JavScript 性能优化

选择最优的循环方法

文档碎片优化节点添加

克隆优化节点操作

直接量替换new Object

课程概述

- 内存管理
- 垃圾回收与常见GC算法
- V8引擎的垃圾回收
- Performance工具
- 代码优化实例

JS内存管理

- 内存：由可读写单元组成，表示一片可操作空间
- 管理：人为的去操作一片空间的申请、使用和释放
- 内存管理：开发者主动申请空间、使用空间、释放空间
- 管理流程：申请--使用--释放

```
// 申请
let obj = {}
// 使用
obj.name = 'wubo';
// 释放
obj = null;
```

JS中的垃圾回收

1. JS中的垃圾：
    - JS中内存管理是自动的
    - 对象不再被引用时是垃圾
    - 对象不能从跟上访问到时是垃圾
2. JS中的可达对象：
    - 可以访问到的对象就是可达对象（引用、作用域链）
    - 可达的标准就是从根出发是否能够被找到
    - JS中的根就可以理解为是全局变量对象

```
function generateObject(obj1, obj2) {
  obj1.next = obj2;
  obj2.prev = obj1;

  return {
    o1: obj1,
    o2: obj2
  }
}

const obj = generateObject({ name: 'obj1' }, { name: 'obj2' });

// 这个时候，我们就无法在访问到对象{ name: 'obj1' }, 它就会被当做“垃圾”
delete obj.o1;
delete obj.o2.prev;
```

GC算法介绍

- GC是一种机制，垃圾回收器完成具体的工作
- 工作的内容就是查找垃圾，释放空间、回收空间
- 算法就是工作时查找和回收所遵循的规则

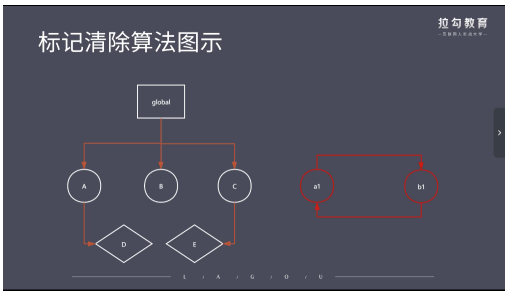
引用计数算法

1. 原理
   - 核心思想：设置引用数，判断当前引用数是否为0
   - 引用计数器
   - 引用关系改变时修改引用数数字
   - 引用数字为0时立即回收
2. 优点
   - 发现垃圾时立即回收
   - 最大限度减少程序暂停，因为一旦内存不够就能够马上去回收垃圾对象
3. 缺点
   - 无法回收循环引用的对象。（obj1.name = obj2；obj2.name = obj1;）
   - 时间开销大
   - 资源消耗大

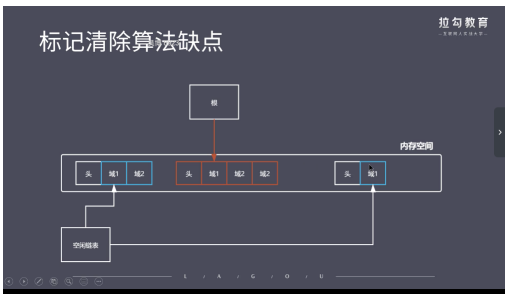
标记清除算法

1. 原理
   - 核心思想：分标记和清除二个阶段完成
   - 遍历所有对象找标记活动对象（从global出发）

- 遍历所有对象清除没有标记对象
- 回收相应的空间

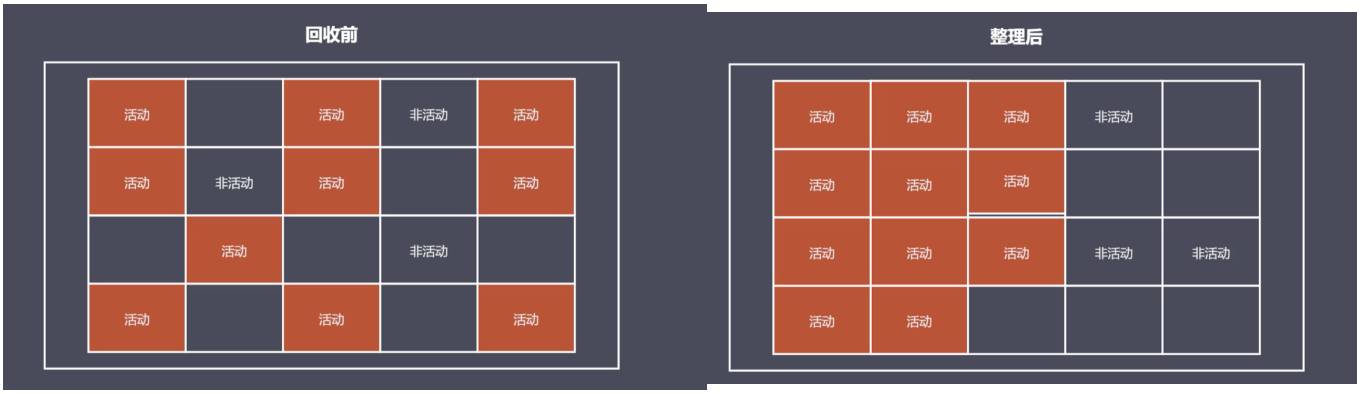


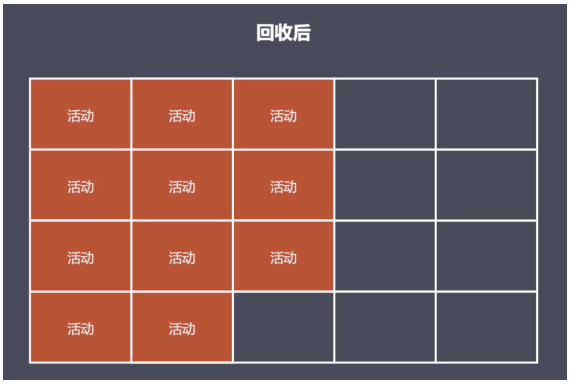
2. 优点
   - 可以解决对象循环引用的操作带来的回收问题（相对于引用计数算法来说）
3. 缺点
   - 造成内存空间碎片化，它们不连续。
   - 不会立即回收垃圾对象



标记整理算法

1. 原理
   - 标记整理可以看做是标记清除的增强
   - 标记阶段的操作和标记清除一致
   - 清除阶段会先执行整理，移动对象位置，让内存地址空间连续





2. 优点：
    - 减少碎片化空间
3. 缺点：
    - 不会立即回收垃圾对象

认识V8

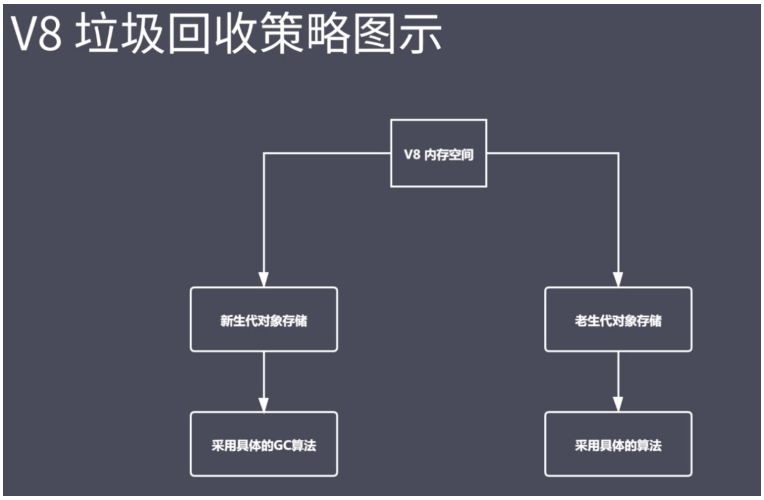
- V8是一款主流的JS执行引擎（很快）
- V8采用即时编译
- V8内存设限（64位 不超过1.5G；32位 不超过800MB）

V8垃圾回收策略

- 采用分代回收的思想
- 内存分为新生代、老生代
- 针对不同对象采用不同算法

V8中常用GC算法

- 分代回收
- 空间复制
- 标记清除
- 标记整理
- 标记增量



V8内存分配

- V8内存空间一分为二
- 小空间用于存储新生代对象（32M | 16M）
- 新生代指的是存活时间较短的对象

V8如何回收新生代对象

新生代对象回收实现

- 回收过程采用复制算法 + 标记整理
- 新生代内存区分为二个等大小空间
- 使用空间为From，空闲空间为To
- 活动对象存储于From空间
- 标记整理后将活动对象拷贝至To

回收细节说明

- 拷贝过程中可能出现晋升
- 晋升就是将新生代对象移动至老生代
- 一轮GC还存活的新生代需要晋升
- To空间的使用率超过25%

V8如何回收老生代对象

- 老年代对象存放在右侧老生代区域

- 64位操作系统1.4G，32位操作系统700M
- 老年代对象就是指存活时间较长的对象

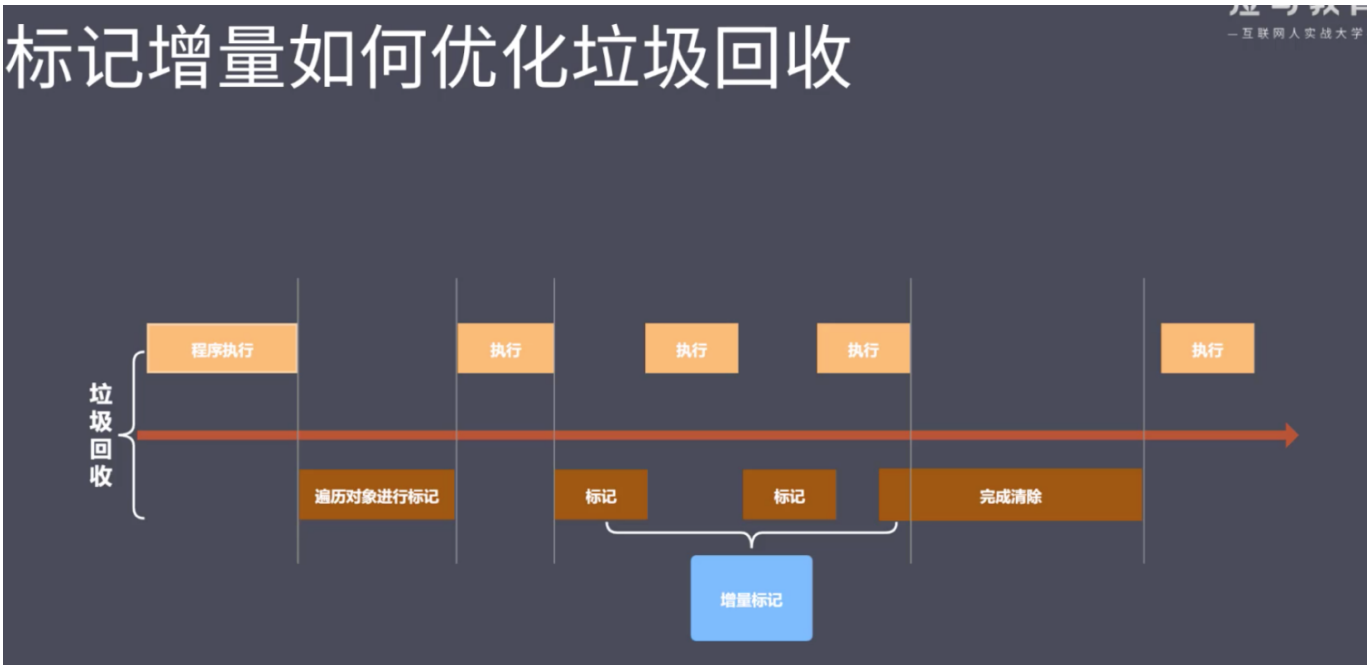
老年代对象回收实现

- 主要采用标记清除、标记整理、增量标记算法
- 首先使用标记清除完成垃圾空间的回收
- 采用标记整理进行空间优化（当新生代对象要晋升移入到老生代空间内时，就做这个整理）
- 采用增量标记进行效率优化

细节对比

- 新生代区域垃圾回收使用空间换时间
- 老生代区域垃圾回收不适合复制算法（一分为二的话，几百M空间不用很浪费，而且老生代多是存在时间就的对象）

看下图，执行和标记是交替执行的，取代了以前一次性做完垃圾回收



V8垃圾回收总结

V8总结

- V8是一款主流的JS执行引擎
- V8内存设置上限（为Web使用，这么点内存够用；为了让内存回收不至于很久，让用户感知不好）
- V8采用基于分代回收思想实现垃圾回收
- V8内存分为新生代和老生代
- V8垃圾回收常见的GC算法

Performance工具介绍

为什么要使用Performance

- GC的目的是为了实现内存空间的良性循环

- 良性循环的基石头是合理使用
- 时刻关注才能确定是否合理
- Performance提供多种监控方式

内存问题的体现

- 页面出现延迟加载或经常性暂停（频繁垃圾回收：突然的内存时间增加）
- 页面持续性出现糟糕的性能（内存膨胀：使用的空间超过了可提供的）
- 页面的性能随时间延长越来越差（内存泄漏）

监控内存的几种方式

界定内存问题的标准

- 内存泄漏：内存使用持续升高
- 内存膨胀：在多数设备上都存在性能问题
- 频繁垃圾回收：通过内存变化图进行分析

监控内存的几种方式

- 浏览器任务管理器
- Timeline时序图记录
- 堆快照查找分离DOM（内存泄漏）
- 判断是否存在频繁的垃圾回收

任务管理器监控内存

Chrome中的更多工具 –> 任务管理器，调出JavaScript内存，可以看到当前页面JS所占用内存情况

Timeline记录内存

可以定位内存问题出现在那个地方，主要讲解的是Chrome的Performance记录后，可以拖动timeline，页面上可以展示具体的变化，从而定位到对应的代码

判断是否存在频繁GC

为什么确定频繁垃圾回收

- GC工作时应用程序是停止的
- 频繁且过长的GC会导致应用假死
- 用户使用中感知应用卡顿

确定频繁的垃圾回收

- Timeline中频繁的上升下降
- 任务管理器中数据频繁的增加减小

堆快照查找分离DOM

什么是分离DOM

- 界面元素存活在DOM树上
- 垃圾对象时的DOM节点
- 分离状态的DOM节点

那些创建了，但是却没有实际放入到页面中的DOM元素，我们需要对其进行置空null

Performance总结

- Performance使用流程
- 内存问题的相关分析
- Performance时序图监控内存变化
- 任务管理器监控内存变化
- 堆快照查找分离DOM

代码优化介绍

1. 为什么要优化代码
2. 如何精准测试JavaScript性能
- 本质上就是采集大量的执行样本进行数学统计和分析
- 使用基于Benchmark.js的https://jsperf.com/完成
3. Jsperf使用流程
- 使用GitHub账号登录
- 填写个人信息（非必须）
- 填写详细的测试用例信息（title、slug）
- 填写准备代码（DOM操作时经常使用）
- 填写必要的setup(前置)与teardown代码(销毁)
- 填写测试代码片段

慎用全局变量

- 全局变量定义在全局执行上下文，是所有作用域链的顶端。
- 全局执行上下文一直存在与上下文执行栈，直到程序退出。给GC造成压力。
- 如果某个局部作用域出现了同名变量则会遮蔽或污染全局。

缓存全局变量

将使用中无法避免的全局变量缓存到局部，避免多次去访问，比如那些DOM操作的变量

```
// Bad
function getBtn() {
  let oBtn1 = document.getElementById('btn1');
  let oBtn3 = document.getElementById('btn3');
  let oBtn5 = document.getElementById('btn5');
  let oBtn7 = document.getElementById('btn7');
  let oBtn9 = document.getElementById('btn9');
}

```

```
// God
function getBtn() {
  let obj = document;
  let oBtn2 = obj.getElementById('btn2');
  let oBtn4 = obj.getElementById('btn4');
  let oBtn6 = obj.getElementById('btn6');
  let oBtn8 = obj.getElementById('btn8');
  let oBtn10 = obj.getElementById('btn10');
}
```

通过原型对象添加附加方法

```
// Bad
let fn1 = function () {
  this.foo = function () {
    console.log(1111);
  }
}

let f1 = new fn1();

// Good
let fn2 = function () {
}
fn2.prototype.foo = function () {
  console.log(1111);
}

let f2 = new fn2();
```

避开闭包陷阱

关于闭包

- 闭包是一种强大的语法
- 闭包使用不当很容易出现内存泄漏
- 不要为了闭包而闭包

```
// Bad
```

```
// 这里的btn元素被el引用着，el的click时间内使用了el形成了闭包，即使btn元素在页面中被删除了，
// 但是el的引用依然存在，所以这段空间就是浪费了，一旦这种代码过多就很容易产生内存泄漏
function foo() {
  var el = document.getElementById('btn');

  el.onclick = function() {
    console.log(el.id);
  }
}

//God
// 解决方式就是，我们主动将el对于btn元素的引用去除。
function foo() {
  var el = document.getElementById('btn');

  el.onclick = function() {
    console.log(el.id);
  }

  el = null;
}
```

避免属性访问方法使用

JavaScript中的面向对象

- JS不需属性的访问方法，所有属性都是外部可见的
- 使用属性访问方法只会增加一层重定义，没有访问的控制力

```
// Bad
function Person() {
  this.name = 'person';
  this.age = 18;
  this.getAge = function() {
    return this.age;
  }
}

```

```
const p1 = new Person();
const a = p1.getAge();

// Good
function Person() {
  this.name = 'person';
  this.age = 18;
  this.getAge = function() {
    return this.age;
  }
}
const p2 = new Person();
const a = p2.age;
```

For循环优化

可以理解为提前存储长度，而不是每次循环都去获取长度的值

```
const arrList = [];
arrList[10000] = 'icoder'

// Bad
for (var i = 0; i < arrList.length; i++) {
  console.log(arrList[i]);
}
// Good
for (var i = arrList.length - 1;i >= 0; i++) {
  console.log(arrList[i]);
}
```

选择最优的循环方法

如果只是简单的循环遍历一个数组，性能顺序为: forEach > for > for...in

文档碎片优化节点添加

节点的添加操作必然会有回流和重绘。

```
// Bad
for (var i = 0; i < 10; i++) {
```

```
  var oP = document.createElement('p');
  oP.innerHTML = i;
  document.body.appendChild(oP);
}
// Good
const fragEle = document.createDocumentFragment();
for (var i = 0; i < 10; i++) {
  var oP = document.createElement('p');
  oP.innerHTML = i;
  fragEle.appendChild(oP);
}
document.appendChild(fragEle);
```

克隆优化节点操作

```
// Bad
for (var i = 0; i < 3; i++) {
  var oP = document.createElement('p');
  oP.innerHTML = i;
  document.body.appendChild(oP);
}
// God
var oldP = document.getElementById('box1');
for (var i = 0; i < 3; i++) {
  var newP = oldP.cloneNode(false);
  newP.innerHTML = i;
  document.body.appendChild(newP);
}
```

直接量替换new Object

```
// Bad
var a1 = new Array(3);
a1[0] = 1;
a1[1] = 2;
a1[2] = 3;

```

```
// Good
var a = [1, 2, 3];
```