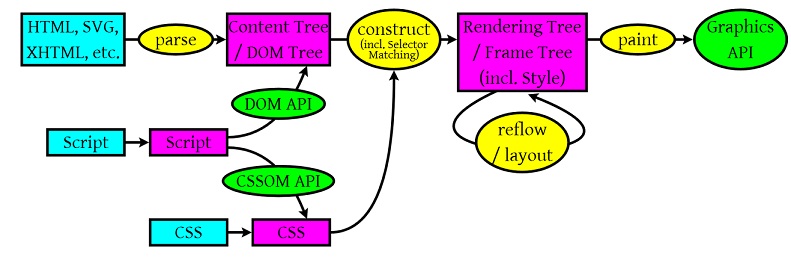
**浏览器工作大流程**



　　从上面这个图中，我们可以看到那么几个事：

　　1）浏览器会解析三个东西：

* 一个是 HTML/SVG/XHTML，事实上，Webkit 有三个 C++ 的类对应这三类文档。解析这三种文件会产生一个 DOM Tree。
* CSS，解析 CSS 会产生 CSS 规则树。
* Javascript，脚本，主要是通过 DOM API 和 CSSOM API 来操作 DOM Tree 和 CSS Rule Tree.

　　2）解析完成后，浏览器引擎会通过 DOM Tree 和 CSS Rule Tree 来构造 Rendering Tree。注意：

* Rendering Tree 渲染树并不等同于 DOM 树，因为一些像 Header 或 display:none 的东西就没必要放在渲染树中了。
* CSS 的 Rule Tree 主要是为了完成匹配并把 CSS Rule 附加上 Rendering Tree 上的每个 Element。也就是 DOM 结点。也就是所谓的 Frame。
* 然后，计算每个 Frame（也就是每个 Element）的位置，这又叫 layout 和 reflow 过程。

　　3）最后通过调用操作系统 Native GUI 的 API 绘制。

**DOM 解析**

　　HTML 的 DOM Tree 解析如下：

<html>

<head>

　　<title>Web page parsing</title>

</head>

<body>

　　<div>

　　<h1>Web page parsing</h1>

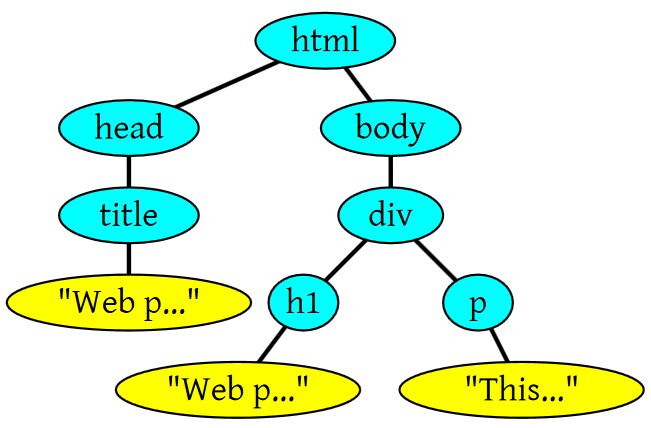
　　<p>This is an example Web page.</p>

　　</div>

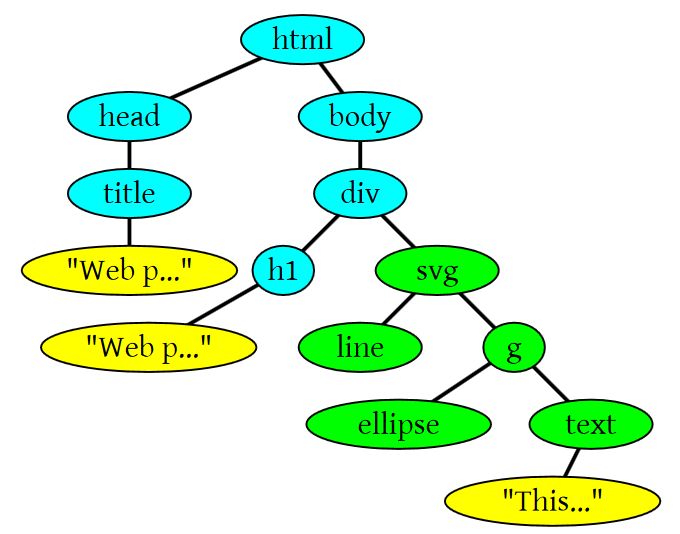
</body>

</html>

　　上面这段 HTML 会解析成这样：



　　下面是另一个有 SVG 标签的情况。



**CSS 解析**

　　CSS 的解析大概是下面这个样子（下面主要说的是 Gecko 也就是 Firefox 的玩法），假设我们有下面的 HTML 文档：

<doc>

<title>A few quotes</title>

<para>

Franklin said that <quote>"A penny saved is a penny earned."</quote>

</para>

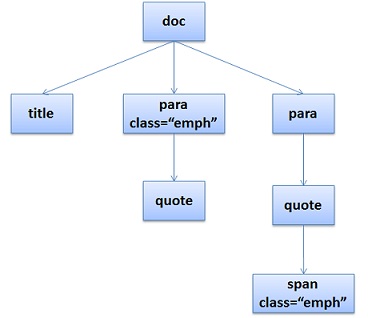
<para>

FDR said <quote>"We have nothing to fear but <span>fear itself.</span>"</quote>

</para>

</doc>

　　于是 DOM Tree 是这个样子：



　　然后我们的 CSS 文档是这样的：

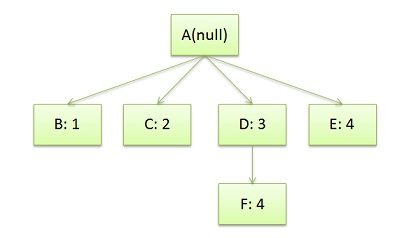
/\* rule 1 \*/ doc { display: block; text-indent: 1em; }

/\* rule 2 \*/ title { display: block; font-size: 3em; }

/\* rule 3 \*/ para { display: block; }

/\* rule 4 \*/ [] { font-style: italic; }

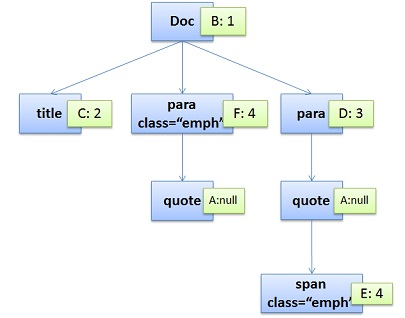
　　于是我们的 CSS Rule Tree 会是这个样子：



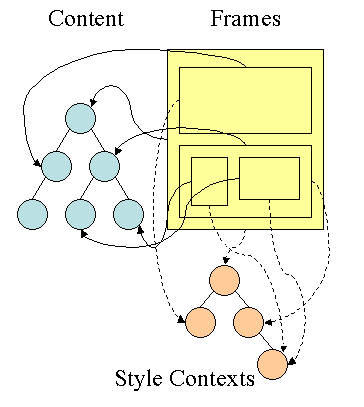
　　注意，图中的第 4 条规则出现了两次，一次是独立的，一次是在规则 3 的子结点。所以，我们可以知道，建立 CSS Rule Tree 是需要比照着 DOM Tree 来的。CSS 匹配 DOM Tree 主要是从右到左解析 CSS 的 Selector，好多人以为这个事会比较快，其实并不一定。关键还看我们的 CSS 的 Selector 怎么写了。

**注意：CSS 匹配 HTML 元素是一个相当复杂和有性能问题的事情。所以，你就会在N多地方看到很多人都告诉你，DOM 树要小，CSS 尽量用 id 和 class，千万不要过渡层叠下去，……**

　　通过这两个树，我们可以得到一个叫 Style Context Tree，也就是下面这样（把 CSS Rule 结点 Attach 到 DOM Tree 上）：



　　所以，Firefox 基本上来说是通过 CSS 解析生成 CSS Rule Tree，然后，通过比对 DOM 生成 Style Context Tree，然后 Firefox 通过把 Style Context Tree 和其 Render Tree（Frame Tree）关联上，就完成了。注意：Render Tree 会把一些不可见的结点去除掉。而 **Firefox 中所谓的 Frame 就是一个 DOM 结点，不要被其名字所迷惑了**。

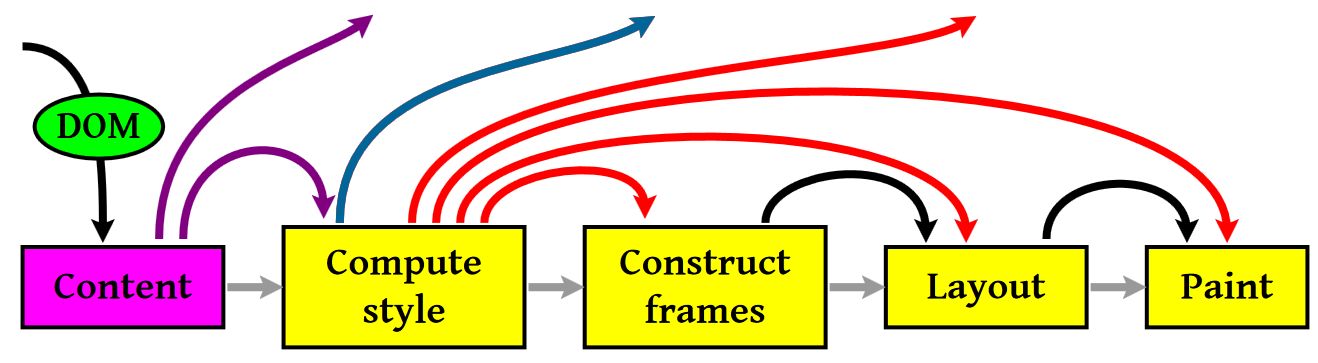


　　注：Webkit 不像 Firefox 要用两个树来干这个，Webkit 也有 Style 对象，它直接把这个 Style 对象存在了相应的 DOM 结点上了。

**渲染**

　　渲染的流程基本上如下（黄色的四个步骤）：

1. 计算 CSS 样式
2. 构建 Render Tree
3. Layout – 定位坐标和大小，是否换行，各种 position, overflow， z-index 属性 ……
4. 正式开画



　　注意：上图流程中有很多连接线，这表示了 Javascript 动态修改了 DOM 属性或是 CSS 属性会导致重新 Layout，有些改变不会，就是那些指到天上的箭头，比如，修改后的 CSS rule 没有被匹配到，等。

　　这里重要要说两个概念，一个是 Reflow，另一个是 Repaint。这两个不是一回事。

* Repaint——屏幕的一部分要重画，比如某个 CSS 的背景色变了。但是元素的几何尺寸没有变。
* Reflow——意味着元件的几何尺寸变了，我们需要重新验证并计算 Render Tree。是 Render Tree 的一部分或全部发生了变化。这就是 Reflow，或是 Layout。（**HTML 使用的是 flow based layout，也就是流式布局，所以，如果某元件的几何尺寸发生了变化，需要重新布局，也就叫 reflow**）reflow 会从 <html> 这个 root frame 开始递归往下，依次计算所有的结点几何尺寸和位置，在 reflow 过程中，可能会增加一些 frame，比如一个文本字符串必需被包装起来。

　　下面是一个打开 Wikipedia 时的 Layout/reflow 的视频（注：HTML 在初始化的时候也会做一次 reflow，叫 *intial reflow*），你可以感受一下：

　　Reflow 的成本比 Repaint 的成本高得多的多。DOM Tree 里的每个结点都会有 reflow 方法，一个结点的 reflow 很有可能导致子结点，甚至父点以及同级结点的 reflow。在一些高性能的电脑上也许还没什么，但是如果 reflow 发生在手机上，那么这个过程是非常痛苦和耗电的。

　　所以，下面这些动作有很大可能会是成本比较高的。

* 当你增加、删除、修改 DOM 结点时，会导致 Reflow 或 Repaint。
* 当你移动 DOM 的位置，或是搞个动画的时候。
* 当你修改 CSS 样式的时候。
* 当你 Resize 窗口的时候（移动端没有这个问题），或是滚动的时候。
* 当你修改网页的默认字体时。

　　注：display:none 会触发 reflow，而 visibility:hidden 只会触发 repaint，因为没有发现位置变化。

　　多说两句关于滚屏的事，通常来说，如果在滚屏的时候，我们的页面上的所有的像素都会跟着滚动，那么性能上没什么问题，因为我们的显卡对于这种把全屏像素往上往下移的算法是很快。但是如果你有一个 fixed 的背景图，或是有些 Element 不跟着滚动，有些 Elment 是动画，那么这个滚动的动作对于浏览器来说会是相当相当痛苦的一个过程。你可以看到很多这样的网页在滚动的时候性能有多差。因为滚屏也有可能会造成 reflow。

　　基本上来说，reflow 有如下的几个原因：

* Initial。网页初始化的时候。
* Incremental。一些 Javascript 在操作 DOM Tree 时。
* Resize。其些元件的尺寸变了。
* StyleChange。如果 CSS 的属性发生变化了。
* Dirty。几个 Incremental 的 reflow 发生在同一个 frame 的子树上。

　　好了，我们来看一个示例吧：

var bstyle = document.body.style; // cache

bstyle.padding = "20px"; // reflow， repaint

bstyle.border = "10px solid red"; // 再一次的 reflow 和 repaint

bstyle.color = "blue"; // repaint

bstyle.backgroundColor = "#fad"; // repaint

bstyle.fontSize = "2em"; // reflow， repaint

// new DOM element - reflow， repaint

document.body.appendChild (document.createTextNode ('dude!'));

　　当然，我们的浏览器是聪明的，它不会像上面那样，你每改一次样式，它就 reflow 或 repaint 一次。**一般来说，浏览器会把这样的操作积攒一批，然后做一次 reflow，这又叫异步 reflow 或增量异步 reflow**。但是有些情况浏览器是不会这么做的，比如：resize 窗口，改变了页面默认的字体，等。对于这些操作，浏览器会马上进行 reflow。

　　但是有些时候，我们的脚本会阻止浏览器这么干，比如：如果我们请求下面的一些 DOM 值：

1. offsetTop, offsetLeft, offsetWidth, offsetHeight
2. scrollTop/Left/Width/Height
3. clientTop/Left/Width/Height
4. IE 中的 getComputedStyle ()， 或 currentStyle

　　因为，如果我们的程序需要这些值，那么浏览器需要返回最新的值，而这样一样会 flush 出去一些样式的改变，从而造成频繁的 reflow/repaint。

**减少 reflow/repaint**

　　下面是一些 Best Practices：

**1）不要一条一条地修改 DOM 的样式。与其这样，还不如预先定义好 css 的 class，然后修改 DOM 的 className。**

// bad

var left = 10,

top = 10;

el.style.left = left + "px";

el.style.top = top + "px";

// Good

el.className += " theclassname";

// Good

el.style.cssText += "; left: " + left + "px; top: " + top + "px;";

**2）把 DOM 离线后修改。如：**

* 使用 documentFragment 对象在内存里操作 DOM。
* 先把 DOM 给 display:none (有一次 repaint)，然后你想怎么改就怎么改。比如修改 100 次，然后再把他显示出来。
* clone 一个 DOM 结点到内存里，然后想怎么改就怎么改，改完后，和在线的那个的交换一下。

　　3）**不要把 DOM 结点的属性值放在一个循环里当成循环里的变量。**不然这会导致大量地读写这个结点的属性。

　　4）**尽可能的修改层级比较低的 DOM**。当然，改变层级比较底的 DOM 有可能会造成大面积的 reflow，但是也可能影响范围很小。

　　5）**为动画的 HTML 元件使用 fixed 或 absoult 的 position**，那么修改他们的 CSS 是不会 reflow 的。

　　6）**千万不要使用 table 布局**。因为可能很小的一个小改动会造成整个 table 的重新布局。