JavaScript内存优化

相对C/C++ 而言，我们所用的JavaScript 在内存这一方面的处理已经让我们在开发中更注重业务逻辑的编写。但是随着业务的不断复杂化，单页面应用、移动HTML5 应用和Node.js 程序等等的发展，JavaScript 中的内存问题所导致的卡顿、内存溢出等现象也变得不再陌生。

1. 语言层面的内存管理

1.1 作用域

作用域(scope)是JavaScript 编程中一个非常重要的运行机制，在同步JavaScript 编程中它并不能充分引起初学者的注意，但在异步编程中，良好的作用域控制技能成为了JavaScript 开发者的必备技能。另外，作用域在JavaScript 内存管理中起着至关重要的作用。

在JavaScript中，能形成作用域的有函数的调用、`with`语句和全局作用域。如以下代码为例：

```
var foo = function() {
  var local = {};
};
foo();
console.log(local); //=> undefined

var bar = function() {
  local = {};
};
bar();
console.log(local); //=> {}
```

这里我们定义了`foo()`函数和`bar()`函数，他们的意图都是为了定义一个名为`local`的变量。但最终的结果却截然不同。

在foo()函数中，我们使用var语句来声明定义了一个local变量，而因为函数体内部会形成一个作用域，所以这个变量便被定义到该作用域中。而且foo()函数体内并没有做任何作用域延伸的处理，所以在该函数执行完毕后，这个local变量也随之被销毁。而在外层作用域中则无法访问到该变量。
而在bar()函数内，local变量并没有使用var语句进行声明，取而代之的是直接把local作为全局变量来定义。故外层作用域可以访问到这个变量。

```
local = {};
// 这里的定义等效于
global.local = {};
```

1.2 作用域链

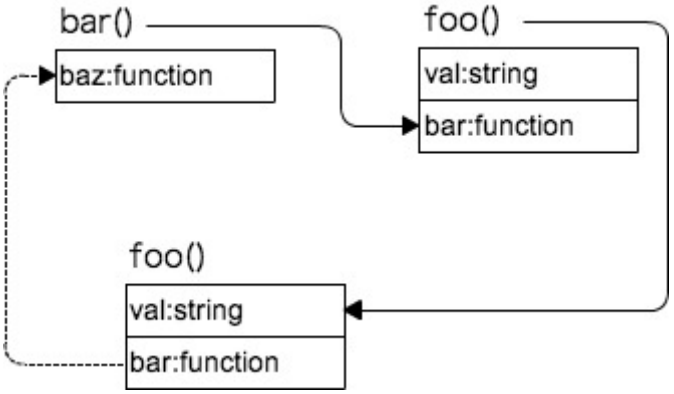
在JavaScript编程中，你一定会遇到多层函数嵌套的场景，这就是典型的作用域链的表示。 如以下代码所示：

```
function foo() {
  var val = 'hello';

  function bar() {
    function baz() {
      global.val = 'world;'
    }
    baz();
    console.log(val); //=> hello
  }
  bar();
}
foo();
```

根据前面关于作用域的阐述，你可能会认为这里的代码所显示的结果是world，但实际的结果却是hello。很多初学者在这里就会开始感到困惑了，那么我们再来看看这段代码是怎么工作的。

由于JavaScript 中，变量标识符的查找是从当前作用域开始向外查找，直到全局作用域为止。所以JavaScript 代码中对变量的访问只能向外进行，而不能逆而行之。



`baz()`函数的执行在全局作用域中定义了一个全局变量`val`。而在`bar()`函数中，对`val`这一标识符进行访问时，按照从内到外厄德查找原则：在`bar`函数的作用域中没有找到，便到上一层，即`foo()`函数的作用域中查找。

然而，使大家产生疑惑的关键就在这里：本次标识符访问在`foo()`函数的作用域中找到了符合的变量，便不会继续向外查找，故在`baz()`函数中定义的全局变量`val`并没有在本次变量访问中产生影响。

1.3 闭包

我们知道JavaScript 中的标识符查找遵循从内到外的原则。但随着业务逻辑的复杂化，单一的传递顺序已经远远不能满足日益增多的新需求。

我们先来看看下面的代码：

```
function foo() {
  var local = 'Hello';
  return function() {
    return local;
  };
}
```

```
var bar = foo();
console.log(bar()); //=> Hello
```

这里所展示的让外层作用域访问内层作用域的技术便是闭包(Closure)。得益于高阶函数的应用，使`foo()`函数的作用域得到『延伸』。

`foo()`函数返回了一个匿名函数，该函数存在于`foo()`函数的作用域内，所以可以访问到`foo()`函数作用域内的`local`变量，并保存其引用。而因这个函数直接返回了`local`变量，所以在外层作用域中便可直接执行`bar()`函数以获得`local`变量。

闭包是JavaScript 的高级特性，我们可以借助它来实现更多更复杂的效果来满足不同的需求。但是要注意的是因为把带有内部变量引用的函数带出了函数外部，所以该作用域内的变量在函数执行完毕后的并不一定会被销毁，直到内部变量的引用被全部解除。所以闭包的应用很容易造成内存无法释放的情况。

2. JavaScript 的内存回收机制

这里我将以Chrome 和Node.js 所使用的，由Google 推出的V8 引擎为例，简要介绍一下JavaScript 的内存回收机制，更详尽的内容可以购买我的好朋友朴灵的书《深入浅出Node.js 》进行学习，其中『内存控制』一章中有相当详细的介绍。

在V8 中，所有的JavaScript 对象都是通过『堆』来进行内存分配的。

当我们在代码中声明变量并赋值时，V8 就会在堆内存中分配一部分给这个变量。如果已申请的内存不足以存储这个变量时，V8 就会继续申请内存，直到堆的大小达到了V8 的内存上限为

止。默认情况下，V8 的堆内存的大小上限在64位系统中为1464MB，在32位系统中则为732MB，即约1.4GB 和0.7GB。
另外，V8 对堆内存中的JavaScript 对象进行分代管理：新生代和老生代。新生代即存活周期较短的JavaScript 对象，如临时变量、字符串等；而老生代则为经过多次垃圾回收仍然存活，存活周期较长的对象，如主控制器、服务器对象等。
垃圾回收算法一直是编程语言的研发中是否重要的一环，而V8中所使用的垃圾回收算法主要有以下几种：

1. Scavange 算法：通过复制的方式进行内存空间管理，主要用于新生代的内存空间；
2. Mark-Sweep 算法和Mark-Compact 算法：通过标记来对堆内存进行整理和回收，主要用于老生代对象的检查和回收。

PS: 更详细的V8 垃圾回收实现可以通过阅读相关书籍、文档和源代码进行学习。
我们再来看看JavaScript 引擎在什么情况下会对哪些对象进行回收。

2.1 作用域与引用

初学者常常会误认为当函数执行完毕时，在函数内部所声明的对象就会被销毁。但实际上这样理解并不严谨和全面，很容易被其导致混淆。
引用(Reference)是JavaScript 编程中十分重要的一个机制，但奇怪的是一般的开发者都不会刻意注意它、甚至不了解它。引用是指『代码对对象的访问』这一抽象关系，它与C/C++ 的指针

有点相似，但并非同物。引用同时也是JavaScript 引擎在进行垃圾回收中最关键的一个机制。

一下面代码为例：

```
// ......
var val = 'hello world';
function foo() {
  return function() {
    return val;
  };
}
global.bar = foo();
// ......
```

阅读完这段代码，你能否说出这部分代码在执行过后，有哪些对象是依然存活的么？

根据相关原则，这段代码中没有被回收释放的对象有`val`和`bar()`，究竟是什么原因使他们无法被回收？

JavaScript 引擎是如何进行垃圾回收的？前面说到的垃圾回收算法只是用在回收时的，那么它是如何知道哪些对象可以被回收，哪些对象需要继续生存呢？答案就是JavaScript 对象的引用。

JavaScript 代码中，哪怕是简单的写下一个变量名称作为单独一行而不做任何操作，JavaScript 引擎都会认为这是对对象的访问行为，存在了对对象的引用。为了保证垃圾回收的行为不影响程序逻辑的运行，JavaScript 引擎就决不能把正在使用的对象进行回收，不然就乱套了。所以判断对象是否正在使用中的标准，就是是否仍然存在对该对象的引用。但事实上，这是一种妥协的做法，因为JavaScript 的引用是可以进行转移的，那么就有可能出现某些引用被带到了全局作用域，但事实上在业务逻辑里已经不需要对其进行访问了，应该被回收，但是JavaScript 引擎仍会死板地认为程序仍然需要它。

如何用正确的姿势使用变量、引用，正是从语言层面优化JavaScript 的关键所在。

3. 优化你的JavaScript

终于进入正题了，非常感谢你秉着耐心看到了这里，经过上面这么多介绍，相信你已经对JavaScript 的内存管理机制有了不错的理解，那么下面的技巧将会让你如虎添翼。

3.1 善用函数

如果你有阅读优秀JavaScript 项目的习惯的话，你会发现，很多大牛在开发前端JavaScript 代码的时候，常常会使用一个匿名函数在代码的最外层进行包裹。

```
;(function() {
  // 主业务代码
})();
```

有的甚至更高级一点：

```
;(function(win, doc, $, undefined) {
  // 主业务代码
})(window, document, jQuery);
```

甚至连如RequireJS, SeaJS, OzJS 等前端模块化加载解决方案，都是采用类似的形式：

```
// RequireJS
define(['jquery'], function($) {
  // 主业务代码
});

// SeaJS
define('module', ['dep', 'underscore'], function($, _) {
  // 主业务代码
});
```

如果你说很多Node.js 开源项目的代码都没有这样处理的话，那你就错了。Node.js 在实际运行代码之前，会把每一个.js 文件进

行包装，变成如下的形式：

```
(function(exports, require, module, __dirname, __filename) {
  // 主业务代码
});
```

这样做有什么好处？我们都知道文章开始的时候就说了，JavaScript中能形成作用域的有函数的调用、`with`语句和全局作用域。而我们也知道，被定义在全局作用域的对象，很有可能是会一直存活到进程退出的，如果是一个很大的对象，那就麻烦了。比如有的人喜欢在JavaScript中做模版渲染：

```
<?php
  $db = mysqli_connect(server, user, password, 'myapp');
  $topics = mysqli_query($db, "SELECT * FROM topics;");
?>
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>你是猴子请来的逗比么？</title>
</head>
<body>
  <ul id="topics"></ul>
  <script type="text/tmpl" id="topic-tmpl">
    <li class="topic">
      <h1><%=title%></h1>
      <p><%=content%></p>
    </li>
  </script>
  <script type="text/javascript">
    var data = <?php echo json_encode($topics); ?>;
    var topicTmpl = document.querySelector('#topic-tmpl').innerHTML;
    var render = function(tmlp, view) {
      var complied = tmlp
        .replace(/\n/g, '\\n')
        .replace(/<%=([\s\S]+?)%>/g, function(match, code) {
          return '" + escape(' + code + ') + "';
        });
```

```
      complied = [
        'var res = "";',
        'with (view || {}) {',
          'res = "' + complied + '";',
        '}',
        'return res;'
      ].join('\n');

      var fn = new Function('view', complied);
      return fn(view);
    };

    var topics = document.querySelector('#topics');
    function init()
      data.forEach(function(topic) {
        topics.innerHTML += render(topicTmpl, topic);
      });
    }
    init();
  </script>
</body>
</html>
```

这种代码在新手的作品中经常能看得到，这里存在什么问题呢？如果在从数据库中获取到的数据的量是非常大的话，前端完成模板渲染以后，`data`变量便被闲置在一边。可因为这个变量是被定义在全局作用域中的，所以JavaScript引擎不会将其回收销毁。如此该变量就会一直存在于老生代堆内存中，直到页面被关闭。可是如果我们作出一些很简单的修改，在逻辑代码外包装一层函数，这样效果就大不同了。当UI渲染完成之后，代码对`data`的引用也就随之解除，而在最外层函数执行完毕时，JavaScript引擎就开始对其中的对象进行检查，`data`也就可以随之被回收。

3.2 绝对不要定义全局变量

我们刚才也谈到了，当一个变量被定义在全局作用域中，默认情况下JavaScript 引擎就不会将其回收销毁。如此该变量就会一直存在于老生代堆内存中，直到页面被关闭。
那么我们就一直遵循一个原则：绝对不要使用全局变量。虽然全局变量在开发中确实很省事，但是全局变量所导致的问题远比其所带来的方便更严重。

1. 使变量不易被回收；
2. 多人协作时容易产生混淆；
3. 在作用域链中容易被干扰。

配合上面的包装函数，我们也可以通过包装函数来处理『全局变量』。

3.3 手工解除变量引用

如果在业务代码中，一个变量已经确切是不再需要了，那么就可以手工解除变量引用，以使其被回收。

```
var data = { /* some big data */ };
// blah blah blah
data = null;
```

3.4 善用回调

除了使用闭包进行内部变量访问，我们还可以使用现在十分流行的回调函数来进行业务处理。

```
function getData(callback) {
  var data = 'some big data';

  callback(null, data);
}

getData(function(err, data) {
```

```
  console.log(data);
});
```

回调函数是一种后续传递风格(Continuation Passing Style, CPS)的技术，这种风格的程序编写将函数的业务重点从返回值转移到回调函数中去。而且其相比闭包的好处也不少：

1. 如果传入的参数是基础类型（如字符串、数值），回调函数中传入的形参就会是复制值，业务代码使用完毕以后，更容易被回收；
2. 通过回调，我们除了可以完成同步的请求外，还可以用在异步编程中，这也就是现在非常流行的一种编写风格；
3. 回调函数自身通常也是临时的匿名函数，一旦请求函数执行完毕，回调函数自身的引用就会被解除，自身也得到回收。

3.5 良好的闭包管理

当我们的业务需求(如循环事件绑定、私有属性、含参回调等)一定要使用闭包时，请谨慎对待其中的细节。

循环绑定事件可谓是JavaScript 闭包入门的必修课，我们假设一个场景：有六个按钮，分别对应六种事件，当用户点击按钮时，在指定的地方输出相应的事件。

```
var btns = document.querySelectorAll('.btn'); // 6 elements
var output = document.querySelector('#output');
var events = [1, 2, 3, 4, 5, 6];

// Case 1
for (var i = 0; i < btns.length; i++) {
  btns[i].onclick = function(evt) {
    output.innerText += 'Clicked ' + events[i];
  };
}
```

```
// Case 2
for (var i = 0; i < btns.length; i++) {
  btns[i].onclick = (function(index) {
    return function(evt) {
      output.innerText += 'Clicked ' + events[index];
    };
  })(i);
}

// Case 3
for (var i = 0; i < btns.length; i++) {
  btns[i].onclick = (function(event) {
    return function(evt) {
      output.innerText += 'Clicked ' + event;
    };
  })(events[i]);
}
```

这里第一个解决方案显然是典型的循环绑定事件错误，这里不细说，详细可以参照我给一个网友的回答；而第二和第三个方案的区别就在于闭包传入的参数。

第二个方案传入的参数是当前循环下标，而后者是直接传入相应的事件对象。事实上，后者更适合在大量数据应用的时候，因为在JavaScript的函数式编程中，函数调用时传入的参数是基本类型对象，那么在函数体内得到的形参会是一个复制值，这样这个值就被当作一个局部变量定义在函数体的作用域内，在完成事件绑定之后就可以对`events`变量进行手工解除引用，以减轻外层作用域中的内存占用了。而且当某个元素被删除时，相应的事件监听函数、事件对象、闭包函数也随之被销毁回收。

3.6 内存不是缓存

缓存在业务开发中的作用举足轻重，可以减轻时空资源的负担。但需要注意的是，不要轻易将内存当作缓存使用。内存对于任何

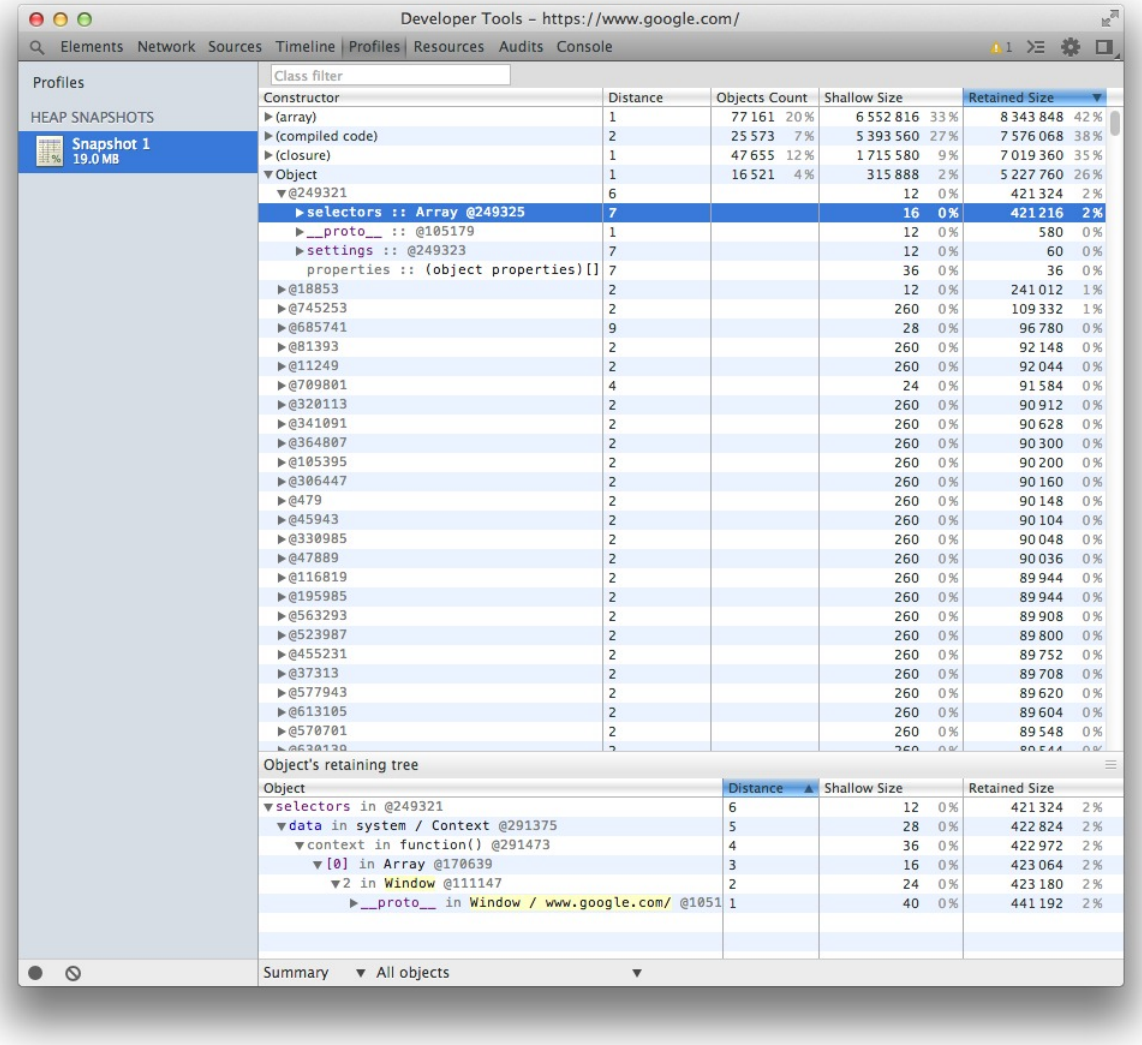
程序开发来说都是寸土寸金的东西，如果不是很重要的资源，请不要直接放在内存中，或者制定过期机制，自动销毁过期缓存。

4. 检查JavaScript 的内存使用情况

在平时的开发中，我们也可以借助一些工具来对JavaScript 中内存使用情况进行分析和问题排查。

4.1 Blink / Webkit 浏览器

在Blink / Webkit 浏览器中（Chrome, Safari, Opera etc.），我们可以借助其中的Developer Tools 的Profiles 工具来对我们的程序进行内存检查。



4.2 Node.js 中的内存检查

在Node.js 中，我们可以使用node-heapdump 和node-memwatch 模块进行内存检查。

```
var heapdump = require('heapdump');
var fs = require('fs');
var path = require('path');
fs.writeFileSync(path.join(__dirname, 'app.pid'), process.pid);
// ...
```

在业务代码中引入node-heapdump 之后，我们需要在某个运行时期，向Node.js 进程发送SIGUSR2 信号，让node-heapdump 抓拍一份堆内存的快照。

```
$ kill -USR2 (cat app.pid)
```

这样在文件目录下会有一个以`heapdump-<sec>.<usec>.heapsnapshot`格式命名的快照文件，我们可以使用浏览器的Developer Tools中的Profiles工具将其打开，并进行检查。

5. 小结

很快又来到了文章的结束，这篇分享主要向大家展示了以下几点内容：

1. JavaScript 在语言层面上，与内存使用息息相关的东西；
2. JavaScript 中的内存管理、回收机制；
3. 如何更高效地使用内存，以至于让出产的JavaScript 能更有拓展的活力；
4. 如何在遇到内存问题的时候，进行内存检查。