WebKit 总结

March 31, 2012

1 简介

现在的 WebKit 项目起源于 KDE 项目的 KHTML 模块，苹果公司开发浏览器时对当时开源浏览器作了分析后决定采用 KHTML 模块，后根据协议将修改过的开源，命名为 WebKit，有三个模块：

- WebKit 与 OS 交互
- WebCore Render engine，是 WebKit 的核心
- JavaScriptCore JavaScript 解析引擎

WebKit 并不是浏览器，只是一个排版引擎，与LaTeX 本质上类似，而排版内容和复杂度不可同日而语。针对不同的平台对 WebKit 的封装和绑定，官网称为 port，比如比较流行的 QtWebKit port 和 GTKWebKit port。本文旨在对 WebKit 内部及 QT 与 GTK port 中 JavaScript 引擎与 WebKit 交互及 graphic system 有个比较深入的了解。

2 内部机制

简单来讲，WebKit 内部的工作机制分以下几个步骤：

1. curl 获得请求的 stream 数据
2. DOM/Render 树的创建
3. JS/CSS 操纵 DOM
4. Render

2.1 获取请求网页的数据

网络处理部分提供 ResourceHandle 类供不同的平台库实现，具体可以在 WebCore/platform/network 里可见。//有时间继续，我们对此部分修改的可能性不大

- 网络资源 load 机制
- Google SPDY 它修改哪部分内容，强悍在哪里？

2.2 DOM/Render 树的创建

2.2.1 DOM/Render 树根节点的创建

HTMLDocument 类中通过私有方法 createParser()来创建 DocumentParser，而 DocumentParser 中通过 HTMLTreeBuilder 和 HTMLTokenizer 来创建 DOM 树。如图 1可见。 收到部分数据以后就开始了 DOM 树和 Render 树的创建。创建流程如图 2所示：

由图 3的流程开始了 Render 树根节点的创建，需要说明以下：

1. 两棵树同时创建，Render 树上只有 DOM 树上可见的元素，而诸如 head 等元素则不会出现在 Render 树中。同步创建的原因是为了更好的用户体验。所以为了尽可能快的呈现内容，不会去等 DOM 树生成好了再去创建 Render 树。
2. HTML 页面的根标签<html>不是 DOM 树的根节点，DOM 树的根节点为 HTMLDocument 类型，而<html>标签所代表的类则是 HTMLHtmlElement 类，与此类似所有 HTML 页面中的标签都可以在 WebCore/html/中找到相对应的类，HTML 页面根标签<html>对应的节点为 DOM 树根节点的子节点。

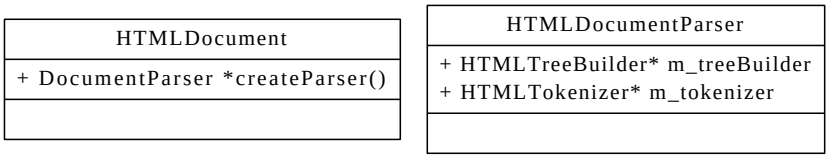

Figure 1: DOM 树创建重要变量

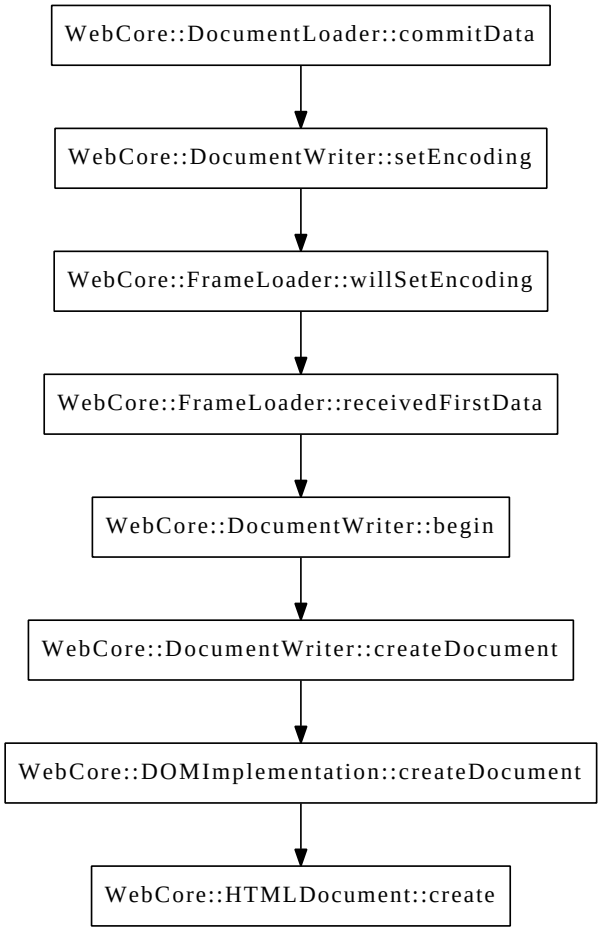

Figure 2: DOM Root creation

2.2.2 DOM/Render 节点的创建与添加

以 HTMLHtmlElement(<html>)节点的创建为例，它的创建过程如图 4所示。

2.3 Layout

1. Layout 和 Paint 这两个过程完全分开。开始执行 Paint 过程以前，必然预先执行过 Layout，否则图形库就不知道在哪里写字以及显示图像。但是这并不意味着，Layout 执行结束后，随即就立刻执行 Paint。实际上，Layout 执行结束后，触发一个事件，这个事件启动 Paint 过程。但是 Paint 过程也可以被其它事件触发，譬如屏幕内容的切换，以及把隐藏的浏览器窗口复原等等。

2. Layout 涵盖了所有 CSS 规定的布局要素。包括页面边缘与内容之间的空白，文字对插入图像的避让(floating)，单列与多列，上下层覆盖(z-index)等等。

3. 图像，视频播放器插件，Applet 等等，在 Layout 被称作 Replaced Render Object。这些 Replaced 元素的宽度和高度可以由 CSS 规定。如果 CSS 没有规定，就解析这些元素的数据流，譬如一个 JPG 照片的 metadata 里，规定了这幅照片原件的宽度和高度。如果元素自己也没有规定宽度高度，就使用 Webkit 提供的缺省值。

4. 文字的宽度根据页面的排版来确定。譬如一页中包含多列文字，则每列文字宽度相等。每列文字的宽度，乘以列数，加上列与列之间的夹缝，加上页面边缘空白等等，应当等于页面总的宽度。假设页面总的宽度已知，边

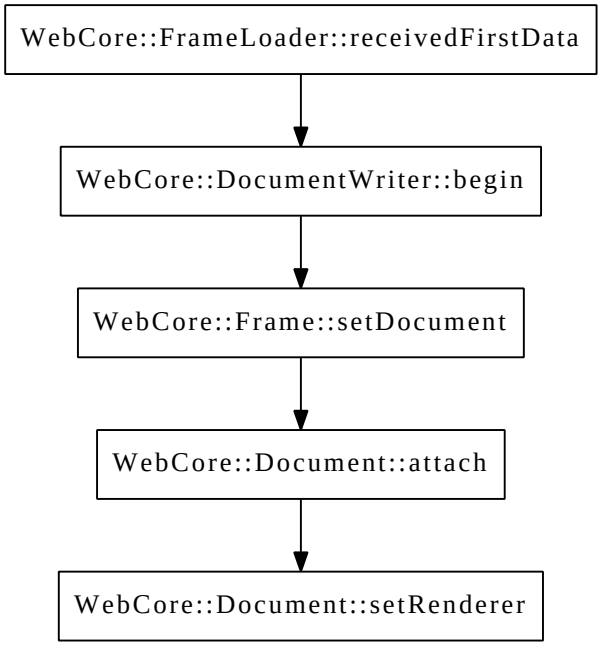

Figure 3: RenderTree root creation

缘空白，和列与列之间的夹缝的宽度也已知，就可以反推文字的宽度。

5. Render Tree 中每个节点在屏幕上的显示，都呈长方形格局。前面第 3 点和第 4 点，描述了宽度的确定。而高度的确定，取决于这个中间节点的所有后代节点的高度的总和。对于 Replaced 元素来说，它的高度相对比较容易确定，而文字段落的高度，需要根据字数，字型，以及字体大小计算得出。
6. 在 Layout 过程中，反复出现以 Repaint 为开头的子过程，例如 repaintAfterLayoutIfNeeded()。这些子过程的意义在于，当确定了某个节点的高度和宽度以后，需要对其前辈节点，和左右兄弟节点的位置，做适当调整。严格意义上来讲，这不是 repaint，而是 relayout。
7. 相对于 Layout 过程，Paint 过程的逻辑要简单得多。Paint 的过程，大致按照深度优先的顺序，遍历整棵 RenderTree。也就是说，从最左边的叶子节点开始，从左向右逐个绘制 RenderTree 所有可以显示的叶子节点。所谓“可以显示的叶子节点”，是因为 CSS 中可以规定，不显示某些叶子。

总结：

1. Layout 是一个计算量很繁重的过程。之所以繁重，主要体现在估算完每个 RenderTree 节点的宽度尤其是高度以后，需要相应调整这个节点的前辈节点以及左邻右舍兄弟节点的位置。对于文字段落而言，它的高度有赖于字数，字体和大小，所以估算不容易准确。
2. 有没有可能把 Layout 过程，与第一遍 Paint 过程合二为一？只要遍历一次 RenderTree 的所有叶子节点，绘制图像并码字。Paint 过程结束后，各个叶子节点对应的长方形的起始位置的(X,Y)坐标，以及宽度和高度都自然迎刃而解。然后再由叶子节点开始，逐步确定 RenderTree 中，各个中间节点的起始位置和宽度高度。这样做的好处是，可以大大降低 Layout 过程的成本。
3. Layout 过程假设每个 RenderTree 的节点都对应一个长方形屏幕区域。受限于这个规定，类似于 Figure 2 的效果，就显示不出来。有没有可能取消这个限制？SVG 不仅提供了强大的绘图能力，而且也提供了强大的排版布局能力。能不能把 CSS 当着 SVG 格式的一个子集来看待？

3 WebKit Qt port

3.1 编译

之前需要安装 QT 以及设置 QTDIR 变量。编译 Qt 源码的步骤如下：

```
./configure -debug -nomake example
make
sudo make install
```

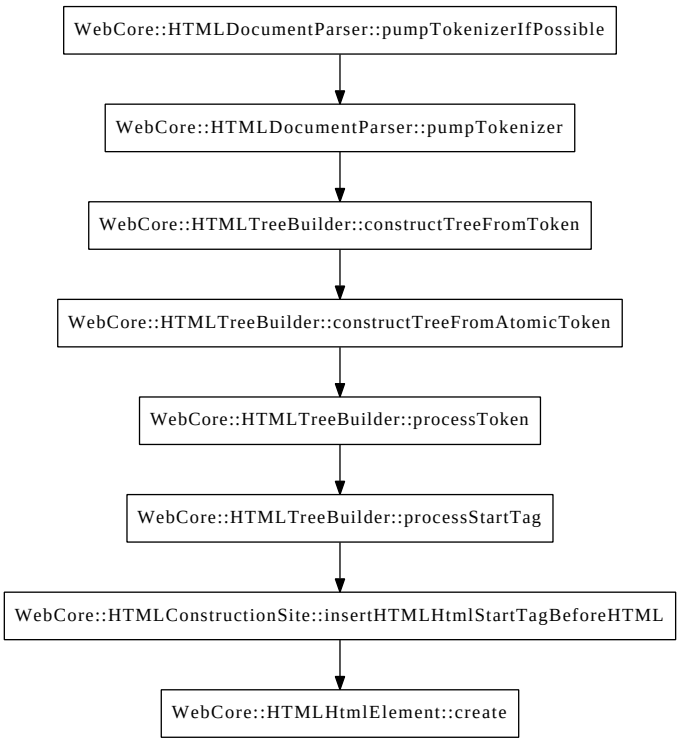

Figure 4: DOM 树非根节点的创建

./Tools/Scripts/build-webkit --qt --debug

4 WebKit Chromium Port

Chromium 采用的也是 WebKit 引擎，不过把 JavaScriptCore 替换成自家的 V8 引擎，在官方网页的文档上提到，做一个绝对安全和永远不会 crash 掉浏览器几乎是不可能的，所以需要有一些架构来退其次的保证浏览器的安全和良好的用户体验，针对安全 Chromium 提出并实现了 sandbox 模型，针对用户体验 Chromium 采用多进程架构来尽可能的不让用户感觉到任何阻塞的行为。主要分 Browser 和 Renderer 两大进程，之间通过有名管道来实现通信（IPC 机制）。

4.1 FileAPI

> Chromium fileapi is so messsy with its multiple process architecture
>
> Chromium FileAPI author: Eric

这是我在询问 FileAPI 实现细节时，Google 工程师给我的回答，所以用 gdb 跟踪一番，发现确实如此，render 端负责将 fileapi 的调用从 javascript 侧调用到 WebKit/WebCore/fileapi 中定义的接口，其次，chrmoium 在 WebKit/WebKit/中加了一层胶水层用以封装 WebKit 中 Fileapi 接口的定义。然后交由 FileDispatcher 通过封装过的有名管道 IPC 发送到 Browser 端。继而在 Browser 端由 FileDispatcherHost 接收消息，然后分发至对应的接口处理。一个 readDirectory 接口具体跟踪如下：

webkitRequestFileSystem in JavaScript
V8 进行操作，会判断是否是 apicall，如果是，则通过 V8 机制调用 WebKit 接口。

third_party/WebKit/Source/WebCore/fileapi/DirectoryReader.cpp
DirectoryReader::readEntries

third_party/WebKit/Source/WebCore/fileapi/DOMFileSystemBase.cpp
DOMFileSystemBaseo:readDirectory

third_party/WebKit/Source/WebKit/chromium/src/AsyncFileSystemChromium.cpp
AsyncFileSystemChromium::readDirectory

content/common/file_system/webfilesystem_impl.cc
WebFileSystemImpl::readDirectory

content/common/file_system/file_system_dispatcher.cc
FileSystemDispatcher::ReadDirectory

==========Render Process File System 部分调用结束 ==========

content/browser/file_system/file_system_dispatcher_host.cc
FileSystemDispatcherHost::OnReadDirectory

browser 进程接收 render 进程 filesystem IPC 消息

webkit/fileapi/file_system_operation.cc
FileSystemOperation::ReadDirectory

webkit/fileapi/file_system_file_util_proxy.cc
FileSystemFileUtilProxy::RelayReadDirectory

========== 发往 file_thread==========

webkit/fileapi/obfuscated_file_util.cc
bfuscatedFileUtil::ReadDirectory

继续往下去会发现 file system 中保存的文件相对于 OS fs 来讲，位置在.config/chromium/Default/File System/，并且用的是自家的 NoSQL 数据库:leveldb 此外在 chromium 源码路径:base/file_util_posix.cc 中定义了跟本地文件系统交互的地方，删除操作如 removeRecursively 可以断到这里查看详情。

4.2 DOM/Render 树生成

出于安全性、cookie、http request 次数考虑，所有与网络与本地系统交互的地方统一交由 Browser 进程去做，这样 WebKit 中资源 Load 一块就被 Chromium 废掉了，相应的在./webkit 中进行实现。资源 Load 完毕后将数据交给 HTMLDocumentParser，然后 HTMLDocumentParser 将文本字符的解析交给 HTMLDocumentTokenizer 来负责，HTMLDocumentTokenizer 解析出一个一个的标签（html 文档时是以标签为单位）,HTMLDocumentParser 将标签交给，HTMLTreeBuilder 来构建 dom 树。有几个基础元素：

- FrameView 负责 laying out rendering tree
- RenderView Render 树的根节点
- RenderObject 可见对象，即 render tree 上的节点数据结构，在 DOM 上都有对应。

4.3 图形系统分析

TODO