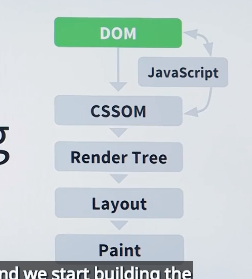
关键呈现路径（The Critical Rendering Path）是指浏览器所经历的一系列步骤

将Html css js 转换为在屏幕上呈现的像素内容

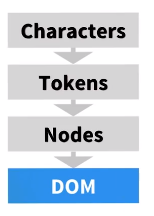


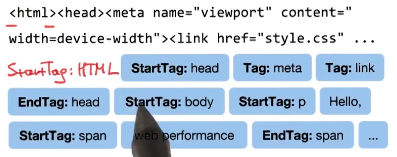
首先浏览器获取HTMl并开始构建文档对象模型（DOM）

然后获取CSS并构建CSS对象模型

然后将两者结合，创建渲染树，布局

HTML如何转换成DOM：



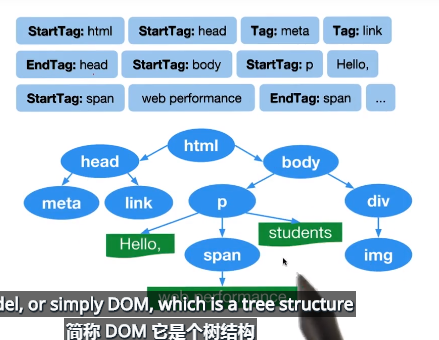
每当我们遇到一个标记，浏览器都发出一个令牌

整个流程都由令牌生成器完成（Chrome浏览器下是这样实现的，我猜的）

例如我们遇到第一个HTML令牌，创建一个html节点，并把

StartTag：Html这个令牌消耗掉，startTag：head在EngTag;html

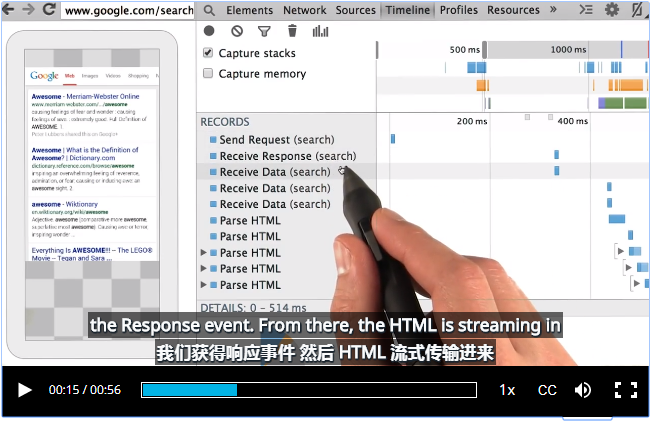
之前，所以head是html的子节点



根据这个浏览器逐步构建DOM树可以有一个返回部分HTML

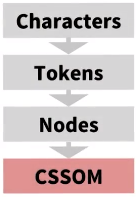
的性能优化策略

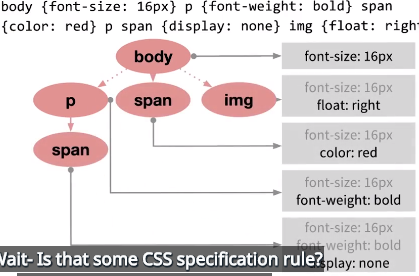
Google Chrome的timeline工具：



如果您打开 Chrome DevTools 并在页面加载时记录时间线，就可以看到执行该步骤实际花费的时间。在上例中，将一堆 HTML 字节转换成 DOM 树大约需要 5 毫秒。对于较大的页面，这一过程需要的时间可能会显著增加。创建流畅动画时，如果浏览器需要处理大量 HTML，这很容易成为瓶颈。

构建cssom



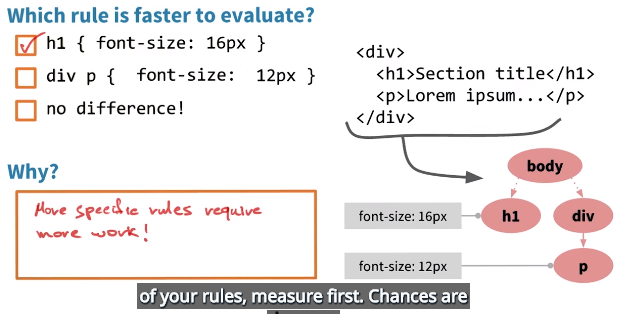


注意，css不能边加载边创建树，因为后来的样式会重写css树，导致耗费的时间

更多。

Css中，选择器越复杂，涉及到的dom节点越多，速度越慢。也就是说我给每个

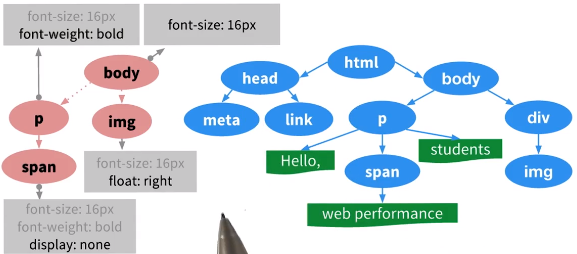
节点加id，用#id这样来选取的话，速度会很快咯？

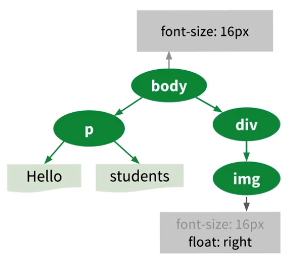


不过这不是影响性能的主要因素

我们的小样式表需要大约 0.6 毫秒的处理时间，影响页面上的 8 个元素 — 虽然不多，但同样会产生开销。不过，这 8 个元素从何而来呢？CSSOM 和 DOM 是独立的数据结构！结果证明，浏览器隐藏了一个重要步骤。接下来，让我们谈一谈将 DOM 与 CSSOM 关联在一起的[渲染树](https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction)。

渲染树：





* DOM 树与 CSSOM 树合并后形成渲染树。
* 渲染树只包含渲染网页所需的节点。
* 布局计算每个对象的精确位置和大小。
* 最后一步是绘制，使用最终渲染树将像素渲染到屏幕上。

为构建渲染树，浏览器大体上完成了下列工作：

1. 从 DOM 树的根节点开始遍历每个可见节点。
   * 某些节点不可见（例如脚本标记、元标记等），因为它们不会体现在渲染输出中，所以会被忽略。
   * 某些节点通过 CSS 隐藏，因此在渲染树中也会被忽略，例如，上例中的 span 节点---不会出现在渲染树中，---因为有一个显式规则在该节点上设置了“display: none”属性。
2. 对于每个可见节点，为其找到适配的 CSSOM 规则并应用它们。
3. 发射可见节点，连同其内容和计算的样式。

注：简单提一句，请注意 visibility: hidden 与 display: none 是不一样的。前者隐藏元素，但元素仍占据着布局空间（即将其渲染成一个空框），而后者 (display: none) 将元素从渲染树中完全移除，元素既不可见，也不是布局的组成部分。

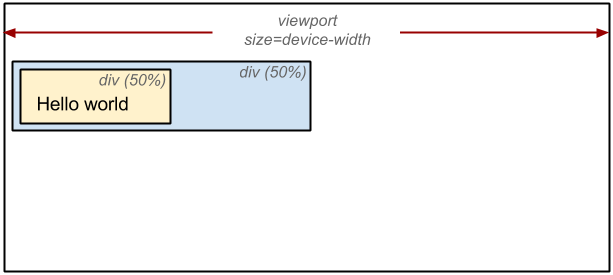
最终输出的渲染同时包含了屏幕上的所有可见内容及其样式信息。**有了渲染树，我们就可以进入“布局”阶段。**

到目前为止，我们计算了哪些节点应该是可见的以及它们的计算样式，但我们尚未计算它们在设备[视口](https://developers.google.com/web/fundamentals/design-and-ui/responsive/fundamentals/set-the-viewport)内的确切位置和大小---这就是“布局”阶段，也称为“自动重排”。

为弄清每个对象在网页上的确切大小和位置，浏览器从渲染树的根节点开始进行遍历。让我们考虑下面这样一个简单的实例：

<html>  
  <head>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <title>Critial Path: Hello world!</title>  
  </head>  
  <body>  
    <div style="width: 50%">  
      <div style="width: 50%">Hello world!</div>  
    </div>  
  </body>  
</html>

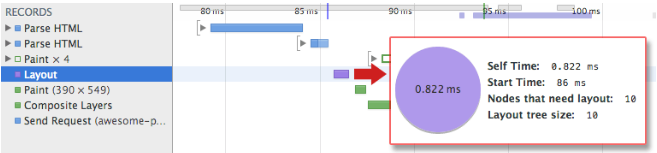
以上网页的正文包含两个嵌套 div：第一个（父）div 将节点的显示尺寸设置为视口宽度的 50%，---父 div 包含的第二个 div---将其宽度设置为其父项的 50%；即视口宽度的 25%。



布局流程的输出是一个“盒模型”，它会精确地捕获每个元素在视口内的确切位置和尺寸：所有相对测量值都转换为屏幕上的绝对像素。

最后，既然我们知道了哪些节点可见、它们的计算样式以及几何信息，我们终于可以将这些信息传递给最后一个阶段：将渲染树中的每个节点转换成屏幕上的实际像素。这一步通常称为“绘制”或“栅格化”。

上述步骤都需要浏览器完成大量工作，所以相当耗时。不过，Chrome DevTools 可以帮助我们对上述所有三个阶段进行深入的了解。让我们看一下最初“hello world”示例的布局阶段：



* “Layout”事件在时间线中捕获渲染树构建以及位置和尺寸计算。
* 布局完成后，浏览器会立即发出“Paint Setup”和“Paint”事件，将渲染树转换成屏幕上的像素。

执行渲染树构建、布局和绘制所需的时间将取决于文档大小、应用的样式，以及运行文档的设备：文档越大，浏览器需要完成的工作就越多；样式越复杂，绘制需要的时间就越长（例如，单色的绘制开销“较小”，而阴影的计算和渲染开销则要“大得多”）。

下面简要概述了浏览器完成的步骤：

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。

我们的演示网页看起来可能很简单，实际上却需要完成相当多的工作。如果 DOM 或 CSSOM 被修改，您只能再执行一遍以上所有步骤，以确定哪些像素需要在屏幕上进行重新渲染。

**\_优化关键渲染路径\_就是指最大限度缩短执行上述第 1 步至第 5 步耗费的总时间。** 这样一来，就能尽快将内容渲染到屏幕上，此外还能缩短首次渲染后屏幕刷新的时间，即为交互式内容实现更高的刷新率。

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。请务必精简您的 CSS，尽快提供它，并利用媒体类型和查询来解除对渲染的阻塞。

在[渲染树构建](https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction)中，我们看到关键渲染路径要求我们同时具有 DOM 和 CSSOM 才能构建渲染树。这会给性能造成严重影响：**HTML 和 CSS 都是阻塞渲染的资源。** HTML 显然是必需的，因为如果没有 DOM，我们就没有可渲染的内容，但 CSS 的必要性可能就不太明显。如果我们在 CSS 不阻塞渲染的情况下尝试渲染一个普通网页会怎样？

* 默认情况下，CSS 被视为阻塞渲染的资源。
* 我们可以通过媒体类型和媒体查询将一些 CSS 资源标记为不阻塞渲染。
* 浏览器会下载所有 CSS 资源，无论阻塞还是不阻塞。

**CSS 是阻塞渲染的资源。需要将它尽早、尽快地下载到客户端，以便缩短首次渲染的时间。**

不过，如果我们有一些 CSS 样式只在特定条件下（例如显示网页或将网页投影到大型显示器上时）使用，又该如何？如果这些资源不阻塞渲染，该有多好。

我们可以通过 CSS“媒体类型”和“媒体查询”来解决这类用例：

<link href="style.css" rel="stylesheet">  
<link href="print.css" rel="stylesheet" media="print">  
<link href="other.css" rel="stylesheet" media="(min-width: 40em)"> [媒体查询](https://developers.google.com/web/fundamentals/design-and-ui/responsive/" \l "use-css-media-queries-for-responsiveness)由媒体类型以及零个或多个检查特定媒体特征状况的表达式组成。例如，上面的第一个样式表声明未提供任何媒体类型或查询，因此它适用于所有情况，也就是说，它始终会阻塞渲染。第二个样式表则不然，它只在打印内容时适用---或许您想重新安排布局、更改字体等等，因此在网页首次加载时，该样式表不需要阻塞渲染。最后，最后一个样式表声明提供由浏览器执行的“媒体查询”：符合条件时，浏览器将阻塞渲染，直至样式表下载并处理完毕。

通过使用媒体查询，我们可以根据特定用例（比如显示或打印），也可以根据动态情况（比如屏幕方向变化、尺寸调整事件等）定制外观。**声明您的样式表资产时，请密切注意媒体类型和查询，因为它们将严重影响关键渲染路径的性能。**

让我们考虑下面这些实例：

<link href="style.css"    rel="stylesheet">  
<link href="style.css"    rel="stylesheet" media="all">  
<link href="portrait.css" rel="stylesheet" media="orientation:portrait">  
<link href="print.css"    rel="stylesheet" media="print">

* 第一个声明阻塞渲染，适用于所有情况。
* 第二个声明同样阻塞渲染：“all”是默认类型，如果您不指定任何类型，则隐式设置为“all”。因此，第一个声明和第二个声明实际上是等效的。
* 第三个声明具有动态媒体查询，将在网页加载时计算。根据网页加载时设备的方向，portrait.css 可能阻塞渲染，也可能不阻塞渲染。
* 最后一个声明只在打印网页时应用，因此网页首次在浏览器中加载时，它不会阻塞渲染。

最后，请注意“阻塞渲染”仅是指浏览器是否需要暂停网页的首次渲染，直至该资源准备就绪。无论哪一种情况，浏览器仍会下载 CSS 资产，只不过不阻塞渲染的资源优先级较低罢了。

JavaScript 允许我们修改网页的方方面面：内容、样式以及它如何响应用户交互。 不过，JavaScript 也会阻止 DOM 构建和延缓网页渲染。 为了实现最佳性能，可以让您的 JavaScript 异步执行，并去除关键渲染路径中任何不必要的 JavaScript。

* JavaScript 可以查询和修改 DOM 与 CSSOM。
* JavaScript 执行会阻止 CSSOM。
* 除非将 JavaScript 显式声明为异步，否则它会阻止构建 DOM。
* JavaScript 是一种运行在浏览器中的动态语言，它允许我们对网页行为的几乎每一个方面进行修改：我们可以通过在 DOM 树中添加和移除元素来修改内容；我们可以修改每个元素的 CSSOM 属性；我们可以处理用户输入，等等。为进行说明，让我们用一个简单的内联脚本对之前的“Hello World”示例进行扩展：
* <html>  
    <head>  
      <meta name="viewport" content="width=device-width,initial-scale=1">  
      <link href="style.css" rel="stylesheet">  
      <title>Critical Path: Script</title>  
    </head>  
    <body>  
      <p>Hello <span>web performance</span> students!</p>  
      <div><img src="awesome-photo.jpg"></div>  
      <script>  
        var span = document.getElementsByTagName('span')[0];  
        span.textContent = 'interactive'; // change DOM text content  
        span.style.display = 'inline';  // change CSSOM property  
        // create a new element, style it, and append it to the DOM  
        var loadTime = document.createElement('div');  
        loadTime.textContent = 'You loaded this page on: ' + new Date();  
        loadTime.style.color = 'blue';  
        document.body.appendChild(loadTime);  
      </script>  
    </body>  
  </html>
* JavaScript 允许我们进入 DOM 并拉取对隐藏的 span 节点的引用 - 该节点可能未出现在渲染树中，却仍然存在于 DOM 内。然后，在我们获得引用后，就可以更改其文本（通过 .textContent），甚至可以将其计算的 display 样式属性从“none”替换为“inline”。现在，我们的页面显示“**Hello interactive students!**”。
* JavaScript 还允许我们在 DOM 中创建、样式化、追加和移除新元素。从技术上讲，我们的整个页面可以是一个大的 JavaScript 文件，此文件能够逐一创建元素并对其进行样式化。尽管这种方法可行，但是在实践中，使用 HTML 和 CSS 要简单得多。在 JavaScript 函数的第二部分，我们会创建一个新的 div 元素，设置其文本内容，对其进行样式化，然后将其追加到正文中。
* 我们通过以上示例修改了现有 DOM 节点的内容和 CSS 样式，并为文档添加了一个全新的节点。我们的网页不会赢得任何设计奖，但它说明了 JavaScript 赋予我们的能力和灵活性。
* 不过，尽管 JavaScript 为我们带来了许多功能，不过也在页面渲染方式和时间方面施加了更多限制。

首先，请注意上例中的内联脚本靠近网页底部。为什么呢？您真应该亲自尝试一下。如果我们将脚本移至 span\_元素之上，您就会注意到脚本运行失败，并提示在文档中找不到对任何 \_span 元素的引用 - 即 getElementsByTagName(‘span') 会返回 null。这透露出一个重要事实：我们的脚本在文档的何处插入，就在何处执行。当 HTML 解析器遇到一个 script 标记时，它会暂停构建 DOM，将控制权移交给 JavaScript 引擎；等 JavaScript 引擎运行完毕，浏览器会从中断的地方恢复 DOM 构建。

换言之，我们的脚本块找不到网页中任何靠后的元素，因为它们尚未接受处理！或者，稍微换个说法：**执行我们的内联脚本会阻止 DOM 构建，也就延缓了首次渲染。**

在网页中引入脚本的另一个微妙事实是，它们不仅可以读取和修改 DOM 属性，还可以读取和修改 CSSOM 属性。实际上，我们在示例中就是这么做的：将 span 元素的 display 属性从 none 更改为 inline。最终结果如何？我们现在遇到了竞态问题。

如果浏览器尚未完成 CSSOM 的下载和构建，而我们却想在此时运行脚本，会怎样？答案很简单，对性能不利：**浏览器将延迟脚本执行和 DOM 构建，直至其完成 CSSOM 的下载和构建。**

简言之，JavaScript 在 DOM、CSSOM 和 JavaScript 执行之间引入了大量新的依赖关系，从而可能导致浏览器在处理以及在屏幕上渲染网页时出现大幅延迟：

* 脚本在文档中的位置很重要。
* 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。
* JavaScript 可以查询和修改 DOM 与 CSSOM。
* JavaScript 执行将暂停，直至 CSSOM 就绪。

“优化关键渲染路径”在很大程度上是指了解和优化 HTML、CSS 和 JavaScript 之间的依赖关系谱。

默认情况下，JavaScript 执行会“阻止解析器”：当浏览器遇到文档中的脚本时，它必须暂停 DOM 构建，将控制权移交给 JavaScript 运行时，让脚本执行完毕，然后再继续构建 DOM。我们在前面的示例中已经见过内联脚本的实用情况。实际上，内联脚本始终会阻止解析器，除非您编写额外代码来推迟它们的执行。

通过 script 标签引入的脚本又怎样？让我们还用前面的例子，将代码提取到一个单独文件中：

<html>  
  <head>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <link href="style.css" rel="stylesheet">  
    <title>Critical Path: Script External</title>  
  </head>  
  <body>  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
    <script src="app.js"></script>  
  </body>  
</html>

**app.js**

var span = document.getElementsByTagName('span')[0];  
span.textContent = 'interactive'; // change DOM text content  
span.style.display = 'inline';  // change CSSOM property  
// create a new element, style it, and append it to the DOM  
var loadTime = document.createElement('div');  
loadTime.textContent = 'You loaded this page on: ' + new Date();  
loadTime.style.color = 'blue';  
document.body.appendChild(loadTime);

无论我们使用 <script> 标记还是内联 JavaScript 代码段，您都可以期待两者能够以相同方式工作。 在两种情况下，浏览器都会先暂停并执行脚本，然后才会处理剩余文档。不过，**如果是外部 JavaScript 文件，浏览器必须停下来，等待从磁盘、缓存或远程服务器获取脚本，这就可能给关键渲染路径增加数十至数千毫秒的延迟。**

默认情况下，所有 JavaScript 都会阻止解析器。由于浏览器不了解脚本计划在页面上执行什么操作，它会作最坏的假设并阻止解析器。向浏览器传递脚本不需要在引用位置执行的信号既可以让浏览器继续构建 DOM，也能够让脚本在就绪后执行；例如，在从缓存或远程服务器获取文件后执行。

为此，我们可以将脚本标记为\_异步\_：

<html>  
  <head>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <link href="style.css" rel="stylesheet">  
    <title>Critical Path: Script Async</title>  
  </head>  
  <body>  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
    <script src="app.js" async></script>  
  </body>  
</html>

向 script 标记添加异步关键字可以指示浏览器在等待脚本可用期间不阻止 DOM 构建，这样可以显著提升性能。

优化关键渲染路径能够让浏览器尽可能快地绘制网页：更快的网页渲染速度可以提高吸引力、增加网页浏览量以及[提高转化率](https://www.google.com/think/multiscreen/success.html)。为了最大程度减少访客看到空白屏幕的时间，我们需要优化加载的资源及其加载顺序。

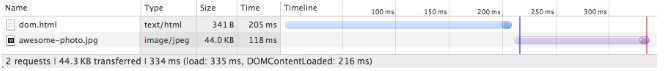
到目前为止，我们只关注了资源（CSS、JS 或 HTML 文件）可供处理后浏览器中会发生的情况，而忽略了从缓存或从网络获取资源所需的时间。我们作以下假设：

* 到服务器的网络往返（传播延迟时间）需要 100 毫秒。
* HTML 文档的服务器响应时间为 100 毫秒，所有其他文件的服务器响应时间均为 10 毫秒。

## Hello World 体验

<html>  
  <head>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <title>Critical Path: No Style</title>  
  </head>  
  <body>  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
  </body>  
</html>

我们将从基本 HTML 标记和单个图像（无 CSS 或 JavaScript）开始。让我们在 Chrome DevTools 中打开 Network 时间线并检查生成的资源瀑布：



注：尽管本文档使用 DevTools 说明 CRP 概念，DevTools 当前并不非常适合 CRP 分析。 如需了解详细信息，请参阅 [DevTools 如何？](https://developers.google.com/web/fundamentals/performance/critical-rendering-path/measure-crp" \l "devtools)。

正如预期的一样，HTML 文件下载花费了大约 200 毫秒。请注意，蓝线的透明部分表示浏览器在网络上等待（即尚未收到任何响应字节）的时间，而不透明部分表示的是收到第一批响应字节后完成下载的时间。HTML 下载量很小 (<4K)，我们只需单次往返便可获取整个文件。因此，获取 HTML 文档大约需要 200 毫秒，其中一半的时间花费在网络等待上，另一半花费在等待服务器响应上。

当 HTML 内容可用后，浏览器会解析字节，将它们转换成令牌，然后构建 DOM 树。请注意，为方便起见，DevTools 会在底部报告 DOMContentLoaded 事件的时间（216 毫秒），该时间同样与蓝色垂直线相符。HTML 下载结束与蓝色垂直线 (DOMContentLoaded) 之间的间隔是浏览器构建 DOM 树所花费的时间 — 在本例中仅为几毫秒。

请注意，我们的“趣照”并未阻止 domContentLoaded 事件。这证明，我们构建渲染树甚至绘制网页时无需等待页面上的每个资产：**并非所有资源都对快速提供首次绘制具有关键作用**。事实上，当我们谈论关键渲染路径时，通常谈论的是 HTML 标记、CSS 和 JavaScript。图像不会阻止页面的首次渲染，不过，我们当然也应该尽力确保系统尽快绘制图像！

即便如此，系统还是会阻止图像上的 load 事件（也称为 onload）：DevTools 会在 335 毫秒时报告 onload 事件。回想一下，onload 事件标记的点是网页所需的**所有资源**均已下载并经过处理的点，这是加载微调框可以在浏览器中停止微调的点（由瀑布中的红色垂直线标记）。

我们的“Hello World 体验”页面虽然看起来简单，但背后却需要做很多工作。在实践中，我们还需要 HTML 之外的其他资源：我们可能需要 CSS 样式表以及一个或多个用于为网页增加一定交互性的脚本。让我们将两者结合使用，看看效果如何：

<html>  
  <head>  
    <title>Critical Path: Measure Script</title>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <link href="style.css" rel="stylesheet">  
  </head>  
  <body onload="measureCRP()">  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
    <script src="timing.js"></script>  
  </body>  
</html>

添加外部 CSS 和 JavaScript 文件将额外增加两个瀑布请求，浏览器差不多会同时发出这两个请求。不过，**请注意，现在 domContentLoaded 事件与 onload 事件之间的时间差小多了。**

这是怎么回事？

* 与纯 HTML 示例不同，我们还需要获取并解析 CSS 文件才能构建 CSSOM，要想构建渲染树，DOM 和 CSSOM 缺一不可。
* 由于网页上还有一个阻止 JavaScript 文件的解析器，系统会在下载并解析 CSS 文件之前阻止 domContentLoaded 事件：因为 JavaScript 可能会查询 CSSOM，我们必须在下载 CSS 文件之前将其阻止，然后才能执行 JavaScript。

**如果我们用内联脚本替换外部脚本会怎样？**即使直接将脚本内联到网页中，浏览器仍然无法在构建 CSSOM 之前执行脚本。简言之，内联 JavaScript 也会阻止解析器。

我们减少了一个请求，但 onload 和 domContentLoaded 时间实际上没有变化。为什么呢？怎么说呢，我们知道，这与 JavaScript 是内联的还是外部的并无关系，因为只要浏览器遇到 script 标记，就会进行阻止，并等到 CSSOM 构建完毕。此外，在我们的第一个示例中，浏览器是并行下载 CSS 和 JavaScript，并且差不多是同时完成。在此实例中，内联 JavaScript 代码并无多大意义。但是，我们可以通过多种策略加快网页的渲染速度。

首先回想一下，所有内联脚本都会阻止解析器，但对于外部脚本，我们可以添加“async”关键字来解除对解析器的阻止。让我们撤消内联，尝试一下

效果好多了！解析 HTML 之后不久即会触发 domContentLoaded 事件；浏览器已得知不要阻止 JavaScript，并且由于没有其他阻止解析器的脚本，CSSOM 构建也可同步进行了。

或者，我们也可以同时内联 CSS 和 JavaScript：

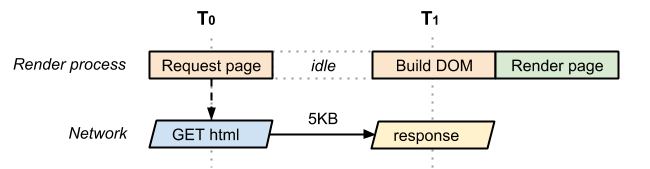
<html>  
  <head>  
    <title>Critical Path: Measure Inlined</title>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <style>  
      p { font-weight: bold }  
      span { color: red }  
      p span { display: none }  
      img { float: right }  
    </style>  
  </head>  
  <body>  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
    <script>  
      var span = document.getElementsByTagName('span')[0];  
      span.textContent = 'interactive'; // change DOM text content  
      span.style.display = 'inline';  // change CSSOM property  
      // create a new element, style it, and append it to the DOM  
      var loadTime = document.createElement('div');  
      loadTime.textContent = 'You loaded this page on: ' + new Date();  
      loadTime.style.color = 'blue';  
      document.body.appendChild(loadTime);  
    </script>  
  </body>  
</html>

请注意，domContentLoaded 时间与前一示例中的时间实际上相同；只不过没有将 JavaScript 标记为异步，而是同时将 CSS 和 JS 内联到网页本身。这会使 HTML 页面显著增大，但好处是浏览器无需等待获取任何外部资源，网页已经内置了所有资源。

如您所见，即便是非常简单的网页，优化关键渲染路径也并非轻而易举：我们需要了解不同资源之间的依赖关系图，我们需要确定哪些资源是“关键资源”，我们还必须在不同策略中做出选择，找到在网页上加入这些资源的恰当方式。这一问题不是一个解决方案能够解决的，每个页面都不尽相同。您需要遵循相似的流程，自行找到最佳策略。

最简单的网页只包括 HTML 标记；没有 CSS，没有 JavaScript，也没有其他类型的资源。要渲染此类网页，浏览器需要发起请求，等待 HTML 文档到达，对其进行解析，构建 DOM，最后将其渲染在屏幕上：

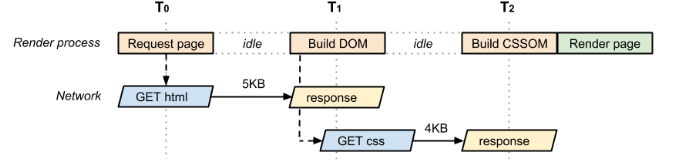
<html>  
  <head>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <title>Critical Path: No Style</title>  
  </head>  
  <body>  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
  </body>  
</html>



**T0 与 T1 之间的时间捕获的是网络和服务器处理时间。**在最理想的情况下（如果 HTML 文件较小），我们只需一次网络往返便可获取整个文档。由于 TCP 传输协议工作方式的缘故，较大文件可能需要更多次的往返。**因此，在最理想的情况下，上述网页具有单次往返（最少）关键渲染路径。**

现在，我们还以同一网页为例，但这次使用外部 CSS 文件：

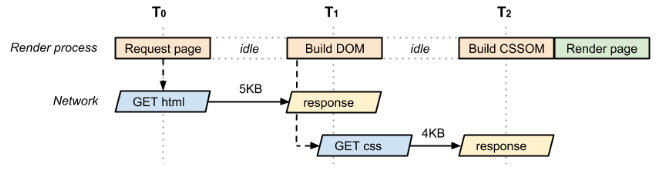
<html>  
  <head>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <link href="style.css" rel="stylesheet">  
  </head>  
  <body>  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
  </body>  
</html>



我们同样需要一次网络往返来获取 HTML 文档，然后检索到的标记告诉我们还需要 CSS 文件；这意味着，浏览器需要返回服务器并获取 CSS，然后才能在屏幕上渲染网页。**因此，这个页面至少需要两次往返才能显示出来。**CSS 文件同样可能需要多次往返，因此重点在于“最少”。

让我们定义一下用来描述关键渲染路径的词汇：

* **关键资源：** 可能阻止网页首次渲染的资源。
* **关键路径长度：** 获取所有关键资源所需的往返次数或总时间。
* **关键字节：** 实现网页首次渲染所需的总字节数，它是所有关键资源传送文件大小的总和。我们包含单个 HTML 页面的第一个示例包含一项关键资源（HTML 文档）；关键路径长度也与 1 次网络往返相等（假设文件较小），而总关键字节数正好是 HTML 文档本身的传送大小。



现在，让我们将其与上面 HTML + CSS 示例的关键路径特性对比一下：

* **2** 项关键资源
* **2** 次或更多次往返的最短关键路径长度
* **9** KB 的关键字节

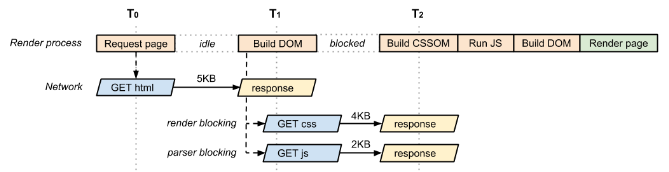
我们同时需要 HTML 和 CSS 来构建渲染树。所以，HTML 和 CSS 都是关键资源：CSS 仅在浏览器获取 HTML 文档后才会获取，因此关键路径长度至少为两次往返。两项资源相加共计 9KB 的关键字节。

我们同时需要 HTML 和 CSS 来构建渲染树。所以，HTML 和 CSS 都是关键资源：CSS 仅在浏览器获取 HTML 文档后才会获取，因此关键路径长度至少为两次往返。两项资源相加共计 9KB 的关键字节。

现在，让我们向组合内额外添加一个 JavaScript 文件。

<html>  
  <head>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <link href="style.css" rel="stylesheet">  
  </head>  
  <body>  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
    <script src="app.js"></script>  
  </body>  
</html>

我们添加了 app.js，它既是网页上的外部 JavaScript 资产，又是一种解析器阻止（即关键）资源。更糟糕的是，为了执行 JavaScript 文件，我们还需要进行阻止并等待 CSSOM；回想一下，JavaScript 可以查询 CSSOM，因此在下载 style.css 并构建 CSSOM 之前，浏览器将会暂停。



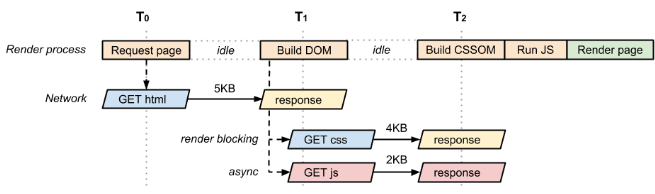
即便如此，如果我们实际查看一下该网页的“网络瀑布”，就会注意到 CSS 和 JavaScript 请求差不多是同时发起的；浏览器获取 HTML，发现两项资源并发起两个请求。因此，上述网页具有以下关键路径特性：

* **3** 项关键资源
* **2** 次或更多次往返的最短关键路径长度
* **11** KB 的关键字节

现在，我们拥有了三项关键资源，关键字节总计达 11 KB，但我们的关键路径长度仍是两次往返，因为我们可以同时传送 CSS 和 JavaScript。**了解关键渲染路径的特性意味着能够确定哪些是关键资源，此外还能了解浏览器如何安排资源的获取时间。**

在与网站开发者交流后，我们意识到我们在网页上加入的 JavaScript 不必具有阻止作用：网页中的一些分析代码和其他代码不需要阻止网页的渲染。了解了这一点，我们就可以向 script 标记添加“async”属性来解除对解析器的阻止：

<html>  
  <head>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <link href="style.css" rel="stylesheet">  
  </head>  
  <body>  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
    <script src="app.js" async></script>  
  </body>  
</html>



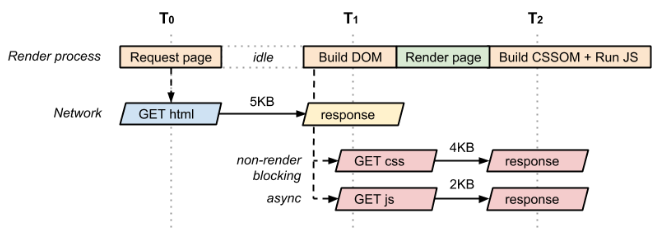
异步脚本具有以下几个优点：

* 脚本不再阻止解析器，也不再是关键渲染路径的组成部分。
* 由于没有其他关键脚本，CSS 也不需要阻止 domContentLoaded 事件。
* domContentLoaded 事件触发得越早，其他应用逻辑开始执行的时间就越早。

因此，我们优化过的网页现在恢复到了具有两项关键资源（HTML 和 CSS），最短关键路径长度为两次往返，总关键字节数为 9 KB。

最后，如果 CSS 样式表只需用于打印，那会如何呢？

<html>  
  <head>  
    <meta name="viewport" content="width=device-width,initial-scale=1">  
    <link href="style.css" rel="stylesheet" media="print">  
  </head>  
  <body>  
    <p>Hello <span>web performance</span> students!</p>  
    <div><img src="awesome-photo.jpg"></div>  
    <script src="app.js" async></script>  
  </body>  
</html>



因为 style.css 资源只用于打印，浏览器不必阻止它便可渲染网页。所以，只要 DOM 构建完毕，浏览器便具有了渲染网页所需的足够信息。因此，该网页只有一项关键资源（HTML 文档），并且最短关键渲染路径长度为一次往返。

为尽快完成首次渲染，我们需要最大限度减小以下三种可变因素：

* 关键资源的数量。
* 关键路径长度。
* 关键字节的数量。
* 关键资源是可能阻止网页首次渲染的资源。这些资源越少，浏览器的工作量就越小，对 CPU 以及其他资源的占用也就越少。
* 同样，关键路径长度受所有关键资源与其字节大小之间依赖关系图的影响：某些资源只能在上一资源处理完毕之后才能开始下载，并且资源越大，下载所需的往返次数就越多。
* 最后，浏览器需要下载的关键字节越少，处理内容并让其出现在屏幕上的速度就越快。要减少字节数，我们可以减少资源数（将它们删除或设为非关键资源），此外还要压缩和优化各项资源，确保最大限度减小传送大小。

**优化关键渲染路径的常规步骤如下：**

1. 对关键路径进行分析和特性描述：资源数、字节数、长度。
2. 最大限度减少关键资源的数量：删除它们，延迟它们的下载，将它们标记为异步等。
3. 优化关键字节数以缩短下载时间（往返次数）。
4. 优化其余关键资源的加载顺序：您需要尽早下载所有关键资产，以缩短关键路径长度。

## 消除阻塞渲染的 JavaScript 和 CSS

要以最快速度完成首次渲染，需要最大限度减少网页上关键资源的数量并（尽可能）消除这些资源，最大限度减少下载的关键字节数，以及优化关键路径长度。

## 优化 JavaScript 的使用

默认情况下，JavaScript 资源会阻塞解析器，除非将其标记为 async 或通过专门的 JavaScript 代码段进行添加。阻塞解析器的 JavaScript 会强制浏览器等待 CSSOM 并暂停 DOM 的构建，继而大大延迟首次渲染的时间。

### 首选使用异步 JavaScript 资源

异步资源不会阻塞文档解析器，让浏览器能够避免在执行脚本之前受阻于 CSSOM。通常，如果脚本可以使用 async 属性，也就意味着它并非首次渲染所必需。可以考虑在首次渲染后异步加载脚本。

### 避免同步服务器调用

使用 navigator.sendBeacon() 方法来限制 XMLHttpRequests 在 unload 处理程序中发送的数据。 因为许多浏览器都对此类请求有同步要求，所以可能减慢网页转换速度，有时还很明显。 以下代码展示了如何利用 navigator.sendBeacon() 向 pagehide 处理程序而不是 unload 处理程序中的服务器发送数据。

<script>  
  function() {  
    window.addEventListener('pagehide', logData, false);  
    function logData() {  
      navigator.sendBeacon(  
        'https://putsreq.herokuapp.com/Dt7t2QzUkG18aDTMMcop',  
        'Sent by a beacon!');  
    }  
  }();  
</script>

新增的 fetch() 方法提供了一种方便的数据异步请求方式。由于它尚未做到随处可用，因此您应该利用功能检测来测试其是否存在，然后再使用。该方法通过 Promise 而非多个事件处理程序来处理响应。不同于对 XMLHttpRequest 的响应，从 Chrome 43 开始，fetch 响应将是 stream 对象。这意味着调用 json() 也会返回 Promise。

<script>  
fetch('./api/some.json')    
  .then(    
    function(response) {    
      if (response.status !== 200) {    
        console.log('Looks like there was a problem. Status Code: ' +  response.status);    
        return;    
      }  
      // Examine the text in the response    
      response.json().then(function(data) {    
        console.log(data);    
      });    
    }    
  )    
  .catch(function(err) {    
    console.log('Fetch Error :-S', err);    
  });  
</script>

fetch() 方法也可处理 POST 请求。

<script>  
fetch(url, {  
  method: 'post',  
  headers: {    
    "Content-type": "application/x-www-form-urlencoded; charset=UTF-8"    
  },    
  body: 'foo=bar&lorem=ipsum'    
}).then(function() { // Aditional code });  
</script>

### 延迟解析 JavaScript

为了最大限度减少浏览器渲染网页的工作量，应延迟任何非必需的脚本（即对构建首次渲染的可见内容无关紧要的脚本）。

### 避免运行时间长的 JavaScript

运行时间长的 JavaScript 会阻止浏览器构建 DOM、CSSOM 以及渲染网页，所以任何对首次渲染无关紧要的初始化逻辑和功能都应延后执行。如果需要运行较长的初始化序列，请考虑将其拆分为若干阶段，以便浏览器可以间隔处理其他事件。

## 优化 CSS 的使用

CSS 是构建渲染树的必备元素，首次构建网页时，JavaScript 常常受阻于 CSS。确保将任何非必需的 CSS 都标记为非关键资源（例如打印和其他媒体查询），并应确保尽可能减少关键 CSS 的数量，以及尽可能缩短传送时间。

### 将 CSS 置于文档 head 标签内

尽早在 HTML 文档内指定所有 CSS 资源，以便浏览器尽早发现 <link> 标记并尽早发出 CSS 请求。

### 避免使用 CSS import

一个样式表可以使用 CSS import (@import) 指令从另一样式表文件导入规则。不过，应避免使用这些指令，因为它们会在关键路径中增加往返次数：只有在收到并解析完带有 @import规则的 CSS 样式表之后，才会发现导入的 CSS 资源。

### 内联阻塞渲染的 CSS

为获得最佳性能，您可能会考虑将关键 CSS 直接内联到 HTML 文档内。这样做不会增加关键路径中的往返次数，并且如果实现得当，在只有 HTML 是阻塞渲染的资源时，可实现“一次往返”关键路径长度。