Web 基础

—— 聊聊性能



周炼赤
智慧检务事业部
2019-08-20

CONTENTS

1 性能的重要性

2 RAIL 模型

3 页面加载

4 页面渲染

1

性能的重要性

一、性能的重要性

1、性能关乎用户的去留

- Pinterest 的搜索引擎流量和注册人数增长 15%，得益于其感知等待时间减少 40%。
- BBC 发现其网站的加载时间每增加一秒，便会多失去 10% 的用户。
- DoubleClick 发现，如果页面加载时间超过 3 秒，53% 的移动网站访问活动将遭到抛弃。
- DoubleClick 研究还表明，与加载时间约为四倍（19 秒）的网站相比，加载时间在 5 秒以内的网站会话加长 70%、跳出率下降 35%、广告可见率上升 25%。

一、性能的重要性

1、性能关乎用户的去留

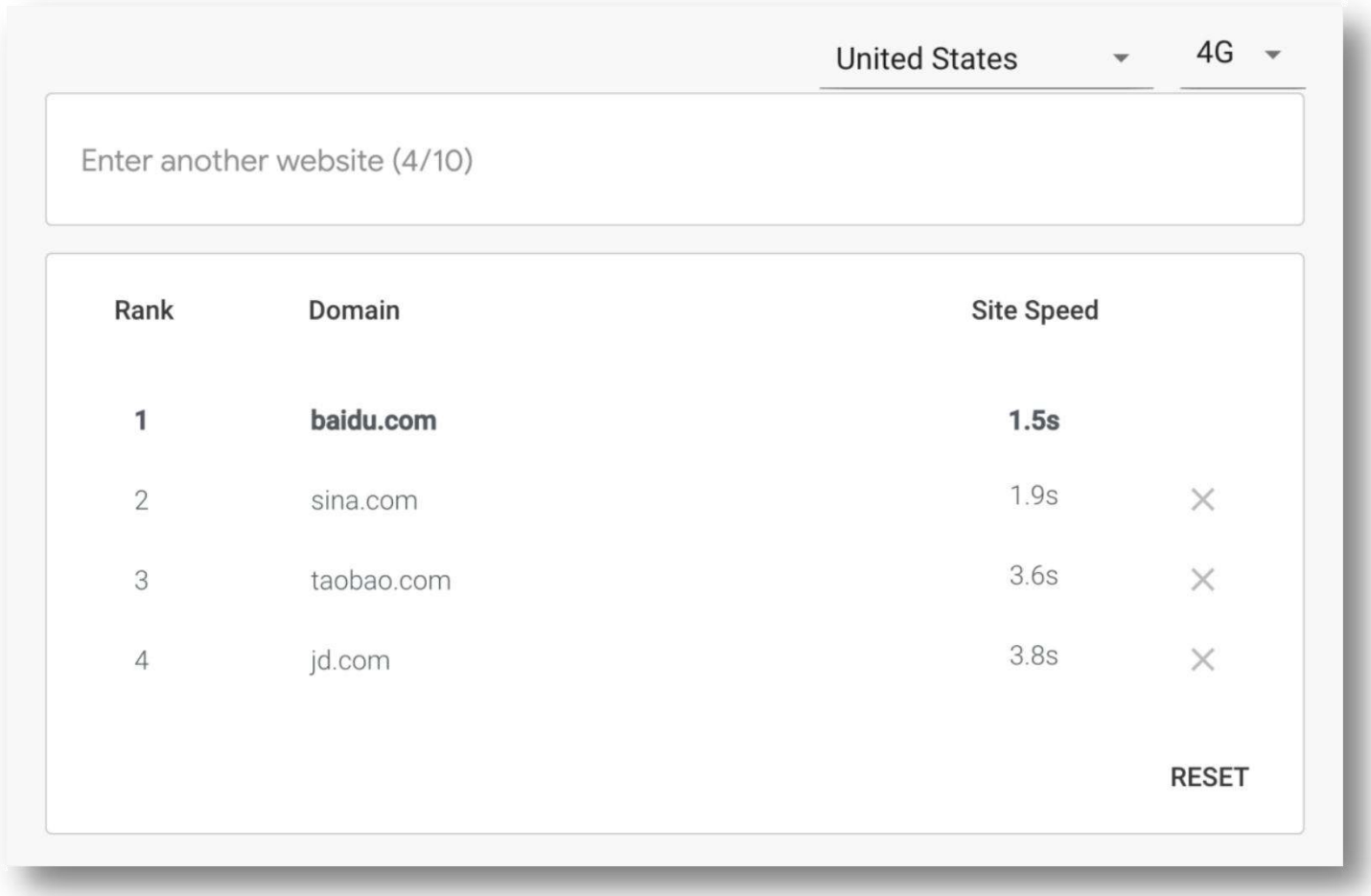


图1 通过 TestMySite 工具进行网站速度评估

一、性能的重要性

2、性能关乎转化率的提升

- 对于 Mobify，首页加载时间每减少 100 毫秒，基于会话的转化率增加 1.11%，年均收入增长近 380,000 美元。 此外，结账页面加载时间减少 100 毫秒，基于会话的转化率增加 1.55%，进而使年均收入增长近 530,000 美元。
- DoubleClick 发现，网站加载时间在 5 秒内的发布商比网站加载时间在 19 秒内的发布商的广告收入多一倍。
- AutoAnything 的页面加载时间减少一半后，其销售额提升 12-13%。

2、性能关乎转化率的提升

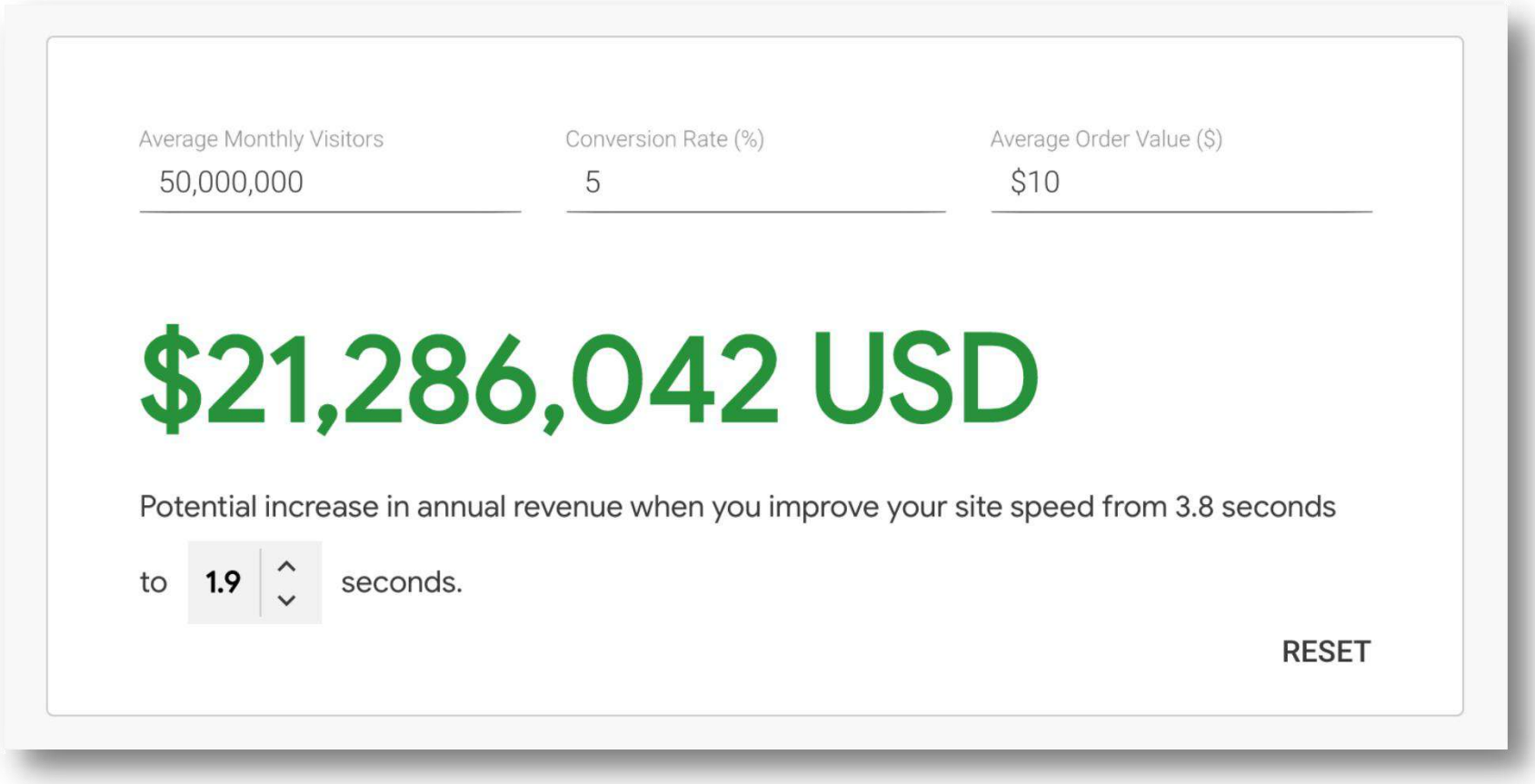


图2 利用 TestMySite 工具对 jd.com 进行网站效益评估

一、性能的重要性

3、性能关乎用户体验和用户成本

- 尤其是对于移动设备，其 CPU 和内存有限，一些未经优化的代码，可能我们认为并没有多大，但却常使其崩溃。
- 自 2011 年起，页面总大小就开始逐步增加，而这一趋势似乎将持续下去。 随着页面发送的数据增多，用户必须经常为其按流量计费，如此一来，用户的成本便会增加。
- 对于医院、消防、危机中心等公共机构，在紧急情况下「提供高效的数据通信和内容展示」变得尤为重要。

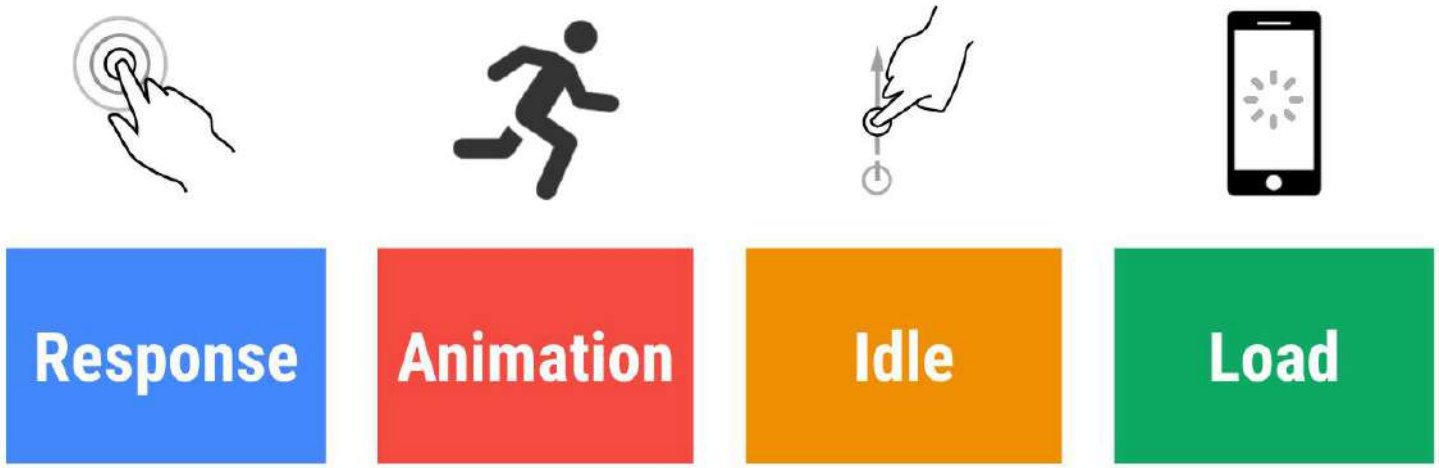
无论网页还是 App，当性能糟糕到需要被「忍受」的地步，用户必定会将其抛弃。

2

RAIL 模型

二、RAIL 模型

RAIL 是一种**以用户为中心**的性能模型，它从四个方面影响着性能：



- R：立即响应用户，在 50 毫秒以内处理事件。
- A：设置动画或滚动时，在 10 毫秒以内生成帧。
- I：最大程度增加主线程的空闲时间。
- L：加载时，在 5 秒以内呈现交互内容。

二、RAIL 模型

0、以用户为中心

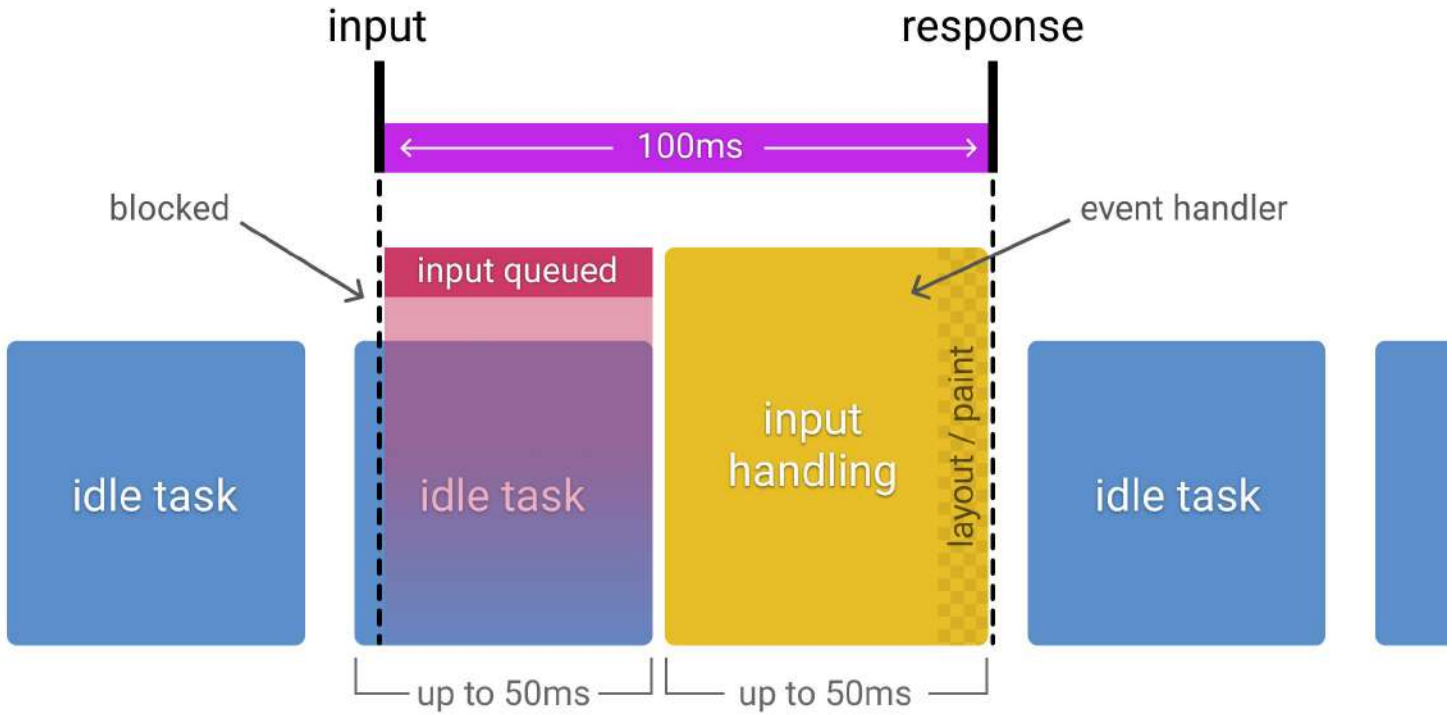
- 用户如何评价延迟：

延迟与用户反应	
0 - 16 毫秒	用户可以感知每秒渲染 60 帧的平滑动画转场。也就是每帧 16 毫秒（包括浏览器将新帧绘制到屏幕上所需的时间），留给应用大约 10 毫秒的时间来生成一帧。
0 - 100 毫秒	在此时间窗口内响应用户操作，他们会觉得可以立即获得结果。
100 - 300 毫秒	用户会遇到轻微可觉察的延迟。
300 - 1000 毫秒	在此窗口内，延迟可能会被认为是任务的一部分。
1000+ 毫秒	超过 1 秒，用户的注意力将离开他们正在执行的任务。
10,000+ 毫秒	用户感到失望，可能会放弃任务；之后他们或许不会再回来。

二、RAIL 模型

1、响应（Response）—— 事件处理时间不超过 50ms

- 在 50 毫秒内处理输入事件，以确保在 100 毫秒内能够产生一个直观的响应。适用于大多数输入，如按钮点击、表单切换、动画启动等。但不适用于触摸拖动或滚动。
- 对于需要超过 50 毫秒才能完成的操作，请始终提供反馈。



二、RAIL 模型

2、动画（Animation）—— 在 10 毫秒内生成一帧

- 我们的目标是每秒生成 60 帧，每一帧必须完成以下所有步骤：



- 从数学角度而言，每帧的预算约为 16 毫秒（1000 毫秒 / 60 帧 = 16.66 毫秒/帧）。 但因为浏览器需要花费时间将新帧绘制到屏幕上，因此只有 10 毫秒来执行代码。
- 如果可能，请利用 100 毫秒的响应时间预先计算开销大的工作，这样就可以尽可能增加实现 60fps 的可能性。

二、RAIL 模型

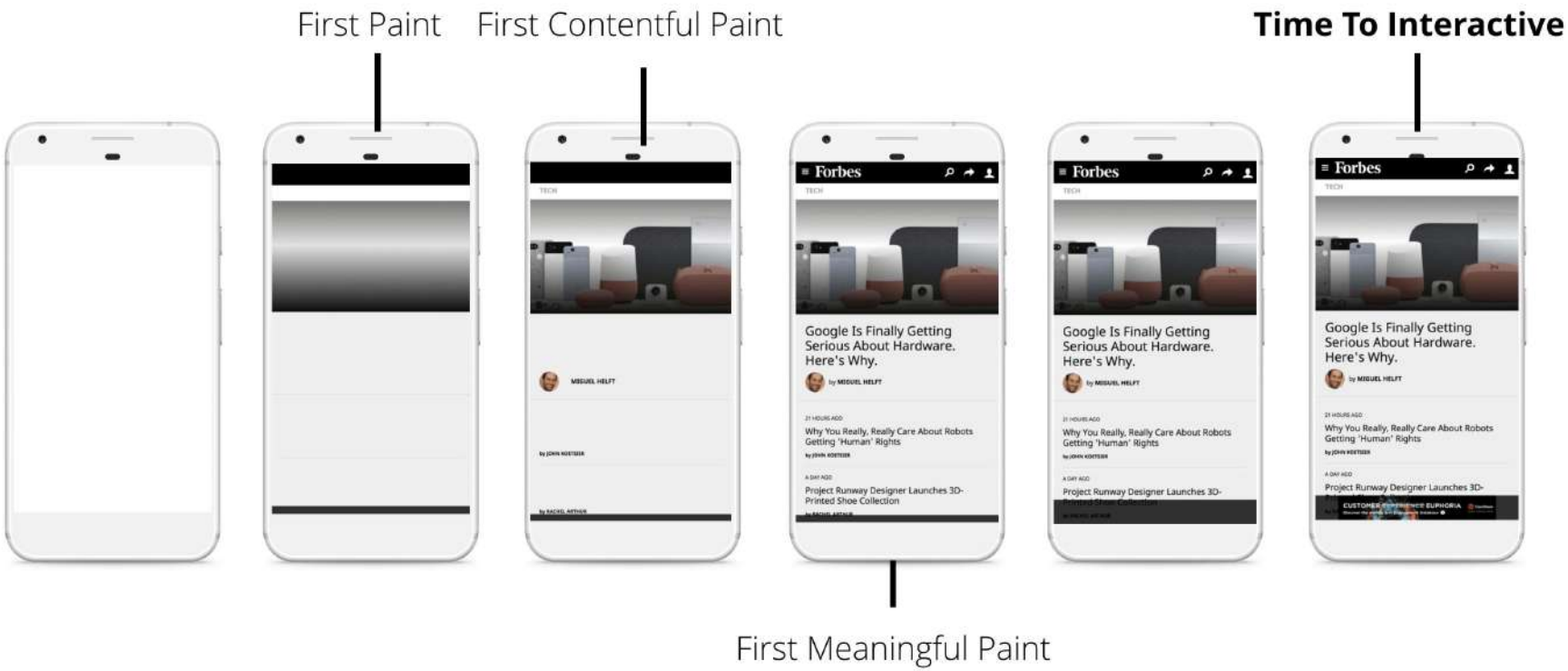
3、空闲（Idle）—— 最大程度增加空闲时间

- 利用空闲时间完成推迟的工作。例如，尽可能减少预加载数据，以便主要数据快速加载，并利用空闲时间加载剩余数据。
- 推迟的工作应分成每个耗时约 50 毫秒的多个块。这样既可以完成任务，又能确保即时的响应。
- 如果一个用户在空闲工作时间与页面有交互，那么请中断空闲工作，并确保用户交互享有最高优先级。

二、RAIL 模型

4、加载（Load）—— 在 5 秒内呈现内容，且使页面「可交互」

- 对于移动设备，在慢速 3G 网络下，须保证页面在 5 秒内加载。
- 对于需要频繁加载的场景，须将时间缩短为 2 秒。



二、RAIL 模型

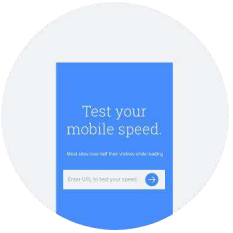
4、衡量 RAIL 的工具



Lighthouse



WebPageTest



TestMySite



PageSpeed Insights



Speed Scorecard



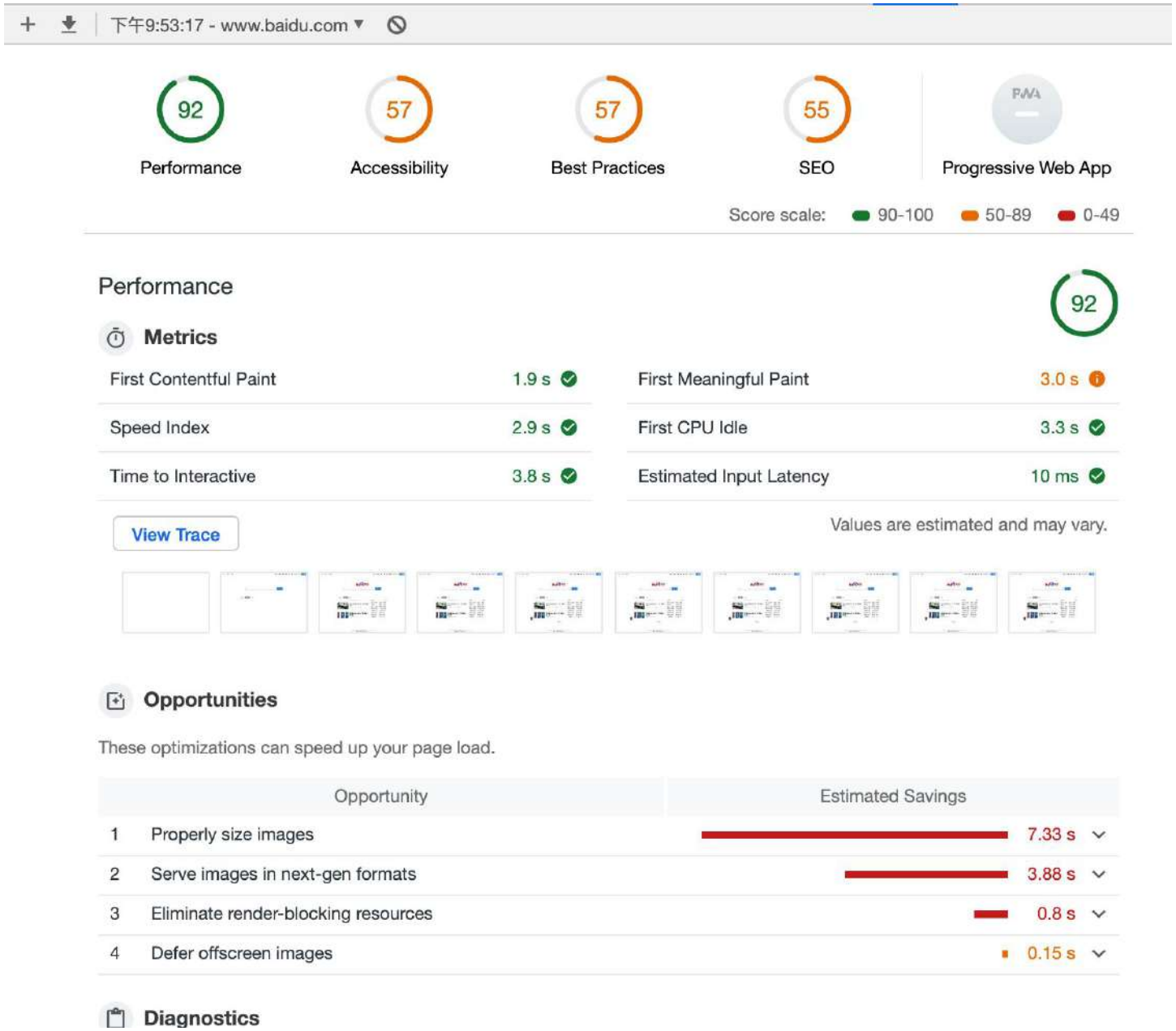
Chrome Developer Tools

3

页面加载

三、页面加载

提高页面加载速度的第一步，是测量当前页面存在哪些问题。



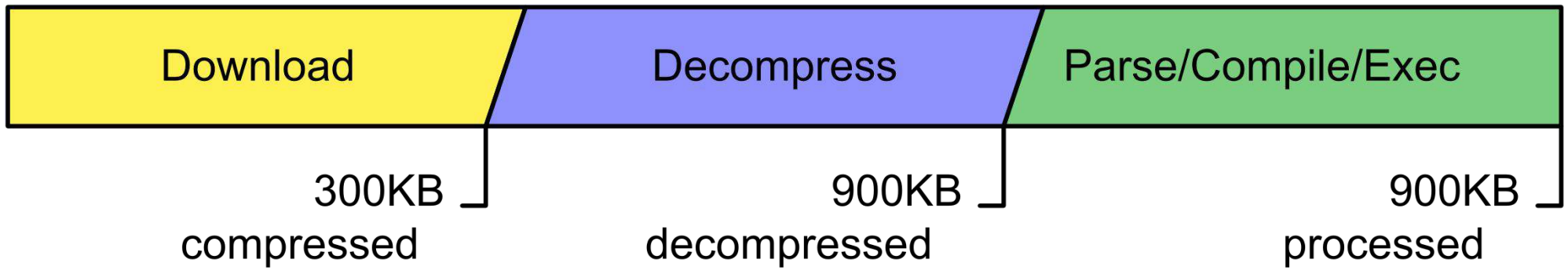
三、页面加载

1、文本优化

- Dev 与 Prod 分开。
- 代码分割 —— Webpack Spliting 懒加载。
- 缩小代码字符（Minifier）。
- 压缩文本资源 —— Gzip（需要对服务器进行配置）。
- 谨慎使用第三方脚本（性能、隐私、安全）：
  - 社交分享按钮；
  - 视频播放器嵌入；
  - iframe
  - 埋点分析和指标脚本；
  - 辅助库（如：日期格式、动画、功能库等）。

三、页面加载

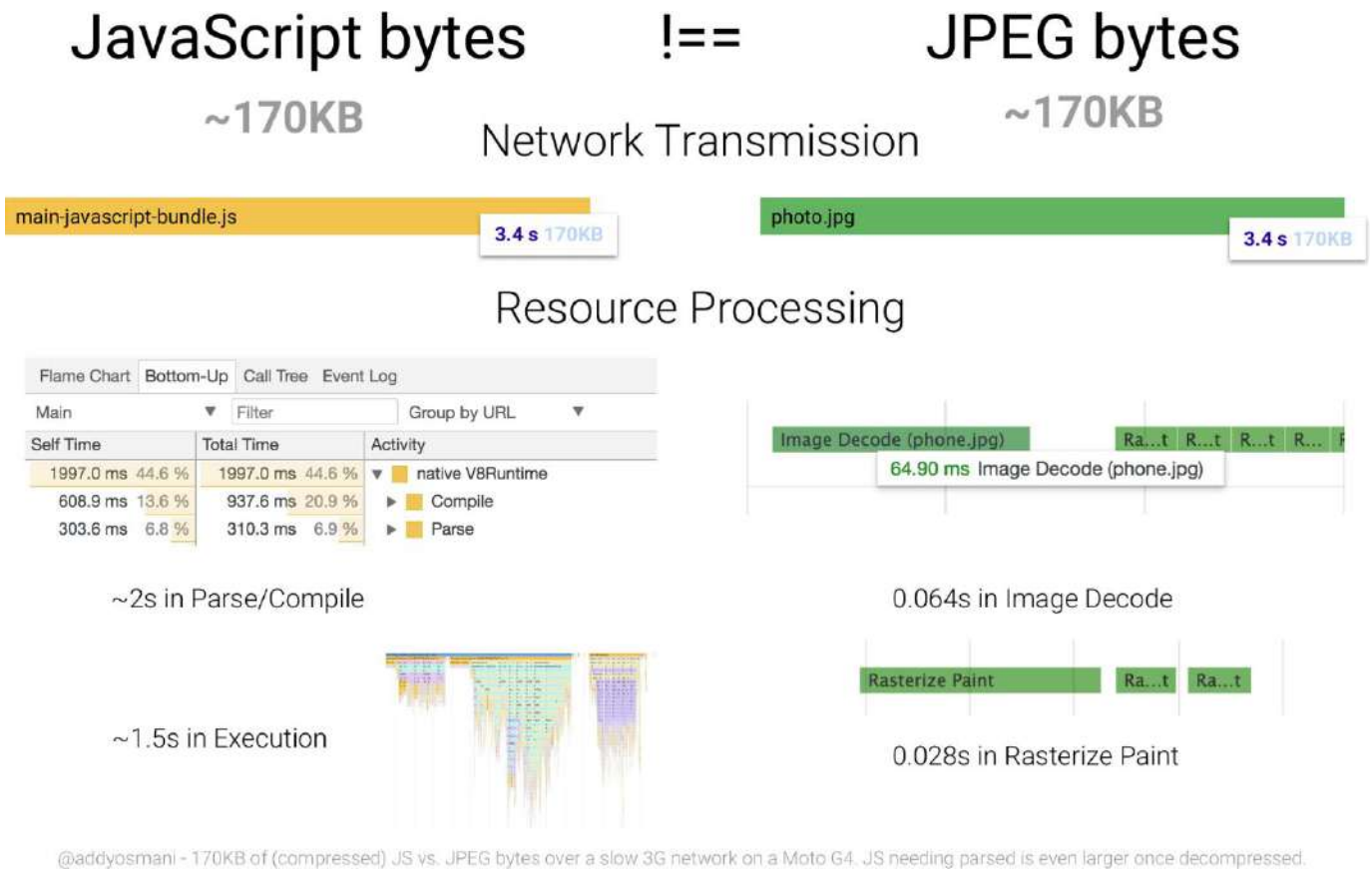
1、文本优化 —— JavaScript



即使脚本的传输大小为 300KB，仍然需要解析、编译和执行 900KB 的 JavaScript。

三、页面加载

1、文本优化 —— JavaScript



JavaScript 是一种昂贵的资源，比图像的处理成本要高很多。

三、页面加载

1、文本优化 —— JavaScript 的 Tree-Shaking

- 含义：把代码中没用到的模块「摇」掉，达到删除无用代码的目的。
- 意义：文件体积越小，网络传输成本越低，代码执行越快。
- 原理：基于 ES6 的模块特性（ES6 模块依赖关系是确定的，和运行时的状态无关，可以进行可靠的静态分析）。

```
import arrayUtils from "array-utils”

import { unique, implode, explode } from "array-utils"
```

建议：在 import 第三方库时，尽量「精准」引入。

三、页面加载

1、文本优化 —— JavaScript 的 和

- ：对以后要使用的非关键资源的低优先级获取。

```
<link rel="prefetch" href="/main.js">
```

  - 浏览器会在空闲时，下载 main.js，并缓存到 disk。当有页面使用时，直接从 disk 缓存中读取。
  - 如果 prefetch 还没下载完之前，浏览器发现 script 标签也引用了同样的资源，浏览器会再次发起请求，这样会加载两次，严重影响性能，所以不要对当前页面马上就要用的资源使用 prefetch，要用 preload。

- ：对当前路由使用的关键资源的高优先级获取。

```
<link rel="preload" href="/main.js" as="script">
```

  - 浏览器会在遇到如上 link 标签时，立刻开始下载 main.js (不阻塞 parser)，并放在内存中，但不会执行其中的 JS 语句。只有当遇到 script 标签加载的也是 main.js 的时候，浏览器才会直接将预先加载的 JS 执行。

三、页面加载

2、图像优化

- 删除不必要的图像（如果图像没有为用户提供信息价值，请务必删除）。
- 选择适当的图像类型：
  - 首选矢量格式；
  - WebP 格式在压缩和质量上表现优异；
  - 用视频替换较大的 GIF。
- 删除图像元数据（通过图像编辑器或者在线工具）。
- 调整图像大小：
  - 根据用途调整大小
  - 剪裁图像，仅显示重要内容
- 压缩图像。比如 svgo。
- CSS Sprites（适用于 HTTP/1.x）。
- 懒加载。

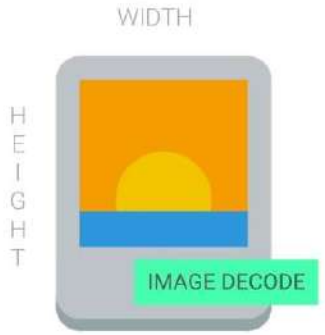
三、页面加载

2、图像优化

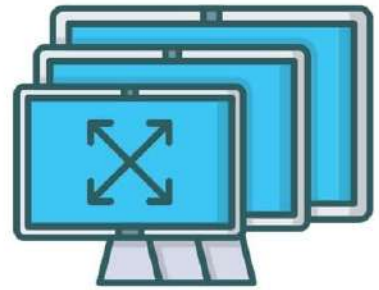
Image Optimisation



Choose the right format



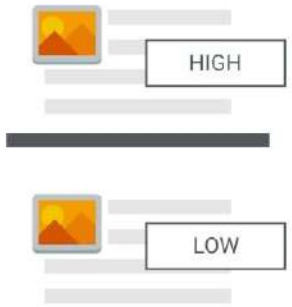
Size appropriately



Adapt intelligently



Compress carefully



Prioritize critical images



Lazy-load the rest



Take care with tools

三、页面加载

3、HTTP 请求

- 对于大资源：
  - 是否合并对于加载时间没有明显影响，但拆分资源可以更好的利用浏览器缓存，不会因为某个资源的更新导致所有资源缓存失效。
  - 还可以利用域名分片技术，将资源拆分部署到不同域名，既可以分散服务器的压力，又可以降低网络抖动带来的影响。
- 对于小资源：
  - 合并资源往往具有更快的加载速度，但在网络带宽状况良好的情况下，因为提升的时间单位以 ms 计量，收益可以忽略。
  - 如果网络延迟很大，服务器响应速度又慢，则可以带来一定收益，在高延迟的网络场景下，又要注意合并资源后可能带来网络往返次数的增加，进而影响到加载时间。
- 注意：对于 HTTP/2，对资源进行合并的方式所带来的的价值并不大。

三、页面加载

4、HTTP 缓存

- 启用缓存
  - 哪些资源需要缓存？
  - 缓存实际上可以存在于 BS 间的任何中间站点。
- Cache Headers（需服务端配置）
  - Cache-Control
  - Etag
  - Last-Modified
  - Expires（过时）

三、页面加载

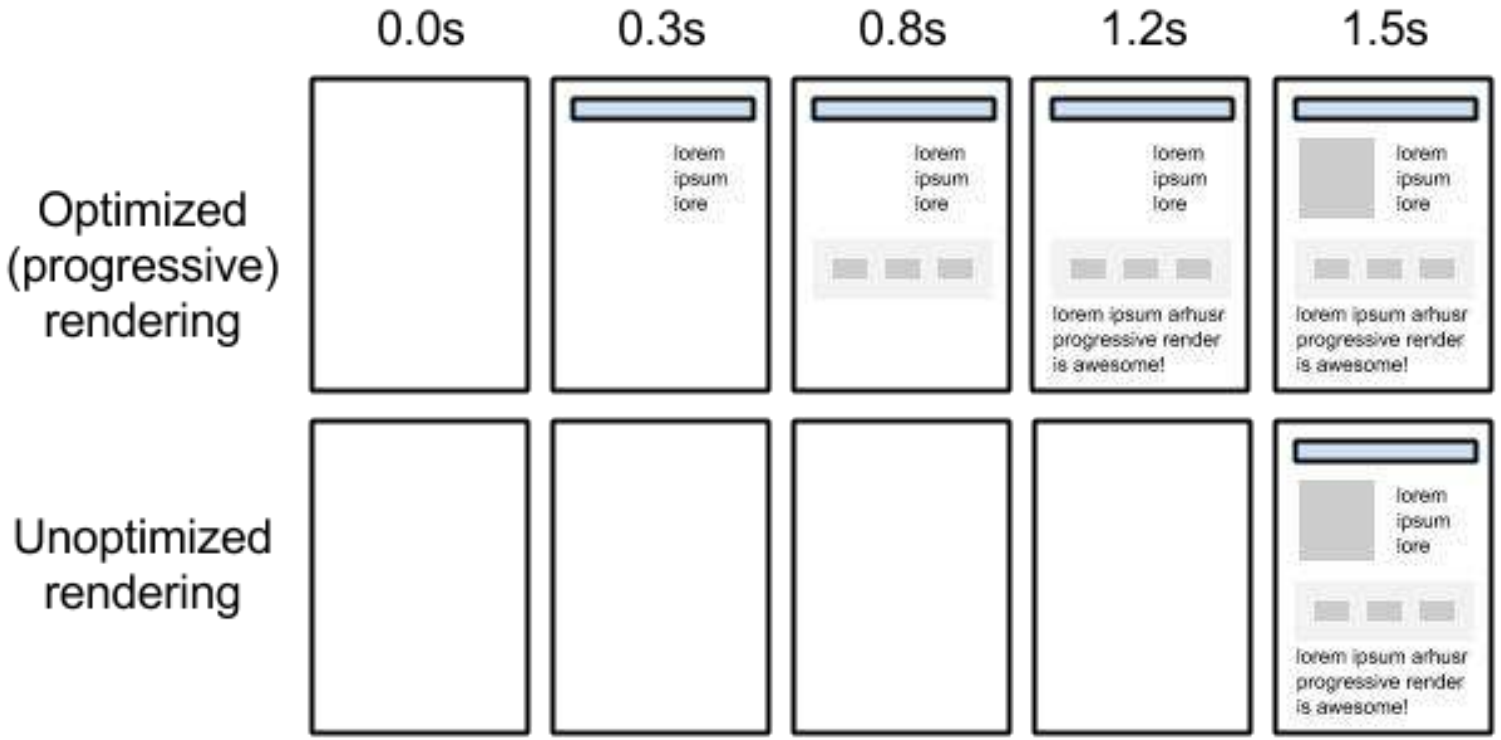
6、一个 API

```
window.performance
```

三、页面加载

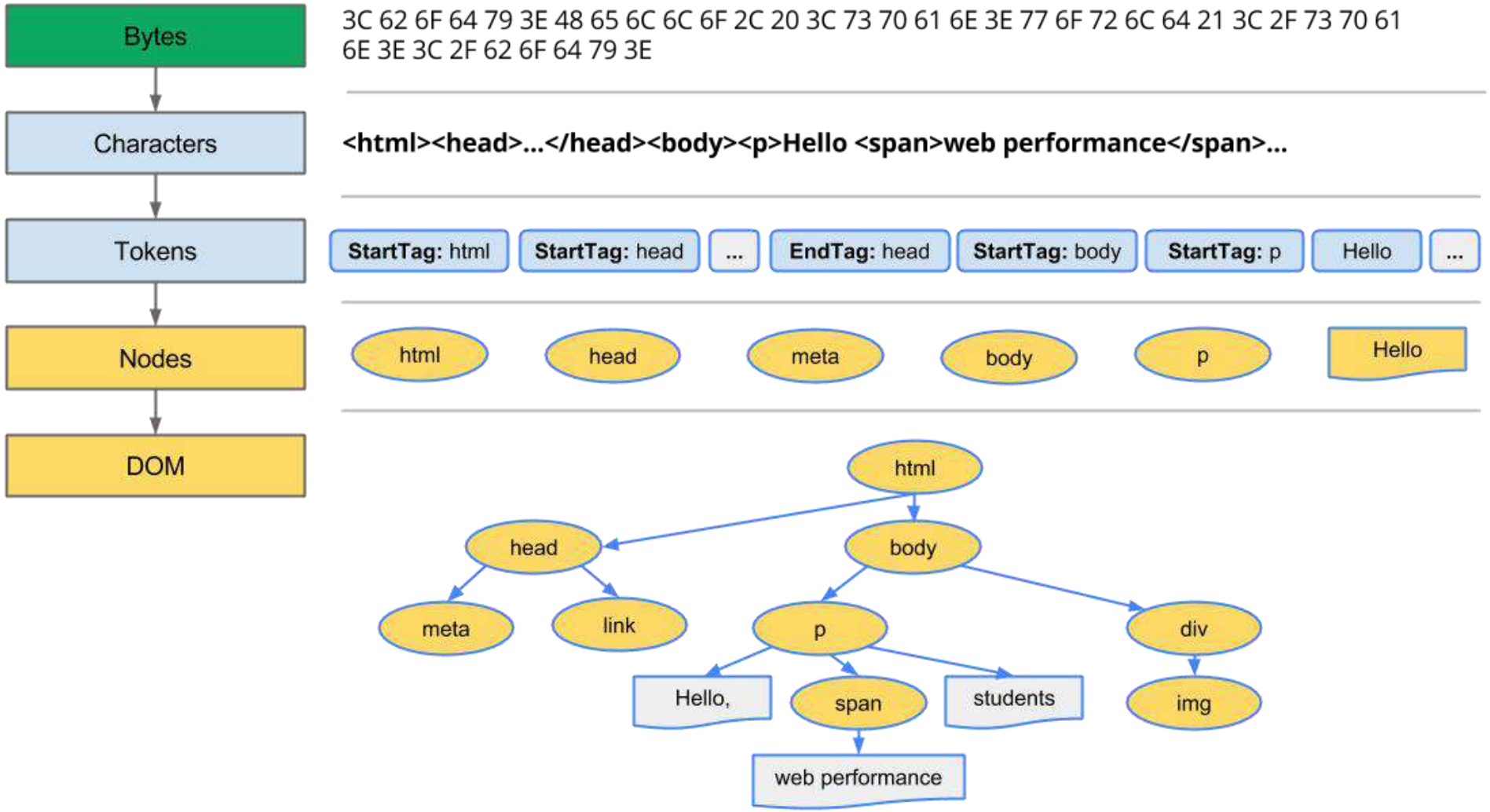
7、优化**关键渲染路径**（Critical Rendering Path）

- 定义：对与当前用户动作相关的内容的显示进行优先级排序。
- 实质：优化浏览器接收 HTML，CSS 和 JavaScript 的过程。
- 目的：缩短页面首次渲染的时间。



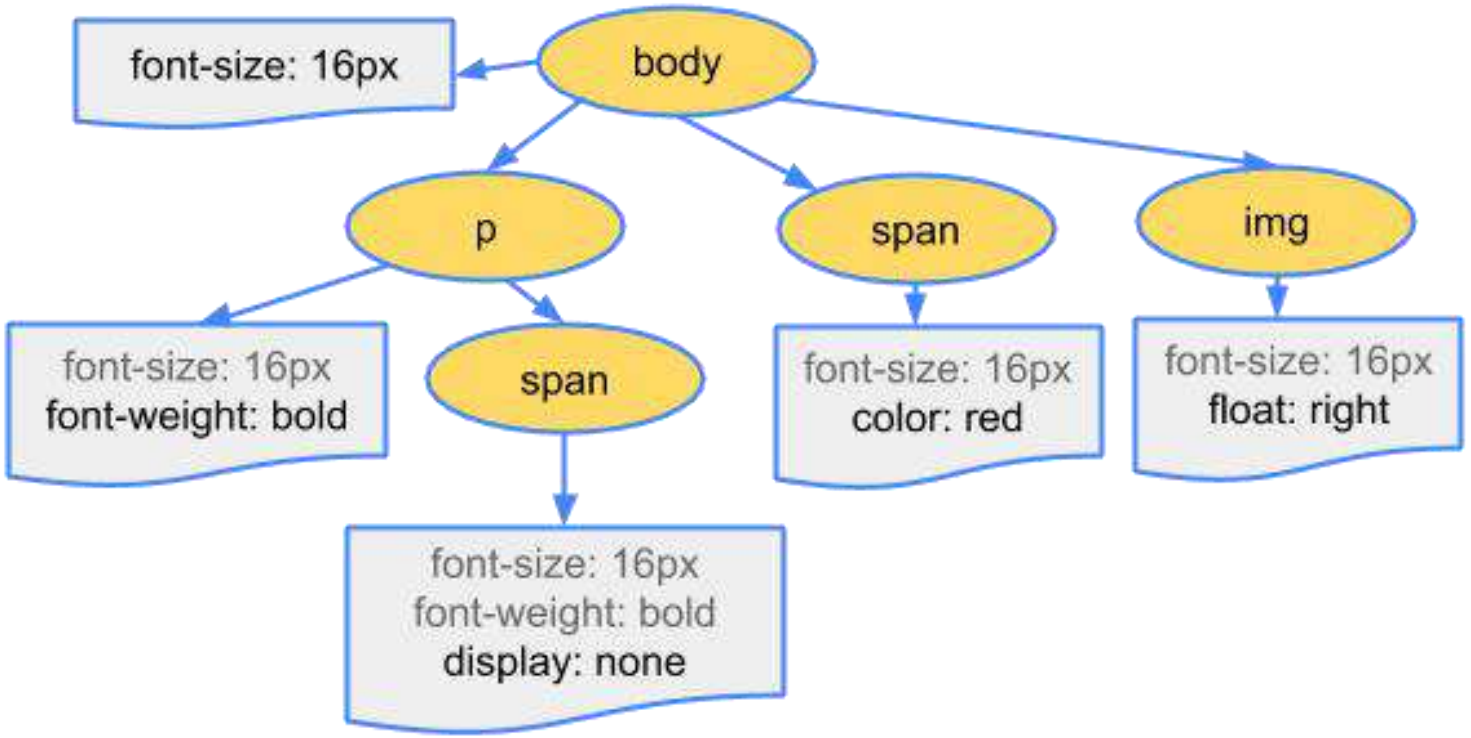
三、页面加载

7、优化关键渲染路径 —— DOM

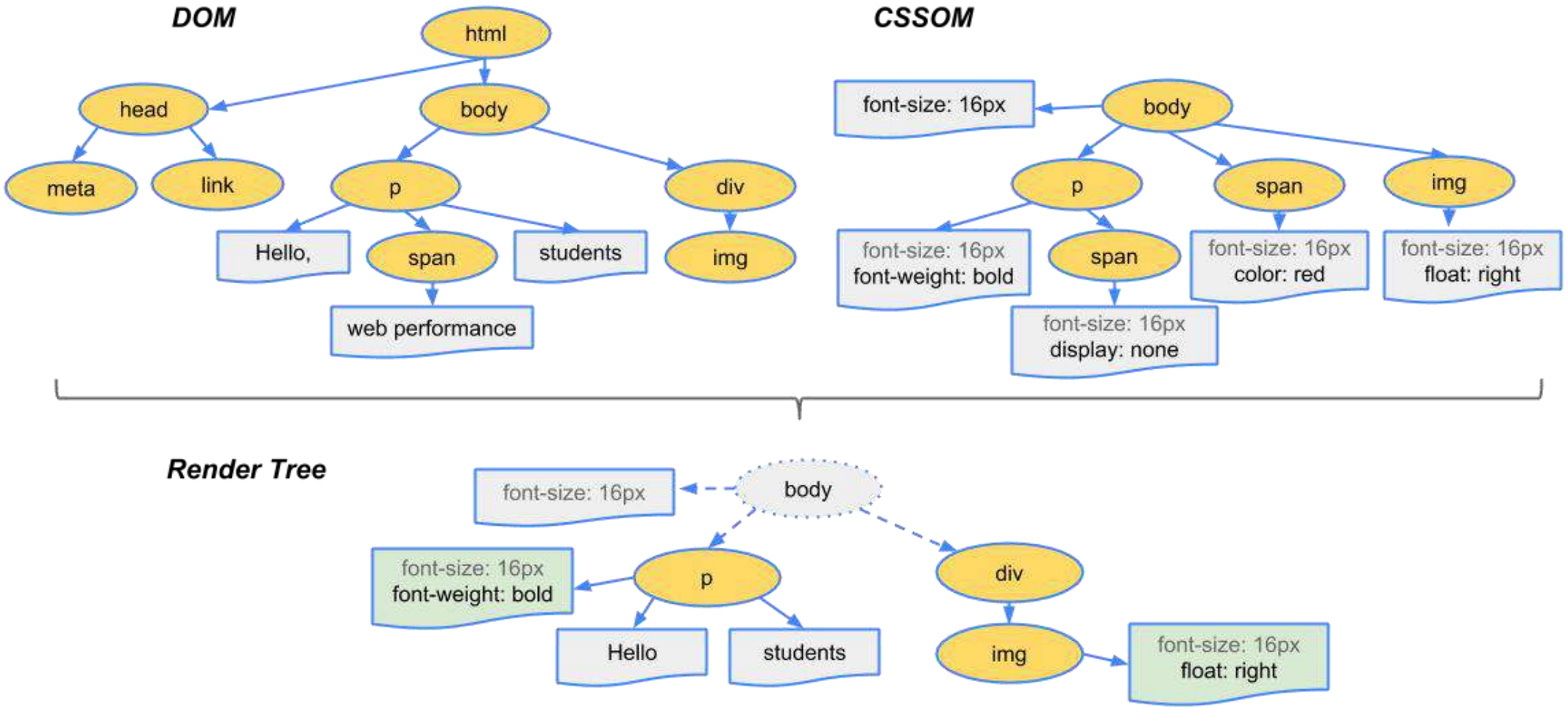


7、优化关键渲染路径 —— CSSOM





7、优化**关键渲染路径** —— 渲染树



7、优化**关键渲染路径** —— 步骤总结

- （1）处理 HTML 标记并构建 DOM 树。
- （2）处理 CSS 标记并构建 CSSOM 树。
- （3）将 DOM 和 CSSOM 组合到渲染树中。
- （4）在渲染树上计算布局。
- （5）将各个节点绘制到屏幕上。

优化关键渲染路径：如何将步骤（1）到（5）所花费的总时间降到最少。

① 表征关键渲染路径的三个变量：关键资源数量，关键路径长度，关键字节数。

② 最大限度地减少关键资源数量：消除它们，推迟下载，将它们标记为异步，等等。

③ 优化关键字节数以减少下载时间（往返次数）。

④ 优化加载剩余关键资源的顺序：尽可能早地下载所有关键资源，以缩短关键路径长度。

4

页面渲染

四、页面渲染

渲染性能：

页面不仅要快速加载，还要运行良好：滚动、动画、交互应该丝般顺滑。

四、页面渲染

关于渲染的五个方面：

- **JavaScript**：动画处理、数据排序、操作 DOM 等，目的是触发视觉变化。另外一些：CSS 动画、过渡和 Web 动画 API。
- **样式计算**：CSS 选择器，n　　≻或≻n　　> ≻n　　。一旦知道规则，就会计算每个元素的最终样式。
- **布局**：一旦浏览器知道哪些规则适用于元素，它就可以开始计算它占用的空间大小以及它在屏幕上的位置。
- **绘制**：填充像素的过程。它涉及绘制文本、颜色、图像、边框和阴影，包含了元素的每个视觉部分。基于「层」。
- **合成**：由于页面的各部分可能被绘制成多个图层，因此需要以正确的顺序将它们绘制到屏幕上。



四、页面渲染

1、JavaScript

- 避免使用>　　>或>　　>进行可视化更新；推荐总是使用>　　。
- 将长时间运行的 JavaScript 从主线程移到 Web Workers。
- 使用微任务在多个帧上进行 DOM 操作。
- Chrome DevTools 的 Performance 面板是衡量 JavaScript 成本的最佳方式。

四、页面渲染

2、样式计算

- 降低选择器的复杂性，比如使用像 BEM 这样的 class 命名方案。
- 减少样式化元素的数量 —— 精简 CSS 结构。

四、页面渲染

3、布局

- 尽可能避免页面重排（reflow）。
- 优先使用 Flexbox 布局模型（在考虑兼容性的前提下），因为更快。
- 谨慎使用「快速、连续、大量的布局操作」。

```
for (var i = 0; i < paragraphs.length; i++) {
    paragraphs[i].style.width = box.offsetWidth + 'px';
}
```

```
var width = box.offsetWidth;

for (var i = 0; i < paragraphs.length; i++) {
    paragraphs[i].style.width = width + 'px';
}
```

四、页面渲染

4、绘制

- 绘制是所有任务中耗时最长的，应尽可能避免。
- 更改非几何属性（如背景，文本颜色或阴影），可以绕过布局而只触发绘制。
- 使用 Chrome DevTools 快速识别绘制瓶颈。



四、页面渲染

5、合成

- 为了避免「布局」和「绘制」造成的性能损耗，仅对「合成」进行更改：
  -
  -
- 将动画元素置于新的图层上（每个图层都会进行内存管理，所以也应谨慎使用）

```
.moving-element {
    will-change: transform;      /* 新浏览器 */
    transform: translateZ(0);    /* 旧浏览器 */
}
```

- 防抖（debounce）和节流（throttle）。

总结

