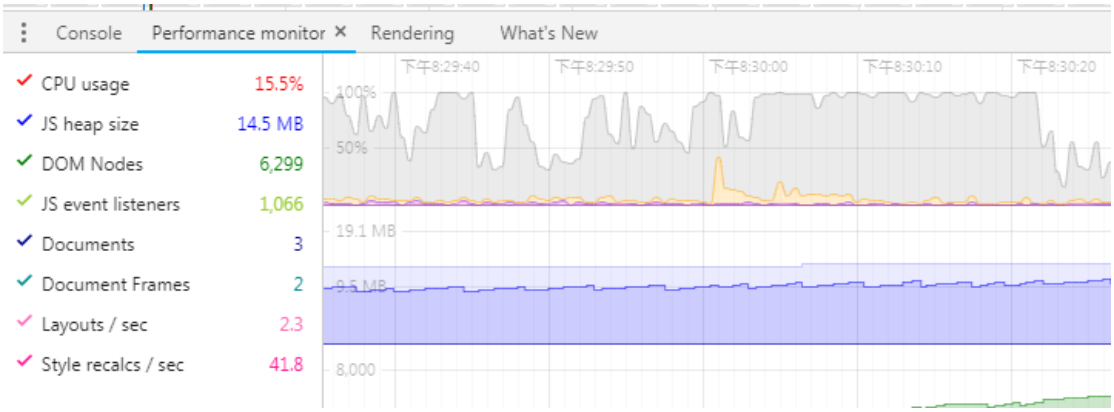
奇秀直播间性能现状浅析（windows）

一、CPU 篇

1. 关于检测工具

本次 CPU 数据分析工具用到了以下几类：

1. performance monitor



CPU usage – 当前站点的 CPU 使用量

JS heap size – 应用的内存占用量

DOM Nodes – 内存中的 dom 节点数目

JS event listeners – 当前页面上注册的时间监听数目

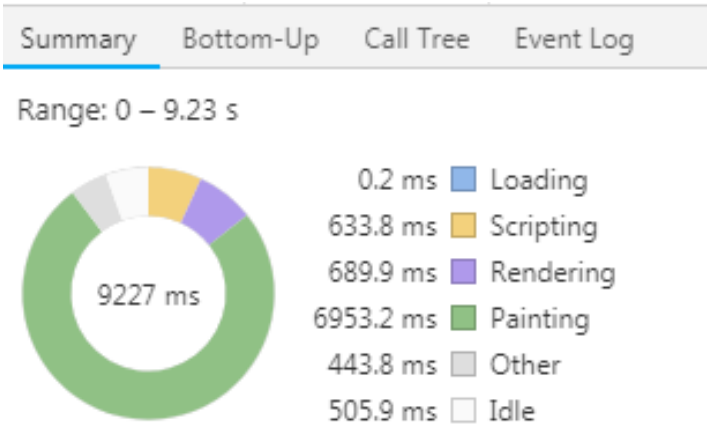
Documents – 当前页面使用的样式或者脚本文件数目

Frames – 当前页面上的 Frames 数目，包括 iframe 与 workers

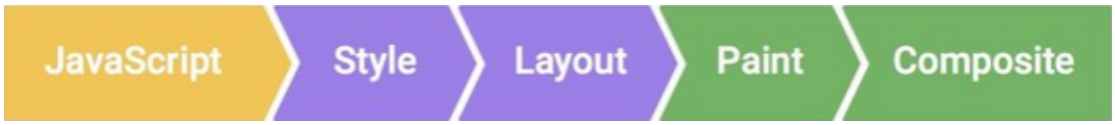
Layouts/sec – 每秒的 DOM 重布局数目

Style recalcs/sec – 浏览器需要重新计算样式的频次

2. performance profile



Summary 面板可以清晰的展示出在监控时间段内，各动作执行或计算所占时长。对于一个页面的展示而言，简单来说可以认为是经历了以下几个步骤



Jascript– 一般来说，我们会使用 JS 来实现一些视觉化的效果。比如做一个动画或者网页面里添加一些 DOM 元素等

Style– 计算样式，这个过程是根据 CSS 选择器，对每一个 DOM 元素匹配对应的 CSS 样式。这一步结束之后，就确定了每个 DOM 元素的 CSS 样式规则

Layout– 布局，上一步确定了每个 DOM 元素的样式规则，这一步是具体计算每个 DOM 元素最终在屏幕上显示的大小和位置。Web 页面中元素的布局是相对的，因此一个元素的布局发生变化，可能会联动引发其他元素的布局发生变化。比如<body>元素的宽度发生变化，起子元素的宽度也会随之发生变化。对浏览器来说，布局过程是经常发生的

Paint– 绘制，本质上就是填充像素的过程，包括文字绘制，颜色，边框，阴影等，也就是一个 DOM 元素的所有可视效果。一般来说，这个过程是在多个层上完成的

Composite– 渲染层合并，由上一步可以知道，对页面中 DOM 元素的绘制是在多个层上进行的，在每个层上完成绘制过程中之后，浏览器会将所有的层按照合理的顺序合并成一个图层，然后显示在屏幕上。对于有位置重叠的元素页面，这个过程尤其重要，因为一旦图层的合并顺序出错，将会导致元素显示异常

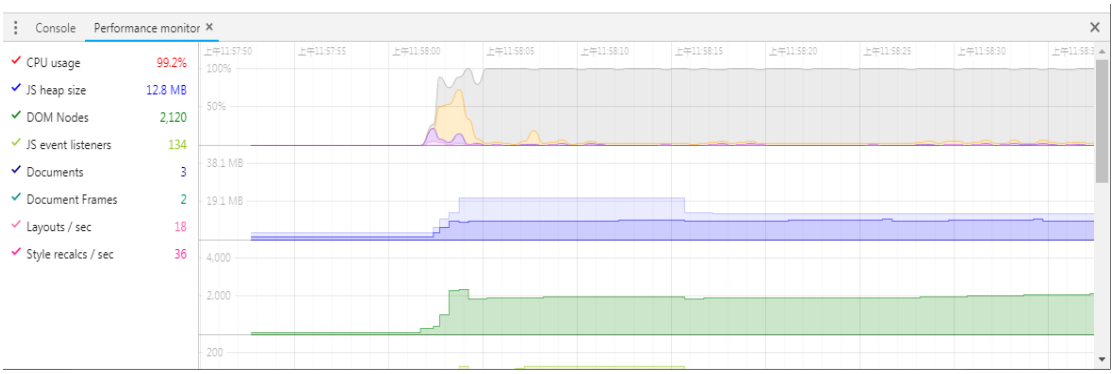
对于一个页面而言，大部分的行为执行和数据计算都是需要 CPU 处理的，所以上图中 Summary 面板可以清晰的展示出在监控的 9.23s 内各项所消耗的时长。也就是说在这一时间段内，各种行为所消耗的 CPU 情况跟起所占时长成正比。

颜色	事件	描述
蓝色(Loading)	Parse HTML	浏览器执行HTML解析
	Finish Loading	网络请求完毕事件
	Receive Data	请求的响应数据到达事件，如果响应数据很大（拆包），可能会多次触发该事件
	Receive Response	响应头报文到达时触发
	Send Request	发送网络请求时触发
紫色(Rendering)	Invalidate layout	当DOM更改导致页面布局失效时触发
	Layout	页面布局计算执行时触发
	Recalculate style	Chrome重新计算元素样式时触发
	Scroll	内嵌的视窗滚动时触发
	Composite Layers	Chrome的渲染引擎完成图片层合并时触发
黄色(Scripting)	Animation Frame Fired	一个定义好的动画帧发生并开始回调处理时触发
	Cancel Animation Frame	取消一个动画帧时触发
	GC Event	垃圾回收时触发
	DOMContentLoaded	当页面中的DOM内容加载并解析完毕时触发
	Evaluate Script	A script was evaluated.
	Event	js事件
	Function Call	只有当浏览器进入到js引擎中时触发
	Install Timer	创建计时器（调用setTimeout()和setInterval()）时触发
	Request Animation Frame	requestAnimationFrame（）调用预定一个新帧
	Remove Timer	当清除一个计时器时触发
	Time	调用console.time()触发
	Time End	调用console.timeEnd()触发
	Timer Fired	定时器激活回调后触发
	XHR Ready State Change	当一个异步请求为就绪状态后触发
	XHR Load	当一个异步请求完成加载后触发
绿色(Painting)	Composite Layers	Chrome的渲染引擎完成图片层合并时触发
	Image Decode	一个图片资源完成解码后触发
	Image Resize	一个图片被修改尺寸后触发
	Paint	合并后的层被绘制到对应显示区域后触发

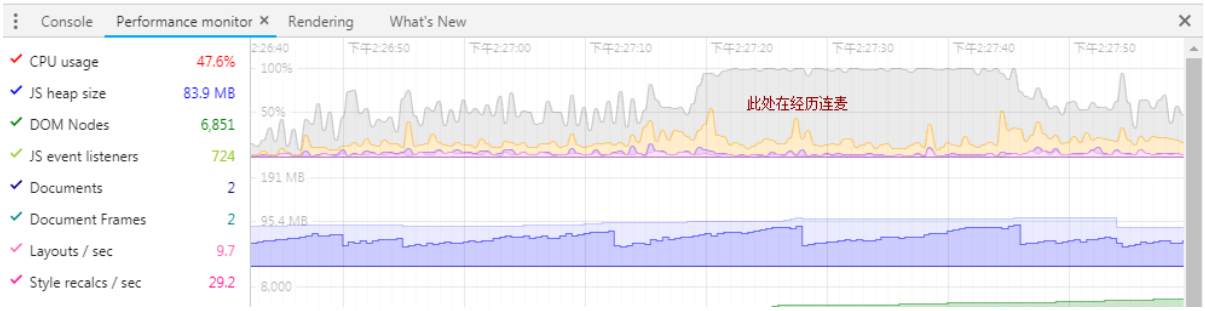
2. CPU 检查现象

1. *performance monitor 结果*

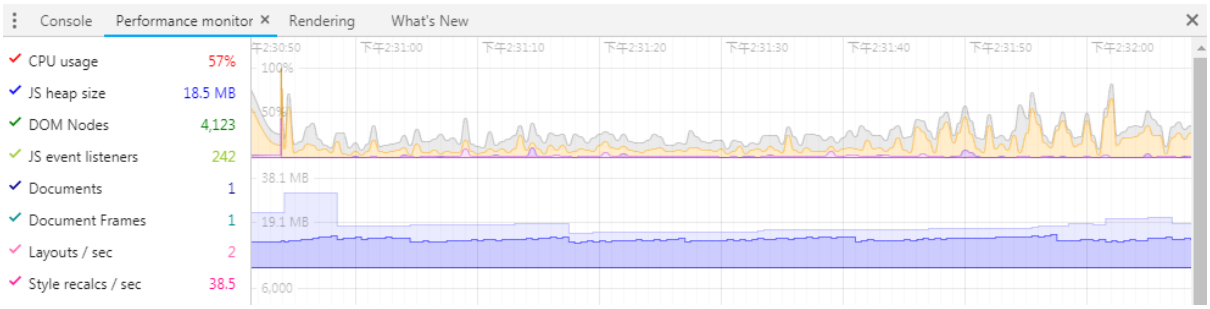
奇秀：



斗鱼：



虎牙：



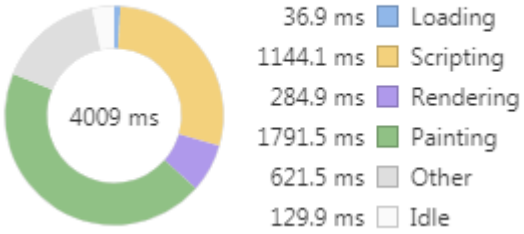
在观察期间发现，奇秀直播间 CPU 持续处于 98%左右的情况，也就是说，在静态看直播的过程中，没有任何交互的情况下。CPU 任然会被打满。而占满 CPU 的区域基本是灰色区域。灰色区域示意为 Painting 占比。

2. *performance profile 结果*

奇秀：

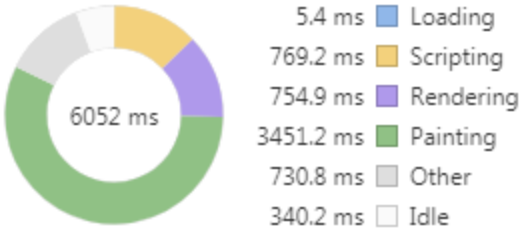
直播间刷新：

Range: 0 – 4.01 s



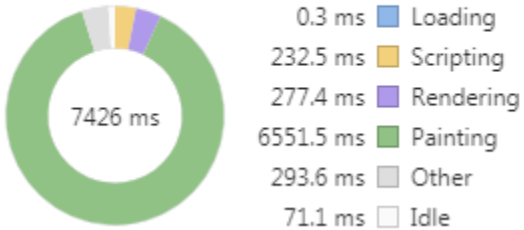
观看过程 1：

Range: 0 – 6.05 s



观看过程 2：

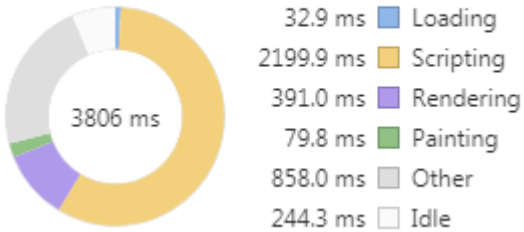
Range: 0 – 7.43 s



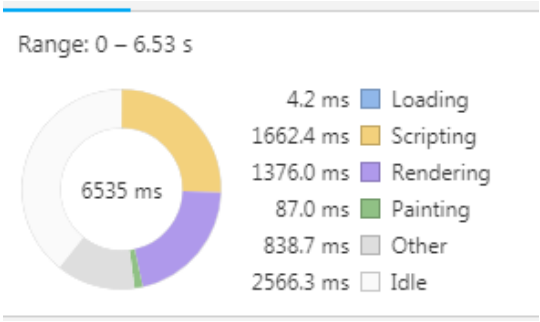
虎牙：

直播间刷新：

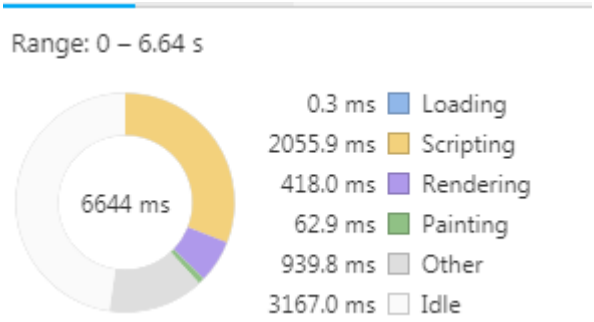
Range: 0 – 3.81 s



观看过程 1：

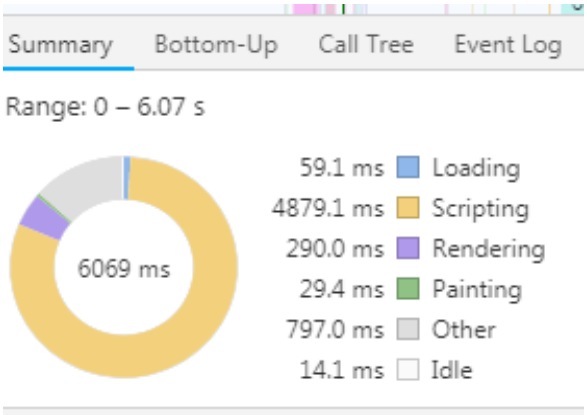


观看过程 2：

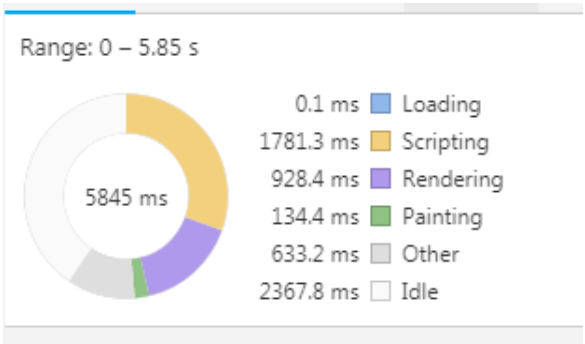


斗鱼：

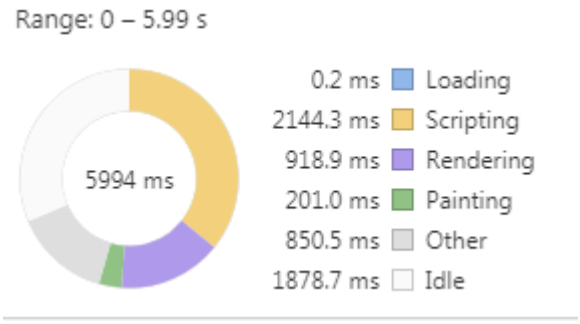
直播间刷新：



观看过程 1：

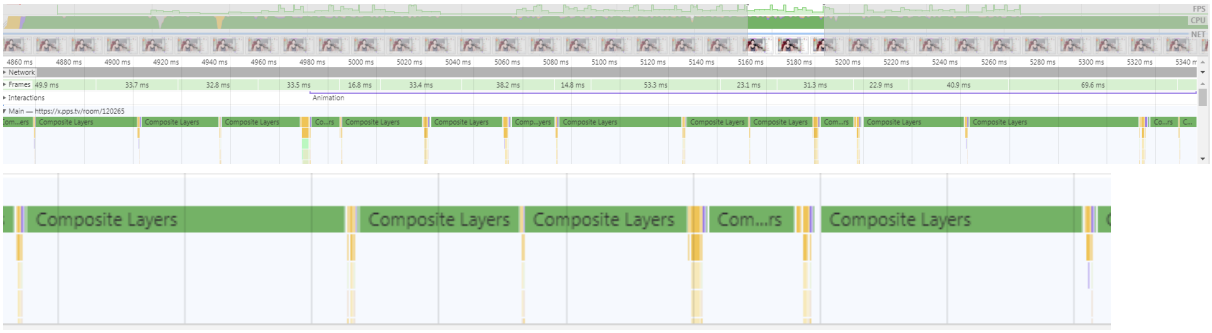


观看过程 2：



经过观察和对比发现。奇秀直播间在被检测的时间段内，CPU 花费大量的时间在处理绿色部分的事情，也就是我们上面提到的 Painting 和 Composite。

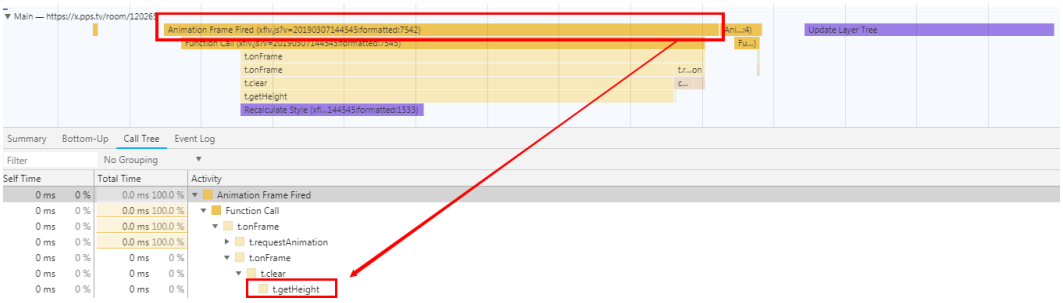
再结合火焰图来看，



几乎 90%的时间，CPU 都在做这个 composite 的事情，这是非常不合理的一个现象，那我们通过调用栈来逐一分析是什么导致的这样频繁进行 composite 的执行。

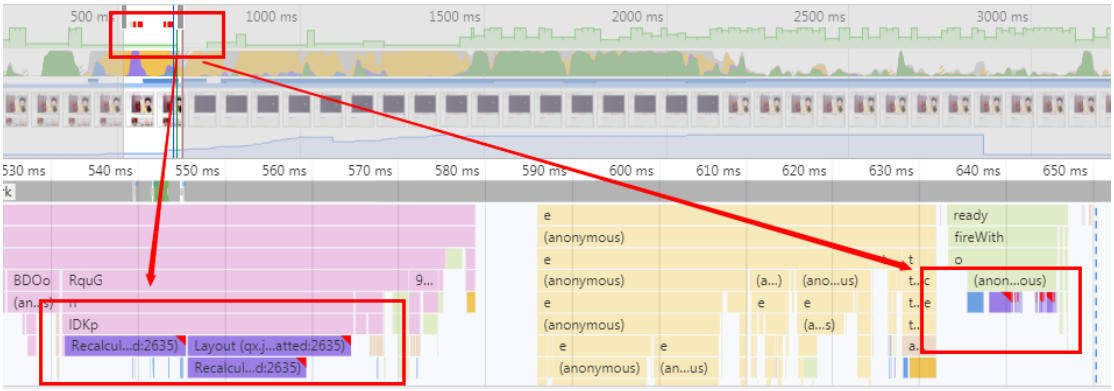
我们发现每次 composite 之前都会有一段 JS 被执行，非常规律。

仔细分析这段神秘的 JS 脚本的调用发现

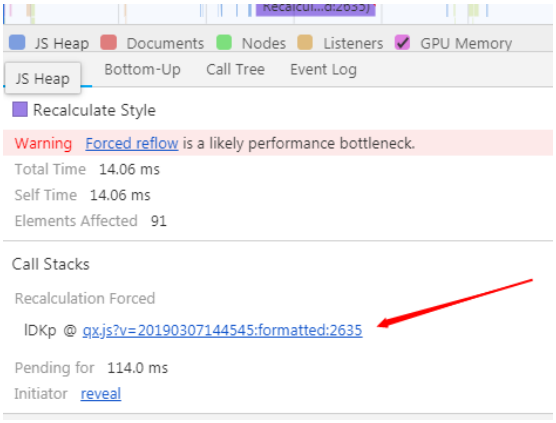


LOOK!!!!!!这里是在获取 canvas 的及时高度。

在分析 FPS 面板时，我发现在页面加载时出现了几处红杠,并且伴随着紫色小高峰。也就是我们说的 rendering



同样的方式我我们分析火焰图




```
2632                cid: "",
2633                lurl: r,
2634                ri: "",
2635                re: document.documentElement.clientWidth + "x" + document.documentElement.clientHeight,
2636                content: "",
2637                sc: "",
2638                tm: 0,
2639                t: "qos",
2640                bstp: "web"
2641            }
2642        }
```

以及

```
p.innerHTML = "<table><tr><td style='" + t + "0;display
k = p.getElementsByTagName("td"),
o = k[0].offsetHeight === 0,
k[0].style.display = "",
k[1].style.display = "none",
b.reliableHiddenOffsets = o && k[0].offsetHeight === 0,
a.getComputedStyle && (p.innerHTML = "",
l = c.createElement("div"),
```

那么问提来了,为啥会导致 composite 呢。下面一节详细分析

3. 结果分析

我们找到了奇秀直播间在初期加载以及观看过程中的一些异常，最大的异常莫过于整个 Summary 面板太绿了，至少在 windows 系统下，实在太绿了。也就是说绝大多数的时间都花费在了 Painting 和 Composite 上。那究竟哪些行为会导致 Painting 和

Composite 的执行呢。

1. ***老生常谈的重绘与重排***

DOM 的修改会导致重绘和重排。

重绘是指一些样式的修改，元素的位置和大小都没有改变；

重排是指元素的位置或尺寸发生了变化，浏览器需要重新计算渲染树，而新的渲染树建立后，浏览器会重新绘制受影响的元素。

对标一下 performance 中的事件名称，**重绘**的步骤包括 **Paint、Composite Layers** 等绘制类的事件。**重排**的步骤包括 **Recalculate Style、Layout、Update Layer Tree** 等渲染类事件。重绘是重排的子集。

了解了什么是重绘和重排，针对奇秀直播间的问题，我们目标是要找到导致 **Composite Layers 事件发生的原因。**那么哪些行为会导致页面的重排与重绘呢

导致页面重排的一些操作：

1. 内容改变（文本改变或图片尺寸改变）
2. DOM 元素的几何属性的变化（例如改变 DOM 元素的宽高值时，原渲染树中的相关节点会失效，浏览器会根据变化后的 DOM 重新排建渲染树中的相关节点。如果父节点的几何属性变化时，还会使其子节点及后续兄弟节点重新计算位置等，造成一系列的重排）
3. DOM 树的结构变化（添加 DOM 节点、修改 DOM 节点位置及删除某个节点都是对 DOM 树的更改，会造成页面的重排。浏览器布局是从上到下的过程，修改当前元素不会对其前边已经遍历过的元素造成影响，但是如果在所有的节点前添加一个新的元素，则后续的所有元素都要进行重排。）
4. 获取某些属性（除了渲染树的直接变化，当获取一些属性值时，浏览器为取得正确的值也会发生重排，这些属性包括：offsetTop、offsetLeft、offsetWidth、offsetHeight、scrollTop、scrollLeft、scrollWidth、scrollHeight、clientTop、clientLeft、clientWidth、clientHeight、getComputedStyle()。）
5. 浏览器窗口尺寸改变（窗口尺寸的改变会影响整个网页内元素的尺寸的改变，即 DOM 元素的集合属性变化，因此会造成重排）。

导致页面重绘的操作

1. 应用新的样式或者修改任何影响元素外观的属性（只改变了元素的样式，并未改变元素大小、位置，此时只涉及到重绘操作。）
2. 重排一定会导致重绘（一个元素的重排一定会影响到渲染树的变化，因此也一定会涉及到页面的重绘。）

所以分析完这些原因，我们发现火焰图中最底层的.getHeight()方法其实就是触发了这次重排的主要原因。但是为什么会有如此频繁的触发呢？其实因为我们调用.getHeight()的地方在 window.requestAnimationFrame()这个方法里，随着每一帧的刷新都在调用.getHeight()导致重排（Composite Layers）的方法。

2．解决方案

由于大量的导致重排方法是在 window.requestAnimationFrame()里调用的，而 requestAnimationFrame 已经确认是在直播间弹幕里才会用到，所以对症下药。

1. 我们可以按需触发，也就是说在已知确实有新的弹幕生成的时候，才去调用该方法。空闲时间做屏蔽处理。
2. 我们可以规避 offsetHeight 操作，或者说将样式读取放在样式设置之前，这样可以避免二次重排。
3. 对于其他竞品的弹幕处理，斗鱼和虎牙目前用的都是 DOM 弹幕，并且利用 CSS3 动画启用 GPU 加速，规避了 JS 直接操作 DOM 带来的巨大消耗。建议可以对 Canvas 弹幕进行整改。

3. 优化策略

谈了那么多理论，最实际不过的，就是解决方案，大家一定都等着急了吧，做好准备，一大波干货来袭：

1. CSS 属性读写分离：浏览器每次对元素样式进行读操作时，都必须进行一次重新渲染（重排 + 重绘），所以我们在使用 JS 对元素样式进行读写操作时，最好将两者分离开，先读后写，避免出现两者交叉使用的情况。最最最客观的解决方案，就是不用 JS 去操作元素样式，这也是我最推荐的。
2. 通过切换 class 或者使用元素的 style.csstext 属性去批量操作元素样式。
3. DOM 元素离线更新：当对 DOM 进行相关操作时，例、appendChild 等都可以使用 Document Fragment 对象进行离线操作，带元素 “组装” 完成后再一次插入页面，或者使用 display:none 对元素隐藏，在元素 “消失” 后进行相关操作。

4. 将没用的元素设为不可见：visibility: hidden，这样可以减小重绘的压力，必要的时候再将元素显示。
5. 压缩 DOM 的深度，一个渲染层内不要有过深的子元素，少用 DOM 完成页面样式，多使用伪元素或者 box-shadow 取代。
6. 图片在渲染前指定大小 :因为 img 元素是内联元素，所以在加载图片后会改变宽高，严重的情况会导致整个页面重排，所以最好在渲染前就指定其大小，或者让其脱离文档流。
7. 对页面中可能发生大量重排重绘的元素单独触发渲染层，使用 GPU 分担 CPU 压力。（这项策略需要慎用，得着重考量以牺牲 GPU 占用率为代价能否换来可期的性能优化，毕竟页面中存在太多的渲染层对于 GPU 而言也是一种不必要的压力，通常情况下，我们会对动画元素采取硬件加速。）

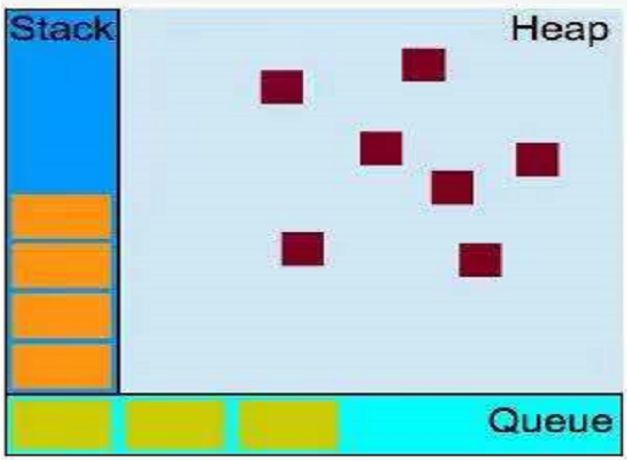
二、内存篇

1.关于内存

先温习一下几个概念

1. JS 的内存模型

直接上图：



堆（Heap）：咱们创建的对象就是存在堆里面的。也是我们关注内存泄漏和定位的主要区

域

栈（Stack）：函数执行栈

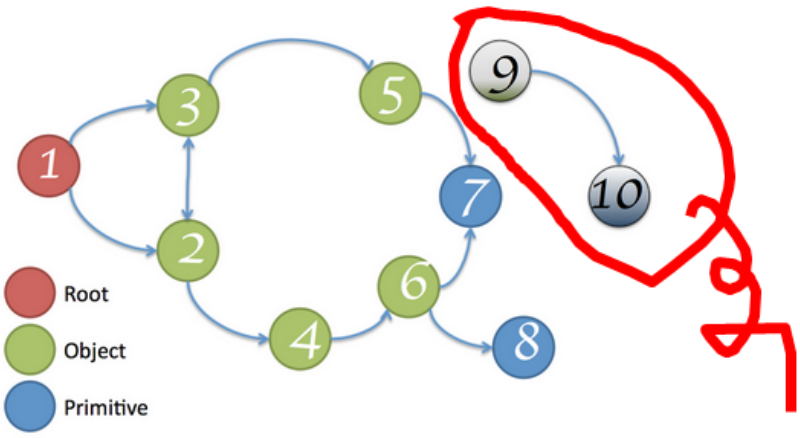
队列（Queue）：消息队列，我理解为异步等待执行所排的队，就是还没有被执行的函数放这里

2.关于内存的申请与回收

JS 不用手动为对象申请内存，只需要声明变量即可，甚至不需要指明类型。就是这么拍脑子办事。

JS 有 Garbage Collection（GC）的语言，也就是我们不需要手动的回收内存。不同的 JavaScript 引擎有不同的垃圾回收机制，这里我们主要以 V8 这个被广泛使用的 JavaScript 引擎为主。

GC 根：一般指全局且不会被垃圾回收的对象，比如：window、document 或者是页面上存在的 dom 元素。JavaScript 的垃圾回收算法会判断某块对象内存是否是 GC 根可达（存在一条由 GC 根对象到该对象的引用），如果不是那这块内存将会被标记回收。来张图：



3.作用域的分类：局部作用域、全局作用域、闭包作用域

3.1 局部作用域

函数调用会创建局部作用域，在局部作用域中的新建的对象，如果函数运行结束后，该对象没有作用域外部的引用，那该对象将会标记回收

3.2 全局作用域

每个 JavaScript 进程都会有一个全局作用域，全局作用域上的引用的对象都是常驻内存的，直到进程退出内存才会自动释放。

手动释放全局作用域上的引用的对象有两种方式：

global.foo = null 重新赋值改变引用

delete global.foo 删除对象属性

3.3 闭包作用域

在 JavaScript 语言中有闭包的概念,闭包指的是包含自由变量的代码块、自由变量不是在这个代码块内或者任何全局上下文中定义的，而是在定义代码块的环境中定义（局部变量）。

```
var closure = (function(){
    //这里是闭包的作用域
    var i = 0 // i 就是自由变量
    return function () {
        console.log(i++)
    }
})()
```

闭包作用域会保持对自由变量的引用。上面代码的引用链就是:

window -> closure -> i

闭包作用域还有一个重要的概念，闭包对象是当前作用域中的所有内部函数作用域共享的，并且这个当前作用域的闭包对象中除了包含一条指向上一层作用域闭包对象的引用外，其余的存储的变量引用一定是当前作用域中的所有内部函数作用域中使用到的变量

4.常见的导致内存泄漏骚操作

4.1 使用全局变量存数据。

我们知道全局变量是永远处于 GC 根可达状态，这样也就没办法在页面没被关闭的情况下被回收了。最骚的是，在开发过程中还会使用各种循环，定时器等方式给已经定义的全局变量不停的增大体积。这样一来，内存便会越来越大。比如下面这样的代码：

```
var x = [];
function createSomeNodes() {
    var div;
    var i = 10000;
    var frag = document.createDocumentFragment();
    for (; i > 0; i--) {
        div = document.createElement("div");
        div.appendChild(document.createTextNode(i + " - " + new
        Date().toTimeString()));
        frag.appendChild(div);
    }
    document.getElementById("nodes").appendChild(frag);
```

```
}
function grow() {
    x.push(new Array(1000000).join('x'));
    createSomeNodes();
    setTimeout(grow, 1000);
}
grow()
```

4.2 脱离 DOM 的引用。

有时，保存 DOM 节点内部数据结构很有用。假如你想快速更新表格的几行内容，把每一行 DOM 存成字典（JSON 键值对）或者数组很有意义。此时，同样的 DOM 元素存在两个引用：一个在 DOM 树中，另一个在字典中。将来你决定删除这些行时，需要把两个引用都清除

```
var elements = {
    button: document.getElementById('button'),
    image: document.getElementById('image'),
    text: document.getElementById('text')
};
function doStuff() {
    image.src = 'http://some.url/image';
    button.click();
    console.log(text.innerHTML);
    // 更多逻辑
}
function removeButton() {
    // 按钮是 body 的后代元素
    document.body.removeChild(document.getElementById('button'));
    // 此时，仍旧存在一个全局的 #button 的引用
    // elements 字典。button 元素仍旧在内存中，不能被 GC 回收。
}
```

4.3 子元素被引用导致的内存泄漏

--黄色是指直接被 js 变量所引用，在内存里

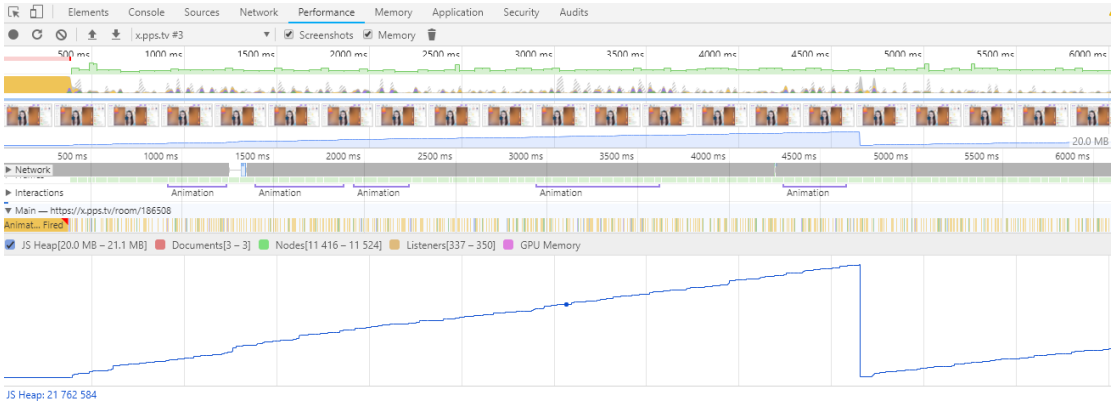
--红色是指间接被 js 变量所引用，如上图，refB 被 refA 间接引用，导致即使 refB 变量被清空，也是不会被回收的

--子元素 refB 由于 parentNode 的间接引用，只要它不被删除，它所有的父元素（图中红色部分）都不会被删除

动图解释：https://www.cnblogs.com/libin-1/p/6013490.html

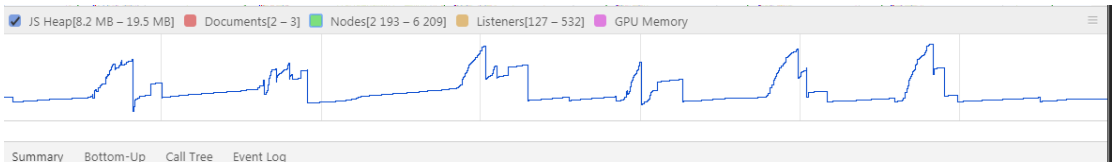
2. 分析工具

1.performance 中勾选 memory

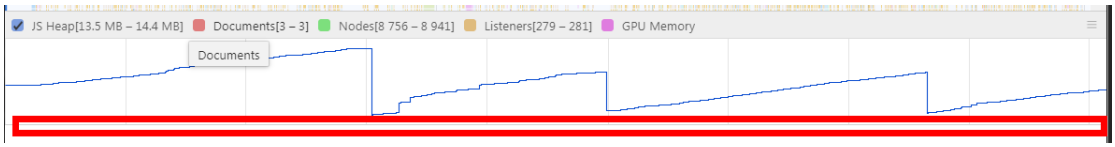


这里的几个选项在 CPU 分析中基本都有做解释，这里不再赘述。我们可以通过这个工具很好的观察到页面在运行过程中的内存总体走势。下面是几个例子：

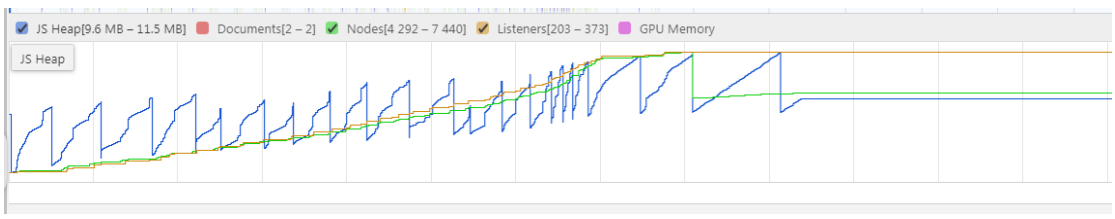
1.点击换一换功能



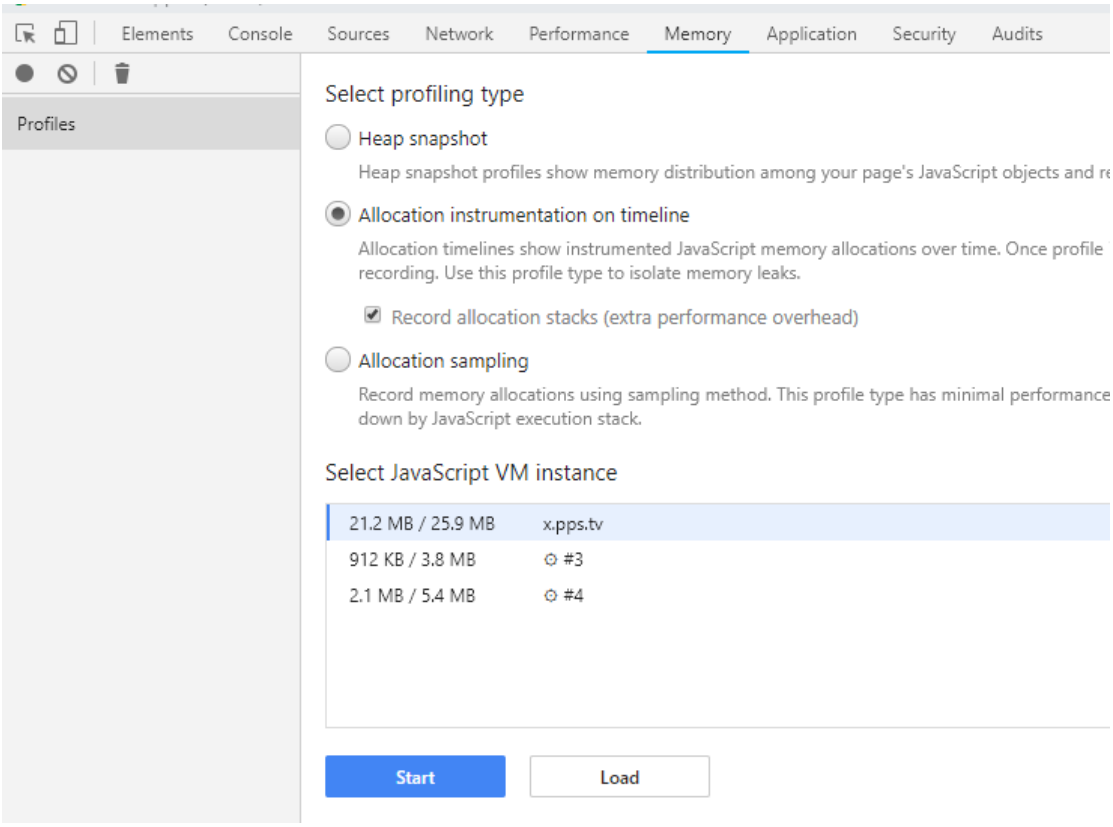
2.正常观看情况下的内存状态



3.持续送礼物情况下

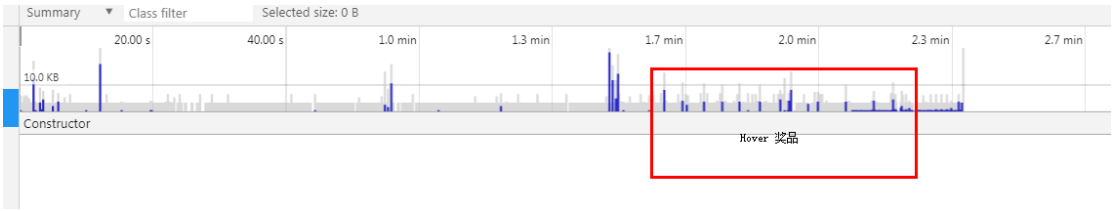


3.memory 面板

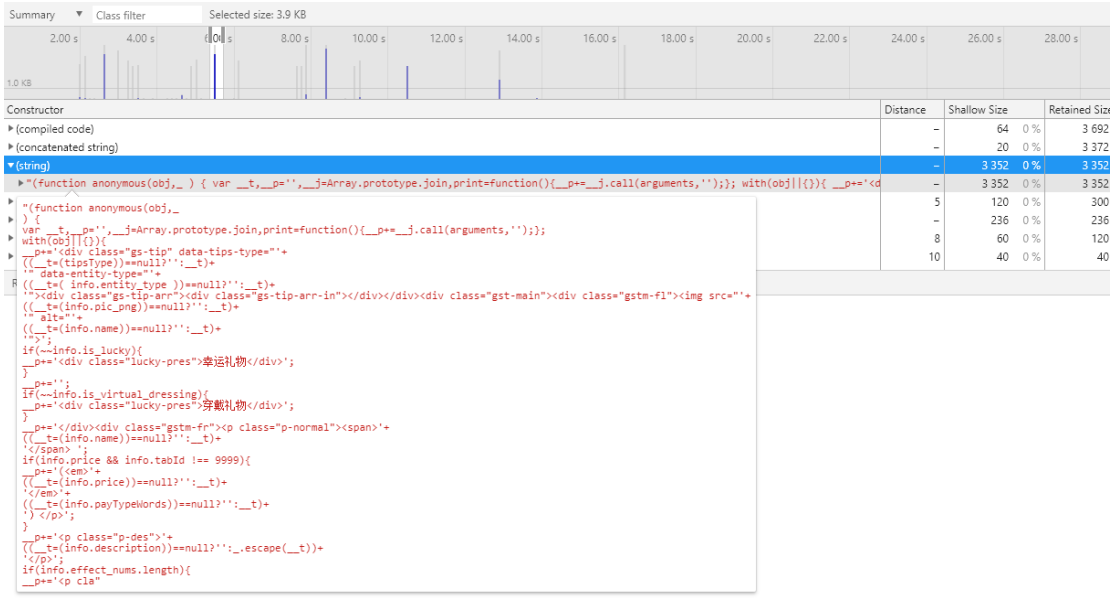


大家应该都知道这里的的 Heap snapshot（内存快照），其实我个人认为这个功能更适合做某些特殊操作的处理观察

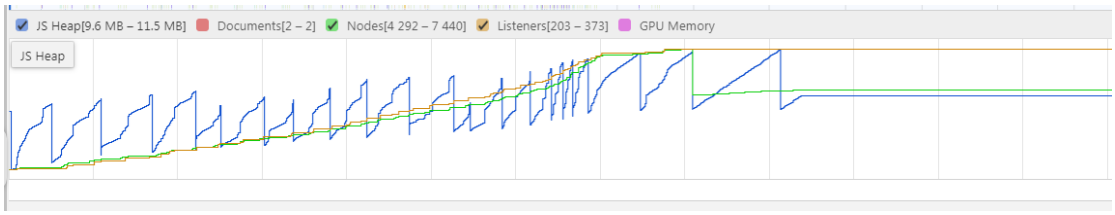
Allocation instrumentation on timeline,这个功能感觉挺好用的，来个例图：



它是结合 timeline 的一个动态记录工具，蓝色高线代表当下申请的内存，灰色代表已经释放的内存。观察发现，长时间无法被释放的内存。基本都是直接或者间接的挂在了 GC 根上的。比如上图鼠标 hover 到礼品上时，出现礼品详情浮层这一动作时。内存总是在被重复申请。并且无法被释放。细化分析之后我们再看



这里贴上一张送礼物时的内存状态：



由于这里的业务逻辑我并不是特别了解，所以还有待完善

3. 我们能做的

3.1 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』。

3.2 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;
```

3.3 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。