浏览器的工作原理：新式网络浏览器幕后揭秘


By Tali Garsiel and Paul Irish

已发布**:** 八月 5th, 2011

Comments: 132

序言

这是一篇全面介绍 WebKit 和 Gecko 内部操作的入门文章，是以色列开发人员塔利·加希尔大量研究的成果。在过去的几年中，她查阅了所有公开发布的关于浏览器内部机制的数据（请参见资源），并花了很多时间来研读网络浏览器的源代码。她写道：

> 在 *IE* 占据 *90%* 市场份额的年代，我们除了把浏览器当成一个“黑箱”，什么也做不了。但是现在，开放源代码的浏览器拥有了过半的市场份额，因此，是时候来揭开神秘的面纱，一探网络浏览器的内幕了。呃，里面只有数以百万行计的 *C++* 代码...

塔利在她的网站上公布了自己的研究成果，但是我们觉得它值得让更多的人来了解，所以我们在此重新整理并公布。

作为一名网络开发人员，学习浏览器的内部工作原理将有助于您作出更明智的决策，并理解那些最佳开发实践的个中缘由。尽管这是一篇相当长的文档，但是我们建议您花些时间来仔细阅读；读完之后，您肯定会觉得所费不虚。

保罗·爱丽诗 *(Paul Irish)*，*Chrome* 浏览器开发人员事务部

简介

网络浏览器很可能是使用最广的软件。在这篇入门文章中，我将会介绍它们的幕后工作原理。我们会了解到，从您在地址栏输入 `google.com` 直到您在浏览器屏幕上看到 Google 首页的整个过程中都发生了些什么。

目录

我们要讨论的浏览器

目前使用的主流浏览器有五个：Internet Explorer、Firefox、Safari、Chrome 浏览器和 Opera。本文中以开放源代码浏览器为例，即 Firefox、Chrome 浏览器和 Safari（部分开源）。根据 StatCounter 浏览器统计数据，目前（2011 年 8 月）Firefox、Safari 和 Chrome 浏览器的总市场占有率将近 60%。由此可见，如今开放源代码浏览器在浏览器市场中占据了非常坚实的部分。

浏览器的主要功能

浏览器的主要功能就是向服务器发出请求，在浏览器窗口中展示您选择的网络资源。这里所说的资源一般是指 HTML 文档，也可以是 PDF、图片或其他的类型。资源的位置由用户使用 URI（统一资源标示符）指定。

浏览器解释并显示 HTML 文件的方式是在 HTML 和 CSS 规范中指定的。这些规范由网络标准化组织 W3C（万维网联盟）进行维护。
多年以来，各浏览器都没有完全遵从这些规范，同时还在开发自己独有的扩展程序，这给网络开发人员带来了严重的兼容性问题。如今，大多数的浏览器都是或多或少地遵从规范。

浏览器的用户界面有很多彼此相同的元素，其中包括：

- 用来输入 URI 的地址栏
- 前进和后退按钮
- 书签设置选项
- 用于刷新和停止加载当前文档的刷新和停止按钮
- 用于返回主页的主页按钮

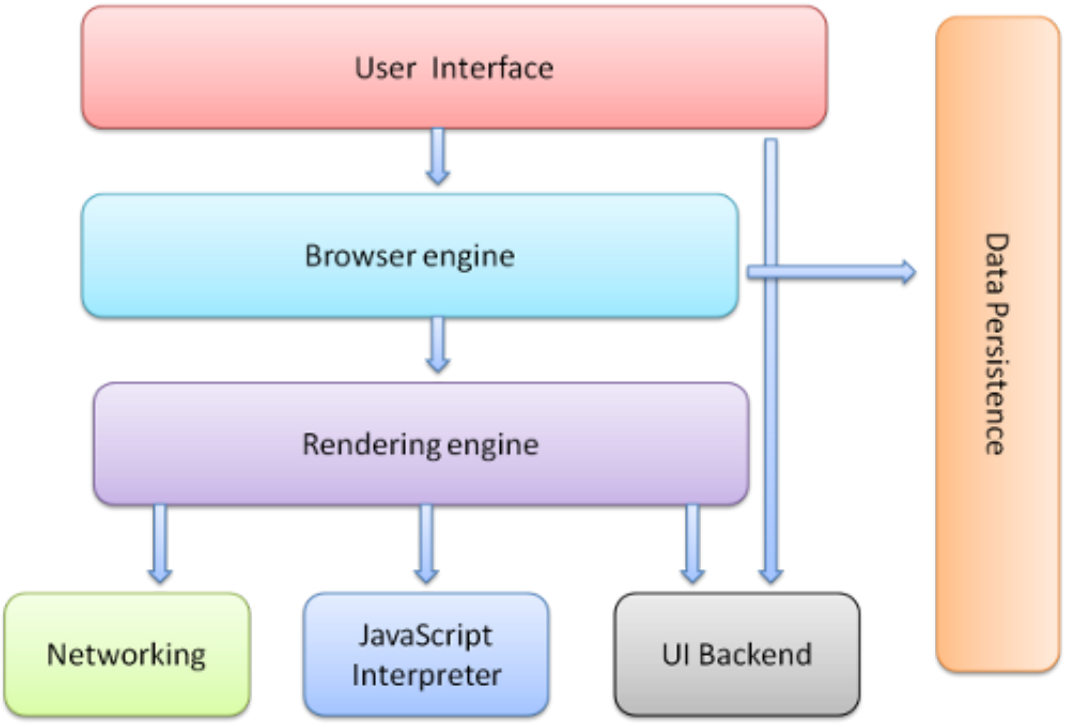
奇怪的是，浏览器的用户界面并没有任何正式的规范，这是多年来的最佳实践自然发展以及彼此之间相互模仿的结果。HTML5 也没有定义浏览器必须具有的用户界面元素，但列出了一些通用的元素，例如地址栏、状态栏和工具栏等。当然，各浏览器也可以有自己独特的功能，比如 Firefox 的下载管理器。

浏览器的高层结构

浏览器的主要组件为 (1.1)：

1. 用户界面 - 包括地址栏、前进/后退按钮、书签菜单等。除了浏览器主窗口显示的您请求的页面外，其他显示的各个部分都属于用户界面。
2. 浏览器引擎 - 在用户界面和呈现引擎之间传送指令。
3. 呈现引擎 - 负责显示请求的内容。如果请求的内容是 HTML，它就负责解析 HTML 和 CSS 内容，并将解析后的内容显示在屏幕上。

4. 网络 - 用于网络调用，比如 HTTP 请求。其接口与平台无关，并为所有平台提供底层实现。
5. 用户界面后端 - 用于绘制基本的窗口小部件，比如组合框和窗口。其公开了与平台无关的通用接口，而在底层使用操作系统的用户界面方法。
6. **JavaScript** 解释器。用于解析和执行 JavaScript 代码。
7. 数据存储。这是持久层。浏览器需要在硬盘上保存各种数据，例如 Cookie。新的 HTML 规范 (HTML5) 定义了“网络数据库”，这是一个完整（但是轻便）的浏览器内数据库。



图：浏览器的主要组件。

值得注意的是，和大多数浏览器不同，Chrome 浏览器的每个标签页都分别对应一个呈现引擎实例。每个标签页都是一个独立的进程。

呈现引擎

呈现引擎的作用嘛...当然就是“呈现”了，也就是在浏览器的屏幕上显示请求的内容。

默认情况下，呈现引擎可显示 HTML 和 XML 文档与图片。通过插件（或浏览器扩展程序），还可以显示其他类型的内容；例如，使用 PDF 查看器插件就能显示 PDF 文档。但是在本章中，我们将集中介绍其主要用途：显示使用 CSS 格式化的 HTML 内容和图片。

呈现引擎

本文所讨论的浏览器（Firefox、Chrome 浏览器和 Safari）是基于两种呈现引擎构

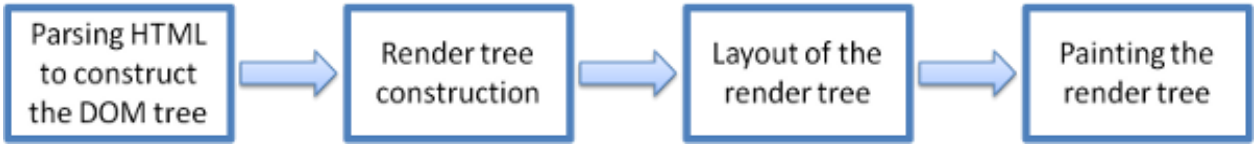
建的。Firefox 使用的是 Gecko，这是 Mozilla 公司“自制”的呈现引擎。而 Safari 和 Chrome 浏览器使用的都是 WebKit。

WebKit 是一种开放源代码呈现引擎，起初用于 Linux 平台，随后由 Apple 公司进行修改，从而支持苹果机和 Windows。有关详情，请参阅 webkit.org。

主流程

呈现引擎一开始会从网络层获取请求文档的内容，内容的大小一般限制在 8000 个块以内。

然后进行如下所示的基本流程：



图：呈现引擎的基本流程。

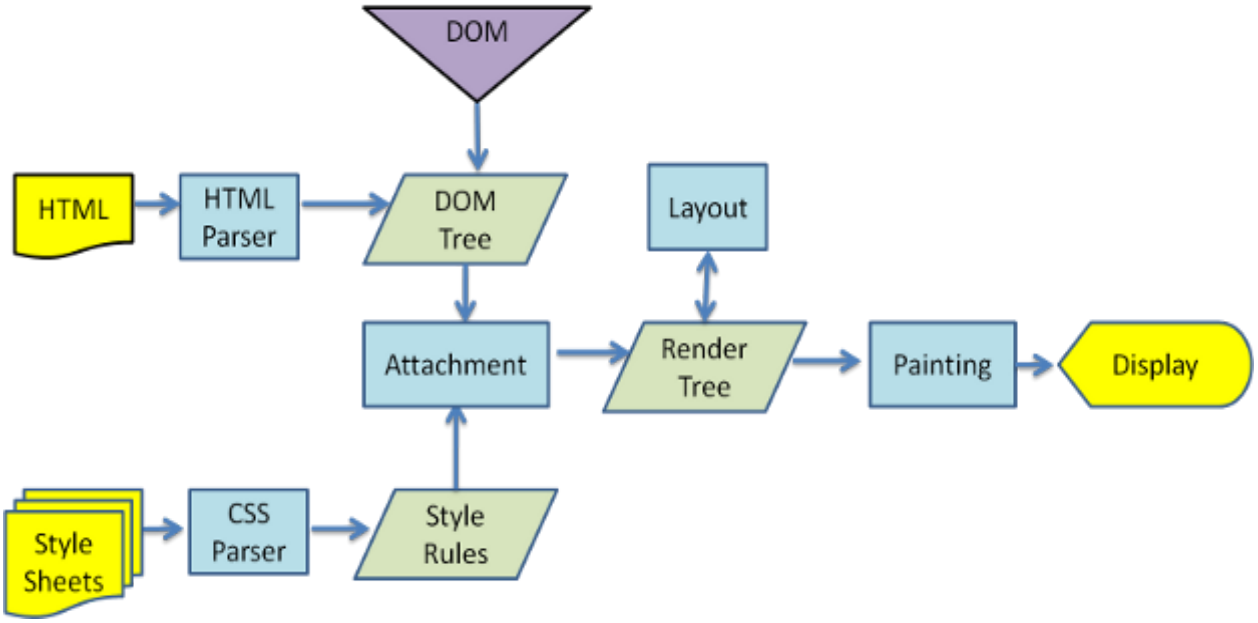
呈现引擎将开始解析 HTML 文档，并将各标记逐个转化成“内容树”上的 DOM 节点。同时也会解析外部 CSS 文件以及样式元素中的样式数据。HTML 中这些带有视觉指令的样式信息将用于创建另一个树结构：呈现树。

呈现树包含多个带有视觉属性（如颜色和尺寸）的矩形。这些矩形的排列顺序就是它们将在屏幕上显示的顺序。

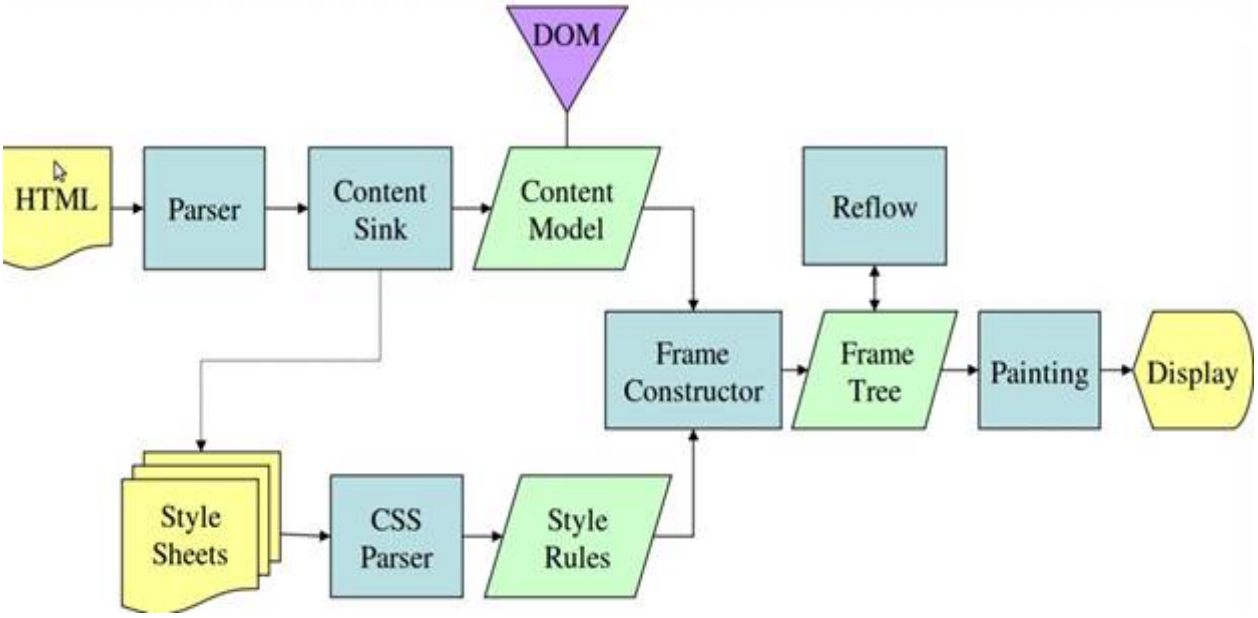
呈现树构建完毕之后，进入“布局”处理阶段，也就是为每个节点分配一个应出现在屏幕上的确切坐标。下一个阶段是绘制 - 呈现引擎会遍历呈现树，由用户界面后端层将每个节点绘制出来。

需要着重指出的是，这是一个渐进的过程。为达到更好的用户体验，呈现引擎会力求尽快将内容显示在屏幕上。它不必等到整个 HTML 文档解析完毕之后，就会开始构建呈现树和设置布局。在不断接收和处理来自网络的其余内容的同时，呈现引擎会将部分内容解析并显示出来。

主流程示例



图：***WebKit*** 主流程



图：***Mozilla*** 的 ***Gecko*** 呈现引擎主流程 ***(3.6)***

从图 3 和图 4 可以看出，虽然 WebKit 和 Gecko 使用的术语略有不同，但整体流程是基本相同的。

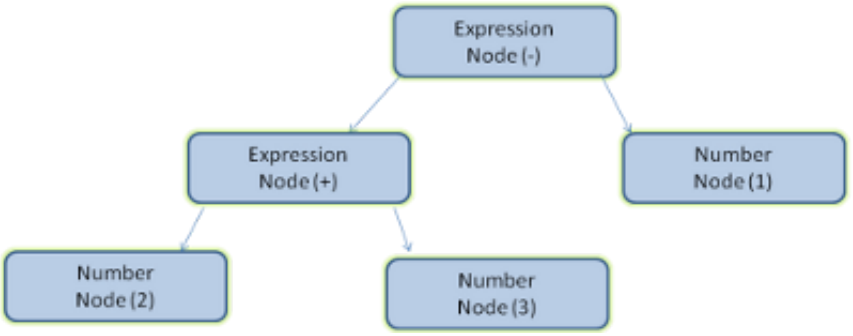
Gecko 将视觉格式化元素组成的树称为“框架树”。每个元素都是一个框架。WebKit 使用的术语是“呈现树”，它由“呈现对象”组成。对于元素的放置，WebKit 使用的术语是“布局”，而 Gecko 称之为“重排”。对于连接 DOM 节点和可视化信息从而创建呈现树的过程，WebKit 使用的术语是“附加”。有一个细微的非语义差别，就是 Gecko 在 HTML 与 DOM 树之间还有一个称为“内容槽”的层，用于生成 DOM 元素。我们会逐一论述流程中的每一部分：

解析 - 综述

解析是呈现引擎中非常重要的一个环节，因此我们要更深入地讲解。首先，来介绍一下解析。

解析文档是指将文档转化成为有意义的结构，也就是可让代码理解和使用的结构。解析得到的结果通常是代表了文档结构的节点树，它称作解析树或者语法树。

示例 - 解析 `2 + 3 - 1` 这个表达式，会返回下面的树：



图：数学表达式树节点

语法

解析是以文档所遵循的语法规则（编写文档所用的语言或格式）为基础的。所有可以解析的格式都必须对应确定的语法（由词汇和语法规则构成）。这称为与上下文无关的语法。人类语言并不属于这样的语言，因此无法用常规的解析技术进行解析。

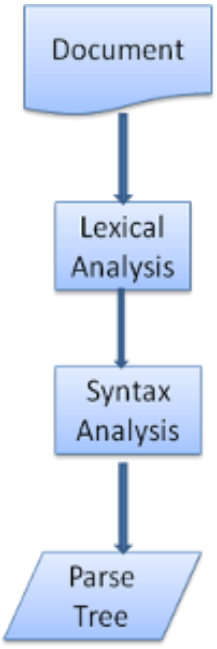
解析器和词法分析器的组合

解析的过程可以分成两个子过程：词法分析和语法分析。

词法分析是将输入内容分割成大量标记的过程。标记是语言中的词汇，即构成内容的单位。在人类语言中，它相当于语言字典中的单词。

语法分析是应用语言的语法规则的过程。

解析器通常将解析工作分给以下两个组件来处理：词法分析器（有时也称为标记生成器），负责将输入内容分解成一个个有效标记；而解析器负责根据语言的语法规则分析文档的结构，从而构建解析树。词法分析器知道如何将无关的字符（比如空格和换行符）分离出来。



图：从源文档到解析树

解析是一个迭代的过程。通常，解析器会向词法分析器请求一个新标记，并尝试将其与某条语法规则进行匹配。如果发现了匹配规则，解析器会将一个对应于该标记的节点添加到解析树中，然后继续请求下一个标记。

如果没有规则可以匹配，解析器就会将标记存储到内部，并继续请求标记，直至找到可与所有内部存储的标记匹配的规则。如果找不到任何匹配规则，解析器就会引发一个异常。这意味着文档无效，包含语法错误。

翻译

很多时候，解析树还不是最终产品。解析通常是在翻译过程中使用的，而翻译是指将输入文档转换成另一种格式。编译就是这样一个例子。编译器可将源代码编译成机器代码，具体过程是首先将源代码解析成解析树，然后将解析树翻译成机器代码文档。

图：编译流程

解析示例

在图 5 中，我们通过一个数学表达式建立了解析树。现在，让我们试着定义一个简单的数学语言，用来演示解析的过程。

词汇：我们用的语言可包含整数、加号和减号。

语法：

1. 构成语言的语法单位是表达式、项和运算符。
2. 我们用的语言可以包含任意数量的表达式。
3. 表达式的定义是：一个“项”接一个“运算符”，然后再接一个“项”。
4. 运算符是加号或减号。
5. 项是一个整数或一个表达式。

让我们分析一下 `2 + 3 - 1`。
匹配语法规则的第一个子串是 `2`，而根据第 5 条语法规则，这是一个项。匹配语法规则的第二个子串是 `2 + 3`，而根据第 3 条规则（一个项接一个运算符，然后再接一个项），这是一个表达式。下一个匹配项已经到了输入的结束。`2 + 3 - 1` 是一个表达式，因为我们已经知道 `2 + 3` 是一个项，这样就符合“一个项接一个运算符，然后再接一个项”的规则。`2 + +` 不与任何规则匹配，因此是无效的输入。

词汇和语法的正式定义

词汇通常用正则表达式表示。

例如，我们的示例语言可以定义如下：

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

正如您所看到的，这里用正则表达式给出了整数的定义。

语法通常使用一种称为 BNF 的格式来定义。我们的示例语言可以定义如下：

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

之前我们说过，如果语言的语法是与上下文无关的语法，就可以由常规解析器进行解析。与上下文无关的语法的直观定义就是可以完全用 BNF 格式表达的语法。有关正式定义，请参阅关于与上下文无关的语法的维基百科文章。

解析器类型

有两种基本类型的解析器：自上而下解析器和自下而上解析器。直观地来说，自上而下的解析器从语法的高层结构出发，尝试从中找到匹配的结构。而自下而上的解析器从低层规则出发，将输入内容逐步转化为语法规则，直至满足高层规则。

让我们来看看这两种解析器会如何解析我们的示例：

自上而下的解析器会从高层的规则开始：首先将 `2 + 3` 标识为一个表达式，然后将 `2 + 3 - 1` 标识为一个表达式（标识表达式的过程涉及到匹配其他规则，但是起点是最高级别的规则）。

自下而上的解析器将扫描输入内容，找到匹配的规则后，将匹配的输入内容替换成规则。如此继续替换，直到输入内容的结尾。部分匹配的表达式保存在解析器的堆栈中。

堆栈	输入
	`2 + 3 - 1`
项	`+ 3 - 1`
项运算	`3 - 1`

表达式	`- 1`
表达式运算符	`1`
表达式	

这种自下而上的解析器称为移位归约解析器，因为输入在向右移位（设想有一个指针从输入内容的开头移动到结尾），并且逐渐归约到语法规则上。

自动生成解析器

有一些工具可以帮助您生成解析器，它们称为解析器生成器。您只要向其提供您所用语言的语法（词汇和语法规则），它就会生成相应的解析器。创建解析器需要对解析有深刻理解，而人工创建并优化解析器并不是一件容易的事情，所以解析器生成器是非常实用的。

WebKit 使用了两种非常有名的解析器生成器：用于创建词法分析器的 Flex 以及用于创建解析器的 Bison（您也可能遇到 Lex 和 Yacc 这样的别名）。Flex 的输入是包含标记的正则表达式定义的文件。Bison 的输入是采用 BNF 格式的语言语法规则。

HTML 解析器

HTML 解析器的任务是将 HTML 标记解析成解析树。

HTML 语法定义

HTML 的词汇和语法在 W3C 组织创建的规范中进行了定义。当前的版本是 HTML4，HTML5 正在处理过程中。

非与上下文无关的语法

正如我们在解析过程的简介中已经了解到的，语法可以用 BNF 等格式进行正式定义。

很遗憾，所有的常规解析器都不适用于 HTML（我并不是开玩笑，它们可以用于解析 CSS 和 JavaScript）。HTML 并不能很容易地用解析器所需的与上下文无关的语法来定义。

有一种可以定义 HTML 的正规格式：DTD（Document Type Definition，文档类型定义），但它不是与上下文无关的语法。

这初看起来很奇怪：HTML 和 XML 非常相似。有很多 XML 解析器可以使用。HTML 存在一个 XML 变体 (XHTML)，那么有什么大的区别呢？

区别在于 HTML 的处理更为“宽容”，它允许您省略某些隐式添加的标记，有时还能

省略一些起始或者结束标记等等。和 XML 严格的语法不同，HTML 整体来看是一种“软性”的语法。

显然，这种看上去细微的差别实际上却带来了巨大的影响。一方面，这是 HTML 如此流行的原因：它能包容您的错误，简化网络开发。另一方面，这使得它很难编写正式的语法。概括地说，HTML 无法很容易地通过常规解析器解析（因为它的语法不是与上下文无关的语法），也无法通过 XML 解析器来解析。

HTML DTD

HTML 的定义采用了 DTD 格式。此格式可用于定义 SGML 族的语言。它包括所有允许使用的元素及其属性和层次结构的定义。如上文所述，HTML DTD 无法构成与上下文无关的语法。

DTD 存在一些变体。严格模式完全遵守 HTML 规范，而其他模式可支持以前的浏览器所使用的标记。这样做的目的是确保向下兼容一些早期版本的内容。最新的严格模式 DTD 可以在这里找到：www.w3.org/TR/html4/strict.dtd

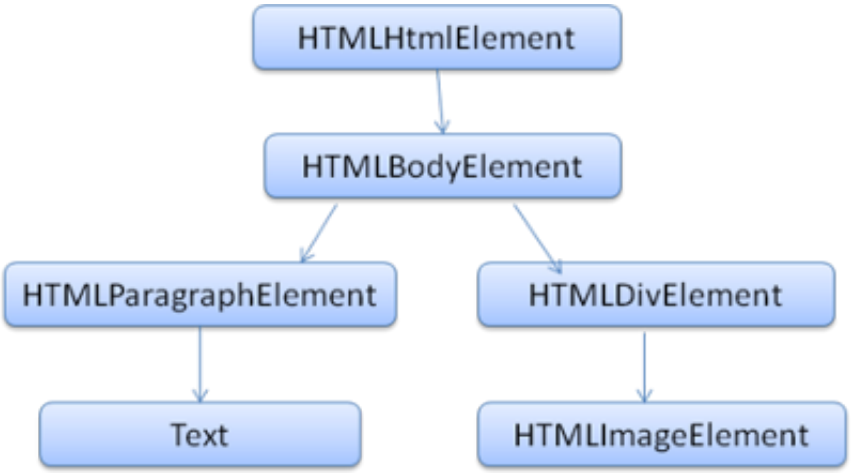
DOM

解析器的输出“解析树”是由 DOM 元素和属性节点构成的树结构。DOM 是文档对象模型 (Document Object Model) 的缩写。它是 HTML 文档的对象表示，同时也是外部内容（例如 JavaScript）与 HTML 元素之间的接口。
解析树的根节点是“Document”对象。

DOM 与标记之间几乎是一一对应的关系。比如下面这段标记：

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

可翻译成如下的 DOM 树：



图：示例标记的 ***DOM*** 树

和 HTML 一样，DOM 也是由 W3C 组织指定的。请参见 www.w3.org/DOM/DOMTR。这是关于文档操作的通用规范。其中一个特定模块描述针对 HTML 的元素。HTML 的定义可以在这里找到：www.w3.org/TR/2003/REC-DOM-Level-2-HTML-20030109/idl-definitions.html。

我所说的树包含 DOM 节点，指的是树是由实现了某个 DOM 接口的元素构成的。浏览器在具体的实现中会有一些供内部使用的其他属性。

解析算法

我们在之前章节已经说过，HTML 无法用常规的自上而下或自下而上的解析器进行解析。

原因在于：

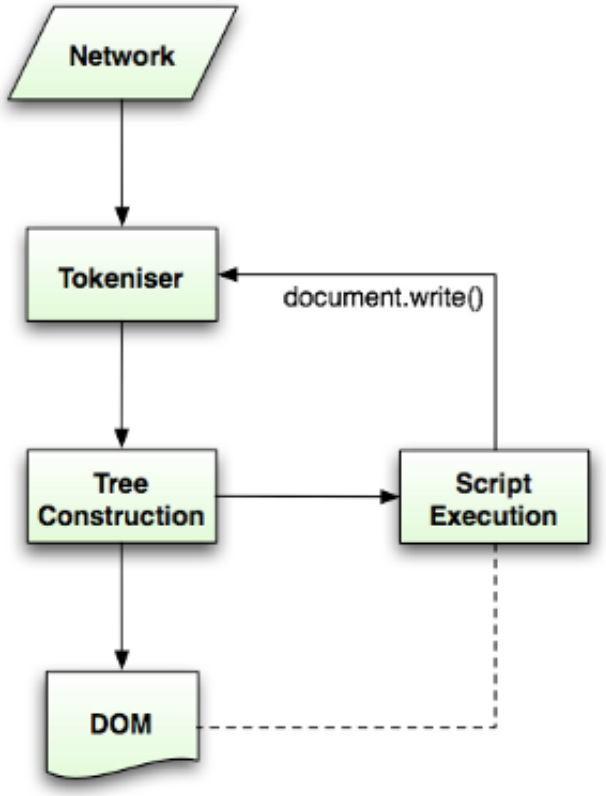
1. 语言的宽容本质。
2. 浏览器历来对一些常见的无效 HTML 用法采取包容态度。
3. 解析过程需要不断地反复。源内容在解析过程中通常不会改变，但是在 HTML 中，脚本标记如果包含 `document.write`，就会添加额外的标记，这样解析过程实际上就更改了输入内容。

由于不能使用常规的解析技术，浏览器就创建了自定义的解析器来解析 HTML。

HTML5 规范详细地描述了解析算法。此算法由两个阶段组成：标记化和树构建。

标记化是词法分析过程，将输入内容解析成多个标记。HTML 标记包括起始标记、结束标记、属性名称和属性值。

标记生成器识别标记，传递给树构造器，然后接受下一个字符以识别下一个标记；如此反复直到输入的结束。



图：***HTML*** 解析流程（摘自 ***HTML5*** 规范）

标记化算法

该算法的输出结果是 HTML 标记。该算法使用状态机来表示。每一个状态接收来自输入信息流的一个或多个字符，并根据这些字符更新下一个状态。当前的标记化状态和树结构状态会影响进入下一状态的决定。这意味着，即使接收的字符相同，对于下一个正确的状态也会产生不同的结果，具体取决于当前的状态。该算法相当复杂，无法在此详述，所以我们通过一个简单的示例来帮助大家理解其原理。

基本示例 - 将下面的 HTML 代码标记化：

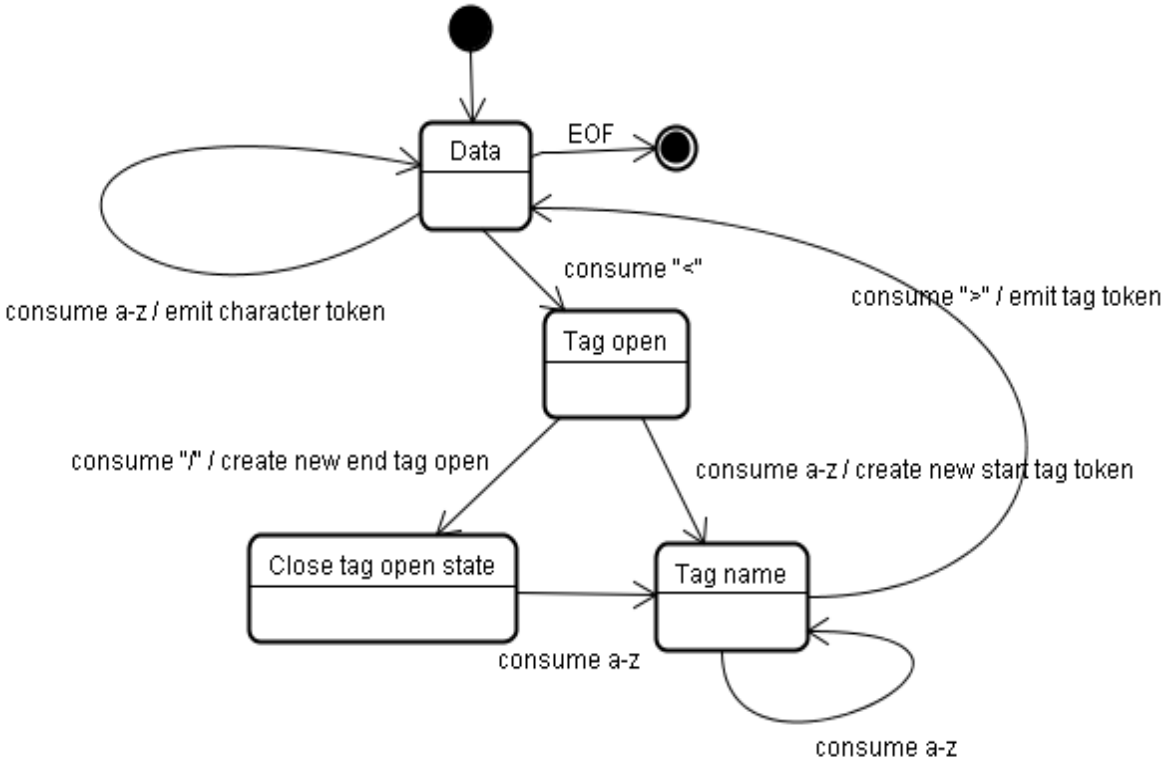
```
<html>
  <body>
    Hello world
  </body>
</html>
```

初始状态是数据状态。遇到字符 < 时，状态更改为“标记打开状态”。接收一个 a-z 字符会创建“起始标记”，状态更改为“标记名称状态”。这个状态会一直保持到接收 > 字符。在此期间接收的每个字符都会附加到新的标记名称上。在本例中，我们创建的标记是 `html` 标记。

遇到 > 标记时，会发送当前的标记，状态改回“数据状态”。<body> 标记也会进行

同样的处理。目前 `html` 和 `body` 标记均已发出。现在我们回到“数据状态”。接收到 `Hello world` 中的 `H` 字符时，将创建并发送字符标记，直到接收 `</body>` 中的 `<`。我们将为 `Hello world` 中的每个字符都发送一个字符标记。

现在我们回到“标记打开状态”。接收下一个输入字符 `/` 时，会创建 `end tag token` 并改为“标记名称状态”。我们会再次保持这个状态，直到接收 `>`。然后将发送新的标记，并回到“数据状态”。`</html>` 输入也会进行同样的处理。



图：对示例输入进行标记化

树构建算法

在创建解析器的同时，也会创建 Document 对象。在树构建阶段，以 Document 为根节点的 DOM 树也会不断进行修改，向其中添加各种元素。标记生成器发送的每个节点都会由树构建器进行处理。规范中定义了每个标记所对应的 DOM 元素，这些元素会在接收到相应的标记时创建。这些元素不仅会添加到 DOM 树中，还会添加到开放元素的堆栈中。此堆栈用于纠正嵌套错误和处理未关闭的标记。其算法也可以用状态机来描述。这些状态称为“插入模式”。

让我们来看看示例输入的树构建过程：

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

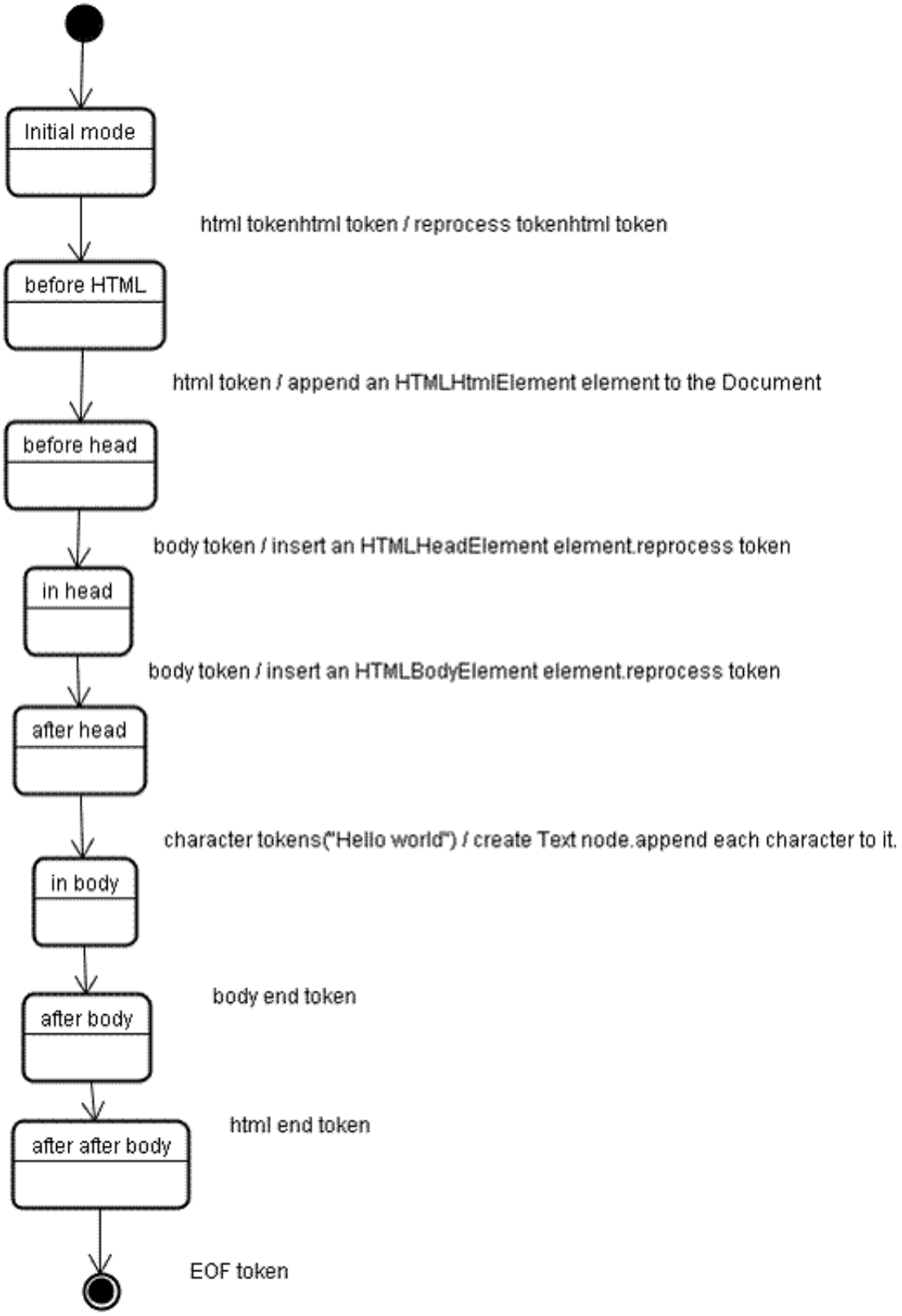
树构建阶段的输入是一个来自标记化阶段的标记序列。第一个模式是**“initial mode”**。接收 HTML 标记后转为**“before html”**模式，并在这个模式下重新处理此标记。这样会创建一个 HTMLHtmlElement 元素，并将其附加到 Document 根对象上。

然后状态将改为**“before head”**。此时我们接收“body”标记。即使我们的示例中没有“head”标记，系统也会隐式创建一个 HTMLHeadElement，并将其添加到树中。

现在我们进入了**“in head”**模式，然后转入**“after head”**模式。系统对 body 标记进行重新处理，创建并插入 HTMLBodyElement，同时模式转变为**“in body”**。

现在，接收由“Hello world”字符串生成的一系列字符标记。接收第一个字符时会创建并插入“Text”节点，而其他字符也将附加到该节点。

接收 body 结束标记会触发**“after body”**模式。现在我们将接收 HTML 结束标记，然后进入**“after after body”**模式。接收到文件结束标记后，解析过程就此结束。



图：示例 ***HTML*** 的树构建

解析结束后的操作

在此阶段，浏览器会将文档标注为交互状态，并开始解析那些处于“deferred”模式的脚本，也就是那些应在文档解析完成后才执行的脚本。然后，文档状态将设置为“完成”，一个“加载”事件将随之触发。

您可以在 HTML5 规范中查看标记化和树构建的完整算法

浏览器的容错机制

您在浏览 HTML 网页时从来不会看到“语法无效”的错误。这是因为浏览器会纠正任何无效内容，然后继续工作。

以下面的 HTML 代码为例：

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

在这里，我已经违反了很多语法规则（“mytag”不是标准的标记，“p”和“div”元素之间的嵌套有误等等），但是浏览器仍然会正确地显示这些内容，并且毫无怨言。因为有大量的解析器代码会纠正 HTML 网页作者的错误。

不同浏览器的错误处理机制相当一致，但令人称奇的是，这种机制并不是 HTML 当前规范的一部分。和书签管理以及前进/后退按钮一样，它也是浏览器在多年发展中的产物。很多网站都普遍存在着一些已知的无效 HTML 结构，每一种浏览器都会尝试通过和其他浏览器一样的方式来修复这些无效结构。

HTML5 规范定义了一部分这样的要求。WebKit 在 HTML 解析器类的开头注释中对此做了很好的概括。

> 解析器对标记化输入内容进行解析，以构建文档树。如果文档的格式正确，就直接进行解析。
>
> 遗憾的是，我们不得不处理很多格式错误的 *HTML* 文档，所以解析器必须具备一定的容错性。
>
> 我们至少要能够处理以下错误情况：
>
> 1. 明显不能在某些外部标记中添加的元素。在此情况下，我们应该关闭所有标记，直到出现禁止添加的元素，然后再加入该元素。
> 2. 我们不能直接添加的元素。这很可能是网页作者忘记添加了其中的一些标记（或者其中的标记是可选的）。这些标签可能包括：*HTML HEAD BODY TBODY TR TD LI*（还有遗漏的吗？）。

3. 向 *inline* 元素内添加 *block* 元素。关闭所有 *inline* 元素，直到出现下一个较高级的 *block* 元素。
4. 如果这样仍然无效，可关闭所有元素，直到可以添加元素为止，或者忽略该标记。

让我们看一些 WebKit 容错的示例：

使用了 </br> 而不是

有些网站使用了 </br> 而不是
。为了与 IE 和 Firefox 兼容，WebKit 将其与
 做同样的处理。
代码如下：

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

请注意，错误处理是在内部进行的，用户并不会看到这个过程。

离散表格

离散表格是指位于其他表格内容中，但又不在任何一个单元格内的表格。
比如以下的示例：

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

WebKit 会将其层次结构更改为两个同级表格：

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

代码如下：

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

WebKit 使用一个堆栈来保存当前的元素内容，它会从外部表格的堆栈中弹出内部表格。现在，这两个表格就变成了同级关系。

嵌套的表单元素

如果用户在一个表单元素中又放入了另一个表单，那么第二个表单将被忽略。
代码如下：

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

过于复杂的标记层次结构

代码的注释已经说得很清楚了。

> 示例网站 *www.liceo.edu.mx* 嵌套了约 *1500* 个标记，全都来自一堆 <b> 标记。我们只允许最多 *20* 层同类型标记的嵌套，如果再嵌套更多，就会全部忽略。

```
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;
}
```

放错位置的 html 或者 body 结束标记

同样，代码的注释已经说得很清楚了。

> 支持格式非常糟糕的 *HTML* 代码。我们从不关闭 *body* 标记，因为一些愚蠢的网页会在实际文档结束之前就关闭。我们通过调用 *end()* 来执行关闭操作。

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

所以网页作者需要注意，除非您想作为反面教材出现在 WebKit 容错代码段的示例中，否则还请编写格式正确的 HTML 代码。

CSS 解析

还记得简介中解析的概念吗？和 HTML 不同，CSS 是上下文无关的语法，可以使用简介中描述的各种解析器进行解析。事实上，CSS 规范定义了 CSS 的词法和语法。

让我们来看一些示例：
词法语法（词汇）是针对各个标记用正则表达式定义的：

```
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 (identifier) 的缩写，比如类名。“name”是元素的 ID（通过“#”来引用）。

语法是采用 BNF 格式描述的。

```
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*] ')' ]
  ;
```

解释：这是一个规则集的结构：

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

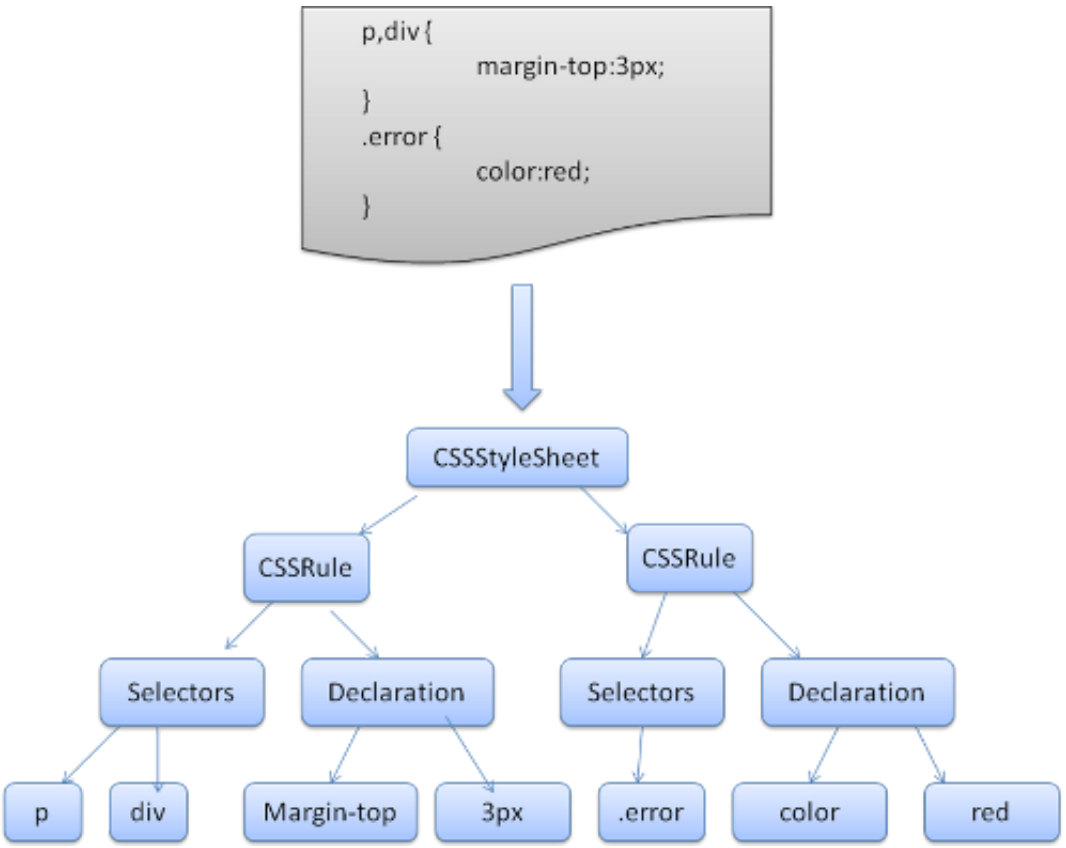
div.error 和 a.error 是选择器。大括号内的部分包含了由此规则集应用的规则。此结构的正式定义是这样的：

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

这表示一个规则集就是一个选择器，或者由逗号和空格（S 表示空格）分隔的多个（数量可选）选择器。规则集包含了大括号，以及其中的一个或多个（数量可选）由分号分隔的声明。“声明”和“选择器”将由下面的 BNF 格式定义。

WebKit CSS 解析器

WebKit 使用 Flex 和 Bison 解析器生成器，通过 CSS 语法文件自动创建解析器。正如我们之前在解析器简介中所说，Bison 会创建自下而上的移位归约解析器。Firefox 使用的是人工编写的自上而下的解析器。这两种解析器都会将 CSS 文件解析成 StyleSheet 对象，且每个对象都包含 CSS 规则。CSS 规则对象则包含选择器和声明对象，以及其他与 CSS 语法对应的对象。



图：解析 ***CSS***

处理脚本和样式表的顺序

脚本

网络的模型是同步的。网页作者希望解析器遇到 <script> 标记时立即解析并执行脚本。文档的解析将停止，直到脚本执行完毕。如果脚本是外部的，那么解析过程会停止，直到从网络同步抓取资源完成后再继续。此模型已经使用了多年，也在 HTML4 和 HTML5 规范中进行了指定。作者也可以将脚本标注为“defer”，这样它就不会停止文档解析，而是等到解析结束才执行。HTML5 增加了一个选项，可将脚本标记为异步，以便由其他线程解析和执行。

预解析

WebKit 和 Firefox 都进行了这项优化。在执行脚本时，其他线程会解析文档的其余部分，找出并加载需要通过网络加载的其他资源。通过这种方式，资源可以在并行连接上加载，从而提高总体速度。请注意，预解析器不会修改 DOM 树，而是将这项工作交由主解析器处理；预解析器只会解析外部资源（例如外部脚本、样式表和图片）的引用。

样式表

另一方面，样式表有着不同的模型。理论上来说，应用样式表不会更改 DOM 树，

因此似乎没有必要等待样式表并停止文档解析。但这涉及到一个问题，就是脚本在文档解析阶段会请求样式信息。如果当时还没有加载和解析样式，脚本就会获得错误的回复，这样显然会产生很多问题。这看上去是一个非典型案例，但事实上非常普遍。Firefox 在样式表加载和解析的过程中，会禁止所有脚本。而对于 WebKit 而言，仅当脚本尝试访问的样式属性可能受尚未加载的样式表影响时，它才会禁止该脚本。

呈现树构建

在 DOM 树构建的同时，浏览器还会构建另一个树结构：呈现树。这是由可视化元素按照其显示顺序而组成的树，也是文档的可视化表示。它的作用是让您按照正确的顺序绘制内容。

Firefox 将呈现树中的元素称为“框架”。WebKit 使用的术语是呈现器或呈现对象。
呈现器知道如何布局并将自身及其子元素绘制出来。
WebKits RenderObject 类是所有呈现器的基类，其定义如下：

```
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
}
```

每一个呈现器都代表了一个矩形的区域，通常对应于相关节点的 CSS 框，这一点在 CSS2 规范中有所描述。它包含诸如宽度、高度和位置等几何信息。
框的类型会受到与节点相关的“display”样式属性的影响（请参阅样式计算章节）。
下面这段 WebKit 代码描述了根据 display 属性的不同，针对同一个 DOM 节点应创建什么类型的呈现器。

```
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;
}
```

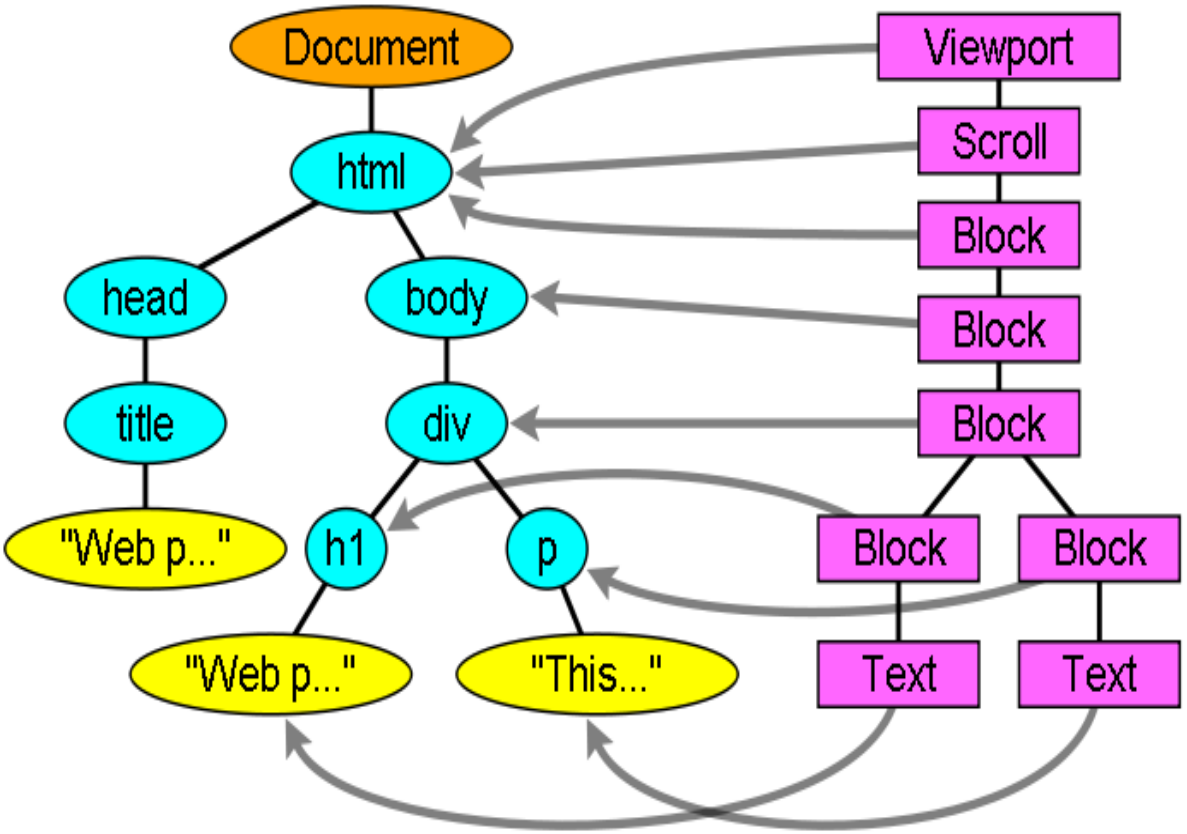
元素类型也是考虑因素之一，例如表单控件和表格都对应特殊的框架。
在 WebKit 中，如果一个元素需要创建特殊的呈现器，就会替换 `createRenderer` 方法。呈现器所指向的样式对象中包含了一些和几何无关的信息。

呈现树和 DOM 树的关系

呈现器是和 DOM 元素相对应的，但并非一一对应。非可视化的 DOM 元素不会插入呈现树中，例如“head”元素。如果元素的 display 属性值为“none”，那么也不会显示在呈现树中（但是 visibility 属性值为“hidden”的元素仍会显示）。

有一些 DOM 元素对应多个可视化对象。它们往往是具有复杂结构的元素，无法用单一的矩形来描述。例如，“select”元素有 3 个呈现器：一个用于显示区域，一个用于下拉列表框，还有一个用于按钮。如果由于宽度不够，文本无法在一行中显示而分为多行，那么新的行也会作为新的呈现器而添加。
另一个关于多呈现器的例子是格式无效的 HTML。根据 CSS 规范，inline 元素只能包含 block 元素或 inline 元素中的一种。如果出现了混合内容，则应创建匿名的 block 呈现器，以包裹 inline 元素。

有一些呈现对象对应于 DOM 节点，但在树中所在的位置与 DOM 节点不同。浮动定位和绝对定位的元素就是这样，它们处于正常的流程之外，放置在树中的其他地方，并映射到真正的框架，而放在原位的是占位框架。



图：呈现树及其对应的 ***DOM*** 树 ***(3.1)***。初始容器 ***block*** 为***“viewport”***，而在 ***WebKit*** 中则为***“RenderView”***对象。

构建呈现树的流程

在 Firefox 中，系统会针对 DOM 更新注册展示层，作为侦听器。展示层将框架创建工作委托给 `FrameConstructor`，由该构造器解析样式（请参阅样式计算）并创建框架。

在 WebKit 中，解析样式和创建呈现器的过程称为“附加”。每个 DOM 节点都有一个“attach”方法。附加是同步进行的，将节点插入 DOM 树需要调用新的节点“attach”方法。

处理 html 和 body 标记就会构建呈现树根节点。这个根节点呈现对象对应于 CSS 规范中所说的容器 block，这是最上层的 block，包含了其他所有 block。它的尺寸就是视口，即浏览器窗口显示区域的尺寸。Firefox 称之为 `ViewPortFrame`，而 WebKit 称之为 `RenderView`。这就是文档所指向的呈现对象。呈现树的其余部分以 DOM 树节点插入的形式来构建。

请参阅关于处理模型的 CSS2 规范。

样式计算

构建呈现树时，需要计算每一个呈现对象的可视化属性。这是通过计算每个元素的样式属性来完成的。

样式包括来自各种来源的样式表、inline 样式元素和 HTML 中的可视化属性（例如“bgcolor”属性）。其中后者将经过转化以匹配 CSS 样式属性。

样式表的来源包括浏览器的默认样式表、由网页作者提供的样式表以及由浏览器用户提供的用户样式表（浏览器允许您定义自己喜欢的样式。以 Firefox 为例，用户可以将自己喜欢的样式表放在“Firefox Profile”文件夹下）。

样式计算存在以下难点：

1. 样式数据是一个超大的结构，存储了无数的样式属性，这可能造成内存问题。
2. 如果不进行优化，为每一个元素查找匹配的规则会造成性能问题。要为每一个元素遍历整个规则列表来寻找匹配规则，这是一项浩大的工程。选择器会具有很复杂的结构，这就会导致某个匹配过程一开始看起来很可能是正确的，但最终发现其实是徒劳的，必须尝试其他匹配路径。

   例如下面这个组合选择器：

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

   这意味着规则适用于作为 3 个 div 元素的子代的 。如果您要检查规则是否适用于某个指定的 元素，应选择树上的一条向上路径进行检查。您可能需要向上遍历节点树，结果发现只有两个 div，而且规则并不适用。然后，您必须尝试树中的其他路径。
3. 应用规则涉及到相当复杂的层叠规则（用于定义这些规则的层次）。

让我们来看看浏览器是如何处理这些问题的：

共享样式数据

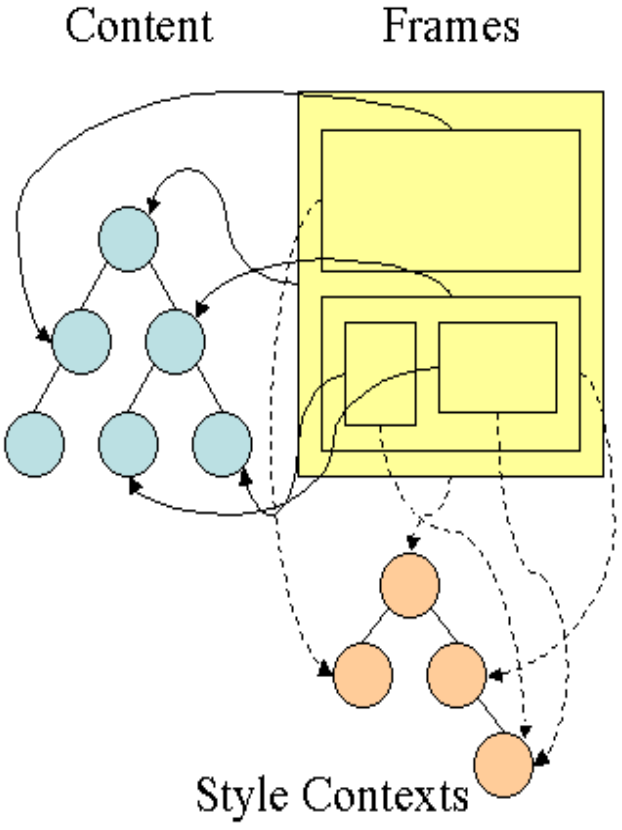
WebKit 节点会引用样式对象 (RenderStyle)。这些对象在某些情况下可以由不同节点共享。这些节点是同级关系，并且：

1. 这些元素必须处于相同的鼠标状态（例如，不允许其中一个是“:hover”状态，而另一个不是）
2. 任何元素都没有 ID
3. 标记名称应匹配
4. 类属性应匹配
5. 映射属性的集合必须是完全相同的
6. 链接状态必须匹配
7. 焦点状态必须匹配

8. 任何元素都不应受属性选择器的影响，这里所说的“影响”是指在选择器中的任何位置有任何使用了属性选择器的选择器匹配
9. 元素中不能有任何 inline 样式属性
10. 不能使用任何同级选择器。WebCore 在遇到任何同级选择器时，只会引发一个全局开关，并停用整个文档的样式共享（如果存在）。这包括 + 选择器以及 :first-child 和 :last-child 等选择器。

Firefox 规则树

为了简化样式计算，Firefox 还采用了另外两种树：规则树和样式上下文树。WebKit 也有样式对象，但它们不是保存在类似样式上下文树这样的树结构中，只是由 DOM 节点指向此类对象的相关样式。

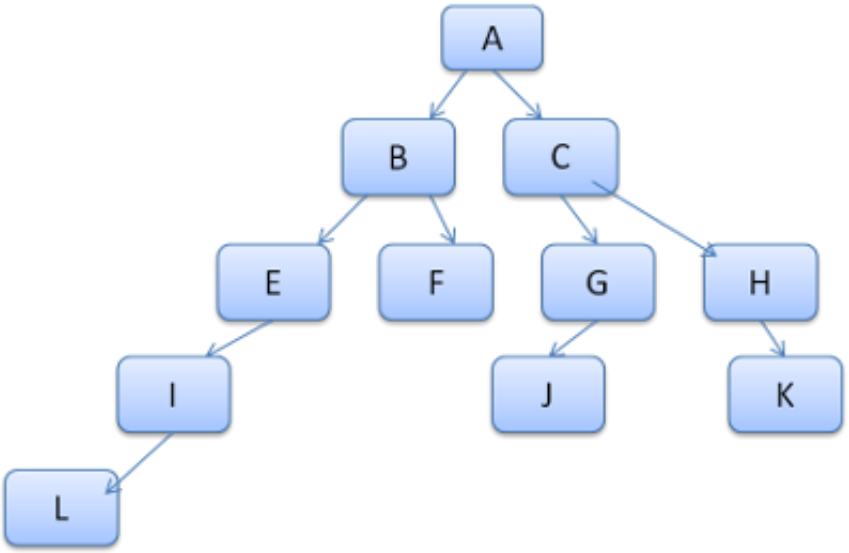


图：***Firefox*** 样式上下文树 ***(2.2)***

样式上下文包含端值。要计算出这些值，应按照正确顺序应用所有的匹配规则，并将其从逻辑值转化为具体的值。例如，如果逻辑值是屏幕大小的百分比，则需要换算成绝对的单位。规则树的点子真的很巧妙，它使得节点之间可以共享这些值，以避免重复计算，还可以节约空间。

所有匹配的规则都存储在树中。路径中的底层节点拥有较高的优先级。规则树包含了所有已知规则匹配的路径。规则的存储是延迟进行的。规则树不会在开始的时候就为所有的节点进行计算，而是只有当某个节点样式需要进行计算时，才会向规则树添加计算的路径。

这个想法相当于将规则树路径视为词典中的单词。如果我们已经计算出如下的规则树：



假设我们需要为内容树中的另一个元素匹配规则，并且找到匹配路径是 B - E - I（按照此顺序）。由于我们在树中已经计算出了路径 A - B - E - I - L，因此就已经有了此路径，这就减少了现在所需的工作量。

让我们看看规则树如何帮助我们减少工作。

结构划分

样式上下文可分割成多个结构。这些结构体包含了特定类别（如 border 或 color）的样式信息。结构中的属性都是继承的或非继承的。继承属性如果未由元素定义，则继承自其父代。非继承属性（也称为“重置”属性）如果未进行定义，则使用默认值。

规则树通过缓存整个结构（包含计算出的端值）为我们提供帮助。这一想法假定底层节点没有提供结构的定义，则可使用上层节点中的缓存结构。

使用规则树计算样式上下文

在计算某个特定元素的样式上下文时，我们首先计算规则树中的对应路径，或者使用现有的路径。然后我们沿此路径应用规则，在新的样式上下文中填充结构。我们从路径中拥有最高优先级的底层节点（通常也是最特殊的选择器）开始，并向上遍历规则树，直到结构填充完毕。如果该规则节点对于此结构没有任何规范，那么我们可以实现更好的优化：寻找路径更上层的节点，找到后指定完整的规范并指向相关节点即可。这是最好的优化方法，因为整个结构都能共享。这可以减少端值的计算量并节约内存。
如果我们找到了部分定义，就会向上遍历规则树，直到结构填充完毕。

如果我们找不到结构的任何定义，那么假如该结构是“继承”类型，我们会在上下文树中指向父代的结构，这样也可以共享结构。如果是 reset 类型的结构，则会使用

默认值。

如果最特殊的节点确实添加了值，那么我们需要另外进行一些计算，以便将这些值转化成实际值。然后我们将结果缓存在树节点中，供子代使用。

如果某个元素与其同级元素都指向同一个树节点，那么它们就可以共享整个样式上下文。

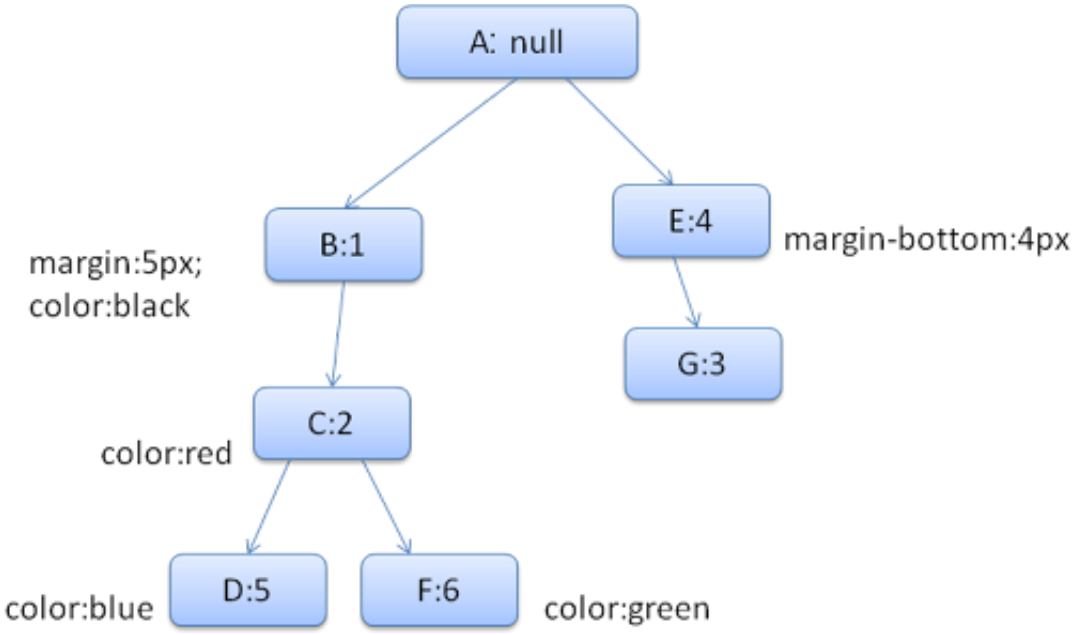
让我们来看一个例子，假设我们有如下 HTML 代码：

```
<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>
```

还有如下规则：

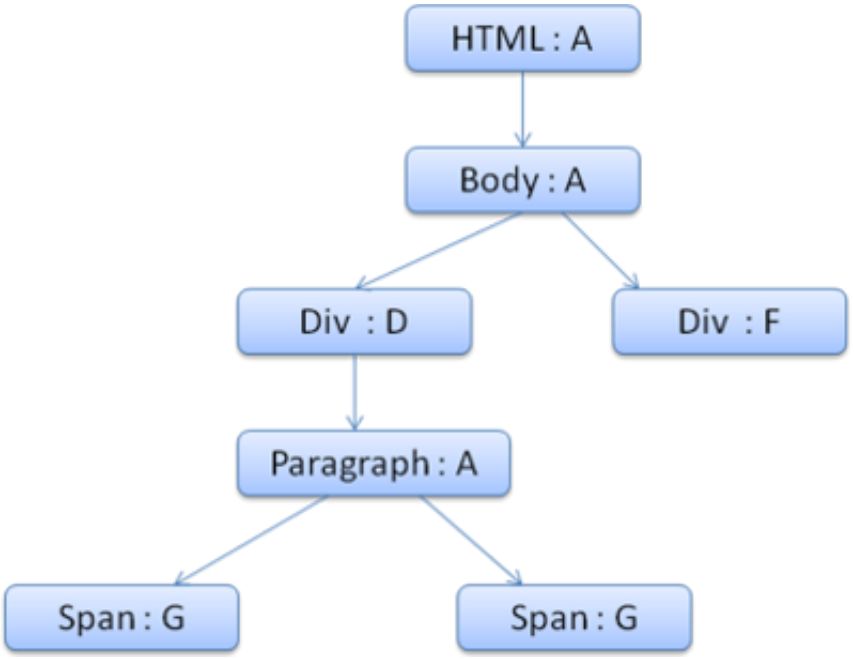
```
1. div {margin:5px;color:black}
2. .err {color:red}
3. .big {margin-top:3px}
4. div span {margin-bottom:4px}
5. #div1 {color:blue}
6. #div2 {color:green}
```

为了简便起见，我们只需要填充两个结构：color 结构和 margin 结构。color 结构只包含一个成员（即“color”），而 margin 结构包含四条边。
形成的规则树如下图所示（节点的标记方式为“节点名：指向的规则序号”）：



图：规则树

上下文树如下图所示（节点名 : 指向的规则节点）：



图：上下文树

假设我们解析 HTML 时遇到了第二个 标记，我们需要为此节点创建样式上下文，并填充其样式结构。

经过规则匹配，我们发现该 的匹配规则是第 1、2 和 6 条。这意味着规则树中已有一条路径可供我们的元素使用，我们只需要再为其添加一个节点以匹配第 6 条规则（规则树中的 F 节点）。

我们将创建样式上下文并将其放入上下文树中。新的样式上下文将指向规则树中的 F 节点。

现在我们需要填充样式结构。首先要填充的是 margin 结构。由于最后的规则节点 (F) 并没有添加到 margin 结构，我们需要上溯规则树，直至找到在先前节点插入中计算过的缓存结构，然后使用该结构。我们会在指定 margin 规则的最上层节点（即 B 节点）上找到该结构。

我们已经有了 color 结构的定义，因此不能使用缓存的结构。由于 color 有一个属性，我们无需上溯规则树以填充其他属性。我们将计算端值（将字符串转化为 RGB 等）并在此节点上缓存经过计算的结构。

第二个 元素处理起来更加简单。我们将匹配规则，最终发现它和之前的 span 一样指向规则 G。由于我们找到了指向同一节点的同级，就可以共享整个样式上下文了，只需指向之前 span 的上下文即可。

对于包含了继承自父代的规则的结构，缓存是在上下文树中进行的（事实上 color 属性是继承的，但是 Firefox 将其视为 reset 属性，并缓存到规则树上）。
例如，如果我们在某个段落中添加 font 规则：

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

那么，该段落元素作为上下文树中的 div 的子代，就会共享与其父代相同的 font 结构（前提是该段落没有指定 font 规则）。

在 WebKit 中没有规则树，因此会对匹配的声明遍历 4 次。首先应用非重要高优先级的属性（由于作为其他属性的依据而应首先应用的属性，例如 display），接着是高优先级重要规则，然后是普通优先级非重要规则，最后是普通优先级重要规则。这意味着多次出现的属性会根据正确的层叠顺序进行解析。最后出现的最终生效。

因此概括来说，共享样式对象（整个对象或者对象中的部分结构）可以解决问题 1 和问题 3。Firefox 规则树还有助于按照正确的顺序应用属性。

对规则进行处理以简化匹配

样式规则有一些来源：

- 外部样式表或样式元素中的 CSS 规则

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

- inline 样式属性及类似内容

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

- HTML 可视化属性（映射到相关的样式规则）

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

后两种很容易和元素进行匹配，因为元素拥有样式属性，而且 HTML 属性可以使用元素作为键值进行映射。

我们之前在第 2 个问题中提到过，CSS 规则匹配可能比较棘手。为了解决这一难题，可以对 CSS 规则进行一些处理，以便访问。

样式表解析完毕后，系统会根据选择器将 CSS 规则添加到某个哈希表中。这些哈希表的选择器各不相同，包括 ID、类名称、标记名称等，还有一种通用哈希表，适合不属于上述类别的规则。如果选择器是 ID，规则就会添加到 ID 表中；如果选择器是类，规则就会添加到类表中，依此类推。
这种处理可以大大简化规则匹配。我们无需查看每一条声明，只要从哈希表中提取元素的相关规则即可。这种优化方法可排除掉 95% 以上规则，因此在匹配过程中根本就不用考虑这些规则了 (4.1)。

我们以如下的样式规则为例：

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

第一条规则将插入类表，第二条将插入 ID 表，而第三条将插入标记表。
对于下面的 HTML 代码段：

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

我们首先会为 p 元素寻找匹配的规则。类表中有一个“error”键，在下面可以找到“p.error”的规则。div 元素在 ID 表（键为 ID）和标记表中有相关的规则。剩下的工作就是找出哪些根据键提取的规则是真正匹配的了。
例如，如果 div 的对应规则如下：

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

这条规则仍然会从标记表中提取出来，因为键是最右边的选择器，但这条规则并不匹配我们的 div 元素，因为 div 没有 table 祖先。

WebKit 和 Firefox 都进行了这一处理。

以正确的层叠顺序应用规则

样式对象具有与每个可视化属性一一对应的属性（均为 CSS 属性但更为通用）。如果某个属性未由任何匹配规则所定义，那么部分属性就可由父代元素样式对象继承。其他属性具有默认值。

如果定义不止一个，就会出现问题，需要通过层叠顺序来解决。

样式表层叠顺序

某个样式属性的声明可能会出现在多个样式表中，也可能在同一个样式表中出现多次。这意味着应用规则的顺序极为重要。这称为“层叠”顺序。根据 CSS2 规范，层叠的顺序为（优先级从低到高）：

1. 浏览器声明
2. 用户普通声明
3. 作者普通声明
4. 作者重要声明
5. 用户重要声明

浏览器声明是重要程度最低的，而用户只有将该声明标记为“重要”才可以替换网页作者的声明。同样顺序的声明会根据特异性进行排序，然后再是其指定顺序。
HTML 可视化属性会转换成匹配的 CSS 声明。它们被视为低优先级的网页作者规则。

特异性

选择器的特异性由 CSS2 规范定义如下：

- 如果声明来自于“style”属性，而不是带有选择器的规则，则记为 1，否则记为 0 (= a)
- 记为选择器中 ID 属性的个数 (= b)
- 记为选择器中其他属性和伪类的个数 (= c)
- 记为选择器中元素名称和伪元素的个数 (= d)

将四个数字按 a-b-c-d 这样连接起来（位于大数进制的数字系统中），构成特异性。

您使用的进制取决于上述类别中的最高计数。
例如，如果 a=14，您可以使用十六进制。如果 a=17，那么您需要使用十七进制；当然不太可能出现这种情况，除非是存在如下的选择器：html body div div p ...（在

选择器中出现了 17 个标记，这样的可能性极低）。

一些示例：

```
 *             {}  /* 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 */
```

规则排序

找到匹配的规则之后，应根据级联顺序将其排序。WebKit 对于较小的列表会使用冒泡排序，而对较大的列表则使用归并排序。对于以下规则，WebKit 通过替换“>”运算符来实现排序：

```
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;
}
```

渐进式处理

WebKit 使用一个标记来表示是否所有的顶级样式表（包括 @imports）均已加载完毕。如果在附加过程中尚未完全加载样式，则使用占位符，并在文档中进行标注，等样式表加载完毕后再重新计算。

布局

呈现器在创建完成并添加到呈现树时，并不包含位置和大小信息。计算这些值的过程称为布局或重排。

HTML 采用基于流的布局模型，这意味着大多数情况下只要一次遍历就能计算出几何信息。处于流中靠后位置元素通常不会影响靠前位置元素的几何特征，因此布局可以按从左至右、从上至下的顺序遍历文档。但是也有例外情况，比如 HTML 表格的计算就需要不止一次的遍历 (3.5)。

坐标系是相对于根框架而建立的，使用的是上坐标和左坐标。

布局是一个递归的过程。它从根呈现器（对应于 HTML 文档的 `<html>` 元素）开始，然后递归遍历部分或所有的框架层次结构，为每一个需要计算的呈现器计算几何信息。

根呈现器的位置左边是 0,0，其尺寸为视口（也就是浏览器窗口的可见区域）。

所有的呈现器都有一个“layout”或者“reflow”方法，每一个呈现器都会调用其需要进行布局的子代的 layout 方法。

Dirty 位系统

为避免对所有细小更改都进行整体布局，浏览器采用了一种“dirty 位”系统。如果某个呈现器发生了更改，或者将自身及其子代标注为“dirty”，则需要进行布局。

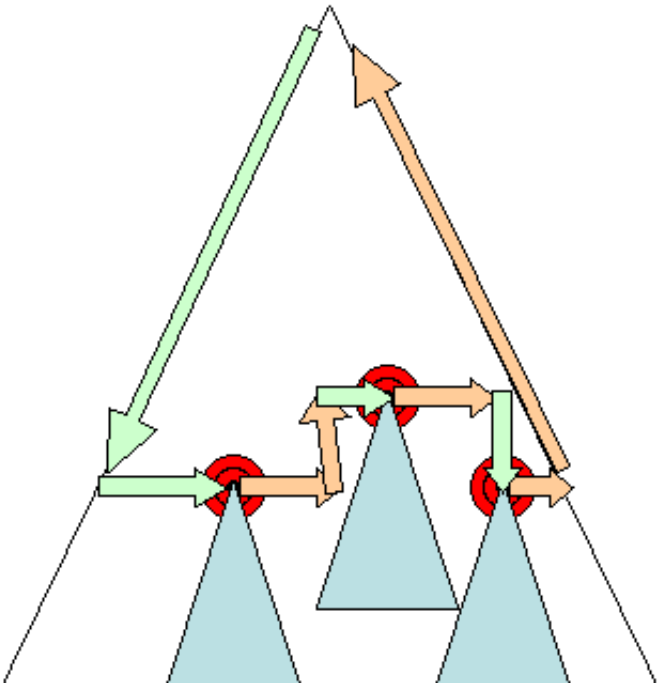
有两种标记：“dirty”和“children are dirty”。“children are dirty”表示尽管呈现器自身没有变化，但它至少有一个子代需要布局。

全局布局和增量布局

全局布局是指触发了整个呈现树范围的布局，触发原因可能包括：

1. 影响所有呈现器的全局样式更改，例如字体大小更改。
2. 屏幕大小调整。

布局可以采用增量方式，也就是只对 dirty 呈现器进行布局（这样可能存在需要进行额外布局的弊端）。
当呈现器为 dirty 时，会异步触发增量布局。例如，当来自网络的额外内容添加到 DOM 树之后，新的呈现器附加到了呈现树中。



图：增量布局 - 只有 ***dirty*** 呈现器及其子代进行布局 ***(3.6)***。

异步布局和同步布局

增量布局是异步执行的。Firefox 将增量布局的“reflow 命令”加入队列，而调度程序会触发这些命令的批量执行。WebKit 也有用于执行增量布局的计时器：对呈现树进行遍历，并对 dirty 呈现器进行布局。
请求样式信息（例如“offsetHeight”）的脚本可同步触发增量布局。
全局布局往往是同步触发的。
有时，当初始布局完成之后，如果一些属性（如滚动位置）发生变化，布局就会作为回调而触发。

优化

如果布局是由“大小调整”或呈现器的位置（而非大小）改变而触发的，那么可以从缓存中获取呈现器的大小，而无需重新计算。
在某些情况下，只有一个子树进行了修改，因此无需从根节点开始布局。这适用于在本地进行更改而不影响周围元素的情况，例如在文本字段中插入文本（否则每次键盘输入都将触发从根节点开始的布局）。

布局处理

布局通常具有以下模式：

1. 父呈现器确定自己的宽度。
2. 父呈现器依次处理子呈现器，并且：

1. 放置子呈现器（设置 x,y 坐标）。
2. 如果有必要，调用子呈现器的布局（如果子呈现器是 dirty 的，或者这是全局布局，或出于其他某些原因），这会计算子呈现器的高度。

3. 父呈现器根据子呈现器的累加高度以及边距和补白的高度来设置自身高度，此值也可供父呈现器的父呈现器使用。
4. 将其 dirty 位设置为 false。

Firefox 使用“state”对象 (nsHTMLReflowState) 作为布局的参数（称为“reflow”），这其中包括了父呈现器的宽度。
Firefox 布局的输出为“metrics”对象 (nsHTMLReflowMetrics)，其包含计算得出的呈现器高度。

宽度计算

呈现器宽度是根据容器块的宽度、呈现器样式中的“width”属性以及边距和边框计算得出的。
例如以下 div 的宽度：

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

将由 WebKit 计算如下（BenderBox 类，calcWidth 方法）：

- 容器的宽度取容器的 availableWidth 和 0 中的较大值。availableWidth 在本例中相当于 contentWidth，计算公式如下：

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

  clientWidth 和 clientHeight 表示一个对象的内部（除去边框和滚动条）。
- 元素的宽度是“width”样式属性。它会根据容器宽度的百分比计算得出一个绝对值。
- 然后加上水平方向的边框和补白。

现在计算得出的是“preferred width”。然后需要计算最小宽度和最大宽度。
如果首选宽度大于最大宽度，那么应使用最大宽度。如果首选宽度小于最小宽度（最小的不可破开单位），那么应使用最小宽度。

这些值会缓存起来，以用于需要布局而宽度不变的情况。

换行

如果呈现器在布局过程中需要换行，会立即停止布局，并告知其父代需要换行。父代会创建额外的呈现器，并对其调用布局。

绘制

在绘制阶段，系统会遍历呈现树，并调用呈现器的“paint”方法，将呈现器的内容显示在屏幕上。绘制工作是使用用户界面基础组件完成的。

全局绘制和增量绘制

和布局一样，绘制也分为全局（绘制整个呈现树）和增量两种。在增量绘制中，部分呈现器发生了更改，但是不会影响整个树。更改后的呈现器将其在屏幕上对应的矩形区域设为无效，这导致 OS 将其视为一块“dirty 区域”，并生成“paint”事件。OS 会很巧妙地将多个区域合并成一个。在 Chrome 浏览器中，情况要更复杂一些，因为 Chrome 浏览器的呈现器不在主进程上。Chrome 浏览器会在某种程度上模拟 OS 的行为。展示层会侦听这些事件，并将消息委托给呈现根节点。然后遍历呈现树，直到找到相关的呈现器，该呈现器会重新绘制自己（通常也包括其子代）。

绘制顺序

CSS2 规范定义了绘制流程的顺序。绘制的顺序其实就是元素进入堆栈样式上下文的顺序。这些堆栈会从后往前绘制，因此这样的顺序会影响绘制。块呈现器的堆栈顺序如下：

1. 背景颜色
2. 背景图片
3. 边框
4. 子代
5. 轮廓

Firefox 显示列表

Firefox 遍历整个呈现树，为绘制的矩形建立一个显示列表。列表中按照正确的绘制顺序（先是呈现器的背景，然后是边框等等）包含了与矩形相关的呈现器。这样等到重新绘制的时候，只需遍历一次呈现树，而不用多次遍历（绘制所有背景，然后绘制所有图片，再绘制所有边框等等）。

Firefox 对此过程进行了优化，也就是不添加隐藏的元素，例如被不透明元素完全遮挡住的元素。

WebKit 矩形存储

在重新绘制之前，WebKit 会将原来的矩形另存为一张位图，然后只绘制新旧矩形之

间的差异部分。

动态变化

在发生变化时，浏览器会尽可能做出最小的响应。因此，元素的颜色改变后，只会对该元素进行重绘。元素的位置改变后，只会对该元素及其子元素（可能还有同级元素）进行布局和重绘。添加 DOM 节点后，会对该节点进行布局和重绘。一些重大变化（例如增大“html”元素的字体）会导致缓存无效，使得整个呈现树都会进行重新布局和绘制。

呈现引擎的线程

呈现引擎采用了单线程。几乎所有操作（除了网络操作）都是在单线程中进行的。在 Firefox 和 Safari 中，该线程就是浏览器的主线程。而在 Chrome 浏览器中，该线程是标签进程的主线程。
网络操作可由多个并行线程执行。并行连接数是有限的（通常为 2 至 6 个，以 Firefox 3 为例是 6 个）。

事件循环

浏览器的主线程是事件循环。它是一个无限循环，永远处于接受处理状态，并等待事件（如布局和绘制事件）发生，并进行处理。这是 Firefox 中关于主事件循环的代码：

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

CSS2 可视化模型

画布

根据 CSS2 规范，“画布”这一术语是指“用来呈现格式化结构的空间”，也就是供浏览器绘制内容的区域。画布的空间尺寸大小是无限的，但是浏览器会根据视口的尺寸选择一个初始宽度。

根据 www.w3.org/TR/CSS2/zindex.html，画布如果包含在其他画布内，就是透明的；否则会由浏览器指定一种颜色。

CSS 框模型

CSS 框模型描述的是针对文档树中的元素而生成，并根据可视化格式模型进行布局的矩形框。

每个框都有一个内容区域（例如文本、图片等），还有可选的周围补白、边框和边距区域。

图：***CSS2*** 框模型

每一个节点都会生成 0..n 个这样的框。
所有元素都有一个“display”属性，决定了它们所对应生成的框类型。示例：

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

默认值是 inline，但是浏览器样式表设置了其他默认值。例如，“div”元素的 display 属性默认值是 block。
您可以在这里找到默认样式表示例：www.w3.org/TR/CSS2/sample.html

定位方案

有三种定位方案：

1. 普通：根据对象在文档中的位置进行定位，也就是说对象在呈现树中的位置和它在 DOM 树中的位置相似，并根据其框类型和尺寸进行布局。
2. 浮动：对象先按照普通流进行布局，然后尽可能地向左或向右移动。
3. 绝对：对象在呈现树中的位置和它在 DOM 树中的位置不同。

定位方案是由“position”属性和“float”属性设置的。

- 如果值是 static 和 relative，就是普通流
- 如果值是 absolute 和 fixed，就是绝对定位

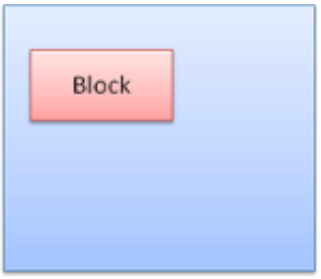
static 定位无需定义位置，而是使用默认定位。对于其他方案，网页作者需要指定位置：top、bottom、left、right。

框的布局方式是由以下因素决定的：

- 框类型
- 框尺寸
- 定位方案
- 外部信息，例如图片大小和屏幕大小

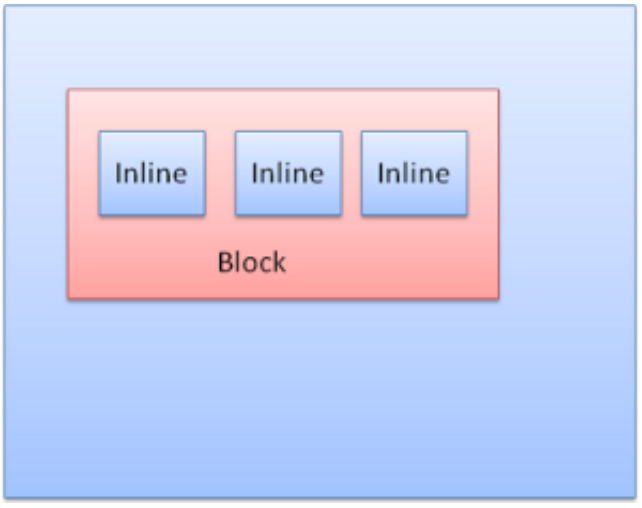
框类型

block 框：形成一个 block，在浏览器窗口中拥有其自己的矩形区域。



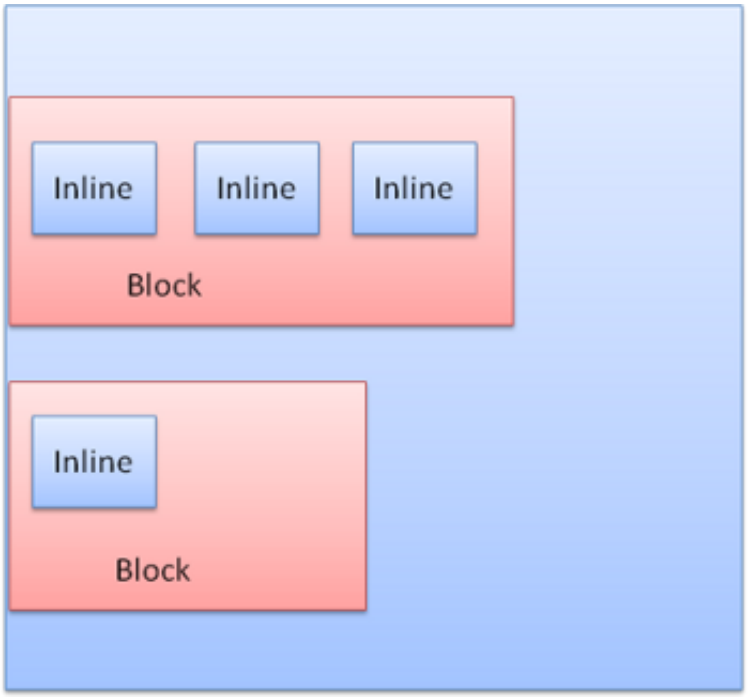
图：***block*** 框

inline 框：没有自己的 block，但是位于容器 block 内。



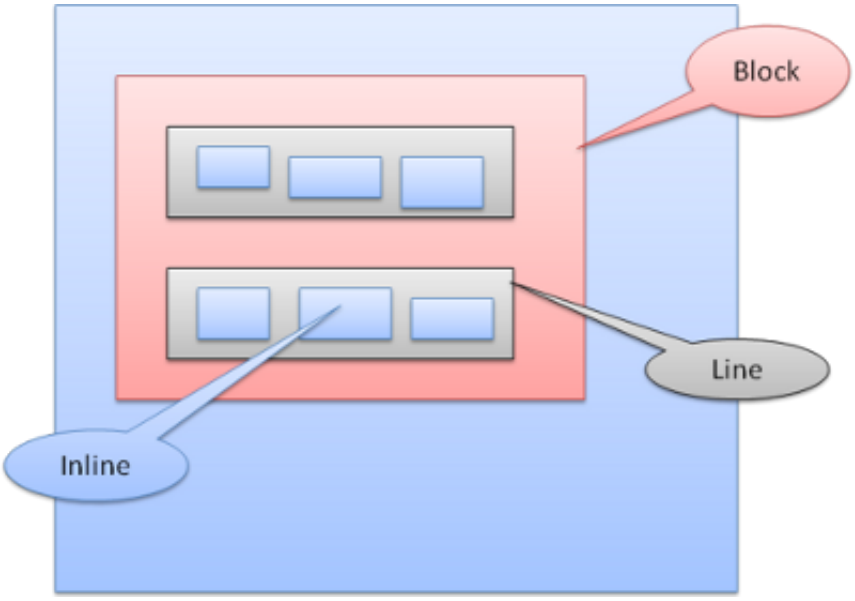
图：***inline*** 框

block 采用的是一个接一个的垂直格式，而 inline 采用的是水平格式。



图：***block*** 和 ***inline*** 格式

inline 框放置在行中或“行框”中。这些行至少和最高的框一样高，还可以更高，当框根据“底线”对齐时，这意味着元素的底部需要根据其他框中非底部的位置对齐。如果容器的宽度不够，inline 元素就会分为多行放置。在段落中经常发生这种情况。



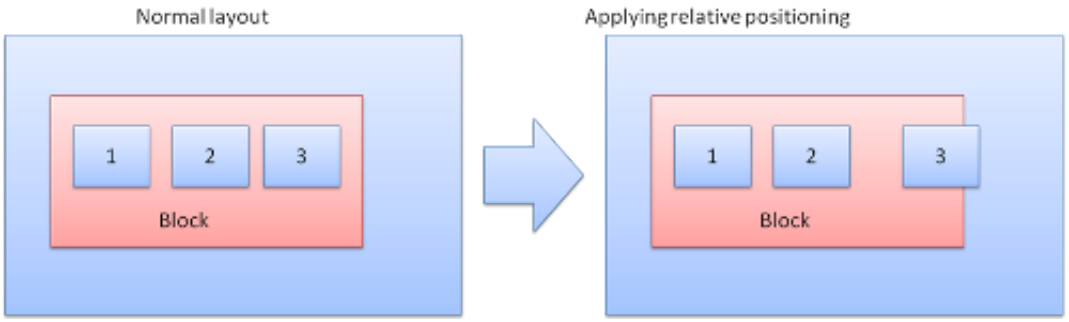
图：行

定位

相对

相对定位：先按照普通方式定位，然后根据所需偏移量进行移动。

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



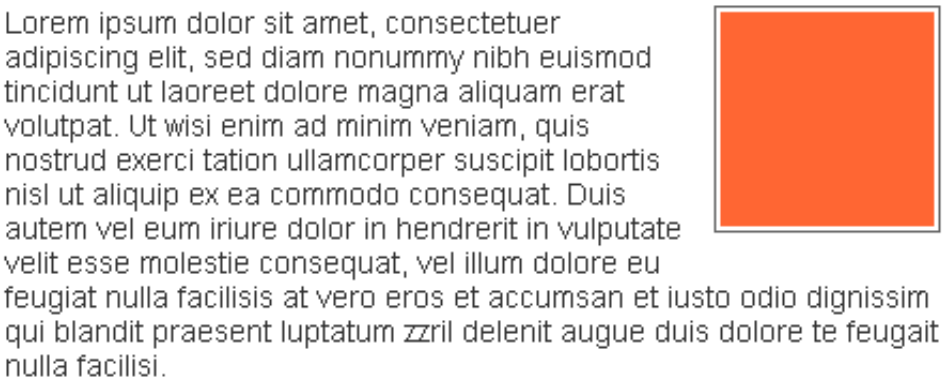
图：相对定位

浮动

浮动框会移动到行的左边或右边。有趣的特征在于，其他框会浮动在它的周围。下面这段 HTML 代码：

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

显示效果如下：

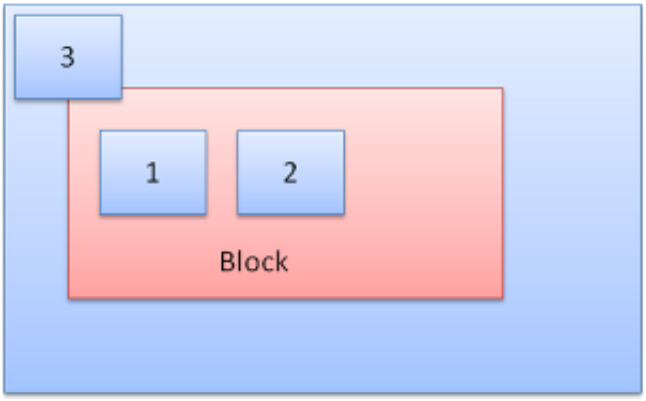


图：浮动

绝对定位和固定定位

这种布局是准确定义的，与普通流无关。元素不参与普通流。尺寸是相对于容器而言的。在固定定位中，容器就是可视区域。

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



图：固定定位

请注意，即使在文档滚动时，固定框也不会移动。

分层展示

这是由 z-index CSS 属性指定的。它代表了框的第三个维度，也就是沿"z 轴"方向的位置。

这些框分散到多个堆栈（称为堆栈上下文）中。在每一个堆栈中，会首先绘制后面的元素，然后在顶部绘制前面的元素，以便更靠近用户。如果出现重叠，新绘制的元素就会覆盖之前的元素。
堆栈是按照 z-index 属性进行排序的。具有"z-index"属性的框形成了本地堆栈。视口具有外部堆栈。

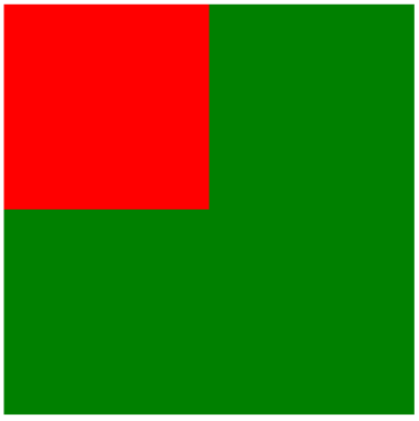
示例：

```
<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>
```

结果如下：



图：固定定位

虽然红色 div 在标记中的位置比绿色 div 靠前（按理应该在常规流程中优先绘制），但是 z-index 属性的优先级更高，因此它移动到了根框所保持的堆栈中更靠前的位置。

参考资料

1. 浏览器架构
    1. Grosskurth, Alan. A Reference Architecture for Web Browsers (pdf)
    2. Gupta, Vineet. How Browsers Work - Part 1 - Architecture
2. 解析
    1. Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools（即“Dragon book”）, Addison-Wesley, 1986
    2. Rick Jelliffe. The Bold and the Beautiful: two new drafts for HTML 5.
3. Firefox
    1. L. David Baron, Faster HTML and CSS: Layout Engine Internals for

Web Developers.

2. L. David Baron, Faster HTML and CSS: Layout Engine Internals for Web Developers（Google 技术访谈视频）
3. L. David Baron, Mozilla's Layout Engine
4. L. David Baron, Mozilla Style System Documentation
5. Chris Waterson, Notes on HTML Reflow
6. Chris Waterson, Gecko Overview
7. Alexander Larsson, The life of an HTML HTTP request

4. WebKit

1. David Hyatt, Implementing CSS（第一部分）
2. David Hyatt, An Overview of WebCore
3. David Hyatt, WebCore Rendering
4. David Hyatt, The FOUC Problem

5. W3C 规范

1. HTML 4.01 规范
2. W3C HTML5 规范
3. 层叠样式表第 2 级第 1 次修改 (CSS 2.1) 规范

6. 浏览器构建说明

1. Firefox. https://developer.mozilla.org/en/Build_Documentation
2. WebKit. http://webkit.org/building/build.html

塔利·加希尔是以色列的一名开发人员。她在 2000 年开始从事网络开发工作，逐渐熟悉了 Netscape 的“邪恶”层模型。就像理查德·费曼 (Richard Feynmann) 一样，她极度热衷于探究事物的原理，因此开始深入了解浏览器的内部原理，并记录研究成果。塔利还发表过一篇关于客户端性能的简短指南。

翻译情况

此网页已两次翻译为日文！浏览器的工作原理：现代网络浏览器幕后揭秘 (ja)，译者：@_kosei_；以及ブラウザってどうやって動いてるの？（モダンWEBブラウザシーンの裏側，译者：@ikeike443 和 @kiyoto01。感谢大家！