浏览器渲染引擎

漆工
HYPERS 前端团队负责人

8 人赞同了该文章

背景

浏览器的内核中主要分为渲染引擎和 javascript 引擎，本篇主要围绕渲染引擎介绍一下浏览器的工作原理。

首先，我们先看几个 user-agent 的字符串：

- Mozilla/ 1.0 (Windows NT 6.1;rv:2.0.1) Gecko/2010010Firefox/4.0.1
- Mozilla/ 4. 0 (compatible; MSIE 7. 0; Windows NT 6. 0)
- Mozilla/ 5. 0 (Linux; Android4. 0. 4; Galaxy Nexus Build/ IMM76B) AppleWebKit/ 535. 19 (KHTML, like Gecko) Chrome/ 18. 0. 1025. 133 Mobile Safari/ 535.

这是3个不同浏览器的 user-agent，第1个是Firefox的，第2个是IE7的，第3个是Chrome的。有没有觉得很奇怪，为什么所有字符串的前面都会有一个 Mozilla 开头呢？ 而且在 Chrome 中包含了很多其他浏览器的标识，Android，Gecko，Safari ... , 这些浏览器厂商为什么要把这个user-agent字符串设计成这样？user-agent 按照正常的理解就是浏览器的标识，包含操作系统和浏览器信息带上版本号就行了，浏览器厂商为什么搞的这么复杂呢？

这需要了解一下 user-agent 字符串历史 ，大致的意思是， 早期的浏览器 Netscape 的 user-agent 是以 `Mozilla/Version [Language] (Platform; Encryption)` 的格式，大多数服务器在加载页面前都会检查 user-agent 是否为该款浏览器，然而在 1995年， IE 发布首款浏览器，如果不兼容Netscape user-agent 字串，用户就根本访问不了页面，于是IE就设计了这种格式：

`Mozilla/2.0 (compatible; MSIE Version; Operating System)` 。 其他新的浏览器发布也是一样的，为了溶入主流而不被踢出局，在 user-agent 字串中放详尽的信息，以便骗取网站的信任使它与其它流行的浏览器兼容。

接下来，简单回顾一下浏览器的历史:

- WorldWideWeb 1991 年
- Mosaic 1993年
- Netscape 1994年
- Opera 1995年
- IE 1995年 一战
- Safari 2003年
- Firefox 2004年 二战
- Chrome 2008年
- Edge 2015年

很多人都以为最早的浏览器是 Netscape， 其实在 Netscape 之前还有 WorldWideWeb 和 Mosaic 两个浏览器，WorldWideWeb 是世界上第一个浏览器，它同时也是一个编辑器，在 1991年发布, 当时 http 的版本还是 0.9 ，只支持 get 请求。

随后在 1993年 由美国伊利诺州的伊利诺大学的 NCSA 组织，发布第一个可以显示图片的浏览器，叫 Mosaic。随后 NCSA 将 Mosaic 的商业运营权转售给了 Spyglass 公司，该公司又向包括微软公司在内的多家公司技术授权，然后，微软的IE浏览器就是从这里开始了。

1994年，在 Mosaic 浏览器开发团队的核心成员，重新成立 Netscape 公司，从此 Netscape 浏览器诞生。

1995年，挪威的本土电信公司 Telenor 开发了一个新的浏览器 Opera，Opera 浏览器他特点就是速度快，但是兼容性不是很好，在浏览器上独创了很多功能，比如标签式浏览就是 Opera 发明的，Opera 在经历很多坎坷后，在 2016 年被奇虎 360 收购。

1995同年，微软在取得 Spyglass Mosaic 的源代码和授权后，发布了首款浏览器 Internet Explorer ，从此展开浏览器第一次大战。

1998年，Netscape 公司内部成立Mozilla组织。

2003年，苹果公司发布了自己第一个浏览器 Safari，同时在两年后，也就是 2005 年开源了自己的浏览器内核 Webkit。

2004年，Mozilla 组织发布了 Firefox 浏览器，然后第二次浏览器大战开始。

2008年，是一个灾难很多的一年，四川大地震，过年回家又遇上雪灾，作为前端开发我的将要开始兼容一个由 Goolge 开发新的 Chrome 浏览器。 但是 Chrome 带来的体验是让人开心的 。

2015年，微软随着 Windows 10 一起发布 Edge 浏览器，伤透大家心的 IE ，估计维护不下去。

当前市场浏览器市场占比最高的是 Chrome 浏览器，其次就是 IE，Firefox ，详细占比查看 Net Marketshare 的统计。

浏览器内核

内核 | 浏览器 | 出生年份 | JS 引擎 | 开源
-------| -----| ---------| -------------| --------|----
Trident | IE4 - IE11| 1997 | JScript，9+chakra |
Gecko | Firefox | 2004 | SpiderMonkey | MPL |
WebKit | Safari,Chromium,Chrome(-2013) ,Android浏览器,ChromeOS,WebOS 等 | 2005| WebCore + JavascriptCore | BSD
Blink | Chrome, Opera | 2013 | V8 | GPL
Edge | Edge | 2015 | EdgeHTML + Chakra | MIT(chakra)

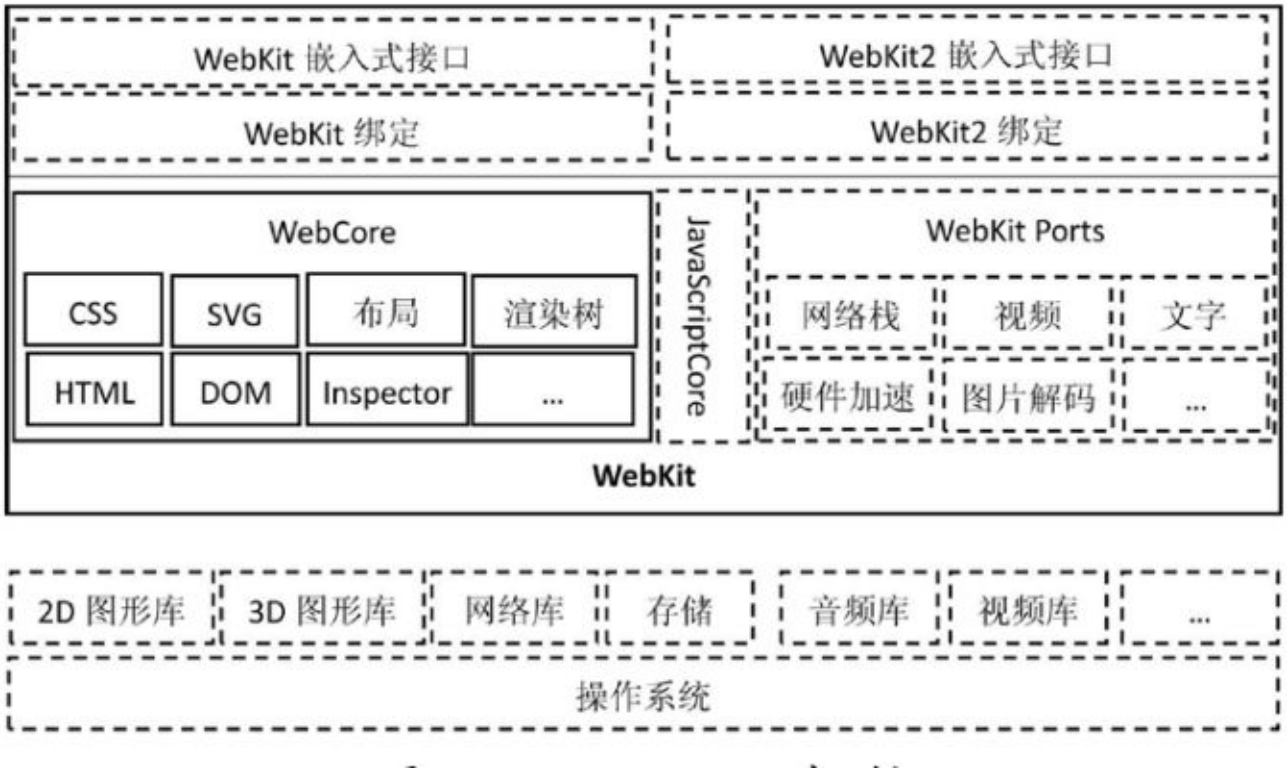
市面上的浏览器都有自己的内核，有些内核之间还存一些关系。 WebKit 是 Apple 公司在 2005 年开源的一个内核，Apple 公司早期使用的引擎是 KHTML，KHTML 是 KDE 社区维护，Apple 技术团队也参与 KHTML 的开发，但是在开发过程中，KDE 社区不太喜欢 Apple 团队人员提交的代码。所以 Apple 公司的人就自己独立出来，在 KHTML 基础之上创建了 WebKit 内核，并在 2005 年开源。 在 2008年，Google 发布 Chrome 浏览器，采用的也是 Webkit 内核，同时技术团队人员也参与 WebKit 项目的开发，但是在设计上与 Apple 团队存在分歧，所以 Google 的人就独立出来，基于 WebKit 开发了 Blink 内核，Blink 在 Webkit 的基础上加入多进程，沙箱等很多技术。

回过头看看国内，有很多浏览器，很牛逼，都是多核的，想要兼容国内银行系统就切换到 Trident 内核，想要访问速度就切换到 Webkit 内核，Blink 发布以后，就把 WebKit 换成了 Blink 。

- QQ浏览器 Trident+Webkit (Blink)
- 360安全浏览器 Trident+Webkit (Blink)
- 猎豹浏览器 Trident+Webkit (Blink)
- 世界之窗 Trident+Webkit (Blink)
- 搜狗高速浏览器 Trident+Webkit (Blink)
- UC浏览器 Trident+Webkit (Blink)
- ....

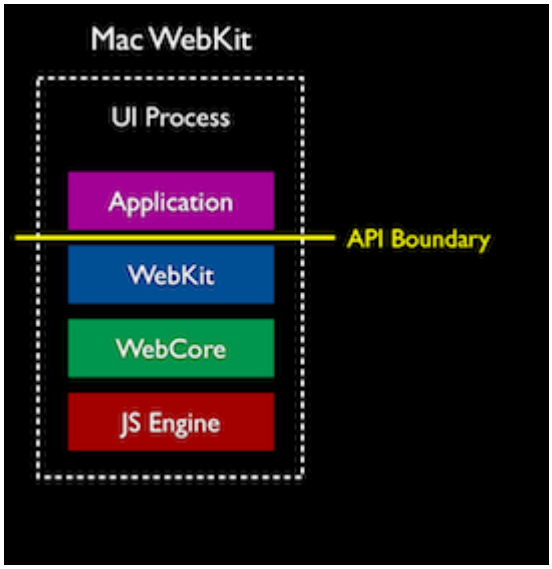
WebKit 架构

接下来我们进入到 Webkit 里面，首先看一下 WebKit 的架构图



在上图中实线部分，也就是 WebCore，基本上在各个浏览器中是共享的，虚线部分在各个浏览器中的存在差异。 WebCore 是渲染引擎，包含的 HTML 解释器，CSS 解释器，处理页面布局渲染等功能。 JavascriptCore 就是 WebKit 内置的 Javascript 引擎。 在最上层是 WebKit 嵌入式接口，这些接口提供给浏览器调用，但是我们可以看到图中有 WebKit 和 WebKit2 两个嵌入式接口，这两有什么区别呢?

WebKit 2 是 2010年4月份发布的，抽象出一组新的编程接口，给开发者用，同时采用了多进程：，一个UI 进程，处理Web平台与浏览器接口的进程，另外一个 Web 进程, Web 页面渲染的进程。 让 web 进程与 UI 进程隔离，在健壮性、安全性以及更好地使用多核 cpu 等方面带来了好处。 以下是WebKit 和 WebKit2的对比图:



更多详细信息可以阅读： trac.webkit.org/wiki/We...

在 Javascript 旁边有很大一块区域是 Webkit Port，所谓 WebKit Port，并没有确切的形式，可以看作是OS，平台（应用程序框架），Javacsript 引擎，以及各种第三方库的一个组合。WebKit Port 提供不同的 Port 接口供外部程序使用，

以 Webkit 为核心存在很多移植：

- Apple's Mac Port
- Apple's Windows Port
- Cairo-based Windows Port
- JSCOnly Port
- WebKitGTK+ Port
- QTWebKit
- ...

WebKit Port 通常处于以下几种目的：

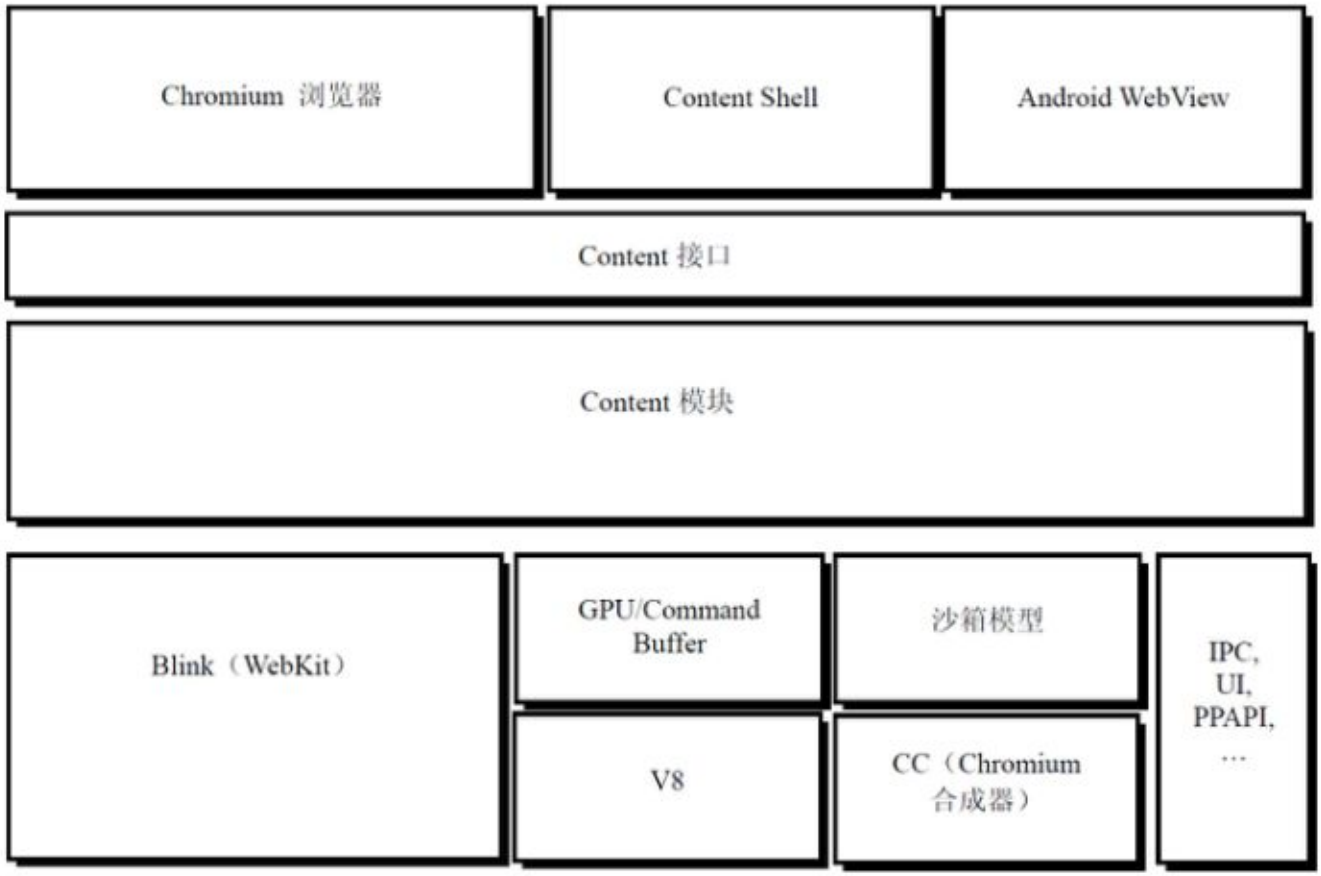
- 使用WebKit作为浏览器（或者类似的User Agent）的页面解析，排版和渲染的核心，如Safari，Chrome
- 对WebKit进行封装，对外提供构建一个浏览器（或者类似的UA）的API接口，如Qt
- 以上两者皆有，如Android，iOS，BlackBerry

不同的 port 关注点不一样

- Mac 的 port 注意力集中在浏览器和操作系统的分割，它通过 Obj-C 和 C++ 代码把（WebKit）渲染引擎嵌入到本地应用中。
- Chromium port 专注在浏览器。
- QtWebKit 则把它的 WebKit 实现作为一个运行时的库或者渲染引擎，同其跨平台 GUI 应用程序框架一起提供给其它应用使用。 比如 无头浏览器 Phantomjs 就是基于 QtWebKit 实现的。

更多详细信息可以阅读：trac.webkit.org/wiki#...

接下来，我们来看一下基于 Chromium port 的浏览器。



在图中我们可以看到左下角是 WebKit 内核，和他平级的还有:

- GPU/Command Buffer , Command Buffer 是 GPU 线程的通信媒介，提供GPU 硬件加速。
- V8，Javascript 引擎
- 沙箱模型，浏览器安全保护的一种设计
- CC（Chromium 合成器），对 RenderLayer Tree 分成渲染
- IPC,UI,PPAPI ...

在上面一层是 Content , 如果没有 Content， 也能正常渲染，不过上面提到的这些GPU 加速，沙箱模型，HTML 5等功能将没有，Content API 提供了公开稳定的接口， 目标是支持所有的HTML5功能和GPU硬件加速等功能。

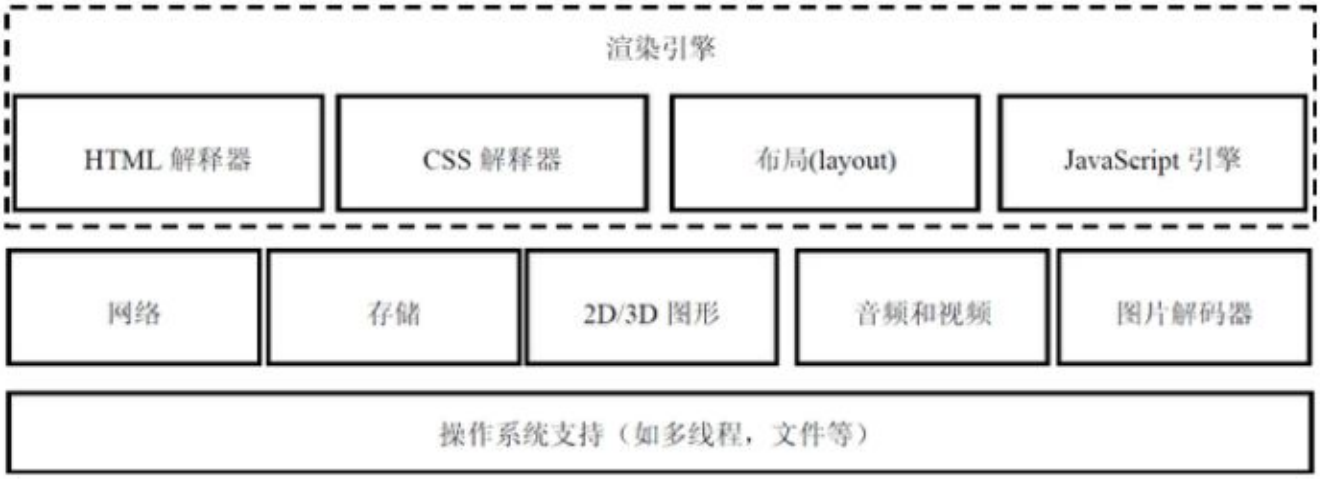
“Chromium浏览器” 和 “ContentShell” 是构建在 ContentAPI 之上的两个“浏览器”，Chromium具有浏览器完整的功能，也就是我们编译出来能看到的浏览器式样。而“ContentShell”是使用ContentAPI来包装的一层简单的“壳”，但是它也是一个简单的“浏览器”，用户可以使用Content模块来渲染和显示网页内容。 ContentShell的作用很明显，其一可以用来测试Content模块很多功能的正确性，例如渲染、硬件加速等；其二是一个参考，可以被很多外部的项目参考来开发基于“ContentAPI”的浏览器或者各种类型的项目。

在 Android 系统上，ContentShell 的作用更大，这是因为同它并排的左侧的 “Chromium浏览器” 部分的代码根本就没有开源，这直接导致开发者只能依赖ContentShell。

“Android WebView”，它是为了满足Android系统上的 WebView 而设计的，其思想是利用Chromium的实现来替换原来Android系统默认的 WebView。

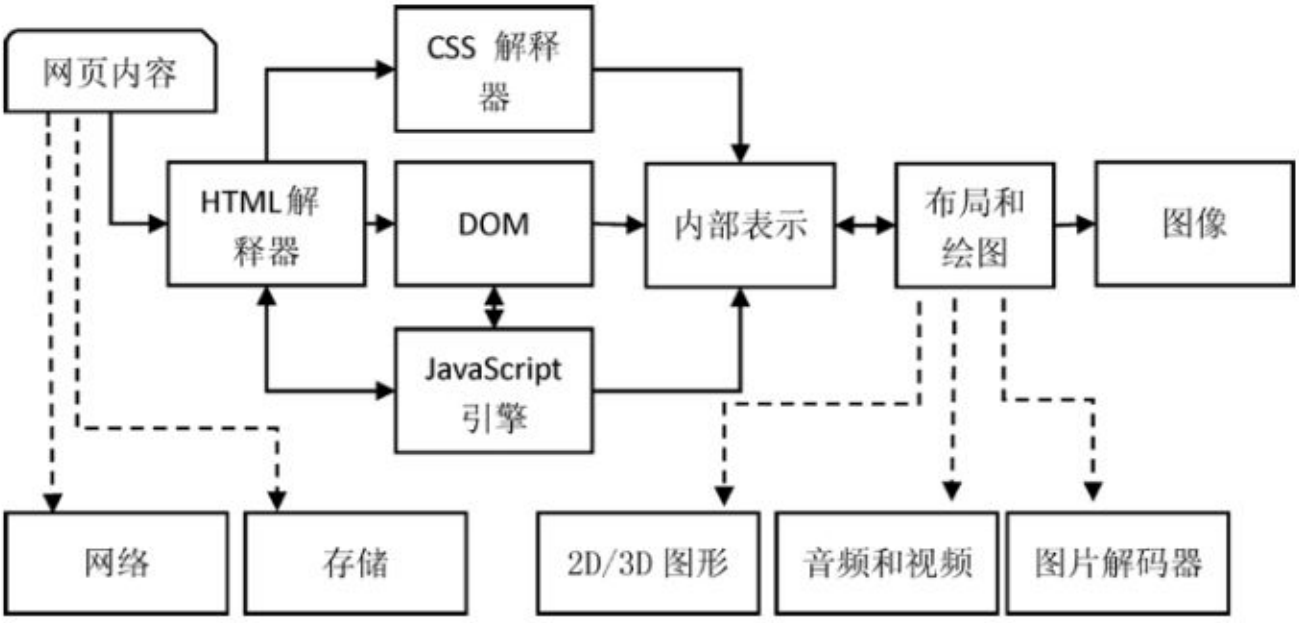
渲染过程

以下是浏览器渲染引擎及依赖模块

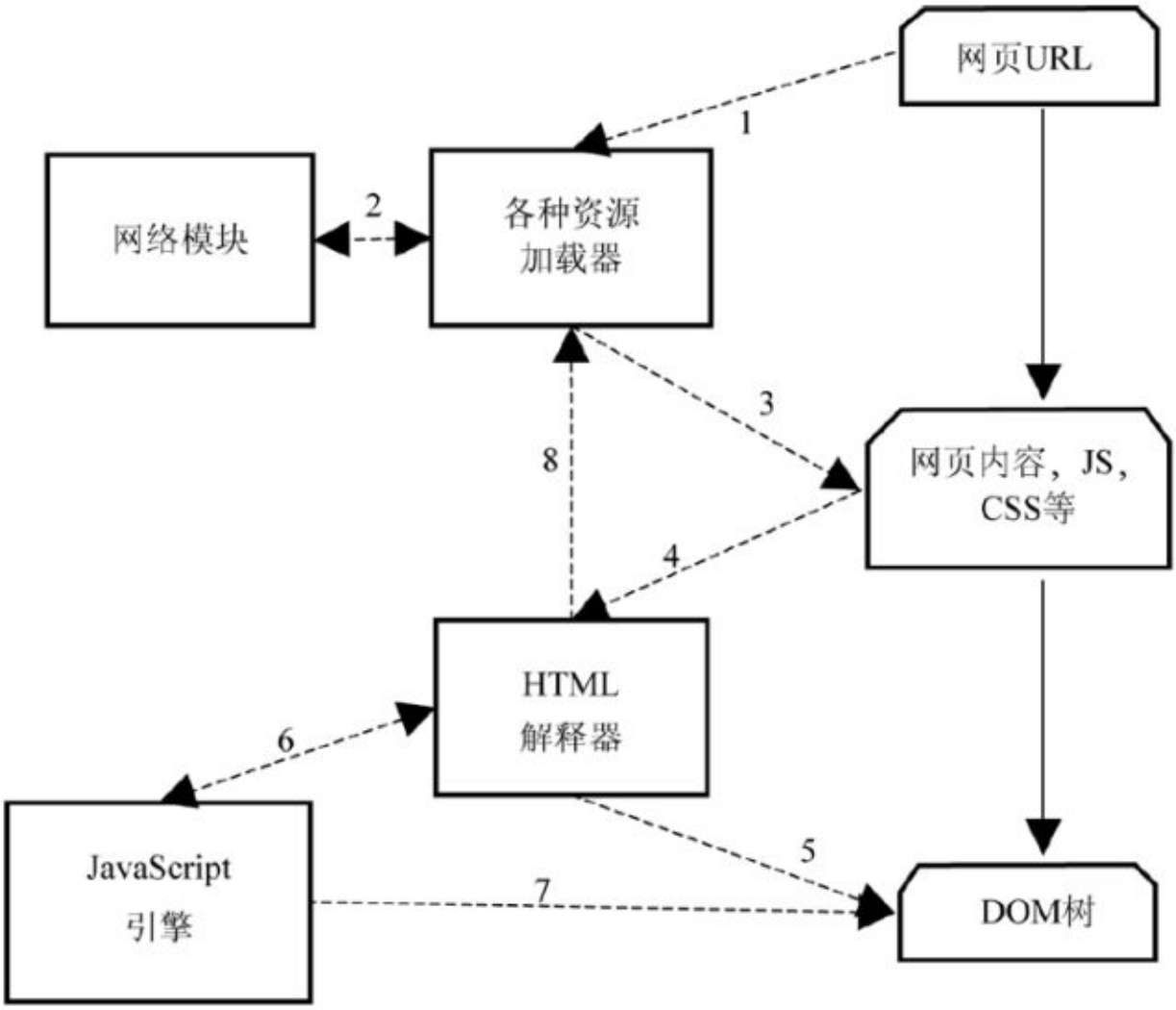


一个渲染引擎主要包括HTML解释器、CSS解释器、布局和JavaScript引擎等，JavaScript引擎现在都已经独立出来。 下面是所依赖的模块，包括网络，存储，2D/3D 图形，音频和视频，图片解码器等等..., 再下面就是操作系统相关的支持。

一个大致的渲染过程及依赖模块关系图如下：

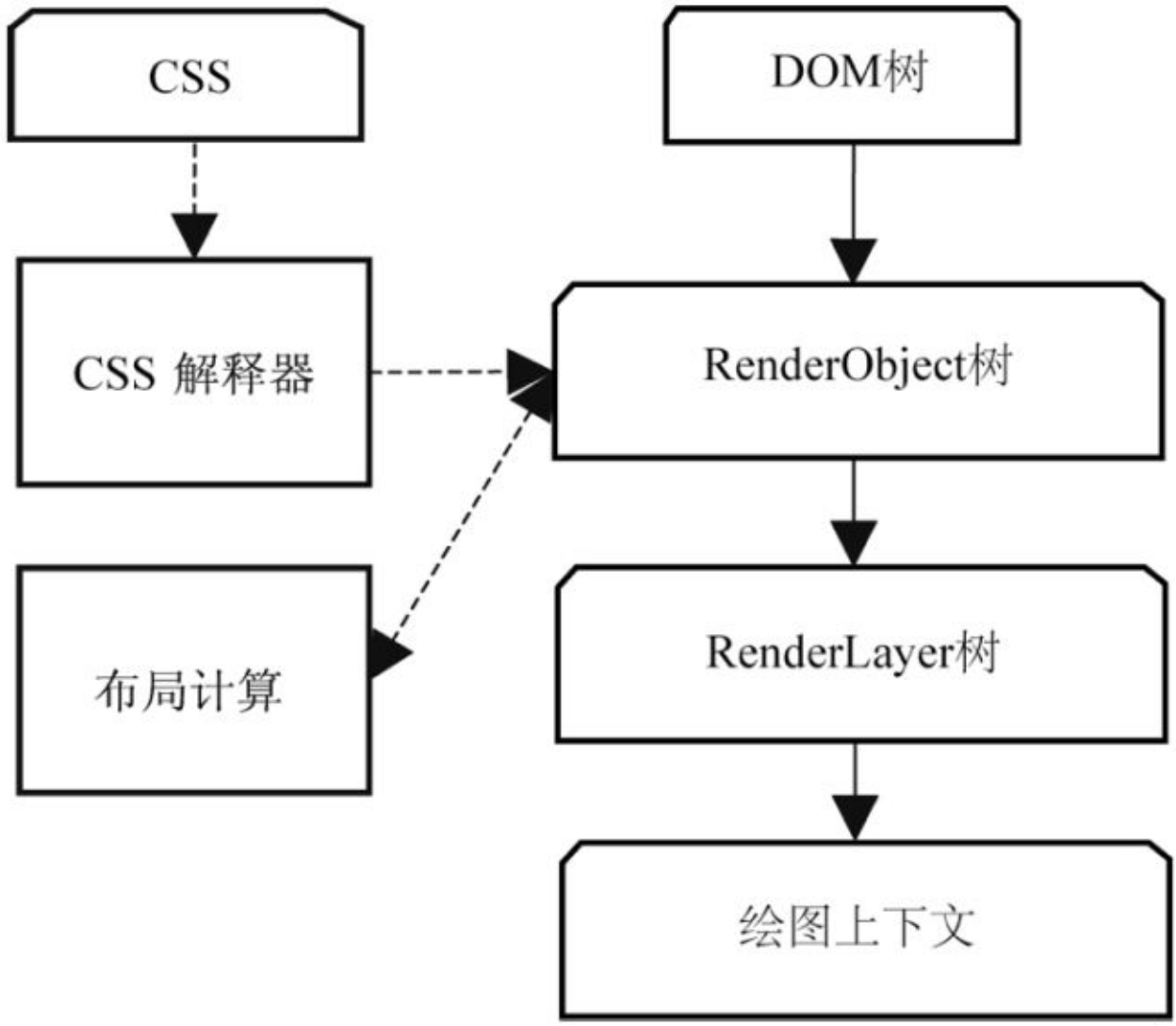


接下来我们再来看一下，在 WebKit 的渲染的详细过程：



首先是在浏览器输入 URL 以后，依赖网络模块加载各种资源，得到一个HTML , HTML 交给 HTML 解析器进行解析，最后生成 DOM 树，如果再解析过程中有存在 Javascript 代码就交给 Javascript 引擎处理，处理完成返回给 DOM 树， 这个环节的主要目的就是构建一个DOM 树。

然后看一下 DOM树到绘制上下文



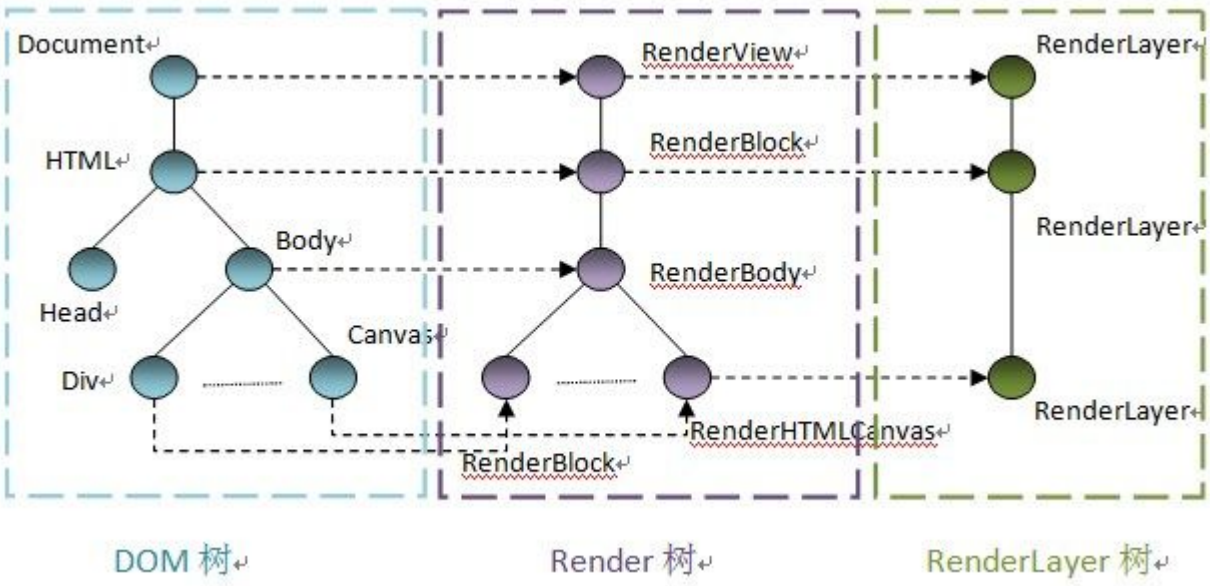
在网络资源中获得 CSS 代码以后，会把 CSS 交给 CSS 解析器处理，同时会计算布局。 DOM 树会构建成一个 RenderObject 树，它和 DOM 树节点是一一对应，然后再和 解析后的CSS 合并分析，生成 RenderLayer 树， 这个树就是最终用于渲染的树，然后绘制上下文。

以下是在网上找到的几张图，简单解释了 DOM 树到 RenderLayer 树最终的过程。

```
<html>
    <head>
    </head>
    <body>
        <div>abc</div>
        <canvas id="webgl" width="80" height="80"></canvas>
        <a href="http://thisisaa"></a>
        <img></img>
        <input type="button"></input>
        <select></select>
        <table width="100" height="50">
            <tr>
                <td>d0</td>
            </tr>
        </table>

        <script type="text/javascript">
            var canvas = document.getElementById("webgl");
            var gl = canvas.getContext("experimental-webgl");
            if (!gl) {
                alert("There's no WebGL context available.");
                return;
            }
        </script>
    </body>
</html>
```

这一段简单的 HTML 代码，其中包含的 body, div canvas script 等元素，通过 HTML，CSS 解析器进行解析，最终会生成一个 RenderLayer 树， 前面在Chromium 架构图的时候有提到 CC 合成器，随着GPU硬件能力的增强，包括在很多小型设备上也是如此，浏览器可以借助于其处理图形方面的性能来对渲染实现加速。此时不再将所有层绘制到一起，而是进行分层渲染，合成之后再显示到屏幕上。



以下 RenderLayer 树的结构，从图中可以看出整个树分了 3 个 Layer，在 Layer 下面包含了 RenderBlock，RenderText 等 Render 节点，每个节点上都包含的坐标，大小，以及背景颜色等渲染依赖的信息。

```
layer at (0,0) size 1028x683
  RenderView at (0,0) size 1028x683
layer at (0,0) size 1028x683
  RenderBlock {HTML} at (0,0) size 1028x683
    RenderBody {BODY} at (8,8) size 1012x667
      RenderBlock {DIV} at (0,0) size 1012x20
        RenderText {#text} at (0,0) size 22x19
          text run at (0,0) width 22: "abc"
      RenderBlock (anonymous) at (0,20) size 1012x88
        RenderText {#text} at (80,65) size 4x19
          text run at (80,65) width 4: " "
        RenderInline {A} at (0,0) size 0x0 [color=#0000EE]
        RenderText {#text} at (0,0) size 0x0
        RenderImage {IMG} at (84,80) size 0x0
        RenderText {#text} at (84,65) size 4x19
          text run at (84,65) width 4: " "
        RenderButton {INPUT} at (90,64) size 16x22 [bgcolor=#DDDDDD] [border: (2px outset #DDDDDD)]
        RenderText {#text} at (108,65) size 4x19
          text run at (108,65) width 4: " "
        RenderMenuList {SELECT} at (114,65) size 25x20 [bgcolor=#DDDDDD] [border: (1px solid #000000)]
          RenderBlock (anonymous) at (1,1) size 23x18
            RenderBR at (4,1) size 0x16 [bgcolor=#DDDDDD]
        RenderText {#text} at (0,0) size 0x0
      RenderTable {TABLE} at (0,108) size 100x50
        RenderTableSection {TBODY} at (0,0) size 100x50
          RenderTableRow {TR} at (0,2) size 100x46
            RenderTableCell {TD} at (2,14) size 96x22 [r=0 c=0 rs=1 cs=1]
              RenderText {#text} at (1,1) size 16x19
                text run at (1,1) width 16: "d0"
layer at (8,28) size 80x80
  RenderHTMLCanvas {CANVAS} at (0,0) size 80x80
```

RenderBlock 其实就是我们 HTML 中的块级元素，我们都知道一个元素是否为块级元素是可以通过 CSS 改变的，所以，一个 RenderLayer 树的结构也会根据CSS的变化变化，如果影响到元素的位置发生变化会都在整个树重新计算，也是就是我们说的回流。 关于回流的解释可以参考：www-archive.mozilla.org... 。

最后我们简单了解一下 Shadow DOM， 在前端开发过程中大家都知道 React, Vue 这些框架中有组件的概念，一个页面中存在很多重复的组件，在一个组件内部又存在很多基础的 HTML 元素，这些元素可以组成一颗DOM树的子树。这样一个组件可以被到处使用，但是问题随之而来，那就是每个使用组件的地方都会知道这个子树的结构。当网页的开发者需要访问网页DOM树的时候，这些控件内部的DOM子树都会暴露出来，这些暴露的节点不仅可能给DOM树的遍历带来很多麻烦，而且也可能给CSS的样式选择带来问题，因为选择器无意中可能会改变这些内部节点的样式，从而导致很奇怪的界面。

那有什么办法可以把这些内部节点封装起来？就像我们写 javascript 模块化一样，W3C 工作组提出了 Shadow DOM 的概念，比如在 HTML5 支持的 等标签，在其内部有很多复杂的结构， 但是播放，暂停等按钮我们在 DOM 树中无法直接找到相应的节点，这其实就是 Shadow DOM 的思想。 Shadow DOM 是可以通过 Javascript 自定义创建，在其内部可以维护自己的DOM, CSS ,事件等, 具有很好的密封性。 自定义 Shadow DOM 目前只有 Chrome 支持，不过，我相信在不远的将来 Shadow DOM 会给组件化开发带来更多美好的体验。

编辑于 2018-04-08