---

link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=107 sentences=136, words=1031

1.V8内存管理

- 程序运行需要分配内存
- V8也会申请内存，申请的内存又会分为堆内存和栈内存

1.1 栈

- 栈用于存放JS中的基本类型和引用类型指针
- 栈的空间是连续的，增加删除只需要移动指针，操作速度非常快
- 栈的空间是有限的，当栈满了，就会抛出一个错误
- 栈一般是在执行函数时创建的，在函数执行完毕后，栈就会被销毁

1.2 堆

- 如果不需要连续空间，或者申请的内存较大，可以使用堆
- 堆主要用于存储JS中的引用类型
- 源码 (https://gitee.com/zhufengpeixun/v8/blob/master/src/heap/heap.cc)

```
const v8 = require('v8');
const heapSpace = v8.getHeapSpaceStatistics();
function format(size) {
    return `${(size / 1024 / 1024).toFixed(2)}M`.padEnd(10, ' ');
}
console.log(`${"空间名称".padEnd(20, ' ')} 空间大小  已用空间大小  可用空间大小  物理空间大小`);
for (let i = 0; i < heapSpace.length; i++) {
    const space = heapSpace[i];
    console.log(`${space.space_name.padEnd(23, ' ')}`,
        `${format(space.space_size)}`,
        `${format(space.space_used_size)}`,
        `${format(space.space_available_size)}`,
        `${format(space.physical_space_size)}`);
}
```

1.2.1 堆空间分类

1.2.1.1 新生代(newspace)

- 新生代内存用于存放一些生命周期比较短的对象数据

1.2.1.2 老生代(old space)

- 老生代 (https://gitee.com/zhufengpeixun/v8/blob/master/src/heap/paged-spaces.cc)内存用于存放一些生命周期比较长的对象数据
- 当 new space的对象进行两个周期的垃圾回收后，如果数据还存在 new space中，则将他们存放到 old space中
- old space又可以分为两部分，分别是Old pointer space和Old data space
    - Old pointer space 存放GC后surviving的指针对象
    - Old data space 存放GC后surviving的数据对象
- Old Space使用标记清除和标记整理的方式进行垃圾回收

1.2.1.3 Code space

- 用于存放JIT已编译的代码
- 唯一拥有执行权限的内存

1.2.1.4 Large object space

- 为了避免大对象的拷贝，使用该空间专门存储大对象
- GC 不会回收这部分内存

1.2.1.5 Map space

- 存放对象的Map信息，即隐藏类
- 隐藏类是为了提升对象属性的访问速度的
- V8 会为每个对象创建一个隐藏类，记录了对象的属性布局，包括所有的属性和偏移量

1.2.2 什么是垃圾

- 在程序运行过程中肯定会用到一些数据，这些数据会放在堆栈中，但是在程序运行结束后，这些数据就不会再被使用了，那些不再使用的数据就是垃圾

```
global.a = { name: 'a' };
global.a.b = { name: 'b1' };
global.a.b = { name: 'b2' };
```

1.2.3 新生代的垃圾回收

- 新生代内存有两个区域，分别是对象区域(from) 和 空闲区域(to)
- 新生代内存使用 Scavenger算法来管理内存,垃圾回收的入口 (https://gitee.com/zhufengpeixun/v8/blob/master/src/heap/heap-inl.h)
    - 广度优先遍历 From-Space 中的对象，从根对象出发，广度优先遍历所有能到达的对象,把存活的对象复制到 To-Space
    - 遍历完成后，清空 From-Space
    - From-Space 和 To-Space 角色互换
- 复制后的对象在 To-Space 中占用的内存空间是连续的，不会出现碎片问题
- 这种垃圾回收方式快速而又高效，但是会造成空间浪费
- 新生代的 GC 比较频繁
- 新生代的对象转移到老生代称为晋升Promote (https://gitee.com/zhufengpeixun/v8/blob/master/src/heap/heap.h#L760),判断晋升 (https://gitee.com/zhufengpeixun/v8/blob/master/src/heap/heap-inl.h#L522)的情况有两种
    - 经过一次 GC 还存活的对象
    - 对象复制到 To-Space 时，To-Space 的空间达到一定的限制

```
global.a={};
global.b = {e: {}}
global.c = {f: {},g:{h:{}}}
global.d = {};
global.d=null;
```

广度优先遍历过程 (https://www.processon.com/diagraming/61b429bbf346fb2874b191e1)

```
bool Heap::ShouldBePromoted(Address old_address) {
  Page* page = Page::FromAddress(old_address);
  Address age_mark = new_space_->age_mark();
  return page->IsFlagSet(MemoryChunk::NEW_SPACE_BELOW_AGE_MARK) &&
         (!page->ContainsLimit(age_mark) || old_address < age_mark);
}
```

1.2.4 老生代的垃圾回收

- 老生代里的对象有些是从新生代晋升过来的，有些是比较大的对象直接分配到老生代里的，所以老生代的对象空间大，活的长
- 如果使用Scavenge算法，浪费一半空间不说，复制如此大块的内存消耗时间将会相当长。所以Scavenge算法显然不适合
- V8在老生代中的垃圾回收策略采用Mark-Sweep(标记清除)和Mark-Compact(标记整理)相结合

1.2.4.1 Mark-Sweep(标记清除)

- 标记清除分为标记和清除两个阶段
- 在标记阶段需要遍历堆中的所有对象，并标记那些活着的对象，然后进入清除阶段。在清除阶段总，只清除没有被标记的对象
- V8采取的是黑色和白色来标记数据，垃圾收集之前，会把所有的数据设置为白色，用来标记所有的尚未标记的对象，然后会从GC根出发，以深度优先的方式把所有的能访问到的数据都标记为黑色，遍历结束后黑色的就是活的数据，白色的就是可以清理的垃圾数据
- 由于标记清除只清除死亡对象，而死亡对象在老生代中占用的比例很小，所以效率较高
- 标记清除有一个问题就是进行一次标记清楚后，内存空间往往是不连续的，会出现很多的内存碎片。如果后续需要分配一个需要内存空间较多的对象时，如果所有的内存碎片都不够用，就会出现内存溢出的问题

深度优先遍历过程 (https://www.processon.com/diagraming/61b42dce5653bb1c942b7544)

1.2.4.2 Mark-Compact（标记整理）

- 标记整理正是为了解决标记清除所带来的内存碎片的问题
- 标记整理在标记清除的基础进行修改，将其的清除阶段变为紧缩极端
- 在整理的过程中，将活着的对象向内存区的一段移动，移动完成后直接清理掉边界外的内存
- 紧缩过程涉及对象的移动，所以效率并不是太好，但是能保证不会生成内存碎片，一般10次标记清理会伴随一次标记整理

深度优先遍历过程 (https://www.processon.com/diagraming/61b42dce5653bb1c942b7544)

1.2.5 优化

- 在执行垃圾回收算法期间，JS脚本需要暂停，这种叫Stop the world(全停顿)
- 如果回收时间过长，会引起卡顿
- 性能优化
    - 把大任务拆分小任务，分步执行，类似fiber
    - 将一些任务放在后台执行，不占用主线程

```
JavaScript执行  垃圾标记、垃圾清理、垃圾整理  JavaScript执行
-------------                              ---------------->
```

1.2.5.1 Parallel(并行执行)

- 新生代的垃圾回收采取并行策略提升垃圾回收速度，它会开启多个辅助线程来执行新生代的垃圾回收工作
- 并行执行需要的时间等于所有的辅助线程时间的总和加上管理的时间
- 并行执行的时候也是全停顿的状态，主线程不能进行任何操作，只能等待辅助线程的完成
- 这个主要应用于新生代的垃圾回收

```
         -------辅助线程----->
         -------辅助线程----->
         -------辅助线程----->
---------                    ---------------------------->
```

1.2.5.2 增量标记

- 老生代因为对象又大又多，所以垃圾回收的时间更长，采用增量标记的方式进行优化
- 增量标记就是把标记工作分成多个阶段，每个阶段都只标记一部分对象，和主线程的执行穿插进行
- 为了支持增量标记，V8必须可以支持垃圾回收的暂停和恢复，所以采用了 黑白灰三色标记法
    - 黑色表示这个节点被GC根引用到了，而且该节点的子节点都已经标记完成了
    - 灰色表示这个节点被 GC根引用到了，但子节点还没被垃圾回收器标记处理，也表明目前正在处理这个节点
    - 白色表示此节点还没未被垃圾回收器发现，如果在本轮遍历结束时还是白色，那么这块数据就会被收回
- 引入了灰色标记后，就可以通过判断有没有灰色节点来判断标记是否完成了，如果有灰色节点，下次恢复的应该从灰色节点继续执行

增量标记 (https://www.processon.com/diagraming/61b439700e3e740451863fd5)

```
---------开始标记---增量标记---增量标记---清理---整理----------------->
```

1.2.5.3 Write-barrier(写屏障)

- 当黑色指向白色节点的时候，就会触发写屏障，这个写屏障会把白色节点设置为灰色

```
global.a = { name: 'a' };
global.a.b = { name: 'b1' };

global.a.b = { name: 'b2' };
```

Write-barrier(写屏障) (https://www.processon.com/diagraming/61b43d161efad42237b28a09)

1.2.5.4 Lazy Sweeping(惰性清理)

- 当增量标记完成后，如果内存够用，先不清理，等JS代码执行完慢慢清理

1.2.5.5 concurrent(并发回收)

- 其实增量标记和惰性清理并没有减少暂停的总时间
- 并发回收就是主线程在执行过程中，辅助线程可以在后台完成垃圾回收工作
- 标记操作全都由辅助线程完，清理操作由主线程和辅助线程配合完成

```
         ----辅助线程标记---->       -----清理整理---->
         ----辅助线程标记---->       -----清理整理---->
         ----辅助线程标记---->       -----清理整理---->
---------------------------执行JS>-----清理整理--->---------------------------->
```

1.2.5.6 并发(concurrent)和并行(parallel)

- 并发和并行都是同时执行任务
- 并行的 同时是同一时刻可以多个进程在运行
- 并发的 同时是经过上下文快速切换，使得看上去多个进程同时都在运行的现象

2.内存泄露

2.1 什么是内存泄露

- 那当不再用到的对象内存没有及时被回收时，我们叫它内存泄漏

2.2 不合理的闭包

- 闭包 (https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Closures)
- 一个函数和对其周围状态(lexical environment,词法环境)的引用捆绑在一起这样的组合就是闭包（closure）
- 也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域。在 JavaScript 中，每当创建一个函数，闭包就会在函数创建的同时被创建出来
- 词法(lexical)一词指的是，词法作用域根据源代码中声明变量的位置来确定该变量在何处可用。嵌套函数可访问声明于它们外部作用域的变量

```
function Person(){}
function fn(){
    let name = 'zhufeng';
    let age = 13;
    let arr = new Array();
    for(let i=0;i<10000;i++){
        arr.push(new Person());
    }
    return function(){
        console.log('hello',arr,name);
    }
}
let hello = fn();
debugger
hello();
```

2.3 隐式全局变量

- 全局变量通常不会被回收,所以要避免额外的全局变量
- 使用完毕后经重置值为null

```
function Person(){}
function fn(){
    p1 = new Person();
    this.p2 = new Person();
}
fn();
p1=null;
p2=null;
```

2.4 分离的DOM

- 当在界面中移除DOM节点时，还要移除相应的节点引用

```

<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Documenttitle>
head>
<body>
    <div id="container">
        <p id="title">p>
    div>
    <script>
       var container = document.getElementById('container');
       var title = document.getElementById('title');
       document.body.removeChild(container);

       container=null;
       title=null;
    script>
body>
html>
```

2.5 定时器

- setInterval(fn,delay) cleanInterval(id)
- setTimeout(fn,delay) cleanTimeout(id)
- setImmediate(fn) cleanImmediate(id)
- requestAnimationFrame(fn) cleanRequestAnimationFrame(id)

```
<script>
 function Person(){}
 function fn(){
    var p1 = new Person();
    var id = setInterval(()=>{
        p1.age = 20;
    },1000);
    clearInterval(id);
 }
 fn();
script>
```

2.6 事件监听器

- 监听函数如果不及时移除，会导致内存泄漏

```
var data = new Array(100000);
class App extends React.Component{
    componentDidMount(){
        document.addEventListener('click',this.handleClick);
    }
    handleClick=()=>{
        console.log('click',data);
    }
    componentWillUnmount(){
        document.removeEventListener('click',this.handleClick);
    }
}
```

2.7 Map、Set对象

- Map 或 Set 存储对象时如果不主动清除也会造成内存不自动回收
- 可以采用 WeakMap，WeakSet 对象同样用来保存键值对，对于键是弱引用(WeakMap 只对于键是弱引用)，且必须为一个对象，而值可以是任意的对象或者原始值，由于是对于对象的弱引用，不会阻止 垃圾回收

```
function Person(){}
let obj = new Person();
let set = new Set([obj]);
let map = new Map([[obj,'zhufeng']]);
obj = null;
```

```
function Person(){}
let obj = new Person();
let set = new WeakSet([obj]);
let map = new WeakMap([[obj,'zhufeng']]);
obj = null;
```

2.8 console

- 浏览器保存了我们输出对象的信息数据引用
- 未清理的 console 如果输出了对象也会造成内存泄漏

```
function Person(){}
function fn(){
    let name = 'zhufeng';
    let age = 13;
    let arr = new Array();
    for(let i=0;i<10000;i++){
        arr.push(new Person());
    }
    return function(){
        console.log('hello',arr);
    }
}
let hello = fn();
hello();
```

3.内存泄漏排查

- **3.1 发现内存泄漏#**

```
<body>
  <div id="container">0div>
  <button id="click">clickbutton>
  <script>
      var rows = [];
      function Person() { }
      function getColumns() {
          var columns = new Array(10000).fill('0');
          for (let i = 0; i < columns.length; i++) {
              columns[i] = new Person();
          }
          return function () {
              return columns;
          }
      }
      click.addEventListener("click", function () {
          rows.push(getColumns());
          container.innerHTML = rows.length;
      });
  script>
body>
```

** 3.2 定位内存泄漏 #**

3.2.1 录制监控

- 刷新录制页面加载
- 监控堆、文档 、节点、监听器、CPU
- 手工GC

3.2.2 内存快照

字段 含义 摘要 按构造函数进行分组，捕获对象和其使用内存的情况 对比 对比某个操作前后的内存快照区别 控制 查看堆的具体内容，可以用来查看对象结构 统计信息 统计视图

3.2.2.1 摘要

字段 含义 构造函数 显示所有的构造函数，点击每一个构造函数可以查看由该构造函数创建的所有对象 距离 显示通过最短的节点路径到根节点的距离，引用层级 浅层大小 显示对象所占内存，不包含内部引用的其他对象所占的内存 保留的大小 显示对象所占的总内存，包含内部引用的其他对象所占的内存

3.2.2.2 对比

字段 含义 新对象数 新建对象数 已删除项 回收对象数 增量 新建对象数减去回收的对象数

4.性能优化

- https://jsbench.me (https://jsbench.me)

** 4.1 少用全局变量 #**

- 全局执行上下文会一直存在于上下文执行栈中，不会销毁，容易内存泄露

- 查找变量的链条比较长，比较消耗性能
- 容易引起命名冲突
- 确定需要使用的全局就是可以局部缓存

```
var a = 'a';
function one() {
    return function two() {
        return function three() {
            let b = a;
            for (let i = 0; i < 100000; i++) {
                console.log(b);
            }
        }
    }
}
one()()();
```

** 4.2 通过原型新增方法 #**

```
var Person = function () {
    this.getName = function () {
        console.log('person');
    }
}
let p1 = new Person();
```

```
var Person = function () {

}
Person.prototype.getName = function () {
    console.log('person');
}
let p1 = new Person();
```

** 4.3 尽量创建对象一次搞定 #**

- V8会为每个对象分配一个隐藏类,如果对象结构发生改变就会重建隐藏类，结构相同的对象会共且隐藏类
- 隐藏类描述了对象的结构和属性偏移地址，可以加速查找属性的时间
- 优化指南
    - 创建对象尽量保持属性顺序一致
    - 尽量不要动态添加和删除属性

```
d8 --allow-natives-syntax main.js
```

```
let p1 = {name:'zhangsan',age:10}
let p2 = {age:10,name:'zhangsan'}

let p = {};
%DebugPrint(point);
p.name = 'wangwu';
%DebugPrint(point);
p.age = 10;
%DebugPrint(point);
```

** 4.4 尽量保持参数结构稳定 #**

- V8中的内联缓存会监听函数执行，记录中间数据，参数结构不同会让优化失效

```
function read(obj){
 console.log(obj.toString());
}
for(let i=0;i<1000;i++){
    read(i)
}

function read(obj){
 console.log(obj.toString());
}
for(let i=0;i<1000;i++){
    read(i%2===0?i:''+i)
}
```

```
function read(obj){
 console.log(obj.toString());
}
function read2(obj){
 console.log(obj.toString());
}
for(let i=0;i<1000;i++){
    read(i%2===0?i:''+i)
    i%2===0?read(i):read2(''+i)
}
```