浏览器的渲染：过程与原理

内容说明

本文不是关于浏览器渲染的底层原理或前端优化具体细节的讲解，而是关于浏览器对页面的渲染——这一过程的描述及其背后原理的解释。这是因为前端优化是一个非常庞大且零散的知识集合，一篇文章如果要写优化的具体方法恐怕只能做一些有限的列举。

然而，如果了解清楚浏览器的渲染过程、渲染原理，其实就掌握了指导原则。根据优化原则，可以实现出无数种具体的优化方案，各种预编译、预加载、资源合并、按需加载方案都是针对浏览器渲染习惯的优化。

关键渲染路径

提到页面渲染，有几个相关度非常高的概念，最重要的是关键渲染路径，其他几个概念都可以从它展开，下面稍作说明。

关键渲染路径（Critical Rendering Path）是指与当前用户操作有关的内容。例如用户刚刚打开一个页面，首屏的显示就是当前用户操作相关的内容，具体就是浏览器收到 HTML、CSS 和 JavaScript 等资源并对其进行处理从而渲染出 Web 页面。

了解浏览器渲染的过程与原理，很大程度上是为了优化关键渲染路径，但优化应该是针对具体问题的解决方案，所以优化没有一定之规。例如为了保障首屏内容的最快速显示，通常会提到渐进式页面渲染，但是为了渐进式页面渲染，就需要做资源的拆分，那么以什么粒度拆分、要不要拆分，不同页面、不同场景策略不同。具体方案的确定既要考虑体验问题，也要考虑工程问题。

浏览器渲染页面的过程

从耗时的角度，浏览器请求、加载、渲染一个页面，时间花在下面五件事情上：

1. DNS 查询
2. TCP 连接

3. HTTP 请求即响应
4. 服务器响应
5. 客户端渲染

本文讨论第五个部分，即浏览器对内容的渲染，这一部分（渲染树构建、布局及绘制），又可以分为下面五个步骤：

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 与 CSSOM 合并成一个渲染树。
4. 根据渲染树来布局，以计算每个节点的几何信息。
5. 将各个节点绘制到屏幕上。

需要明白，这五个步骤并不一定一次性顺序完成。如果 DOM 或 CSSOM 被修改，以上过程需要重复执行，这样才能计算出哪些像素需要在屏幕上进行重新渲染。实际页面中，CSS 与 JavaScript 往往会多次修改 DOM 和 CSSOM，下面就来看看它们的影响方式。

阻塞渲染：CSS 与 JavaScript

谈论资源的阻塞时，我们要清楚，现代浏览器总是并行加载资源。例如，当 HTML 解析器（HTML Parser）被脚本阻塞时，解析器虽然会停止构建 DOM，但仍会识别该脚本后面的资源，并进行预加载。

同时，由于下面两点：

1. 默认情况下，CSS 被视为阻塞渲染的资源，这意味着浏览器将不会渲染任何已处理的内容，直至 CSSOM 构建完毕。
2. JavaScript 不仅可以读取和修改 DOM 属性，还可以读取和修改 CSSOM 属性。

存在阻塞的 CSS 资源时，浏览器会延迟 JavaScript 的执行和 DOM 构建。另外：

1. 当浏览器遇到一个 script 标记时，DOM 构建将暂停，直至脚本完成执行。
2. JavaScript 可以查询和修改 DOM 与 CSSOM。

3. CSSOM 构建时，JavaScript 执行将暂停，直至 CSSOM 就绪。

所以，script 标签的位置很重要。实际使用时，可以遵循下面两个原则：

1. CSS 优先：引入顺序上，CSS 资源先于 JavaScript 资源。
2. JavaScript 应尽量少影响 DOM 的构建。

浏览器的发展日益加快（目前的 Chrome 官方稳定版是 61），具体的渲染策略会不断进化，但了解这些原理后，就能想通它进化的逻辑。下面来看看 CSS 与 JavaScript 具体会怎样阻塞资源。

CSS

```
<style> p { color: red; }</style>
<link rel="stylesheet" href="index.css">
```

这样的 link 标签（无论是否 inline）会被视为阻塞渲染的资源，浏览器会优先处理这些 CSS 资源，直至 CSSOM 构建完毕。

渲染树（Render-Tree）的关键渲染路径中，要求同时具有 DOM 和 CSSOM，之后才会构建渲染树。即，HTML 和 CSS 都是阻塞渲染的资源。HTML 显然是必需的，因为包括我们希望显示的文本在内的内容，都在 DOM 中存放，那么可以从 CSS 上想办法。

最容易想到的当然是精简 CSS 并尽快提供它。除此之外，还可以用媒体类型（media type）和媒体查询（media query）来解除对渲染的阻塞。

```
<link href="index.css" rel="stylesheet">
<link href="print.css" rel="stylesheet" media="print">
<link href="other.css" rel="stylesheet" media="(min-width: 30em) and (orien
```

第一个资源会加载并阻塞。
第二个资源设置了媒体类型，会加载但不会阻塞，print 声明只在打印网页时使用。
第三个资源提供了媒体查询，会在符合条件时阻塞渲染。

JavaScript

JavaScript 的情况比 CSS 要更复杂一些。观察下面的代码：

```
<p>Do not go gentle into that good night,</p>
<script>console.log("inline")</script>
<p>Old age should burn and rave at close of day;</p>
<script src="app.js"></script>
<p>Rage, rage against the dying of the light.</p>

<p>Do not go gentle into that good night,</p>
<script src="app.js"></script>
<p>Old age should burn and rave at close of day;</p>
<script>console.log("inline")</script>
<p>Rage, rage against the dying of the light.</p>
```

这样的 script 标签会阻塞 HTML 解析，无论是不是 inline-script。上面的 P 标签会从上到下解析，这个过程会被两段 JavaScript 分别打算一次（加载、执行）。

所以实际工程中，我们常常将资源放到文档底部。

改变阻塞模式：defer 与 async

为什么要将 script 加载的 defer 与 async 方式放到后面呢？因为这两种方式是的出现，全是由于前面讲的那些阻塞条件的存在。换句话说，defer 与 async 方式可以改变之前的那些阻塞情形。

首先，注意 async 与 defer 属性对于 inline-script 都是无效的，所以下面这个示例中三个 script 标签的代码会从上到下依次执行。

```
<!-- 按照从上到下的顺序输出 1 2 3 -->
<script async>
  console.log("1");
</script>
<script defer>
  console.log("2");
</script>
<script>
  console.log("3");
</script>
```

故，下面两节讨论的内容都是针对设置了 src 属性的 script 标签。

defer

```
<script src="app1.js" defer></script>
<script src="app2.js" defer></script>
<script src="app3.js" defer></script>
```

defer 属性表示延迟执行引入的 JavaScript，即这段 JavaScript 加载时 HTML 并未停止解析，这两个过程是并行的。整个 document 解析完毕且 defer-script 也加载完成之后（这两件事情的顺序无关），会执行所有由 defer-script 加载的 JavaScript 代码，然后触发 DOMContentLoaded 事件。

defer 不会改变 script 中代码的执行顺序，示例代码会按照 1、2、3 的顺序执行。所以，defer 与相比普通 script，有两点区别：载入 JavaScript 文件时不阻塞 HTML 的解析，执行阶段被放到 HTML 标签解析完成之后。

async

```
<script src="app.js" async></script>
<script src="ad.js" async></script>
<script src="statistics.js" async></script>
```

async 属性表示异步执行引入的 JavaScript，与 defer 的区别在于，如果已经加载好，就会开始执行——无论此刻是 HTML 解析阶段还是 DOMContentLoaded 触发之后。需要注意的是，这种方式加载的 JavaScript 依然会阻塞 load 事件。换句话说，async-script 可能在 DOMContentLoaded 触发之前或之后执行，但一定在 load 触发之前执行。

从上一段也能推出，多个 async-script 的执行顺序是不确定的。值得注意的是，向 document 动态添加 script 标签时，async 属性默认是 true，下一节会继续这个话题。

document.createElement

使用 document.createElement 创建的 script 默认是异步的，示例如下。

```
console.log(document.createElement("script").async); // true
```

所以，通过动态添加 script 标签引入 JavaScript 文件默认是不会阻塞页面的。如果想同步执行，需要将 async 属性人为设置为 false。

如果使用 document.createElement 创建 link 标签会怎样呢?

```
const style = document.createElement("link");
style.rel = "stylesheet";
style.href = "index.css";
document.head.appendChild(style); // 阻塞?
```

其实这只能通过试验确定，已知的是，Chrome 中已经不会阻塞渲染，Firefox、IE 在以前是阻塞的，现在会怎样我没有试验。

document.write 与 innerHTML

通过 document.write 添加的 link 或 script 标签都相当于添加在 document 中的标签，因为它操作的是 document stream（所以对于 loaded 状态的页面使用 document.write 会自动调用 document.open，这会覆盖原有文档内容）。即正常情况下， link 会阻塞渲染，script 会同步执行。不过这是不推荐的方式，Chrome 已经会显示警告，提示未来有可能禁止这样引入。如果给这种方式引入的 script 添加 async 属性，Chrome 会检查是否同源，对于非同源的 async-script 是不允许这么引入的。

如果使用 innerHTML 引入 script 标签，其中的 JavaScript 不会执行。当然，可以通过 eval() 来手工处理，不过不推荐。如果引入 link 标签，我试验过在 Chrome 中是可以起作用的。另外，outerHTML、insertAdjacentHTML() 应该也是相同的行为，我并没有试验。这三者应该用于文本的操作，即只使用它们添加 text 或普通 HTML Element。

参考资料

Mobile Analysis in PageSpeed Insights

Web Fundamentals

MDN - HTML element reference