22-DOM树：JavaScript是如何影响DOM树构建的？

在上一篇文章中，我们通过开发者工具中的网络面板，介绍了网络请求过程的几种**性能指标**以及对页面加载的影响。

而在渲染流水线中，后面的步骤都直接或者间接地依赖于DOM结构，所以本文我们就继续沿着网络数据流路径来**介绍DOM树是怎么生成的**。然后再基于DOM树的解析流程介绍两块内容：第一个是在解析过程中遇到JavaScript脚本，DOM解析器是如何处理的？第二个是DOM解析器是如何处理跨站点资源的？

什么是DOM

从网络传给渲染引擎的HTML文件字节流是无法直接被渲染引擎理解的，所以要将其转化为渲染引擎能够理解的内部结构，这个结构就是DOM。DOM提供了对HTML文档结构化的表述。在渲染引擎中，DOM有三个层面的作用。

- 从页面的视角来看，DOM是生成页面的基础数据结构。
- 从JavaScript脚本视角来看，DOM提供给JavaScript脚本操作的接口，通过这套接口，JavaScript可以对DOM结构进行访问，从而改变文档的结构、样式和内容。
- 从安全视角来看，DOM是一道安全防护线，一些不安全的内容在DOM解析阶段就被拒之门外了。

简言之，DOM是表述HTML的内部数据结构，它会将Web页面和JavaScript脚本连接起来，并过滤一些不安全的内容。

DOM树如何生成

在渲染引擎内部，有一个叫**HTML解析器（HTMLParser）**的模块，它的职责就是负责将HTML字节流转换为DOM结构。所以这里我们需要先要搞清楚HTML解析器是怎么工作的。

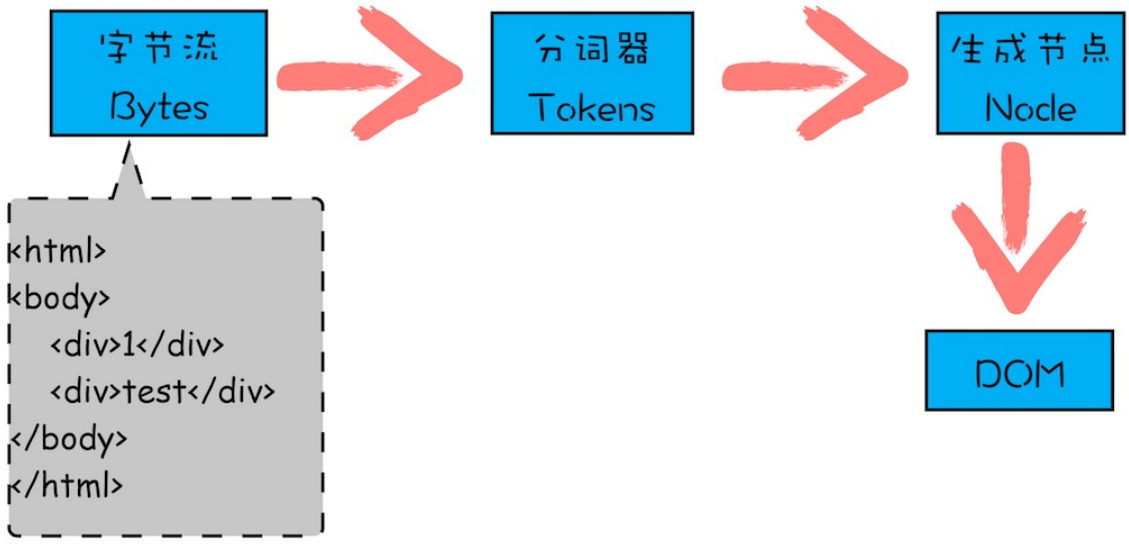
在开始介绍HTML解析器之前，我要先解释一个大家在留言区问到过好多次的问题：**HTML解析器是等整个HTML文档加载完成之后开始解析的，还是随着HTML文档边加载边解析的？**

在这里我统一解答下，HTML解析器并不是等整个文档加载完成之后再解析的，而是**网络进程加载了多少数据，HTML解析器便解析多少数据**。

那详细的流程是怎样的呢？网络进程接收到响应头之后，会根据请求头中的content-type字段来判断文件的类型，比如content-type的值是“text/html”，那么浏览器就会判断这是一个HTML类型的文件，然后为该请求选择或者创建一个渲染进程。渲染进程准备好之后，**网络进程和渲染进程之间会建立一个共享数据的管道**，网络进程接收到数据后就往这个管道里面放，而渲染进程则从管道的另外一端不断地读取数据，并同时将读取的数据“喂”给HTML解析器。你可以把这个管道想象成一个“水管”，网络进程接收到的字节流像水一样倒进这个“水管”，而“水管”的另外一端是渲染进程的HTML解析器，它会动态接收字节流，并将其解析为DOM。

解答完这个问题之后，接下来我们就可以来详细聊聊DOM的具体生成流程了。

前面我们说过代码从网络传输过来是字节流的形式，那么后续字节流是如何转换为DOM的呢？你可以参考下图：



字节流转换为DOM

从图中你可以看出，字节流转换为DOM需要三个阶段。

第一个阶段，通过分词器将字节流转换为Token。

前面《14 | 编译器和解释器：V8是如何执行一段JavaScript代码的？》文章中我们介绍过，V8编译JavaScript过程中的第一步是做词法分析，将JavaScript先分解为一个个Token。解析HTML也是一样的，需要通过分词器先将字节流转换为一个个Token，分为Tag Token和文本Token。上述HTML代码通过词法分析生成的Token如下所示：

生成的Token示意图

由图可以看出，Tag Token又分StartTag 和 EndTag，比如<body>就是StartTag ，</body>就是EndTag，分别对于图中的蓝色和红色块，文本Token对应的绿色块。

至于后续的第二个和第三个阶段是同步进行的，需要将Token解析为DOM节点，并将DOM节点添加到DOM树中。

HTML解析器维护了一个**Token栈结构**，该Token栈主要用来计算节点之间的父子关系，在第一个阶段中生成的Token会被按照顺序压到这个栈中。具体的处理规则如下所示：

- 如果压入到栈中的是**StartTag Token**，HTML解析器会为该Token创建一个DOM节点，然后将该节点加入到DOM树中，它的父节点就是栈中相邻的那个元素生成的节点。
- 如果分词器解析出来是**文本Token**，那么会生成一个文本节点，然后将该节点加入到DOM树中，文本Token是不需要压入到栈中，它的父节点就是当前栈顶Token所对应的DOM节点。

- 如果分词器解析出来的是**EndTag标签**，比如是EndTag div，HTML解析器会查看Token栈顶的元素是否是StarTag div，如果是，就将StartTag div从栈中弹出，表示该div元素解析完成。

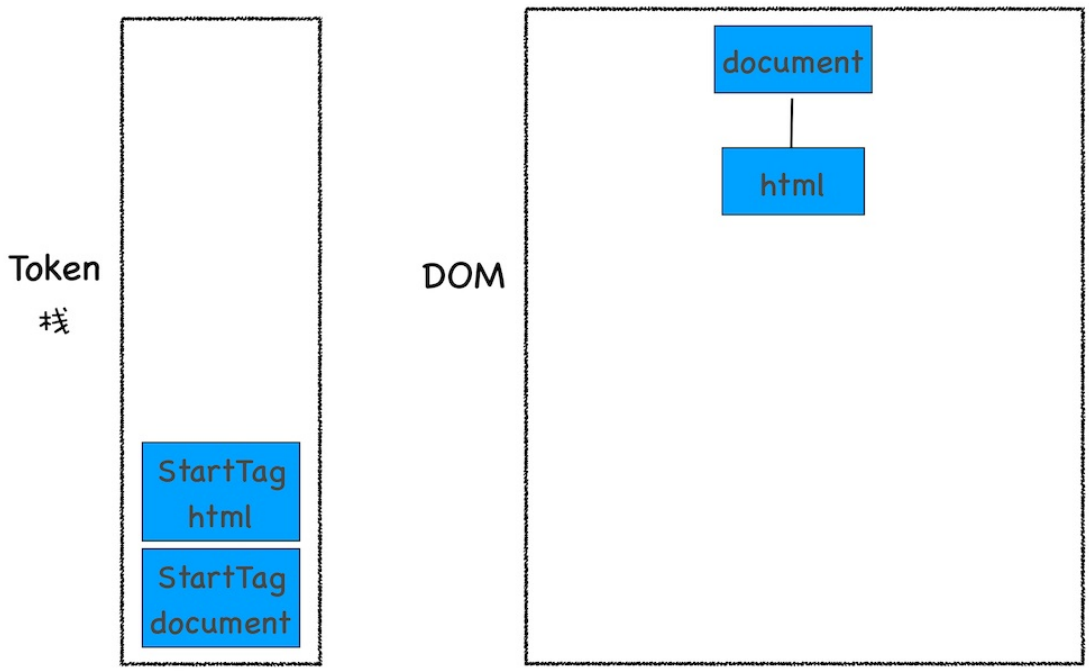
通过分词器产生的新Token就这样不停地压栈和出栈，整个解析过程就这样一直持续下去，直到分词器将所有字节流分词完成。

为了更加直观地理解整个过程，下面我们结合一段HTML代码（如下），来一步步分析DOM树的生成过程。

```
<html>
<body>
    <div>1</div>
    <div>test</div>
</body>
</html>
```

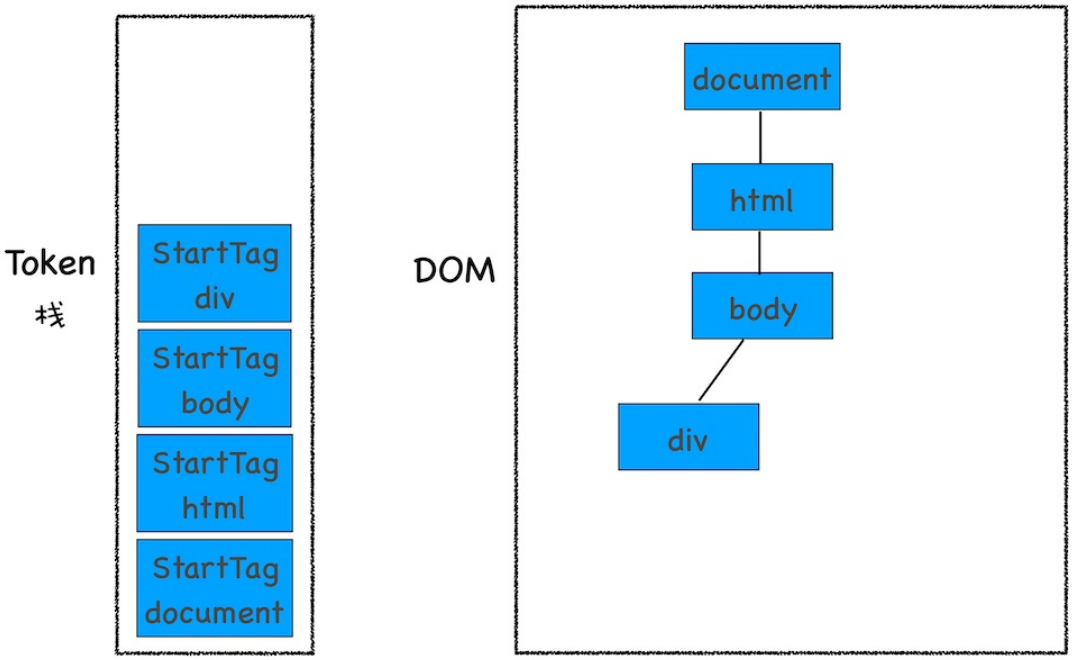
这段代码以字节流的形式传给了HTML解析器，经过分词器处理，解析出来的第一个Token是StartTag html，解析出来的Token会被压入到栈中，并同时创建一个html的DOM节点，将其加入到DOM树中。

这里需要补充说明下，**HTML解析器开始工作时，会默认创建了一个根为document的空DOM结构**，同时会将一个StartTag document的Token压入栈底。然后经过分词器解析出来的第一个StartTag html Token会被压入到栈中，并创建一个html的DOM节点，添加到document上，如下图所示：



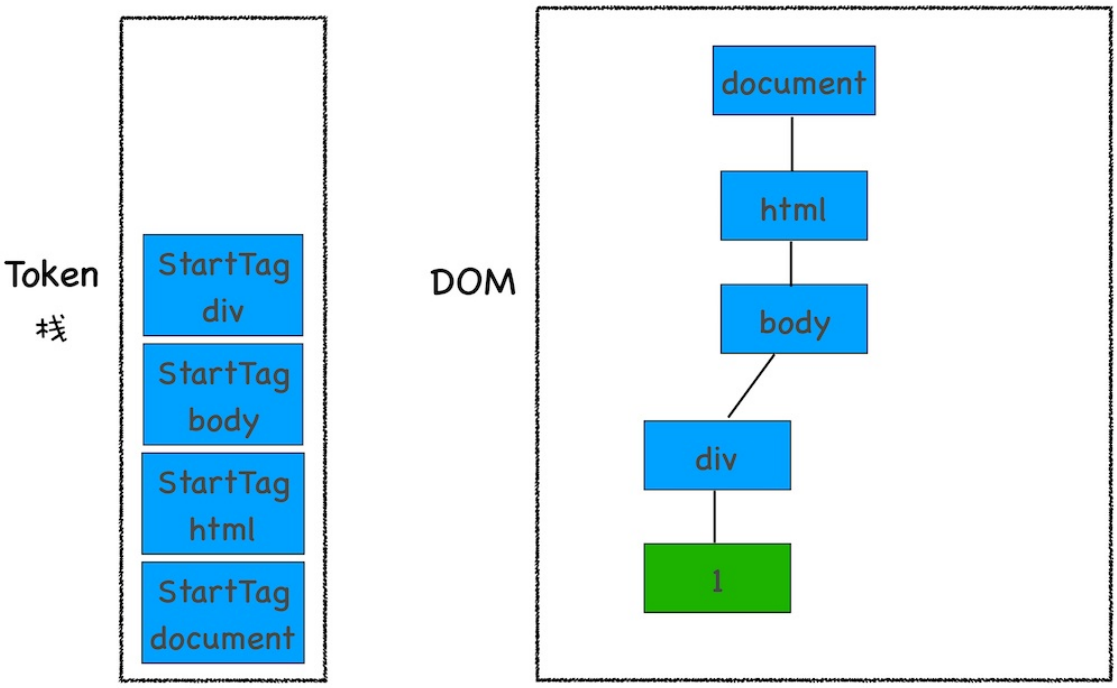
解析到StartTag html时的状态

然后按照同样的流程解析出来StartTag body和StartTag div，其Token栈和DOM的状态如下图所示：



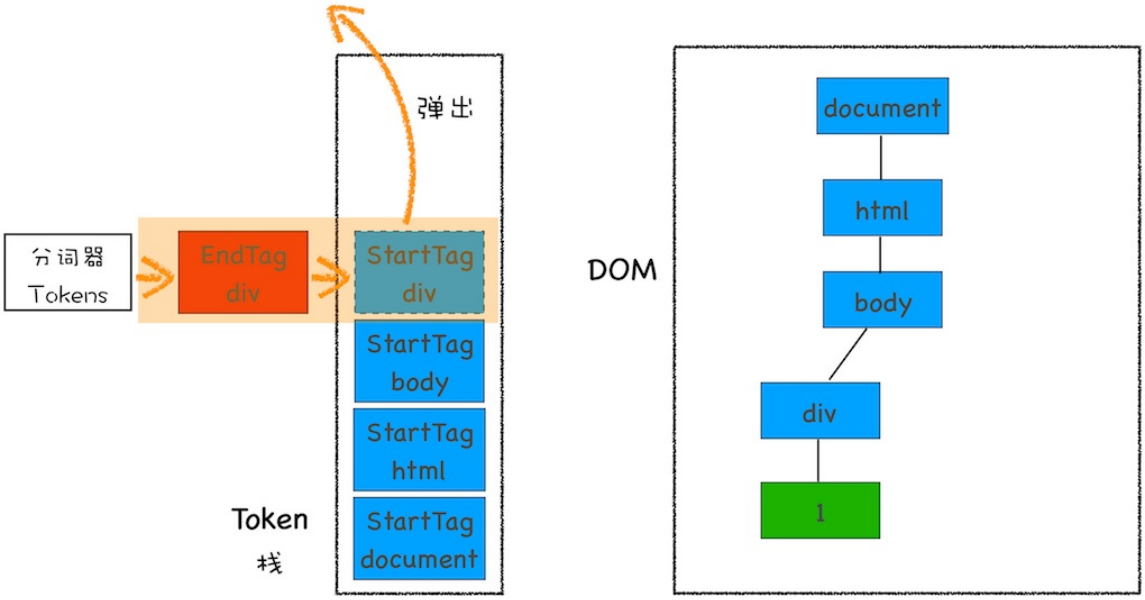
解析到StartTag div时的状态

接下来解析出来的是第一个div的文本Token，渲染引擎会为该Token创建一个文本节点，并将该Token添加到DOM中，它的父节点就是当前Token栈顶元素对应的节点，如下图所示：



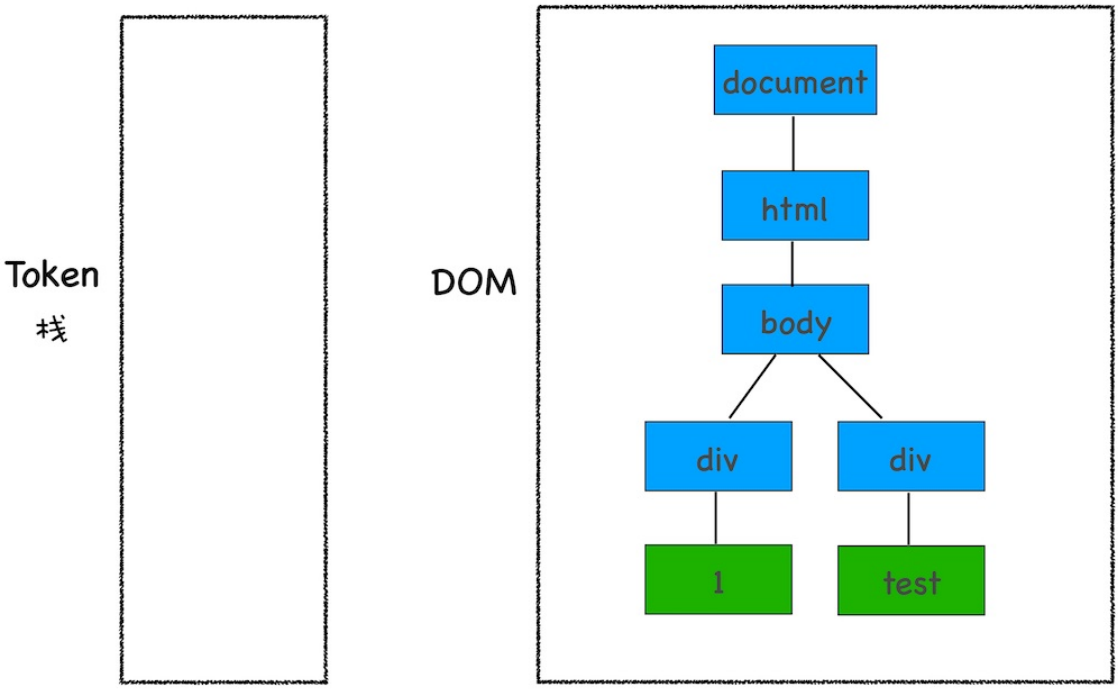
解析出第一个文本Token时的状态

再接下来，分词器解析出来第一个EndTag div，这时候HTML解析器会去判断当前栈顶的元素是否是StartTag div，如果是则从栈顶弹出StartTag div，如下图所示：



元素弹出Token栈示意图

按照同样的规则，一路解析，最终结果如下图所示：



最终解析结果

通过上面的介绍，相信你已经清楚DOM是怎么生成的了。不过在实际生产环境中，HTML源文件中既包含CSS和JavaScript，又包含图片、音频、视频等文件，所以处理过程远比上面这个示范Demo复杂。不过理解了这个简单的Demo生成过程，我们就可以往下分析更加复杂的场景了。

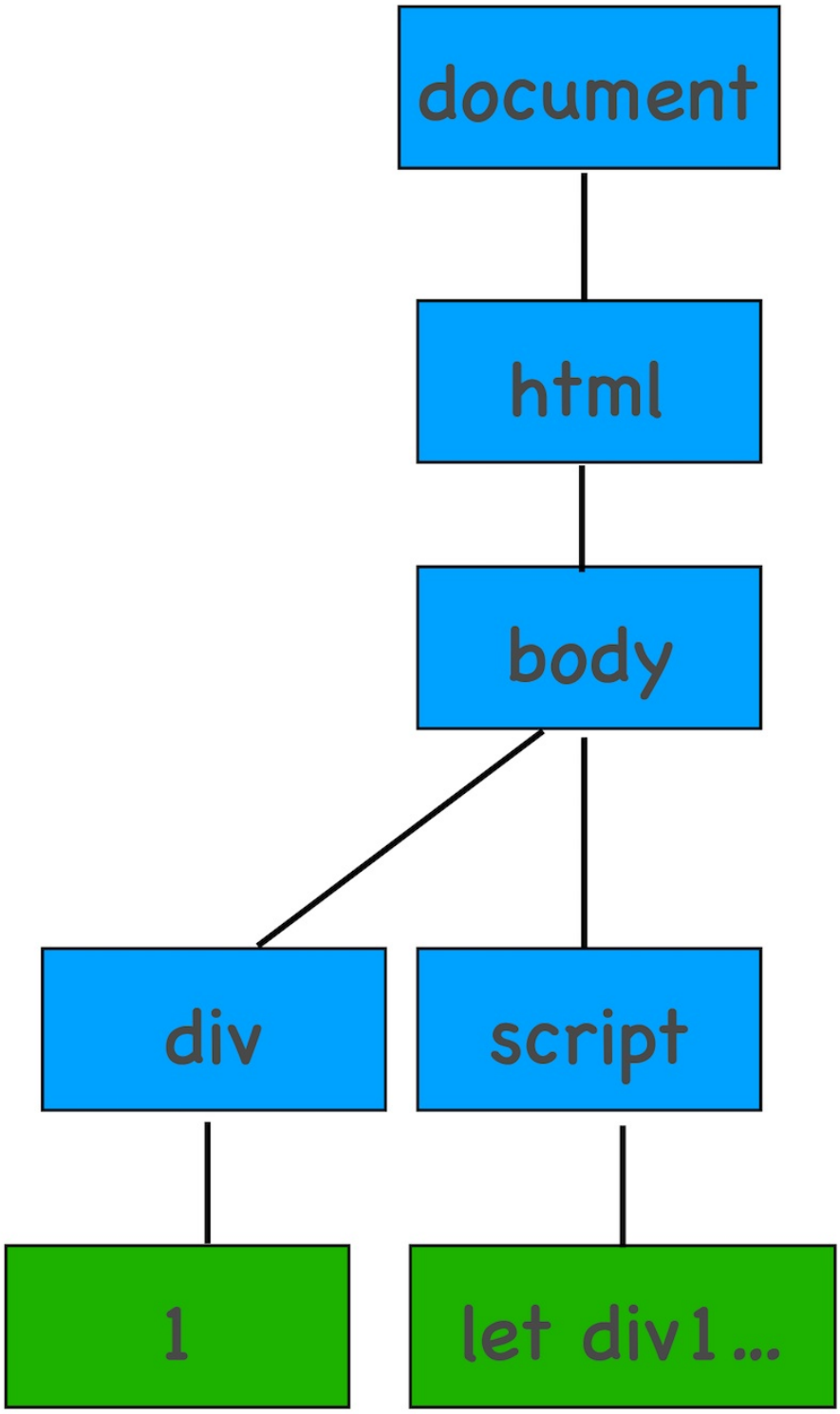
JavaScript是如何影响DOM生成的

我们再来看看稍微复杂点的HTML文件，如下所示：

```
<html>
<body>
    <div>1</div>
    <script>
    let div1 = document.getElementsByTagName('div')[0]
    div1.innerText = 'time.geekbang'
    </script>
    <div>test</div>
</body>
</html>
```

我在两段div中间插入了一段JavaScript脚本，这段脚本的解析过程就有点不一样了。<script>标签之前，所有的解析流程还是和之前介绍的一样，但是解析到<script>标签时，渲染引擎判断这是一段脚本，此时HTML解析器就会暂停DOM的解析，因为接下来的JavaScript可能要修改当前已经生成的DOM结构。

通过前面DOM生成流程分析，我们已经知道当解析到script脚本标签时，其DOM树结构如下所示：



执行脚本时DOM的状态

这时候HTML解析器暂停工作，JavaScript引擎介入，并执行script标签中的这段脚本，因为这段JavaScript脚本修改了DOM中第一个div中的内容，所以执行这段脚本之后，div节点内容已经修改为time.geekbang了。脚本执行完成之后，HTML解析器恢复解析过程，继续解析后续的内容，直至生成最终的DOM。

以上过程应该还是比较好理解的，不过除了在页面中直接内嵌JavaScript脚本之外，我们还通常需要在页面中引入JavaScript文件，这个解析过程就稍微复杂了些，如下面代码：

```
//foo.js
let div1 = document.getElementsByTagName('div')[0]
div1.innerText = 'time.geekbang'
```

```
<html>
<body>
    <div>1</div>
    <script type="text/javascript" src='foo.js'></script>
    <div>test</div>
</body>
</html>
```

这段代码的功能还是和前面那段代码是一样的，不过这里我把内嵌JavaScript脚本修改成了通过JavaScript文件加载。其整个执行流程还是一样的，执行到JavaScript标签时，暂停整个DOM的解析，执行JavaScript代码，不过这里执行JavaScript时，需要先下载这段JavaScript代码。这里需要重点关注下载环境，因为**JavaScript文件的下载过程会阻塞DOM解析**，而通常下载又是非常耗时的，会受到网络环境、JavaScript文件大小等因素的影响。

不过Chrome浏览器做了很多优化，其中一个主要的优化是**预解析操作**。当渲染引擎收到字节流之后，会开启一个预解析线程，用来分析HTML文件中包含的JavaScript、CSS等相关文件，解析到相关文件之后，预解析线程会提前下载这些文件。

再回到DOM解析上，我们知道引入JavaScript线程会阻塞DOM，不过也有一些相关的策略来规避，比如使用CDN来加速JavaScript文件的加载，压缩JavaScript文件的体积。另外，如果JavaScript文件中没有操作DOM相关代码，就可以将该JavaScript脚本设置为异步加载，通过async 或defer来标记代码，使用方式如下所示：

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

```
<script defer type="text/javascript" src='foo.js'></script>
```

async和defer虽然都是异步的，不过还有一些差异，使用async标志的脚本文件一旦加载完成，会立即执行；而使用了defer标记的脚本文件，需要等到DOMContentLoaded事件之后执行。

现在我们知道了JavaScript是如何阻塞DOM解析的了，那接下来我们再来结合文中代码看看另外一种情况：

```
//theme.css
div {color:blue}
```

```
<html>
    <head>
        <style src='theme.css'></style>
    </head>
<body>
    <div>1</div>
    <script>
            let div1 = document.getElementsByTagName('div')[0]
            div1.innerText = 'time.geekbang' //需要DOM
            div1.style.color = 'red'  //需要CSSOM
        </script>
    <div>test</div>
</body>
</html>
```

该示例中，JavaScript代码出现了 `div1.style.color = 'red'` 的语句，它是用来操纵CSSOM的，所以在执行JavaScript之前，需要先解析JavaScript语句之上所有的CSS样式。所以如果代码里引用了外部的CSS文件，那么在执行JavaScript之前，还需要等待外部的CSS文件下载完成，并解析生成CSSOM对象之后，才能执行JavaScript脚本。

而JavaScript引擎在解析JavaScript之前，是不知道JavaScript是否操纵了CSSOM的，所以渲染引擎在遇到JavaScript脚本时，不管该脚本是否操纵了CSSOM，都会执行CSS文件下载，解析操作，再执行JavaScript脚本。

所以说JavaScript脚本是依赖样式表的，这又多了一个阻塞过程。至于如何优化，我们在下篇文章中再来深入探讨。

通过上面的分析，我们知道了JavaScript会阻塞DOM生成，而样式文件又会阻塞JavaScript的执行，所以在实际的工程中需要重点关注JavaScript文件和样式表文件，使用不当会影响到页面性能的。

总结

好了，今天就讲到这里，下面我来总结下今天的内容。

首先我们介绍了DOM是如何生成的，然后又基于DOM的生成过程分析了JavaScript是如何影响到DOM生成的。因为CSS和JavaScript都会影响到DOM的生成，所以我们又介绍了一些加速生成DOM的方案，理解了这些，能让你更加深刻地理解如何去优化首次页面渲染。

额外说明一下，渲染引擎还有一个安全检查模块叫XSSAuditor，是用来检测词法安全的。在分词器解析出来Token之后，它会检测这些模块是否安全，比如是否引用了外部脚本，是否符合CSP规范，是否存在跨站点请求等。如果出现不符合规范的内容，XSSAuditor会对该脚本或者下载任务进行拦截。详细内容我们会在后面的安全模块介绍，这里就不赘述了。

思考时间

看下面这样一段代码，你认为打开这个HTML页面，页面显示的内容是什么？

```
<html>
<body>
    <div>1</div>
    <script>
    let div1 = document.getElementsByTagName('div')[0]
    div1.innerText = 'time.geekbang'

    let div2 = document.getElementsByTagName('div')[1]
    div2.innerText = 'time.geekbang.com'
    </script>
    <div>test</div>
</body>
</html>
```

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言：

- mfist 2019-09-24 06:48:43
  开始看文章的时候就在想如果js获取的dom还没有解析出来，会如何处理，结果思考题就是这个。

  会两行显示，一行是time.geekbang 另外一行是test。原因是script脚本执行的时候获取想不到第二个div，所以不会对后来的div有影响。

  今日总结：
  1. 首先介绍了什么是DOM（表述渲染引擎内部数据结构，它将Web页面和JavaScript脚本连接起来，并过滤不安全内容）、DOM树如何生成（网络进程和渲染进程建立一个流式管道，HTML解析器直接解析，不需要等待text/html类型的接口 接受完毕再进行解析），第一步：通过分词器将字节流转换为Token；第二步：将Token解析为DOM节点；第三步：将DOM节点添加到DOM树中。
  2. JavaScript是如何影响DOM生成的？暂停html解析，下载解析执行完毕js之后再进行html解析（如果这期间使用到了cssDom，需要等待相应css过程）。预解析线程的优化（提前加载相应js css文件）

3. 渲染引擎还有一个安全检查模块XSSAuditor用来检测词法安全的