浏览器渲染过程与性能优化

2017-10-03

| 分类于 前端， 浏览器 | | 阅读次数

大家都知道万维网的应用层使用了HTTP协议，并且用浏览器作为入口访问网络上的资源。用户在使用浏览器访问一个网站时需要先通过HTTP协议向服务器发送请求，之后服务器返回HTML文件与响应信息。这时，浏览器会根据HTML文件来进行解析与渲染（该阶段还包括向服务器请求非内联的CSS文件与JavaScript文件或者其他资源），最终再将页面呈现在用户面前。

现在知道了网页的渲染都是由浏览器完成的，那么如果一个网站的页面加载速度太慢会导致用户体验不够友好，本文通过详解浏览器渲染页面的过程来引入一些基本的浏览器性能优化方案。让浏览器更快地渲染你的网页并快速响应从而提高用户体验。

> 本文作者为: SylvanasSun(sylvanas.sun@gmail.com).转载请务必将下面这段话置于文章开头处(保留超链接).
> 本文首发自SylvanasSun Blog,原文链接:
> https://sylvanassun.github.io/2017/10/03/2017-10-03-BrowserCriticalRenderingPath

关键渲染路径

浏览器接收到服务器返回的HTML、CSS和JavaScript字节数据并对其进行解析和转变成像素的渲染过程被称为关键渲染路径。通过优化关键渲染路径即可以缩短浏览器渲染页面的时间。

浏览器在渲染页面前需要先构建出DOM树与CSSOM树（如果没有DOM树和CSSOM树就无法确定页面的结构与样式，所以这两项是必须先构建出来的）。

DOM树全称为Document Object Model文档对象模型，它是HTML和XML文档的编程接口，提供了对文档的结构化表示，并定义了一种可以使程序对该结构进行访问的方式（比如JavaScript就是通过DOM来操作结构、样式和内

容）。DOM将文档解析为一个由节点和对象组成的集合，可以说一个WEB页面其实就是一个DOM。

CSSOM树全称为Cascading Style Sheets Object Model层叠样式表对象模型，它与DOM树的含义相差不大，只不过它是CSS的对象集合。

构建DOM树与CSSOM树

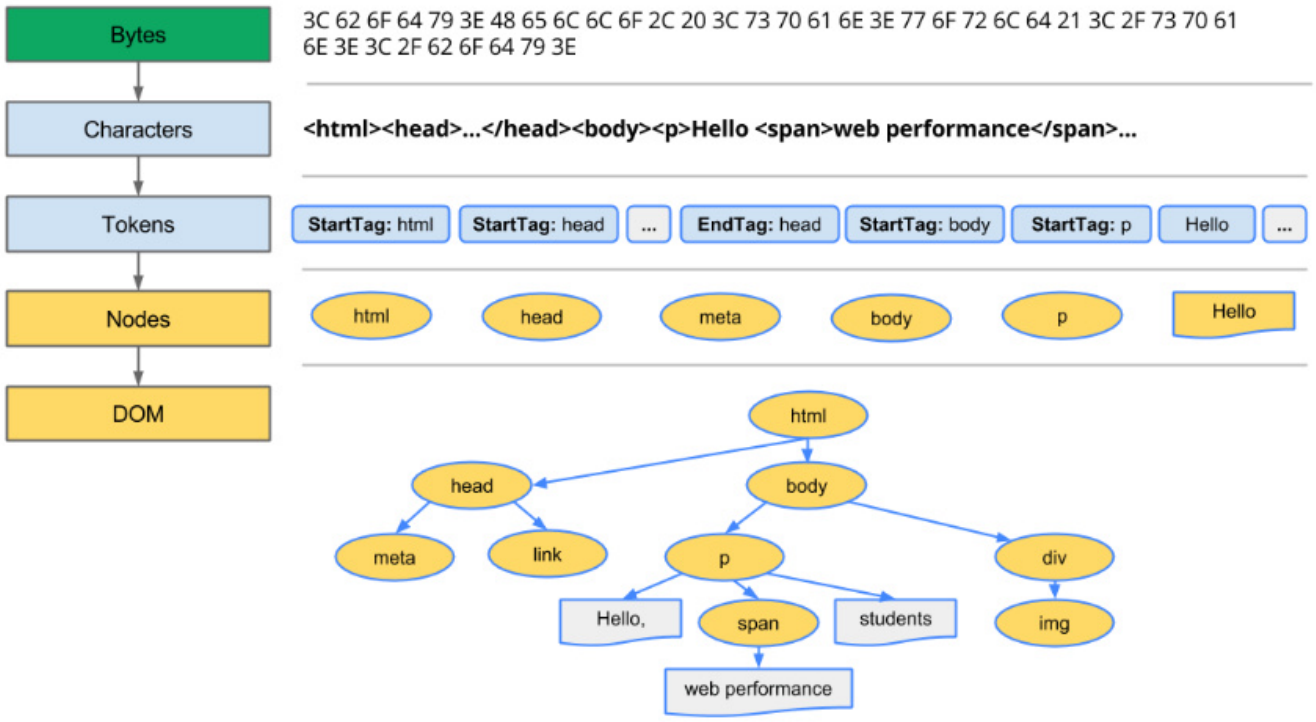
浏览器从网络或硬盘中获得HTML字节数据后会经过一个流程将字节解析为DOM树：

- 编码： **先将HTML的原始字节数据转换为文件指定编码的字符**。
- 令牌化： **然后浏览器会根据HTML规范来将字符串转换成各种令牌**（如<html>、<body>这样的标签以及标签中的字符串和属性等都会被转化为令牌，每个令牌具有特殊含义和一组规则）。令牌记录了标签的开始与结束，通过这个特性可以轻松判断一个标签是否为子标签（假设有<html>与<body>两个标签，当<html>标签的令牌还未遇到它的结束令牌</html>就遇见了<body>标签令牌，那么<body>就是<html>的子标签）。
- 生成对象： **接下来每个令牌都会被转换成定义其属性和规则的对象（这个对象就是节点对象）**。
- 构建完毕： **DOM树构建完成，整个对象集合就像是一棵树形结构**。可能有人会疑惑为什么DOM是一个树形结构，这是因为标签之间含有复杂的父子关系，树形结构正好可以诠释这个关系（CSSOS同理，层叠样式也含有父子关系。例如： `div p {font-size: 18px}`，会先寻找所有p标签并判断它的父标签是否为div之后才会决定要不要采用这个样式进行渲染）。

整个DOM树的构建过程其实就是： **字节 -> 字符 -> 令牌 -> 节点对象 -> 对象模型**，下面将通过一个示例HTML代码与配图更形象地解释这个过程。

```
1  <html>
```

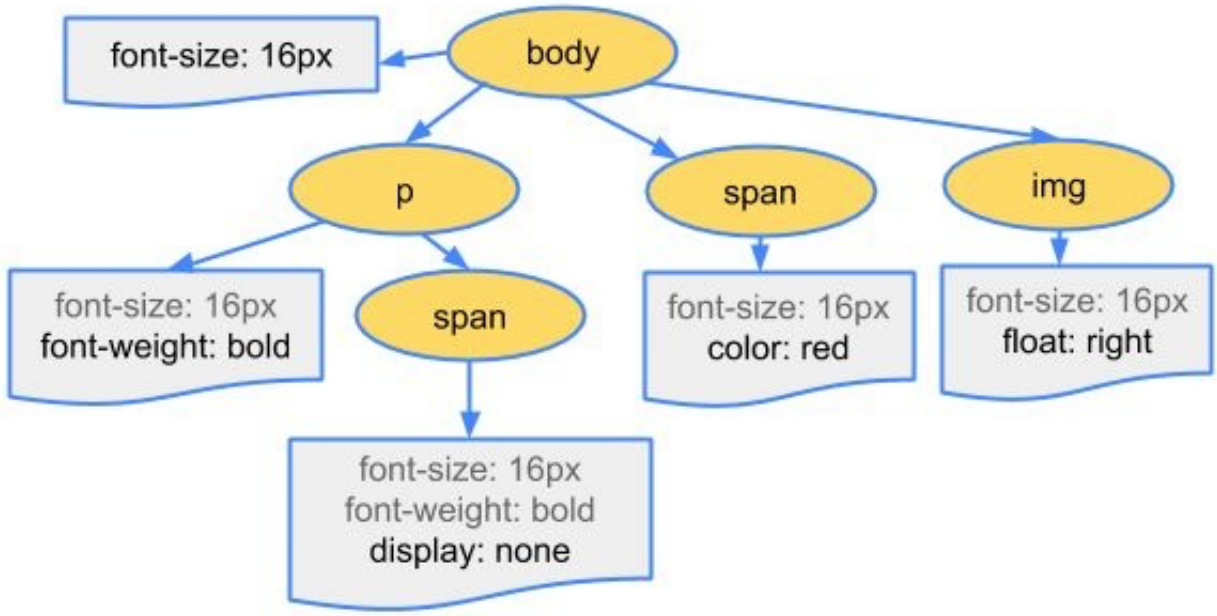
```
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Critical Path</title>
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
  </body>
</html>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



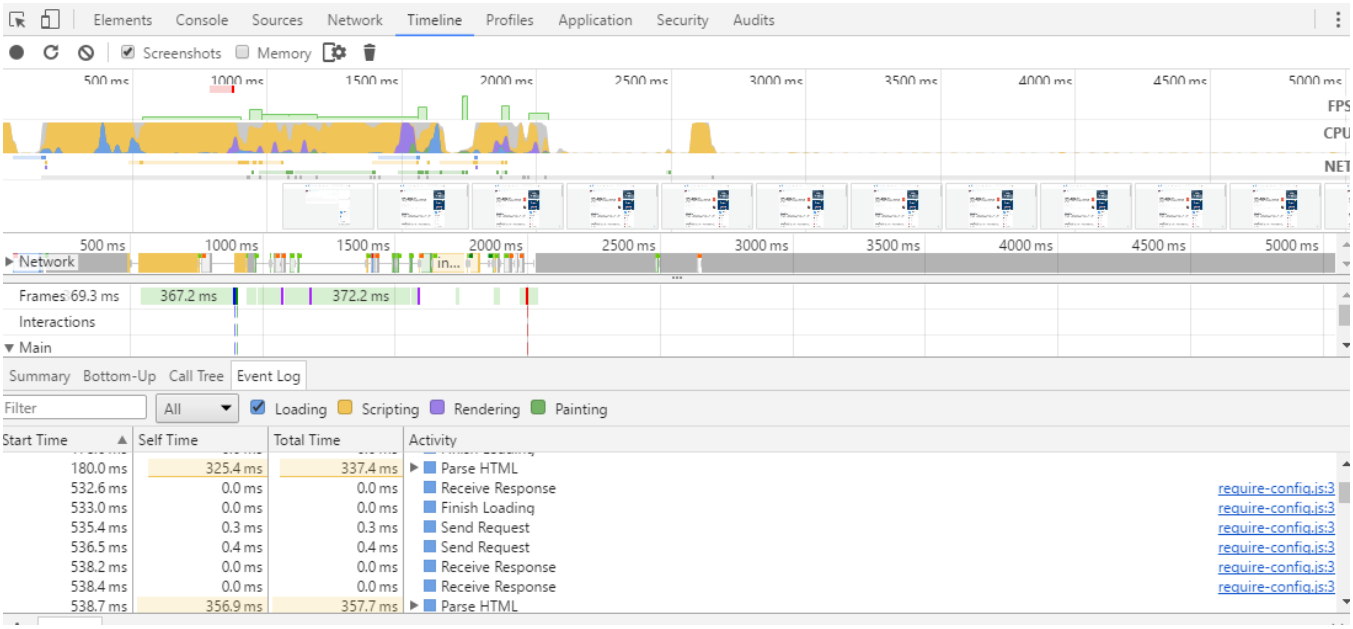
当上述HTML代码遇见<link>标签时，浏览器会发送请求获得该标签中标记的CSS文件（使用内联CSS可以省略请求的步骤提高速度，但没有必要为了这点速度而丢失了模块化与可维护性），style.css中的内容如下：

```
body { font-size: 16px }
p { font-weight: bold }
span { color: red }
p span { display: none }
img { float: right }
```

浏览器获得外部CSS文件的数据后，就会像构建DOM树一样开始构建CSSOM树，这个过程没有什么特别的差别。

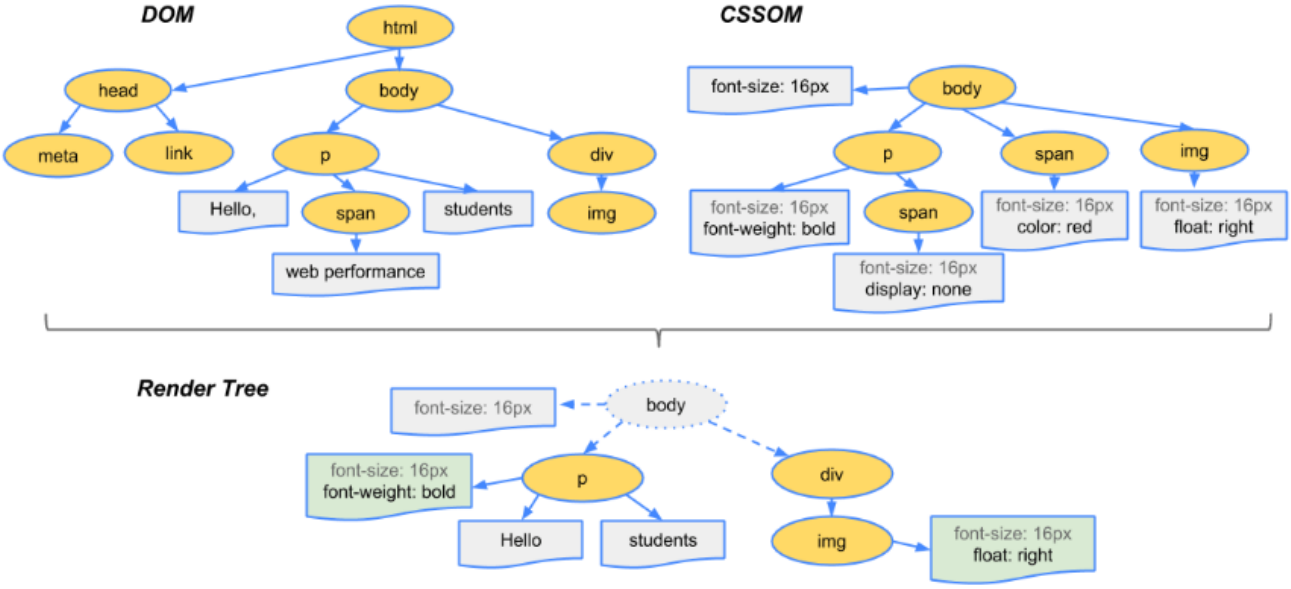


如果想要更详细地去体验一下关键渲染路径的构建，可以使用Chrome开发者工具中的Timeline功能，它记录了浏览器从请求页面资源一直到渲染的各种操作过程，甚至还可以录制某一时间段的过程（建议不要去看太大的网站，信息会比较杂乱）。



构建渲染树

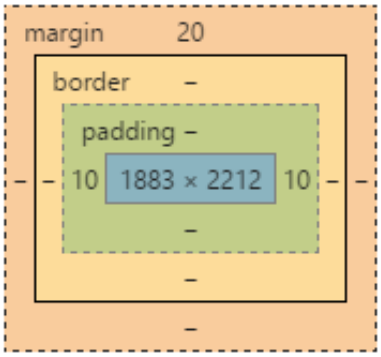
在构建了DOM树和CSSOM树之后，浏览器只是拥有了两个互相独立的对象集合，DOM树描述了文档的结构与内容，CSSOM树则描述了对文档应用的样式规则，**想要渲染出页面，就需要将DOM树与CSSOM树结合在一起**，这就是渲染树。



- 浏览器会先从DOM树的根节点开始遍历每个可见节点（不可见的节点自然就没必要渲染到页面了，不可见的节点还包括被CSS设置了display: none属性的节点，值得注意的是visibility: hidden属性并不算是不可见属性，它的语义是隐藏元素，但元素仍然占据着布局空间，所以它会被渲染成一个空框）。
- 对每个可见节点，找到其适配的CSS样式规则并应用。
- 渲染树构建完成，每个节点都是可见节点并且都含有其内容和对应规则的样式。

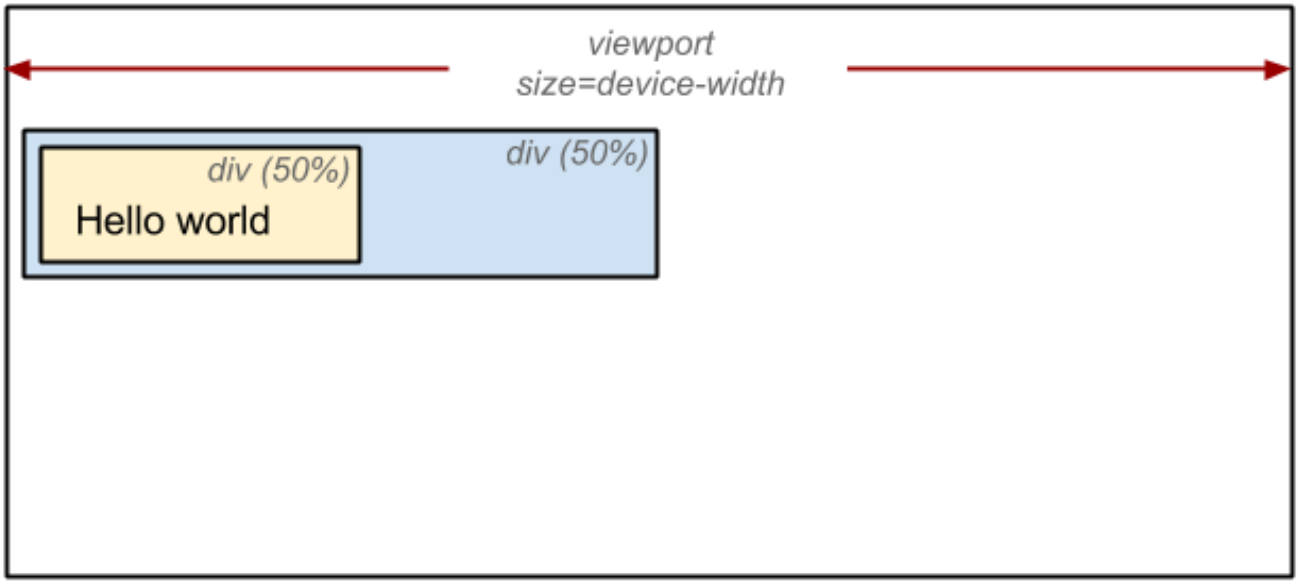
渲染树构建完毕后，浏览器得到了每个可见节点的内容与其样式，下一步工作则**需要计算每个节点在窗口内的确切位置与大小，也就是布局阶段**。

CSS采用了一种叫做盒子模型的思维模型来表示每个节点与其他元素之间的距离，盒子模型包括外边距(Margin)，内边距(Padding)，边框(Border)，内容(Content)。页面中的每个标签其实都是一个个盒子。



布局阶段会从渲染树的根节点开始遍历，然后确定每个节点对象在页面上的确切大小与位置，布局阶段的输出是一个盒子模型，它会精确地捕获每个元素在屏幕内的确切位置与大小，所有相对的测量值也都会被转换为屏幕内的绝对像素值。

```
1   <html>
2     <head>
3       <meta name="viewport" content="width=device-width,initial-scale=1">
4       <title>Critial Path: Hello world!</title>
5     </head>
6     <body>
7       <div style="width: 50%">
8         <div style="width: 50%">Hello world!</div>
9       </div>
10    </body>
11  </html>
```



当Layout布局事件完成后，浏览器会立即发出Paint Setup与Paint事件，开

始将渲染树绘制成像素，绘制所需的时间跟CSS样式的复杂度成正比，绘制完成后，用户就可以看到页面的最终呈现效果了。

我们对一个网页发送请求并获得渲染后的页面可能也就经过了1~2秒，但浏览器其实已经做了上述所讲的非常多的工作，总结一下浏览器关键渲染路径的整个过程：

- 处理HTML标记数据并生成DOM树。
- 处理CSS标记数据并生成CSSOM树。
- 将DOM树与CSSOM树合并在一起生成渲染树。
- 遍历渲染树开始布局，计算每个节点的位置信息。
- 将每个节点绘制到屏幕。

渲染阻塞的优化方案

浏览器想要渲染一个页面就必须先构建出DOM树与CSSOM树，如果HTML与CSS文件结构非常庞大与复杂，这显然会给页面加载速度带来严重影响。

所谓渲染阻塞资源，即是对该资源发送请求后还需要先构建对应的DOM树或CSSOM树，这种行为显然会延迟渲染操作的开始时间。**HTML、CSS、JavaScript都是会对渲染产生阻塞的资源，HTML是必需的（没有DOM还谈何渲染），但还可以从CSS与JavaScript着手优化，尽可能地减少阻塞的产生。**

优化CSS

如果可以让CSS资源只在特定条件下使用，这样这些资源就可以在首次加载时先不进行构建CSSOM树，只有在符合特定条件时，才会让浏览器进行阻塞渲染然后构建CSSOM树。

CSS的媒体查询正是用来实现这个功能的，它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。

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

使用媒体查询可以让CSS资源不在首次加载中阻塞渲染，但不管是哪种CSS资源它们的下载请求都不会被忽略，浏览器仍然会先下载CSS文件

优化JavaScript

当浏览器的HTML解析器遇到一个script标记时会暂停构建DOM，然后将控制权移交至JavaScript引擎，这时引擎会开始执行JavaScript脚本，直到执行结束后，浏览器才会从之前中断的地方恢复，然后继续构建DOM。每次去执行JavaScript脚本都会严重地阻塞DOM树的构建，如果JavaScript脚本还操作了CSSOM，而正好这个CSSOM还没有下载和构建，浏览器甚至会延迟脚本执行和构建DOM，直至完成其CSSOM的下载和构建。显而易见，如果对JavaScript的执行位置运用不当，这将会严重影响渲染的速度。

下面代码中的JavaScript脚本并不会生效，这是因为DOM树还没有构建到<p>标签时，JavaScript脚本就已经开始执行了。这也是为什么经常有人在HTML文件的最下方写内联JavaScript代码，又或者使用window.onload()和JQuery中的$(function(){})（这两个函数有一些区别，window.onload()是等待页面完全加载完毕后触发的事件，而$(function(){})在DOM树构建完毕后就会执行）。

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
    <title>Hello,World</title>
    <script type="text/javascript">
        var p = document.getElementsByTagName('p')[0];
```

```
        p.textContent = 'SylvanasSun';
    </script>
  </head>
  <body>
    <p>Hello,World!</p>
  </body>
</html>
```

使用**async**可以通知浏览器该脚本不需要在引用位置执行，这样浏览器就可以继续构建DOM，JavaScript脚本会在就绪后开始执行，这样将显著提升页面首次加载的性能（async只可以在src标签中使用也就是外部引用的JavaScript文件）。

```
<script type="text/javascript" src="demo_async.js" async="async"></script>
<script type="text/javascript" src="demo_async.js" async></script>
```

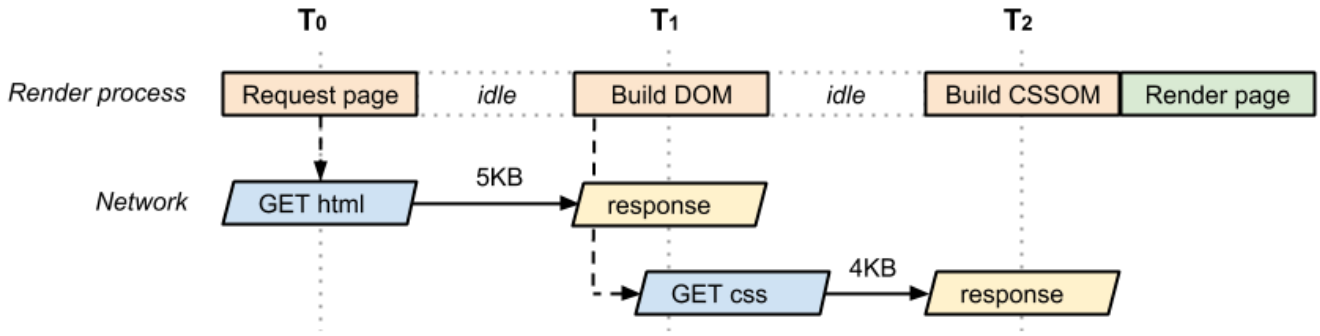
优化关键渲染路径总结

上文已经完整讲述了浏览器是如何渲染页面的以及渲染之前的准备工作，接下来我们以下面的案例来总结一下优化关键渲染路径的方法。

假设有一个HTML页面，它只引入了一个CSS外部文件：

```
<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>
```

它的关键渲染路径如下：



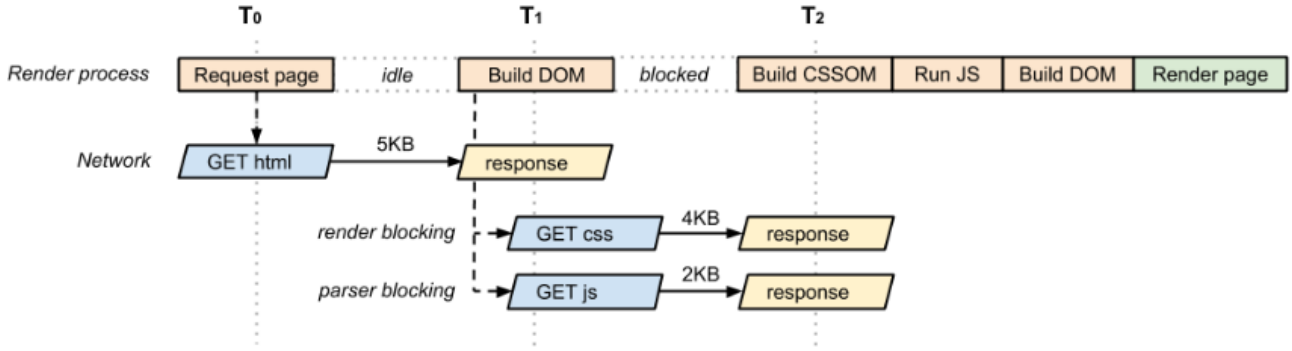
首先浏览器要先对服务器发送请求获得HTML文件，得到HTML文件后开始构建DOM树，在遇见<link>标签时浏览器需要向服务器再次发出请求来获得CSS文件，然后则是继续构建DOM树和CSSOM树，浏览器合并出渲染树，根据渲染树进行布局计算，执行绘制操作，页面渲染完成。

有以下几个用于描述关键渲染路径性能的词汇：

- 关键资源：可能阻塞网页首次渲染的资源（上图中为2个，HTML文件与外部CSS文件style.css）。
- 关键路径长度： 获取关键资源所需的往返次数或总时间（上图为2次或以上，一次获取HTML文件，一次获取CSS文件，这个次数基于TCP协议的最大拥塞窗口，一个文件不一定能在一次连接内传输完毕）。
- 关键字节：所有关键资源文件大小的总和（上图为9KB）。

接下来，案例代码的需求发生了变化，它新增了一个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>
```

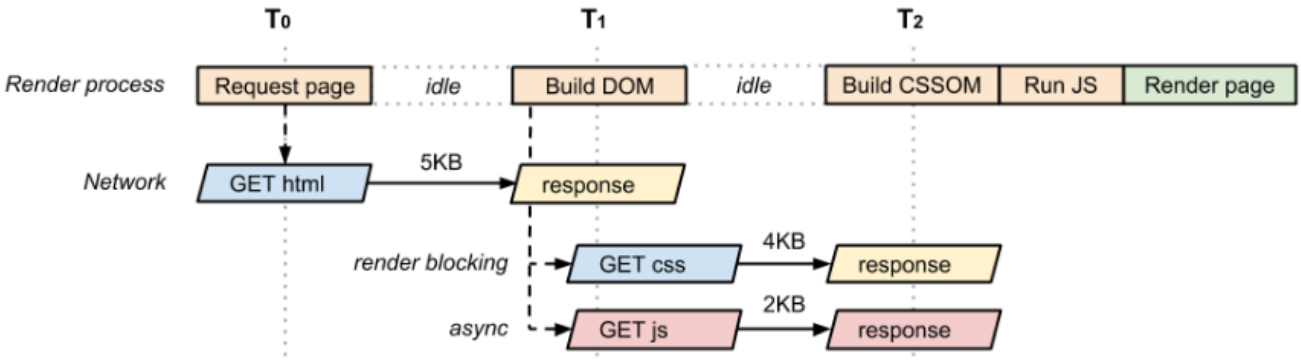


JavaScript文件阻塞了DOM树的构建，并且在执行JavaScript脚本时还需要先等待构建CSSOM树，上图的关键渲染路径特性如下：

- 关键资源： 3（HTML、style.css、app.js）
- 关键路径长度： 2或以上（浏览器会在一次连接中一起下载style.css和app.js）
- 关键字节：11KB

现在，我们要优化关键渲染路径，首先将<script>标签添加异步属性async，这样浏览器的HTML解析器就不会阻塞这个JavaScript文件了。

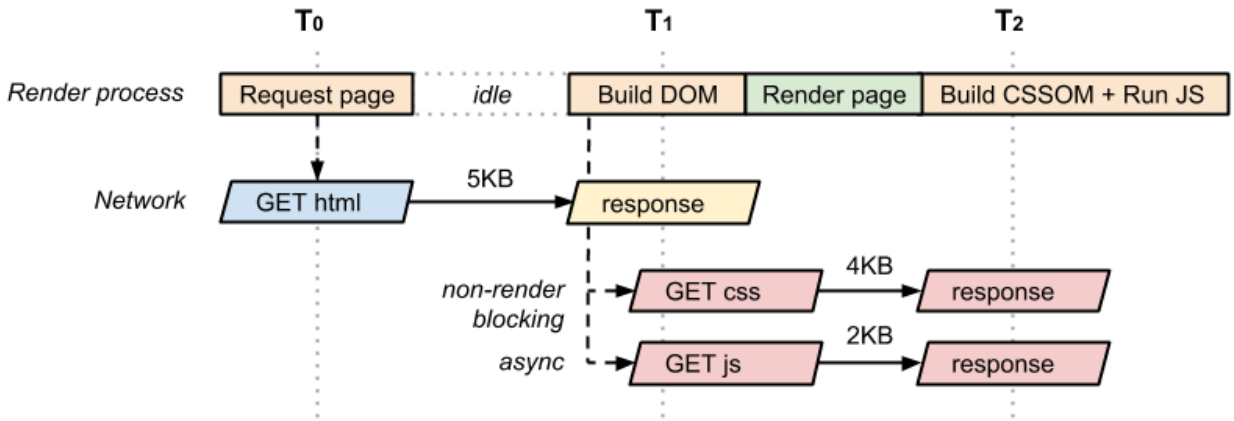
```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <link href="style.css" rel="stylesheet">
  </head>
  <body>
    <p>Hello <span>web performance</span> students!</p>
    <div><img src="awesome-photo.jpg"></div>
    <script src="app.js" async></script>
  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



- 关键资源：2（app.js为异步加载，不会成为阻塞渲染的资源）
- 关键路径长度： 2或以上
- 关键字节： 9KB（app.js不再是关键资源，所以没有算上它的大小）

接下来对css进行优化，比如添加上媒体查询。

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



- 关键资源：1（app.js为异步加载，style.css只有在打印时才会使用，所以只剩下HTML一个关键资源，也就是说当DOM树构建完毕，浏览器就

会开始进行渲染）

- 关键路径长度：1或以上
- 关键字节：5KB

优化关键渲染路径就是在对关键资源、关键路径长度和关键字节进行优化。关键资源越少，浏览器在渲染前的准备工作就越少；同样，关键路径长度和关键字节关系到浏览器下载资源的效率，它们越少，浏览器下载资源的速度就越快。

其他优化方案

除了异步加载JavaScript和使用媒体查询外还有很多其他的优化方案可以使页面的首次加载变得更快，这些方案可以综合起来使用，但核心的思想还是针对关键渲染路径进行了优化。

加载部分HTML

服务端在接收到请求时先只响应回HTML的初始部分，后续的HTML内容在需要时再通过AJAX获得。由于服务端只发送了部分HTML文件，这让构建DOM树的工作量减少很多，从而让用户感觉页面的加载速度很快。

注意，这个方法不能用在CSS上，浏览器不允许CSSOM只构建初始部分，否则会无法确定具体的样式。

压缩

通过对外部资源进行压缩可以大幅度地减少浏览器需要下载的资源量，它会减少关键路径长度与关键字节，使页面的加载速度变得更快。

对数据进行压缩其实就是使用更少的位数来对数据进行重编码。如今有非常多的压缩算法，且每一个的作用领域也各不相同，它们的复杂度也不相同，不过在这里我不会讲压缩算法的细节，感兴趣的朋友可以自己Google。

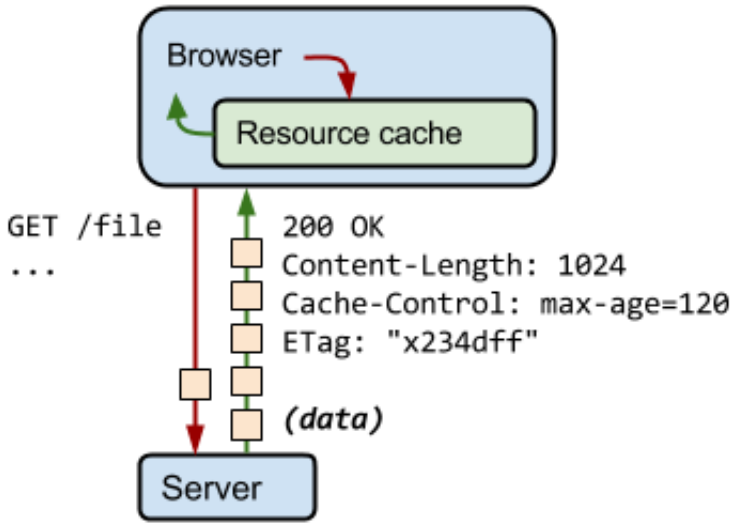
在对HTML、CSS和JavaScript这些文件进行压缩之前，还需要先进行一次冗

余压缩。所谓冗余压缩，就是去除多余的字符，例如注释、空格符和换行符。这些字符对于程序员是有用的，毕竟没有格式化的代码可读性是非常恐怖的，但它们对于浏览器是没有任何意义的，去除这些冗余可以减少文件的数据量。在进行完冗余压缩之后，再使用压缩算法进一步对数据本身进行压缩，例如GZIP（GZIP是一个可以作用于任何字节流的通用压缩算法，它会记忆之前已经看到的内容，然后再尝试查找并替换重复的内容。）。

HTTP缓存

通过网络来获取资源通常是缓慢的，如果资源文件过于膨大，浏览器还需要与服务器之间进行多次往返通信才能获得完整的资源文件。缓存可以复用之前获取的资源，既然后端可以使用缓存来减少访问数据库的开销，那前端自然也可以使用缓存来复用资源文件。

浏览器自带了HTTP缓存的功能，只需要确保每个服务器响应的头部都包含了以下的属性：



- ETag： ETag是一个传递验证令牌，它对资源的更新进行检查，如果资源未发生变化时不会传送任何数据。当浏览器发送一个请求时，会把ETag一起发送到服务器，服务器会根据当前资源核对令牌（ETag通常是对内容进行Hash后得出的一个指纹），如果资源未发生变化，服务器将返回304 Not Modified响应，这时浏览器不必再次下载资源，而是继续复用缓存。

- Cache-Control： Cache-Control定义了缓存的策略，它规定在什么条件

下可以缓存响应以及可以缓存多久。

- no-cache： no-cache表示必须先与服务器确认返回的响应是否发生了变化，然后才能使用该响应来满足后续对同一网址的请求（每次都会根据ETag对服务器发送请求来确认变化，如果未发生变化，浏览器不会下载资源）。
- no-store： no-store直接禁止浏览器以及所有中间缓存存储任何版本的返回响应。简单的说，该策略会禁止任何缓存，每次发送请求时，都会完整地下载服务器的响应。
- public&private： 如果响应被标记为public，则即使它有关联的HTTP身份验证，甚至响应状态代码通常无法缓存，浏览器也可以缓存响应。如果响应被标记为private，那么这个响应通常只为单个用户缓存，因此不允许任何中间缓存（CDN）对其进行缓存，private一般用在缓存用户私人信息页面。
- max-age： max-age定义了从请求时间开始，缓存的最长时间，单位为秒。

资源预加载

Pre-fetching是一种提示浏览器预先加载用户之后可能会使用到的资源的方法。

使用dns-prefetch来提前进行DNS解析，以便之后可以快速地访问另一个主机名（浏览器会在加载网页时对网页中的域名进行解析缓存，这样你在之后的访问时无需进行额外的DNS解析，减少了用户等待时间，提高了页面加载速度）。

```
<link rel="dns-prefetch" href="other.hostname.com">
```

使用prefetch属性可以预先下载资源，不过它的优先级是最低的。

```
<link rel="prefetch"  href="/some_other_resource.jpeg">
```

Chrome允许使用subresource属性指定优先级最高的下载资源（当所有属性为subresource的资源下载完完毕后，才会开始下载属性为prefetch的资源）。

```
<link rel="subresource"  href="/some_other_resource.js">
```

prerender可以预先渲染好页面并隐藏起来，之后打开这个页面会跳过渲染阶段直接呈现在用户面前（推荐对用户接下来必须访问的页面进行预渲染，否则得不偿失）。

```
<link rel="prerender"  href="//domain.com/next_page.html">
```

参考文献

- Web Fundamentals | Google Developers
- Flushing the Document Early | High Performance Web Sites
- Introduction to the DOM - Web APIs | MDN
- How the Browser Pre-loader Makes Pages Load Faster - Andy Davies