e 时，则需要分两步走：

1. 从栈中获取变量对应对象的引用（即它在堆内存中的地址）
2. 拿着 1 中获取到的地址，再去堆内存空间查询，才能拿到我们想要的数据

垃圾回收机制

前面咱们讲完了内存的分配和使用，这节我们来看下内存的释放，也就是 “还坑” 这个动作。

每隔一段时间，JS 的垃圾收集器就会对变量做 “巡检”。当它判断一个变量不再被需要之后，它就会把这个变量所占用的内存空间给释放掉，这个过程叫做垃圾回收。

那么 JS 是如何知道一个变量是否不被需要的呢？ 这里就引出了内存管理的又一个考点 —— 垃圾回收算法。

在 JS 中，我们讨论的垃圾回收算法有两种 —— 引用计数法和标记清除法。

引用计数法

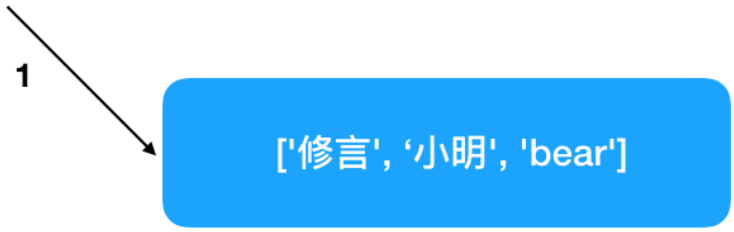
这是最初级的垃圾回收算法，它在现代浏览器里几乎已经被淘汰得干干净净，但是仍有一些面试官执着于询问该方法的思路、以便于判断你对 JS 的了解是否足够全面和深入。因此，我们抛砖引玉，先把这个已经 out 了的方法跟大家唠唠，顺便再一起看看它是怎么 out 的。

在 JS 中，我们强调 “引用” 仅仅用来描述引用类型的内存地址。不过大家要是有一些 JAVA 基础，你会了解到，“引用” 这个概念，其实可以认为它描述的是**变量**所处那块内存的内存地址 —— 这里所说的 “变量” 是一个泛的概念，它泛指所有类型的变量，而不局限于某一种类型。在 “引用计数法” 中，“引用” 这个概念，其主语也正是 JS 环境中的**所有实体**。

当我们用一个变量指向了一个值，那么就创建了一个针对这个值的 “引用”：

```
const students = ['修言', '小明', 'bear']
```

如图，大家知道赋值表达式是从右向左读的。这行代码首先是开辟了一块内存，把右侧这个数组塞了进去，此时这个数组就占据了一块内存。随后 students 变量指向它，这就是创建了一个指向该数组的 “引用”。此时数组的引用计数就是 1（如下图）。

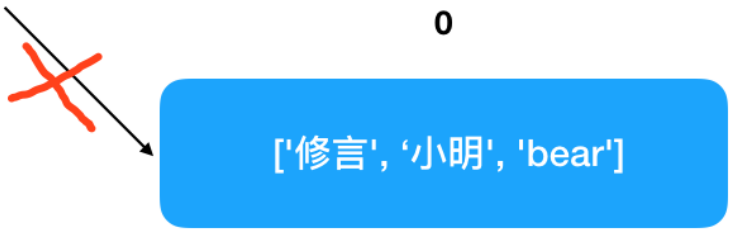


在引用计数法的机制下，内存中的每一个值都会对应一个引用计数。当垃圾收集器感知到某个值的引用计数为 0 时，就判断它 “没用” 了，随即这块内存就会被释放。

比如我们此时如果把 students 指向一个 null：

```
students = null
```

那么 [‘xiuyan’, ‘xiaoming’, ‘bear’] 这个数组所具备的引用计数就会跟着变成 0（如下图），它就变成了一块没用的内存，即将面临着作为 “垃圾” 被回收的命运。



引用计数法糟糕在哪？

引用计数法的限制，是一个比引用计数法本身还要重要的考点。

大家现在来看这样一个例子：

```
function badCycle() {
  var cycleObj1 = {}
  var cycleObj2 = {}
  cycleObj1.target = cycleObj2
  cycleObj2.target = cycleObj1
}

badCycle()
```

在代码第 8 行，我们执行了 badCycle 这个函数。大家知道，函数作用域的生命非常短暂，当函数执行完之后，作用域内的变量也会全部被视作 “垃圾” 进而移除。

但如果咱们用了引用计数法，那么即使 badCycle 执行完毕，cycleObj1 和 cycleObj2 还是会活得好好的 —— 因为 cycleObj2 的引用计数为 1（cycleObj1.target），而 cycleObj1 的引用计数也为 1 （cycleObj2.target）（如下图）。

没错，引用计数法无法甄别**循环引用**场景下的 “垃圾”！

如果任由 cycleObj1、cycleObj2 这样的变量肆虐内存，那么内存泄漏将是不可避免的结局。

标记清除法

考虑到引用计数法存在严重的局限性，自 2012 年起，所有浏览器都使用了标记清除算法。可以说，标记清除法是现代浏览器的标准垃圾回收算法。

在标记清除算法中，一个变量是否被需要的判断标准，是**它是否可抵达** 。

这个算法有两个阶段，分别是标记阶段和清除阶段：

- 标记阶段：垃圾收集器会先找到根对象，在浏览器里，根对象是 Window；在 Node 里，根对象是 Global。从根对象出发，垃圾收集器会扫描所有可以通过根对象触及的变量，这些对象会被标记为 “可抵达”。
- 清除阶段： 没有被标记为 “可抵达” 的变量，就会被认为是不需要的变量，这波变量会被清除

现在大家按照标记清除法的思路，再来看这段代码：

```
function badCycle() {
  var cycleObj1 = {}
  var cycleObj2 = {}
  cycleObj1.target = cycleObj2
  cycleObj2.target = cycleObj1
}

badCycle()
```

badCycle 执行完毕后，从根对象 Window 出发，cycleObj1 和 cycleObj2 都会被识别为不可达的对象，它们会按照预期被清除掉。这样一来，循环引用的问题，就被标记清除干脆地解决掉了。

闭包与内存泄漏

啥是内存泄漏？

该释放的变量（内存垃圾）没有被释放，仍然霸占着原有的内存不松手，导致内存占用不断攀高，带来性能恶化、系统崩溃等一系列问题，这种现象就叫内存泄漏。

闭包并不是洪水猛兽

在面试过程中，内存泄漏这个点会被一些面试官通过闭包来引出。

然而，事实上，单纯由闭包导致的内存泄漏，极少极少（除非你的操作极其不规范，但那就不是闭包的问题了，是代码写得有问题）。真正导致内存泄漏的原因，我们还需要从其他方面来看。不过为了保证大家没有知识盲区，我们先从闭包对内存的“威胁”说起。

面试官未必会直接问你“闭包是如何导致内存泄漏的？”（如果他是一个足够严谨的工程师，想必也不会这样问），而更倾向于让你“看代码找问题”。在“有问题”的代码里，最为大家所津津乐道的其实是“theThing”问题。也就是下面这段流传已久的代码：

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

思考：上面这段代码有什么问题？

要想揪出其中的问题，大家需要对 V8 引擎有所了解，尤其是这一点：**在 V8 中，一旦不同的作用域位于同一个父级作用域下，那么它们会共享这个父级作用域。**

在这段代码里， unused 是一个不会被使用的闭包，但和它共享同一个父级作用域的 someMethod，则是一个“可抵达”（也就意味着可以被使用）的闭包。unused 引用了 originalThing，这导致和它共享作用域的 someMethod 也间接地引用了 originalThing。结果就是 someMethod “被迫”产生了对 originalThing 的持续引用，originalThing 虽然没有任何意义和作用，却永远不会被回收。不仅如此，originalThing 每次 setInterval 都会改变一次指向（指向最近一次的 theThing 赋值结果），这导致无法被回收的无用 originalThing 越堆积越多，最终导致严重的内存泄漏。

内存泄漏成因分析

除了上面分析这种情况，可能导致内存泄漏的因素，还可以考虑以下几点：

1. “手滑”导致的全局变量

```
function test() {
  me = 'xiuyan'
}
```

当你在非严格模式下写代码时，me 而非 var me 这种写法，会导致这个 me 被默默地挂载到全局对象上。

根据我们前面所讲的垃圾回收策略，本来 me 这个变量，如果被 var 声明过，它作为函数作用域内的变量，在函数调用结束后就会消失 —— 这也是我们所期望的。但现在它是一个全局变量了，永远无法被清除。这样的变量一多，问题就来了。

2. 忘记清除的 setInterval 和 setTimeout

我们在实现轮询效果时，会用到 setInterval：

```
setInterval(function() {
    // 函数体
}, 1000);
```

或者链式调用 setTimeout：

```
setTimeout(function() {
  // 函数体
  setTimeout(arguments.callee, 1000);
}, 1000);
```

在 setInterval 和链式调用的 setTimeout 这两种场景下，定时器的工作可以说都是无穷无尽的。当定时器囊括的函数逻辑不再被需要、而我们又忘记手动清除定时器时，它们就会永远保持对内存的占用。因此当我们使用定时器时，一定要先问问自己：我打算什么时候干掉这玩意儿？

3. 清除不当的 DOM

```
const myDiv = document.getElementById('myDiv')

function handleMyDiv() {
    // 一些与myDiv相关的逻辑
}

// 使用myDiv
handleMyDiv()

// 尝试”删除“ myDiv
document.body.removeChild(document.getElementById('myDiv'));
```

有同学以为第 11 行这种写法，就足以对 myDiv 这个 DOM 进行删除。这种想法非常天真，因为 myDiv 这个变量对这个 DOM 的引用仍然存在，它仍然是一块“可抵达”的内存。这种你以为已经清除、但其实活得好好的 DOM 一旦堆积，将带来不可预期的内存隐患。

这里我们针对内存泄漏成因，分析了以上几点。事实上，能够导致内存泄露的，还有无数种“骚操作”，它们有一个共同的名字 —— 误操作。

没错，内存泄漏其实并不是啥高深的命题，导致内存泄露的，往往是低级错误。说得直接点，那就是代码功夫不到家。因此，各位同学无论处在哪个学习阶段，都应该保持严谨的态度 —— 敬畏手中的键盘、敬畏每一行代码！

}

05 闭包的应用

07 this 基本指向原则解析