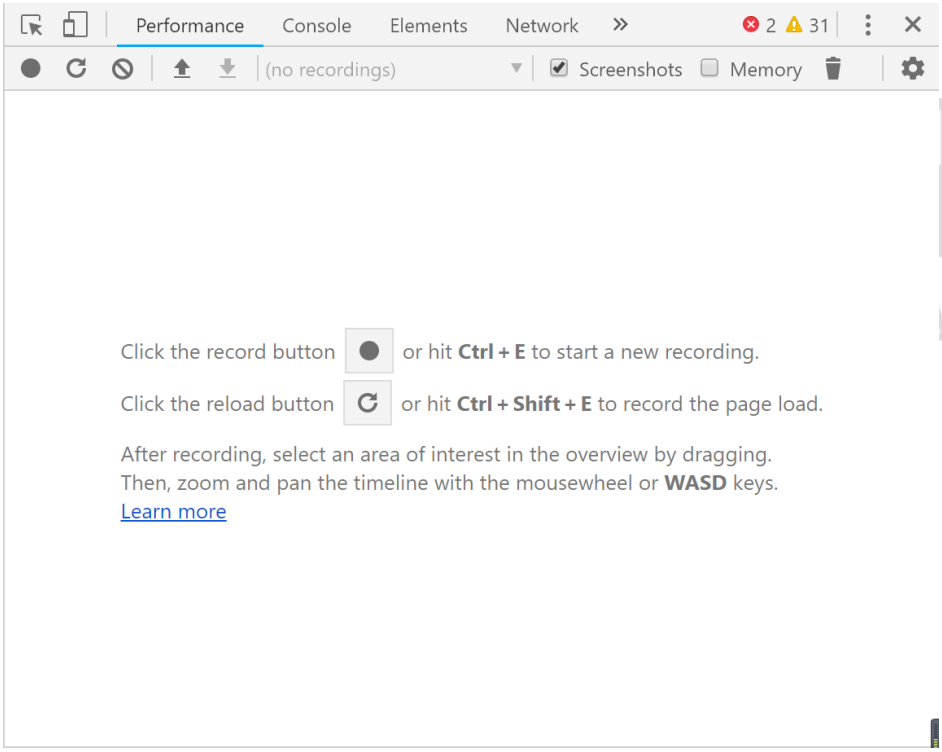
前端性能分型工具介绍（Performance）

运行时性能表现（runtime performance）指的是当你的页面在浏览器运行时的性能表现，而不是在下载页面的时候的表现。这篇文章将会告诉你怎么用 Chrome DevTools Performance 功能去分析运行时性能表现。在 RAIL 性能评估模型下，你可以在这篇文章中可以学到怎么去用这个 performance 功能去分析 Response, Animation, 以及 Idle 这三个性能指标。这里使用 Google 官网提供的 DEMO 了进行分析的。大概看了一下该网站用例，用例是通过修改蓝色图标的 top 值来完成动画效果。其优化是将 offsetTop 值进行了缓存，以避免频繁读取引发的回流。但是优化后，仍然是修改元素的 top 值来完成动画，此处还可以使用 translate 来代替 top 来进行优化。

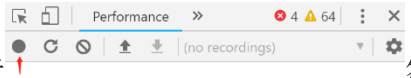
接下来看看我们要说的工具 Performance

先看看基本的页面是什么样子的？

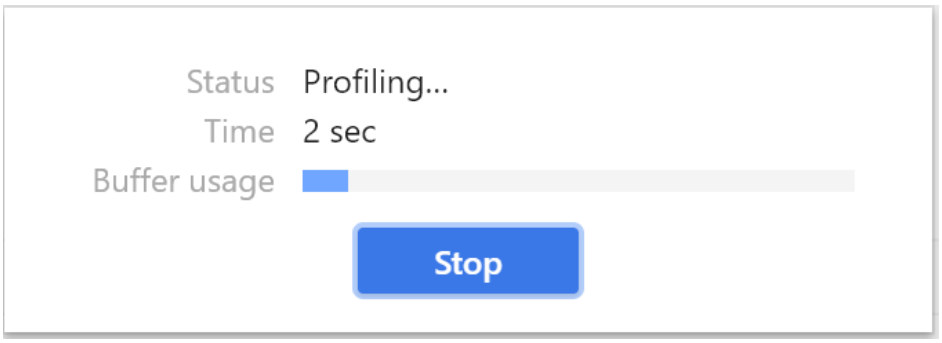


什么都没有该怎么使用呢？

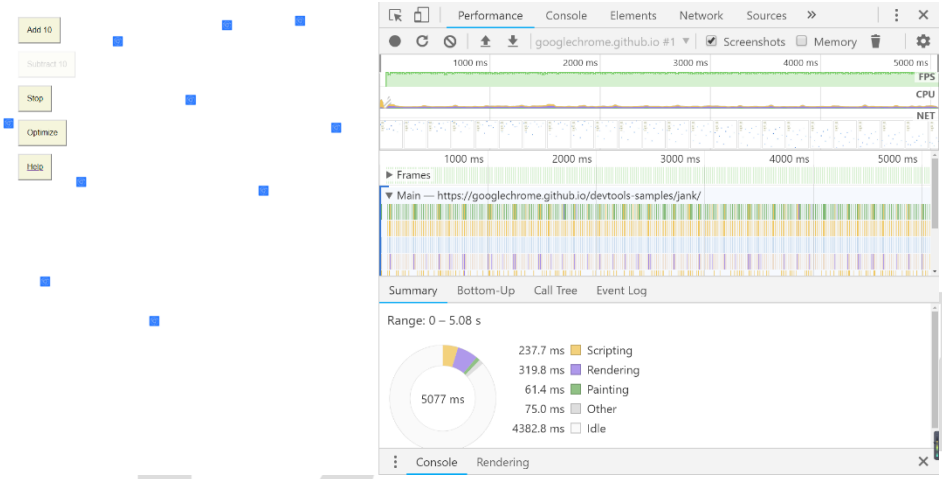
两个按钮分别用来记录页面运行时的性能，另一个时表示页面加载时的信息记录。

通过点击 

然后经过几秒钟中后点击 stop 像这样



记录之后会看到一个这样的图（内容非常的多）：

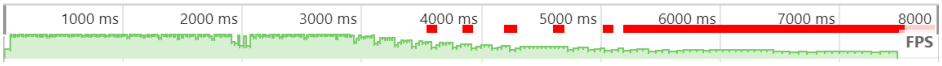


这里面都有什么呢？

- 工具栏
- FPS 图表
- GPU 图表
- NET 图表
- Network 图表
- Frames 图表
- Main 图表
- Frame 图表
- Raster 图表
- CPU 图表
- Summary 面板

1、FPS 图表

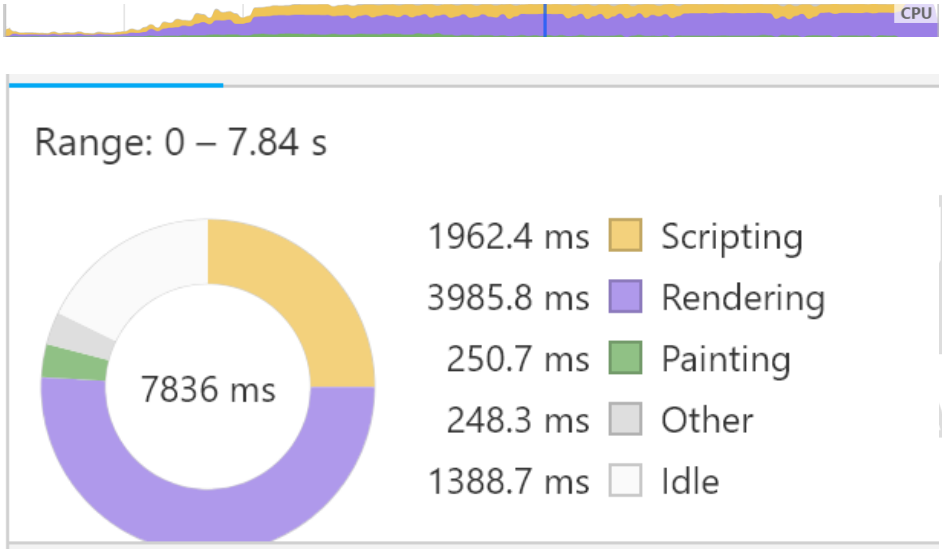
FPS（frames per second）每秒帧数，是来衡量动画的一个性能指标，该指标若能保持在 60，就能带来不错的用户体验。若出现了一个红色的长条，那就说明这些帧存在严重的问题。



想这个后面的红色一样，说明存在严重的性能问题。

2、CPU 图表

CPU 图表中的颜色和面板底部的 Summary 面板中的颜色是匹配的。每种颜色分别代码一种处理。颜色条越长，表示 CPU 在该处理上所花费的时间就越长。



各颜色含义：

- 蓝色(Loading)：网络通信和 HTML 解析
- 黄色(Scripting)：JavaScript 执行
- 紫色(Rendering)：样式计算和布局，即重排
- 绿色(Painting)：重绘
- 灰色(other)：其它事件花费的时间
- 白色(Idle)：空闲时间

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
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
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
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

3、NET

NET 中每条横杠表示一种资源，横杠越长，表示请求资源所需的时间越长

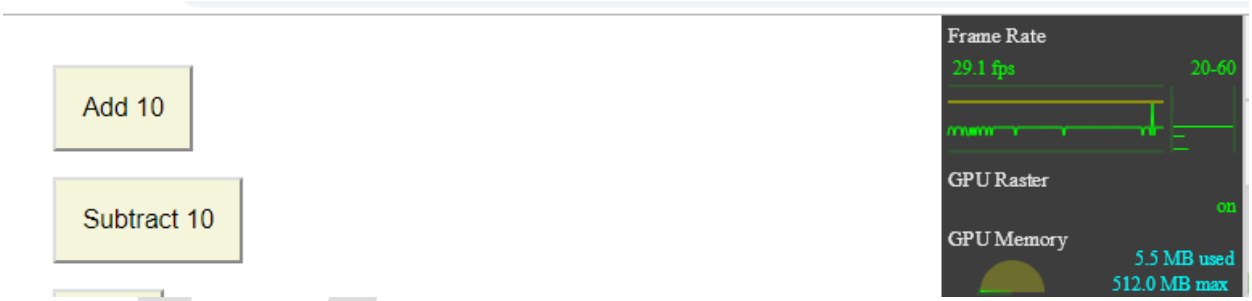
4、Frames

在 Frames 部分，如果将你的鼠标移动至绿色方块部分，会显示在该特定帧上的 FPS 值，此例中每帧可能远低于 60FPS 的目标。

在这个例子中，页面的性能很差并且能很明显地被观察到，但是在实际场景中，可能并不是那么的容易，所以，要用所有这些工具来进行综合测量。

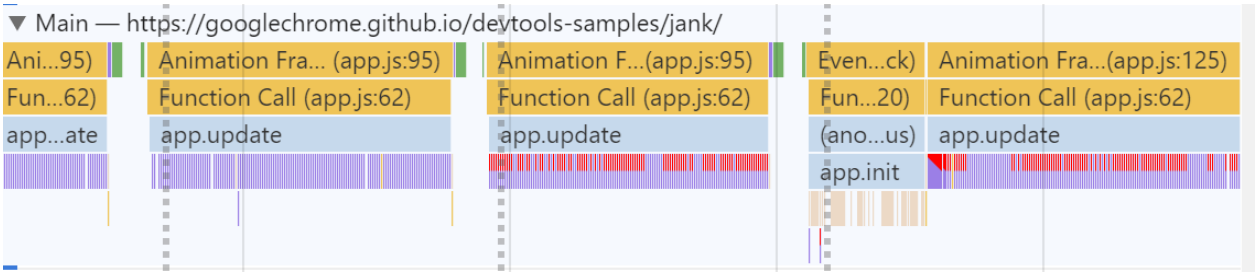
实时显示 FPS 面板：more tools -> Rendering -> 在 Rendering 面板打开 FPS Meter。

like this.



5、Main

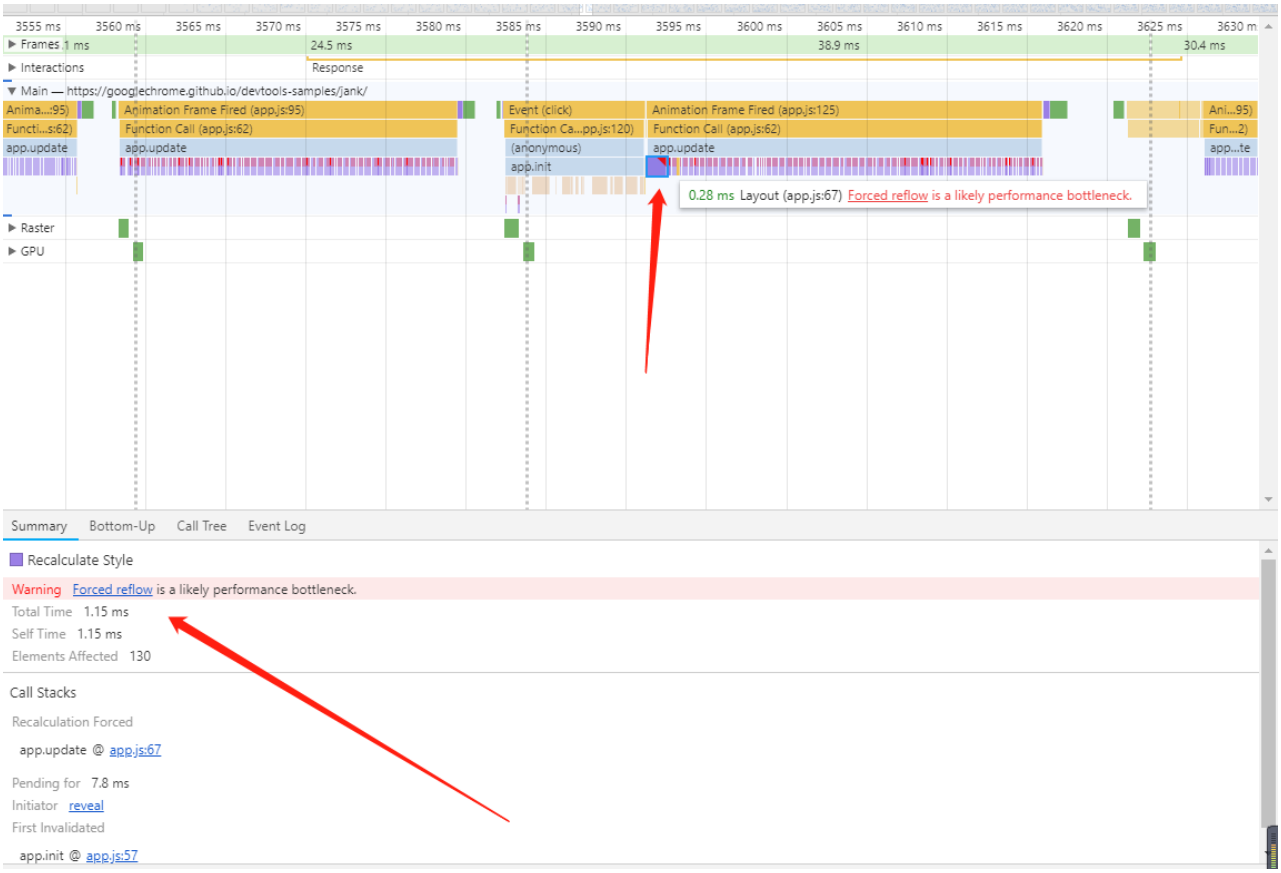
展开 Main 部分，DevTools 将显示主线程上的随着时间推移的活动火焰图。x 轴代表随时间推移的记录，每个长条代表一个事件，长条越宽，代表这个事件花费的时间越长。y 轴代表调用堆栈，当你看到堆叠在一起的事件时，意味着上层事件发起了下层事件。



可以通过单击、鼠标滚动或者拖动来选中 FPS,CPU 或 NET 图标中的一部分，Main 部分和 Summary Tab 就会只显示选中部分的录制信息。注意 Animation Frame Fired 事件右上角的红色三角形图标，该红色三角图标是这个事件有问题的警告。

在 Summary 面板中，将会显示有导致问题的代码行号。比如点击 Initiator 的 reveal 或者 Initiator 下面的： app.js:67，点击将会跳转到对应的代码行。

通过上面的方法可以分析该例子中，由于 update 导致了回流，所以优化方法是将该值缓存，避免重复读取。



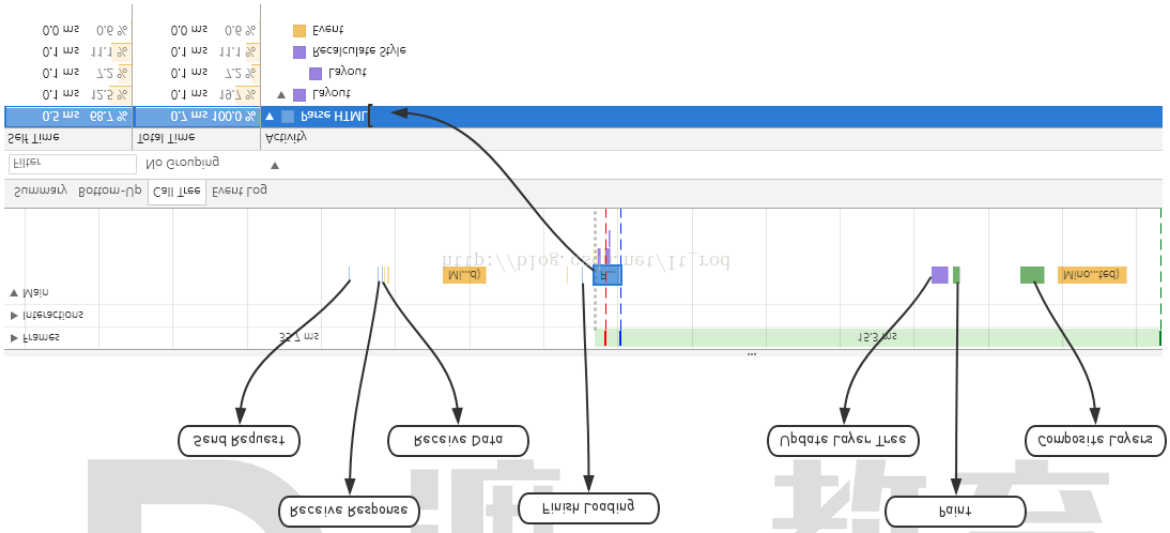
扩展：浏览器渲染的过程

这里使用一个简单的代码，配合着 Performance 工具来说明一下。

```
<!Doctye html>
<html>
<head>
</head>
<body>
 <div>
    Test dom load.
  </div>
</body>
```

</html>

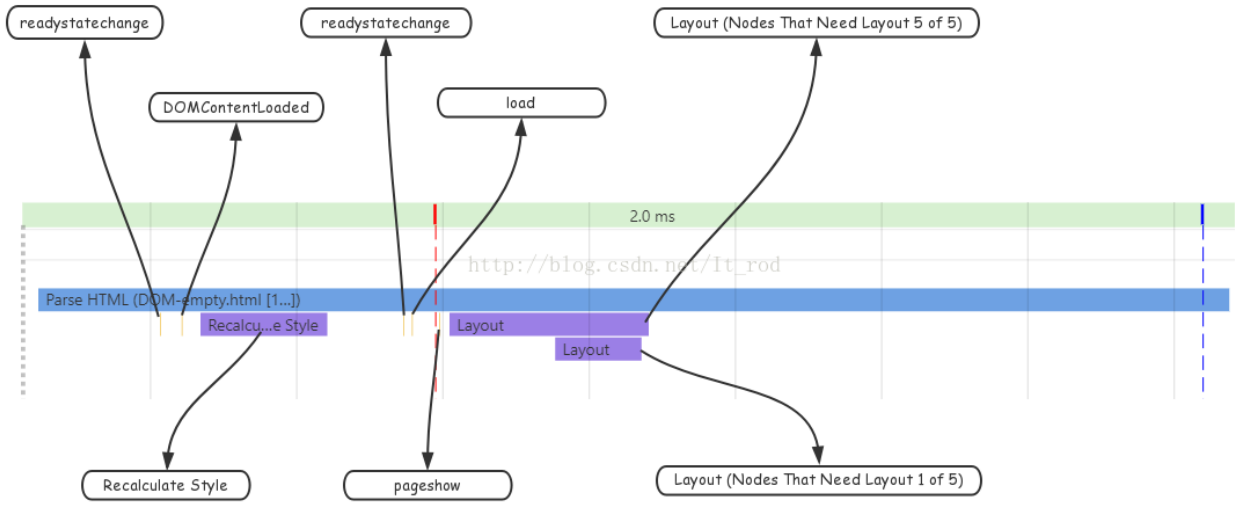
然后打开 chrome performence 查看页面的渲染过程：



Send Request　　发送网络请求时触发
Receive Response　　响应头报文到达时触发
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
Finish Loading　　网络请求完毕事件
Parse HTML　　浏览器执行 HTML 解析
Paint　　确定渲染树上的节点的大小和位置后，便可以对节点进行涂鸦（paint）
Composite Layers　　合成层；当渲染树上的节点涂鸦完毕后，便生成位图（bitmap），浏览器把此位图从 CPU 传输到 GPU

Parse Html（without css and js）

单纯看一下 Parse Html 的过程，现在讲 Parse Html 的过程放大来看：



然后我们按照时间线从左到右来看一下里面的事情（先说一下，这个图怎么看，下面的内容是上面内容的调用，这里也就是上面带图中带有文字说明的内容都是在 ParseHtml 的周期内调用的内容）：

readystatechange（第一个）

说这个的时候就需要说一个事件 DOM readystatechange，readyState 属性描述了文档的加载状态，在整个加载过程中 document.readyState 会不断变化，每次变化都会触发 readystatechange 事件。（可以访问查看例子）

readyState 有以下状态：

loading 加载 document 仍在加载。

interactive 互动文档已经完成加载，文档已被解析，但是诸如图像，样式表和框架之类的子资源仍在加载。

complete 完成 DOM 文档和所有子资源已完成加载。状态表示 load 事件即将被触发。

那么这里的事件执行的是哪一步呢？

这里执行的是 interactive。因为这个事件后面紧跟着的是 DOMContentLoaded 事件，而且如果你亲自访问这个页面去看一下，在 parseHtml 前面还有一次 readystatechange，那里应该是 loading 。

DOMContentLoaded （构建 DOM 树成功）

DOM 树渲染完成时触发 DOMContentLoaded 事件，此时可能外部资源还在加载。这里表示 DOM 树加载完成。

Recalculate Style（构建 CSSOM 树）

从文字的字面意义理解也就是重新计算样式。为什么是 Re-caculate Style 呢？这是因为浏览器本身有 User Agent StyleSheet，所以最终的样式是我们的样式代码样式与用户代理默认样式覆盖/重新计算得到的。这里也是在构建 CSSOM 树。

readystatechange（第二个）

从第一个事件的地方可有了解到这里执行的是 complete，表示页面上的 DOM 树和 CSSOM 树已经形成并且合并成 Render 树。此时页面上所有的资源都已经加载完成。其实从后面的 load 事件也可以看出来。

load 事件

所有的资源全部加载完成会触发 window 的 load 事件。

pageshow 事件

当一条会话历史记录被执行的时候将会触发页面显示(pageshow)事件。(这包括了后退/前进按钮操作，同时也会在 load 事件触发后初始化页面时触发)。

以下示例将会在控制台打印由前进/后退按钮以及 load 事件触发后引起的 pageshow 事件：

```
window.addEventListener('pageshow', function(event) {
console.log('pageshow:');
console.log(event);
});
```


Layout

将渲染树上的节点，根据它的高度，宽度，位置，为节点生成盒子（layout）。为元素添加盒子模型。上图中有两个 layout，二者之间的不同是 Nodes That Need Layout 1/5 of 5
这里的 5 代表应该是页面上的 5 个 node（文本内容是文本节点），但是对于这个 1，还是没有一个明确的说明，但是影响不大，毕竟还是属于 layout。

其实看到这里：我们想一下 ParseHtml 做了哪些内容：

构建 DOM 树 -> 构建 CSSOM 树 -> 构建 Render 树 -> 布局 layout

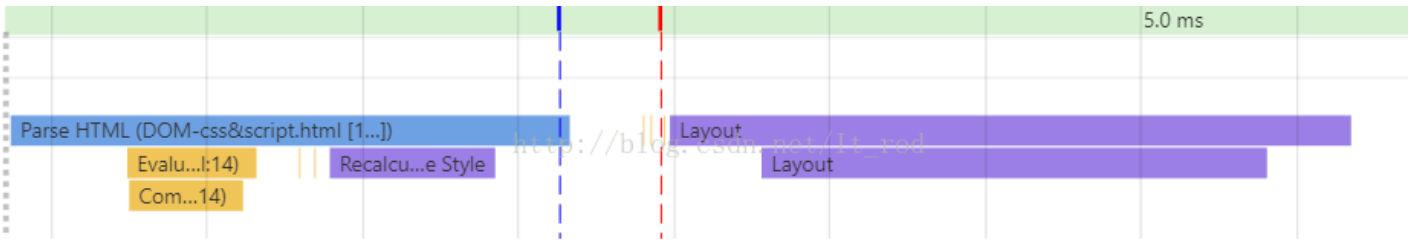
Note: 这里执行的情况是这样的，但是当我们加上内部 css 和内部 js 的时候这个步骤就有了不一样的变化。

3. Parse Html（with css and js）
还是先把代码贴出来（可以访问 DOM (with css and js)）：

```
<!Doctye html>
<html>
<head>
<style type="text/css">
.div {
color: blue
}
</style>
</head>
<body>
<div class='div'>
Test dom load.
</div>
<script type="text/javascript">
var a = 1 + 1;
</script>
</body>
</html>
```


然后以同样的方式将 chorme performence 的过程贴出来：

这里就只说和上面图中不一样的地方：



首先是多个两个黄颜色的 js 相关的内容，一个叫做 Evaluate Script（加载 js），另一个是 Compile Script（js 预编译处理，可以查看文章，这里也已经对 js 文件执行了）。

第二个不同的地方是从第二个 readystatechange 事件起一直到 layout 都已经不在 ParseHtml 内部完成了。这里我简单的去做了一个测试，只有 css 或者 js 存在的情况

下，还是和现在一样的结果，这里我假设是因为 css 或者 js 阻塞了整个页面的渲染过程，因为 js 和 css 都有可能对标签进行样式的设置，从而影响了 layout 的执行。那么现在 ParseHtml 就执行了：构建 DOM 树 -> 构建 CSSOM 树 -> 构建 Render 树。