浏览器内部工作原理

目录

英文原文：How Browsers Work: Behind the Scenes of Modern Web Browsers

一、介绍

浏览器可以被认为是使用最广泛的软件，本文将介绍浏览器的工作原理，我们将看到，从你在地址栏输入google.com到你看到google主页过程中都发生了什么。

将讨论的浏览器

今天，有五种主流浏览器——IE、Firefox、Safari、Chrome及Opera。

本文将基于一些开源浏览器的例子——Firefox、Chrome及Safari，Safari是部分开源的。

根据W3C（World Wide Web Consortium万维网联盟）的浏览器统计数据，当前（2011年5月），Firefox、Safari及Chrome的市场占有率综合已接近60%。（原文为2009年10月，数据没有太大变化）因此，可以说开源浏览器已经占据了浏览器市场的半壁江山。

浏览器的主要功能

浏览器的主要功能是将用户选择的web资源呈现出来，它需要从服务器请求资源，并将其显示在浏览器窗口中，资源的格式通常是HTML，也包括PDF、image及其他格式。用户用URI（Uniform Resource Identifier统一资源标识符）来指定所请求资源的位置，在网络一章有更多讨论。

HTML和CSS规范中规定了浏览器解释html文档的方式，由W3C组织对这些规范进行维护，W3C是负责制定web标准的组织。

HTML规范的最新版本是HTML4(http://www.w3.org/TR/html401/)，HTML5还在制定中（译注：两年前），最新的CSS规范版本是2（http://www.w3.org/TR/CSS2），CSS3也还正在制定中（译注：同样两年前）。

这些年来，浏览器厂商纷纷开发自己的扩展，对规范的遵循并不完善，这为web开发者带来了严重的兼容性问题。

但是，浏览器的用户界面则差不多，常见的用户界面元素包括：

- 用来输入URI的地址栏
- 前进、后退按钮
- 书签选项
- 用于刷新及暂停当前加载文档的刷新、暂停按钮
- 用于到达主页的主页按钮

奇怪的是，并没有哪个正式公布的规范对用户界面做出规定，这些是多年来各浏览器厂商之间相互模仿和不断改进的结果。

HTML5并没有规定浏览器必须具有的UI元素，但列出了一些常用元素，包括地址栏、状态栏及工具栏。还有一些浏览器有自己专有的功能，比如Firefox的下载管理。更多相关内容将在后面讨论用户界面时介绍。

浏览器的主要构成（High Level Structure）

浏览器的主要组件包括：

1. 用户界面 – 包括地址栏、后退/前进按钮、书签目录等，也就是你所看到的除了用来显示你所请求页面的主窗口之外的其他部分。

2. 浏览器引擎 – 用来查询及操作渲染引擎的接口。

3. 渲染引擎 – 用来显示请求的内容，例如，如果请求内容为html，它负责解析html及css，并将解析后的结果显示出来。

4. 网络 – 用来完成网络调用，例如http请求，它具有平台无关的接口，可以在不同平台上工作。

5. UI后端 – 用来绘制类似组合选择框及对话框等基本组件，具有不特定于某个平台的通用接口，底层使用操作系统的用户接口。

6. JS解释器 – 用来解释执行JS代码。

7. 数据存储 – 属于持久层，浏览器需要在硬盘中保存类似cookie的各种数据，HTML5定义了web database技术，这是一种轻量级完整的客户端存储技术

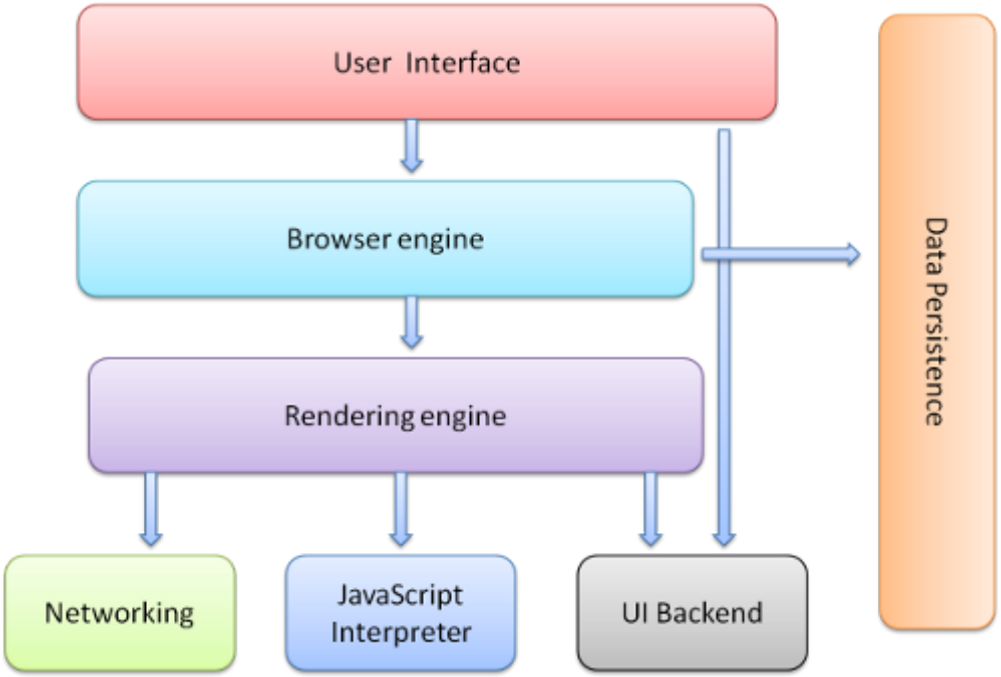


图1：浏览器主要组件

需要注意的是，不同于大部分浏览器，Chrome为每个Tab分配了各自的渲染引擎实例，每个Tab就是一个独立的进程。

对于构成浏览器的这些组件，后面会逐一详细讨论。

二、渲染引擎（The rendering engine）

渲染引擎的职责就是渲染，即在浏览器窗口中显示所请求的内容。

默认情况下，渲染引擎可以显示html、xml文档及图片，它也可以借助插件（一种浏览器扩展）显示其他类型数据，例如使用PDF阅读器插件，可以显示PDF格式，将由专门一章讲解插件及扩展，这里只讨论渲染引擎最主要的用途——显示应用了CSS之后的html及图片。

渲染引擎简介

本文所讨论的浏览器——Firefox、Chrome和Safari是基于两种渲染引擎构建的，Firefox使用Geoko——Mozilla自主研发的渲染引擎，Safari和Chrome都使用webkit。

Webkit是一款开源渲染引擎，它本来是为Linux平台研发的，后来由Apple移植到Mac及Windows上，相关内容请参考http://webkit.org。

渲染主流程（The main flow）

渲染引擎首先通过网络获得所请求文档的内容，通常以8K分块的方式完成。

下面是渲染引擎在取得内容之后的基本流程：

解析html以构建dom树 -> 构建render树 -> 布局render树 -> 绘制render树

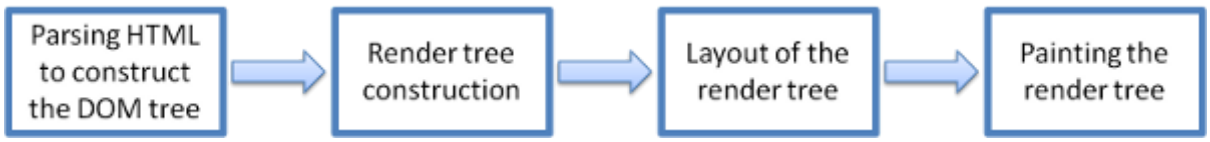


图2：渲染引擎基本流程

渲染引擎开始解析html，并将标签转化为内容树中的dom节点。接着，它解析外部CSS文件及style标

签中的样式信息。这些样式信息以及html中的可见性指令将被用来构建另一棵树——render树。

Render树由一些包含有颜色和大小等属性的矩形组成，它们将被按照正确的顺序显示到屏幕上。

Render树构建好了之后，将会执行布局过程，它将确定每个节点在屏幕上的确切坐标。再下一步就是绘制，即遍历render树，并使用UI后端层绘制每个节点。

值得注意的是，这个过程是逐步完成的，为了更好的用户体验，渲染引擎将会尽可能早的将内容呈现到屏幕上，并不会等到所有的html都解析完成之后再去构建和布局render树。它是解析完一部分内容就显示一部分内容，同时，可能还在通过网络下载其余内容。

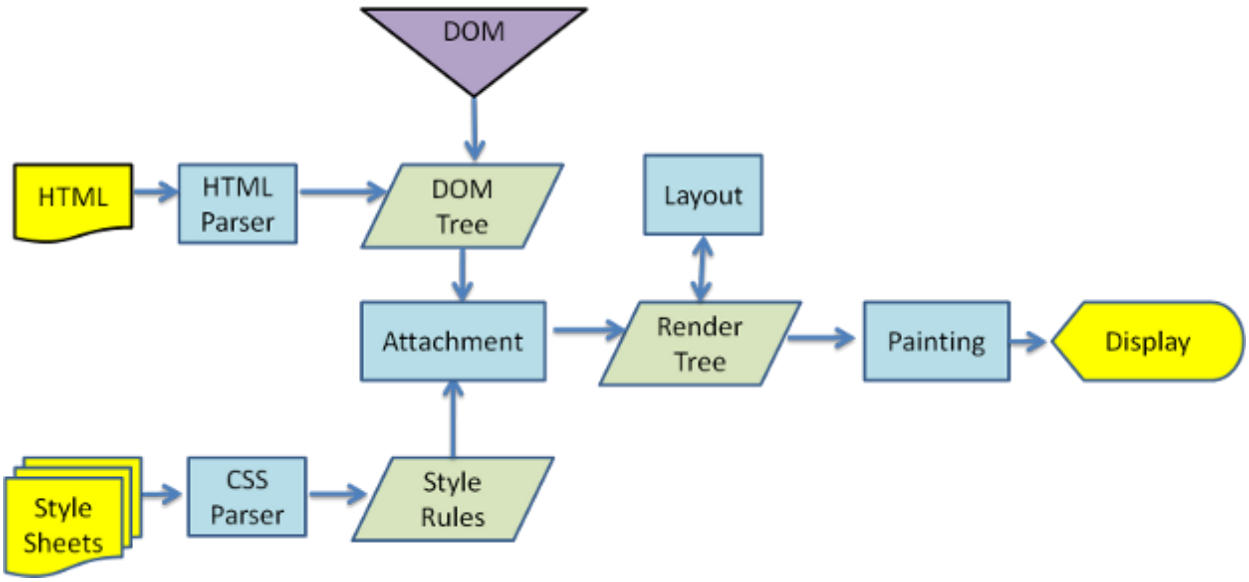


图3：webkit主流程

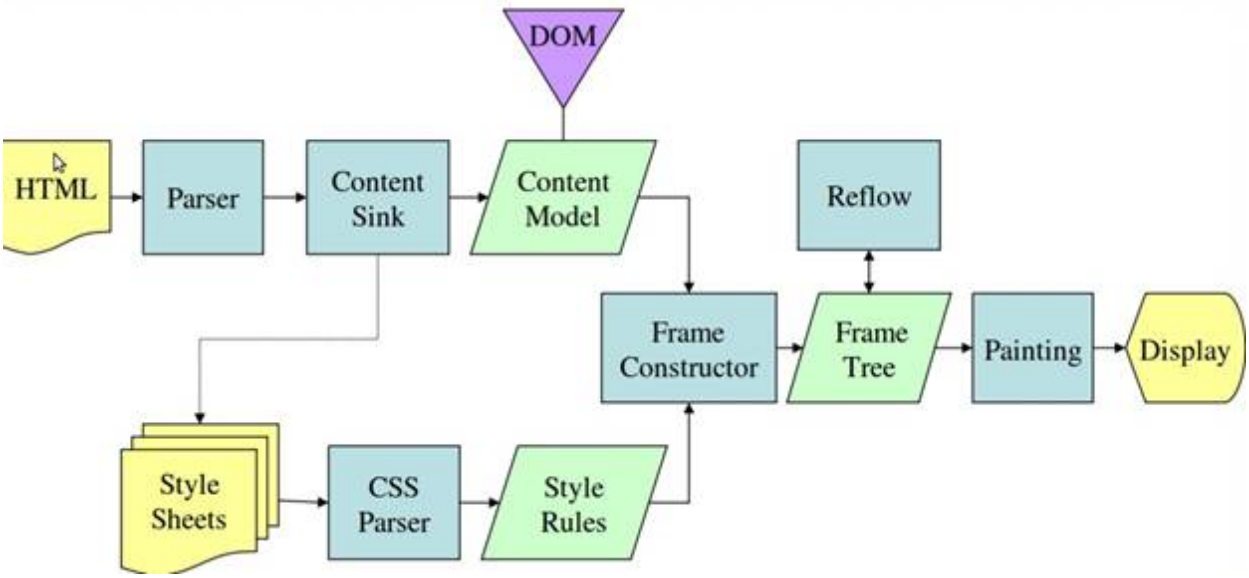


图4：Mozilla的Geoko渲染引擎主流程

从图3和4中可以看出，尽管webkit和Gecko使用的术语稍有不同，他们的主要流程基本相同。Gecko称可见的格式化元素组成的树为frame树，每个元素都是一个frame，webkit则使用render树这个名词来命名由渲染对象组成的树。Webkit中元素的定位称为布局，而Gecko中称为回流。Webkit称利用dom节点及样式信息去构建render树的过程为attachment，Gecko在html和dom树之间附加了一层，这层称为内容接收器，相当制造dom元素的工厂。下面将讨论流程中的各个阶段。

三、解析与DOM树构建（Parsing and DOM tree construction）

解析（Parsing－general）

既然解析是渲染引擎中一个非常重要的过程，我们将稍微深入的研究它。首先简要介绍一下解析。

解析一个文档即将其转换为具有一定意义的结构——编码可以理解和使用的东西。解析的结果通常是表达文档结构的节点树，称为解析树或语法树。

例如，解析"2＋3－1"这个表达式，可能返回这样一棵树。

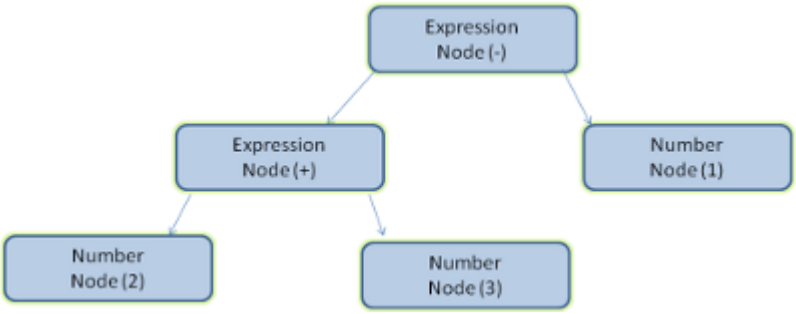


图5：数学表达式树节点

文法（Grammars）

解析基于文档依据的语法规则——文档的语言或格式。每种可被解析的格式必须具有由词汇及语法规则组成的特定的文法，称为上下文无关文法。人类语言不具有这一特性，因此不能被一般的解析技术所解析。

解析器－词法分析器（Parser－Lexer combination）

解析可以分为两个子过程——语法分析及词法分析

词法分析就是将输入分解为符号，符号是语言的词汇表——基本有效单元的集合。对于人类语言来说，它相当于我们字典中出现的所有单词。

语法分析指对语言应用语法规则。

解析器一般将工作分配给两个组件——词法分析器（有时也叫分词器）负责将输入分解为合法的符号，解析器则根据语言的语法规则分析文档结构，从而构建解析树，词法分析器知道怎么跳过空白和换行之类的无关字符。

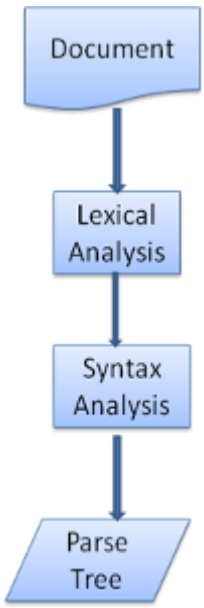


图6：从源文档到解析树

解析过程是迭代的，解析器从词法分析器处取到一个新的符号，并试着用这个符号匹配一条语法规则，如果匹配了一条规则，这个符号对应的节点将被添加到解析树上，然后解析器请求另一个符号。如果没有匹配到规则，解析器将在内部保存该符号，并从词法分析器取下一个符号，直到所有内部保存的符号能够匹配一项语法规则。如果最终没有找到匹配的规则，解析器将抛出一个异常，这意味着文档无效或是包含语法错误。

转换（Translation）

很多时候，解析树并不是最终结果。解析一般在转换中使用——将输入文档转换为另一种格式。编译就是个例子，编译器在将一段源码编译为机器码的时候，先将源码解析为解析树，然后将该树转换为一个机器码文档。

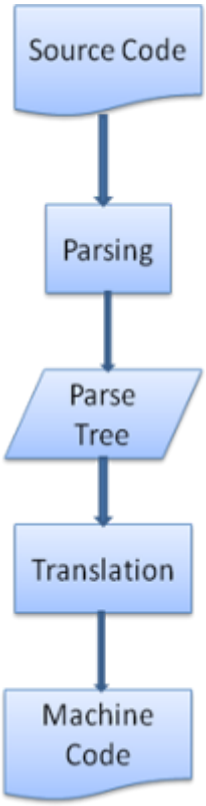


图7：编译流程

解析实例Parsing example

图5中，我们从一个数学表达式构建了一个解析树，这里定义一个简单的数学语言来看下解析过程。

词汇表：我们的语言包括整数、加号及减号。

语法：

1. 该语言的语法基本单元包括表达式、term及操作符
2. 该语言可以包括多个表达式
3. 一个表达式定义为两个term通过一个操作符连接
4. 操作符可以是加号或减号
5. term可以是一个整数或一个表达式

现在来分析一下“2＋3－1”这个输入

第一个匹配规则的子字符串是“2”，根据规则5，它是一个term，第二个匹配的是“2+3”，它符合第2条规则——一个操作符连接两个term，下一次匹配发生在输入的结束处。“2+3-1”是一个表达式，因为我们已经知道“2+3”是一个term，所以我们有了一个term紧跟着一个操作符及另一个term。“2++”将不会匹配任何规则，因此是一个无效输入。

词汇表及语法的定义

词汇表通常利用正则表达式来定义。

例如上面的语言可以定义为：

INTEGER：0 | [1－9] [0－9] *

PLUS：+

MINUS：－

正如看到的，这里用正则表达式定义整数。

语法通常用BNF格式定义，我们的语言可以定义为：

expression := term operation term

operation := PLUS | MINUS

term := INTEGER | expression

如果一个语言的文法是上下文无关的，则它可以用正则解析器来解析。对上下文无关文法的一个直观的定义是，该文法可以用BNF来完整的表达。可查看http://en.wikipedia.org/wiki/Context-free_grammar。

解析器类型（Types of parsers）

有两种基本的解析器——自顶向下解析及自底向上解析。比较直观的解释是，自顶向下解析，查看语法的最高层结构并试着匹配其中一个；自底向上解析则从输入开始，逐步将其转换为语法规则，从底层规则开始直到匹配高层规则。

来看一下这两种解析器如何解析上面的例子：

自顶向下解析器从最高层规则开始——它先识别出“2+3”，将其视为一个表达式，然后识别出”2+3-1“为一个表达式（识别表达式的过程中匹配了其他规则，但出发点是最高层规则）。

自底向上解析会扫描输入直到匹配了一条规则，然后用该规则取代匹配的输入，直到解析完所有输入。部分匹配的表达式被放置在解析堆栈中。

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

expression	- 1
expression operation	1
expression	

自底向上解析器称为shift reduce解析器，因为输入向右移动（想象一个指针首先指向输入开始处，并向右移动），并逐渐简化为语法规则。

自动化解析（Generating parsers automatically）

解析器生成器这个工具可以自动生成解析器，只需要指定语言的文法——词汇表及语法规则，它就可以生成一个解析器。创建一个解析器需要对解析有深入的理解，而且手动的创建一个由较好性能的解析器并不容易，所以解析生成器很有用。Webkit使用两个知名的解析生成器——用于创建语法分析器的Flex及创建解析器的Bison（你可能接触过Lex和Yacc）。Flex的输入是一个包含了符号定义的正则表达式，Bison的输入是用BNF格式表示的语法规则。

HTML解析器（HTML Parser）

HTML解析器的工作是将html标识解析为解析树。

HTML文法定义（The HTML grammar definition）

W3C组织制定规范定义了HTML的词汇表和语法。

非上下文无关文法（Not a context free grammar）

正如在解析简介中提到的，上下文无关文法的语法可以用类似BNF的格式来定义。

不幸的是，所有的传统解析方式都不适用于html（当然我提出它们并不只是因为好玩，它们将用来解析css和js），html不能简单的用解析所需的上下文无关文法来定义。

Html有一个正式的格式定义——DTD（Document Type Definition文档类型定义）——但它并不是上下文无关文法，html更接近于xml，现在有很多可用的xml解析器，html有个xml的变体——xhtml，它们间的不同在于，html更宽容，它允许忽略一些特定标签，有时可以省略开始或结束标签。总的来说，它是一种soft语法，不像xml呆板、固执。

显然，这个看起来很小的差异却带来了很大的不同。一方面，这是html流行的原因——它的宽容使web开发人员的工作更加轻松，但另一方面，这也使很难去写一个格式化的文法。所以，html的解析并不简单，它既不能用传统的解析器解析，也不能用xml解析器解析。

HTML DTD

Html适用DTD格式进行定义，这一格式是用于定义SGML家族的语言，包括了对所有允许元素及它们的属性和层次关系的定义。正如前面提到的，html DTD并没有生成一种上下文无关文法。

DTD有一些变种，标准模式只遵守规范，而其他模式则包含了对浏览器过去所使用标签的支持，这么做是为了兼容以前内容。最新的标准DTD在http://www.w3.org/TR/html4/strict.dtd

DOM

输出的树，也就是解析树，是由DOM元素及属性节点组成的。DOM是文档对象模型的缩写，它是html

文档的对象表示，作为html元素的外部接口供js等调用。

树的根是“document”对象。

DOM和标签基本是一一对应的关系，例如，如下的标签：

```
<html>
<body>
<p>
Hello DOM
</p>
<div><img src=”example.png” /></div>
</body>
</html>
```

将会被转换为下面的DOM树：

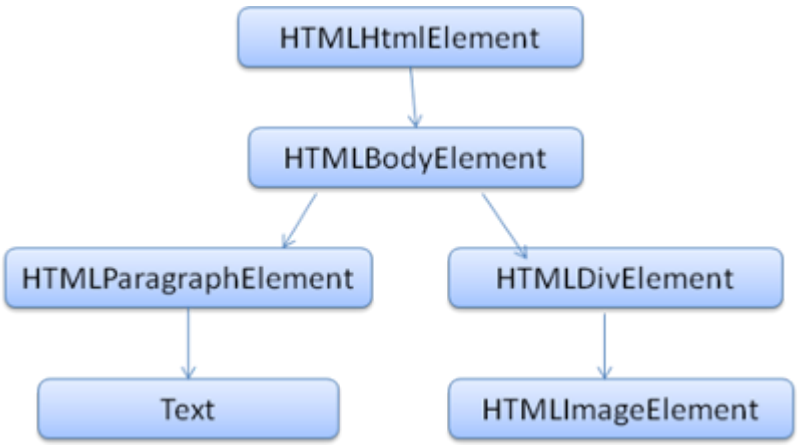


图8：示例标签对应的DOM树

和html一样，DOM的规范也是由W3C组织制定的。访问http://www.w3.org/DOM/DOMTR，这是使用文档的一般规范。一个模型描述一种特定的html元素，可以在http://www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/idl-definitions.htm查看html定义。

这里所谓的树包含了DOM节点是说树是由实现了DOM接口的元素构建而成的，浏览器使用已被浏览器内部使用的其他属性的具体实现。

解析算法（The parsing algorithm）

正如前面章节中讨论的，hmtl不能被一般的自顶向下或自底向上的解析器所解析。

原因是：

1. 这门语言本身的宽容特性

2. 浏览器对一些常见的非法html有容错机制

3. 解析过程是往复的，通常源码不会在解析过程中发生改变，但在html中，脚本标签包含的“document.write”可能添加标签，这说明在解析过程中实际上修改了输入。

不能使用正则解析技术，浏览器为html定制了专属的解析器。

Html5规范中描述了这个解析算法，算法包括两个阶段——符号化及构建树。

符号化是词法分析的过程，将输入解析为符号，html的符号包括开始标签、结束标签、属性名及属性值。

符号识别器识别出符号后，将其传递给树构建器，并读取下一个字符，以识别下一个符号，这样直到处理完所有输入。

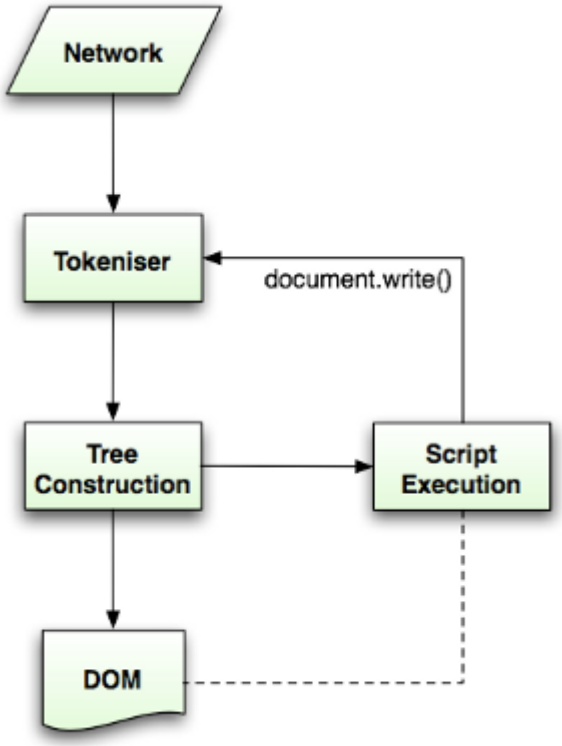


图9：HTML解析流程

符号识别算法（The tokenization algorithm）

算法输出html符号，该算法用状态机表示。每次读取输入流中的一个或多个字符，并根据这些字符转移到下一个状态，当前的符号状态及构建树状态共同影响结果，这意味着，读取同样的字符，可能因为当前状态的不同，得到不同的结果以进入下一个正确的状态。

这个算法很复杂，这里用一个简单的例子来解释这个原理。

基本示例——符号化下面的html：

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

初始状态为“Data State”，当遇到“<”字符，状态变为“Tag open state”，读取一个a－z的字符将产生一个开始标签符号，状态相应变为“Tag name state”，一直保持这个状态直到读取到“>”，每个字符都附加到这个符号名上，例子中创建的是一个html符号。

当读取到“>”，当前的符号就完成了，此时，状态回到“Data state”，“<body>”重复这一处理过程。到这里，html和body标签都识别出来了。现在，回到“Data state”，读取“Hello world”中的字符“H”将创

建并识别出一个字符符号，这里会为“Hello world”中的每个字符生成一个字符符号。

这样直到遇到“</body>”中的“<”。现在，又回到了“Tag open state”，读取下一个字符“/”将创建一个闭合标签符号，并且状态转移到“Tag name state”，还是保持这一状态，直到遇到“>”。然后，产生一个新的标签符号并回到“Data state”。后面的“</html>”将和“</body>”一样处理。

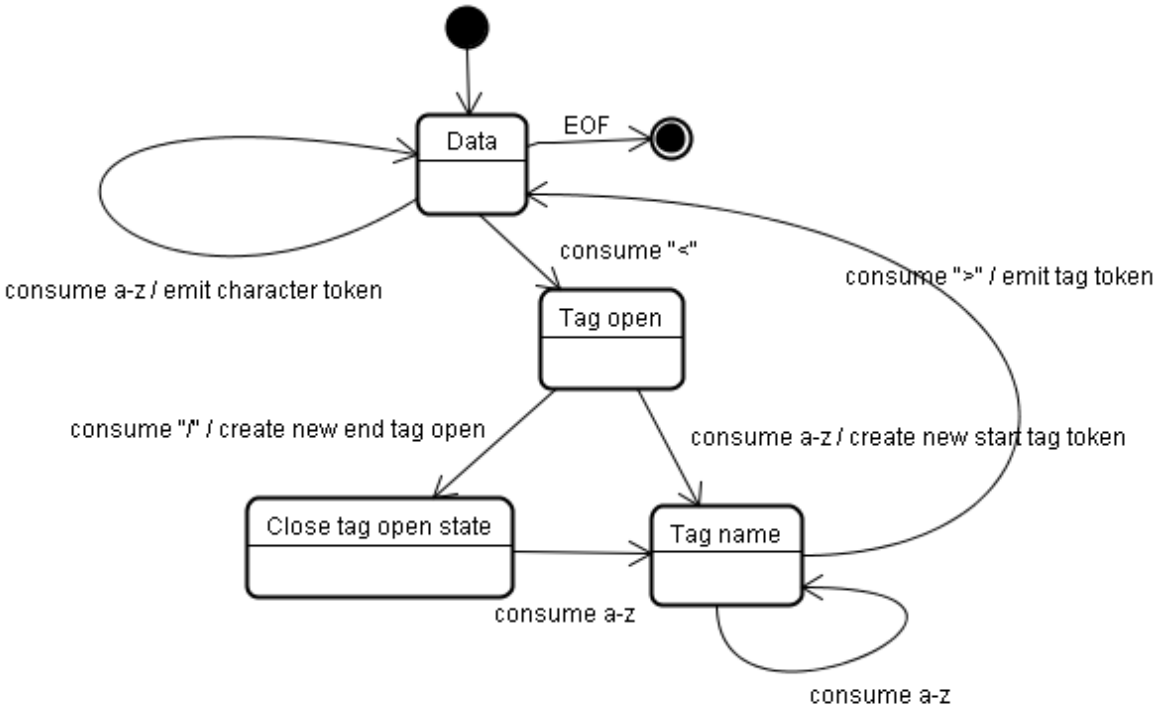


图10：符号化示例输入

树的构建算法（Tree construction algorithm）

在树的构建阶段，将修改以Document为根的DOM树，将元素附加到树上。每个由符号识别器识别生成的节点将会被树构造器进行处理，规范中定义了每个符号相对应的Dom元素，对应的Dom元素将会被创建。这些元素除了会被添加到Dom树上，还将被添加到开放元素堆栈中。这个堆栈用来纠正嵌套的未匹配和未闭合标签，这个算法也是用状态机来描述，所有的状态采用插入模式。

来看一下示例中树的创建过程：

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

构建树这一阶段的输入是符号识别阶段生成的符号序列。

首先是“initial mode”，接收到html符号后将转换为“before html”模式，在这个模式中对这个符号进行再处理。此时，创建了一个HTMLHtmlElement元素，并将其附加到根Document对象上。

状态此时变为“before head”，接收到body符号时，即使这里没有head符号，也将自动创建一个HTMLHeadElement元素并附加到树上。

现在，转到“in head”模式，然后是“after head”。到这里，body符号会被再次处理，将创建一个HTMLBodyElement并插入到树中，同时，转移到“in body”模式。

然后，接收到字符串“Hello world”的字符符号，第一个字符将导致创建并插入一个text节点，其他字符将附加到该节点。

接收到body结束符号时，转移到“after body”模式，接着接收到html结束符号，这个符号意味着转移到了“after after body”模式，当接收到文件结束符时，整个解析过程结束。

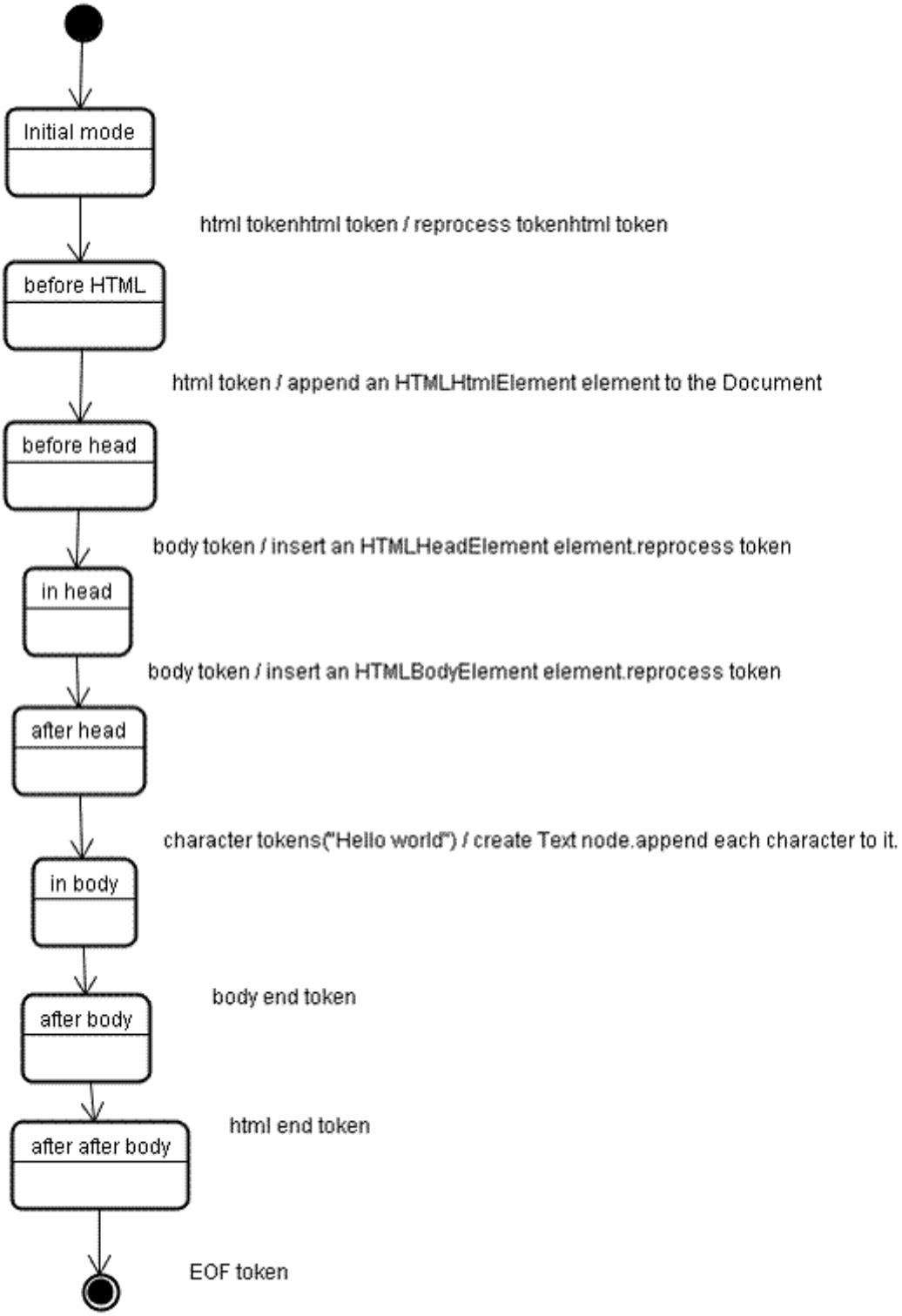


图11：示例html树的构建过程

解析结束时的处理（Action when the parsing is finished）

在这个阶段，浏览器将文档标记为可交互的，并开始解析处于延时模式中的脚本——这些脚本在文档解析后执行。

文档状态将被设置为完成，同时触发一个load事件。

Html5规范中有符号化及构建树的完整算法(http://www.w3.org/TR/html5/syntax.html#html-parser)。

浏览器容错（Browsers error tolerance）

你从来不会在一个html页面上看到"无效语法"这样的错误，浏览器修复了无效内容并继续工作。

以下面这段html为例：

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

这段html违反了很多规则（mytag不是合法的标签，p及div错误的嵌套等等），但是浏览器仍然可以没有任何怨言的继续显示，它在解析的过程中修复了html作者的错误。

浏览器都具有错误处理的能力，但是，另人惊讶的是，这并不是html最新规范的内容，就像书签及前进后退按钮一样，它只是浏览器长期发展的结果。一些比较知名的非法html结构，在许多站点中出现过，浏览器都试着以一种和其他浏览器一致的方式去修复。

Html5规范定义了这方面的需求，webkit在html解析类开始部分的注释中做了很好的总结。

解析器将符号化的输入解析为文档并创建文档，但不幸的是，我们必须处理很多没有很好格式化的html文档，至少要小心下面几种错误情况。

1. 在未闭合的标签中添加明确禁止的元素。这种情况下，应该先将前一标签闭合

2. 不能直接添加元素。有些人在写文档的时候会忘了中间一些标签（或者中间标签是可选的），比如HTML HEAD BODY TR TD LI等

3. 想在一个行内元素中添加块状元素。关闭所有的行内元素，直到下一个更高的块状元素

4. 如果这些都不行，就闭合当前标签直到可以添加该元素。

下面来看一些webkit容错的例子：

</br>替代

一些网站使用</br>替代
，为了兼容IE和Firefox，webkit将其看作
。

代码：

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

Note －这里的错误处理在内部进行，用户看不到。

迷路的表格

这指一个表格嵌套在另一个表格中，但不在它的某个单元格内。

比如下面这个例子：

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

webkit将会将嵌套的表格变为两个兄弟表格：

```
<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使用堆栈存放当前的元素内容，它将从外部表格的堆栈中弹出内部的表格，则它们变为了兄弟表格。

嵌套的表单元素

用户将一个表单嵌套到另一个表单中，则第二个表单将被忽略。

代码：

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

太深的标签继承

www.liceo.edu.mx是一个由嵌套层次的站点的例子，最多只允许20个相同类型的标签嵌套，多出来

的将被忽略。

代码：

```
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;
}
```

放错了地方的html、body闭合标签

又一次不言自明。

支持不完整的html。我们从来不闭合body，因为一些愚蠢的网页总是在还未真正结束时就闭合它。我们依赖调用end方法去执行关闭的处理。

代码：

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

所以，web开发者要小心了，除非你想成为webkit容错代码的范例，否则还是写格式良好的html吧。

CSS解析（CSS parsing）

还记得简介中提到的解析的概念吗，不同于html，css属于上下文无关文法，可以用前面所描述的解析器来解析。Css规范定义了css的词法及语法文法。

看一些例子：

每个符号都由正则表达式定义了词法文法（词汇表）：

```
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”是识别器的缩写，相当于一个class名，“name”是一个元素id（用“#”引用）。

语法用BNF进行描述：

```
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：一个规则集合有这样的结构

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

div.error和a.error时选择器，大括号中的内容包含了这条规则集合中的规则，这个结构在下面的定义中正式的定义了：

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

这说明，一个规则集合具有一个或是可选个数的多个选择器，这些选择器以逗号和空格（S表示空格）进行分隔。每个规则集合包含大括号及大括号中的一条或多条以分号隔开的声明。声明和选择器在后面进行定义。

Webkit CSS解析器（Webkit CSS parser）

Webkit使用Flex和Bison解析生成器从CSS语法文件中自动生成解析器。回忆一下解析器的介绍，Bison创建一个自底向上的解析器，Firefox使用自顶向下解析器。它们都是将每个css文件解析为样式表对象，每个对象包含css规则，css规则对象包含选择器和声明对象，以及其他一些符合css语法的对象。

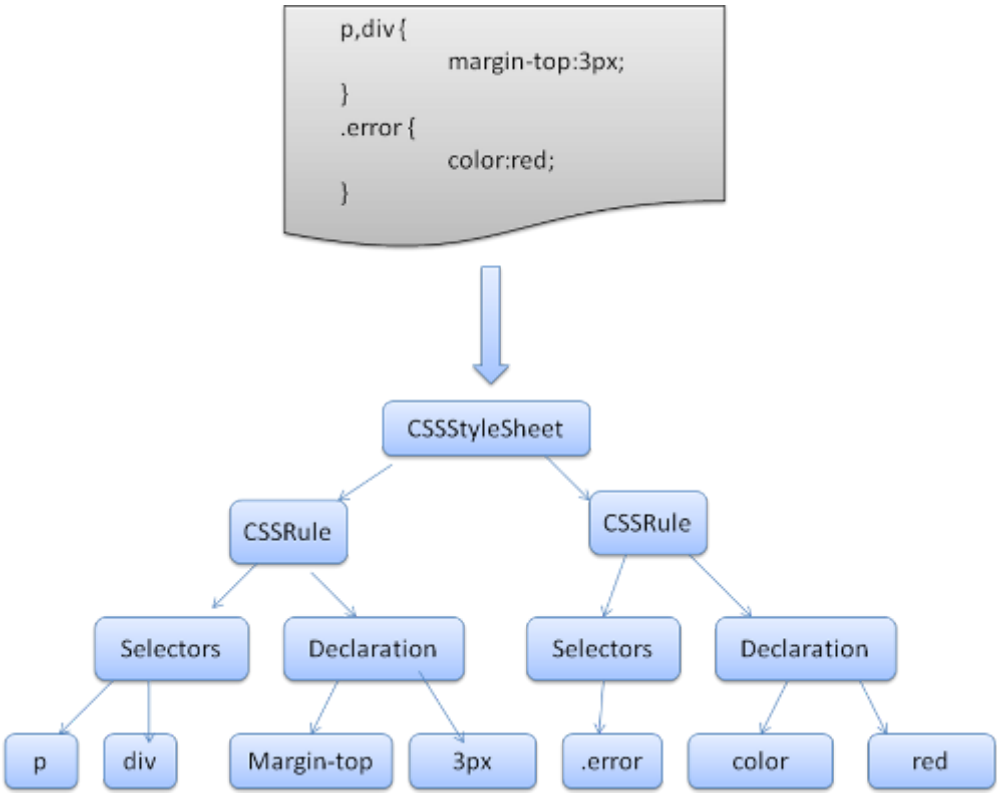


图12：解析css

处理脚本及样式表的顺序（The order of processing scripts and style sheets）

脚本

web的模式是同步的，开发者希望解析到一个script标签时立即解析执行脚本，并阻塞文档的解析直到脚本执行完。如果脚本是外引的，则网络必须先请求到这个资源——这个过程也是同步的，会阻塞文档的解析直到资源被请求到。这个模式保持了很多年，并且在html4及html5中都特别指定了。开发者可以将脚本标识为defer，以使其不阻塞文档解析，并在文档解析结束后执行。Html5增加了标记脚本为异步的选项，以使脚本的解析执行使用另一个线程。

预解析（Speculative parsing）

Webkit和Firefox都做了这个优化，当执行脚本时，另一个线程解析剩下的文档，并加载后面需要通过网络加载的资源。这种方式可以使资源并行加载从而使整体速度更快。需要注意的是，预解析并不改变Dom树，它将这个工作留给主解析过程，自己只解析外部资源的引用，比如外部脚本、样式表及图片。

样式表（Style sheets）

样式表采用另一种不同的模式。理论上，既然样式表不改变Dom树，也就没有必要停下文档的解析等待它们，然而，存在一个问题，脚本可能在文档的解析过程中请求样式信息，如果样式还没有加载和解析，脚本将得到错误的值，显然这将会导致很多问题，这看起来是个边缘情况，但确实很常见。Firefox在存在样式表还在加载和解析时阻塞所有的脚本，而Chrome只在当脚本试图访问某些可能被未加载的样式表所影响的特定的样式属性时才阻塞这些脚本。

四、渲染树构建（Render tree construction）

当Dom树构建完成时，浏览器开始构建另一棵树——渲染树。渲染树由元素显示序列中的可见元素组成，它是文档的可视化表示，构建这棵树是为了以正确的顺序绘制文档内容。

Firefox将渲染树中的元素称为frames，WebKit则用renderer或渲染对象来描述这些元素。

一个渲染对象知道怎么布局及绘制自己及它的children。

RenderObject是Webkit的渲染对象基类，它的定义如下：

```
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
}
```

每个渲染对象用一个和该节点的css盒模型相对应的矩形区域来表示，正如css2所描述的那样，它包含诸如宽、高和位置之类的几何信息。盒模型的类型受该节点相关的display样式属性的影响（参考样式计算章节）。下面的webkit代码说明了如何根据display属性决定某个节点创建何种类型的渲染对象。

```
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;
}
```

元素的类型也需要考虑，例如，表单控件和表格带有特殊的框架。

在Webkit中，如果一个元素想创建一个特殊的渲染对象，它需要重写“createRenderer”方法，使渲染对象指向不包含几何信息的样式对象。

渲染树和Dom树的关系（The render tree relation to the DOM tree）

渲染对象和Dom元素相对应，但这种对应关系不是一对一的，不可见的Dom元素不会被插入渲染树，例如head元素。另外，display属性为none的元素也不会在渲染树中出现（visibility属性为hidden的元素将出现在渲染树中）。

还有一些Dom元素对应几个可见对象，它们一般是一些具有复杂结构的元素，无法用一个矩形来描述。例如，select元素有三个渲染对象——一个显示区域、一个下拉列表及一个按钮。同样，当文本因为宽度不够而折行时，新行将作为额外的渲染元素被添加。另一个多个渲染对象的例子是不规范的html，根据css规范，一个行内元素只能仅包含行内元素或仅包含块状元素，在存在混合内容时，将会创建匿名的块状渲染对象包裹住行内元素。

一些渲染对象和所对应的Dom节点不在树上相同的位置，例如，浮动和绝对定位的元素在文本流之外，在两棵树上的位置不同，渲染树上标识出真实的结构，并用一个占位结构标识出它们原来的位置。

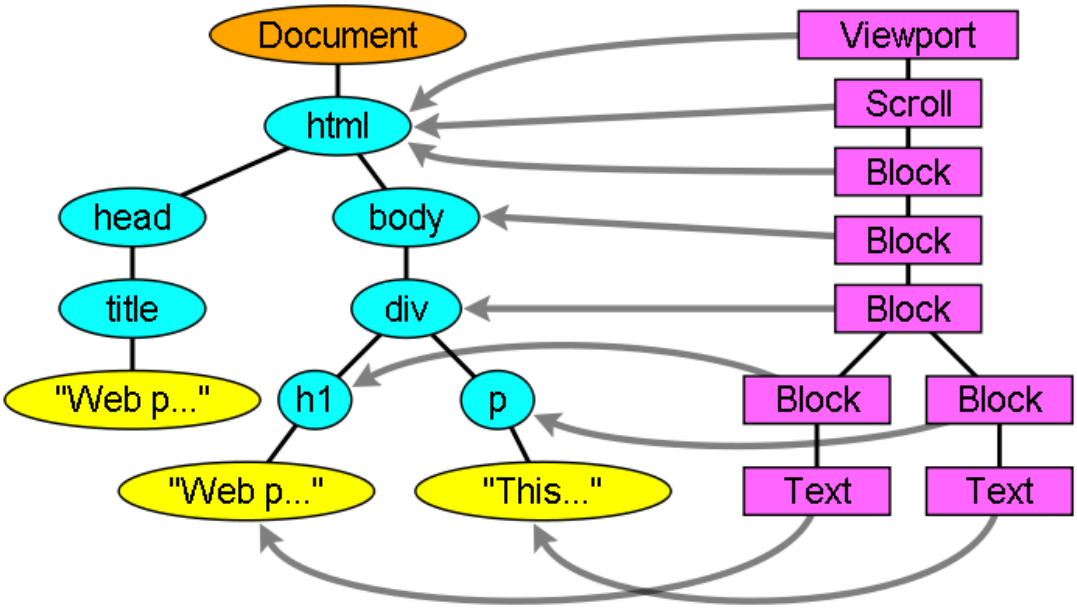


图13：渲染树及对应的Dom树

创建树的流程（The flow of constructing the tree）

Firefox中，表述为一个监听Dom更新的监听器，将frame的创建委派给Frame Constructor，这个构建器计算样式（参看样式计算）并创建一个frame。

Webkit中，计算样式并生成渲染对象的过程称为attachment，每个Dom节点有一个attach方法，attachment的过程是同步的，调用新节点的attach方法将节点插入到Dom树中。

处理html和body标签将构建渲染树的根，这个根渲染对象对应被css规范称为containing block的元素——包含了其他所有块元素的顶级块元素。它的大小就是viewport——浏览器窗口的显示区域，Firefox称它为viewPortFrame，webkit称为RenderView，这个就是文档所指向的渲染对象，树中其他的部分都将作为一个插入的Dom节点被创建。

样式计算（Style Computation）

创建渲染树需要计算出每个渲染对象的可视属性，这可以通过计算每个元素的样式属性得到。

样式包括各种来源的样式表，行内样式元素及html中的可视化属性（例如bgcolor），可视化属性转化为css样式属性。

样式表来源于浏览器默认样式表，及页面作者和用户提供的样式表——有些样式是浏览器用户提供的（浏览器允许用户定义喜欢的样式，例如，在Firefox中，可以通过在Firefox Profile目录下放置样式表实现）。

计算样式的一些困难：

1. 样式数据是非常大的结构，保存大量的样式属性会带来内存问题。

2. 如果不进行优化，找到每个元素匹配的规则会导致性能问题，为每个元素查找匹配的规则都需要遍历整个规则表，这个过程有很大的工作量。选择符可能有复杂的结构，匹配过程如果沿着一条开始看似正确，后来却被证明是无用的路径，则必须去尝试另一条路径。

例如，下面这个复杂选择符

div div div div｛…｝

这意味着规则应用到三个div的后代div元素，选择树上一条特定的路径去检查，这可能需要遍历节点树，最后却发现它只是两个div的后代，并不使用该规则，然后则需要沿着另一条路径去尝试

3. 应用规则涉及非常复杂的级联，它们定义了规则的层次

我们来看一下浏览器如何处理这些问题：

共享样式数据（Sharing style data）

WebkKit节点引用样式对象（渲染样式），某些情况下，这些对象可以被节点间共享，这些节点需要是兄弟或是表兄弟节点，并且：

1. 这些元素必须处于相同的鼠标状态（比如不能一个处于hover，而另一个不是）

2. 不能有元素具有id

3. 标签名必须匹配

4. class属性必须匹配

5. 对应的属性必须相同

6. 链接状态必须匹配

7. 焦点状态必须匹配

8. 不能有元素被属性选择器影响

9. 元素不能有行内样式属性

10. 不能有生效的兄弟选择器，webcore在任何兄弟选择器相遇时只是简单的抛出一个全局转换，并且在它们显示时使整个文档的样式共享失效，这些包括＋选择器和类似:first-child和:last-child这样的选择器。

Firefox规则树（Firefox rule tree）

Firefox用两个树用来简化样式计算－规则树和样式上下文树，WebKit也有样式对象，但它们并没有存储在类似样式上下文树这样的树中，只是由Dom节点指向其相关的样式。

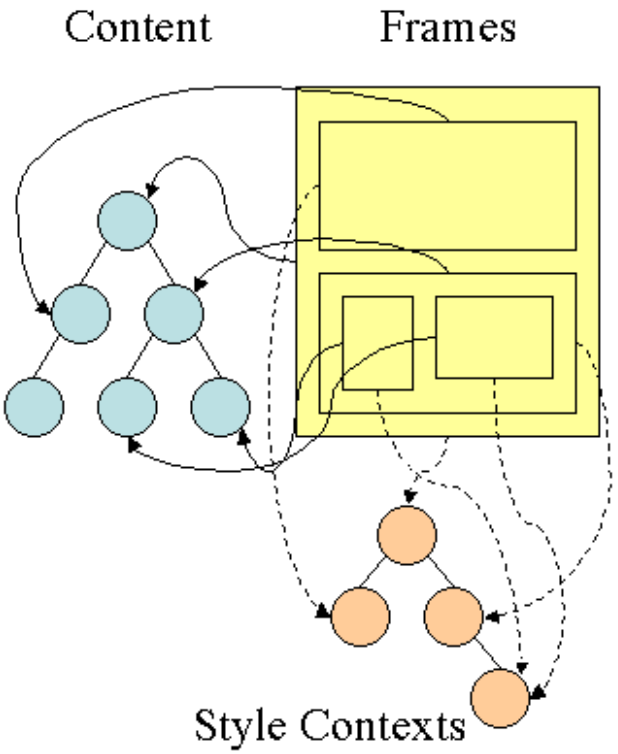
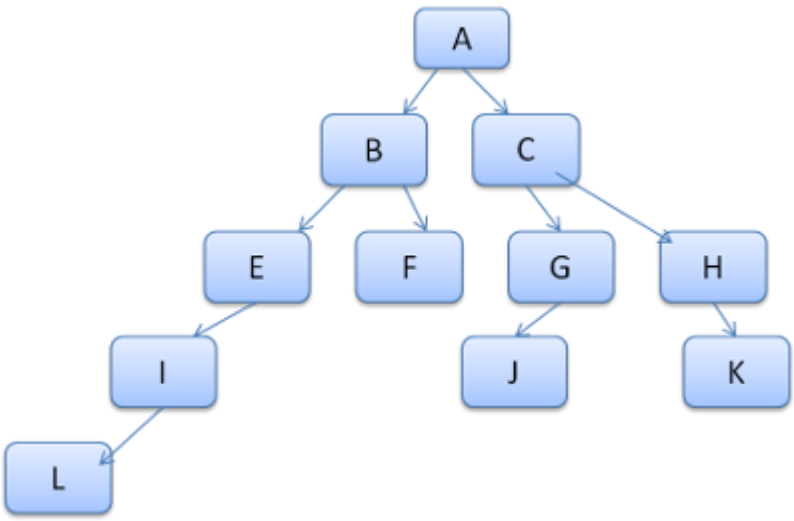


图14：Firefox样式上下文树

样式上下文包含最终值，这些值是通过以正确顺序应用所有匹配的规则，并将它们由逻辑值转换为具体的值，例如，如果逻辑值为屏幕的百分比，则通过计算将其转化为绝对单位。样式树的使用确实很巧妙，它使得在节点中共享的这些值不需要被多次计算，同时也节省了存储空间。

所有匹配的规则都存储在规则树中，一条路径中的底层节点拥有最高的优先级，这棵树包含了所找到的所有规则匹配的路径（译注：可以取巧理解为每条路径对应一个节点，路径上包含了该节点所匹配的所有规则）。规则树并不是一开始就为所有节点进行计算，而是在某个节点需要计算样式时，才进行相应的计算并将计算后的路径添加到树中。

我们将树上的路径看成辞典中的单词，假如已经计算出了如下的规则树：



假如需要为内容树中的另一个节点匹配规则，现在知道匹配的规则（以正确的顺序）为B-E-I，因为我们已经计算出了路径A-B-E-I-L，所以树上已经存在了这条路径，剩下的工作就很少了。

现在来看一下树如何保存。

结构化

样式上下文按结构划分，这些结构包括类似border或color这样的特定分类的样式信息。一个结构中的所有特性不是继承的就是非继承的，对继承的特性，除非元素自身有定义，否则就从它的parent继承。非继承的特性（称为reset特性）如果没有定义，则使用默认的值。

样式上下文树缓存完整的结构（包括计算后的值），这样，如果底层节点没有为一个结构提供定义，则使用上层节点缓存的结构。

使用规则树计算样式上下文

当为一个特定的元素计算样式时，首先计算出规则树中的一条路径，或是使用已经存在的一条，然后使用路径中的规则去填充新的样式上下文，从样式的底层节点开始，它具有最高优先级（通常是最特定的选择器），遍历规则树，直到填满结构。如果在那个规则节点没有定义所需的结构规则，则沿着路径向上，直到找到该结构规则。

如果最终没有找到该结构的任何规则定义，那么如果这个结构是继承型的，则找到其在内容树中的parent的结构，这种情况下，我们也成功的共享了结构；如果这个结构是reset型的，则使用默认的值。

如果特定的节点添加了值，那么需要做一些额外的计算以将其转换为实际值，然后在树上的节点缓存该值，使它的children可以使用。

当一个元素和它的一个兄弟元素指向同一个树节点时，完整的样式上下文可以被它们共享。

来看一个例子：假设有下面这段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"> verybigerror</span>
error
</p>
</div>
<div class="err" id="div2">another error</div>
</body>
</html>
```

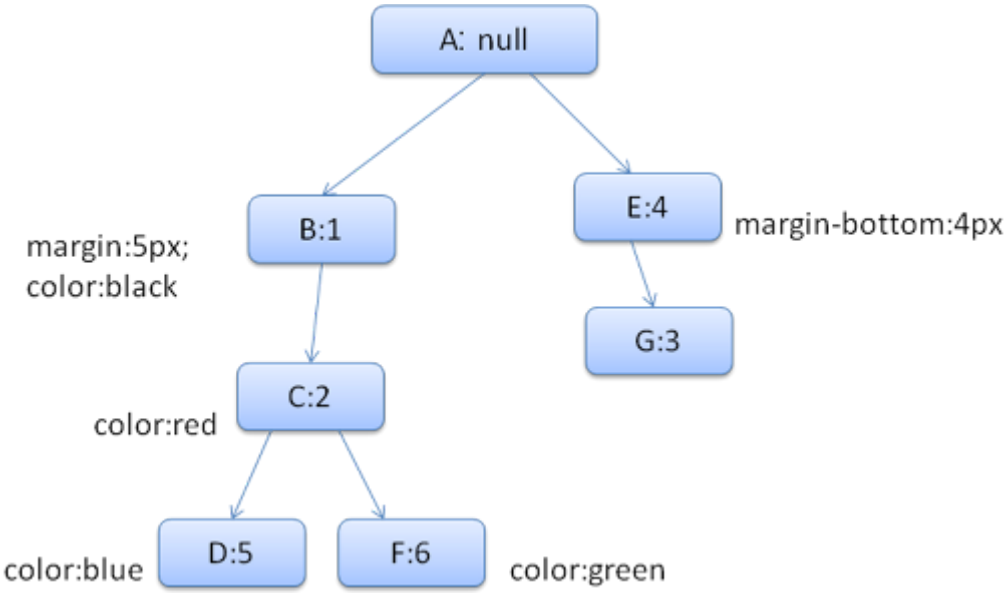
以及下面这些规则

```
1.div {margin:5px;color:black}
```

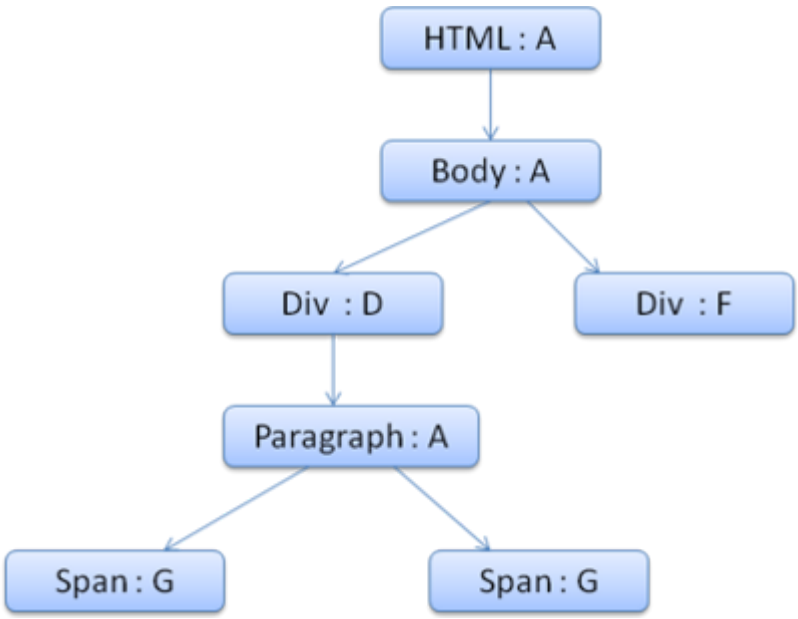
```
2..err {color:red}
3..big {margin-top:3px}
4.div span {margin-bottom:4px}
5.#div1 {color:blue}
6.#div2 {color:green}
```

简化下问题，我们只填充两个结构——color和margin，color结构只包含一个成员－颜色，margin结构包含四边。

生成的规则树如下（节点名：指向的规则）



上下文树如下（节点名：指向的规则节点）



假设我们解析html，遇到第二个div标签，我们需要为这个节点创建样式上下文，并填充它的样式结构。

我们进行规则匹配，找到这个div匹配的规则为1、2、6，我们发现规则树上已经存在了一条我们可以使用的路径1、2，我们只需为规则6新增一个节点添加到下面（就是规则树中的F）。

然后创建一个样式上下文并将其放到上下文树中，新的样式上下文将指向规则树中的节点F。

现在我们需要填充这个样式上下文，先从填充margin结构开始，既然最后一个规则节点没有添加margin结构，沿着路径向上，直到找到缓存的前面插入节点计算出的结构，我们发现B是最近的指定margin值的节点。因为已经有了color结构的定义，所以不能使用缓存的结构，既然color只有一个属性，也就不需要沿着路径向上填充其他属性。计算出最终值（将字符串转换为RGB等），并缓存计算后的结构。

第二个span元素更简单，进行规则匹配后发现它指向规则G，和前一个span一样，既然有兄弟节点指向同一个节点，就可以共享完整的样式上下文，只需指向前一个span的上下文。

因为结构中包含继承自parent的规则，上下文树做了缓存（color特性是继承来的，但Firefox将其视为reset并在规则树中缓存）。

例如，如果我们为一个paragraph的文字添加规则：

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

那么这个p在内容树中的子节点div，会共享和它parent一样的font结构，这种情况发生在没有为这个div指定font规则时。

Webkit中，并没有规则树，匹配的声明会被遍历四次，先是应用非important的高优先级属性（之所以先应用这些属性，是因为其他的依赖于它们－比如display），其次是高优先级important的，接着是一般优先级非important的，最后是一般优先级important的规则。这样，出现多次的属性将被按照正确的级联顺序进行处理，最后一个生效。

总结一下，共享样式对象（结构中完整或部分内容）解决了问题1和3，Firefox的规则树帮助以正确的顺序应用规则。

对规则进行处理以简化匹配过程

样式规则有几个来源：

- 外部样式表或style标签内的css规则
- 行内样式属性
- html可视化属性（映射为相应的样式规则）

后面两个很容易匹配到元素，因为它们所拥有的样式属性和html属性可以将元素作为key进行映射。

就像前面问题2所提到的，css的规则匹配可能很狡猾，为了解决这个问题，可以先对规则进行处理，以使其更容易被访问。

解析完样式表之后，规则会根据选择符添加一些hash映射，映射可以是根据id、class、标签名或是任何不属于这些分类的综合映射。如果选择符为id，规则将被添加到id映射，如果是class，则被添加到class映射，等等。

这个处理是匹配规则更容易，不需要查看每个声明，我们能从映射中找到一个元素的相关规则，这个优化使在进行规则匹配时减少了95+%的工作量。

来看下面的样式规则：

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

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

第一条规则将被插入class映射，第二条插入id映射，第三条是标签映射。

下面这个html片段：

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

我们首先找到p元素对应的规则，class映射将包含一个“error”的key，找到p.error的规则，div在id映射和标签映射中都有相关的规则，剩下的工作就是找出这些由key对应的规则中哪些确实是正确匹配的。

例如，如果div的规则是

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

这也是标签映射产生的，因为key是最右边的选择符，但它并不匹配这里的div元素，因为这里的div没有table祖先。

Webkit和Firefox都会做这个处理。

以正确的级联顺序应用规则

样式对象拥有对应所有可见属性的属性，如果特性没有被任何匹配的规则所定义，那么一些特性可以从parent的样式对象中继承，另外一些使用默认值。

这个问题的产生是因为存在不止一处的定义，这里用级联顺序解决这个问题。

样式表的级联顺序

一个样式属性的声明可能在几个样式表中出现，或是在一个样式表中出现多次，因此，应用规则的顺序至关重要，这个顺序就是级联顺序。根据css2的规范，级联顺序为（从低到高）：

1. 浏览器声明
2. 用户声明
3. 作者的一般声明
4. 作者的important声明
5. 用户important声明

浏览器声明是最不重要的，用户只有在声明被标记为important时才会覆盖作者的声明。具有同等级别的声明将根据specifity以及它们被定义时的顺序进行排序。Html可视化属性将被转换为匹配的css声明，它们被视为最低优先级的作者规则。

Specifity

Css2规范中定义的选择符specifity如下：

- 如果声明来自style属性，而不是一个选择器的规则，则计1，否则计0（=a）
- 计算选择器中id属性的数量（=b）
- 计算选择器中class及伪类的数量（=c）

- 计算选择器中元素名及伪元素的数量（=d）

连接a－b－c－d四个数量（用一个大基数的计算系统）将得到specifity。这里使用的基数由分类中最高的基数定义。例如，如果a为14，可以使用16进制。不同情况下，a为17时，则需要使用阿拉伯数字17作为基数，这种情况可能在这个选择符时发生html body div div …（选择符中有17个标签，一般不太可能）。

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 */

/* a=1 b=0 c=0 d=0 -> specificity = 1,0,0,0 */
```

规则排序

规则匹配后，需要根据级联顺序对规则进行排序，WebKit先将小列表用冒泡排序，再将它们合并为一个大列表，WebKit通过为规则复写“>”操作来执行排序：

```
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;
}
```

逐步处理Gradual process

webkit使用一个标志位标识所有顶层样式表都已加载，如果在attch时样式没有完全加载，则放置占位符，并在文档中标记，一旦样式表完成加载就重新进行计算。

五、布局（Layout）

当渲染对象被创建并添加到树中，它们并没有位置和大小，计算这些值的过程称为layout或reflow。

Html使用基于流的布局模型，意味着大部分时间，可以以单一的途径进行几何计算。流中靠后的元素并不会影响前面元素的几何特性，所以布局可以在文档中从右向左、自上而下的进行。也存在一些例外，比如html tables。

坐标系统相对于根frame，使用top和left坐标。

布局是一个递归的过程，由根渲染对象开始，它对应html文档元素，布局继续递归的通过一些或所有的frame层级，为每个需要几何信息的渲染对象进行计算。

根渲染对象的位置是0,0，它的大小是viewport－浏览器窗口的可见部分。

所有的渲染对象都有一个layout或reflow方法，每个渲染对象调用需要布局的children的layout方法。

Dirty bit系统

为了不因为每个小变化都全部重新布局，浏览器使用一个dirty bit系统，一个渲染对象发生了变化或是被添加了，就标记它及它的children为dirty——需要layout。存在两个标识——dirty及children are dirty，children are dirty说明即使这个渲染对象可能没问题，但它至少有一个child需要layout。

全局和增量layout

当layout在整棵渲染树触发时，称为全局layout，这可能在下面这些情况下发生：

1. 一个全局的样式改变影响所有的渲染对象，比如字号的改变。

2. 窗口resize。

layout也可以是增量的，这样只有标志为dirty的渲染对象会重新布局（也将导致一些额外的布局）。增量layout会在渲染对象dirty时异步触发，例如，当网络接收到新的内容并添加到Dom树后，新的渲染对象会添加到渲染树中。

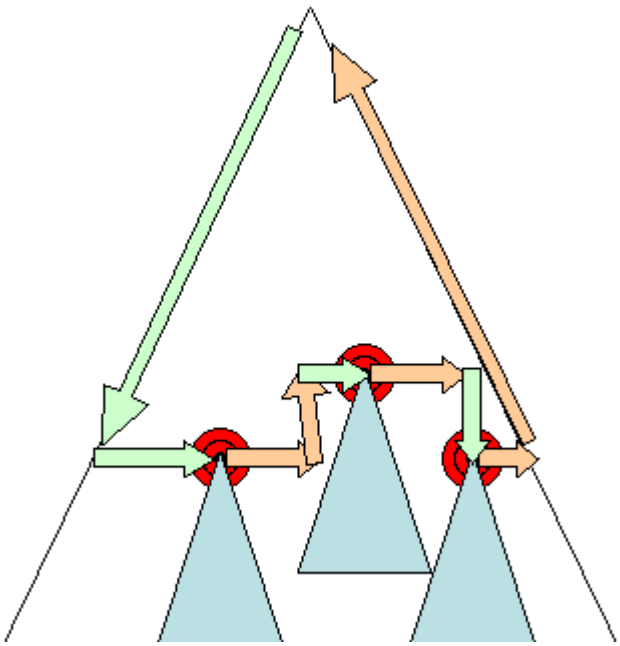


图20：增量layout

异步和同步layout

增量layout的过程是异步的，Firefox为增量layout生成了reflow队列，以及一个调度执行这些批处理命令。WebKit也有一个计时器用来执行增量layout－遍历树，为dirty状态的渲染对象重新布局。

另外，当脚本请求样式信息时，例如“offsetHeight”，会同步的触发增量布局。

全局的layout一般都是同步触发。

有些时候，layout会被作为一个初始layout之后的回调，比如滑动条的滑动。

优化

当一个layout因为resize或是渲染位置改变（并不是大小改变）而触发时，渲染对象的大小将会从缓存中读取，而不会重新计算。

一般情况下，如果只有子树发生改变，则layout并不从根开始。这种情况发生在，变化发生在元素自身并且不影响它周围元素，例如，将文本插入文本域（否则，每次击键都将触发从根开始的重排）。

layout过程

layout一般有下面这几个部分：

1. parent渲染对象决定它的宽度

2. parent渲染对象读取chilidren，并：

    a. 放置child渲染对象（设置它的x和y）

    b. 在需要时（它们当前为dirty或是处于全局layout或者其他原因）调用child渲染对象的layout，这将计算child的高度

    c. parent渲染对象使用child渲染对象的累积高度，以及margin和padding的高度来设置自己的高度－这将被parent渲染对象的parent使用

    d. 将dirty标识设置为false

Firefox使用一个“state”对象（nsHTMLReflowState）做为参数去布局（firefox称为reflow），state包含parent的宽度及其他内容。

Firefox布局的输出是一个“metrics”对象（nsHTMLReflowMetrics）。它包括渲染对象计算出的高度。

宽度计算

渲染对象的宽度使用容器的宽度、渲染对象样式中的宽度及margin、border进行计算。例如，下面这个div的宽度：

webkit中宽度的计算过程是（RenderBox类的calcWidth方法）：

- 容器的宽度是容器的可用宽度和0中的最大值，这里的可用宽度为：contentWidth=clientWidth()-paddingLeft()-paddingRight()，clientWidth和clientHeight代表一个对象内部的不包括border和滑动条的大小

- 元素的宽度指样式属性width的值，它可以通过计算容器的百分比得到一个绝对值
- 加上水平方向上的border和padding

到这里是最佳宽度的计算过程，现在计算宽度的最大值和最小值，如果最佳宽度大于最大宽度则使用最大宽度，如果小于最小宽度则使用最小宽度。最后缓存这个值，当需要layout但宽度未改变时使用。

Line breaking

当一个渲染对象在布局过程中需要折行时，则暂停并告诉它的parent它需要折行，parent将创建额外的渲染对象并调用它们的layout。

六、绘制（Painting）

绘制阶段，遍历渲染树并调用渲染对象的paint方法将它们的内容显示在屏幕上，绘制使用UI基础组件，这在UI的章节有更多的介绍。

全局和增量

和布局一样，绘制也可以是全局的——绘制完整的树——或增量的。在增量的绘制过程中，一些渲染对象以不影响整棵树的方式改变，改变的渲染对象使其在屏幕上的矩形区域失效，这将导致操作系统将其看作dirty区域，并产生一个paint事件，操作系统很巧妙的处理这个过程，并将多个区域合并为一个。Chrome中，这个过程更复杂些，因为渲染对象在不同的进程中，而不是在主进程中。Chrome在一定程度上模拟操作系统的行为，表现为监听事件并派发消息给渲染根，在树中查找到相关的渲染对象，重绘这个对象（往往还包括它的children）。

绘制顺序

css2定义了绘制过程的顺序——http://www.w3.org/TR/CSS21/zindex.html。这个就是元素压入堆栈的顺序，这个顺序影响着绘制，堆栈从后向前进行绘制。

一个块渲染对象的堆栈顺序是：

1. 背景色
2. 背景图
3. border
4. children
5. outline

Firefox显示列表

Firefox读取渲染树并为绘制的矩形创建一个显示列表，该列表以正确的绘制顺序包含这个矩形相关的渲染对象。

用这样的方法，可以使重绘时只需查找一次树，而不需要多次查找——绘制所有的背景、所有的图片、所有的border等等。

Firefox优化了这个过程，它不添加会被隐藏的元素，比如元素完全在其他不透明元素下面。

WebKit矩形存储

重绘前，WebKit将旧的矩形保存为位图，然后只绘制新旧矩形的差集。

七、动态变化

浏览器总是试着以最小的动作响应一个变化，所以一个元素颜色的变化将只导致该元素的重绘，元素位置的变化将大致元素的布局和重绘，添加一个Dom节点，也会大致这个元素的布局和重绘。一些主要的变化，比如增加html元素的字号，将会导致缓存失效，从而引起整数的布局和重绘。

八、渲染引擎的线程

渲染引擎是单线程的，除了网络操作以外，几乎所有的事情都在单一的线程中处理，在Firefox和Safari中，这是浏览器的主线程，Chrome中这是tab的主线程。

网络操作由几个并行线程执行，并行连接的个数是受限的（通常是2－6个）。

事件循环

浏览器主线程是一个事件循环，它被设计为无限循环以保持执行过程的可用，等待事件（例如layout和paint事件）并执行它们。下面是Firefox的主要事件循环代码。

```
while (!mExiting)


NS_ProcessNextEvent(thread);
```

九、CSS2可视模型（CSS2 visual module）

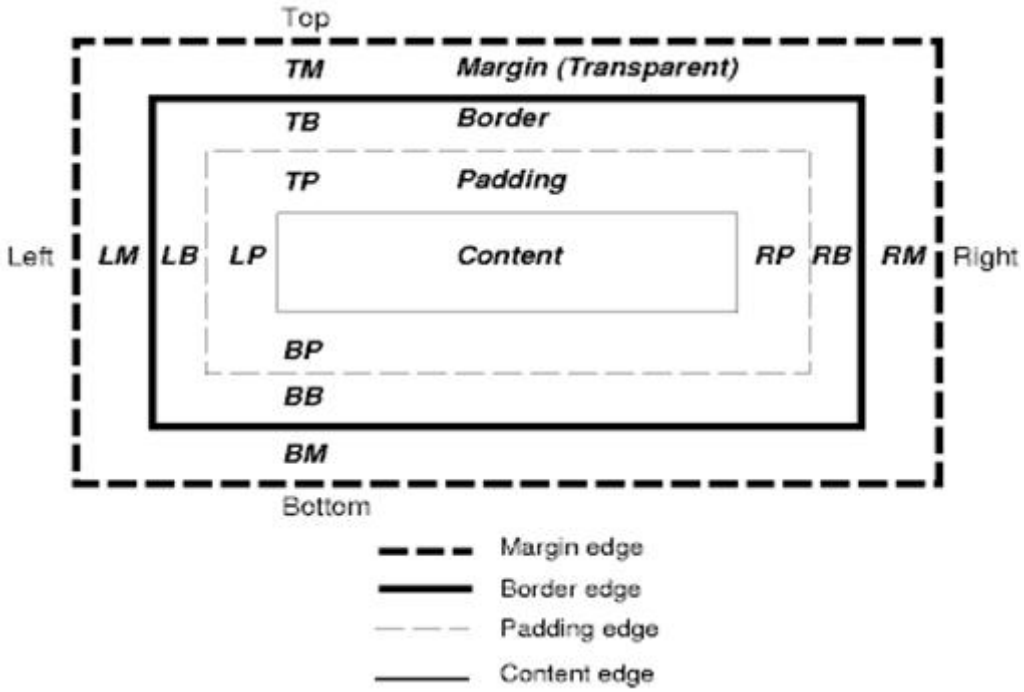
画布The Canvas

根据CSS2规范，术语canvas用来描述格式化的结构所渲染的空间——浏览器绘制内容的地方。画布对每个维度空间都是无限大的，但浏览器基于viewport的大小选择了一个初始宽度。

根据http://www.w3.org/TR/CSS2/zindex.html的定义，画布如果是包含在其他画布内则是透明的，否则浏览器会指定一个颜色。

CSS盒模型

CSS盒模型描述了矩形盒，这些矩形盒是为文档树中的元素生成的，并根据可视的格式化模型进行布局。每个box包括内容区域（如图片、文本等）及可选的四周padding、border和margin区域。



每个节点生成0－n个这样的box。

所有的元素都有一个display属性，用来决定它们生成box的类型，例如：

block －生成块状box

inline －生成一个或多个行内box

none －不生成box

默认的是inline，但浏览器样式表设置了其他默认值，例如，div元素默认为block。可以访问http://www.w3.org/TR/CSS2/sample.html查看更多的默认样式表示例。

定位策略Position scheme

这里有三种策略：

1. normal －对象根据它在文档的中位置定位，这意味着它在渲染树和在Dom树中位置一致，并根据它的盒模型和大小进行布局。

2. float －对象先像普通流一样布局，然后尽可能的向左或是向右移动。

3. absolute －对象在渲染树中的位置和Dom树中位置无关。

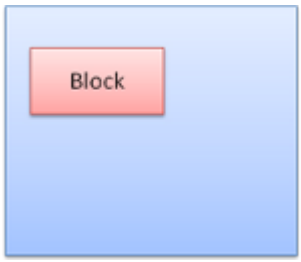
static和relative是normal，absolute和fixed属于absolute。

在static定位中，不定义位置而使用默认的位置。其他策略中，作者指定位置——top、bottom、left、right。

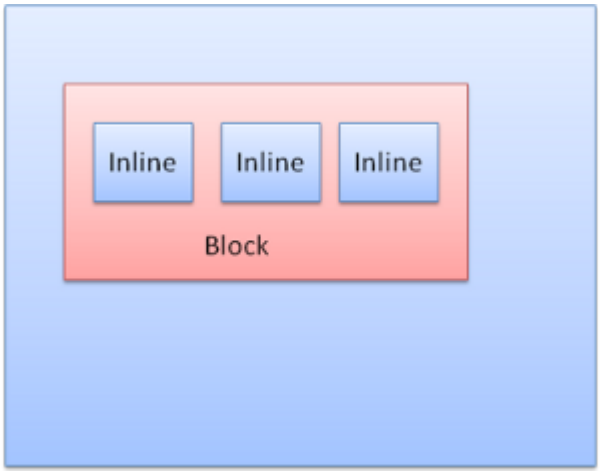
Box布局的方式由这几项决定：box的类型、box的大小、定位策略及扩展信息（比如图片大小和屏幕尺寸）。

Box类型

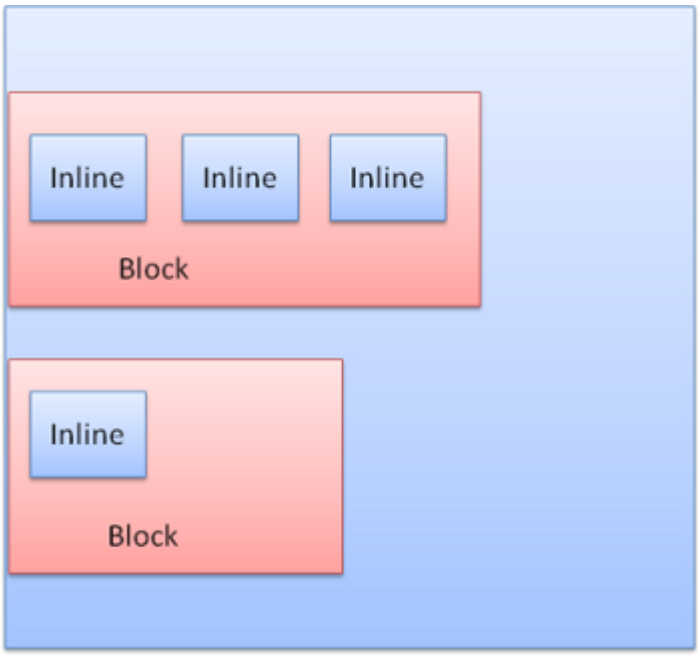
Block box：构成一个块，即在浏览器窗口上有自己的矩形



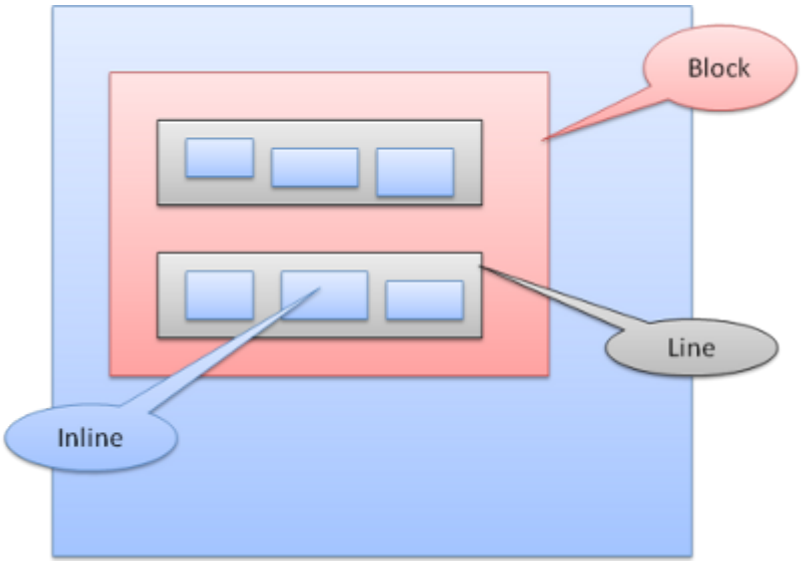
Inline box：并没有自己的块状区域，但包含在一个块状区域内



block一个挨着一个垂直格式化，inline则在水平方向上格式化。



Inline盒模型放置在行内或是line box中，每行至少和最高的box一样高，当box以baseline对齐时——即一个元素的底部和另一个box上除底部以外的某点对齐，行高可以比最高的box高。当容器宽度不够时，行内元素将被放到多行中，这在一个p元素中经常发生。

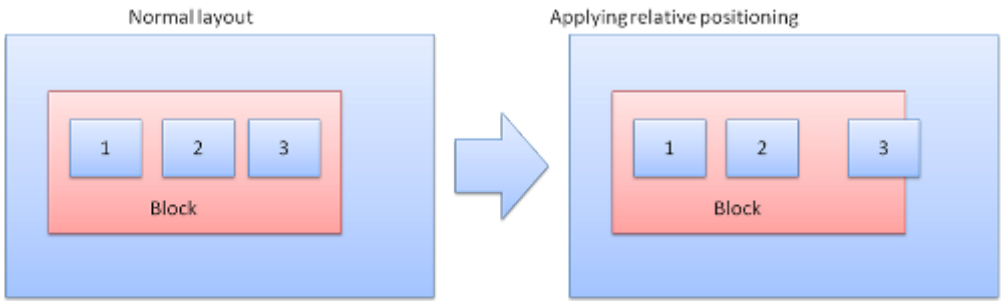


定位Position

Relative

相对定位——先按照一般的定位，然后按所要求的差值移动。

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

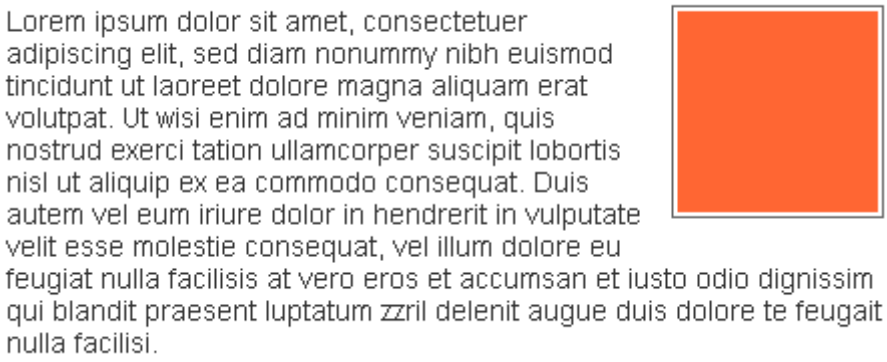


Floats

一个浮动的box移动到一行的最左边或是最右边，其余的box围绕在它周围。下面这段html：

```
<p>
<img src="images/image.gif" width="100" height="100">Lorem ipsum dolor sit amet, consec
tetuer...
</p>
```

将显示为：



Absolute和Fixed

这种情况下的布局完全不顾普通的文档流，元素不属于文档流的一部分，大小取决于容器。Fixed时，容器为viewport（可视区域）。

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

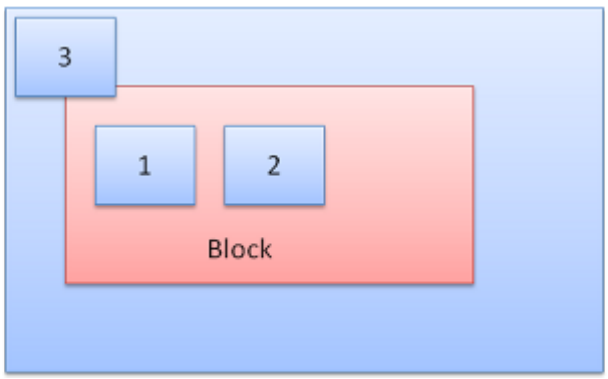


图17：fixed

注意－fixed即使在文档流滚动时也不会移动。

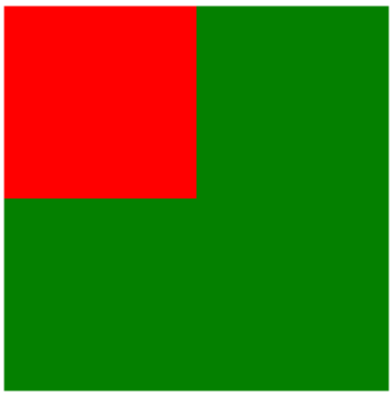
Layered representation

这个由CSS属性中的z-index指定，表示盒模型的第三个大小，即在z轴上的位置。Box分发到堆栈中（称为堆栈上下文），每个堆栈中靠后的元素将被较早绘制，栈顶靠前的元素离用户最近，当发生交叠时，将隐藏靠后的元素。堆栈根据z-index属性排序，拥有z-index属性的box形成了一个局部堆栈，viewport有外部堆栈，例如：

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

```
</STYLE>
<P>
<DIV
>
</DIV>
<DIV
>
</DIV>
</p>
```

结果是：



虽然绿色div排在红色div后面，可能在正常流中也已经被绘制在后面，但z-index有更高优先级，所以在根box的堆栈中更靠前。