

16毫秒的优化

Web前端性能优化的微观分析

个人介绍

- 网名: 舜子、PJ
- 2004年, 发布开源博客PJBlog
- 2006年加盟腾讯公司至今
 - 腾讯资深前端开发工程师
 - 曾负责QQ空间, 腾讯开放平台等海量服务的前端架构设计和性能优化
 - 目前在SNG应用团队, 负责创新项目的研发工作, 以及移动终端Web前端性能优化的研究工作。





智能终端的普及改变了人们对互联网的使用习惯

跨终端是给Web开发工程师带来新的挑战机会



用户都希望Mobile Web可以获得和PC Web接近的加载速度，甚至更快的加载速度。同时又希望Mobile Web能获得如同APP一样交互体验。

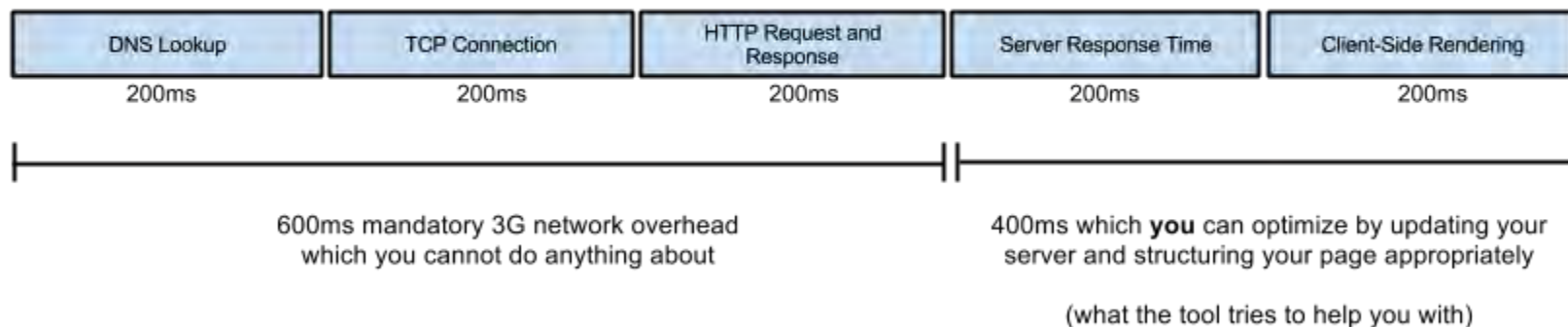


不管环境如何变化
速度依旧是刚性需求！

- 一个2秒内无法打开的web就意味着用户的流失。

Google提出1秒钟完成终端页面首屏的渲染

Rendering a mobile page in 1 second



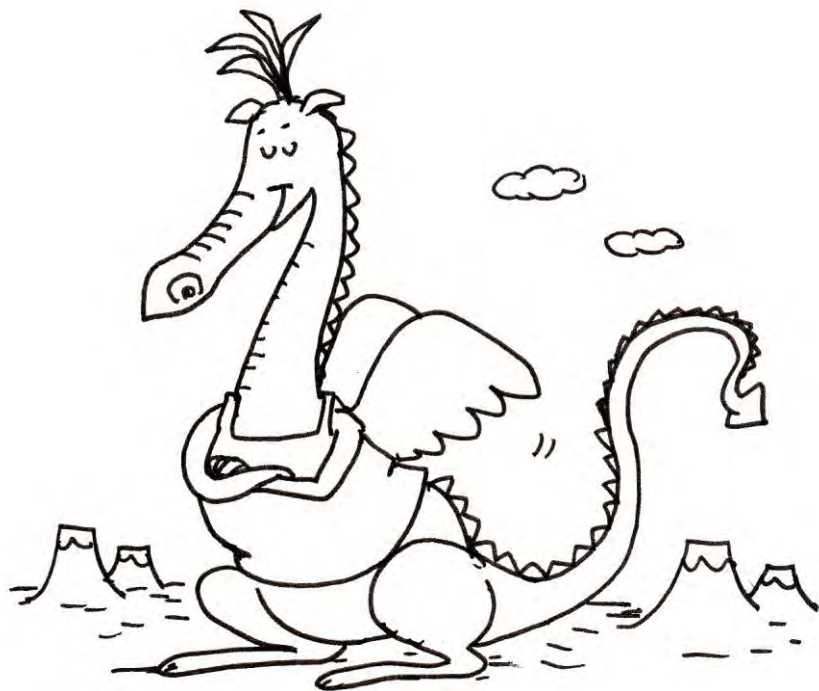
1. 服务器响应应该小于200ms
2. 尽量少的重定向
3. 尽量少的第一次渲染的请求数
4. 避免过多堵塞的JS和CSS的堵塞
5. 给浏览器留200ms的渲染时间
6. 优化我们的JS执行效率和渲染时间

网络消耗

JS执行效率和渲染效率

网络请求的优化，我们有非常多的经验

- DNS Lookup
- 减少重定向
- 并行请求
- 压缩
- 缓存
- 按需加载
- 前端模块化
- ...

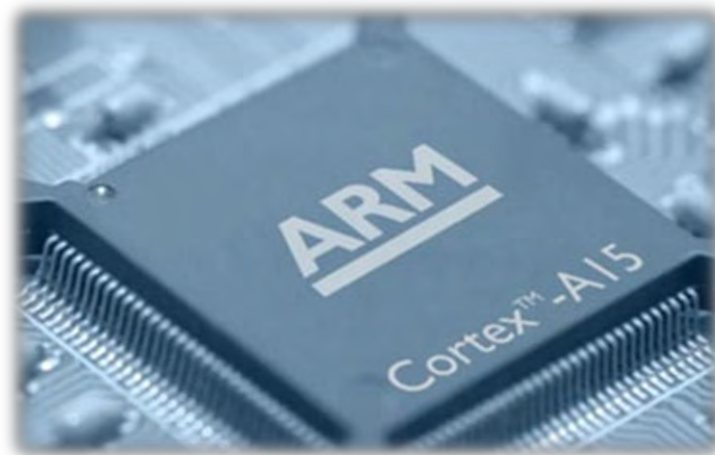


为什么强调需要思考渲染问题？



终端和PC的性能差异巨大

- x86的性能是ARM的5倍以上甚至更多，原先存在的许多性能问题被掩盖。终端市场爆发后，问题随之而来
 - 用户所持的终端CPU优劣差异巨大
 - 运行速度慢
 - 用户操作卡顿
 - 内存不足
 - 本地存储空间不足
 - 可控的资源有限
 - ...



IOS vs Android

- 早期为何大家会觉得iphone比android手机好用?

- 如丝般顺滑的动画和滑动

流畅

- 省电

节能

- 很少crash

稳定

	型号	CPU	RAM
iOS	iPhone 4S	双核A5 800MHZ	512M
	iPhone 4	A4 800MHZ	512M
	iPhone 3GS	S5PC100 600MHZ	256M
Android	Galaxy Note	Exynos 双核 1.4GHZ	1G
	Nexus One	高通 1GHZ	512M
	MOTO XT615	高通 800MHZ	512M
	HTC Legend	高通 600MHZ	384M



更完整的Web前端性能优化



学会发现Web渲染性能问题

- Tips

- 在庞大的网络优化面前，渲染优化显得杯水车薪，但是它任然值得我们去探索。
- 不要认为渲染优化，只是浏览器理所当然做的事情。

一切性能优化都建立在可衡量的 数据分析基础之上

- You can't optimize what you cannot measure

三个准备工作

- 确定渲染性能的分析标准
- 准备尺子去衡量渲染性能标准
- 了解浏览器基础的渲染的工作原理

1. 判断渲染性能的标准

- 帧数

- 显示设备通常的刷新率通常是50~60Hz
- $1000\text{ms} / 60 \approx 16.6\text{ms}$ (1毫秒的优化意味着6%的性能提升)

Web渲染性能优化目标:

脚本时间 + 渲染时间 + 绘制时间 < 16ms (每帧)

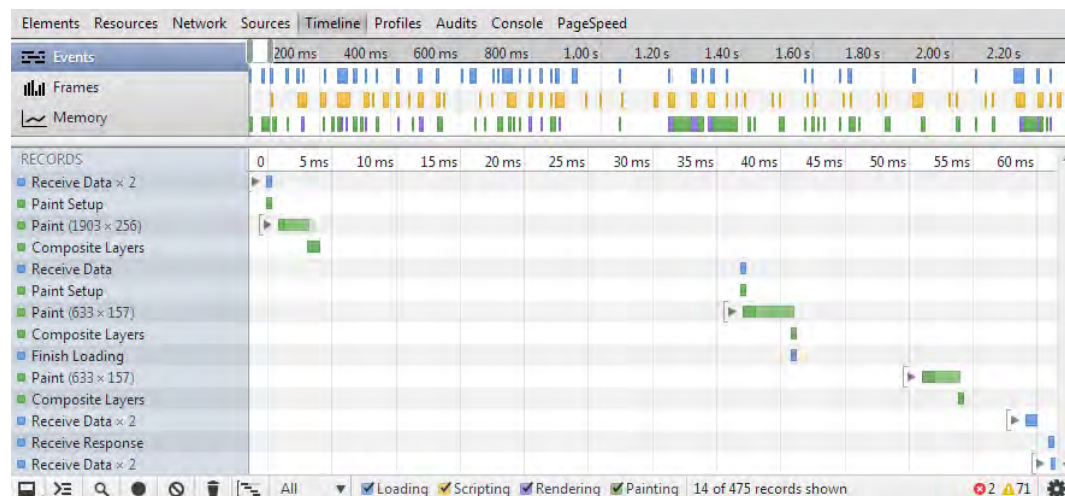
脚本执行

渲染

绘制/合成

2. 准备好尺子

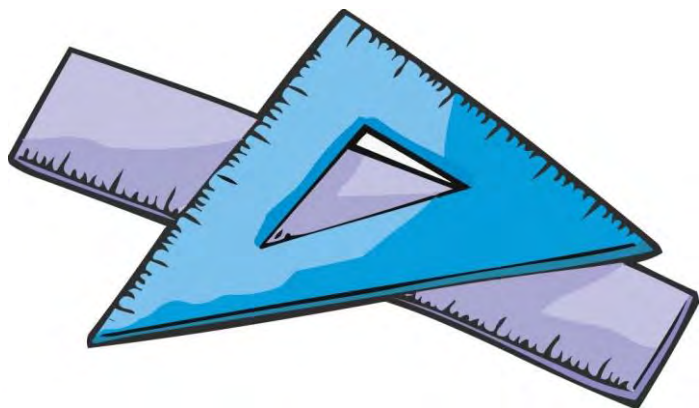
- DevTools (chrome/safari)
 - Timeline
 - Profiles
- IE11 DevTools
 - UI Responsiveness
 - Profiler



Source: <https://developers.google.com/chrome-developer-tools/docs/timeline>

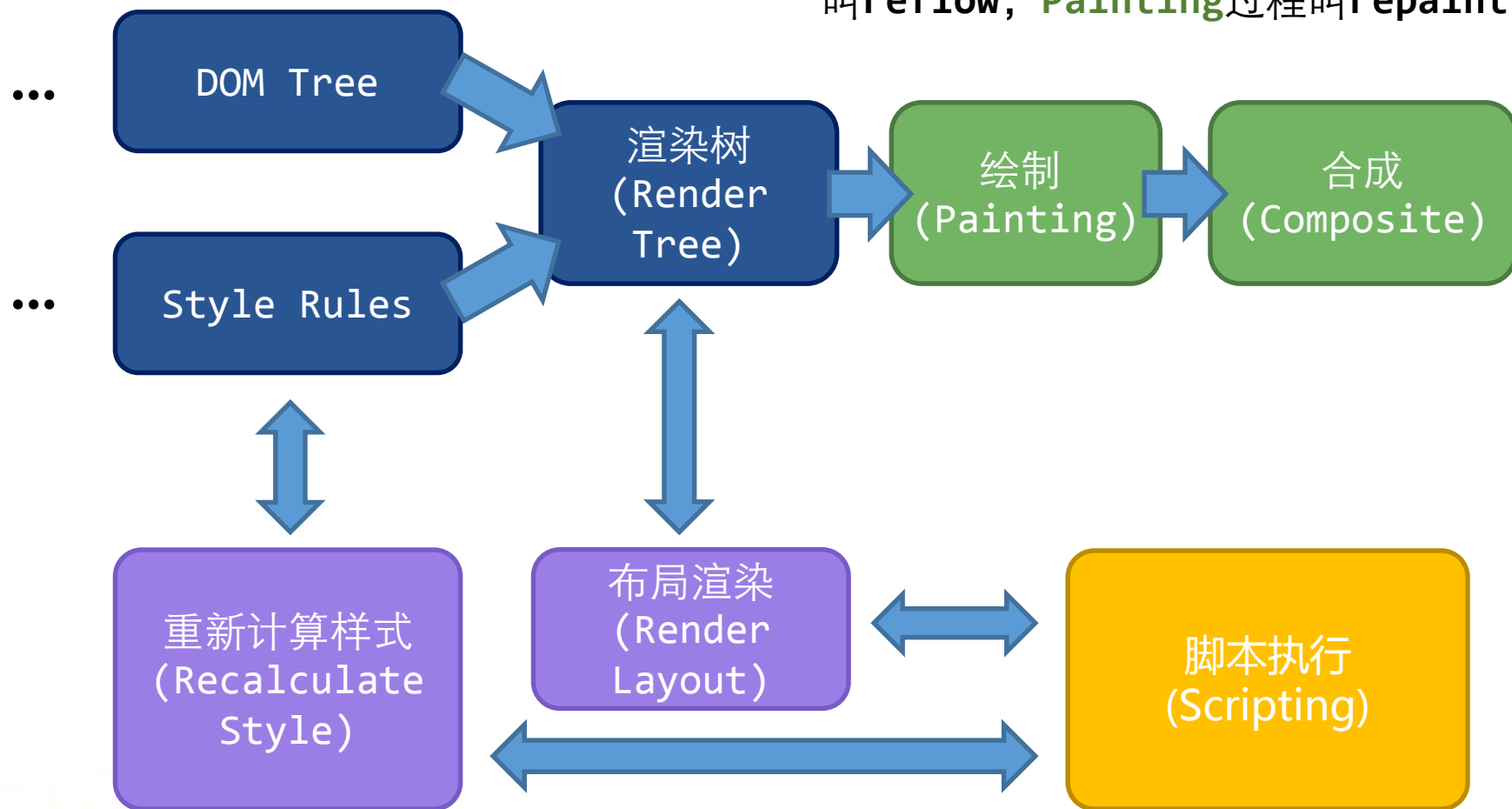


Source: [http://msdn.microsoft.com/en-us/library/ie/dn255009\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ie/dn255009(v=vs.85).aspx)

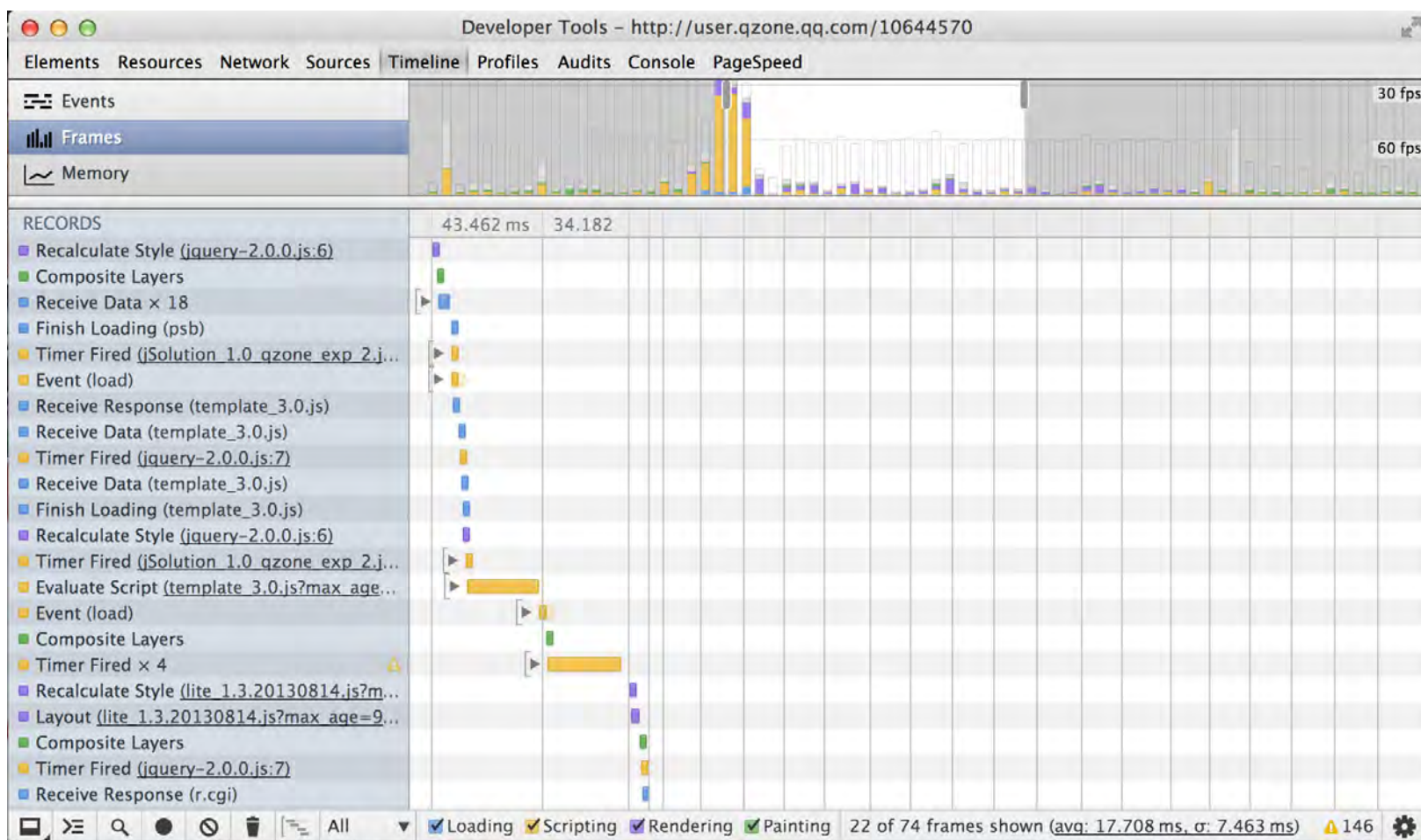


3. 了解一下浏览器渲染引擎

TIPS: 在firefox中, **Layout**的过程叫**reflow**, **Painting**过程叫**repaint**



Timeline整体概览

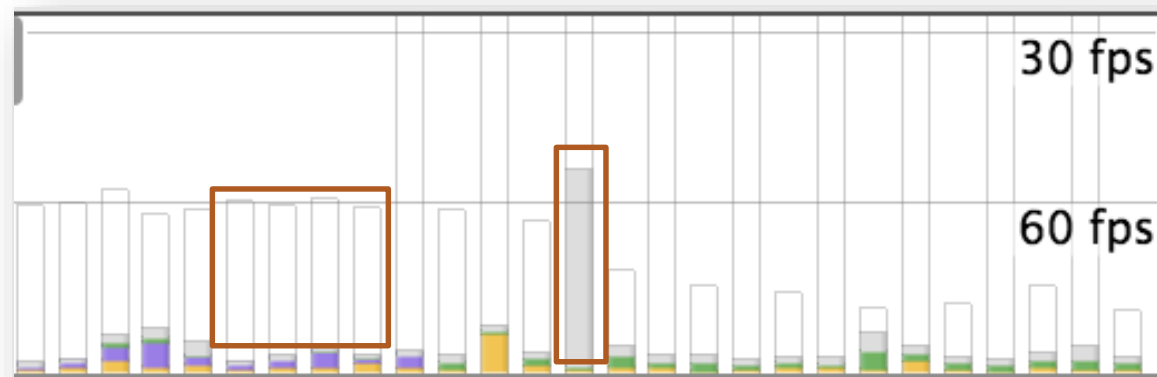


Timeline 两个常用的模式

- Events

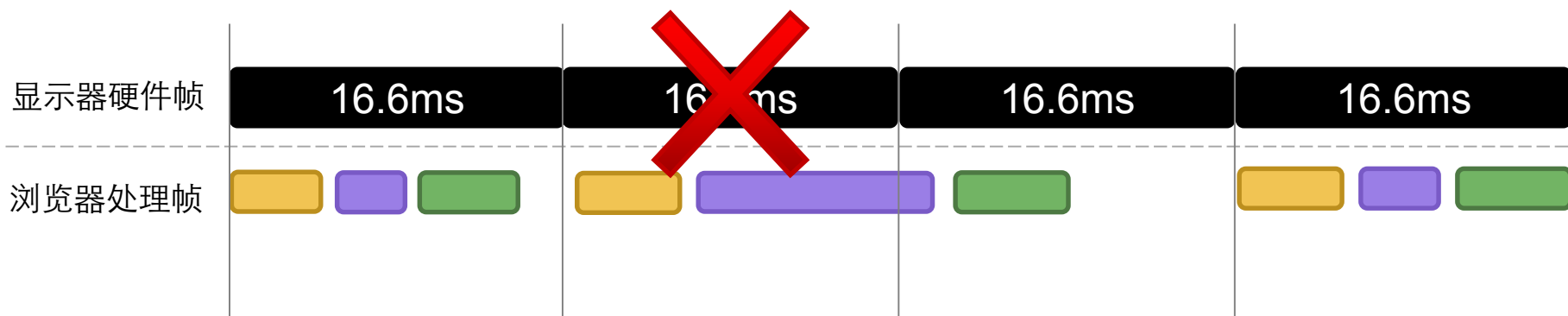


- Frames
(Chrome only)



- TIPS:**
1. Timeline记录行为的快捷键是 Ctrl+E (Mac:Cmd+E)
 2. 灰色区域是非渲染引擎发生的渲染行为。
 3. 透明的线框,是在两次显示器刷新周期中的等待时间

浏览器渲染的丢帧现象



脚本执行 渲染 绘制/合成

来认识一下我们的敌人

- 重新计算样式(Recalculate Style)
 - 计算布局(Layout)
 - 绘制(Paint Setup/Paint(size x size))
- } Rendering/Reflow
- } Painting/Repaint

这三个行为是web渲染优化重点思考的问题

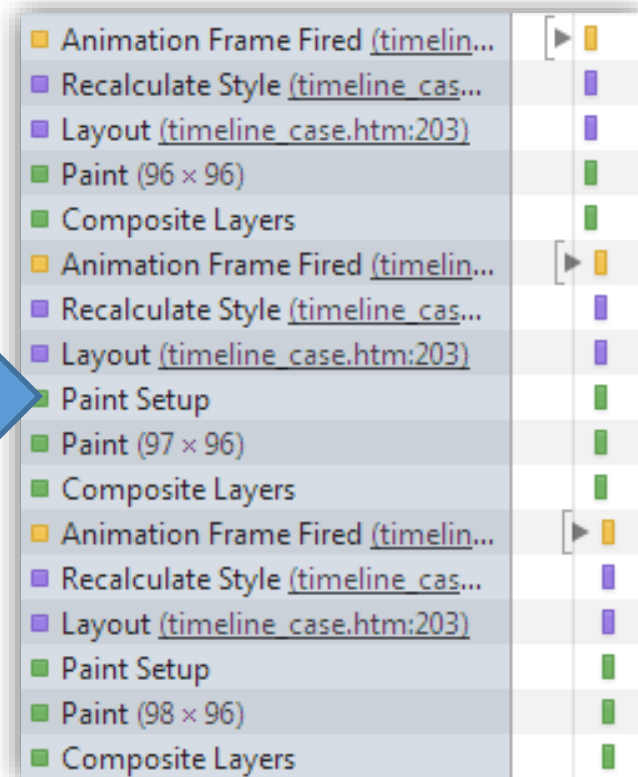


避免常见的重复渲染

- 保证每帧最多只发生2次渲染树更新

不在16ms内多次发生重复的渲染行为

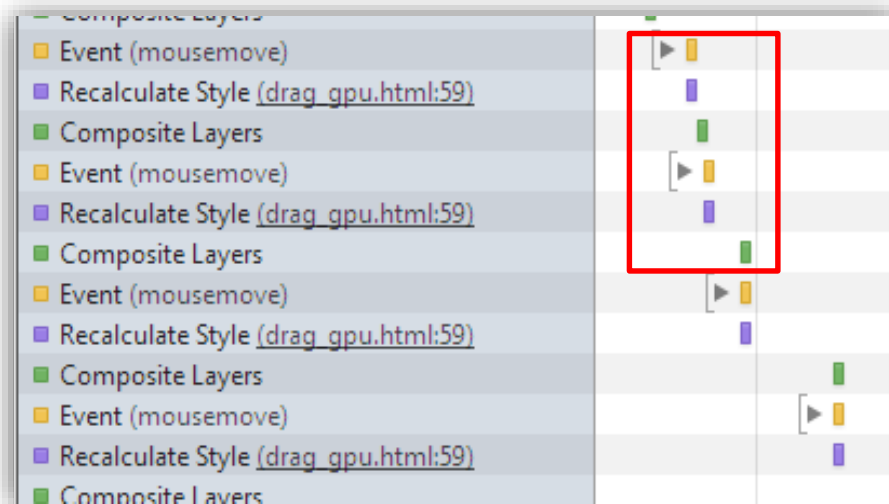
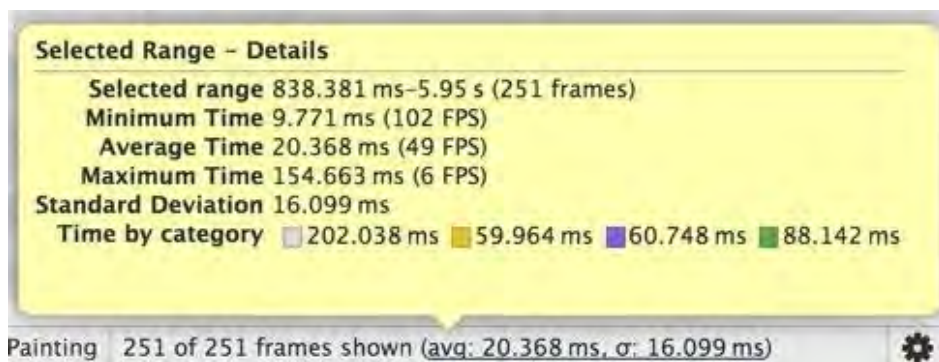
- 不要使用setTimeout/setInterval
 - 使用 requestAnimationFrame



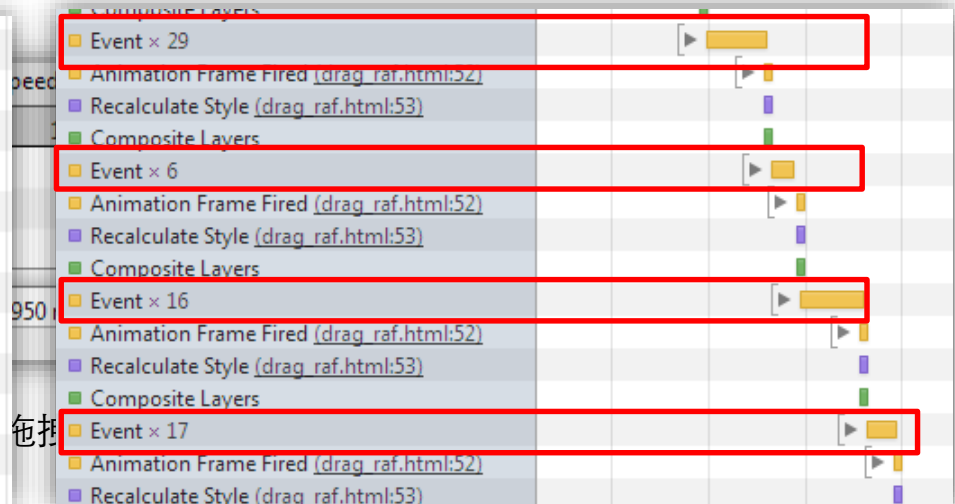
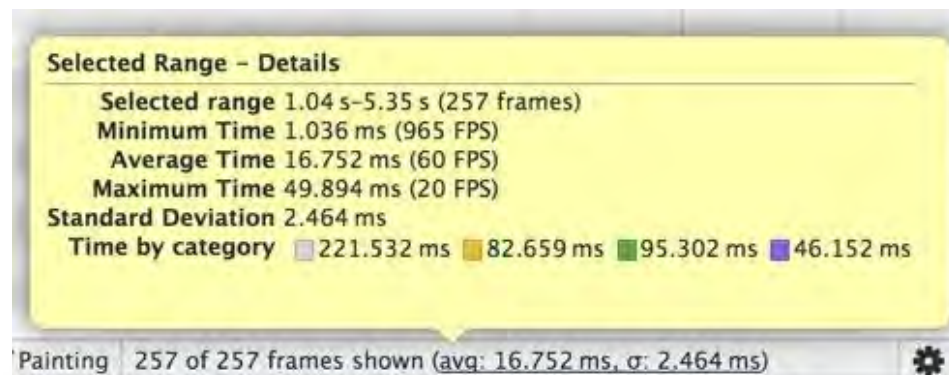
requestAnimationFrame进阶玩法

- 在PC电脑和手机终端. 操作系统对用户行为响应时间差是小于16ms的.
 - 在16ms内, 用户可能会触发多次events
 - 每次events会带来没必要的js callback

requestAnimationFrame进阶玩法



直接通过**event**行为触发
 渲染浪费的渲染帧,不会被硬件帧最终显示
 腾讯大讲堂



通过**RAF**控制渲染频次,合并
 了大量**event**的计算,
 减少了无效的渲染次数

使用RAF来保证事件行为选择正确的时间进行渲染

```
var _drag = document.getElementById("drag");
var _clientX,_clientY,rAF;

var _mousedown = function(evt){
  _clientX = evt.clientX - (_drag._x || 0);
  _clientY = evt.clientY - (_drag._y || 0);

  document.addEventListener("mousemove",update,true);
  document.addEventListener("mouseup",clearEvent,true);

  draw(); // 鼠标点击开始, 以及开始监听RAF
}
```

```
// 只用来计算对话框新的位置, 不做表现渲染
var update = function(evt){
  _drag._x = evt.clientX - _clientX;
  _drag._y = evt.clientY - _clientY;
}
```

```
// 获取最新的坐标, 绘制对话框位置
var draw = function(){
  _drag.style.webkitTransform = "translate3D(" + _drag._x + "px," + _drag._y + "px,0px)";
  rAF = requestAnimationFrame(draw);
}
```

不要破坏Dom操作应有的”原子性”

```
// Suboptimal, causes layout twice.  
var newWidth = aDiv.offsetWidth + 10; // Read  
aDiv.style.width = newWidth + 'px'; // Write  
  
var newHeight = aDiv.offsetHeight + 10; // Read  
aDiv.style.height = newHeight + 'px'; // Write
```

```
// Better, only one layout.  
var newWidth = aDiv.offsetWidth + 10; // Read  
var newHeight = aDiv.offsetHeight + 10; // Read  
aDiv.style.width = newWidth + 'px'; // Write  
aDiv.style.height = newHeight + 'px'; // Write
```

糟糕的DOM操作

```
// Suboptimal, causes layout twice.
var newWidth = aDiv.offsetWidth + 10; // Read
aDiv.style.width = newWidth + 'px'; // Write

var newHeight = aDiv.offsetHeight + 10; // Read
aDiv.style.height = newHeight + 'px'; // Write
```

1. 浏览器记录备用的样式

2. 忽略挂起的样式,重新计算Layout

类型	初始化程序
使样式无效	doBadLayout — rendering_Cycle.html:93
重新计算样式	doBadLayout — rendering_Cycle.html:95
使布局无效	doBadLayout — rendering_Cycle.html:95
布局	doBadLayout — rendering_Cycle.html:95
使样式无效	doBadLayout — rendering_Cycle.html:96
重新计算样式	—
使布局无效	—
布局	—

TIPS: chromium 的源码中, `updateLayoutIgnorePendingStylesheets()` 方法就是用来忽略备用的样式重新计算布局

读写分离, 批量操作

```
// Better, only one layout.  
var newWidth = aDiv.offsetWidth + 10; // Read  
var newHeight = aDiv.offsetHeight + 10; // Read  
aDiv.style.width = newWidth + 'px'; // Write  
aDiv.style.height = newHeight + 'px'; // Write
```

类型	初始化程序
使样式无效	 doGoodLayout — rendering_Cycle.html:104
重新计算样式	—
使布局无效	—
布局	—

使用 classList 代替 className

- className 只要赋值，就一定出现一次 rendering 计算。
- classList 的 add 和 remove，浏览器会进行样式名是否存在的判断，以减少重复的 rendering

```
function addClassName(name){
    aDiv.classList.add(name);
}

function removeClassName(name){
    aDiv.classList.remove(name);
}
```

避免重复的rendering行为

- 使用RAF来动画逻辑在合理的刷新时间内
- 保证渲染引擎的原子操作, 分离批量dom的读写行为
- 高端浏览器中尽量使用原生的classList操作样式的切换

构造渲染树的两个渲染行为的触发条件

- 重新计算样式
(Recalculate Style)

- style = ""
- className = ""
- classList.add()
- classList.remove()
- appendChild()

- 布局渲染
(Render Layout)

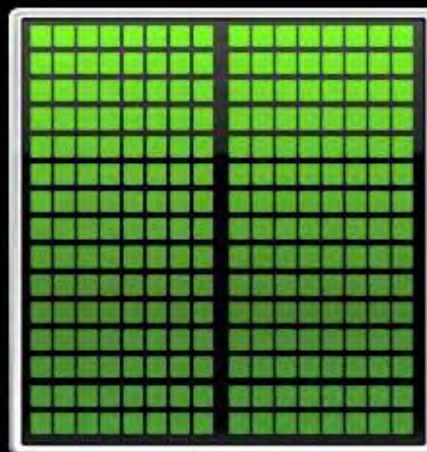
- display
- float
- position
- 盒模型的坐标和外形变化：
 - margin/padding
 - width/height
 - border
 - left/right/top/bottom

进阶: Painting 优化

- GPU加速 (Composite)



CPU



GPU

GPU在浏览器中是如何工作的？

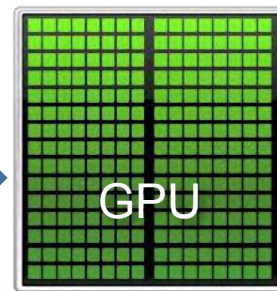
- 高端浏览器中 (webkit, chrome), **Painting** 行为是 CPU 为 GPU 准备纹理素材。



TIPS: chrome devtools 中可以开启 Show composited layer borders 查看图片纹理素材



平铺纹理
(tiles texture)



可以利用的GPU优势

- 纹理缓存
- 图形层



GPU Hack



-webkit-transform: `translate3D(0px,0px,0px);` ==> 创建Layout,并且在GPU中进行图层移动。



-webkit-transform: `translateZ(0);` ==> 创建Layout, 并缓存。

GPU Hack

- 强制把需要进行动画行为的dom对象，在GPU中创建Layout缓存

```
-webkit-transform: translateZ(0);
```

```
-webkit-backface-visibility: hidden;
```

- TranslateZ
- Perspective
- backface-visibility
- ScaleZ
- RotateX
- RotateY
- RotateZ
- Translate3D
- Scale3D
- Rotate3D

真实案例

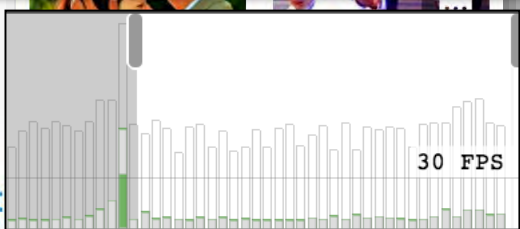
测试环境: MI2 + Chrome

测试目的: 测试轮播banner的滚动速度



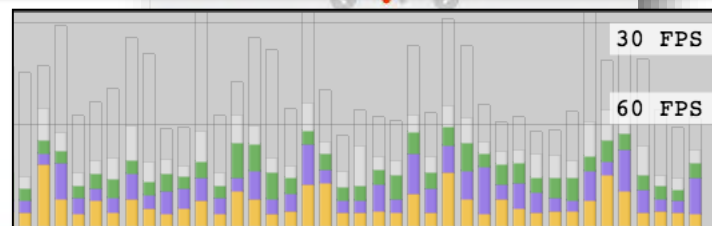
Selected Range - Details

Selected range 0.275 ms-4.22 s (53 frames)
 Minimum Time 56.310 ms (18 FPS)
 Average Time 79.637 ms (13 FPS)
 Maximum Time 126.110 ms (8 FPS)
 Standard Deviation 18.397 ms

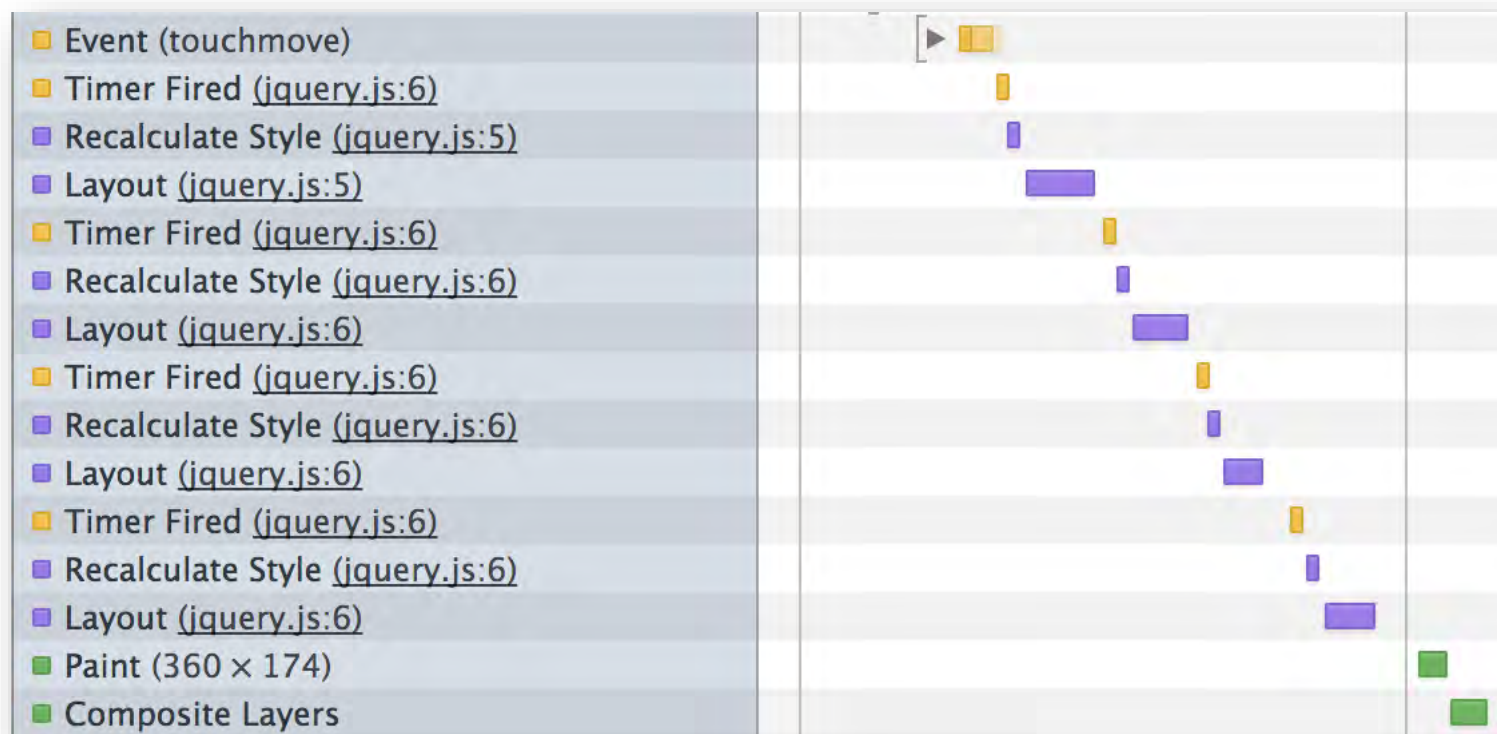


Selected Range - Details

Selected range 146.590 ms-2.98 s (111 frames)
 Minimum Time 13.032 ms (77 FPS)
 Average Time 25.524 ms (39 FPS)
 Maximum Time 156.997 ms (6 FPS)
 Standard Deviation 14.581 ms



Youku触屏版的问题



- 大量没有合并的rendering
- 不应该出现setTimeout的Timer Fired
- 使用不太适合终端的jQuery框架

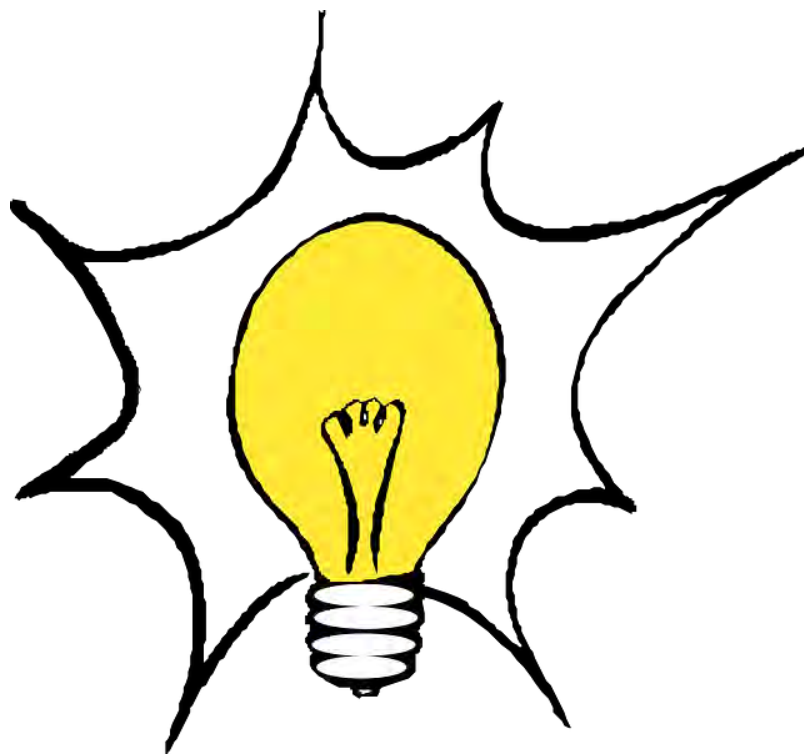
Youku触屏版的问题



- Banner 上有个隐藏的gif loading 动画导致浏览器不断进行绘制，栅格化和合成。浪费手机性能和能耗
- 不建议在终端上使用 GIF 动画

TIPS: Rasterize 栅格化，是高ppi分辨率手机都会遇到的问题。而且有部分手机都无法硬件去完成这个工作。所以浏览器采用的多线程软件栅格化。

- GPU Hack 好用，但是不是银弹

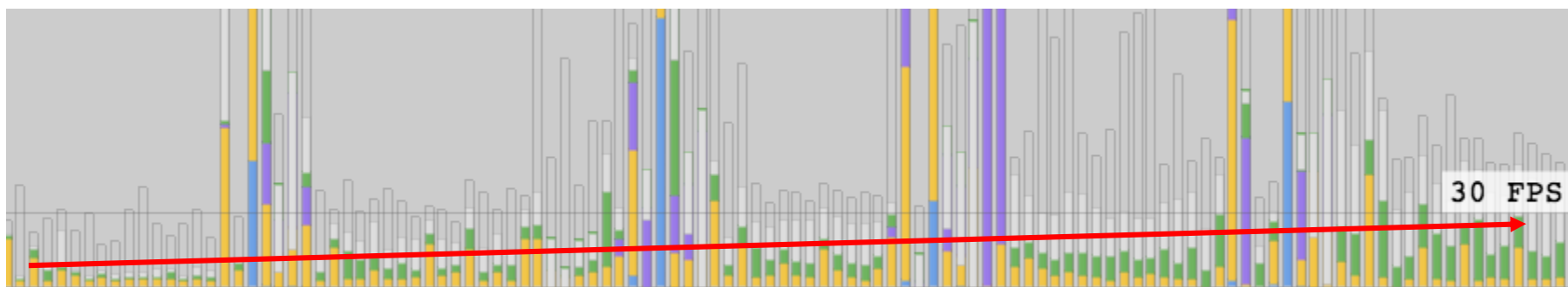


触屏版空间的滚动优化

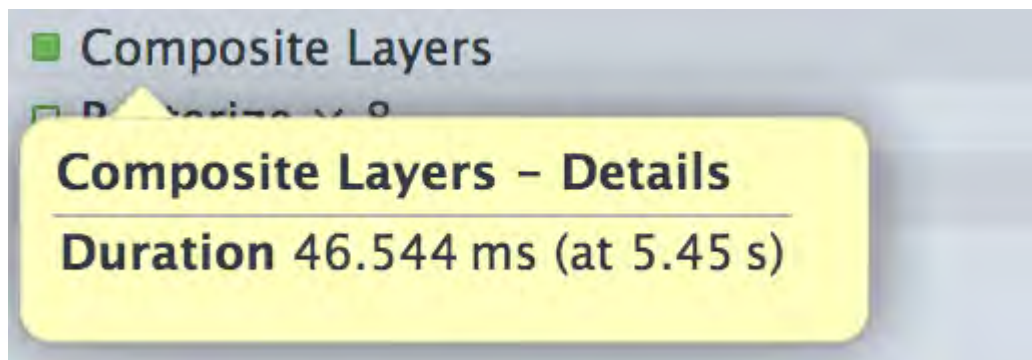


触屏版空间的滚动优化

- 空间触屏版每加载一屏feeds。
滚动feeds的效率下降 7%~10%

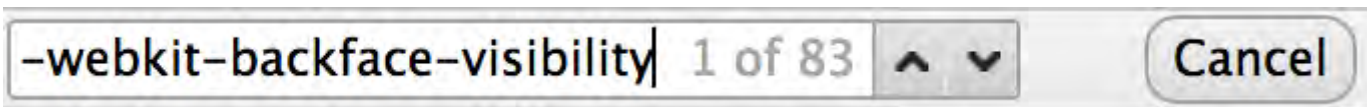


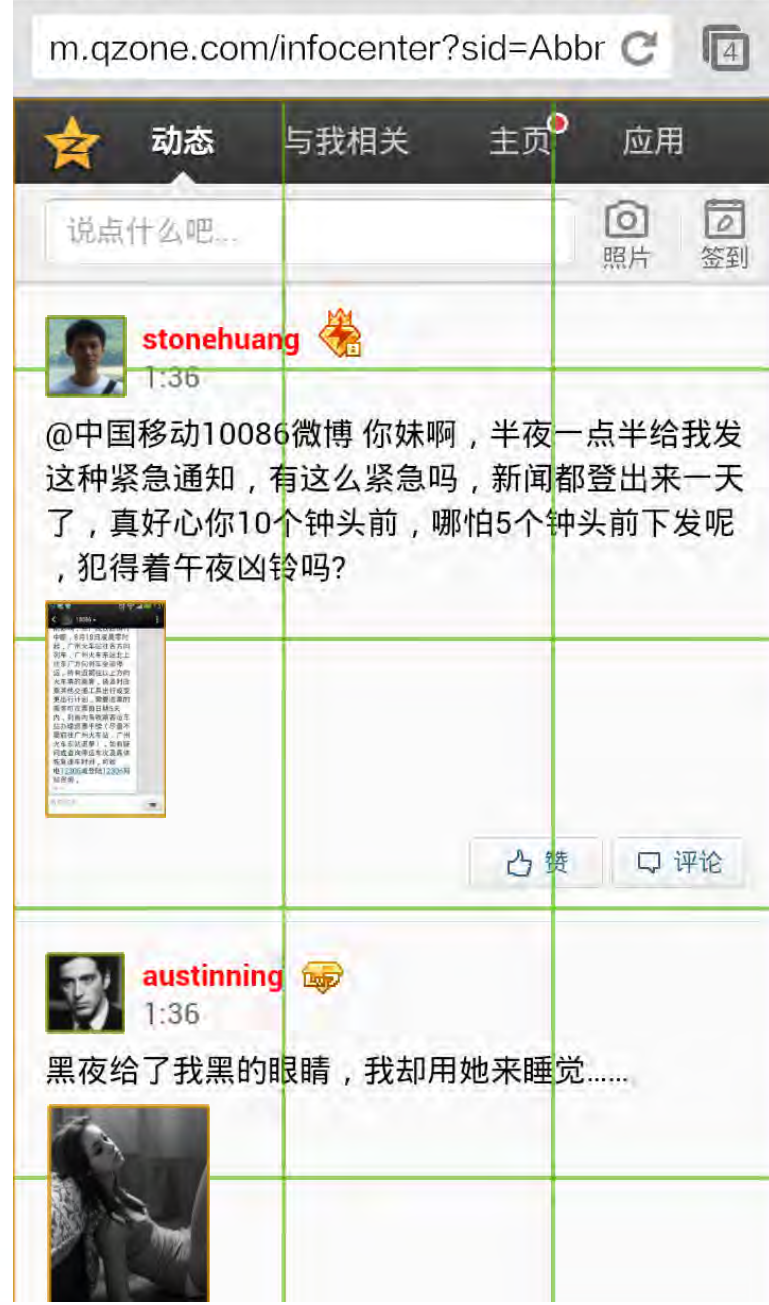
- 看图浮层移动图片，每次composite超过16ms



触屏版空间的滚动优化

- 加载了5屏的Feeds后，产生了非常大量的GPU层缓存。







TIPS: chrome devtools 中可以开启“Show composited layer borders”查看图片纹理素材

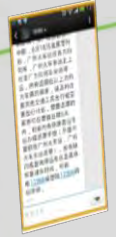
m.qqzone.com/infocenter?sid=Abbr

动态 与我相关 主页 应用



说点什么吧... 照片 签到

 **stonehuang** 
1:36

@中国移动10086微博 你妹啊，半夜一点半给我发这种紧急通知，有这么紧急吗，新闻都登出来一天了，真好心你10个钟头前，哪怕5个钟头前下发呢，犯得着午夜凶铃吗？



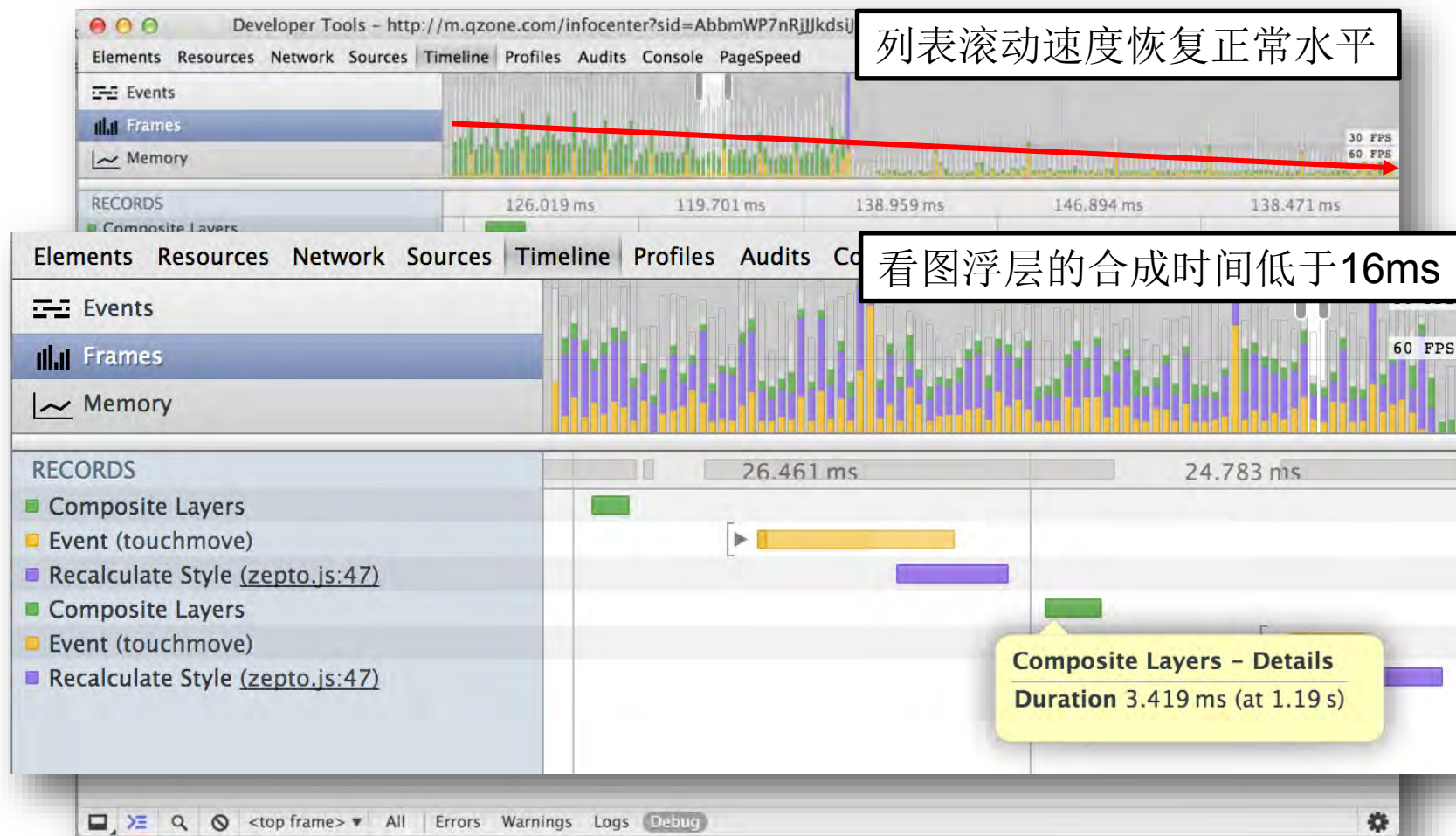
赞 评论

 **austinning** 
1:36

黑夜给了我黑的眼睛，我却用她来睡觉.....



触屏版空间的滚动优化



合理使用 GPU 加速

- 使用 GPU 加速, 其实是利用了 GPU 层的缓存, 让渲染资源可以重复利用。
- GPU 加速是把双刃剑, 过多的 GPU 层同样会带来性能开销, 请留意 Composite Layouts 是否超出了 16ms。
- 只在用户行为和动画需要的场景去创建 GPU 层, 但是需要及时回收。

平滑的动画

- 需要被脚本控制的动画，合理利用GPU缓存，减少Painting
- 使用 CSS transition 完成一次性动画
- 动画过程避免布局渲染

不同位移变化带来的渲染和绘制的运算量

	Left	translate2D	translate3D
Timeout	$(\text{Recalculate Style} * n + \text{Layout} * n + \text{Paint (size)} + \text{Composite Layers}) * \text{次数}$	$(\text{Recalculate Style} * n + \text{Layout} * 2 + \text{Paint (size)} + \text{Composite Layers}) * \text{次数}$	$\text{Paint (size) for cache} + (\text{Recalculate Style} * n + \text{Composite Layers}) * \text{次数}$
RAF	$(\text{Recalculate Style} + \text{Layout} + \text{Paint (size)} + \text{Composite Layers}) * \text{次数}$	$(\text{Recalculate Style} + \text{Layout} + \text{Paint (size)} + \text{Composite Layers}) * \text{次数}$	$\text{Paint (size) for cache} + (\text{Recalculate Style} + \text{Composite Layers}) * \text{次数}$
CSS3	$(\text{Recalculate Style} + \text{Layout} + \text{Paint (size)} + \text{Composite Layers}) * \text{次数}$	$\text{Recalculate Style} * 2$ $\text{Paint (size)} * 2$ $\text{Composite Layers} * 2$	$\text{Recalculate Style} * 2$ $\text{Paint (size) for cache}$ $\text{Composite Layers} * 2$

使用CSS3动画需要知道这些事情

- 单独使用 `translate2D` 并不会独立产生GPU层，不会在GPU中进行合成。
- CSS的补间动画配合`translate2D`，可以在GPU中产生一次临时的层以进行运算，但是遇到“布局计算”的变化则无效。
- CSS3动画通常是不被阻塞的。你可以获得独立的线程进行绘制

写在最后

- 避免重复更新渲染树
 - 16ms内最多出现一次Recalculate Style 和 Render Layout
 - 使用RequestAnimationFrame
 - Dom的批量操作读/写
 - 使用classList
- 绘制优化：使用GPU加速
 - 位置变化的 Render Layout 计算可以在GPU里完成
 - GPU使用也需要注意缓存回收
 - 一次性动画使用CSS transition 完成

16ms内还可以考虑其他事情

- CPU资源合理调度
- 避免内存回收
- 如果需要完全避免渲染树的计算，可以考虑Canvas



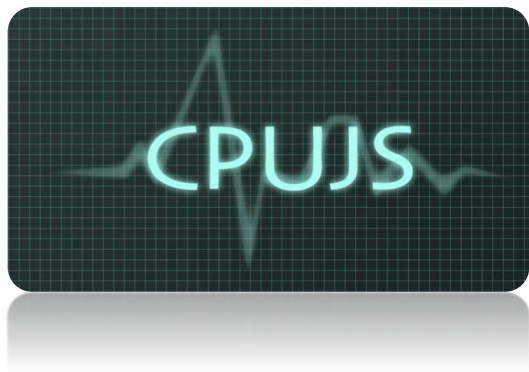


Thanks.

PPT 和 案例地址

- https://github.com/puterjam/speed_render

通过分析CPU工作状态进行渲染优化



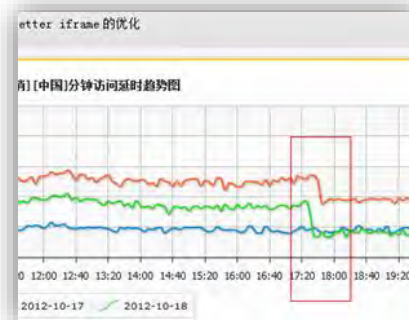
1. 监控功能

负责监控CPU繁忙时段



2. 统计功能

负责数据上报



3. 任务调度

根据当前CPU开销数据进行智能判断是否进行下一个任务

