How browsers work

浏览器是如何工作的

Behind the scenes of modern web browsers

浏览器的幕后工作原理

简介

web 浏览器应该是使用最广泛的软件了。在这里，我将要讲述下它们在幕后是怎么工作的。我们来看一下从你在地址栏里输入“google.com”到谷歌的页面显示在浏览器窗口里都发生了什么。

讨论的浏览器

现在有五款主要的浏览器：Internet Explorer、Firefox、Safari、 Chrome 和 Opera。
我会给出几个例子，它们来自于开源浏览器 Firefox、chrome 还有部分开源的 safari。
根据 W3C 浏览器统计信息，当前（2009 年 10 月），Firefox、 Safari 和 Chrome 的市场占有率接近 60%。
所以当前开源浏览器已经成为了浏览器行业的重要组成部分。

浏览器的主要功能

浏览器的主要功能是展现你所需要的 web 资源，它从服务器端请求资源并显示在浏览器窗口上，资源格式一般为 html,但是也可以是 pdf，image 或者其他的格式。资源的地址是由用户通过 URI（统一资源定位符）来指定的，关于这个在网络那章里做更多的介绍。
HTML 和 CSS 规范里指定了浏览器解析和显示 HTML 文件的方式，W3C 维护这些规范，它们是 web 的标准化组织。
HTML 的当前版本是 4，第 5 版正在进行中。CSS 的当前版本是 2，第 3 版正在进行中。
过去的很多年里，各个浏览器都是只遵守规范的一部分，然后做它们自己的扩展，这对于 web 开发者来说引起了严重的兼容性问题，现在浏览器大都或多或少的遵守了规范。
各浏览器的用户界面彼此有很多共同的地方，共同的元素有：
1.键入 URI 的地址栏
2.后退和前进按钮
3.书签菜单
4.用来刷新和停止当前文档加载的刷新和停止按钮
5.帮你返回主页的主页按钮
说来奇怪，并没有任何正式的规范指定浏览器的用户界面，这只是多年试验以及各浏览器互相模仿形成的一个好的做法。HTML5 规范中没有定义浏览器一定会有哪些 UI 元素，但是列举出了一些通用的元素，有地址栏、状态栏和工具栏。当然，特定浏览器还有一些独一无二的特征，比如 Firefox 的下载管理。关于这个在 UI 那章里做更多的介绍。

浏览器的高级结构

浏览器的主要结构如下：

1. 用户界面-包括地址栏、后退/前进按钮、书签菜单等。Every part of the browser display except the main window where you see the requested page.
2. 浏览器引擎- 询问和操作渲染引擎的接口
3. 渲染引擎-负责展现所请求的内容，比如如果请求的是 html，它就负责解析 html 和 css 并且在屏幕上呈现解析后的内容。
4. 网络模块-网络请求用，比如 http 请求。它有跨平台的接口以及对应于每个平台的底层实现。
5. UI 后端-用来绘制基本构件，如下拉框和窗口。UI 后端暴露出一个平台无关性的公共接口，底层的它使用操作系统的 UI 方法。
6. javascript 解释器—用来解析和执行 javascript 代码。
7. 数据存储--这是一个可存储的层。浏览器需要在硬盘上保存各种数据，比如 cookies。新的 html 规范（html5）把浏览器中的“web database”定义为一个完完全全的（虽然很轻）数据库。

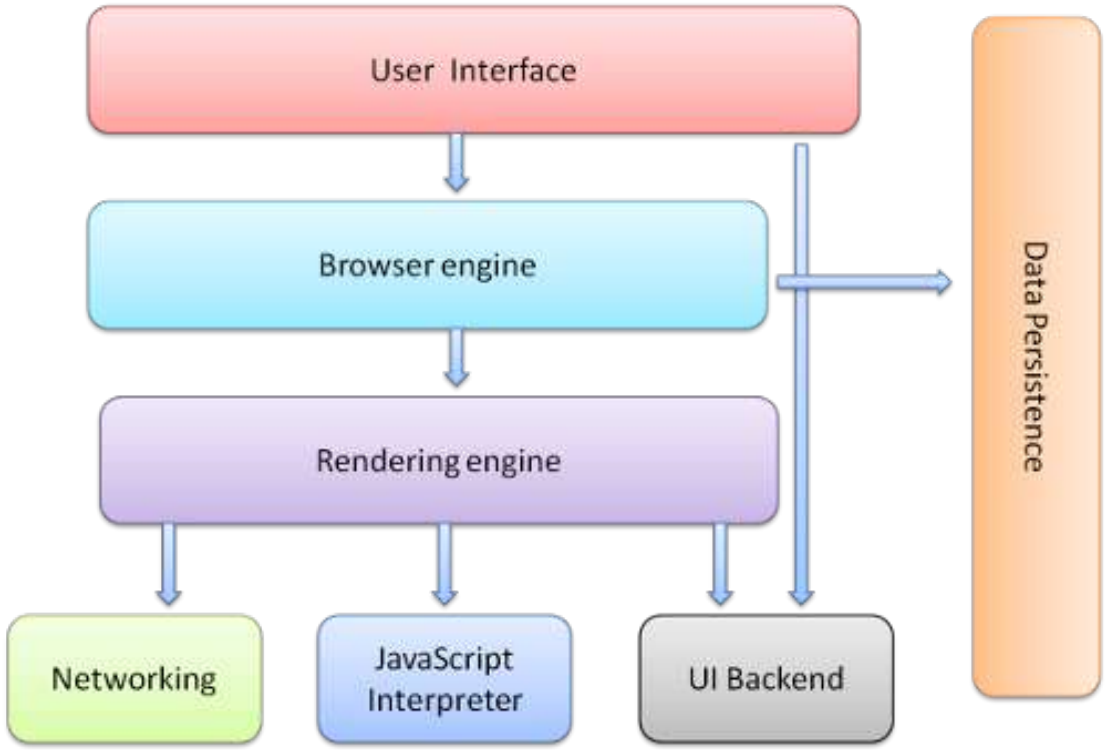


图 1 浏览器主要组件

注释一下 Chrome 很重要，和大多数浏览器不同，它保持渲染引擎的多个实例--每个选项卡一个，每个选项卡都是独立的进程。
我会为每个组件写一章。

组件间通信

Firefox 和 Chrome 都开发了一个专门的通信基本件。
这个会在专门的一章里讨论。

渲染引擎

渲染引擎的职责是……好吧，渲染，就是把请求的内容展现在浏览器屏幕上。
渲染引擎默认可以展现 html 和 XML 文档以及图片，它可以通过插件（浏览器扩展）展现其他格式的文件，有个例子就是用一个 PDF 浏览插件来展现 PDF。我们会用专门的一章来讨论插件和扩展，本章我们专注于主要应用-展现 css 格式过的 HTML 和图片。

渲染引擎

我们提到的浏览器-Firefox、Chrome 和 Safari 基于两个渲染引擎。Firefox 使用 Geoko--一个 Mozilla 自制的渲染引擎，Safari 和 Chrome 都使用 Webkit。
Webkit 是源于 Linux 平台的一个开源渲染引擎，Apple 公司修改后支持 Mac 和 Windows，更多的细节参见 http://webkit.org/ 。

主要流程

渲染引擎开始会从网络模块获取要请求的文档内容，一般会以 8k 大小的区块获取。
在获取内容之后，是渲染引擎的基本流程：

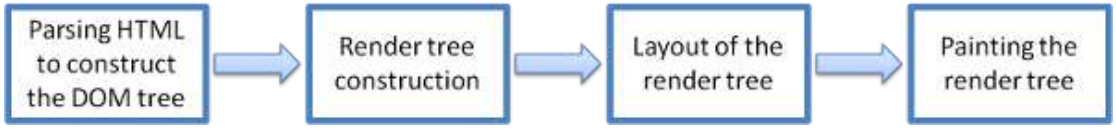


图 2 渲染引擎的基本流程

渲染引擎开始解析 HTML 文档，并且把标签转换成“内容树”上的 DOM 节点，然后会解析样式，包

括外部的 css 文件和 style 元素里的数据。这些样式信息将会和 HTML 中视觉性的属性组合在一起创建另一个树—渲染树。
渲染树由带有如颜色和大小等视觉属性的矩形区域构成，这些矩形区域按它们将要显示在屏幕上的顺序排列。
渲染树构造完成后，进入到布局阶段，会把每个节点精确地调整到它应该在屏幕上出现的位置上。下一步是绘制，渲染树将会被遍历，每个节点都会通过 UI 后端层来绘制。
理解这是一个渐进的过程相当重要。为了有更好的用户体验，渲染引擎将会尽可能早的把内容在屏幕上显示出来，不会等到所有的 HTML 都被解析完才开始建造和布局渲染树，当进程还在继续解析源源不断的来自于网络的内容的时候，一部分内容会被解析并且显示出来。

主要流程

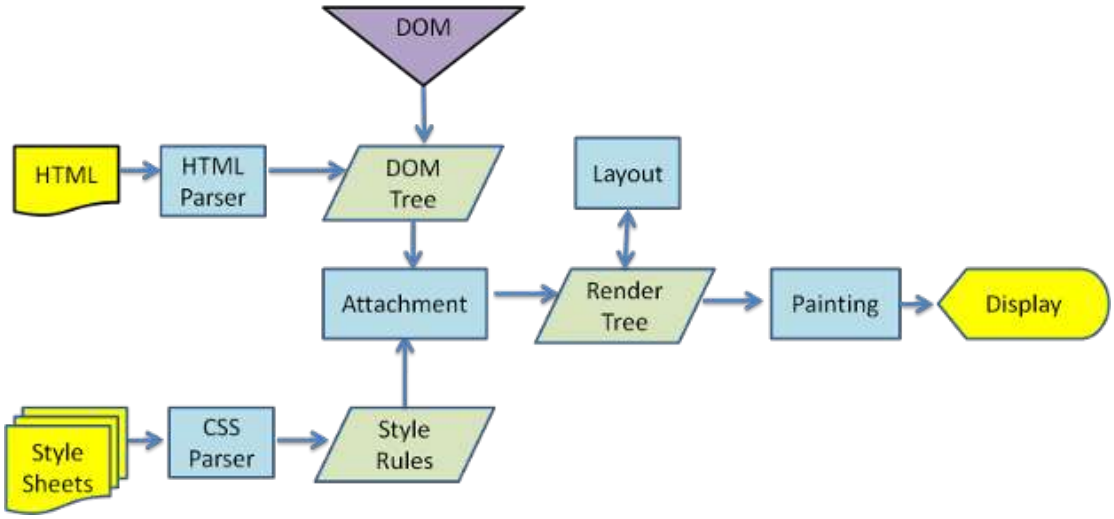


图 3 webkit 主要流程

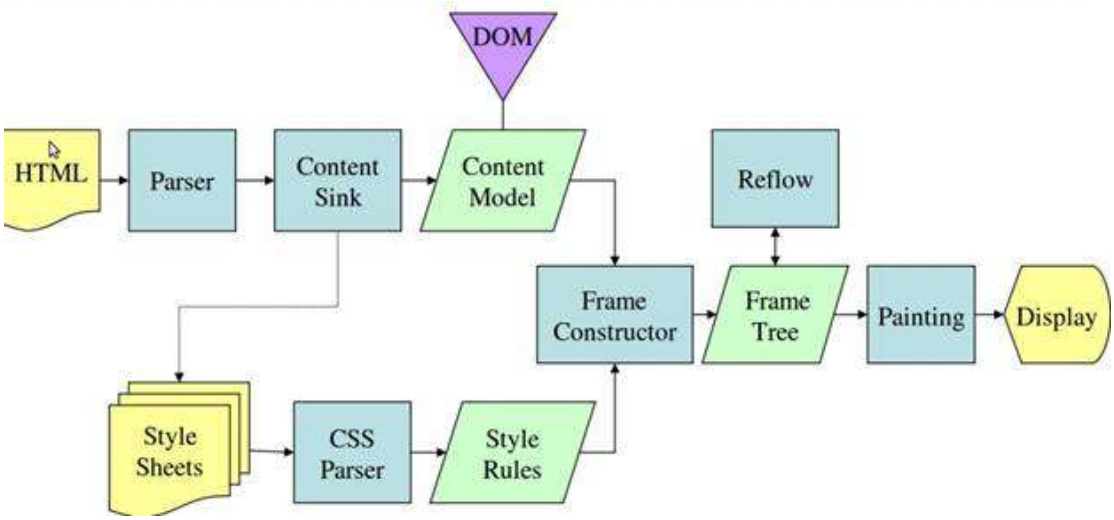


图 4 Mozilla's Gecko 渲染引擎主要流程

从图 3 和图 4 中可以看到，尽管 Webkit 和 Gecko 使用了少量不同的名称，流程基本上是一样的。Gecko 把这个由视觉上格式化了的元素构成的树叫做“Frame Tree”，每个元素都是一个帧。webkit 用了“Render Tree”这个名称，这个树由“渲染对象”构成。webkit 用了“layout”来表示元素的放置，而 Gecko 把它叫做“reflow”，“ Attachment”是 webkit 中用来表示把 DOM 节点和可见信息组合创建渲染树的术语。一个不大重要的非语义上的不同是 Gecko 在 HTML 和 DOM 树之间有额外的一层，叫做“content sink”，它是生成 DOM 元素的地方。我们将会讨论这个流程的各个部分。

解析-概述

因为渲染引擎里的解析是一个非常重要的过程，我们将会稍微深入一下。让我们先介绍一下解析。

解析文档的意思是把文档转成有意义的一些结构—代码能够理解并且使用的东西，解析后的结果通常是一个能表现文档结构的节点树，叫做解析树或者语法树。
例1— 解析表达式“2+3-1”会返回下面的树：

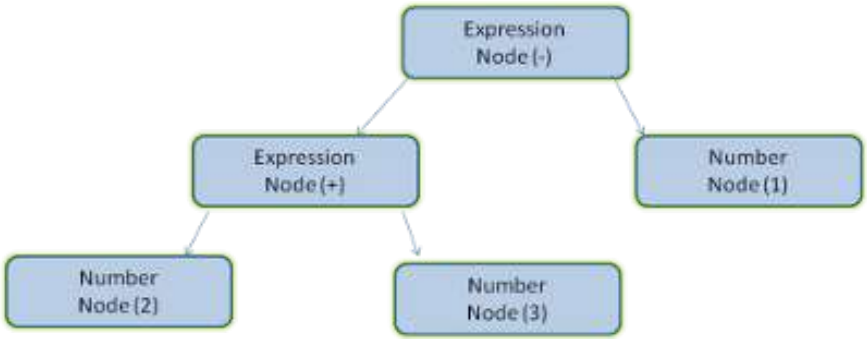


图 5 算术表达式的树节点

语法

解析是根据文档所遵循的语法规则—文档被写成的语言或者格式，每种格式都必须有由词典和语法规则组成的确定的语法，这被叫做上下文无关语法。人类语言不是这样的语言，也因此不能用常用的解析技术来解析。

解析器—词法组合

解析可以分成两个子过程—词法分析和语法分析。
词法分析是把输入信息分成多个子串的过程，这些子串是这个语言的词典—有效构造块的集合，在人类语言中包括在词典中出现的所有词汇。
语法分析是语言语法规则的应用。
解析器通常把这个工作分给两个部件—负责把输入信息分成有效子串的词法分析器（有时被叫做编译器），以及负责根据语法规则分析文档结构并构建解析树的解析器，词法分析器知道怎么去分解诸如空格和换行等无关紧要的字符。

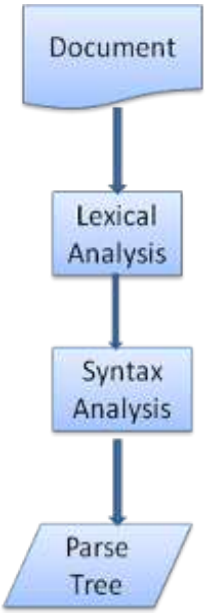


图 6 从原始文档到解析树

这个解析过程是迭代的。解析器通常会向词法分析器请求新的子串然后试图用某些语法规则来匹配这个子串，如果匹配了，会在解析树上加上这个子串相应的节点，然后请求新的子串；如果没有匹配，解析器将会在内部保存这个子串，然后继续请求新的子串直到内部保存的子串匹配了一个规则。如果没有规则可以匹配，解析器将会报出一个异常，这意味着文档是不合法的，包含着一个语法错误。

转换

很多时候解析树并不是最终的结果。在转换中经常使用解析—把文档转换成另一种格式。有个例子是编译，负责把源代码编译成机器代码的编译器先会把文档解析成解析树，然后把解析树转换成机器代码的文档。

图 7 编译流程

解析示例

在图 5 中我们把一个数学表达式构建成了一个解析树，我们定义一种简单的数学语言，来看看这个解析过称。
词典：我们的语言包含整数、加号和减号。
语法：
1. 这个语言的语法构造块包括表达式、项和操作符。
2. 我们的语言能包含很多表达式。
3. 表达式的定义是一项后跟着操作符，再跟着一项。
4. 操作符为加号或者减号
5. 项为整数或者表达式

我们来分析一下“2+3-1”。
第一个符合规则的子字符串是“2”，根据第 5 条规则它是一个项，第二个符合的是“2+3”，它符合第 2 条—一项后跟着一个操作符，再跟着一项，再下一个符合的就到了最后了，“2+3-1”是一个表达式，因为我们已经知道了“2+3”是一项，它后面是一个操作符后再跟着一项。“2++”不符合任何规则因此是不合法的。

词典和语法的正式定义

词典通常被表示成正则表达式的形式。
例如我们的语言定义为：
整数 :0|[1-9][0-9]*
加号 : +
减号: -
正如你看到的，整数是通过正则表达式定义的。
语法通常被定义成一种叫作 BNF 的格式，我们的语言定义为：

```
expression :=  term  operation  term
operation :=  PLUS | MINUS
term := INTEGER | expression
```

我们说过如果一种语言的语法是上下文无关语法的话，它就能被常用的解析器所解析。上下文无关语法直观的定义是能够被 BNF 表示的语法，正式的定义可以去看 http://en.wikipedia.org/wiki/Context-free_grammar。

解析器类型

有两种基本类型的解析器—自上而下的和自下而上的解析器，直观的解释就是自上而下的解析器观察语法的整体结构然后尝试匹配，自下而上的解析器则是从低级别的规则开始直到符合高级别的规则，把式子逐步的转换成语法规则。
让我们看一下这两种解析器是如何解析我们的示例的。
自下而上的解析器从更高级别的规则开始—它会把“2+3”识别为是一个表达式，然后把“2+3-1”识别为一个表达式（识别表达式的过程演变为匹配其他的规则，但是起点是最高级别的规则）。
自下而上的解析器则会浏览这个式子，当有规则匹配的时候就会用这个规则来代替被匹配的式子，然后继续直到式子结束，部分匹配了的表达式放置在解析栈里。

Stack	Input
	2 + 3 - 1
term	+ 3 - 1
term operation	3 - 1
expression	- 1
expression operation	1
expression	

这种自下而上的解析器被叫作移位规约解析器，因为式子被移到右边（想象一个起初在式子开头然后移到右边的指针）并且被逐步的归到语法规则去。

自动生成解析器

有不少工具能够帮你生成一个解析器，它们叫作解析器生成器。把你的语言的语法—语言的词典和语法规则交给解析器生成器，它就会生成一个可以工作的解析器。做一个解析器需要对解析有很深入的理解，而且手动的做一个优化了的解析器不会很容易，所以解析器生成器很有用处。
Webkit 使用两款非常出名的解析器生成器—生成词法分析器的 Flex 和生成语法分析器的 Bison（你可以合称为 Flex and Yacc），Flex 接收的是包含用正则表达式定义的子串的文件，Bison 接收 BNF 格式的语法规则。

HTML 解析器

HTML 解析器的工作是把 HTML 标记转换成解析树。

HTML 语法定义

HTML 的词典和语法规则定义在 W3C 制定的规范中，当前版本是 HTML4，HTML5 正在制定中。

并非上下文无关语法

我们在解析器概述中了解到，语法规则能够用 BNF 格式正式的定义。
但是常用解析器所有的规则都不适用于 HTML（我不是为了好玩提到这个—可以在解析 css 和 javascript 的时候使用），HTML 没法简单的通过上下文无关语法来定义。有个正式的定义 HTML 的格式—DTD（文档类型定义），但是它不是上下文无关语法。
这看起来有点奇怪—HTML 和 XML 相当接近，却有很多通用的 XML 解析器。HTML 有一个

XML 的变形—XHTML，那最大的不同是什么呢？
不同在于 HTML 的实现更加“宽容”，它允许你省略某些标签，有时是标签的开始或者结束。整体上它是一种“软”语法，相对于 XML 死板的严格的语法。
很显然这哥看起来很小的不一样产生了很大的不同。一方面这是 HTML 如此受欢迎的主要原因—它容忍你的错误，让 web 开发者更加容易，另一方面这让写格式化的语法变得困难。概括一下—因为 HTML 的语法不是上下文无关语法，所以不能被常用解析器解析，也不能被 XML 解析器解析。

HTML DTD

HTML 的定义是 DTD 格式的，这种格式被用来定义 SGML 系列的语言，包含所有允许元素的定义、它们的属性和层次结构。如我们之前了解的，DTD 不是上下文无关语法。

DTD 有几种变化形式，strict 模式完全的遵守规范，其他模式则支持早期浏览器用到的标记，这是为了向后和以前的内容兼容。当前 strict DTD 在这：
http://www.w3.org/TR/html4/strict.dtd

DOM

输出树—解析树是一个带有 DOM 元素和属性节点的树，DOM 是文档对象模型的简称，它是 HTML 文档的对象表示方式，也是 HTML 元素面向外部世界如 javascript 的接口。

树的根节点是“Document”对象。

DOM 树和标记几乎是一对一的关系。例子，标记：

```
<html>
        <body>
                <p>
                        Hello World
                </p>
                <div> <img src="example.png"/></div>
        </body>
</html>
```

将会被转换成下面的树：

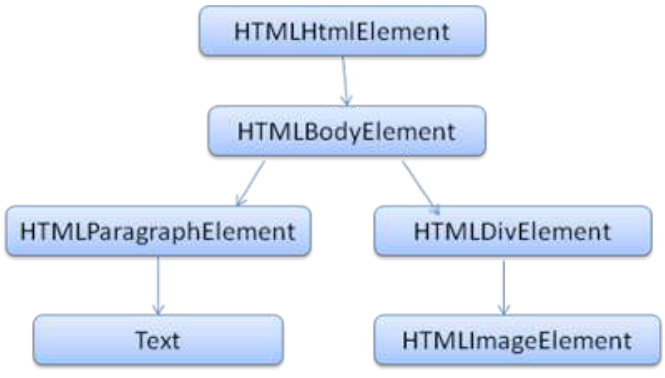


图 8　例子标记的 DOM 树

和 HTML 一样，DOM 也是被 W3C 指定的，参见 http://www.w3.org/DOM/DOMTR。这是一个操作文档的通用规范，用特定的模块来描绘 HTML 特定的元素。HTML 定义可以在这找到：http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/idl-definitions.html。

当我说树包含 DOM 节点的时候，意思是树是由实现 DOM 接口的元素构建的，Browsers use concrete implementations that have other attributes used by the browser internally。

解析算法

如我们上一节所了解的，HTML 不能用普通的自上向下或者自下向上的解析器解析。
原因如下：
1. 语言宽容的特性
2. 浏览器对人们熟知的非法 HTML 有容错性的事实
3. 解析过程是可中断的。通常资源在解析过程中是不变的，但是在 HTML 里，包含“document.write”的脚本可以增加额外的子串，因此解析过程实际上改变了初始内容。

因为不能使用常用的解析技术，浏览器做了专门的解析器来解析 HTML。
HTML5 规范中详细描述了解析算法，算法由两个过程组成—断词和树构造。断词是词法分析，把初始内容解析成子串，在 HTML 中子串是开始标签、结束标签、属性名和属性值，分词器识别子串，交给树构造，然后从下一个字符来识别下一个子串，以此类推直到内容的结束。

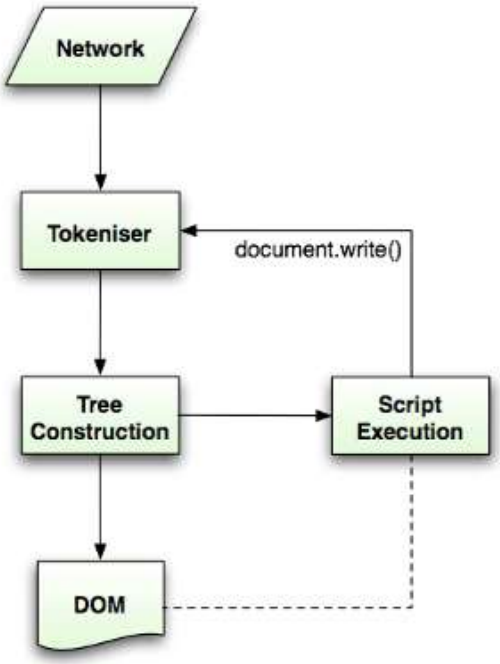


图 6 HTML 解析流程（来自 HTML5 规范）

分词算法

这个算法返回的是一个 HTML 子串，算法以状态机来表示。每个状态消耗输入内容的一或多个字符并且依据这些字符来更新下一个状态，指令是受当前分词状态和树构造状态影响的，这意味着根据当前状态，同样的字符会产生不同的结果交给下一个状态。这个算法太复杂了，不可能全部拿出来说，让我们看一个简单的例子来理解主要的东西。
例1— 把下面的 HTML 分词

```
<html>
        <body>
```

```
            Hello world
        </body>
</html>
```

初始状态是“数据状态”，当遇到“<”符号的时候，状态变成“标签开放状态”，“a-z”的字符会导致产生一个“标签开始标志位”，状态变成“标签名状态”，这个状态会延续到遇到一个“>”符号，每个字符都会被装到这个新的标记名里，在我们的例子里创建的标记是一个 HTML 标记。

当到达了“>”符号的时候，当前标志位被释放，状态变回“数据状态”，“<body>”标签通过同样的步骤处理。现在“html”和“body”标签都被释放了，我们又回到了“数据状态”，“Hello world”的“H”会导致一个字符标记的生成和释放，这会延续到遇到“</body>”的“<”,我们将会为“Hello world”的每个字符都释放一个字符标记。

现在我们回到了“标签开放状态”，下一个字符“/”会导致一个“标签闭合标记”的产生，还有状态会变成“标签名状态”，我们又一次延续这个状态直到遇到“>”符号，然后新的标签标记被释放，重新回到“数据状态”，“</html>”会像上一个情况一样处理。

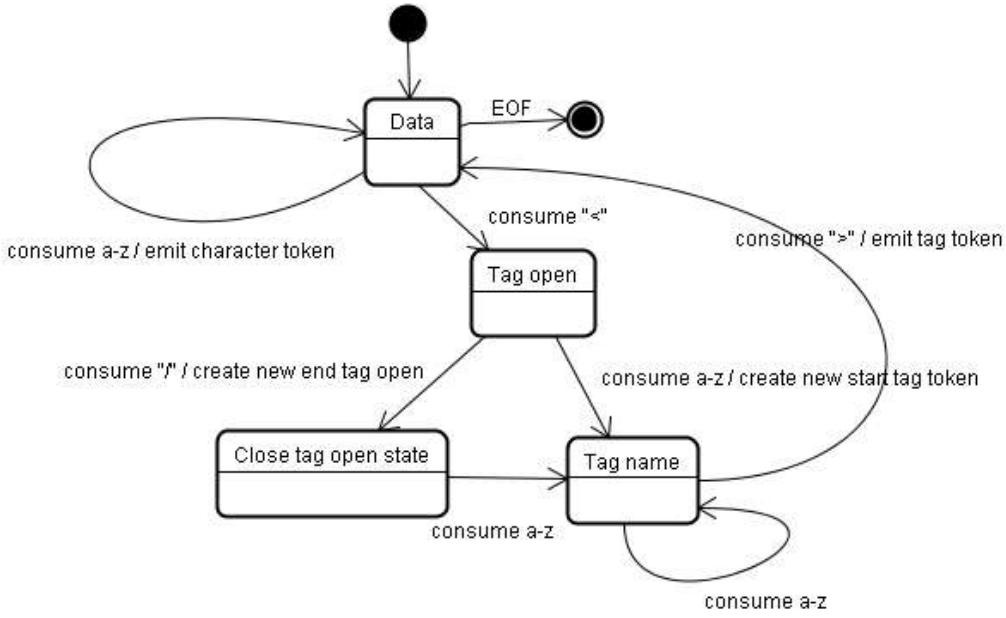


图 9 分词例子

树构造算法

当解析器被创建的时候文档对象也被创建了，在树构造阶段，以 Document 为根节点的 DOM 树将被修改，元素被添加上去，分词器释放的每个节点都会被树构造器处理。对于每一个标记，规范都定义了是哪个 DOM 元素和它对应。这些标记除了添加到 DOM 树上，还会加入到开放元素的栈上，这个栈是用来修正嵌套不匹配以及没闭合的标签的。这个算法也以状态机来表示，这些状态叫做“接入模式”。

让我们看一下这个例子的树构建过程。

```
<html>
        <body>
            Hello world
        </body>
</html>
```

树构造阶段接收的是来自于分词阶段的一系列标记。第一个模式是“初始模式”，接收到 html 标记的时候变成“before html”模式并且在这个模式下对标记进行再加工，这会导致 HTMLHtmlElement 元素的产生，并且插入到根节点 Document 对象。

状态变成“before head”模式，我们接收了“body”标记，尽管没有“head”标记，仍然会创建一个 HTMLHeadElement 元素并且添加到这个树上。

我们转换到了“in head”模式然后又到了“after head”模式，body 标记被再加工，HTMLBodyElement 被创建并插入到树上，同时状态变成“in body”模式。
现在“Hello world”的字符标记被接收，第一个字符将会导致一个“Text”节点的创建和插入，其他的字符会被插入到这个节点。
body 结束标记的接收导致转换到“after body”模式，现在我们接收到 html 结束标签，这会转到“after after body”模式，接收到文件的末尾时会结束整个解析。

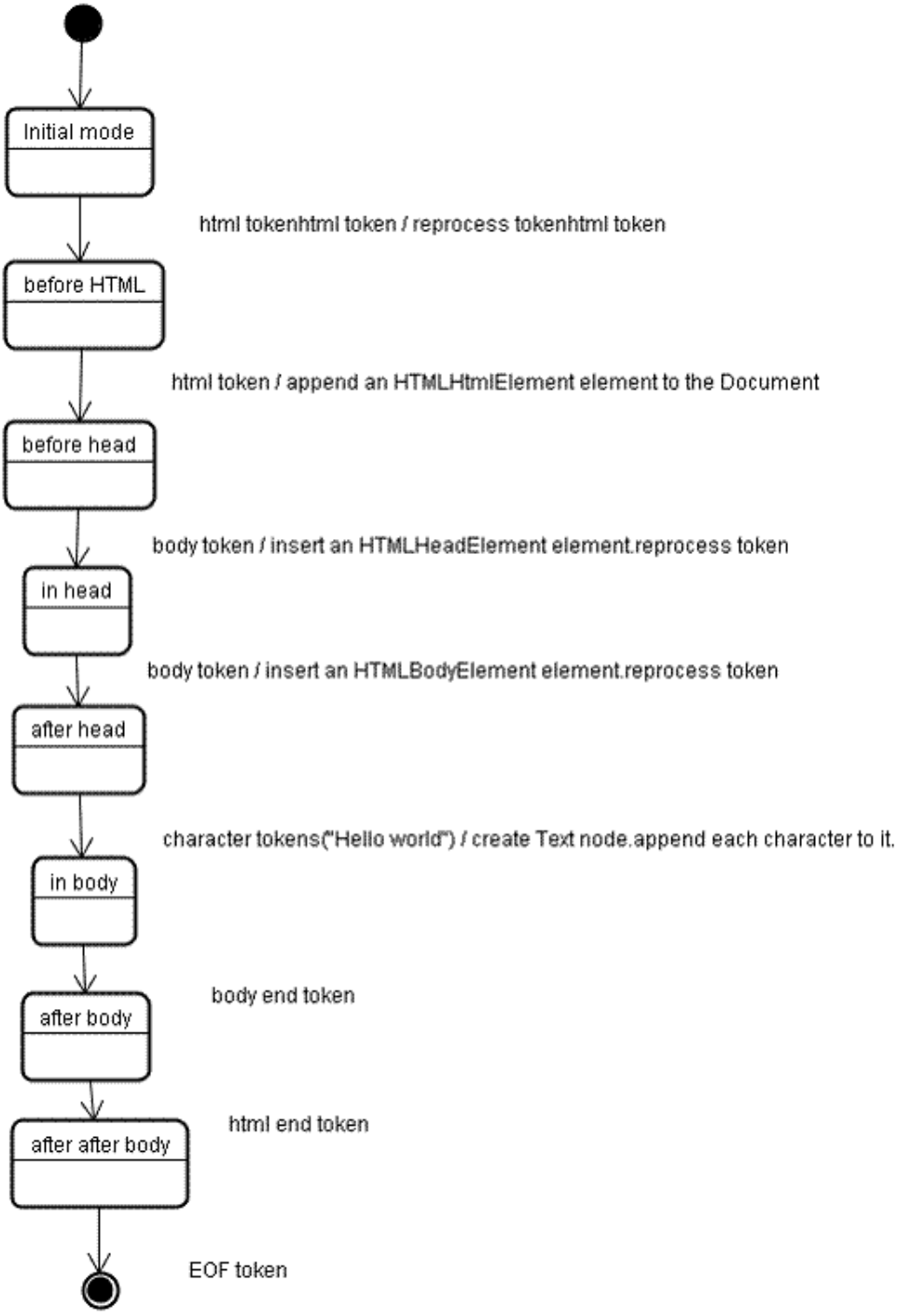


图 10　例子中 html 的树构造

解析完成后的动作

在这个阶段，浏览器将会把文档标记成可交互的并且开始解析“deffered”模式下的脚本—那些在文档解析完成后才执行的脚本，然后文档状态被设置为“complete”，同时一个“load 事件被触发。

浏览器的容错

在 HTML 页面里，你永远不会得到一个“无效语法”的错误，浏览器会修正无效的内容。
以这段 HTML 为例：

```
<html>
  <mytag>
  </mytag>
  <div>
  <p>
  </div>
        Really lousy HTML
  </p>
</html>
```

我肯定违反了很多个规则（“mytag”不是一个标准标签，错误嵌套了 div 和 p 等等），但是浏览器没有提示，仍然正常的显示，有很多为了修正 HTML 开发者的错误的解析器代码。
各个浏览器的错误处理都是非常接近的，奇怪的是这并不是规范中的一部分，就像书签和后退/前进按钮一样，这是浏览器多年发展的结果。有很多已知的很多网站都存在的无效 HTML 结构，浏览器都试图用和其他浏览器一致的方式修正它们。
HTML5 规范定义了这些需求中的一些，webkit 在 HTML 解析器讲解的开始非常好的总结了这些：
解析器把分好词的内容解析成文档并构建文档树，如果文档树结构良好，解析就很容易了，但是不幸的是我们需要处理很多结构不良的 HTML 文档，解析器也不得不容忍这些错误。
我们至少要处理下面几种情况：

1. 将要添加的元素被明确禁止出现在某些标签里面。这种情况，我们应该闭合所有标签直到禁止这个元素的标签，然后在它后面添加。
2. 不允许直接添加这个元素。可能是写代码的人漏掉了一些标签（或者标签是可选的），下面的标签有这种情况：HTML HEAD BODY TBODY TR TD LI（还有没记起来的吗）。
3. 在内联元素里添加块状元素。闭合所有的内联元素直到下个块状元素。
4. 如果还不行，在允许添加这个元素之前闭合前面元素，要不就忽略这个标签。

让我们看一下 webkit 容错的例子。
**</br> instead of
**
有些网站用</br>代替
，为了和 IE、Firefox 一致，webkit 把这个和
一样对待。
代码：

```
if (t->isCloseTag(brTag) && m_document->inCompatMode()) {
     reportError(MalformedBRError);
     t->beginTag = true;
}
```

注释：错误处理是在内部的--不会呈现给用户。
散乱的表
散乱的表是指一个表格在另一个表格内但又不在单元格内。

```
<table>
        <table>
                <tr><td>inner table</td></tr>
        </table>
        <tr><td>outer table</td></tr>
</table>
```

webkit 会把这种层次关系转成两个同级的表格。

```
<table>
        <tr><td>outer table</td></tr>
</table>
<table>
        <tr><td>inner table</td></tr>
 </table>
```

代码：

```
if (m_inStrayTableContent && localName == tableTag)
```

```
        popBlock(tableTag);
```
Webkit 中用栈来处理当前元素内容—它会将内部的表格从外部表格里冒泡出来，两个表格成为同级关系。

嵌套的表单元素

一旦用户把一个表单放到了另一个表单里面，后来的表单会被忽略。

代码：

```
if (!m_currentFormElement) {
      m_currentFormElement = new HTMLFormElement(formTag,   m_document);
}
```

太深的标签层次

注释里说的很明白。

www.liceo.edu.mx 是一个多达 1500 个标签嵌套的例子，都是一大堆 b 标签，我们只允许同样的标签最多嵌套 20 次，其余的将会一起忽略。

```
bool HTMLParser::allowNestedRedundantTag(const AtomicString& tagName)
{

unsigned i = 0;
for (HTMLStackElem* curr = m_blockStack;
       i < cMaxRedundantTagDepth && curr && curr->tagName == tagName;
    curr = curr->next, i++) { }
return i != cMaxRedundantTagDepth;
}
```

错位的 html 和 body 闭合标签

注释仍然说的很明白。

支持中断的 html。

我们不会闭合 body 标签，因为一些比较蠢的 web 页面在文档真正结束之前就闭合了它。

我们依赖 end（）调用来闭合东西。

```
if (t->tagName == htmlTag || t->tagName == bodyTag )
      return;
```

web 开发者要小心了—除非是想展现一个 Webkit 容错代码的例子--写结构良好的 HTML。

CSS 解析

还记得解析概述里解析的概念吗？不像 HTML，CSS 是上下文无关语法，能用概述里说的解析器来解析。事实上，css 规范里定义了 CSS 的句法和语法规则 (http://www.w3.org/TR/CSS2/grammar.html)。

我们看几个例子。

词法规则（词典）中标记以正则表达式的形式定义。

```
comment         \/\*[^*]*\*+([^/*][^*]*\*+)*\/
num             [0-9]+|[0-9]*".″[0-9]+
nonascii        [\200-\377]
nmstart         [_a-z]|{nonascii}|{escape}
nmchar          [_a-z0-9-]|{nonascii}|{escape}
name            {nmchar}+
ident           {nmstart}{nmchar}*
```

“ident”是标识符的简称，如 class 的名字，“name”是元素 id（通过#来引用）。

语法规则以 BNF 描述。

```
ruleset
  : selector [ ',' S* selector ]*
    '{' S* declaration [ ';' S* declaration ]* '}' S*
  ;
selector
  : simple_selector [ combinator selector | S+ [ combinator selector ] ]
  ;
```

```
simple_selector
  : element_name [ HASH | class | attrib | pseudo ]*
  | [ HASH | class | attrib | pseudo ]+
  ;
class
  : '.' IDENT
  ;
element_name
  : IDENT | '*'
  ;
attrib
  : '[' S* IDENT S* [ [ '=' | INCLUDES | DASHMATCH ] S*
    [ IDENT | STRING ] S* ] ']'
  ;
pseudo
  : ':' [ IDENT | FUNCTION S* [IDENT S*] ')' ]
  ;
```

说明：规则集是这种结构：

```
div.error , a.error {
        color:red;
        font-weight:bold;
}
```

div.error 和 a.error 是选择器，大括号内的部分包含规则集应用的规则，这种结构是在规范中被正式定义了的。

规则集：

```
: selector [ ',' S* selector ]*
    '{' S* declaration [ ';' S* declaration ]* '}' S*
  ;
```

这意味着规则集是一个选择器或者用逗号和空格(S 代表空格)分隔开的任意数目的一组选择器，规则集包括大括号和它内部的一个或者用分号隔开的多个声明，“declaration”和“selector”会在后面的 BNF 定义中定义。

webkit CSS 解析器

webkit 使用 Flex and Bison 解析器生成器来根据 CSS 语法文件自动地生成解析器，如你在解析器概述里了解的，Bison 生成一个自下向上的移位规约解析器。Firefox 使用的是手动编写的自上向下的解析器，这两种情况 css 文件都会被解析成一个 StyleSheet 对象，每个对象包含 css 规则，CSSrule 对象包含选择器和声明对象以及符合 CSS 规则的其他对象。

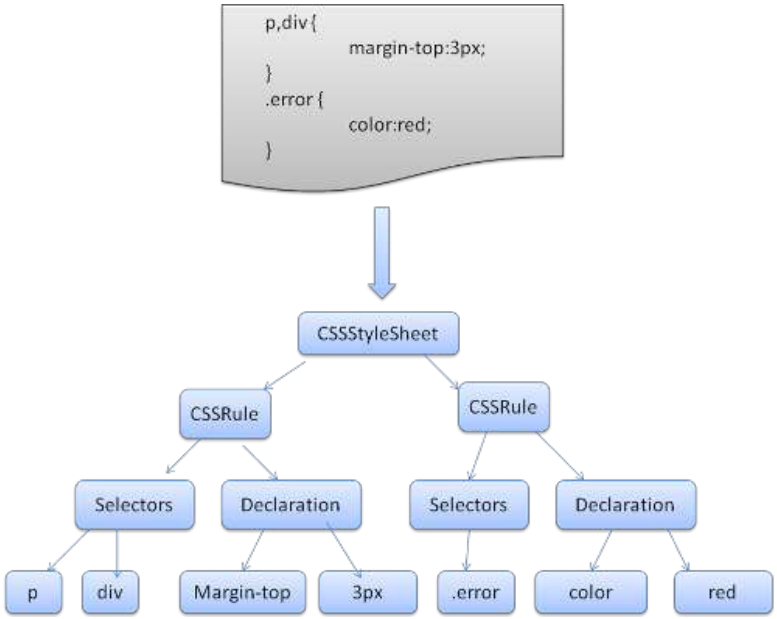


图 7 解析 CSS

解析脚本

这个会在关于 javascript 的章节里讨论。（shit）

处理脚本和样式的顺序

脚本

web 模型是同步的，开发者希望当解析器遇到<script>标签的时候脚本就被解析并且执行，脚本执行完成前暂停文档的解析，如果脚本是外部的，必须首先从网络上获取这个资源—这也是同步的，获取到这个资源之前会暂停文档的解析，这是用了多年的模型，当然也在 HTML4 和 5 中被定义了。开发者可以把脚本标记为“defer”，这样就不会暂停文档的解析，而是等文档解析完才执行。HTML5 增加了一个选择，可以把脚本标记为异步的，这样就会通过一个不同的线程来解析和执行它。

取巧性解析

Webkit 和 Firefox 都做了这个优化。当执行脚本的时候，另一个线程会解析其余的文档，寻找哪些资源需要从网络上加载并且加载它们，这种方式资源会被平行地载入，整体速度会好一些，要注意—取巧性解析器并不改变 DOM 树而是把 DOM 留给主解析器，它只解析引用的外部资源，比如外部脚本、样式表和图片。

样式表

样式表是不同的模型，理论上来说因为样式表并不改变 DOM 树，所以没有理由为了等待它们而去停止解析文档，然而却有一个问题，在文档解析阶段的时候，脚本会请求样式信息，如果样式还没有加载或者解析，脚本会得到错误的信息，很明显这会导致很多问题。这种情况看起来边缘但实际上很普遍，当样式表在被加载和解析的时候 Firefox 会阻塞所有的脚本，而只有当脚本试图访问某些会被还未加载的样式影响的属性的时候，Webkit 才会阻塞它们。

渲染树构造

当 DOM 树被创建的同时，浏览器创建了另一棵树，渲染树。这个树由按显示顺序排列的可见元素组成，是文档的视觉性的表现，这个树的作用是确保按正确顺序绘制内容。
Firefox 把渲染树中的元素叫做“帧”，Webkit 则使用名词渲染程序或者渲染对象。

渲染对象知道如何布局和绘制它本身以及它的子节点。
Webkit 的 RenderObject 类，渲染对象的基类，有下面的定义：

```
class RenderObject{
        virtual void layout();
        virtual void paint(PaintInfo);
        virtual void rect repaintRect();
        Node* node;  //the DOM node
        RenderStyle* style;  // the computed style
        RenderLayer* containgLayer; //the containing z-index layer
}
```

如 CSS2 规范中描述的，每个渲染对象表示一个通常对应于这个节点的 CSS 盒子的矩形区域，它包含像宽、高和位置这样的几何信息。
盒子的类型受这个节点所对应的“display”属性影响（参见 style computation 章节），下面是 webkit 中，根据 display 属性应该为这个节点创建哪种渲染对象的代码：

```
RenderObject* RenderObject::createObject(Node* node, RenderStyle* style)
{
    Document* doc = node->document();
    RenderArena* arena = doc->renderArena();
    ...
    RenderObject* o = 0;

    switch (style->display()) {
        case NONE:
            break;
        case INLINE:
            o = new (arena) RenderInline(node);
            break;
        case BLOCK:
            o = new (arena) RenderBlock(node);
            break;
        case INLINE_BLOCK:
            o = new (arena) RenderBlock(node);
            break;
        case LIST_ITEM:
            o = new (arena) RenderListItem(node);
            break;
       ...
    }

    return o;
}
```

元素的类型也被考虑到了，例如表单控制和表格有专门的渲染对象。
在 Webkit 中如果一个元素想创建一个特殊的渲染对象，就会重写“createRenderer”方法，这个渲染对象指向包含非几何信息的 style 对象。

渲染树和 DOM 树的关系

渲染对象和 DOM 元素是相一致的，但是并不是一对一的关系，非可见元素不会被插入到渲染树上，有个例子是“head”元素，还有 display 属性为 none 的元素不会出现在树中（visibility 为 hidden 的元素会出现）。
有的 DOM 元素对应好几个可见元素，它们通常都有一个独立的矩形区域描述不了的复杂结构，

例如“select”元素有三个渲染对象—一个负责显示区域，一个负责下拉列表框，还有一个负责按钮。还有当行宽不够文本被切成多行的时候，新行会作为额外的解析器被添加。
另一个有多个渲染对象的是不合法的 HTML。根据 CSS 规范，一个内联元素要么只包含块状元素，要么只包含内联元素，万一有混合型的内容，会创建匿名的块状渲染对象来包裹内联元素。
有的渲染对象对应一个 DOM 节点但是在树中不是同样的位置。浮动和固定定位的元素脱离文档流，被放置在树上不同的位置，然后映射到实际的渲染对象上，在它们应该在的位置上有个占位帧。

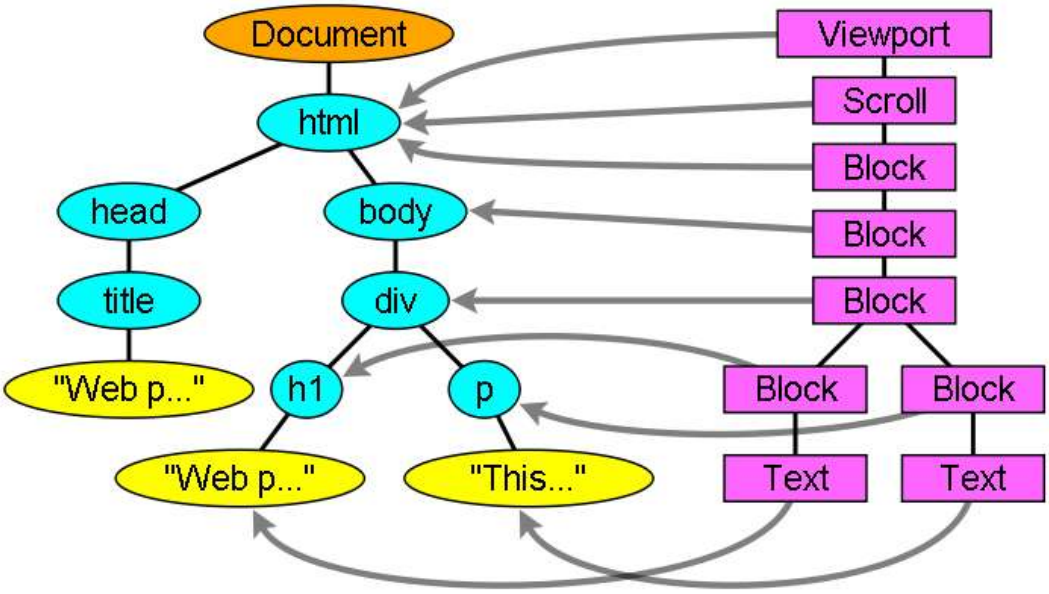


图 11 渲染树和对应的 DOM 树，“viewport”是初始包含块，在 Webkit 中是“RenderView”对象

构建树的流程

在 Firefox 中，这种方式是被当作 DOM 更新的监听器提出的，把帧的创建委派给“FrameConstructor”构造器，这个构造器解析样式(参见 style computation)并且创建帧。
在 Webkit 中，解析样式和创建解析对象的过程叫做“attachment”，每个 DOM 节点有一个“attach”方法，绑定是同步的，节点插入到 DOM 树中调用新节点的“attach”方法。
处理 html 和 body 标签引起渲染树根节点的创建，根节点渲染对象对应于 css 规范中叫做包含块的东西—包含所有其他块的最顶层的块，它的大小就是视角区域—浏览器窗口显示区域的大小，Firefox 把它叫作“ViewPortFrame”而 Webkit 则叫作 RenderView，这是 documnet 指向的渲染对象，树的其它部分随着 DOM 节点的插入而创建。
看一下 CSS2 的这个话题--http://www.w3.org/TR/CSS21/intro.html#processing-model。

样式计算

构建渲染树需要计算每个渲染对象的视觉性的属性，这是通过计算各个元素的样式属性完成的。
样式来自于不同来源的样式表，内联样式或者 HTML 中视觉性的属性（就像“bgground”属性），后者会被转换以符合 CSS 的样式属性。
样式表来源于浏览器默认的样式表、页面开发者提供的样式表以及用户的样式表—这些是浏览器用户提供的样式表（浏览器允许你定义自己喜欢的样式，例如在火狐中，这些样式表放置在“Firefox Profile”文件夹中）。
样式计算带来一些困难：

1. 样式数据是个很大的结构，保持这些大量的样式属性可能会导致内存问题。
2. 如果没有优化，为每个元素查找匹配的样式会导致性能问题。为每个元素查找匹配规则去遍历整个规则列表是项很大的工作，选择器可能会有很复杂的结构，匹配过程从一个看起来有戏但是却被证明是无效的路径开始，然后不得不尝试另一个路径。
   例如，这个多重的选择器：

```
div div div div{
...
```

}

意味着这个规则适用于是 3 个 div 的子孙的“ ”，假设你想检查这个规则是否适用于一个给定的 div，你选择了某个路径，向上遍历节点树却发现只有两个 div，因此规则不适用，你就需要尝试树上的另一个路径。

3. 应用规则牵涉到很复杂的定义规则继承的层叠规则。

共享样式数据

webkit 中节点和样式对象（RenderStyle）相关联，在一些条件下这些对象能被多个节点共享，节点是兄弟姐妹以及：

1. 这些元素必须在同样的鼠标状态下（例如，不能一个是:hover 而另一个不是）
2. 节点都不能有 id
3. 标签名称必须匹配
4. class 属性应该匹配
5. 映射的属性组必须是完全相同的
6. 链接状态必须匹配
7. 焦点状态必须匹配
8. 节点都不能被属性选择器影响，
9. 元素上必须没有内联样式
10. 必须完全没有使用兄弟选择器，当碰到兄弟选择器的时候 WebCore 会简单的抛出一个全局的切换，并且当它们被展现的时候禁止整个文档的样式共享，这包括+选择器，还有与:first-child 和:last-child 类似的选择器。

为了更容易地计算样式，Firefox 有两个额外的树—规则树和样式上下文树，Webkit 同样也有样式对象但是并没存储在像样式上下文树这样的树上，只是 DOM 节点指向它对应的样式。

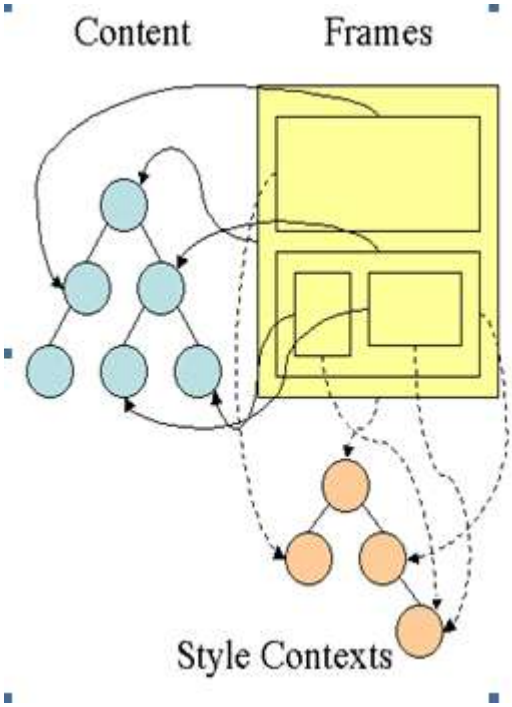
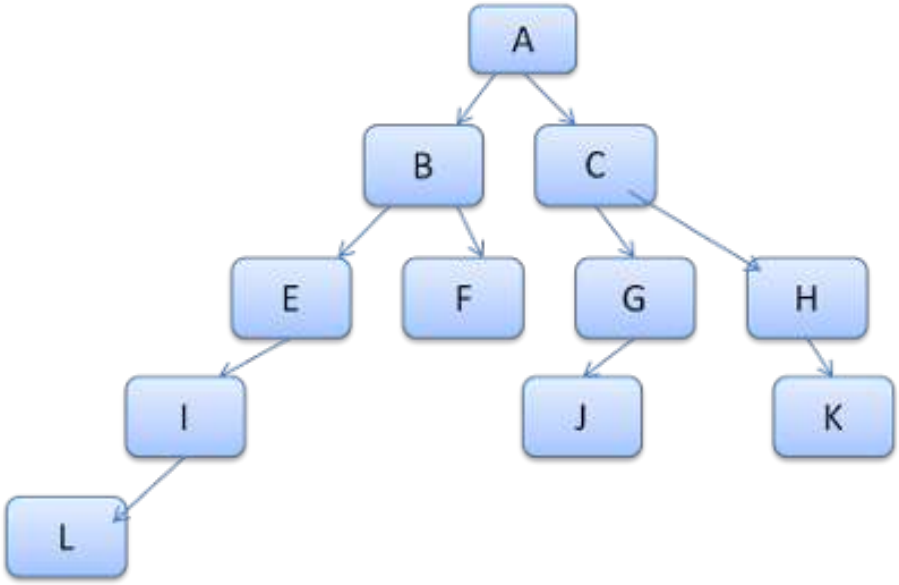


图 13 Firefox 样式上下文树

样式上下文包含最终值，通过按正确顺序调用所有匹配规则并把逻辑值转换成具体值，这些值被计算出来，例如—如果逻辑值是屏幕的百分比，就会计算并被转换成绝对单位。规则树这个想法确实非常好，能够避免在不同节点共享这些值的时候再次计算它们，这同样节省空间。

所有匹配的规则都被存储在一个树上，一个路径靠下的节点有更高的优先级，这个树包含所有被找到的规则匹配的路径，存储这些规则是惰性的，开始的时候树并不是每一个节点都被计算，但是一旦节点样式需要计算，计算过的路径就会被添加到树上。

这个想法把树路径看作是词典里的词汇，看下已经计算过了的规则树：



假设我们需要为内容树上的一个元素匹配规则并且找到符合的规则（按正确顺序）是 B-E-I，我们在这个树上已经有了这个路径，因为我们已经有了计算路径 A - B - E - I – L，我们要做的很少。
让我们看一下这个树是如何保存的。

分割成结构体

样式上下文会被分成很多份结构体，那些结构体包含某一类的样式信息，比如 border 或者 color，一个结构体中的所有属性要么是可继承的要么是非可继承的。如果元素本身没有定义，可继承的属性继承它的父节点，非可继承的属性（叫作“reset”属性）使用默认值。
这个树帮助我们缓存树上全部的结构体（包含计算过的最终值），想法是如果下面的节点没有给结构体提供一个定义，就可以使用上面节点上缓存的结构体。

使用规则树计算样式上下文

当为某个元素计算样式上下文的时候，我们首先计算出规则树里的一条路径或者使用现有的路径，然后应用路径中的规则把结构体填充到新的样式上下文中去。我们从路径的最下面的节点—最高优先级的节点（通常是最特殊的那个选择器）开始，向上遍历整个树直到指向这个元素—这是最好的优化—整个结构体都被共享，这节省最终值的计算和内存。
如果没能找到对这个结构体的任何定义，还要防止这个结构体是“可继承的”类型—我们指向上下文树上父节点的结构，这种情况下我们仍然能够共享结构体。如果它是 reset 类型的结构体就使用默认值。
如果这个最特殊的节点要添加值，就需要一些额外的运算来把它转换成实际值，然后在树的节点中缓存这个结果，这样就能被它的子节点使用。
如果一个元素有指向同一个树节点的兄弟元素，整个样式上下文都会被它们共享。
让我们看个例子，假设我们有这样的 HTML：

```
<html>
	<body>
		<div class="err" id="div1">
			<p>
			   this is a <span class="big"> big error </span>
			   this is also a
			   <span class="big"> very  big  error</span> error
			</p>
		</div>
		<div class="err" id="div2">another error</div>
	</body>
</html>
```

还用下面的规则：

```
1.      div {margin:5px;color:black}
2.      .err {color:red}
3.      .big {margin-top:3px}
4.      div span {margin-bottom:4px}
5.      #div1 {color:blue}
6.      #div2 {color:green}
```

为了简化事情，假定我们只需要填充两个结构--颜色和边距。颜色的结构体只包含一个成员--color，边距包含四个边。

结果规则树看起来像这样（元素用元素名来标记--规则的#指向的）。

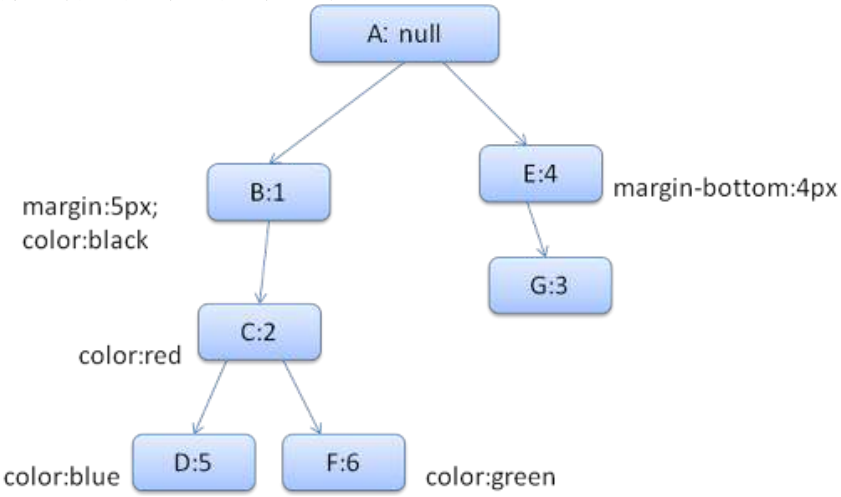


图 12 规则树

上下文树看起来像这样：

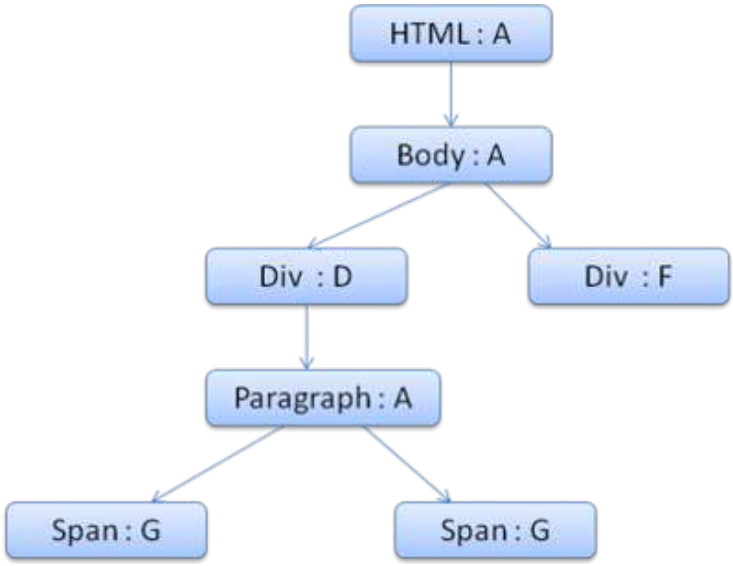


图 13 上下文树

假设我们解析这段 HTML 到了第二个 标签，我们需要为这个节点创建一个样式上下文并且填充上它的样式结构。

我们匹配规则，发现这个 符合的规则是 1、2 和 6，这意思是在树上有我们的元素能够使用的已经存在的路径，我们只需要在它上面为规则 6（在规则树上是节点 F）添加另一个节点。

我们将会创建一个样式上下文并且把它放在上下文树上，在规则树上这个样式上下文会指向节点 F。

现在我们需要填样式结构体。我们从填充边距结构体开始，因为最后一个规则节点（F）不加在边距结构体上，我们顺着树向上，直到找到一个前一个节点插入时缓存的计算过的结构体并且使用它，我们在 B 节点上找到了，它是指定了 margin 规则的最高的节点。

在 color 结构体上我们有一个定义，因此不能用缓存的结构体。因为 color 有一个属性，我们不用沿树向上去填其他的属性，我们会计算最终值（把字符串转换成 RGB 等）并且在这个节点上缓存计算过的结构体。

在第二个元素上的工作更加容易，我们匹配规则得到它指向规则 G 的结论，就像前一个 span，因为有指向同一个节点的兄弟节点，我们能共享整个样式上下文，只要指向前一个 span 的上下文就行了。
对于那些包含从父节点继承的规则的结构体，缓存是在上下文树上完成的（color 属性确实是可继承的，但是 Firefox 中把它当作是 reset 类型的，在规则树上缓存），
比如我们在段落上增加字体的规则：

```
p {font-family:Verdana;font size:10px;font-weight:bold}
```

作为上下文树上这个段落的子节点，div 元素可能和它的父节点共享同一个字体结构体，如果没有字体样式，这个结构体就会指定给这个 div。
在没有规则树的 Webkit 中，会遍历 4 次匹配的规则，首先是高优先级非重要的属性（那些应该被首先应用的属性，因为其他的依赖于它们—比如 display）被应用，然后是高优先级重要的，然后是普通优先级非重要的，再然后是普通优先级重要的规则，这样多次出现的属性就能根据正确的顺序被解析。The last wins.
所以要总结的是—共享样式对象（它们中结构的全部或者一部分）解决了问题 1 和 3，Firefox 的规则树同样对按正确顺序应用属性有所帮助。

处理规则使匹配更容易

样式规则有几个来源：
CSS 规则，在外部样式表或者 style 元素里的。

```
p {color:blue}
```

内联 style 属性，如：

```
<p style="color:blue" />
```

HTML 视觉性属性（映射到对应的规则）

```
<p bgcolor="blue" />
```

后两个很容易匹配到元素，因为它拥有 style 属性，还有 HTML 属性能用元素作为主键来映射。
如之前问题 2 里提到的，CSS 规则匹配能更有技巧性，为了解决这个问题，会处理这些规则，让存取变得更加容易。
样式表解析后，根据选择器，规则被分别添加到几种哈希图上的一个上。这些哈希图有 id 方式的，有 class name 方式的，有 tag name 方式的，还有一个是为了不符合上面这几类的一般性的图。如果选择器是 id，规则会添加到 id 图上，如果是 class，就被添加到 class 图上以此类推。
这种处理使规则匹配更加容易，没有必要去查看每一个声明—我们能从这些图中提取出元素对应的规则，这种优化能够排除规则中 95%以上的部分，这样在匹配过程中就不用去考虑它们（4.1）。
让我们举个例子，来看一下下面的规则：

```
p.error {color:red}
#messageDiv {height:50px}
div {margin:5px}
```

第一个规则会插入到 class 映射图上，第二个插入到 id 映射图上，第三个插入到标签映射图上。
有这样的 HTML 结构：

```
<p class="error">an error occurred </p>
<div id=" messageDiv">this is a message</div>
```

我们首先尝试 p 元素适用的规则，class 映射图上包含一个“error”的关键字，在这个关键字下面“p.error”这个规则被找到了，div 元素在 id 映射图（关键字是 id）和标签映射图上有对应的规则，剩下的仅有的工作就是找出通过关键字提取出的规则哪些是真正符合的。
例如如果 div 的规则是：

```
table div {margin:5px}
```

它仍然会从标签映射图中提取出来，因为关键字是最右边的选择器，但是它并不匹配我们的 div 元素，这个元素并没有 table 祖先。
Webkit 和 Firefox 都会做这个处理。

以正确的层叠顺序应用规则

style 对象有对应于每一个视觉性属性（所有 css 属性但是更广泛）的属性，如果这个属性没有被任何匹配的规则定义—一些属性就会从父元素的 style 对象继承，其它的属性使用默认值。
问题是当有多与一个定义的时候—让层叠顺序来解决这个问题。

样式表层叠顺序

一个 style 属性的声明能出现在多个样式表中或者在一个样式中出现多次，这意味着应用规则的顺序是非常重要的，这被称为“层叠”顺序。根据 CSS2 规范，层叠顺序是（从低到高）:

1. 浏览器声明
2. 用户的普通声明
3. 开发者的普通声明
4. 开发者的重要声明
5. 用户的重要声明

浏览器声明是最不重要的，只有被标记为重要的时候用户的声明才会重写开发者的声明，有同样优先级的声明将会先根据特性然后是被指定的顺序分类，HTML 视觉性属性被转换成对应的 CSS 声明，当做低优先级的开发者声明处理。

特性

选择器特性通过如下的 CSS2 规范定义：

- 如果声明来自 style 属性而不是带选择器的规则，计 1，其它的计 0（=a）
- 计选择器中 ID 属性的数量（=b）
- 计选择器中其它属性和伪类的数量（=c）
- 计选择器中元素名和伪类的数量（=d）

把这四个数字 a-b-c-d 连成一串得出特性。
你需要使用的进制是这四个数中最大的决定的。
例如，如果 a=14，你可以使用十六进制，在不大可能出现的 a=17 这种情况里，你需要一个 17 位进制，后一种情况可能在有像这样的选择器时发生 html body div div p ...（17 个标签，不大可能……）。
一些例子：

```
*             {}  /* a=0 b=0 c=0 d=0 -> specificity = 0,0,0,0 */
 li            {}  /* a=0 b=0 c=0 d=1 -> specificity = 0,0,0,1 */
 li:first-line {}  /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
 ul li         {}  /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
 ul ol+li      {}  /* a=0 b=0 c=0 d=3 -> specificity = 0,0,0,3 */
 h1 + *[rel=up]{}  /* a=0 b=0 c=1 d=1 -> specificity = 0,0,1,1 */
 ul ol li.red  {}  /* a=0 b=0 c=1 d=3 -> specificity = 0,0,1,3 */
 li.red.level  {}  /* a=0 b=0 c=2 d=1 -> specificity = 0,0,2,1 */
 #x34y         {}  /* a=0 b=1 c=0 d=0 -> specificity = 0,1,0,0 */
 style=""          /* a=1 b=0 c=0 d=0 -> specificity = 1,0,0,0 */
```

为规则分类

规则被匹配以后，它们会根据层叠的规则分类，Webkit 使用冒泡排序分成小的列表，再用归并排序得到大的,Webkit 通过重写规则的“>”操作符实现排序。

```
static bool operator >(CSSRuleData& r1, CSSRuleData& r2)
{
    int spec1 = r1.selector()->specificity();
    int spec2 = r2.selector()->specificity();
    return (spec1 == spec2) : r1.position() > r2.position() : spec1 > spec2;
```

}

渐进的处理

Webkit 使用一个标志位来标记是否所有的样式表（包括@imports）都被载入了，如果指派的时候样式没有被完全载入—就会使用占位符，并且把它在文档中标记，一旦样式表被载入了，它们就会被重新计算。

布局

当渲染器被创建并添加到树上的时候，它没有位置和大小，计算这些值被叫做布局或者回流。
HTML 使用基于布局模型的流，意思是大多数时候是一次就能计算出这些几何信息，“流”中后面的元素并不会影响“流”中前面元素的几何信息，因此布局能够从文档中自左向右、自上向下进行。当然也有例外—例如，HTML 表格可能需要多于一次的操作（3.5）。
坐标系统是相对于根部对象的， 上和左坐标被使用。
布局是个递归的过程，从对应于 HTML 文档元素的根部解析器开始，穿过一部分或者全部解析器，递归地计算每个有需要的解析器的几何信息。
根解析器的位置是 0，0 和它的大小是视角区—浏览器窗口的可见的部分。
所有解析器有一个“layout”或者“reflow”方法，每个解析器调用需要布局的子节点的 layout 方法。

重写标志位系统

为了不为了每个小的改变都做完全的布局，浏览器使用“重写标志位”系统，被改变或者被添加的解析器会把它自身和它的子节点标记成“重写的”—需要布局。
有两个标记—“dirty”和“children are dirty”，子节点重写的意思是尽管解析器本身可能是没问题的，但是至少有一个子节点需要布局。

全局的和增量的布局

布局能够在整个树上被触发—这是“全局的”布局，这种情况发生是由于：
1. 影响所有解析器的全局样式改变，比如字体大小改变
2. 屏幕改变大小

布局也可以是增量的，只有重写的解析器被布局（这会导致需要额外布局的一些危害），当解析器是重写状态的时候，增量式布局被触发（异步的），例如，当额外的内容从网络中进来并且添加到 DOM 树之后，新的解析器被添加到渲染树上的时候。

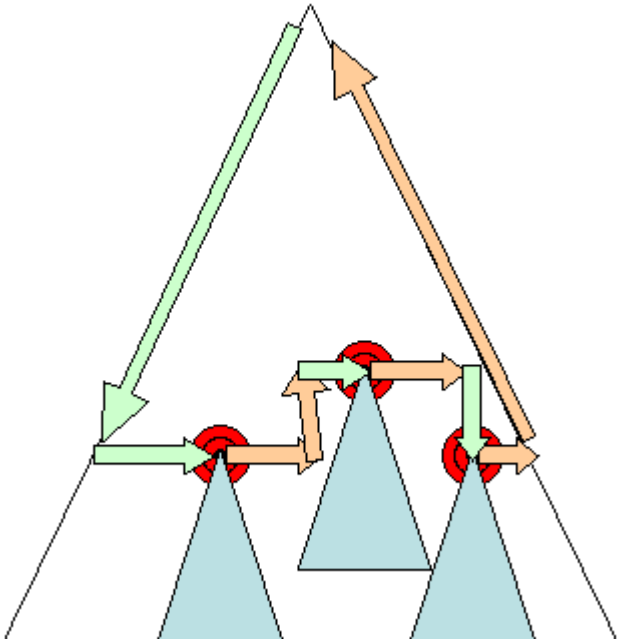


图 20 增量式布局—只有重写状态的解析器和它们的子孙被布局(3.6)

异步的和同步的布局

增量式布局是异步完成的。Firefox 为增量式布局把“回流命令”排成队列并且用一个时间表来触发这些命令的批量执行，Webkit 同样有一个执行增量式布局的计时器—树被遍历并且“重写的”解析器被重新布局。
脚本请求样式信息，比如“offsetHeight”会同步地触发增量式布局。
全局式布局通常会被同步地触发。
有时会因为一些属性，比如滚动位置的改变，布局被当做初始布局后的一个回调被触发。

优化

当布局被“resize”或者解析器位置（并且不是大小）的改变触发的时候，解析器的尺寸会从缓存中获取，不会重新计算。
有一些情况—只有子树被改变并且布局不从根部开始，当改变是局部的并且不影响周围环境的时候这种情况会发生，就像文本插入到文本域里（其它情况每一次键入都触发了从根部开始布局）。

布局过程

布局通常有下面的模式：
1. 父解析器决定它自身的宽度
2. 父解析器检查它的子节点并且：
    1. 放置这个子解析器（设置它的 x 和 y）
    2. 如果需要调用子解析器布局（子解析器是重写的或者在全局式布局下或者一些其它的原因）--这计算子解析器的高

3.父解析器使用子解析器累积的高和 margin、padding 的高来设置它自身的高—这会被父解析器的父解析器使用。
4.把它的重写标志为设置为 false
Firefox 使用“状态”对象（nsHTMLReflowState）作为参数来布局（Firefox 叫作回流），其中这个状态包括父解析器的宽，Firefox 布局输出的是一个“度量”对象（nsHTMLReflowMetrics），它包含解析器计算出的高。

宽度计算

解析器的宽度是用包含块的宽计算的，解析器样式的“width”属性、边距和边框。
例如下面 div 的宽度：

```
<div style="width:30%"/>
```

Webkit 中计算如下（RenderBox 类 calcWidth 方法）：
1. 容器宽度是容器 availableWidth 和 0 的最大值，这种情况里的 availableWidth 是如下计算出的 contentWidth：

```
clientWidth() - paddingLeft() - paddingRight()
```

clientWidth 和 clientHeight 表示一个对象不包含边框和滚动条的内侧部分。

2. 元素宽度是“width”样式属性，它将会通过计算容器的百分比得出一个绝对值。
3. 水平的边框和内边距被添加。
到现在为止是“首选宽度”的计算，现在最小和最大宽度将会被计算。
如果首选宽度比最大宽度要大—最大宽度被使用，如果比最小宽度（最小的不可拆的单元）小，最小宽度被使用。
这个值会被缓存，以防需要布局但是宽度不改变的情况。

断行

当布局当中的解析器决定需要断行的时候，布局会停止并且告知它的父解析器它需要断行，父解析器会创建额外的解析器并且请求对它布局。

绘制

在绘制阶段，解析树被遍历，解析器的“paint”方法被调用来把内容显示在屏幕上。绘制使用 UI 基础组件，这方面更多的东西在关于 UI 的那章里。

全局的和递增的

和布局一样，绘制也可以是全局式的—整个树被绘制—或者增量式的。在增量式的绘制中，解析器中的一些以不影响整个树的方式改变。改变了的解析器使屏幕上它的区域无效，这导致操作系统把它看作是“重写区域”并且触发“绘制”方法，操作系统系统很巧妙的做这些，它会把几个区域合并成一个。在 Chrome 里变得更加复杂，因为解析器是在一个不同的进程而不是主进程里，Chrome 在一定程度上模仿操作系统，这种方式会监听这些事件，并且把消息委托给渲染树的根节点，这个树被遍历直到到达对应的解析器，解析器会重绘它自身（通常还有它的子节点）。

绘制顺序

CSS2 定义了绘制过程的顺序-- http://www.w3.org/TR/CSS21/zindex.html，正是元素存储于栈式上下文时的栈内顺序，这个顺序会影响绘制，因为栈是从后向前绘制的，块状解析器的栈内顺序是：

1. 背景颜色
2. 背景图片
3. 边框
4. 子节点
5. 轮廓

Firefox 显示列表

Firefox 查看渲染树并且为绘制过矩形区域创建一个显示列表，它包含这个矩形区域所对应的解析器，并且是在正确的绘制顺序下（解析器的背景，然后是边框等等）。
这种方式这个树只需要遍历一次而不是几次—绘制所有的背景，然后所有的图片，然后所有的边框等等。
Firefox 通过不添加将会被隐藏的元素来优化这个过程，比如一个元素完全的在一个非透明的元素下面。

Webkit 矩形区域存储

在绘制前，webkit 把旧的矩形区域保存成位图，然后只绘制新的和就的矩形区域之间的地带。

动态改变

浏览器尽量做最小可能的动作来响应改变，所以一个元素颜色的改变只会引起这个元素的重绘，元素位置的改变会引起这个元素、它的子节点以及可能的兄弟节点的布局和重绘，添加 DOM 节点会引起这个节点的布局和重绘，大范围的改变，就像增大“html”元素的字体大小，会引起缓存的无效，整个树的重新布局和重绘。

渲染引擎的线程

渲染引擎是单线程的，除了网络操作，所有事情都在一个线程里发生，在 Firefox 和 safari 中，是在浏览器的主线程，在 Chrome 中，是在选项卡进程的主线程里。
网络操作是通过几个并行线程执行的，并行连接的数量是有限制的（通常是 2-6 个，例如 Firefox 就是用 6 个）。

事件环

浏览器的主线程是一个事件环，它是一个保持进程活跃的无限循环，等待事件（比如布局和绘制

事件）并处理它们，这是 Firefox 里主事件环的代码：

```
while (!mExiting)
    NS_ProcessNextEvent(thread);
```

CSS2 视觉性模型

canvas

根据 CSS2 规范，名词 canvas 描述了“渲染格式化结构的空间”—浏览器绘制内容的地方。
canvas 中空间的大小是无限的，但是浏览器选择了一个基于视角大小的初始宽度。
根据 http://www.w3.org/TR/CSS2/zindex.html，包含在另一个 canvas 里的 canvas 是透明的，还有如果它没有指定颜色，就会被指定一个浏览器定义的颜色。

CSS 盒模型

CSS 盒模型描述了这个为文档树中元素生成并且根据视觉性格式化模型放置的矩形盒子，每个盒子有一个内容区域（比如文本、图片等等）以及可选的内边距、边框、边距区域。

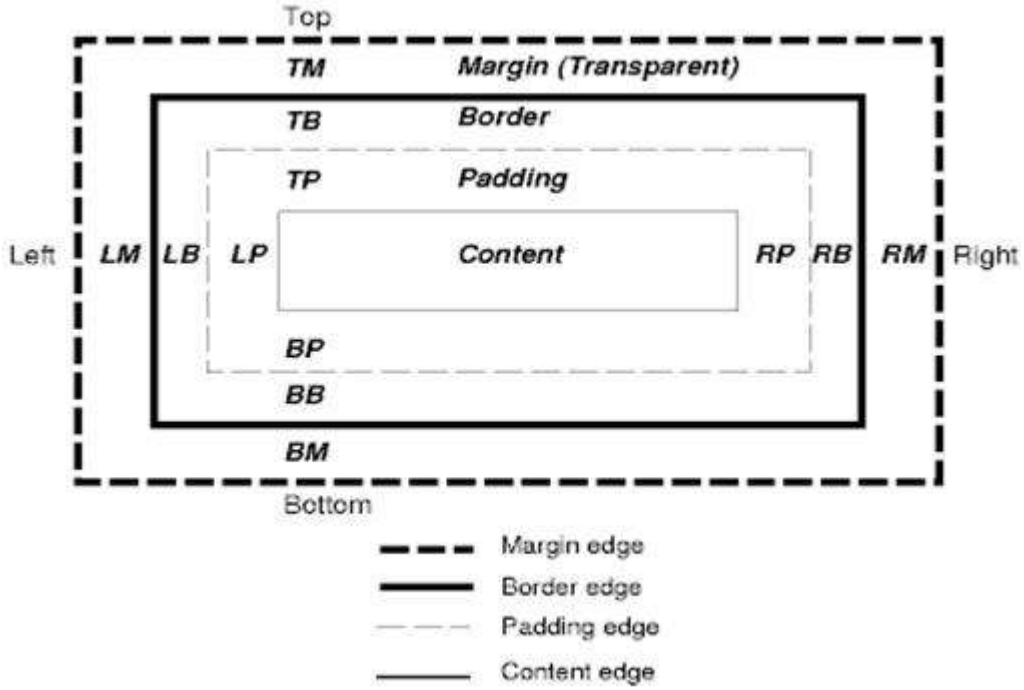


图 14 CSS2 盒模型

每个节点生成 0…n 这样的盒子。
所有元素都有一个决定它们所生成盒子的类型的 display 属性。
例子：

```
block  - generates a block box.
inline - generates one or more inline boxes.
none - no box is generated.
```

默认的是内联的但是浏览器样式表设置了其他的默认样式，例如—“div”元素默认的显示是块状的。
你可以在这找到默认的样式表实例-- http://www.w3.org/TR/CSS2/sample.htm。

定位方式

定位有三种方式：

1. Normal—对象根据它在文档中的位置定位—这意味着它在渲染树中的位置和它在 DOM 树中的位置以及根据它的盒类型和大小放置的位置是一样的。
2. Float—对象首先像普通流一样被放置，然后被移动到最左或者最右边。
3. Absolute—对象被放置在渲染树上的位置和在 DOM 树上位置不一样。

定位是通过“position”和“float”属性设置的。

1. static 和 relative 导致普通流
2. absolute 和 fixed 导致绝对定位

在静态定位中没有定位被定义，默认的位置被使用，在其他的方式中，开发者指定位置—top、bottom、left、right。

盒子被放置的方式由下面决定：

- 盒子类型
- 盒子大小
- 定位方式
- 外部信息—比如图片大小和屏幕尺寸

盒子类型

块状盒子—形成一个块—在浏览器窗口上有它们自己的矩形区域

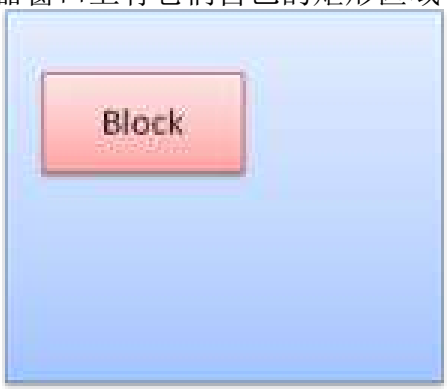


图 15 块状盒子

内联盒子—没有它自己的块，但是在一个包含块内部

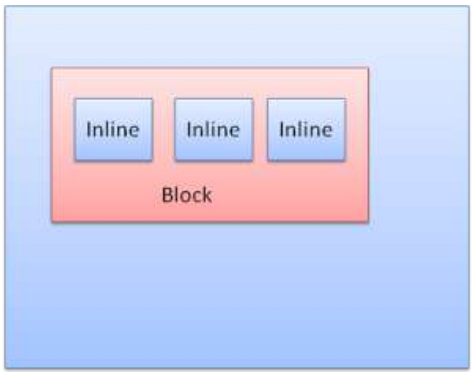


图 15 内联盒子

块状盒子一个接一个的垂直地编排，内联盒子水平地编排。

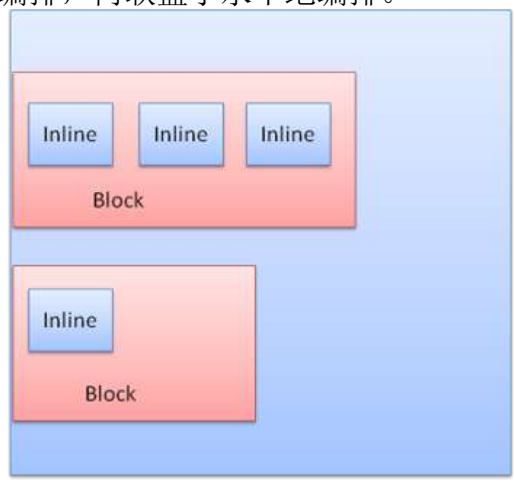


图 16 块状和内联盒子排版

内联盒子放置在行或者“行盒子”里，这个行至少要和最高的盒子一样高，但是也可以更高，当

盒子以“baseline”对齐的时候—意思是一个元素的底部和另一个盒子一点而不是底部对齐。一旦容器宽度不够，内连盒子会放在几行里，这通常发生在段落里。

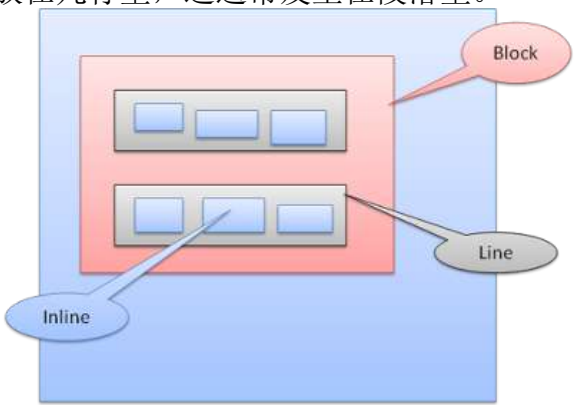


图 17 行

定位

Relative

相对定位—像普通的一样定位，然后根据一个必要的差值移动。

```
<html>
        <div>
                <span>1</span>
                <span>2</span>
                <span style="position:relative;left:5px">3</span>
        </div>
</html>
```

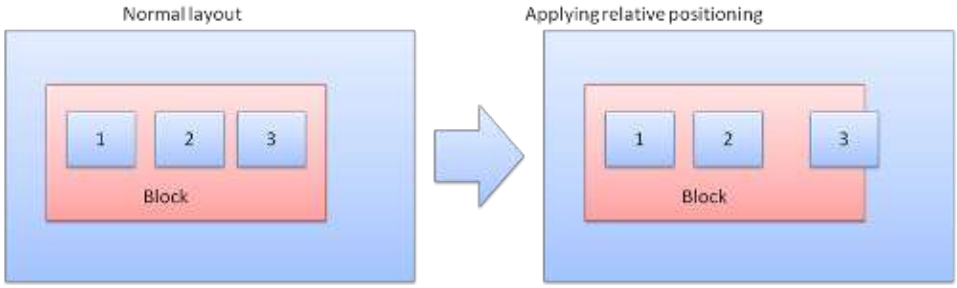


图 18 相对定位

float

浮动的盒子移动到一行的左或右边，有趣的特性是其它盒子围绕它流动。

HTML：

```
<p>
<img style="float:right" src="images/image.gif" width="100" height="100">Lorem ipsum dolor sit amet, consectetuer...
</p>
```

将会这样：

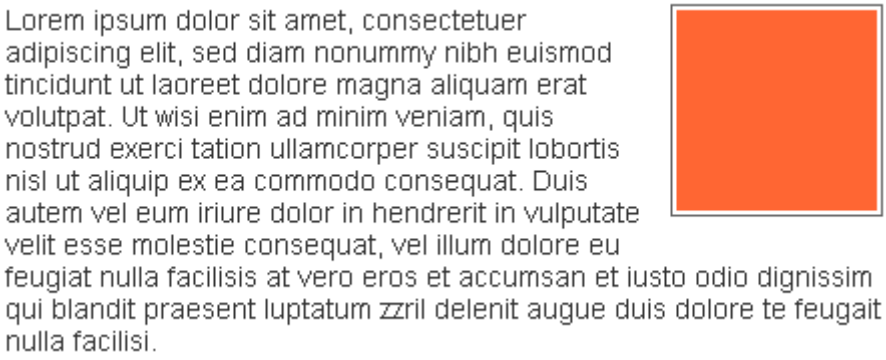


图 19 浮动

Absolute and fixed

这种布局不考虑普通流，元素不参与普通流，大小与容器有关，在 fixed 方式里—容器是可见区。

```
<html>
        <div>
                <span>1</span>
                <span>2</span>
                <span style="position:fixed;top:5px,left:5px">3</span>
        </div>
</html>
```

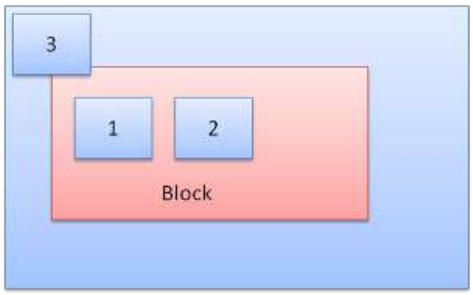


图 20 固定定位

标注—即便文档股东的时候固定定位的盒子也不移动。

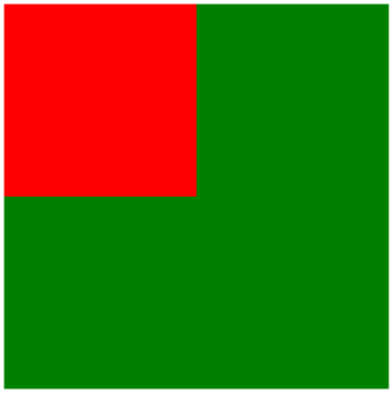
层表现

这是被 CSS 属性 z-index 指定的，代表盒子的第三维度，z 轴上的位置。
盒子被分隔到多个栈里（叫作栈式上下文），在每个栈里后面的元素会先被绘制，然后是靠近用户的前面的元素，一旦重叠，就会隐藏前面的元素。
这些栈根据 z-index 属性排序，有 zindex 属性的盒子形成一个局部栈，可见拥有外层的栈。
例子：

```
<STYLE type="text/css">
      div {
        position: absolute;
        left: 2in;
        top: 2in;
      }
    </STYLE>

  <P>
            <DIV
                 style="z-index: 3;background-color:red; width: 1in; height: 1in; ">
            </DIV>
            <DIV
                 style="z-index: 1;background-color:green;width: 2in; height: 2in;">
            </DIV>
   </p>
```

结果是这样：



尽管红色的 div 在绿色的前面，并且在流中应该绘制在前面，但是它的 z-index 更高，所以在根盒子持有的栈中它更靠前。

原文地址：http://taligarsiel.com/Projects/howbrowserswork1.htm
原作者：Tali Garsiel
译者：zh（新浪乐居） 微博：http://t.sina.com.cn/callback

注：书只写完了这些，后半段越来越有应付的感觉，错误明显增多，和原作者联系，回复说没有时间继续。

Introduction

Web browsers are probably the most widely used software. In this book I will explain how they work behind the scenes. We will see what happens when you type 'google.com' in the address bar until you see the Google page on the browser screen.

The browsers we will talk about

There are five major browsers used today - Internet Explorer, Firefox, Safari, Chrome and Opera.
I will give examples from the open source browsers - Firefox,Chrome and Safari, which is partly open source.
According to the W3C browser statistics, currently(October 2009), the usage share of Firefox, Safari and Chrome together is nearly 60%.
So nowdays open source browsers are a substantial part of the browser business.

The browser's main functionality

The browser main functionality is to present the web resource you choose, by requesting it from the server and displaying it on the browser window. The resource format is usually HTML but also PDF, image and more. The location of the resource is specified by the user using a URI (Uniform resource Identifier). More on that in the network chapter.

The way the browser interprets and displays HTML files is specified in the HTML and CSS specifications. These specifications are maintained by the W3C (World Wide Web Consortium) organization, which is the standards organization for the web.
The current version of HTML is 4 (http://www.w3.org/TR/html401/). Version 5 is in progress. The current CSS version is 2 (http://www.w3.org/TR/CSS2/) and version 3 is in progress.
For years browsers conformed to only a part of the specifications and developed their own extensions. That caused serious compatibility issues for web authors. Today most of the browsers more or less conform to the specifications.

Browsers' user interface have a lot in common with each other. Among the common user interface elements are:

- Address bar for inserting the URI
- Back and forward buttons
- Bookmarking options
- A refresh and stop buttons for refreshing and stopping the loading of current documents
- Home button that gets you to your home page

Strangely enough, the browser's user interface is not specified in any formal specification, it is just good practices shaped over years of experience and by browsers imitating each other. The HTML5 specification doesn't define UI elements a browser must have, but lists some common elements. Among those are the address bar, status bar and tool bar. There are, of course, features unique to a specific browser like Firefox downloads manager.
More on that in the user interface chapter.

The browser's high level structure

The browser's main components are (1.1):

1. The user interface - this includes the address bar, back/forward button, bookmarking menu etc. Every part of the browser display except the main window where you see the requested page.
2. The browser engine - the interface for querying and manipulating the rendering engine.
3. The rendering engine - responsible for displaying the requested content. For example if the requested content is HTML, it is responsible for parsing the HTML and CSS and displaying the parsed content on the screen.
4. Networking - used for network calls, like HTTP requests. It has platform independent interface and underneath implementations for each platform.
5. UI backend - used for drawing basic widgets like combo boxes and windows. It exposes a generic interface that is not platform specific. Underneath it uses the operating system user interface methods.
6. JavaScript interpreter. Used to parse and execute the JavaScript code.
7. Data storage. This is a persistence layer. The browser needs to save all sorts of data on the hard disk, for examples, cookies. The new HTML specification (HTML5) defines 'web database' which is a complete (although light) database in the browser.

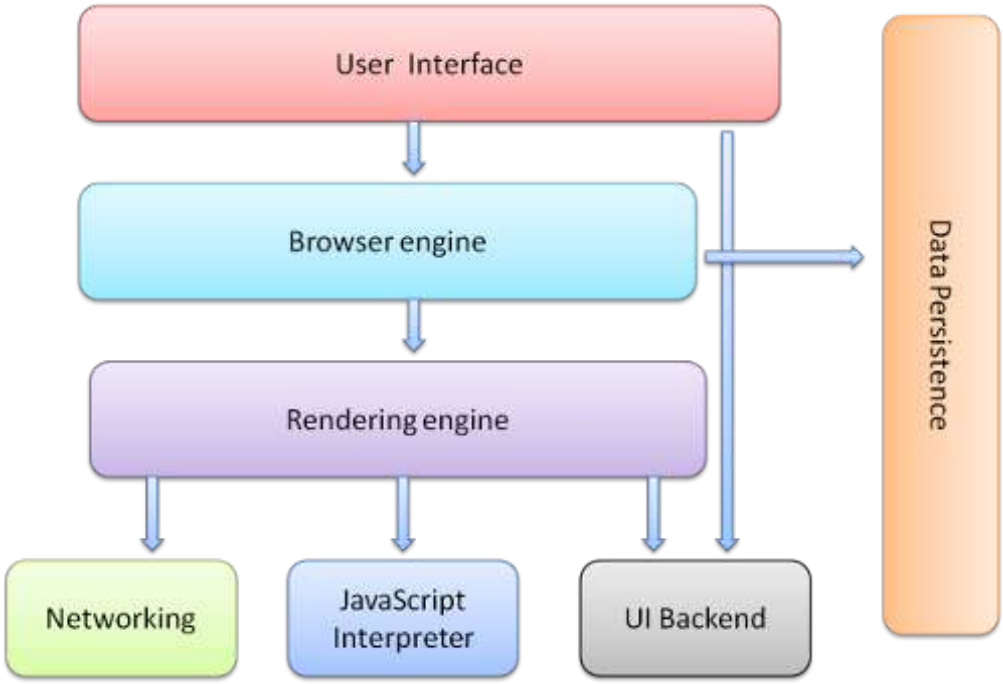


Figure 1: Browser main components.

It is important to note that Chrome, unlike most browsers, holds multiple instances of the rendering engine - one for each tab,. Each tab is a separate process.

I will devote a chapter for each of these components.

Communication between the components

Both Firefox and Chrome developed a special communication infrastructures.
They will be discussed in a special chapter.

The rendering engine

The responsibility of the rendering engine is well... Rendering, that is display of the requested contents on the browser screen.

By default the rendering engine can display HTML and XML documents and images. It can display other types through a plug-in (a browser extension). An example is displaying PDF using a PDF viewer plug-

in. We will talk about plug-ins and extensions in a special chapter. In this chapter we will focus on the main use case - displaying HTML and images that are formatted using CSS.

Rendering engines

Our reference browsers - Firefox, Chrome and Safari are built upon two rendering engines. Firefox uses Gecko - a "home made" Mozilla rendering engine. Both Safari and Chrome use Webkit.

Webkit is an open source rendering engine which started as an engine for the Linux platform and was modified by Apple to support Mac and Windows. See http://webkit.org/ for more details.

The main flow

The rendering engine will start getting the contents of the requested document from the networking layer. This will usually be done in 8K chunks.

After that this is the basic flow of the rendering engine:

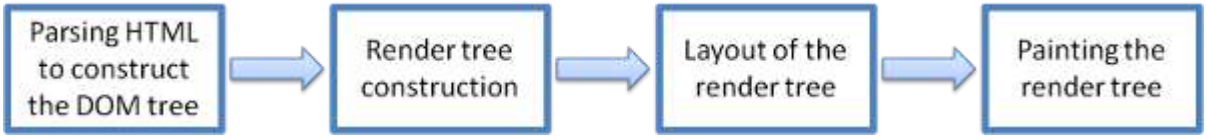


Figure 2:Rendering engine basic flow.

The rendering engine will start parsing the HTML document and turn the tags to DOM nodes in a tree called the "content tree". It will parse the style data, both in external CSS files and in style elements. The styling information together with visual instructions in the HTML will be used to create another tree - the render tree.

The render tree contains rectangles with visual attributes like color and dimensions. The rectangles are in the right order to be displayed on the screen.

After the construction of the render tree it goes through a "layout" process. This means giving each node the exact coordinates where it should appear on the screen. The next stage is painting - the render tree will be traversed and each node will be painted using the UI backend layer.

It's important to understand that this is a gradual process. For better user experience, the rendering engine will try to display contents on the screen as soon as possible. It will not wait until all HTML is parsed before starting to build and layout the render tree. Parts of the content will be parsed and displayed, while the process continues with the rest of the contents that keeps coming from the network.

Main flow examples

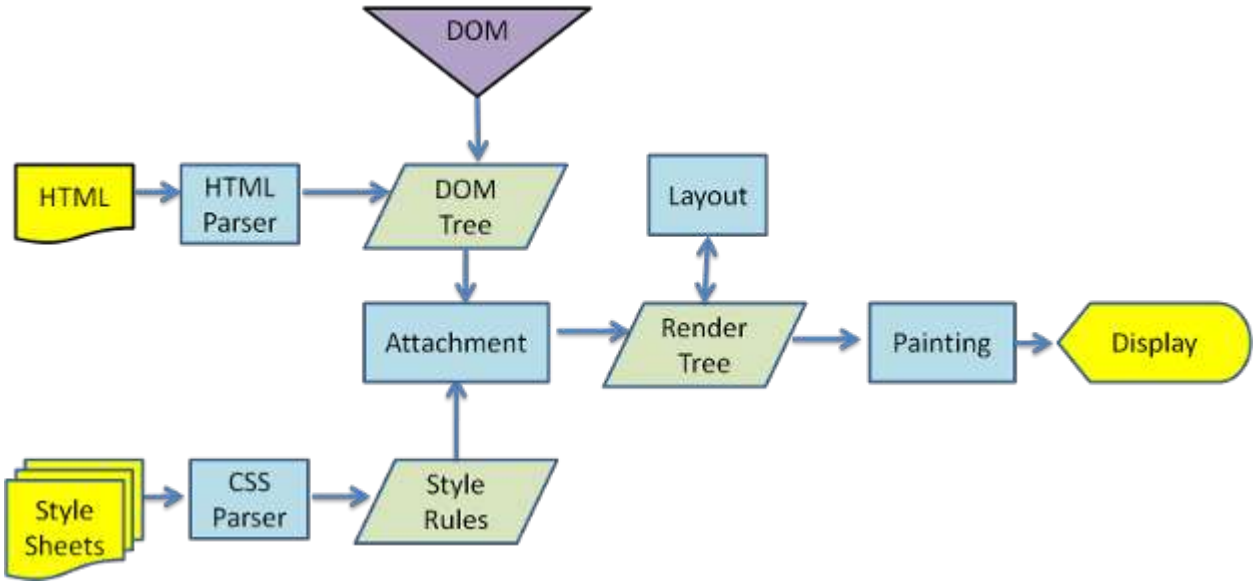


Figure 3: Webkit main flow

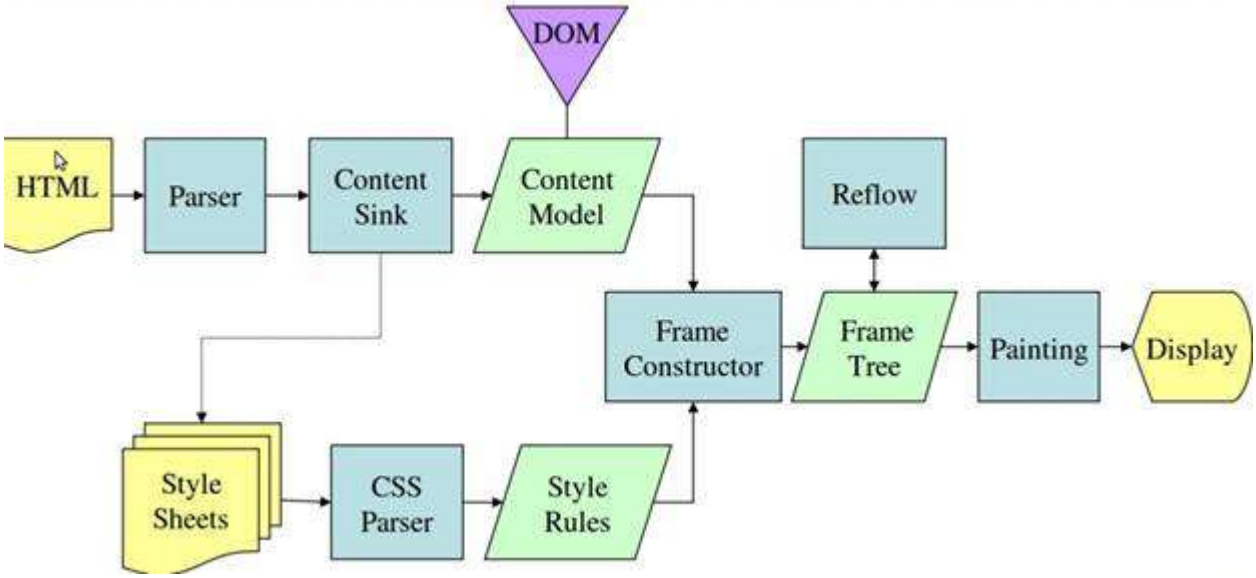


Figure 4: Mozilla's Gecko rendering engine main flow(3.6)

From figures 3 and 4 you can see that although Webkit and Gecko use slightly different terminology, the flow is basically the same.
Gecko calls the tree of visually formatted elements - Frame tree. Each element is a frame. Webkit uses the term "Render Tree" and it consists of "Render Objects". Webkit uses the term "layout" for the placing of elements, while Gecko calls it "Reflow". "Attachment" is Webkit's term for connecting DOM nodes and visual information to create the render tree. A minor non semantic difference is that Gecko has an extra layer between the HTML and the DOM tree. It is called the "content sink" and is a factory for making DOM elements. We will talk about each part of the flow:

Parsing - general

Since parsing is a very significant process within the rendering engine, we will go into it a little more deeply. Let's begin with a little introduction about parsing.

Parsing a document means translating it to some structure that makes sense - something the code can understand and use. The result of parsing is usually a tree of nodes that represent the structure of the document. It is called a parse tree or a syntax tree.

Example - parsing the expression "2 + 3 - 1" could return this tree:

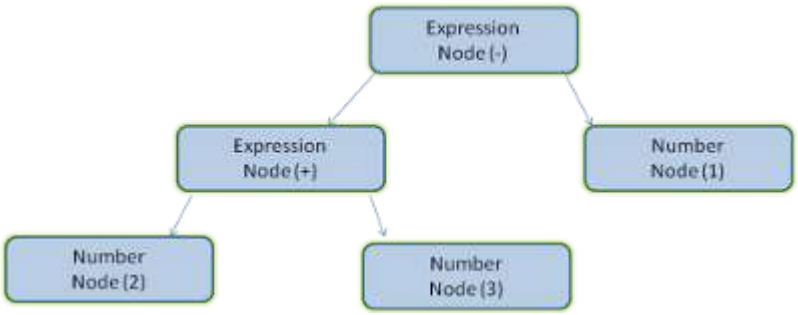


Figure 5: mathematical expression tree node

Grammars

Parsing is based on the syntax rules the document obeys - the language or format it was written in. Every format you can parse must have deterministic grammar consisting of vocabulary and syntax rules. It is called a context free grammar. Human languages are not such languages and therefore cannot be parsed with conventional parsing techniques.

Parser - Lexer combination

Parsing can be separated into two sub processes - lexical analysis and syntax analysis.

Lexical analysis is the process of breaking the input into tokens. Tokens are the language vocabulary - the collection of valid building blocks. In human language it will consist of all the words that appear in the dictionary for that language.

Syntax analysis is the applying of the language syntax rules.

Parsers usually divide the work between two components - the **lexer**(sometimes called tokenizer) that is responsible for breaking the input into valid tokens, and the **parser** that is responsible for constructing the parse tree by analyzing the document structure according to the language syntax rules. The lexer knows how to strip irrelevant characters like white spaces and line breaks.

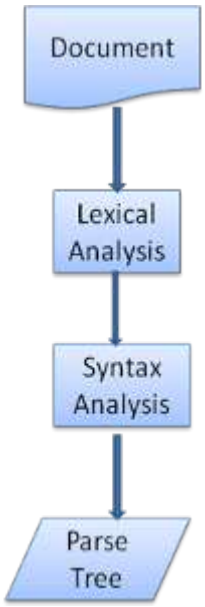


Figure 6: from source document to parse trees

The parsing process is iterative. The parser will usually ask the lexer for a new token and try to match the

token with one of the syntax rules. If a rule is matched, a node corresponding to the token will be added to the parse tree and the parser will ask for another token.
If no rule matches, the parser will store the token internally, and keep asking for tokens until a rule matching all the internally stored tokens is found. If no rule is found then the parser will raise an exception. This means the document was not valid and contained syntax errors.

Translation

Many times the parse tree is not the final product. Parsing is often used in translation - transforming the input document to another format. An example is compilation. The compiler that compiles a source code into machine code first parses it into a parse tree and then translates the tree into a machine code document.

Figure 7: compilation flow

Parsing example

In figure 5 we built a parse tree from a mathematical expression. Let's try to define a simple mathematical language and see the parse process.

Vocabulary: Our language can include integers, plus signs and minus signs.

Syntax:

1. The language syntax building blocks are expressions, terms and operations.
2. Our language can include any number of expressions.
3. A expression is defined as a "term" followed by an "operation" followed by another term
4. An operation is a plus token or a minus token
5. A term is an integer token or an expression

Let's analyze the input "2 + 3 - 1".
The first substring that matches a rule is "2", according to rule #5 it is a term. The second match is "2 + 3" this matches the second rule - a term followed by an operation followed by another term. The next match

will only be hit at the end of the input. "2 + 3 - 1" is an expression because we already know that ?2+3? is a term so we have a term followed by an operation followed by another term. "2 + + "will not match any rule and therefore is an invalid input.

Formal definitions for vocabulary and syntax

Vocabulary is usually expressed by regular expressions.

For example our language will be defined as:

```
INTEGER :0|[1-9][0-9]*
PLUS : +
MINUS: -
```

As you see, integers are defined by a regular expression.

Syntax is usually defined in a format called BNF. Our language will be defined as:

```
expression :=  term  operation  term
operation :=  PLUS | MINUS
term := INTEGER | expression
```

We said that a language can be parsed by regular parsers if its grammar is a context frees grammar. An intuitive definition of a context free grammar is a grammar that can be entirely expressed in BNF. For a formal definition see http://en.wikipedia.org/wiki/Context-free_grammar

Types of parsers

There are two basic types of parsers - top down parsers and bottom up parsers. An intuitive explanation is that top down parsers see the high level structure of the syntax and try to match one of them. Bottom up parsers start with the input and gradually transform it into the syntax rules, starting from the low level rules until high level rules are met.

Let's see how the two types of parsers will parse our example:

Top down parser will start from the higher level rule - it will identify "2 + 3" as an expression. It will then identify "2 + 3 - 1" as an expression (the process of identifying the expression evolves matching the other rules, but the start point is the highest level rule).

The bottom up parser will scan the input until a rule is matched it will then replace the matching input with the rule. This will go on until the end of the input. The partly matched expression is placed on the parsers stack.

Stack	Input
	2 + 3 - 1
term	+ 3 - 1
term operation	3 - 1
expression	- 1

expression
operation 1

expression

This type of bottom up parser is called a shift reduce parser, because the input is shifted to the right (imagine a pointer pointing first at the input start and moving to the right) and is gradually reduced to syntax rules.

Generating parsers automatically

There are tools that can generate a parser for you. They are called parser generators. You feed them with the grammar of your language - its vocabulary and syntax rules and they generate a working parser. Creating a parser requires a deep understanding of parsing and its not easy to create an optimized parser by hand, so parser generators can be very useful.

Webkit uses two well known parser generators - Flex for creating a lexer and Bison for creating a parser (you might run into them with the names Lex and Yacc). Flex input is a file containing regular expression definitions of the tokens. Bison's input is the language syntax rules in BNF format.

HTML Parser

The job of the HTML parser is to parse the HTML markup into a parse tree.

The HTML grammar definition

The vocabulary and syntax of HTML are defined in specifications created by the w3c organization. The current version is HTML4 and work on HTML5 is in progress.

Not a context free grammar

As we have seen in the parsing introduction, grammar syntax can be defined formally using formats like BNF.
Unfortunately all the conventional parser topics do not apply to HTML (I didn't bring them up just for fun - they will be used in parsing CSS and JavaScript). HTML cannot easily be defined by a context free grammar that parsers need.
There is a formal format for defining HTML - DTD (Document Type Definition) - but it is not a context free grammar.
This appears strange at first site - HTML is rather close to XML .There are lots of available XML parsers. There is an XML variation of HTML - XHTML - so what's the big difference?
The difference is that HTML approach is more "forgiving", it lets you omit certain tags which are added implicitly, sometimes omit the start or end of tags etc. On the whole it's a "soft" syntax, as opposed to XML's stiff and demanding syntax.
Apparently this seemingly small difference makes a world of a difference. On one hand this is the main reason why HTML is so popular - it forgives your mistakes and makes life easy for the web author. On the other hand, it makes it difficult to write a format grammar. So to summarize - HTML cannot be parsed easily, not by conventional parsers since its grammar is not a context free grammar, and not by XML parsers.

HTML DTD

HTML definition is in a DTD format. This format is used to define languages of the SGML family. The format contains definitions for all allowed elements, their attributes and hierarchy. As we saw earlier, the HTML DTD doesn't form a context free grammar.

There are a few variations of the DTD. The strict mode conforms solely to the specifications but other modes contain support for markup used by browsers in the past. The purpose is backwards compatibility with older content. The current strict DTD is here: http://www.w3.org/TR/html4/strict.dtd

DOM

The output tree - the parse tree is a tree of DOM element and attribute nodes. DOM is short for Document Object Model. It is the object presentation of the HTML document and the interface of HTML elements to the outside world like JavaScript.
The root of the tree is the "Document" object.

The DOM has an almost one to one relation to the markup. Example, this markup:

```
<html>
        <body>
                <p>
                        Hello World
                </p>
                <div> <img src="example.png"/></div>
        </body>
</html>
```

Would be translated to the following DOM tree:

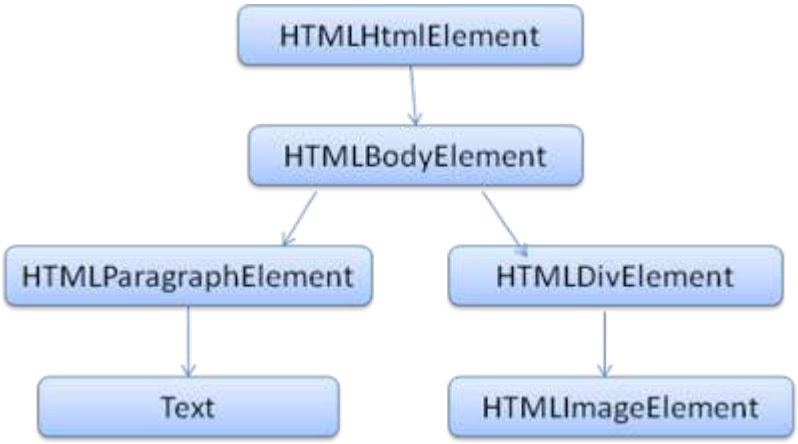


Figure 8: DOM tree of the example markup

Like HTML, DOM is specified by the w3c organization. See http://www.w3.org/DOM/DOMTR. It is a generic specification for manipulating documents. A specific module describes HTML specific elements. The HTML definitions can be found here: http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/idl-definitions.html.

When I say the tree contains DOM nodes, I mean the tree is constructed of elements that implement one of the DOM interfaces. Browsers use concrete implementations that have other attributes used by the browser internally.

The parsing algorithm

As we saw in the previous sections, HTML cannot be parsed using the regular top down or bottom up

parsers.

The reasons are:

1. The forgiving nature of the language.
2. The fact that browsers have traditional error tolerance to support well known cases of invalid HTML.
3. The parsing process in reentrant. Usually the source doesn't change during parsing, but in HTML, script tags containing "document.write" can add extra tokens, so the parsing process actually modifies the input.

Unable to use the regular parsing techniques, browsers create custom parsers for parsing HTML.

The parsing algorithm is described in details by the HTML5 specification. The algorithm consists of two stages - tokenization and tree construction.

Tokenization is the lexical analysis, parsing the input into tokens. Among HTML tokens are start tags, end tags, attribute names and attribute values.

The tokenizer recognizes the token, gives it to the tree constructor and consumes the next character for recognizing the next token and so on until the end of the input.

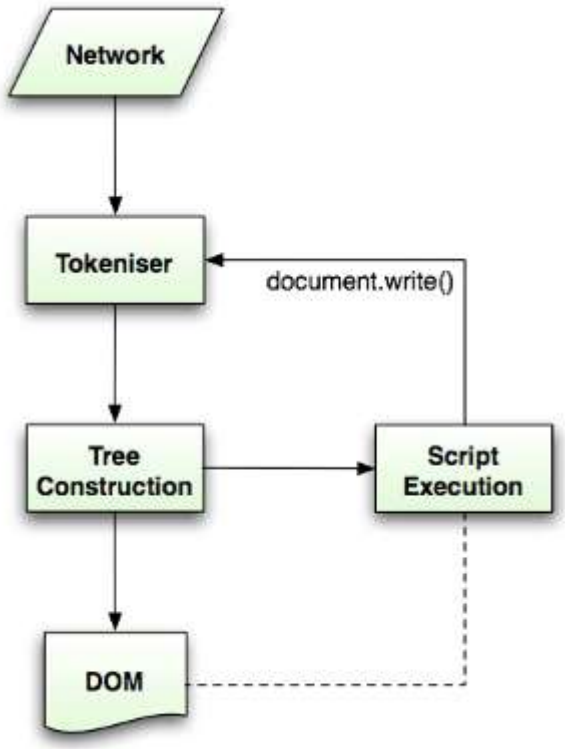


Figure 6: HTML parsing flow (taken from HTML5 spec)

The tokenization algorithm

The algorithm's output is an HTML token. The algorithm is expressed as a state machine. Each state consumes one or more characters of the input stream and updates the next state according to those characters. The decision is influenced by the current tokenization state and by the tree construction state. This means the same consumed character will yield different results for the correct next state, depending on the current state. The algorithm is too complex to bring fully, so let's see a simple example that will help us understand the principal.

Basic example - tokenizing the following HTML:

```
<html>
        <body>
                Hello world
        </body>
</html>
```

The initial state is the "Data state". When the "<" character is encountered, the state is changed to **"Tag open state"**. Consuming an "a-z" character causes creation of a "Start tag token", the state is change to **"Tag name state"**. We stay in this state until the ">" character is consumed. Each character is appended to the new token name. In our case the created token is an "html" token.
When the ">" tag is reached, the current token is emitted and the state changes back to the **"Data state"**. The "<body>" tag will be treated by the same steps. So far the "html" and "body" tags were emitted. We are now back at the **"Data state"**. Consuming the "H" character of "Hello world" will cause creation and emitting of a character token, this goes on until the "<" of "</body>" is reached. We will emit a character token for each character of "Hello world".
We are now back at the **"Tag open state"**. Consuming the next input "/" will cause creation of an "end tag token" and a move to the **"Tag name state"**. Again we stay in this state until we reach ">".Then the new tag token will be emitted and we go back to the **"Data state"**. The "</html>" input will be treated like the previous case.

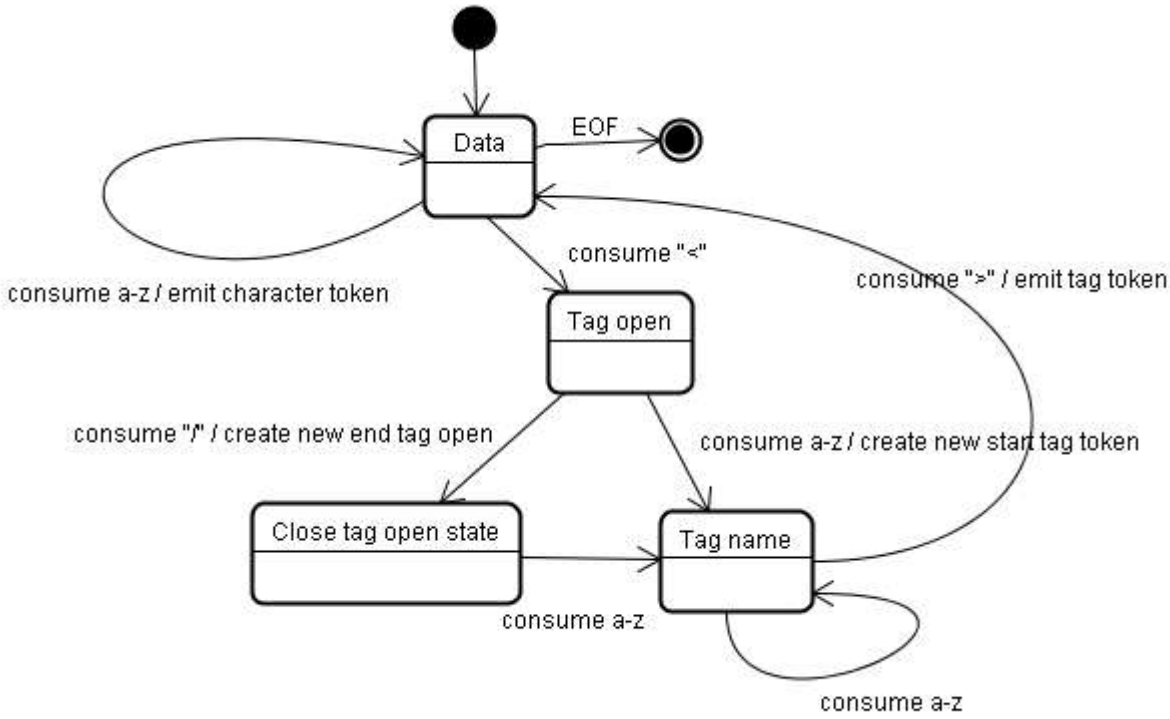


Figure 9: Tokenizing the example input

Tree construction algorithm

When the parser is created the Document object is created. During the tree construction stage the DOM tree with the Document in its root will be modified and elements will be added to it. Each node emitted by the tokenizer will be processed by the tree constructor. For each token the specification defines which DOM element is relevant to it and will be created for this token. Except of adding the element to the DOM tree it is also added to a stack of open elements. This stack is used to correct nesting mismatches and unclosed tags. The algorithm is also described as a state machine. The states are called "insertion modes".

Let's see the tree construction process for the example input:

```
<html>
        <body>
                Hello world
        </body>
</html>
```

The input to the tree construction stage is a sequence of tokens from the tokenization stage The first mode is the **"initial mode"**. Receiving the html token will cause a move to the **"before html"** mode and a reprocessing of the token in that mode. This will cause a creation of the HTMLHtmlElement element and it will be appended to the root Document object.
The state will be changed to **"before head"**. We receive the "body" token. An HTMLHeadElement will be created implicitly although we don't have a "head" token and it will be added to the tree.
We now move to the **"in head"** mode and then to **"after head"**. The body token is reprocessed, an HTMLBodyElement is created and inserted and the mode is transferred to **"in body"**.
The character tokens of the "Hello world" string are now received. The first one will cause creation and insertion of a "Text" node and the other characters will be appended to that node.
The receiving of the body end token will cause a transfer to **"after body"** mode. We will now receive the html end tag which will move us to **"after after body"** mode. Receiving the end of file token will end the parsing.

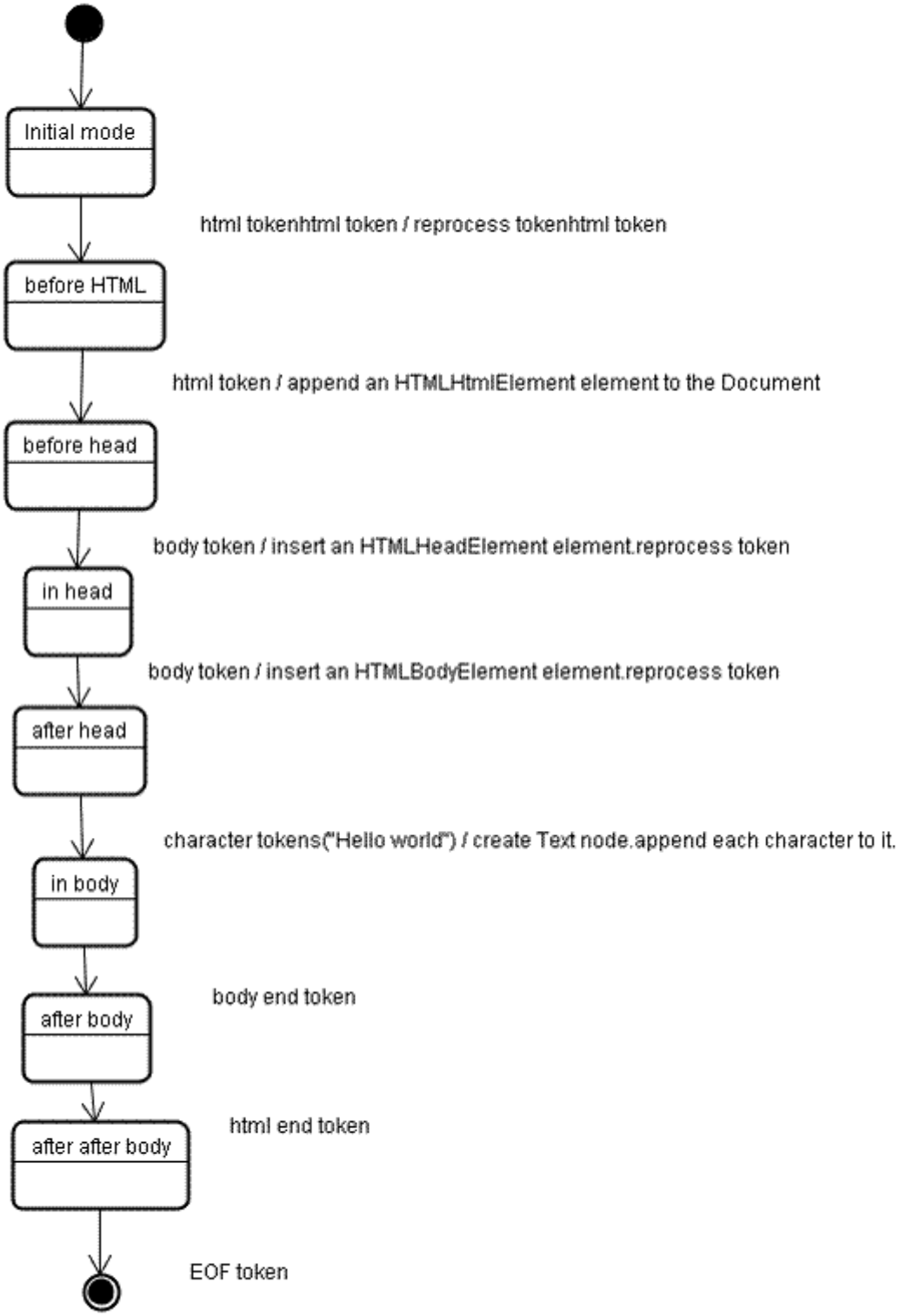


Figure 10: tree construction of example html

Actions when the parsing is finished

At this stage the browser will mark the document as interactive and start parsing scripts that are in "deferred" mode - those who should be executed after the document is parsed. The document state will be then set to "complete" and a "load" event will be fired.

You can see the full algorithms for tokenization and tree construction in HTML5 specification - http://www.w3.org/TR/html5/syntax.html#html-parser

Browsers error tolerance

You never get an "Invalid Syntax" error on an HTML page. Browsers fix an invalid content and go on. Take this HTML for example:

```
<html>
  <mytag>
  </mytag>
  <div>
  <p>
  </div>
        Really lousy HTML
  </p>
</html>
```

I must have violated about a million rules ("mytag" is not a standard tag, wrong nesting of the "p" and "div" elements and more) but the browser still shows it correctly and doesn't complain. So a lot of the parser code is fixing the HTML author mistakes.

The error handling is quite consistent in browsers but amazingly enough it's not part of HTML current specification. Like bookmarking and back/forward buttons it's just something that developed in browsers over the years. There are known invalid HTML constructs that repeat themselves in many sites and the browsers try to fix them in a conformant way with other browsers.

The HTML5 specification does define some of these requirements. Webkit summarizes this nicely in the comment at the beginning of the HTML parser class

```
The parser parses tokenized input into the document, building up the document tree.
If the document is well-formed, parsing it is straightforward.

Unfortunately, we have to handle many HTML documents that are not well-formed, so the
parser has to be tolerant about errors.

We have to take care of at least the following error conditions:

1. The element being added is explicitly forbidden inside some outer tag.
In this case we should close all tags up to the one, which forbids the element, and
add it afterwards.

2. We are not allowed to add the element directly.
It could be that the person writing the document forgot some tag in between (or that
the tag in between is optional).
This could be the case with the following tags: HTML HEAD BODY TBODY TR TD LI (did I
forget any?).

3. We want to add a block element inside to an inline element. Close all inline
elements up to the next higher block element.

4. If this doesn't help, close elements until we are allowed to add the element or
ignore the tag.
```

Let's see some Webkit error tolerance examples:

</br> instead of

Some sites use </br> instead of
. In order to be compatible with IE and Firefox Webkit treats this like
.
The code:

```
if (t->isCloseTag(brTag) && m_document->inCompatMode()) {
     reportError(MalformedBRError);
     t->beginTag = true;
```

```
}
```

Note - the error handling is internal - it won't be presented to the user.

A stray table

A stray table is a table inside another table contents but not inside a table cell.
Like this example:

```
<table>
        <table>
                <tr><td>inner table</td></tr>
         </table>
        <tr><td>outer table</td></tr>
</table>
```

Webkit will change the hierarchy to two sibling tables:

```
<table>
        <tr><td>outer table</td></tr>
</table>
<table>
        <tr><td>inner table</td></tr>
 </table>
```

The code:

```
if (m_inStrayTableContent && localName == tableTag)
        popBlock(tableTag);
```

Webkit uses a stack for the current element contents - it will pop the inner table out of the outer table stack. The tables will now be siblings.

Nested form elements

In case the user puts a form inside another form, the second form is ignored.
The code:

```
if (!m_currentFormElement) {
        m_currentFormElement = new HTMLFormElement(formTag,    m_document);
}
```

A too deep tag hierarchy

The comment speaks for itself.

```
www.liceo.edu.mx is an example of a site that achieves a level of nesting of about
1500 tags, all from a bunch of <b>s.
We will only allow at most 20 nested tags of the same type before just ignoring them
all together.

bool HTMLParser::allowNestedRedundantTag(const AtomicString& tagName)
{

unsigned i = 0;
for (HTMLStackElem* curr = m_blockStack;
          i < cMaxRedundantTagDepth && curr && curr->tagName == tagName;
     curr = curr->next, i++) { }
return i != cMaxRedundantTagDepth;
```

```
}
```

Misplaced html or body end tags

Again - the comment speaks for itself.

```
Support for really broken html.
We never close the body tag, since some stupid web pages close it before the actual
end of the doc.
Let's rely on the end() call to close things.

if (t->tagName == htmlTag || t->tagName == bodyTag )
        return;
```

So web authors beware - unless you want to appear as an example in a Webkit error tolerance code - write well formed HTML.

CSS parsing

Remember the parsing concepts in the introduction? Well, unlike HTML, CSS is a context free grammar and can be parsed using the types of parsers described in the introduction. In fact the CSS specification defines CSS lexical and syntax grammar (http://www.w3.org/TR/CSS2/grammar.html).

Let's see some examples:
The lexical grammar (vocabulary) is defined by regular expressions for each token:

```
comment         \/\*[^*]*\*+([^/*][^*]*\*+)*\/
num             [0-9]+|[0-9]*"."[0-9]+
nonascii        [\200-\377]
nmstart         [_a-z]|{nonascii}|{escape}
nmchar          [_a-z0-9-]|{nonascii}|{escape}
name            {nmchar}+
ident           {nmstart}{nmchar}*
```

"ident" is short for identifier, like a class name. "name" is an element id (that is referred by "#")

The syntax grammar is described in BNF.

```
ruleset
  : selector [ ',' S* selector ]*
    '{' S* declaration [ ';' S* declaration ]* '}' S*
  ;
selector
  : simple_selector [ combinator selector | S+ [ combinator selector ] ]
  ;
simple_selector
  : element_name [ HASH | class | attrib | pseudo ]*
  | [ HASH | class | attrib | pseudo ]+
  ;
class
  : '.' IDENT
  ;
element_name
  : IDENT | '*'
  ;
attrib
  : '[' S* IDENT S* [ [ '=' | INCLUDES | DASHMATCH ] S*
    [ IDENT | STRING ] S* ] ']'
  ;
pseudo
```

```
  : ':' [ IDENT | FUNCTION S* [IDENT S*] ')' ]
  ;
```

Explanation: A ruleset is this structure:

```
div.error , a.error {
        color:red;
        font-weight:bold;
}
```

div.error and a.error are selectors. The part inside the curly braces contains the rules that are applied by this ruleset. This structure is defined formally in this definition:

```
ruleset
  : selector [ ',' S* selector ]*
    '{' S* declaration [ ';' S* declaration ]* '}' S*
  ;
```

This means a ruleset is a selector or optionally number of selectors separated by a coma and spaces (S stands for white space). A ruleset contains curly braces and inside them a declaration or optionally a number of declarations separated by a semicolon. "declaration" and "selector" will be defined in the following BNF definitions.

Webkit CSS parser

Webkit uses Flex and Bison parser generators to create parsers automatically from the CSS grammar files. As you recall from the parser introduction, Bison creates a bottom up shift reduce parser. Firefox uses a top down parser written manually. In both cases each CSS file is parsed into a StyleSheet object, each object contains CSS rules. The CSS rule objects contain selector and declaration objects and other object corresponding to CSS grammar.

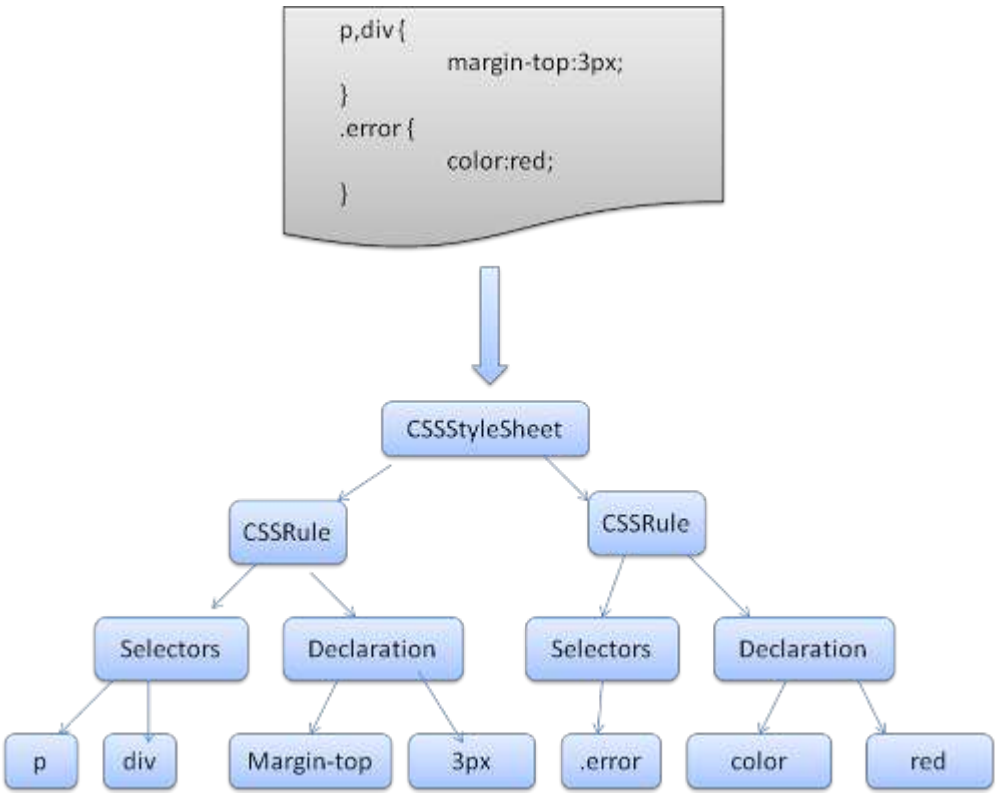


Figure 7: parsing CSS

Parsing scripts

This will be dealt with in the chapter about JavaScript

The order of processing scripts and style sheets

Scripts

The model of the web is synchronous. Authors expect scripts to be parsed and executed immediately when the parser reaches a <script> tag. The parsing of the document halts until the script was executed. If the script is external then the resource must be first fetched from the network - this is also done synchronously, the parsing halts until the resource is fetched. This was the model for many years and is also specified in HTML 4 and 5 specifications. Authors could mark the script as "defer" and thus it will not halt the document parsing and will execute after it is parsed. HTML5 adds an option to mark the script as asynchronous so it will be parsed and executed by a different thread.

Speculative parsing

Both Webkit and Firefox do this optimization. While executing scripts, another thread parses the rest of the document and finds out what other resources need to be loaded from the network and loads them. These way resources can be loaded on parallel connections and the overall speed is better. Note - the speculative parser doesn't modify the DOM tree and leaves that to the main parser, it only parses references to external resources like external scripts, style sheets and images.

Style sheets

Style sheets on the other hand have a different model. Conceptually it seems that since style sheets don't change the DOM tree, there is no reason to wait for them and stop the document parsing. There is an issue, though, of scripts asking for style information during the document parsing stage. If the style is not loaded and parsed yet, the script will get wrong answers and apparently this caused lots of problems. It seems to be an edge case but is quite common. Firefox blocks all scripts when there is a style sheet that is still being loaded and parsed. Webkit blocks scripts only when they try to access for certain style properties that may be effected by unloaded style sheets.

Render tree construction

While the DOM tree is being constructed, the browser constructs another tree, the render tree. This tree is of visual elements in the order in which they will be displayed. It is the visual representation of the document. The purpose of this tree is to enable painting the contents in their correct order.

Firefox calls the elements in the render tree "frames". Webkit uses the term renderer or render object.
A renderer knows how to layout and paint itself and it's children.
Webkits RenderObject class, the base class of the renderers has the following definition:

```
class RenderObject{
        virtual void layout();
        virtual void paint(PaintInfo);
        virtual void rect repaintRect();
        Node* node;  //the DOM node
        RenderStyle* style;  // the computed style
        RenderLayer* containgLayer; //the containing z-index layer
}
```

Each renderer represents a rectangular area usually corresponding to the node's CSS box, as described by the CSS2 spec. It contains geometric information like width, height and position.
The box type is affected by the "display" style attribute that is relevant for the node (see the style computation section). Here is Webkit code for deciding what type of renderer should be created for a DOM node, according to the display attribute.

```
RenderObject* RenderObject::createObject(Node* node, RenderStyle* style)
{
    Document* doc = node->document();
    RenderArena* arena = doc->renderArena();
    ...
    RenderObject* o = 0;

    switch (style->display()) {
        case NONE:
            break;
        case INLINE:
            o = new (arena) RenderInline(node);
            break;
        case BLOCK:
            o = new (arena) RenderBlock(node);
            break;
        case INLINE_BLOCK:
            o = new (arena) RenderBlock(node);
            break;
        case LIST_ITEM:
            o = new (arena) RenderListItem(node);
            break;
       ...
    }

    return o;
}
```

The element type is also considered, for example form controls and tables have special frames.
In Webkit if an element wants to create a special renderer it will override the "createRenderer" method.
The renderers points to style objects that contains the non geometric information.

The render tree relation to the DOM tree

The renderers correspond to the DOM elements, but the relation is not one to one. Non visual DOM elements will not be inserted in the render tree. An example is the "head" element. Also elements whose display attribute was assigned to "none" will not appear in the tree (elements with "hidden" visibility attribute will appear in the tree).

There are DOM elements which correspond to several visual objects. These are usually elements with complex structure that cannot be described by a single rectangle. For example, the "select" element has 3 renderers - one for the display area, one for the drop down list box and one for the button. Also when text is broken into multiple lines because the width is not sufficient for one line, the new lines will be added as extra renderers.
Another example of several renderers is broken HTML. According to CSS spec an inline element must contain either only block element or only inline elements. In case of mixed content, anonymous block renderers will be created to wrap the inline elements.

Some render objects correspond to a DOM node but not in the same place in the tree. Floats and absolutely positioned elements are out of flow, placed in a different place in the tree, and mapped to the real frame. A placeholder frame is where they should have been.

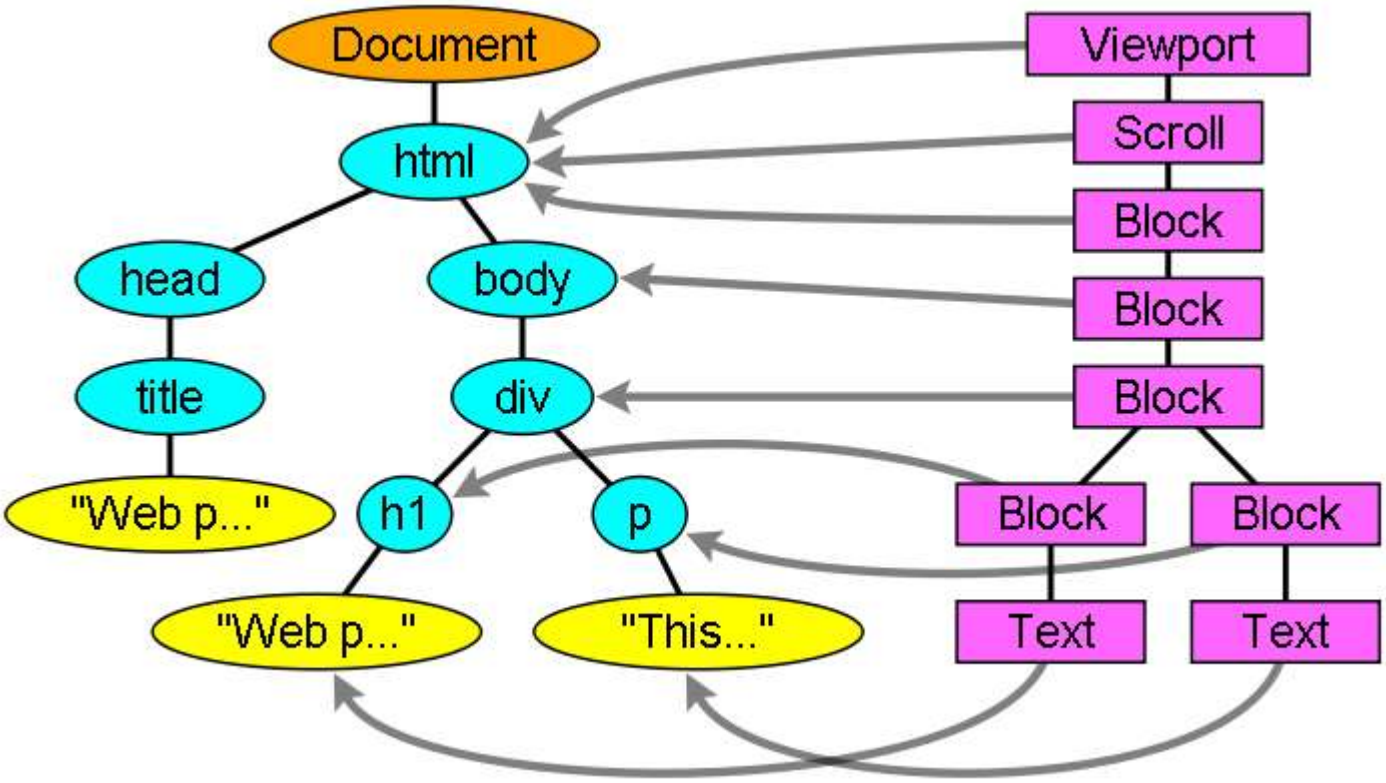


Figure 11: The render tree and the corresponding DOM tree(3.1). The "Viewport" is the initial containing block. In Webkit it will be the "RenderView" object.

The flow of constructing the tree

In Firefox, the presentation is registered as a listener for DOM updates. The presentation delegates frame creation to the "FrameConstructor" and the constructor resolves style(see style computation) and creates a frame.

In Webkit the process of resolving the style and creating a renderer is called "attachment". Every DOM node has an "attach" method. Attachment is synchronous, node insertion to the DOM tree calls the new node "attach" method.

Processing the html and body tags results in the construction of the render tree root. The root render object corresponds to what the CSS spec calls the containing block - the top most block that contains all other blocks. Its dimensions are the viewport - the browser window display area dimensions. Firefox calls it ViewPortFrame and Webkit calls it RenderView. This is the render object that the document point to. The rest of the tree is constructed as a DOM nodes insertion.
See CSS2 on this topic - http://www.w3.org/TR/CSS21/intro.html#processing-model

Style Computation

Building the render tree requires calculating the visual properties of each render object. This is done by calculating the style properties of each element.

The style includes style sheets of various origins, inline style elements and visual properties in the HTML (like the "bgcolor" property).The later is translated to matching CSS style properties.

The origins of style sheets are the browser's default style sheets, the style sheets provided by the page author and user style sheets - these are style sheets provides by the browser user (browsers let you define your favorite style. In Firefox, for instance, this is done by placing a style sheet in the "Firefox Profile" folder).

Style computation brings up a few difficulties:

1. Style data is a very large construct, holding the numerous style properties, this can cause memory problems.
2. Finding the matching rules for each element can cause performance issues if it's not optimized. Traversing the entire rule list for each element to find matches is a heavy task. Selectors can have complex structure that can cause the matching process to start on a seemingly promising path that is proven to be futile and another path has to be tried.
   For example - this compound selector:

   ```
   div div div div{
   ...
   }
   ```

   Means the rules apply to a " " who is the descendant of 3 divs.Suppose you want to check if the rule applies for a given " " element. You choose a certain path up the tree for checking. You may need to traverse the node tree up just to find out there are only two divs and the rule does not apply. You then need to try other paths in the tree.
3. Applying the rules involves quite complex cascade rules that define the hierarchy of the rules.

Let's see how the browsers face these issues:

Sharing style data

Webkit nodes references style objects (RenderStyle) These objects can be shared by nodes in some conditions. The nodes are siblings or cousins and:

1. The elements must be in the same mouse state (e.g., one can't be in :hover while the other isn't)
2. Neither element should have an id
3. The tag names should match
4. The class attributes should match
5. The set of mapped attributes must be identical
6. The link states must match
7. The focus states must match
8. Neither element should be affected by attribute selectors, where affected is defined as having any selector match that uses an attribute selector in any position within the selector at all
9. There must be no inline style attribute on the elements
10. There must be no sibling selectors in use at all. WebCore simply throws a global switch when any sibling selector is encountered and disables style sharing for the entire document when they are present. This includes the + selector and selectors like :first-child and :last-child.

Firefox rule tree

Firefox has two extra trees for easier style computation - the rule tree and style context tree. Webkit also has style objects but they are not stored in a tree like the style context tree, only the DOM node points to its relevant style.

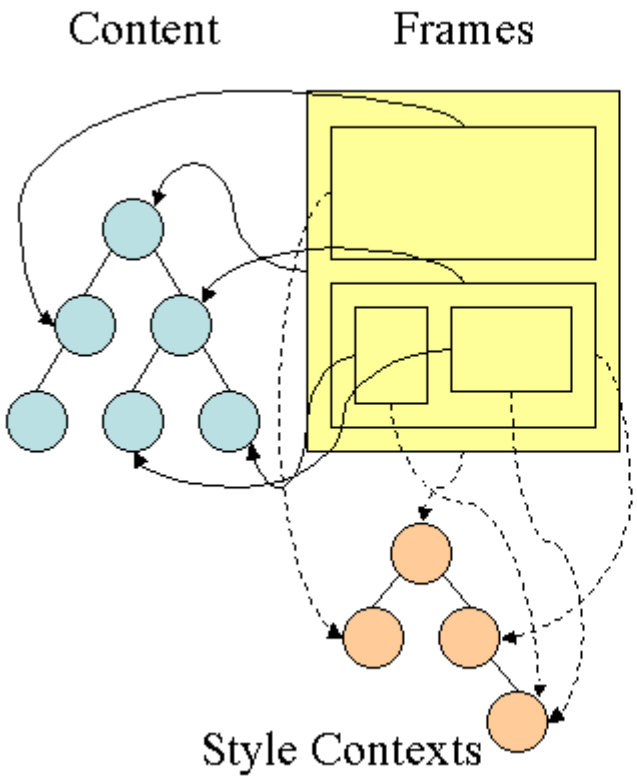
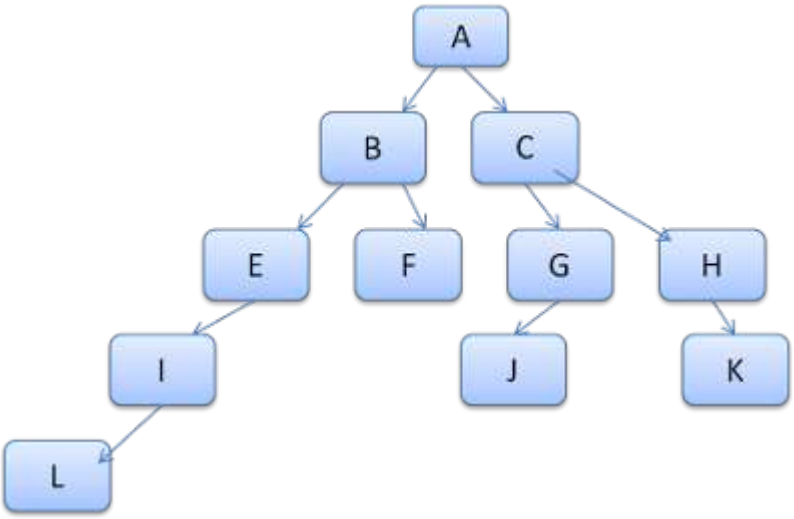


Figure 13: Firefox style context tree(2.2)

The style contexts contain end values. The values are computed by applying all the matching rules in the correct order and performing manipulations that transform them from logical to concrete values. For example - if the logical value is percentage of the screen it will be calculated and transformed to absolute units. The rule tree idea is really clever. It enables sharing these values between nodes to avoid computing them again. This also saves space.

All the matched rules are stored in a tree. The bottom nodes in a path have higher priority. The tree contains all the paths for rule matches that were found. Storing the rules is done lazily. The tree isn't calculated at the beginning for every node, but whenever a node style needs to be computed the computed paths are added to the tree.

The idea is to see the tree paths as words in a lexicon. Lets say we already computed this rule tree:



Suppose we need to match rules for another element in the content tree, and find out the matched rules (in the correct order) are B - E - I. We already have this path in the tree because we already computed path A - B - E - I - L. We will now have less work to do.
Let's see how the tree saves as work.

Division into structs

The style contexts are divided into structs. Those structs contain style information for a certain category like border or color. All the properties in a struct are either inherited or non inherited. Inherited properties are properties that unless defined by the element, are inherited from its parent. Non inherited properties (called "reset" properties) use default values if not defined.

The tree helps us by caching entire structs (containing the computed end values) in the tree. The idea is that if the bottom node didn't supply a definition for a struct, a cached struct in an upper node can be used.

Computing the style contexts using the rule tree

When computing the style context for a certain element, we first compute a path in the rule tree or use an existing one. We then begin to apply the rules in the path to fill the structs in our new style context. We start at the bottom node of the path - the one with the highest precedence (usually the most specific selector) and traverse the tree up until our struct is full. If there is no specification for the struct in that rule node, then we can greatly optimize - we go up the tree until we find a node that specifies it fully and simply point to it - that's the best optimization - the entire struct is shared. This saves computation of end values and memory.
If we find partial definitions we go up the tree until the struct is filled.

If we didn't find any definitions for our struct, then in case the struct is an "inherited" type - we point to the struct of our parent in the **context tree**, in this case we also succeeded in sharing structs. If its a reset struct then default values will be used.

If the most specific node does add values then we need to do some extra calculations for transforming it to actual values. We then cache the result in the tree node so it can be used by children.

In case an element has a sibling or a brother that points to the same tree node then the **entire style context** can be shared between them.

Lets see an example: Suppose we have this HTML

```
<html>
        <body>
                <div class="err" id="div1">
                        <p>
                          this is a <span class="big"> big error </span>
                          this is also a
                          <span class="big"> very  big  error</span> error
                        </p>
                </div>
                <div class="err" id="div2">another error</div>
        </body>
</html>
```

And the following rules:

```
1.      div {margin:5px;color:black}
2.      .err {color:red}
3.      .big {margin-top:3px}
4.      div span {margin-bottom:4px}
5.      #div1 {color:blue}
6.      #div 2 {color:green}
```

To simplify things let's say we need to fill out only two structs - the color struct and the margin struct. The color struct contains only one member - the color The margin struct contains the four sides.
The resulting rule tree will look like this (the nodes are marked with the node name : the # of rule they point at):

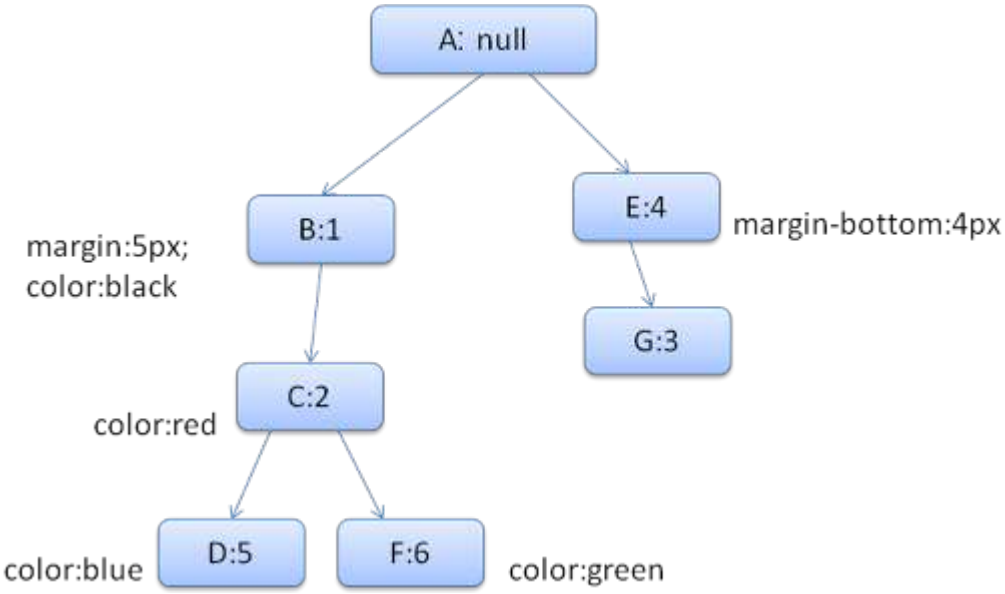


Figure 12: The rule tree

The context tree will look like this (node name : rule node they point to):

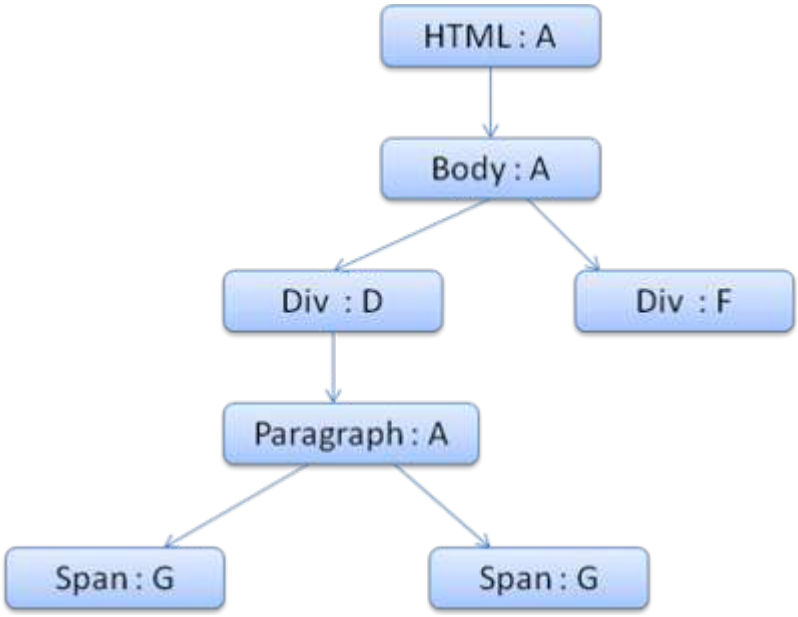


Figure 13: The context tree

Suppose we parse the HTML and get to the second tag. We need to create a style context for this node and fill its style structs.
We will match the rules and discover that the matching rules for the are 1 ,2 and 6. This means there is already an existing path in the tree that our element can use and we just need to add another node to it for rule 6 (node F in the rule tree).
We will create a style context and put it in the context tree. The new style context will point to node F in the rule tree.

We now need to fill the style structs. We will begin by filling out the margin struct. Since the last rule node(F) doesn't add to the margin struct, we can go up the tree until we find a cached struct computed in a previous node insertion and use it. We will find it on node B, which is the uppermost node that specified margin rules.

We do have a definition for the color struct, so we can't use a cached struct. Since color has one attribute we don't need to go up the tree to fill other attributes. We will compute the end value (convert string to RGB etc) and cache the computed struct on this node.

The work on the second element is even easier. We will match the rules and come to the conclusion that it points to rule G, like the previous span. Since we have siblings that point to the same node, we can share the entire style context and just point to the context of the previous span.

For structs that contain rules that are inherited from the parent, caching is done on the context tree (the color property is actually inherited, but Firefox treats it as reset and caches it on the rule tree).
For instance if we added rules for fonts in a paragraph:

```
p {font-family:Verdana;font size:10px;font-weight:bold}
```

Then the div element, which is a child of the paragraph in the context tree, could have shared the same font struct as his parent. This is if no font rules where specified for the "div".

In Webkit, who does not have a rule tree, the matched declarations are traversed 4 times. First non important high priority properties (properties that should be applied first because others depend on them - like display) are applied, than high priority important, then normal priority non important, then normal priority important rules. This means that properties that appear multiple times will be resolved according to the correct cascade order. The last wins.

So to summarize - sharing the style objects(entirely or some of the structs inside them) solves issues 1 and 3. Firefox rule tree also helps in applying the properties in the correct order.

Manipulating the rules for an easy match

There are several sources for style rules:

- CSS rules, either in external style sheets or in style elements.

  ```
  p {color:blue}
  ```

- Inline style attributes like

  ```
  <p style="color:blue" />
  ```

- HTML visual attributes (which are mapped to relevant style rules)

  ```
  <p bgcolor="blue" />
  ```

The last two are easily matched to the element since he owns the style attributes and HTML attributes can be mapped using the element as the key.

As noted previously in issue #2, the CSS rule matching can be trickier. To solve the difficulty, the rules are manipulated for easier access.

After parsing the style sheet, the rules are added one of several hash maps, according to the selector. There are maps by id, by class name, by tag name and a general map for anything that doesn't fit into those categories. If the selector is an id, the rule will be added to the id map, if it's a class it will be added to the class map etc.
This manipulation makes it much easier to match rules. There is no need to look in every declaration - we can extract the relevant rules for an element from the maps. This optimization eliminates 95+% of the rules, so that they need not even be considered during the matching process(4.1).

Let's see for example the following style rules:

```
p.error {color:red}
#messageDiv {height:50px}
```

```
div {margin:5px}
```

The first rule will be inserted into the class map. The second into the id map and the third into the tag map.
For the following HTML fragment;

```
<p class="error">an error occurred </p>
<div id=" messageDiv">this is a message</div>
```

We will first try to find rules for the p element. The class map will contain an "error" key under which the rule for "p.error" is found. The div element will have relevant rules in the id map (the key is the id) and the tag map. So the only work left is finding out which of the rules that were extracted by the keys really match.
For example if the rule for the div was

```
table div {margin:5px}
```

it will still be extracted from the tag map, because the key is the rightmost selector, but it would not match our div element, who does not have a table ancestor.

Both Webkit and Firefox do this manipulation.

Applying the rules in the correct cascade order

The style object has properties corresponding to every visual attribute (all css attributes but more generic). If the property is not defined by any of the matched rules - then some properties can be inherited by the parent element style object. Other properties have default values.

The problem begins when there is more than one definition - here comes the cascade order to solve the issue.

Style sheet cascade order

A declaration for a style property can appear in several style sheets, and several times inside a style sheet. This means the order of applying the rules is very important. This is called the "cascade" order. According to CSS2 spec, the cascade order is (from low to high):

1. Browser declarations
2. User normal declarations
3. Author normal declarations
4. Author important declarations
5. User important declarations

The browser declarations are least important and the user overrides the author only if the declaration was marked as important. Declarations with the same order will be sorted by specifity and then the order they are specified. The HTML visual attributes are translated to matching CSS declarations . They are treated as author rules with low priority.

Specifity

The selector specifity is defined by the CSS2 specification as follows:

- count 1 if the declaration is from is a 'style' attribute rather than a rule with a selector, 0 otherwise (= a)

- count the number of ID attributes in the selector (= b)
- count the number of other attributes and pseudo-classes in the selector (= c)
- count the number of element names and pseudo-elements in the selector (= d)

Concatenating the four numbers a-b-c-d (in a number system with a large base) gives the specificity.

The number base you need to use is defined by the highest count you have in one of the categories. For example, if a=14 you can use hexadecimal base. In the unlikely case where a=17 you will need a 17 digits number base. The later situation can happen with a selector like this: html body div div p ... (17 tags in your selector.. not very likely).

Some examples:

```
*             {}  /* a=0 b=0 c=0 d=0 -> specificity = 0,0,0,0 */
li            {}  /* a=0 b=0 c=0 d=1 -> specificity = 0,0,0,1 */
li:first-line {}  /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
ul li         {}  /* a=0 b=0 c=0 d=2 -> specificity = 0,0,0,2 */
ul ol+li      {}  /* a=0 b=0 c=0 d=3 -> specificity = 0,0,0,3 */
h1 + *[rel=up]{}  /* a=0 b=0 c=1 d=1 -> specificity = 0,0,1,1 */
ul ol li.red  {}  /* a=0 b=0 c=1 d=3 -> specificity = 0,0,1,3 */
li.red.level  {}  /* a=0 b=0 c=2 d=1 -> specificity = 0,0,2,1 */
#x34y         {}  /* a=0 b=1 c=0 d=0 -> specificity = 0,1,0,0 */
style=""          /* a=1 b=0 c=0 d=0 -> specificity = 1,0,0,0 */
```

Sorting the rules

After the rules are matched, they are sorted according to the cascade rules. Webkit uses bubble sort for small lists and merge sort for big ones. Webkit implements sorting by overriding the ">" operator for the rules:

```
static bool operator >(CSSRuleData& r1, CSSRuleData& r2)
{
    int spec1 = r1.selector()->specificity();
    int spec2 = r2.selector()->specificity();
    return (spec1 == spec2) : r1.position() > r2.position() : spec1 > spec2;
}
```

Gradual process

Webkit uses a flag that marks if all top level style sheets (including @imports) have been loaded. If the style is not fully loaded when attaching - place holders are used and it s marked in the document, and they will be recalculated once the style sheets were loaded.

Layout

When the renderer is created and added to the tree, it does not have a position and size. Calculating these values is called layout or reflow.

HTML uses a flow based layout model, meaning that most of the time it is possible to compute the geometry in a single pass. Elements later ``in the flow'' typically do not affect the geometry of elements that are earlier ``in the flow'', so layout can proceed left-to-right, top-to-bottom through the document. There are exceptions - for example, HTML tables may require more than one pass (3.5).

The coordinate system is relative to the root frame. Top and left coordinates are used.

Layout is a recursive process. It begins at the root renderer, which corresponds to the element of the

HTML document. Layout continues recursively through some or all of the frame hierarchy, computing geometric information for each renderer that requires it.

The position of the root renderer is 0,0 and its dimensions is the viewport - the visible part of the browser window.

All renderers have a "layout" or "reflow" method, each renderer invokes the layout method of its children that need layout.

Dirty bit system

In order not to do a full layout for every small change, browser use a "dirty bit" system. A renderer that is changed or added marks itself and its children as "dirty" - needing layout.

There are two flags - "dirty" and "children are dirty". Children are dirty means that although the renderer itself may be ok, it has at least one child that needs a layout.

Global and incremental layout

Layout can be triggered on the entire render tree - this is "global" layout. This can happen as a result of:

1. A global style change that affects all renderers, like a font size change.
2. As a result of a screen being resized

Layout can be incremental, only the dirty renderers will be layed out (this can cause some damage which will require extra layouts).
Incremental layout is triggered (asynchronously) when renderers are dirty. For example when new renderers are appended to the render tree after extra content came from the network and was added to the DOM tree.

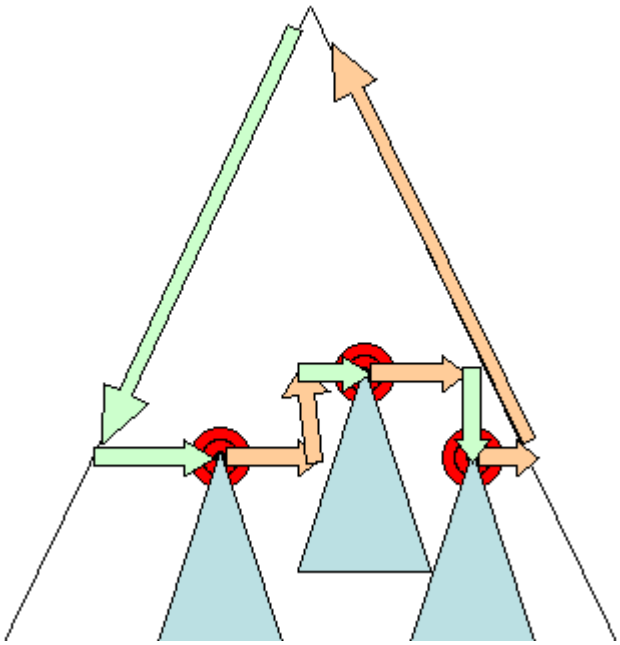


Figure 20:Incremental layout - only dirty renderers and their children are layed out(3.6).

Asynchronous and Synchronous layout

Incremental layout is done asynchronously. Firefox queues "reflow commands" for incremental layouts

and a scheduler triggers batch execution of these commands. Webkit also has a timer that executes an incremental layout - the tree is traversed and "dirty" renderers are layout out.
Scripts asking for style information, like "offsightHeight" can trigger incremental layout synchronously.
Global layout will usually be triggered synchronously.
Sometimes layout is triggered as a callback after an initial layout because some attributes , like the scrolling position changed.

Optimizations

When a layout is triggered by a "resize" or a change in the renderer position(and not size), the renders sizes are taken from a cache and not recalculated..
In some cases - only a sub tree is modified and layout does not start from the root. This can happen in cases where the change is local and does not affect its surroundings - like text inserted into text fields (otherwise every keystroke would have triggered a layout starting from the root).

The layout process

The layout usually has the following pattern:

1. Parent renderer determines its own width.
2. Parent goes over children and:
    1. Place the child renderer (sets its x and y).
    2. Calls child layout if needed(they are dirty or we are in a global layout or some other reason) - this calculates the child's height.
3. Parent uses children accumulative heights and the heights of the margins and paddings to set it own height - this will be used by the parent renderer's parent.
4. Sets its dirty bit to false.

Firefox uses a "state" object(nsHTMLReflowState) as a parameter to layout (termed "reflow"). Among others the state includes the parents width.
The output of Firefox layout is a "metrics" object(nsHTMLReflowMetrics). It will contain the renderer computed height.

Width calculation

The renderer's width is calculated using the container block's width , the renderer's style "width" property, the margins and borders.
For example the width of the following div:

```
<div style="width:30%"/>
```

Would be calculated by Webkit as following(class RenderBox method calcWidth):

- The container width is the maximum of the containers availableWidth and 0. The availableWidth in this case is the contentWidth which is calculated as:

  ```
  clientWidth() - paddingLeft() - paddingRight()
  ```

  clientWidth and clientHeight represent the interior of an object excluding border and scrollbar.
- The elements width is the "width" style attribute. It will be calculated as an absolute value by computing the percentage of the container width.
- The horizontal borders and paddings are now added.

So far this was the calculation of the "preferred width". Now the minimum and maximum widths will be

calculated.
If the preferred width is higher then the maximum width - the maximum width is used. If it is lower then the minimum width (the smallest unbreakable unit) hen the minimum width is used.

The values are cached, in case a layout is needed but the width does not change.

Line Breaking

When a renderer in the middle of layout decides it needs to break. It stops and propagates to its parent it needs to be broken. The parent will create the extra renderers and calls layout on them.

Painting

In the painting stage, the render tree is traversed and the renderers "paint" method is called to display their content on the screen. Painting uses the UI infrastructure component. More on that in the chapter about the UI.

Global and Incremental

Like layout, painting can also be global - the entire tree is painted - or incremental. In incremental painting, some of the renderers change in a way that does not affect the entire tree. The changed renderer invalidates it's rectangle on the screen. This causes the OS to see it as a "dirty region" and generate a "paint" event. The OS does it cleverly and coalesces several regions into one. In Chrome it is more complicated because the renderer is in a different process then the main process. Chrome simulates the OS behavior to some extent. The presentation listens to these events and delegates the message to the render root. The tree is traversed until the relevant renderer is reached. It will repaint itself (and usually its children).

The painting order

CSS2 defines the order of the painting process - http://www.w3.org/TR/CSS21/zindex.html. This is actually the order in which the elements are stacked in the stacking contexts. This order affects painting since the stacks are painted from back to front. The stacking order of a block renderer is:

1. background color
2. background image
3. border
4. children
5. outline

Firefox display list

Firefox goes over the render tree and builds a display list for the painted rectangular. It contains the renderers relevant for the rectangular, in the right painting order (backgrounds of the renderers, then borders etc).
That way the tree needs to be traversed only once for a repaint instead of several times - painting all backgrounds, then all images , then all borders etc.
Firefox optimizes the process by not adding elements that will be hidden, like elements completely beneath other opaque elements.

Webkit rectangle storage

Before repainting, webkit saves the old rectangle as a bitmap. It then paints only the delta between the new and old rectangles.

Dynamic changes

The browsers try to do the minimal possible actions in response to a change. So changes to an elements color will cause only repaint of the element. Changes to the element position will cause layout and repaint of the element, its children and possibly siblings. Adding a DOM node will cause layout and repaint of the node. Major changes, like increasing font size of the "html" element, will cause invalidation of caches, relyout and repaint of the entire tree.

The rendering engine's threads

The rendering engine is single threaded. Almost everything, except network operations, happens in a single thread. In Firefox and safari this is the main thread of the browser. In chrome it's the tab process main thread.
Network operations can be performed by several parallel threads. The number of parallel connections is limited (usually 2 - 6 connections. Firefox 3, for example, uses 6).

Event loop

The browser main thread is an event loop. Its an infinite loop that keeps the process alive. It waits for events (like layout and paint events) and processes them. This is Firefox code for the main event loop:

```
while (!mExiting)
    NS_ProcessNextEvent(thread);
```

CSS2 visual model

The canvas

According to CCS2 specification, the term canvas describes "the space where the formatting structure is rendered." - where the browser paints the content.
The canvas is infinite for each dimension of the space but browsers choose an initial width based on the dimensions of the viewport.

According to http://www.w3.org/TR/CSS2/zindex.html, the canvas is transparent if contained within another, and given a browser defined color if it is not.

CSS Box model

The CSS box model describes the rectangular boxes that are generated for elements in the document tree and laid out according to the visual formatting model.
Each box has a content area (e.g., text, an image, etc.) and optional surrounding padding, border, and margin areas.

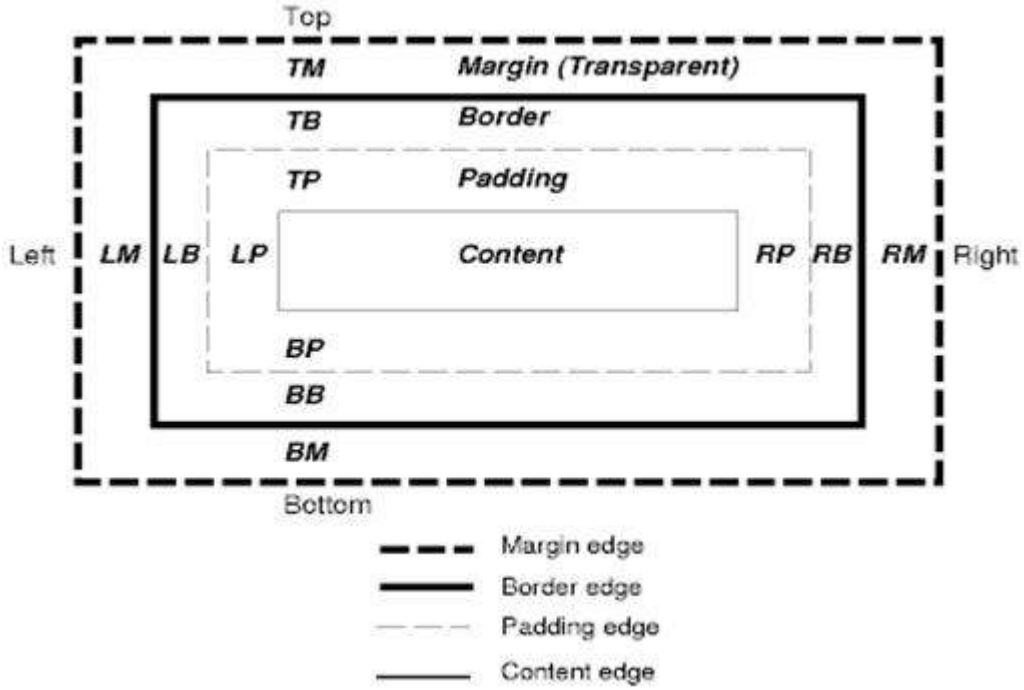


Figure 14:CSS2 box model

Each node generates 0..n such boxes.
All elements have a "display" property that determines their type of box that will be generated.
Examples:

```
block  - generates a block box.
inline - generates one or more inline boxes.
none - no box is generated.
```

The default is inline but the browser style sheet set other defaults. For example - the default display for "div" element is block.
You can find a default style sheet example here http://www.w3.org/TR/CSS2/sample.html

Positioning scheme

There are three schemes:

1. Normal - the object is positioned according to its place in the document - this means its place in the render tree is like its place in the dom tree and layed out according to its box type and dimensions
2. Float - the object is first layed out like normal flow, then moved as far left or right as possible
3. Absolute - the object is put in the render tree differently than its place in the DOM tree

The positioning scheme is set by the "position" property and the "float" attribute.

- static and relative cause a normal flow
- absolute and fixed cause an absolute positioning

In static positioning no position is defined and the default positioning is used. In the other schemes, the author specifies the position - top,bottom,left,right.

The way the box is layed out is determined by:

- Box type

- Box dimensions
- Positioning scheme
- External information - like images size and the size of the screen

Box types

Block box: forms a block - have their own rectangle on the browser window.

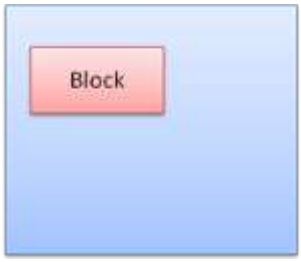


Figure 15:Block box

Inline box: does not have its own block, but is inside a containing block.

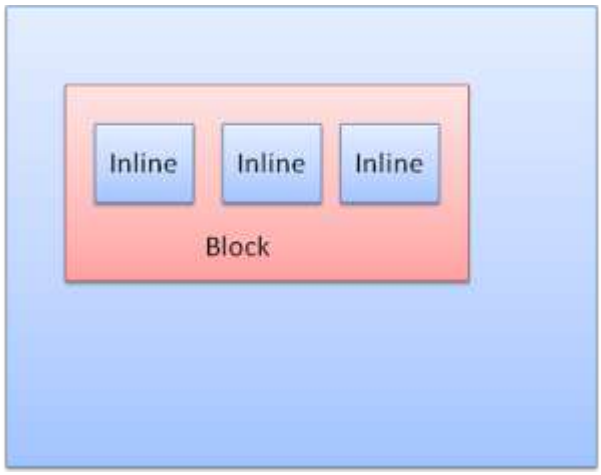


Figure 15:Inine boxes

Blocks are formatted vertically one after the other. Inlines are formatted horizontally.

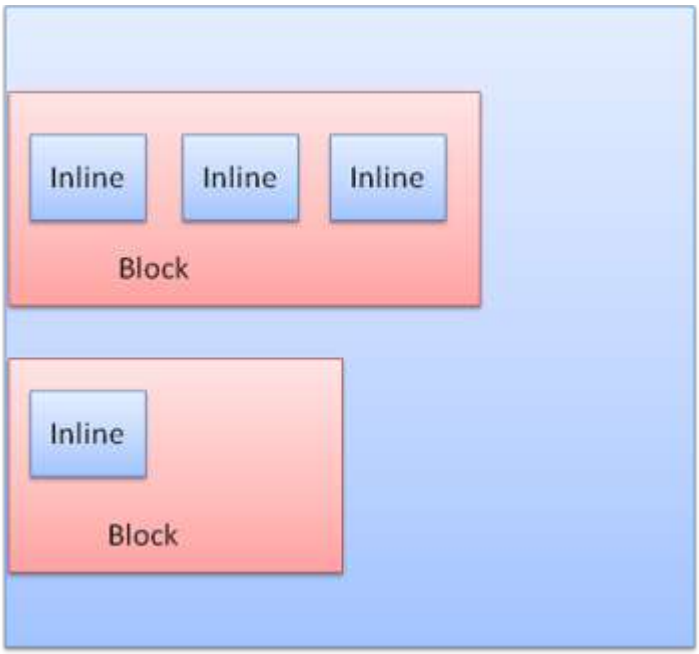


Figure 16:Block and Inline formatting

Inline boxes are put inside lines or "line boxes". The lines are at least as tall as the tallest box but can be taller, when the boxes are aligned "baseline" - meaning the bottom part of an element is aligned at a point of another box other then the bottom. In case the container width is not enough, the inlines will be put in several lines. This is usually what happens in a paragraph.

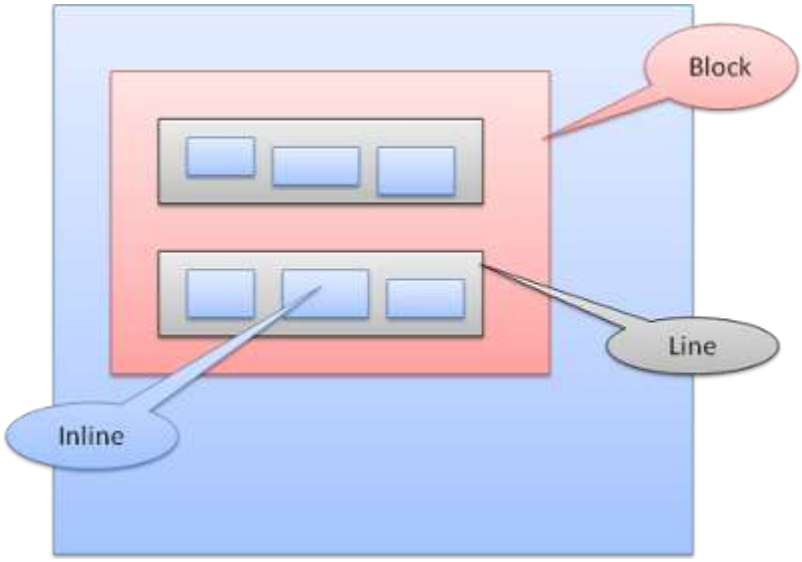


Figure 17:Lines

Positioning

Relative

Relative positioning - positioned like usual and then moved by the required delta.

```
<html>
        <div>
                <span>1</span>
                <span>2</span>
                <span style="position:relative;left:5px">3</span>
        </div>
</html>
```

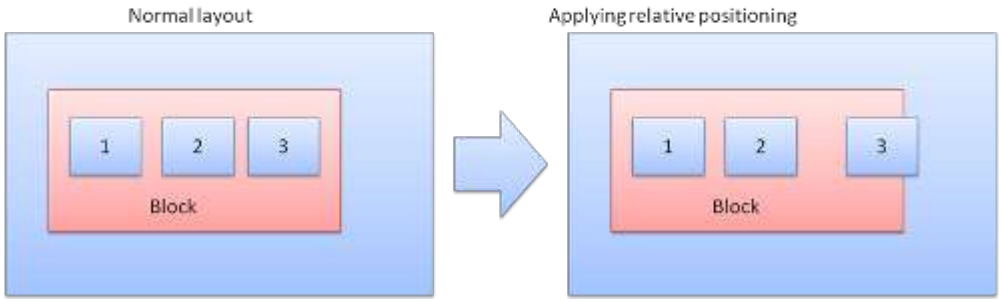


Figure 18:Relative positioning

Floats

A float box is shifted to the left or right of a line. The interesting feature is that the other boxes flow around it The HTML:

```
<p>
<img style="float:right" src="images/image.gif" width="100" height="100">Lorem ipsum
dolor sit amet, consectetuer...
</p>
```

Will look like:

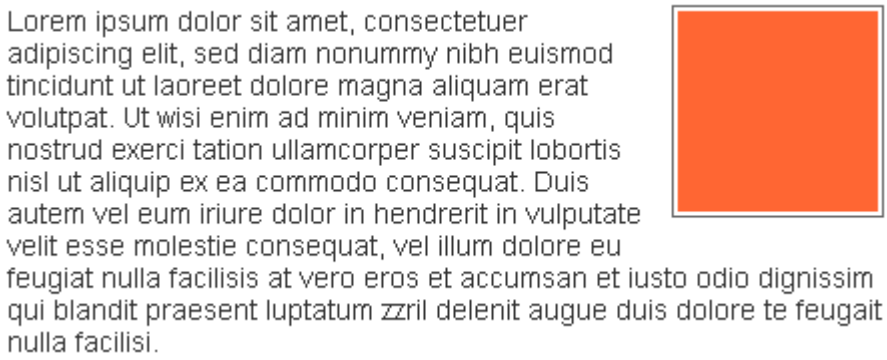


Figure 19:Float

Absolute and fixed

The layout is defined exactly regardless of the normal flow. The element does not participate in the normal flow. The dimensions are relative to the container. In fixed - the container is the view port.

```
<html>
        <div>
                <span>1</span>
                <span>2</span>
                <span style="position:fixed;top:5px,left:5px">3</span>
        </div>
</html>
```

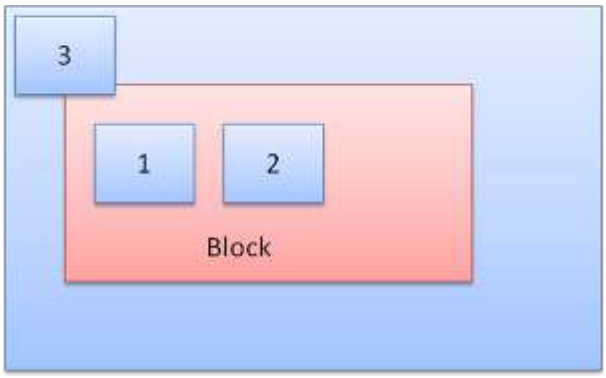


Figure 20:Fixed positioning

Note - the fixed box will not move even when the document is scrolled!

Layered representation

It is specified by the z-index CSS property. It represents the 3rd dimension of the box, its position along the "z axis".

The boxes are divided to stacks (called stacking contexts). In each stack the back elements will be painted first and the forward elements on top, closer to the user. In case of overlap the will hide the former element.
The stacks are ordered according to the z-index property. Boxes with "z-index" property form a local stack. The viewport has the outer stack.
Example:

```
<STYLE type="text/css">
      div {
        position: absolute;
        left: 2in;
        top: 2in;
       }
    </STYLE>

  <P>
            <DIV
                 style="z-index: 3;background-color:red; width: 1in; height: 1in; ">
            </DIV>
            <DIV
                 style="z-index: 1;background-color:green;width: 2in; height: 2in;">
            </DIV>
   </p>
```

The result will be this:

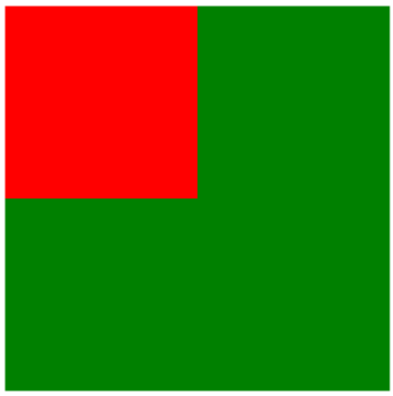


Figure 20:Fixed positioning

Although the green div comes before the red one, and would have been painted before in the regular flow, the z-index property is higher, so it is more forward in the stack held by the root box.

Resources

1. Browser architecture
    1. Grosskurth, Alan. A Reference Architecture for Web Browsers. http://grosskurth.ca/papers/browser-refarch.pdf.
2. Parsing
    1. Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools (aka the "Dragon book"), Addison-Wesley, 1986
    2. Rick Jelliffe. The Bold and the Beautiful: two new drafts for HTML 5. http://broadcast.oreilly.com/2009/05/the-bold-and-the-beautiful-two.html.
3. Firefox
    1. L. David Baron, Faster HTML and CSS: Layout Engine Internals for Web Developers. http://dbaron.org/talks/2008-11-12-faster-html-and-css/slide-6.xhtml.
    2. L. David Baron, Faster HTML and CSS: Layout Engine Internals for Web Developers(Google tech talk video). http://www.youtube.com/watch?v=a2_6bGNZ7bA.
    3. L. David Baron, Mozilla's Layout Engine. http://www.mozilla.org/newlayout/doc/layout-2006-07-12/slide-6.xhtml.
    4. L. David Baron, Mozilla Style System Documentation. http://www.mozilla.org/newlayout/doc/style-system.html.
    5. Chris Waterson, Notes on HTML Reflow. http://www.mozilla.org/newlayout/doc/reflow.html.
    6. Chris Waterson, Gecko Overview. http://www.mozilla.org/newlayout/doc/gecko-overview.htm.

7. Alexander Larsson, The life of an HTML HTTP request. https://developer.mozilla.org/en/The_life_of_an_HTML_HTTP_request.
4. Webkit
    1. David Hyatt, Implementing CSS(part 1). http://weblogs.mozillazine.org/hyatt/archives/cat_safari.html.
    2. David Hyatt, An Overview of WebCore. http://weblogs.mozillazine.org/hyatt/WebCore/chapter2.html.
    3. David Hyatt, WebCore Rendering. http://webkit.org/blog/114/.
    4. David Hyatt, The FOUC Problem. http://webkit.org/blog/66/the-fouc-problem/.
5. W3C Specifications
    1. HTML 4.01 Specification. http://www.w3.org/TR/html4/.
    2. HTML5 Specification. http://dev.w3.org/html5/spec/Overview.html.
    3. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. http://www.w3.org/TR/CSS2/.
6. Browsers build instructions
    1. Firefox. https://developer.mozilla.org/en/Build_Documentation
    2. Webkit. http://webkit.org/building/build.html