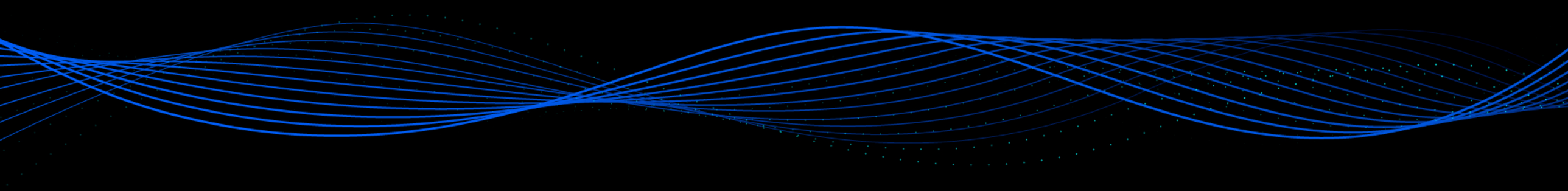
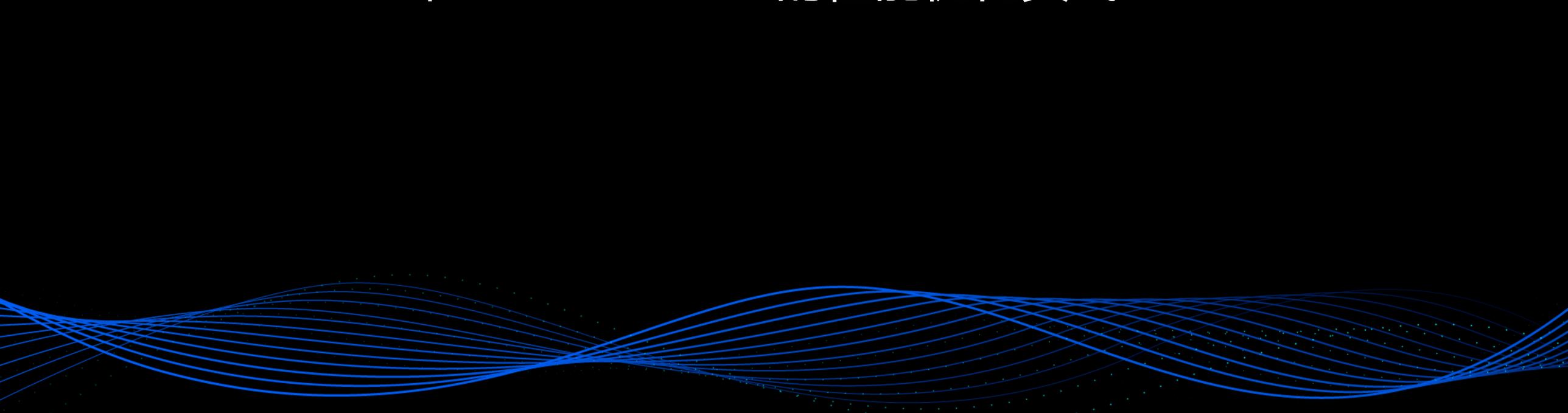
H5动画在移动平台上的性能优化实践

今日头条 张祖俭



大纲

Part 1. H5动画 在移动平台上的性能问题
Part 2. 解决思路—从浏览器渲染入手
Part 3. 在H5Animator上的性能优化实践



Part 1：CSS动画的问题

流畅的动画

- 60 fps最完美，30fps~60fps感觉流畅
- 30fps以下能感受到卡顿
- fps大于60，则超越了人眼能感知的刷新频率

动画实现方式

- 动画实现方式：
  - setTimeout：jQuery.animate()，div.style.left=“xxx”
  - Css transition /animation：目前最常用
  - requestAnimationFrame
  - web-animation
  - Canvas、WebGL

CSS动画的优点和问题

- 优点：
  - 易编写、语义化、功能强大
  - 可以利用GPU加速渲染
- 缺点：
  - 低端手机容易遇到性能问题：卡顿、掉帧
  - 不能精确控制每一帧的行为

解决CSS动画性能问题的思路

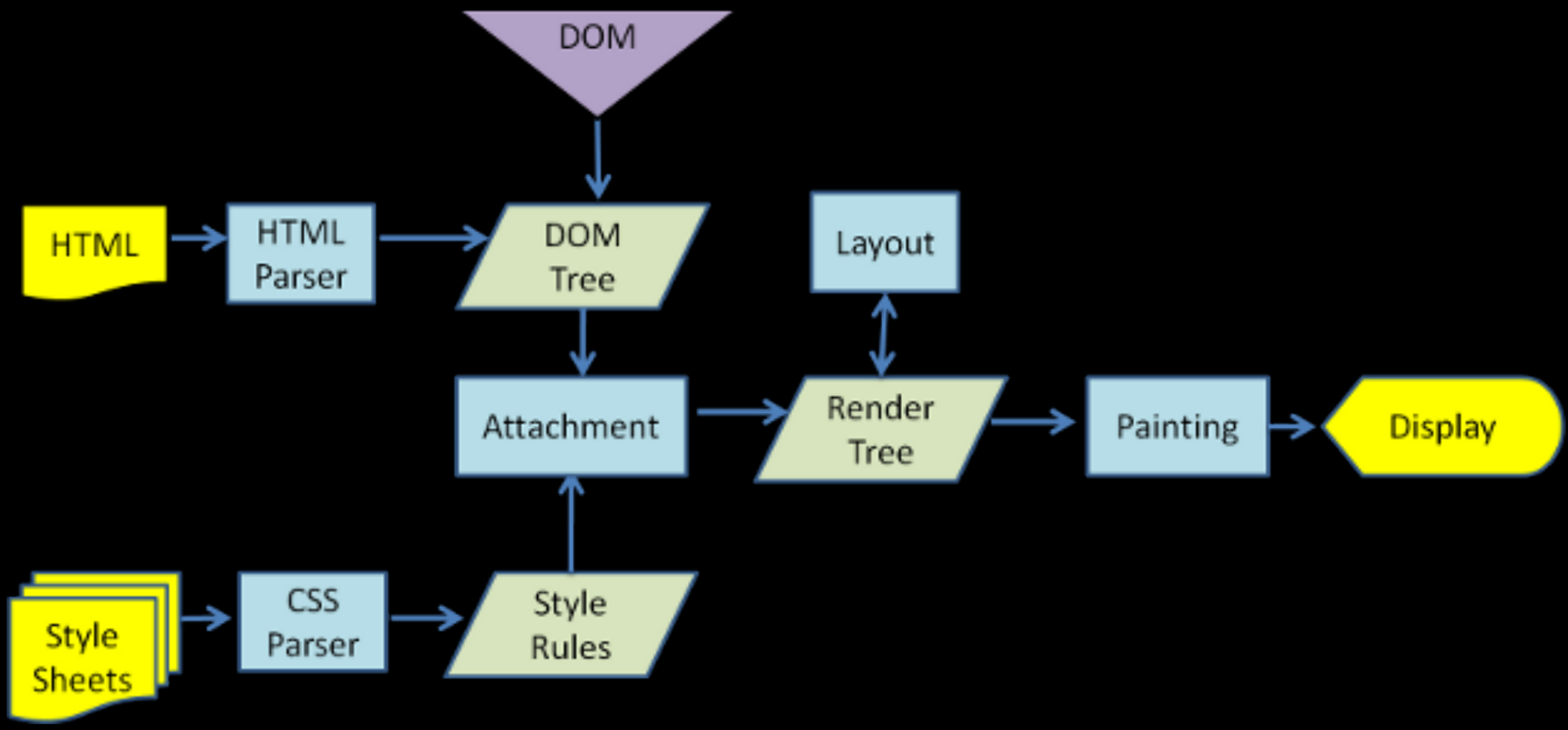
- 原理：理解webkit渲染过程
- 工具：chrome dev tools
- 实践：检验真理的唯一标准

Part2：浏览器渲染原理

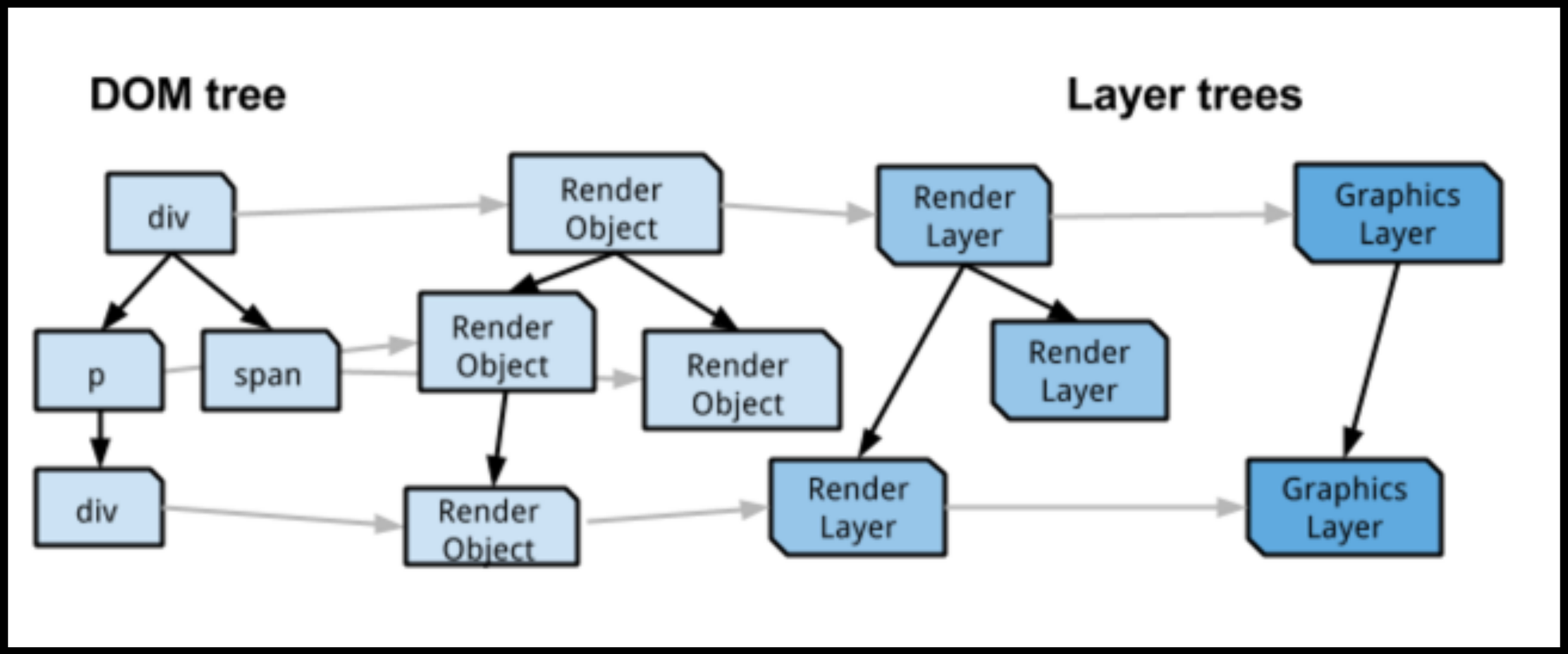
Blink的线程与进程

- Blink主要线程
  - MainThread：也叫RenderThread或者WebCoreThread 负责页面解析、布局、绘制、运行JS
  - Compositor线程：合成线程，负责将RenderLayer树的内容最终绘制(drawing)到屏幕上，也称为 Impl Thread。
  - IO Thread：负责网络、IPC等
  - UI Thread: 接受用户输入
- Chrome桌面版多进程；Android单进程

Webkit核心工作过程



树的转换过程



RenderObject Tree

- DOM 中的每个可视化节点对应一个RenderObject（或称LayoutObject，所有RenderObject组成一棵RenderObject Tree，也就是渲染树

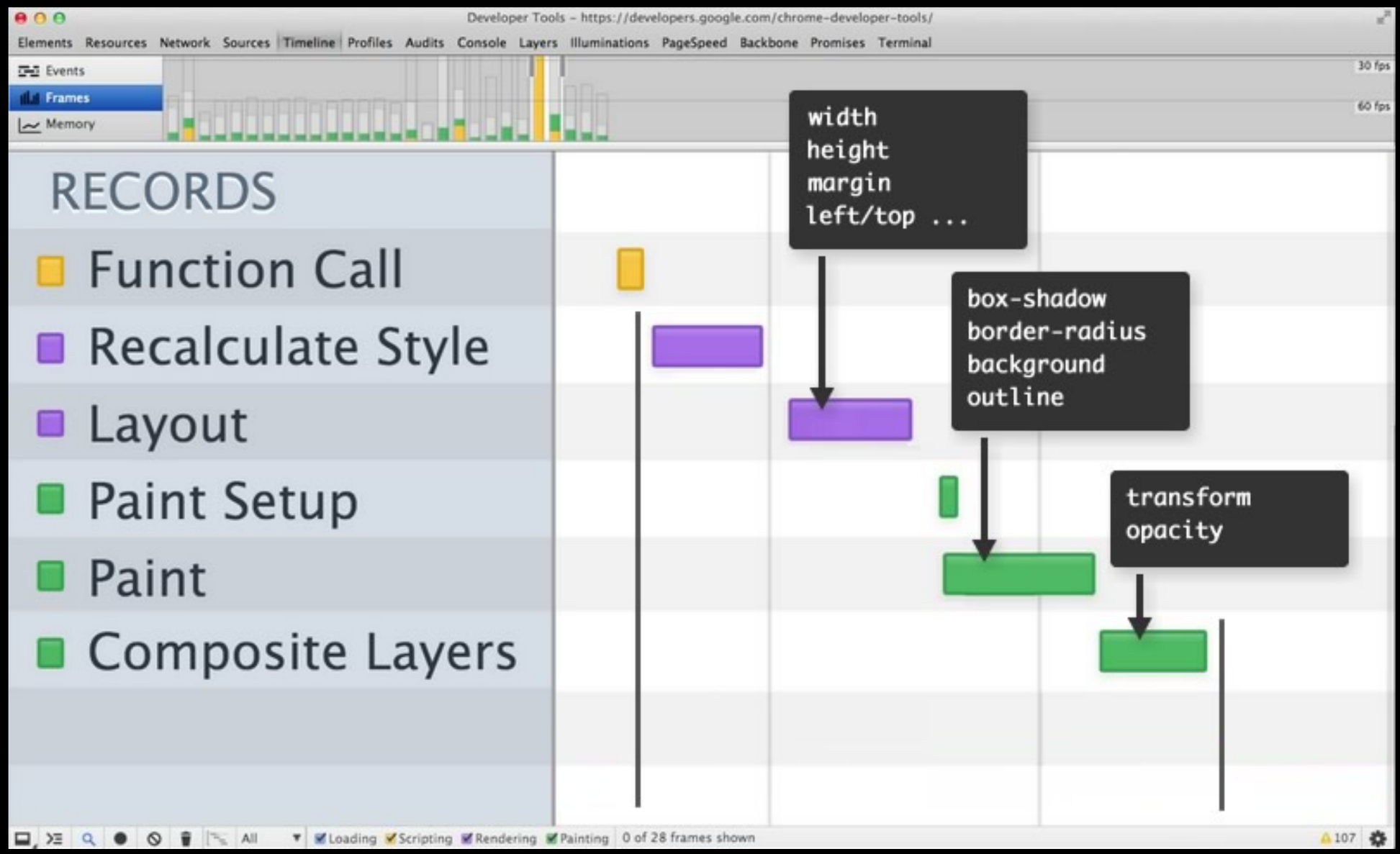
```
class RenderObject{
  virtual void layout();
  virtual void paint(PaintInfo);
  virtual void rect repaintRect();
  Node* node;  //the DOM node
  RenderStyle* style;  // the computed style
  RenderLayer* containgLayer; //the containing z-index layer
}
```

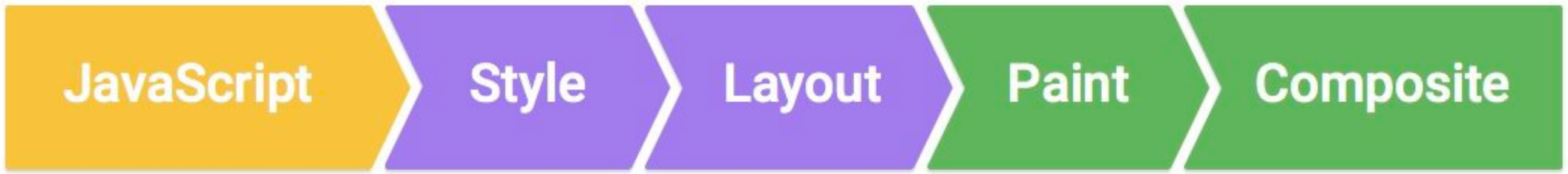
RenderLayer Tree

- 属于同一坐标空间的RenderObject属于一个RenderLayer，绘制到同一层。所有RenderLayer组成一棵RenderLayer Tree。

- RenderObject形成RenderLayer的条件
  - CSS position属性：relative、absolute、transform
  - 节点是透明的
  - 有overflow、alpha mask、reflection、filter等CSS属性
  - 根元素、2D Canvas、WebGL、Video元素

Blink渲染过程





Layout & Paint

- Layout是计算RenderObject Tree中各RenderObject的大小和位置的过程。Layout开销较大
- Paint：遍历RenderLayer Tree将元素内容画到各个Bitmap的过程。Paint是一个开销巨大的过程。

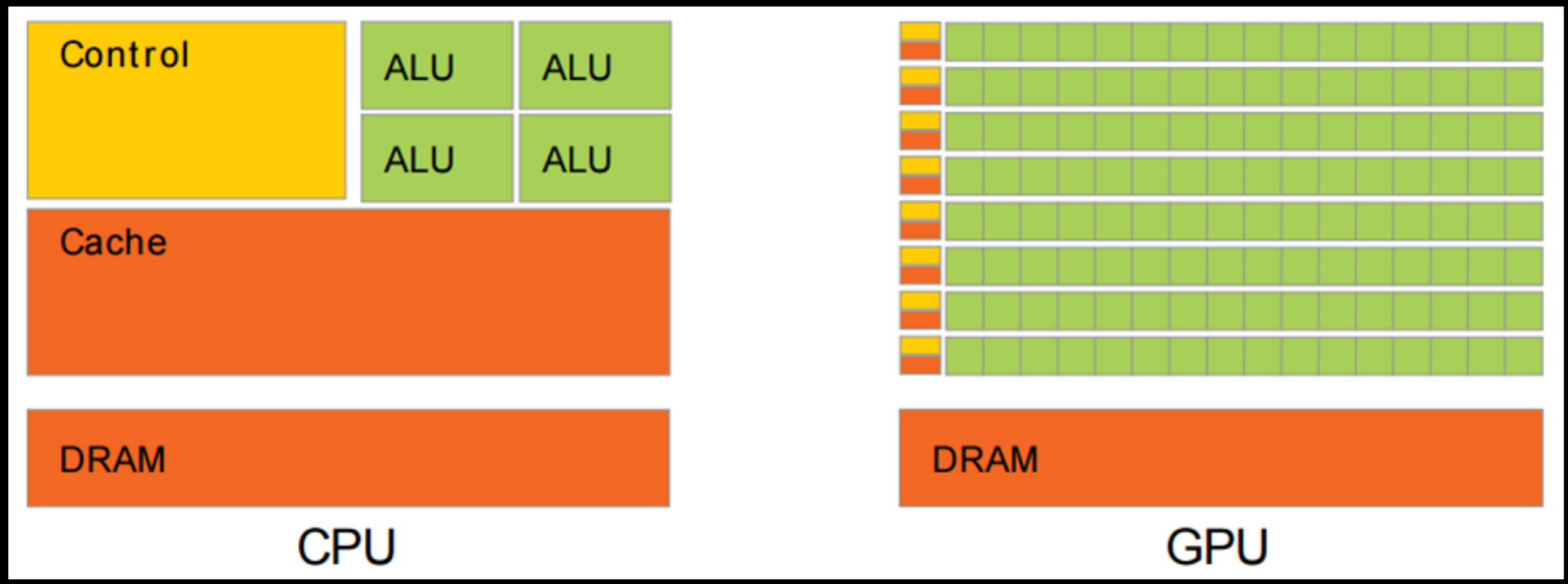
re-layout & repaint

- 触发re-layout：
  - 修改盒模型（大小、位置）、字体相关的属性
  - 读取位置大小也能触发layout。 包括clientXXX, getBoundingClientRect(), getClientRects(), innerText, offsetXXX, outerText,scrollLeft, scrollTop, scrollWidth 、 focus().....
- 触发repaint：修改border-radius, box-shadow, color、background-color等展示相关属性时

Drawing--Compositing

- GPU Accelerate Drawing
- Benefits VS overhead

GPU Accelerate Drawing



- 硬件加速的主要原理，将CPU不擅长的图形计算转换成GPU专用指令，由GPU完成。

Compositing Details

- 每个RenderLayer关联一个Graphic Layer ,也称为Compositing Layer。其包含的Graphic Context，负责具体的绘制工作。
- Graphic Layer绘制的结果(Bitmap)上传到GPU作为Texture。
- Compositor Thread利用GPU对Texture合成并绘制（Drawing）到屏幕上
- GPU可以快速对texture执行缩放、偏移、旋转、修改透明度等操作
- 在用户滚动的时候，Compositing Thread可以立刻对Texture执行偏移操作，从而快速响应用户动作

RenderLayer→Composited Layer

- 有3D transform或透视变换(perspective transform) 属性
- 对 opacity 、transform属性做 CSS 动画
- 使用加速视频解码的 <video> 元素
- 拥有 3D (WebGL) 上下文或加速 2D 上下文的 <canvas> 元素
- CSS Filter元素
- 一个RenderLayer包含一个子RenderLayer，子RenderLayer有自己的复合层
- RenderLayer在复合层上面时

Compositing : benefits

- 下列操作会跳过repaint，由GPU对缓存的Texture直接做变换，实现高性能渲染
  - 元素做3D Transform变换，改变其opacity、filter等属性
  - 网页滚动
  - WebGL, hardware video decoding
- CPU和GPU 并行运行不同任务，提高效率
- 分块渲染，局部更新

Compositing : overhead

- 计算开销：
  - 维护composited layer tree需要时间
  - 多个Layer 层叠计算量大
- 存储开销：需要系统RAM和GPU VRAM
- 传输开销：CPU上传到GPU耗费一定时间
- Composited Layer太多开销巨大，严重影响性能

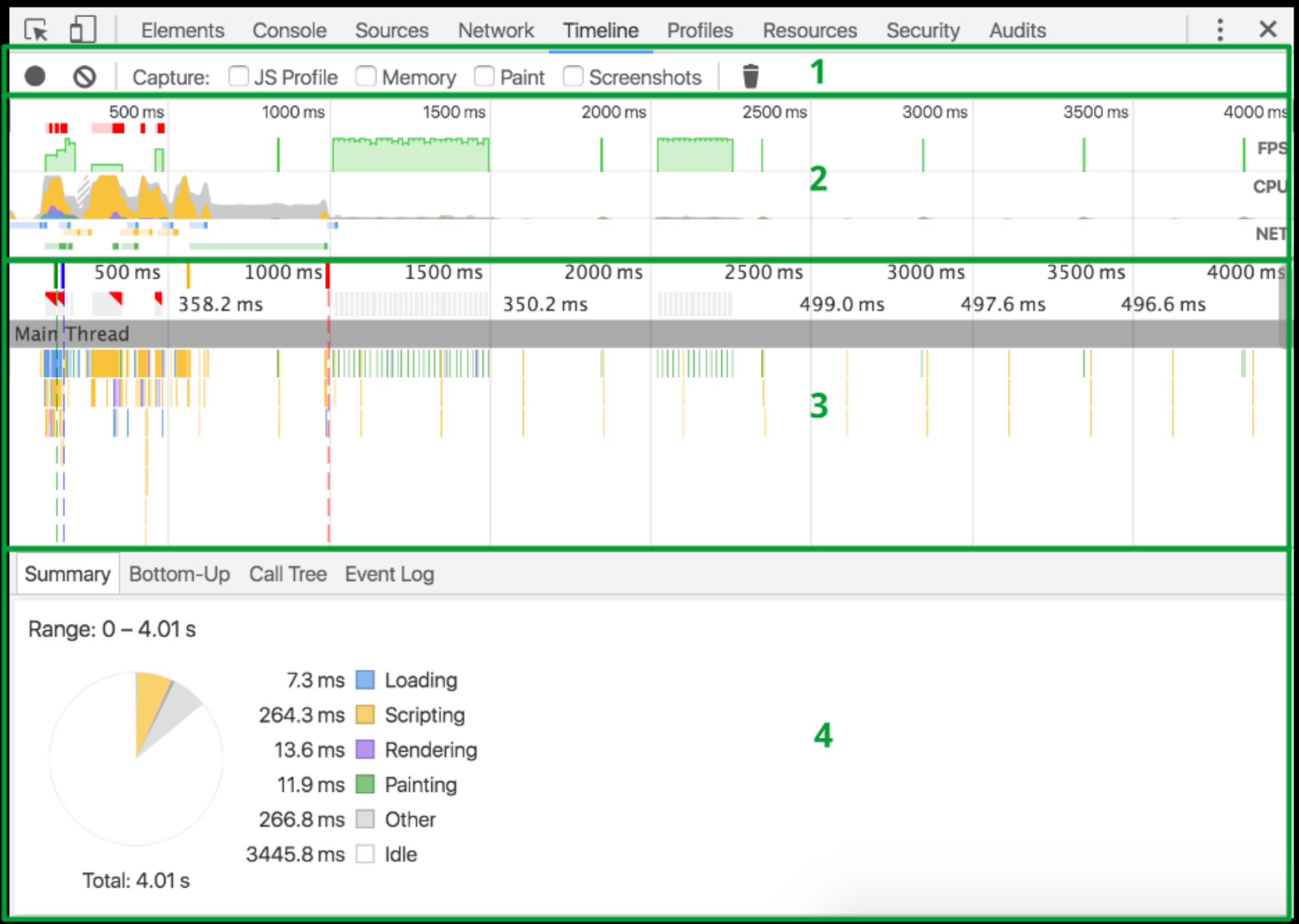
CSS动画优化路线图

- Layout and Paint is expensive，Compositing is cheap
- Layout->Paint->Compositing ,触发阶段越靠后越好，开销越小

Part 3：性能优化实践

- 以H5Animator为例 http://slide.toutiao.com

Chrome DevTools—性能分析神器



使用translate实现滚动

- Translate实现滚动不会触发re-layout和repaint，只执行compositing
- 滚动时解绑touchmove、touchend

```
element.style {
    height: 2657px;
    transform: translateY(-163px)
        translateZ(0px);
}
```

z-Index会导致意外Layer

- 问题背景：利用z-index对元素做层叠，层叠元素有动画，发现在低端Android手机上卡顿现象明显。

- 原因：在Composited Layer上面的RenderLayer也形成Composited Layer，Layer太多有性能问题。

- 解决方法：不使用z-Index，在生成DOM时利用元素的先后顺序来达到层叠的目的。

避免在切换动画之间做耗时操作

- 具体例子：在翻页的时候渲染图片Node，导致翻页卡顿
- 原因：图片的decode会耗费较多的时间，layout、paint也耗费时间
- 解决办法：利用空闲时间来做，在翻页结束后的空闲阶段插入下一页的图片Node；

使用opacity做元素显示与隐藏

- 问题背景：做动画时需要在动画开始前隐藏图片，动画开始时显示
- 历程：display→visibility→opacity
- 解释：使用opacity可以避免layout和paint

避免过大的Composited Layer

- 过大的Composited Layer导致每次改变小的部分时会重新绘制整个层，开销较大。
- Bad Case：给body加transform 3D属性

避免交替读写布局属性引起re-layout

- 交替读写布局属性会引起立即re-layout。
- 例子：一组DOM元素要将其高度和宽度设置成一样。
不好的做法：

```
for(var i = 0,len = divs.length; i<len; i++){
    var width = divs[i].clientWidth;
    divs[i].style.height = width + 'px';
}
```

- 解决方案：分离读写操作，批量执行，利用requestAnimationFrame推迟读或者写的批量执行

```
for(let i = 0,len = divs.length; i<len; i++){
    let width = divs[i].clientWidth;
    requestAnimationFrame(()=>{
        divs[i].style.height = width + 'px';
    })
}
```

尽量避免gif图片

- Gif图片会引起paint，尽量少用gif图
- 即使Gif 图被覆盖，仍然会paint
- 如果 必须使用gif图，在gif图不显示时要将其visibility设为none。

合理使用will-change

- 使用will-change属性可以让webkit可以提前做一些优化。例如：

  - .box {
    will-change: transform, opacity;
    }

- 合理使用：在动画即将开始前200ms加上这个属性

- 不要过度使用这个属性，否则会浪费浏览器资源

合理使用touch-action

- 背景：禁止某些元素触发翻页行为
- 不好的做法：event.preventDefault()
- 优化：使用touch-action:none阻止touch的默认行为

减少DOM层级、减少css代码

- DOM层级越多，解析和渲染代价越大；复杂容易引起预料之外的re-layout和repaint
- 无谓的CSS也会有一定的开销。例子：

```
@keyframe fade-out
{
  0%:{
        opacity:1;
        transform:translate(0,0) scale(1,1) skew(0,0) rotate(0);
    }

  100%:{
        opacity:0;
        transform:translate(0,0) scale(1,1) skew(0,0) rotate(0);
    }

}
```

THANKS!