理解WebKit和Chromium

朱永盛 著

版权说明

书 名 《理解 WebKit 和 Chromium》
著 朱永盛
版本信息 2013 年 5 月
定 价 0.00 元

前言

这几年来，我阅读了很多 WebKit 和 Chromium 的代码，也给它们提交过一些的代码补丁包(patch)，一个感觉是代码量真的很大，常常看了这块忘了那块；另一个感觉是文档真的很少，特别是 WebKit（chromium 有不少设计文档，但是还不够）。这让我觉得非常痛苦，常常摸不着头绪。鉴于自己的经历，觉得很有必要把阅读代码后的理解和总结记录下来，一来帮助自己回忆，二来可以分享给其他人，三来方便大家一起交流。

这将会是一个系列，该系列的介绍方式会以一个个专题的形式来给出，例如 WebKit 的 DOM 树，Render 树，Chromium 多进程模型，消息处理，IPC 等等，每个专题大概分成以下几个部分来描述：

了解背景知识，理解基本的结构和流程;
熟悉各个模块的架构和设计，以及它们是如何工作的;
帮助阅读和理解 chromium 的代码。

根据这些专题所涉及的内容，大概把它们分为三个部分，第一个部分是基础篇，第二部分是高级篇，第三部分是开放篇。

基础篇

根据这些专题所涉及的内容，大概把它们分为三个部分，第一个部分是基础篇，第二部分是高级篇，第三部分是开放篇。

WebKit, WebKit2, Chromium 和 Chrome 介绍

概述

在介绍本系列各个专题之前，有必要先解释一下极其容易混淆的几个概念，它们是 WebKit，WebKit2，Chromium 和 Chrome。

首先来了解 WebKit。广义上来说，WebKit 是一个开源的项目，其前身是来源于 KDE 的 KHTML 和 KJS。该项目专注于网页内容的展示，开发出一流的网页渲染引擎。它不是浏览器，而且也不想成为浏览器。 该项目包含两个部分，第一是 WebCore，其中包含了对 HTML，CSS 等很多 W3C 规范的实现；第二部分就是狭义上的 WebKit，它主要是各个平台的的移植并提供相对应的 Web 接口，也就是 WebView 或者类似 WebView，这些接口提供操作和显示网页的能力。目前使用 WebKit 的主流的浏览器或者 WebView 包括 Chrome, Safari, QtWebKit, Android Browser 以及众多的移动平台的浏览器。

WebKit2 相对于狭义上的 WebKit 而言，它不是 WebKit 简单的第二个版本，它是一个新的 API 层，其最主要的变化在于将网页的渲染置于单独的进程，而接口层则在另外一个进程，它们之间通过 IPC 来通讯。对于接口的调用者来说，中间的 IPC 和底下的实现是透明的，这样做的好处有很多，一个很明显的好处是，当网页的渲染出现问题时，不会阻碍 Web 接口的调用者进程，这会在很大程度上解决或者帮助解决浏览器或者这些调用者的稳定性和安全性等问题。

Chromium 是一个建立在 WebKit 之上的浏览器开源项目，由 Google 发起的。该项目被创建以来发展迅速，很多先进的技术被采用，如跨进程模型，沙箱模型等等。同时，很多新的规范被支持，例如 WebGL，Canvas2D，CSS3 以及其他很多的 HTML5 特性，基本上每天你都可以看到它的变化，它的版本升级很快。在性能方面，其也备受称赞，包括快速启动，网页加载迅速等。

Chrome 是 Google 公司的浏览器产品，它基于 chromium 开源项目，一般选择稳定的版本作为它的基础，它和 chromium 的不同点在于 chromium 是开源试验场，会尝试很多新的东西，当这些东西稳定之后，chrome 才会集成进来，这也就是说 chrome 的版本会落后于 chromium。另外一个就是，chrome 里面会加入一些私有的 codec，这些仅在 chrome 中才会出现。再次，chrome 还会整合 Google 的很多服务， 最后 chrome 还会有自动更新的功能，这也是 chromium 所没有的。

参考文献

http://www.webkit.org/
http://trac.webkit.org/wiki/WebKit2

WebKit 和 Blink

关注 Web 和 HTML5 领域的人最近应该都有了解 WebKit 项目的重磅消息，那就是 Google 退出 WebKit 项目，创建自己的渲染引擎 Blink。这其实不能说完全没有先兆，合合分分，纯属正常。其实，之前关于 WebKit2，双方的争论就非常的大。Apple 希望它可以随便加入和删除代码而无需担心它会破坏其它 Ports 的代码，这遭到很多人的反对和不满。同时，另一方面，Google 有很多新的功能希望加入 WebKit 中，但是 WebKit 可能并不认可他们。双方分歧越来越多，终于分道扬镳。

这里面有个误区，就是 Google 的 Blink 是一个全新的引擎。其实不是这样，Blink 目前就是从 WebKit 直接复制出一个版本出来，然后将与 chromium 无关的 Ports 全部移除掉，将代码结构重新整理，就目前而言，Blink 的渲染和 WebKit 是一样，但是，以后两者将各自走不同的路。这有点类似于之前 WebKit 从 KHTML 中复制出来一样，历史总是惊人的相似。

目前参与 Blink 和 Chromium 大致一样，拥有 Chromium 的 commit 权限对 Blink 也适用。原来一些 WebKit 的 committer 和 reviewer 也开始成为 blink 的 committer。它的提交代码流程，review 流程等都是 chromium 的风格，这对 chromium 的开发者来说非常熟悉。

Blink 从 WebKit 继承而来，那么未来它会在哪些方面做改变呢？根据 chromium 官方的说法，目前大概有两个比较大的，后面应该有更多的改变：

跨进程的 iframe(out-of-process iframes)：为 iframes 内容创建单独的沙箱进程来渲染它们
将 DOM 移入 JavaScript 中，这样 JavaScript 可以更快的访问 DOM

今后，Blink 会和 WebKit 差别越来越大，对 Web 标准支持也不尽相同，未来的发展如何，让我们拭目以待吧。 顺便插一句，以后可能要改这个系列的标题了，呵呵。

参考资料

http://www.chromium.org/blink

WebKit 和 Chromium 代码目录结构介绍

WebKit 和 Chromium 的代码量很大(这两个项目都是几百万行代码的级别，不包括它们依赖的第三方库），读起来是相当的不容易。但是良好的代码组织结构很好的帮助了开发者和学习者们，下面大致介绍一下它们的目录结构及其用处，方便了解和学习，进而快速地理解整个项目。因为目录实在太多，所以这里介绍其中主要的部分。

先来看看 WebKit。(WebKit 项目在 chromium 中的目录是 src/third_party/WebKit)

- src – 顶层代码目录
  - base – 基础类库
  - breakpad – 崩溃报告
  - build – 编译脚本和工具
  - chrome – 浏览器相关功能的实现
    - app – 程序的主入口
    - browser – 浏览器功能的实现
    - renderer – Renderer 进程中相关 Content API 回调函数和某些功能的实现
  - chromeos - ChromeOS 相关
  - content – 包括 Browser 进程和 Renderer 进程的页面内容显示和通信设施
    - app – Content API 的 app 部分接口的实现
    - browser – browser 进程部分接口的实现
    - common – 公共类
    - gpu – gpu 进程的管理和入口
    - plugin – plugin 进程的管理和入口
    - public – Content API 的接口
    - renderer – Content API 的 renderer 部分接口实现
  - crypto – 加密的基础类
  - googleurl – Google 的 URL 解析库
  - gpu – 跨进程的 GPU 硬件加速机制
  - ipc – 进程间通信机制
  - media – 音频视频的支持
  - native_client – Native Client 代码
  - native_client_sdk – Native Client SDK 代码
  - net – 网络协议支持库
  - ppapi – PPAPI 代码
  - sandbox – 沙箱模型
  - sync – 浏览器数据同步
  - third_party – chromium 所使用的第三方项目
    - WebKit – WebKit 代码
  - tools – 各式各样的工具脚本，例如提交代码
  - ui – 浏览器和网页的界面基础设施，包括 views，aura 和 gpu 底层支持
    - aura – aura 窗口管理器
    - base – 基础库
    - gfx – 硬件加速相关
    - ui_controls – ui 的界面元素
    - views – views 图形工具库
  - v8 - V8 引擎
  - webkit – 对 WebKit 中部分 HTML5 功能的实现

再来看看 Chromium。

WebKit 和 Chromium 功能模块

在“WebKit，WebKit2，Chromium 和 Chrome 介绍”中，大致了解了 WebKit 是一个渲染引擎，Chromium 是一个浏览器，它们那么分别包含哪些不同的功能模块？它们是如何划分地？本章节来为大家详细解读一下。

WebKit:

HTML 解析：负责 HTML 语言的解析
CSS 解析：负责 CSS 的解析工作
图片解码：支持不同编码格式的图片
JavaScript 引擎：JavaScript 语言的解析引擎，缺省的是 JavaScriptCore，但是目前 Google 的 V8 JavaScript 被广泛使用
正则表达式
布局：负责布局(layout)的计算和更新工作
文档对象模型(DOM)：DOM 是 W3C 定义的对象模型，该部分负责 DOM 树及其相应的接口
渲染：与渲染相关的基础设施，例如渲染树，渲染层次树等等
SVG：对 SVG 的支持
XML 解析：XML 语言的解析
XSLT：XSLT 语言的解析执行
URL 解析器：URL 规范的解析
Unicode 编解码器：各种编码解码工作
移植：WebKit 中比较大的一部分，因为 WebKit 要工作需要不同平台上有具体的实现，因而不同的移植有不同的实现。chromium 的移植很复杂，因为其支持跨平台，所以它的移植需要在 windows，linux 和 mac 上工作。 由上面的模块大致可以 WebKit 主要是跟网页的解析和渲染相关的工作，其不涉及浏览器的历史，书签，下载，cookie 管理等等方面的工作。

Chromium:

Cookie 管理器：cookie 生命周期的管理
历史管理器：历史记录的管理
密码管理器：网页中密码登录信息管理
窗口管理：多个 Tab 窗口的管理和切换
地址栏：地址栏功能，智能地址填充与书签的协同工作
安全浏览黑名单管理：安全浏览机制
网络栈：与网络传输相关的工作，其有很多创新的东西
SSL/TLS：网络传输安全
磁盘缓存：磁盘缓存页面及其替换策略等生命周期的管理
下载管理器：管理下载相关
粘帖板：clipboard 的功能
书签管理：书签的组织和管理
URL 解析器：同 WebKit
Unicode 编解码器：同 WebKit

Chromium 主要是实现浏览器相关的功能，如上面中的网络栈等等。其实以上只是一些浏览器基本功能，chromium 实现的远不止这些，这其中包含沙箱模型，NaCl，扩展机制，硬件加速架构等等。这些我们将在之后的章节中逐一介绍它们。

URL 解析器和 Unicode 编解码器在两者中都存在是因为它们都要使用到。

Chromium 界面（UI）

Chromium 的界面相当简洁，这是她的设计理念。大体上可以把界面分成两个主要部分：网页内容和外边的修饰控件（例如，tab 管理，工具栏，设置按钮等）。

整个 chromium 浏览器是个顶层窗口。每个 tab 都对应一个顶层窗口的子窗口，每个网页内容都会绘制在一个子窗口中。当然这个是现有的窗口结构，但在新的 views 框架中，窗口将会被移除，详细的后面有专门介绍。 Chromium 界面另一个主要的控件是设置按钮，里面包含了所有有关 chromium 属性设置的部分。值得一提的是，里面有很多设置界面都是由 HTML 来撰写的，而不是传统的语言，例如 c/c++。这很大程度上得益于 chromium 的扩展机制及其提供的 API，这会在扩展章节详细介绍。

大家可能会觉得 chromium 界面简洁，用户或者能看到的浏览器信息有限，其实不然。尝试在地址栏里输入 chrome://chrome-urls/，你会看到很多的 chrome 地址。这些地址提供给用户或者开发者关于浏览器的丰富的信息，可以说是包罗万象，你能想象的信息基本都能从这里看到。这些信息其实非常的有用，特别对于理解 chromium 的内部机制非常有帮助，很多细节我们会在后面的章节中逐一揭露。

下面节选自 chrome://chrome-urls/的输出：

List of Chrome URLs

chrome://appcache-internals
chrome://blob-internals
chrome://bookmarks
chrome://cache
chrome://chrome-urls
chrome://crashes
chrome://credits
chrome://dns
chrome://downloads
chrome://extensions
chrome://flags
chrome://flash
... ...

Chromium 多进程模型

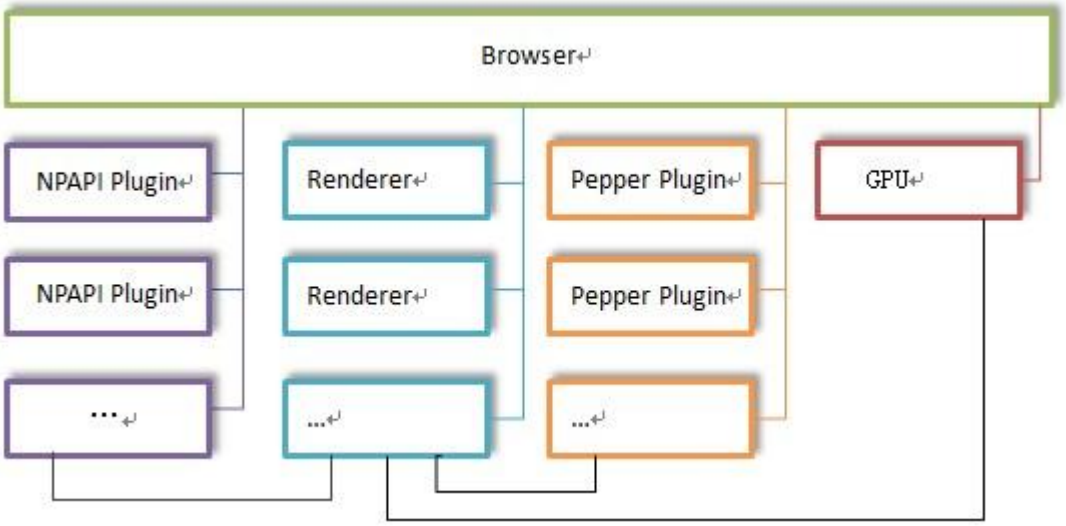
概述

相信你一定有这样的经历：打开很多个页面，不幸的是其中某个页面不响应了或者崩溃了，随之而来的是更不幸的事，所有页面都不响应或者都崩溃了。最让人崩溃的是其中一些页面还有未保存或者未发送的信息！

这绝对是不堪回首的过去。但是，现在好了，现代浏览器很多都支持多进程模型，这个模型可以很好地避免上面的问题，虽然它很复杂而且也有自身的问题，例如更多的资源消耗，但是它的优势也是非常明显地。

chromium 的多进程架构至少带来三点好处，其一是避免单个页面的不响应或者奔溃影响整个浏览器的稳定性；其二是当第三方插件奔溃时候不会影响页面或者浏览器的稳定性；其三是方便了安全模型的实施，也就是说沙箱模型是基于多进程架构的。其实，这很大程度上也是 WebKit2 产生的原因。那么，这是怎么做到的呢?

下图给出了缺省的 chromium 浏览器的进程模型。方框代表进程，连接线代表 IPC 进程间通信。



通常来讲，chromium 浏览器包括以下主要进程类型：

Browser 进程：浏览器的主进程，负责浏览器界面的显示，各个页面的管理，其他各种进程的管理；
Render 进程：页面的渲染进程，负责页面的渲染工作，WebKit 的工作主要在这个进程中完成；
NPAPI 插件进程：每种类型的插件只会有一个进程，每个插件进程可以被多个 Render 进程共享；
GPU 进程：最多只有一个，当且仅当 GPU 硬件加速打开的时候才会被创建，主要用于对 3D 加速调用的实现；
Pepper 插件进程：同 NPAPI 插件进程，不同的是为 Pepper 插件而创建的进程

Chromium 浏览器的进程模型，包括以下特征：

browser 进程和页面是分开的，这保证了页面的奔溃不会导致浏览器主界面的奔溃；
每个页面是独立的进程，这保证了页面之间相互不影响；
插件进程也是独立的，插件的问题不会影响浏览器主界面和页面；
GPU 硬件加速进程也是独立的。 因为这么多的进程，开发者通常需要知道进程列表中的进程类别，这很简单，可以通过进程的命令行参数"--type"来识别。 有趣的是，就在我写下上面这段文字的时候，我的 chrome 浏览器的 flash 插件崩溃了，幸运的是其他一切都很好，感谢 chrome 的多进程模型！

模型的类型

其实介绍了进程模型，其实 Chromium 支持多种进程模型，特别是对页面而言，下面简单的介绍以下模型的类型：

Process-per-site-instance

该类型的含义是对同一个域的实例都会创建独立的进程。举个例子来讲，例如，用户访问了 milado_nju 的 CSDN 博客（我的博客），然后从个人主页打开多篇文章时，每篇文章的页面都是该域的一个实例，因而它们都共享同一个的进程。如果新打开 CSDN 博客的主页，那么就是另一个实例，会重新创建进程来渲染它。这带来的好处是每个页面互不影响，坏处自然是资源的巨大浪费。

Process-per-site

该类型的含义是不同一个域会创建独立的进程，同一域的不同实例共享同一个进程。好处是对于不同的域可以共享，相对较小的内存消耗，坏处是可能会有特别大的 Renderer 进程。可以在命令行加入参数--process-per-site 来尝试它。

Process-per-tab

该类型的含义是为每个标签页创建一个独立的进程，这也是 chrome/chromium 的缺省行为

Single process

该类型的含义是不为页面创建任何独立的进程，所有渲染工作都在 browser 进程中。但是这个类型只是实验性质的，不稳定，因而不推荐使用，只有在比较单进程和多进程时候比较有用，可以在命令行加入参数--single-process 来尝试它。

沙箱模型

在页面的多进程模型中，页面的渲染是运行在沙箱模型中的 Render 进程中实现的，这些渲染引擎没有访问本地资源的能力（例如文件系统，窗口系统，等等），这可以保护渲染引擎被入侵。

参考文献

http://www.chromium.org/developers/design-documents/process-models

Chromium 的多线程机制

概述

前面我们介绍过 Chromium 是基于多进程模型的架构设计，那么各个进程内的情况呢？事实是每个进程都有很多的线程，特别是 browser 进程，因而它也基于多线程模型的。介绍多线程机制之前，先来看一下残酷的现实吧，下面是各个进程的线程信息情况(基于 Linux 平台，其它平台的可能略有不同)，相信保证让你头大。是的，你需要泡杯茶，然

browser 进程的线程信息

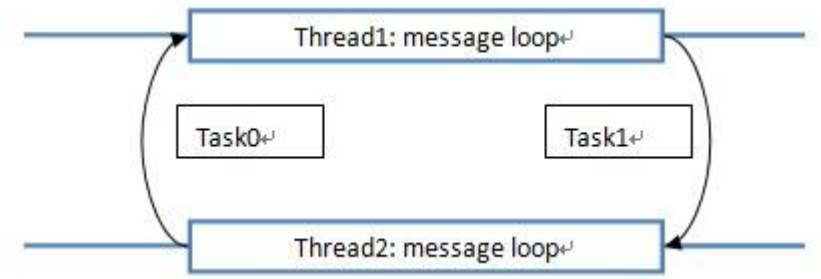
后静下心来了解一下它们：

为什么这么多的线程呢？Chromium 的官方说法告诉我们，主要目的就是为了保持 UI 的高响应度，保证 UI 线程（chrome 线程，主线程）不会被任何其它费时的操作阻碍从而影响了对用户的响应。这些其它的操作很多，例如本地文件读写，socket 读写，数据库操作等等。既然它们会阻碍其它操作，那好，把它们放在单独的线程里自己忙或者等待去吧，所以你就看到那么多与这些相关的线程（线程名显然也暴露了这一切）。

问题来了，它们之间如何通信和同步呢？这是多线程的一个非常难缠的问题，因为这会造成死锁或者竞争冲突等问题。Chromium 精心设计了一套机制来处理它们，那就是绝大多数的场景使用事件和一种 chromium 新创建的任务传递机制，仅在非用不可的情况下使用锁或者线程安全对象，这有严格的要求，详细的情况请查看以下链接以便作近一步的了解：http://www.chromium.org/developers/lock-and-condition-variable。

问题又来了，那么每个线程内部是如何处理这些事件和任务的呢？答案是 MessageLoop。每个线程会有一个自己的 MessageLoop，它们用来处理这些事件和任务。通常的 MessageLoop 只是处理事件，Chromium 中的 MessageLoop 可以同时处理事件和任务。MessageLoop 也是值得研究的，详细情况我们将在以后的一章加以介绍。

任务和 MessageLoop 的基本原理如下图所示。任务被派发到进程的某个线程的 MessageLoop 的队列中，MessageLoop 会调度执行这些 Task。



关于上面这些线程，多数可以通过它们的名字猜出用途，这里鉴于篇幅和噪音考虑，不一一介绍，下面说明几个重要和诡异的线程：

chrome 线程：进程的主线程，browser 进程重要主要是负责 UI，当然也是管家；Renderer 进程中则是管家兼处理 WebKit 渲染的；gpu 进程中则是负责处理处理绘图请求并调用 openGL 进行绘制工作的。

Chrome_IOThread/Chrome_ChildIOThread 线程：用来接受来自其它进程的 IPC 消息和派发自身消息到其它进程。

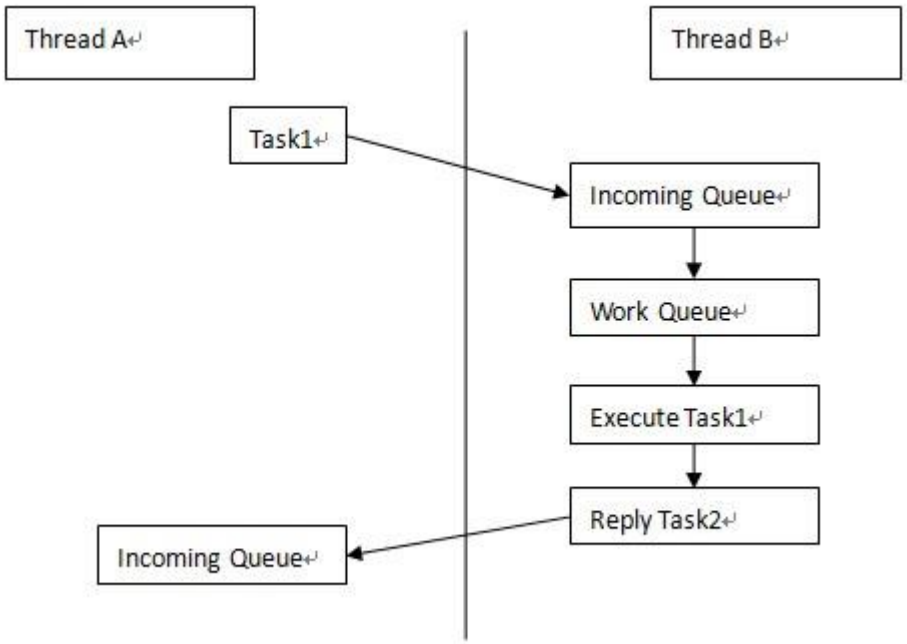
SignalSender 线程：V8 JavaScript 引擎中用于处理 Linux 信号的线程。

任务（task）

Chromium 的特色就是在事件的基础上，加入了一个新的机制－任务。当需要执行某个操作时候，可以把该操作封装成一个任务，由任务派发机制传递给相应的进程的 MessageLoop。 下面看看线程内和线程间分别是如何操作的。

首先看线程内部是如何进行的。但你需要进行费时的操作时候，可以派发一个事件和回调函数给自身线程的 MessageLoop，然后 MessageLoop 会调度该回调函数以执行其操作。问题是这有必要吗？直接调用不就可以吗？答案是不可以，或者说是最好不要这么做，其原因在于，如果当前的 MessageLoop 里面有优先级更高的事件和任务需要处理时，你这样做会阻碍它们的执行。 其次看一看线程间通信。假如一个线程 A 需要把任务传递给一个另外的线程 B，大致有三个阶段：

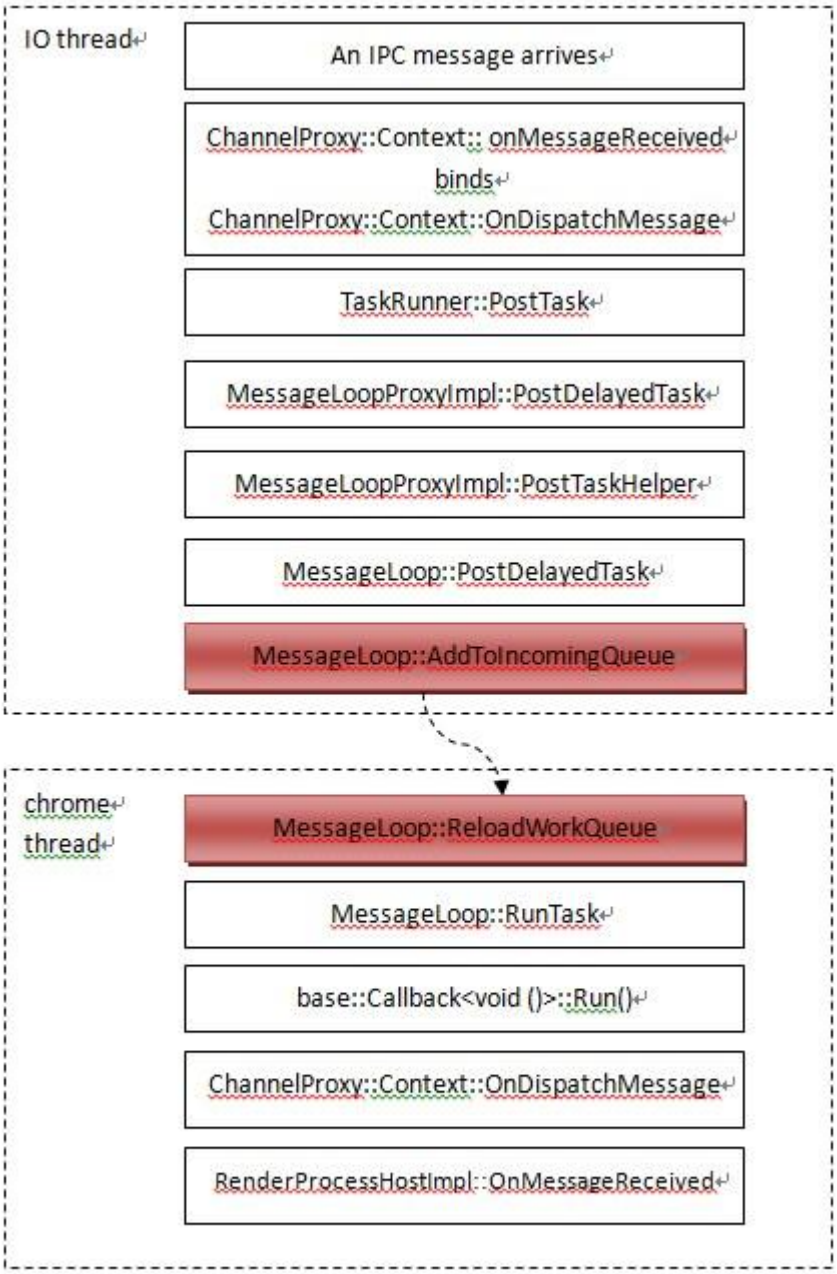
首先，线程 A 把该任务传递给线程 B；
其次，线程 B 调度执行该任务；
最后，线程 B 执行完任务后回复 A。很多情况下，线程 A 不需要回复。 下图描述的是一个带回复的典型的任务传递过程。



如果不需要回复，那么上图中的‘Reply Task2’就不需要了。当需要回复时候，chromium 的做法是新建一个新的任务，该任务封装原来的任务，新任务被放入线程 B，B 执行新任务时候调用其 Run 方法，里面首先执行原来的任务，然后派发 Reply 任务给线程 A，操作完成。 后面会有一个具体的例子来描述上面这个过程，下面了解一下 chromium 支持 Task 所涉及的几个主要类。 Callback：回调类，其本质是封装了一个由调用者（例如线程 A）设置的回调函数。包含一个 run 方法，当 MessageLoop 调度执行该 Task 时候，运行该方法，该方法调用回调函数 Closure：定义为 Callback<void(void)> tracked_objects:一系列的类用于追踪 Task 生成位置，编译调试 Task：包含一个 Closure，追踪信息，表示一个任务

一个例子

下面用一个实际的例子来理解线程间是如何传递任务的。该例子是 chromium 中非常常见的一个场景：browser 进程接收到来自 renderer 进程的 IPC 消息，它由 IO 线程接收管理，然后派发给 chrome 线程处理，具体过程如下图所示：



注：红色表示该步骤需要加锁

上面的图基本上对应了前面的关于任务派发过程的图，只是少了回复环节，这是因为 IO 线程并不需要回复。基本的步骤是，当 IO 线程收到消息后，其派发一个任务，该任务将 ChannelProxy::Context::OnDispatchMessage 设置会回调函数，这个任务保存在 chrome 线程的输入任务队列中。chrome 线程将输入队列拷贝到工作队列后，执行该任务的 run 方法，该方法会调用回调函数 ChannelProxy::Context::OnDispatchMessage，该函数会调用事件的处理函数，这里也就是 RenderProcessHostImpl::OnMessageReceived，这个函数实际上会根据事件类型来调用各个相应函数。

源文件目录

```
base/threading/
        线程相关的基础类
```

参考文献

http://www.chromium.org/developers/design-documents/threading
http://bigasp.com/archives/478
http://www.chromium.org/developers/lock-and-condition-variable

消息循环

概述

前面介绍了线程间如何传递 chromium 自定义任务(task)，那么在线程内，消息循环（messageloop）是如何处理这所有的消息和任务呢？本章节重点介绍消息循环的工作原理。

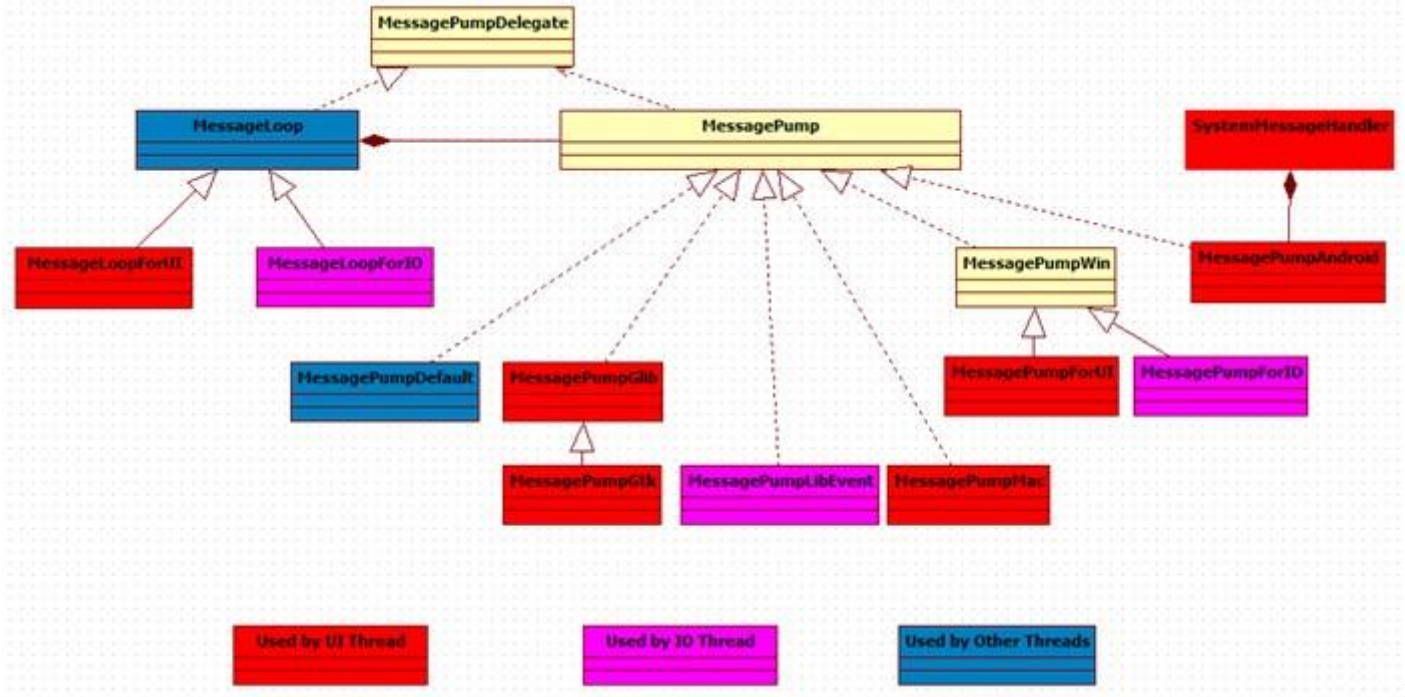
在 Chromium 里，需要处理三种类型的消息：chromium 自定义的任务，Socket 或者文件等 IO 操作以及用户界面(UI)的消息。这里面，chromium 自定义任务是平台无关的，而后面两种类型的消息是平台相关的。回忆一下前面多线程模型章节中列举的众多线程，例如主线程（UI 线程）需要处理 UI 相关的消息和自定义任务；IO 线程则需要处理 Socket 和自定义任务；History 线程则只需要处理自定义任务；其它线程需要处理消息类型不会超出以上三个线程。根据三个组合，chromium 定义和实现三种相对应的类来支持它们，下面让我们来看看具体的实现吧。

Chromium 中主要的类

这是本章中最重要的三个基类，依次介绍它们： 类 RunLoop：一个辅助类，主要封装消息循环 MessageLoop 类，其本身没有特别的功能，主要提供一组公共接口被调用，其实质是调用 MessageLoop 类的接口和实现。

类 MessageLoop：主消息循环，原理上讲，它应该可以处理三种类型的消息，包括支持不同平台的消息。事实上，如果让它处理所有这些消息，这会让其代码结构复杂不清难以理解。因此，根据上面对各个线程的分析，消息循环也只需要三种类型，一种仅能处理自定义任务，一种能处理自定义任务和 IO 操作，一种是能处理自定义任务和 UI 消息。很自然地，Chromium 定义一个基类 MessageLoop 用于处理自定义任务，两个子类对应于第二和第三种类型。如下图所示。对于第二和第三种 MessageLoop 类型，它们除了要处理任务外，还要处理平台相关的消息，为了结构清晰，chromium 定义一个新的基类及其子类来负责处理它们，这就是 MessagePump。MessagePump 的每个子类针对不同平台和不同的消息类型。事实上，不仅如此，消息处理的主循环也在 MessagePump 中，这有些令人意外，后面会详细介绍。因此，MessageLoop 通过实现 MessagePumpDelegate 的接口来负责处理 Chromium 自定义任务。如下图所示。

类 MessagePump: 一个抽象出来的基类，可以用来处理第二和第三种消息类型。对于每个平台，它们有不同的 MessagePump 的子类来对应，这些子类被包含在 MessageLoopForUI 和 MessageLoopForIO 类中。



结

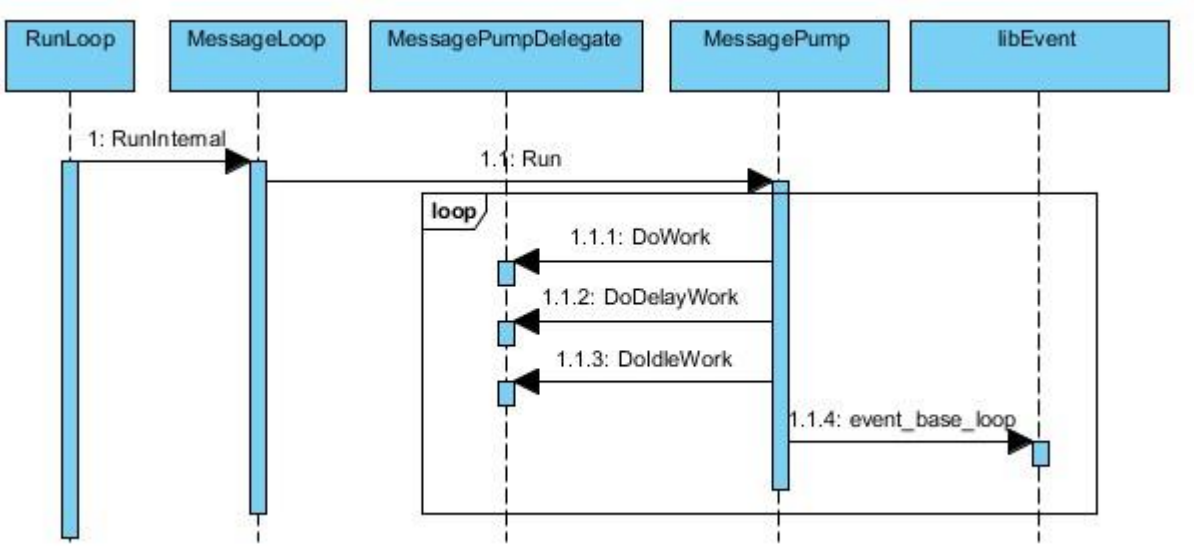
合上面的图，针对三种类型的消息循环，三种典型的线程在不同平台所使用的类如下：

	主线程	IO线程	History线程
Linux	MessageLoopForUI, MessagePumpGtk	MessageLoopForIO, MessagePumpLibEvent	MessageLoop, MessagePumpDefault
Windows	MessageLoopForUI, MessagePumpForUI	MessageLoopForIO, MessagePumpForIO	MessageLoop, MessagePumpDefault
Mac	MessageLoopForUI, MessagePumpMac	MessageLoopForIO, MessagePumpLibEvent	MessageLoop, MessagePumpDefault
Android	MessagePumpForUI Android.os.Looper SystemMessageHandler MessagePumpAndroid	MessageLoopForIO, MessagePumpLibEvent	MessageLoop, MessagePumpDefault

对于所有平台来说，History 线程所使用的类是一样的；UI 线程和 IO 线程分别对应不同的 MessagePump。相信大家注意到了，最后一个平台 Android 跟其它的稍有不同，那就是它的 UI 线程，原因在于主循环在 Java 层，用户界面的事件派发机制都在 Java 代码来处理，因而需要在 Android 的消息循环机制中加入对自定义任务的处理，后面会作介绍。

无限循环

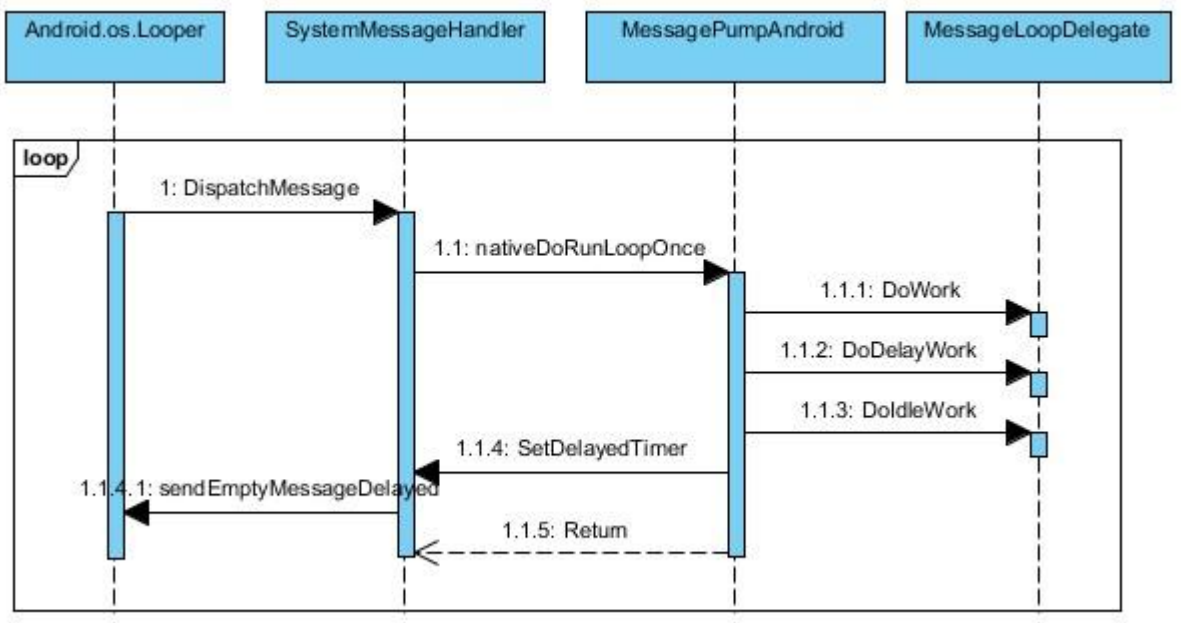
这里面还有一个重要的部分需要介绍，那就是消息循环的主逻辑。该循环本质就是一个无限循环，不停的处理消息循环接收到的任务和消息，直到需要推出为止。如前面所述，主消息循环的逻辑在 MessagePump 中，而 MessageLoop 只是负责处理自定义的任务，MessagePump 通过调用 MessagePumpDelegate 来达到该目的。下面是 IO 线程的消息处理主逻辑的时序图。



前面章节我们介绍过 MessageLoop 有任务队列来保存需要处理的任务，这些任务可能有不同的优先级，例如需要即时处理，或者延迟处理，或者 Idle 时处理。上图中，当调用 DoWork 时候，首先将出入任务队列(incoming task)拷贝到工作任务队列中，然后依次执行该队列中的任务。之后，同理处理调用 DoDelayWork 和 DoIdleWork。当这些处理完后，它会阻塞和等待在 IO 操作。

对于 Android 的 UI 线程来说，情况稍有不同，因为主循环逻辑是由 Java 层的 Android.os.Looper 来控制的，所以 MessagePumpAndroid 其实是被 Looper 调用的。当它被调用时，其会把执行任务的工作交给 MessagePumpDelegate，这里也就是 MessageLoopForUI 来完成，如下图所示的 UI 线程的消息循环主逻辑。Java 层的 SystemMessageHandler

和 C++层的 MessagePumpAndroid 其实都是辅助 Looper 调用执行 chromium 的自定义类的。



最后还有一个问题，就是如何等待自定义的任务。假设现在 MessageLoop 没有任务和消息需要处理，它应该等待 Socket 或者 IO 或者任务，OS 系统支持 Socket 和 IO 唤醒它，问题是如何等待自定义任务呢？总不能忙式的检查吧，那样太耗费资源了。Chromium 设计了一个巧妙的方法来解决该问题，以 MessagePumpLibEvent 为例：在 Linux 平台上，该类创建一个管道，它等待读取这个管道的内容，当有自定义的新任务到来时，写入一个字节到这个管道，从而 MessageLoop 被唤醒，非常地简单和直接。

源文件目录

```
base/message_ loop.h|cc

base/message_pump.h|cc

  消息循环相关的众多类基本上都位于该目录中，文件名以”message_”开头
base/android/java/src/org/chromium/base/SystemMessageHandler.java
 Android 平台下支持消息循环的辅助类
```

参考文献

http://bigasp.com/archives/478

HTML 解析和 DOM

概述

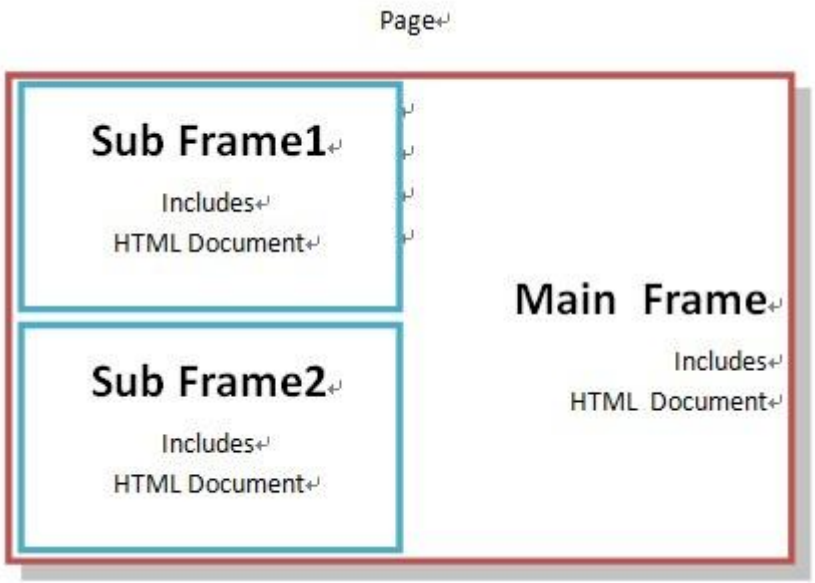
前面介绍了很多眼花缭乱的新技术，关于渲染，关于硬件加速，关于布局，关于其他很多，同大家一样，我也花了很多时间来消化它们。本章介绍稍微基础些的话题（本系列的写作顺序完全是随心所欲地），就是在渲染整个过程的初始阶段---HTML 解析。不过这不表示它简单，其实这里是非常绕人的。在前面描述渲染过程，其实也是回避了这些方面的很多细节，原因也很简单，我自己也没有完全仔细地了解清楚. :-(

现在又重新阅读和 debug 一下代码，因此，借用本章节，想完整的描述一下从 HTML 及其资源（例如图片等）下载后到 DOM 树建立好这一段过程的细节及其 WebKit 所涉及的这一切的一切。大体地，本章内容主要包括网页结构描述，WebKit 对网页结构的表示，WebKit 解析 HTML 的基础设施，一般过程，DOM 规范和 DOM 树。

按照惯例，这里给出一个例子。个人崇尚简单，只要能说明问题就行，如下所示，后面的解释和过程是基于此例子展开地。

```
1 <html>
2     <body>
3         <p> HTML DOM </p>
4         <div>
5             A Test Text.
6         </div>
7     </body>
8 </html>
```

让我们首先了解网页的基本结构，这样便于后面理解 WebKit 的相关设施。其实，它并不复杂，如下图所示。

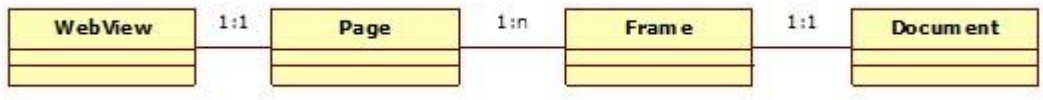


整个是一个网页，这里称之为 Page。每个 Page 都有一个主框（Main Frame），该框通常包含一个 HTML Document，主框也可能包含子框（sub frame），如图所示，这就是网页的多框结构，虽然这一结构饱受诟病，但现实还是这样。现在，更多的网页仅有主框，这样便于搜索引擎的检索和处理。本章的例子是一个仅有主框的最简单的网页结构。

这些框构成一个树型结构，以主框为根节点，每个框也可能包含自己的 HTML Document，它是一颗 DOM 树。

WebKit 设施

介绍了网页的基本结构，那么 WebKit 的设施也相对容易理解，下图是 WebKit 相对应的类，和网页的结构是一一对应的。这其中 WebView 是网页对外的接口。

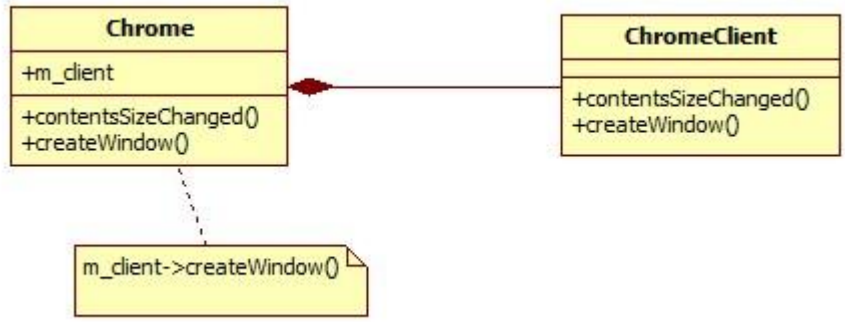


在介绍 WebKit 中其他类之前，有必要先介绍一个 WebKit 普遍使用的设计模式。首先看一个例子。下图是 Chrome 和 ChromeClient，这两个类非常重要。此 Chrome 非彼 Chrome，这里的 Chrome 是 WebKit 的一个类，不是 Google 的浏览器产品 Chrome。类 Chrome 后面会介绍，这里强调的是因为 WebKit 有很多种移植(port)，所以不同移植中的 Chrome 需要有不同的实现，所以 Chrome 类需要满足两类要求：

1） Chrome 需要具备有获取各个平台资源的能力，例如 WebKit 可以调用 Chrome 来创建一个新窗口；

2） Chrome 需要把 WebKit 的状态和进度等信息分发给外部的调用者或者说是 WebKit 的使用者；

WebKit 内部跟这两类要求相关的需求都是通过 Chrome 的接口来完成，这时候有个问题，那就是如何让 WebKit 和外部调用者既不紧密耦合，而能方便支持不同的平台？WebKit 使用 ChromeClient 抽象类来实现。每个 port 实现类 ChromeClient，一方面监听 WebKit 状态，一方面返回 WebKit 所需要的资源和信息。WebKit 直接调用 Chrome 的接口，Chrome 调用 ChromeClient 的接口，而 ChromeClient 的实现由各个移植来完成。



在这一部分中，很多都是该模式的类组合，例如 FrameLoader 和 FrameLoaderClient，ImageLoader 和 ImageLoaderClient，ContextMenu 和 ContextMenuClient 等等。这些类比较容易理解，不再阐述，这里简单介绍几组类的关系（很多其他的组类似）：

Frame 和 FrameLoader：Frame 表示的是页面框和框的加载器，一个负责页面的表示，一个负责加载需要的接口及实现，还有很多类似的组合类，例如 Document 和 DocumentLoader，CachedImage 和 ImageLoader 等。

WebView 和 Page：二者一一对应，Page 是 WebKit 内部表示网页的类，WebView 是 WebKit 对外表示网页的类，Page 只有一个实现，WebView 在不同的移植中有不同的实现。其他类似的组合类如 WebFrame 和 Frame。

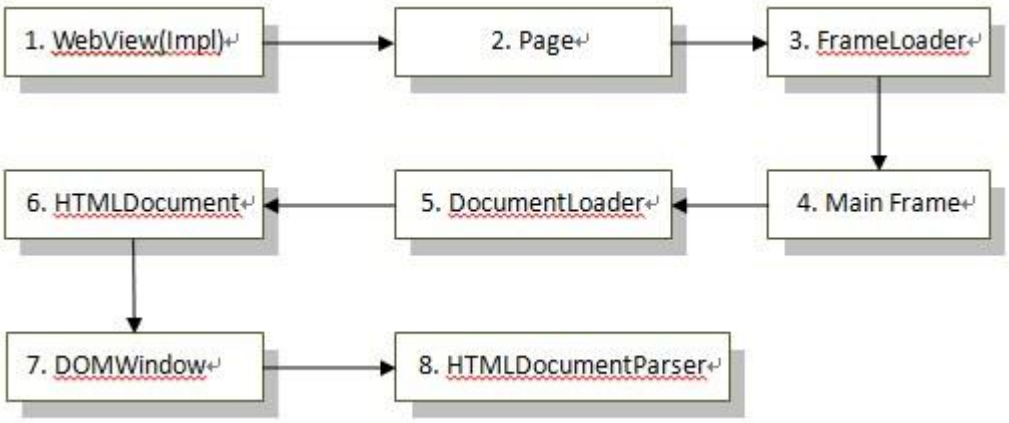
最后接上面介绍的类 Chrome，它是一个非常重要的类，是 WebKit 与它的使用者之间的桥梁，主要负责 UI 和渲染相关的需要用到平台相关接口。以下是它的一些主要功能：

跟 UI 和渲染显示相关的需要移植实现的接口集合类；
继承自 HostWindow（宿主窗口），其包含一系列接口，用来通知重绘或者更新相应整个窗口，滚动窗口等等；
窗口相关操作，例如显示，隐藏等；
显示/隐藏窗口中的 toolbar，statusbar，scroolbar 等；
显示 JavaScript 相关的窗口，例如 JavaScript 的 Alert，confirm, prompt 窗口等；

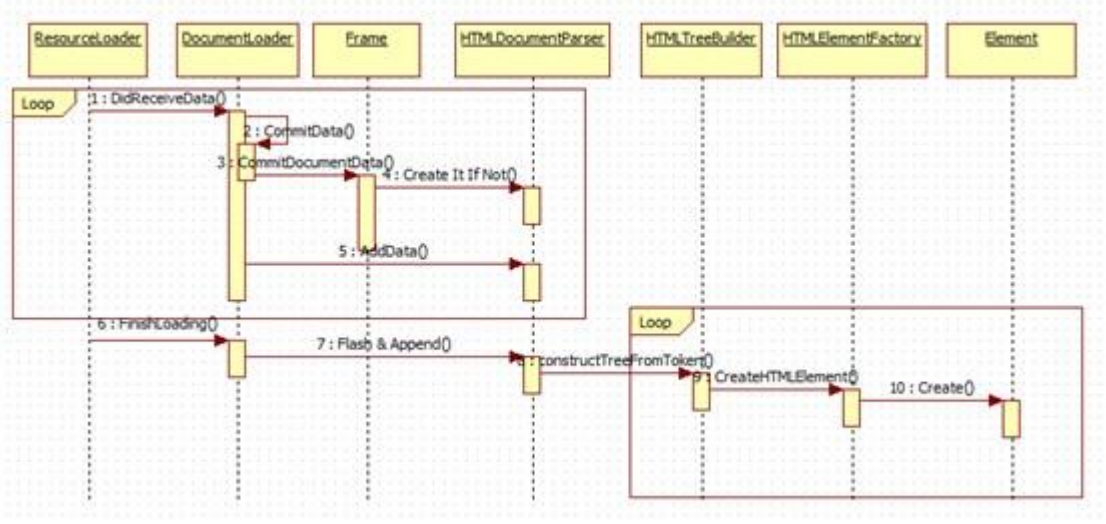
其他一些跟显示相关的，例如 colorchooser 等。

HTML 解析的一般过程

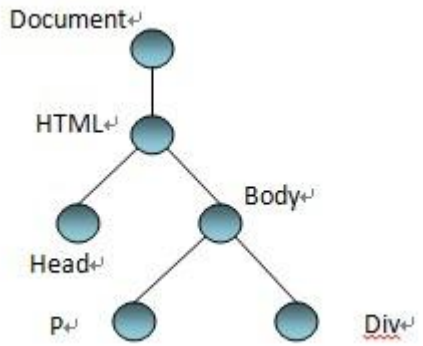
首先介绍主要对象的创建顺序，如下图所示，简单明了。



下面是解析和创建 DOM 树的一般过程，依旧是用图说话，为简单起见，省略其中的若干类和步骤。



上述图中的顺序，简单来讲是，先接受服务器发送过来的数据，之后解析成一系列的 Tokens，然后在加载结束后建立 DOM 树，这还是比较容易理解地，不再详细描述。下图是解析完例子后所创建的 DOM 树。简单吧？我也觉得:-)



DOM 标准

DOM 定义的是一组平台无关和语言无关的接口，该接口允许编程语言动态访问和更改结构化文档。本人(:-))见的比较多的应用对象是 XML 和 HTML。W3C 标准化组织定义一系列 DOM 接口，随着时间的发展，目前包括 DOM level 1, 2, 3。每个新版本是对以前版本的补充和新功能的加入。下面是对这三个版本的简单介绍： DOM level 1: 1. Core：一组底层的接口，其接口可以表示任何结构化的文档，同时也允许对其进行扩展，典型的例子是支持 XML 文档。 2. HTML：一组基于 Core 定义的接口的上层接口，主要是为了方便 HTML 文档的访问。 DOM level 2： 1. Core：对 DOM level1 中 core 部分的扩展，其中著名的就是 getElementById（没用过的请举手），还有很多跟名空间(namespace)相关的接口； 2. Views：书上说 views 是“允许动态访问和修改文档内容的表示，主要是两个接口 AbstractView 和 DocumentView”，没接触过加不懂，谁来救救我？ 3. Events：这个很重要，引入了对时间的处理，个人觉得是个重要的变化，主要有 EventTarget， Mouse events 等接口，但不支持 Keyboard，这个在 DOM level3 才被加入； 4. Style(CSS)：加入接口可以修改样式属性； 5. Traversal and range：这个容易理解，就是遍历树（NodeIterator 和 TreeWalker）加上对制定范围的文档修改删除等操作； 6. HTML：扩充 DOM level1 的 HTML 部分，允许动态访问和修改 HTML 文档。 DOM level 3: 1. Core：加入了新的 adoptNode() 和 textContent 支持； 2. Load and save：动态加载和序列化 DOM 表示； 3. Validation：根据 scheme 验证文档的有效性； 4. Events：主要扩展对 keyboard 的支持。HTML5 和触屏技术如火如荼，所以对 Touch 的支持很快就会进入规范； 5. XPath： 使用 XPath1.0 来访问 DOM 树，XPath 是一种简单直观的检索 DOM 树节点的方式，具体见 W3CXPath 标准。 DOM 标准对具体的表示方法没有任何限制，只是定义了接口，因此它在现实中有很多种实现。DOM 的树状表示是其中比较普遍的方式，还可以是二进制表示，其有很多的优点（例如 binary xml），详情请 google 之。 好了，该结束本章了，但这不是全部，这部分还有很多的细节值得研究和探讨，例如个人比较关注的是 DOM 树上的事件（Event）分发机制。有兴趣的话，可以自行阅读以下目录中的 WebKit 代码，相信你会获益非浅。

源文件目录

```
third_party/WebKit/Source/WebCore/loader
  加载相关的类，例如负责加载 Document，Frame，image 等
third_party/WebKit/Source/WebCore/page
  表示页面的类，例如 page， frame 等， 及页面相关的设施，例如 contextmenu，JavaScript 内置的对象，例如，window,console, navigator, dom 等。
third_party/WebKit/Source/WebCore/dom
  DOM 相关的类，包括节点定义类
third_party/WebKit/Source/WebCore/html
 html 标注的相应的 DOM 节点类定义
third_party/WebKit/Source/WebCore/html/parser
 HTML 解析器相关类
```

参考文献

http://www.w3.org/DOM/
http://en.wikipedia.org/wiki/Document_Object_Model
http://www.w3.org/TR/DOM-Level-3-Events/
http://www.w3.org/TR/DOM-Level-3-Core/core.html
http://www.w3schools.com/htmldom/default.asp
https://developer.mozilla.org/en-US/docs/DOM_Levels
http://www.ibm.com/developerworks/cn/xml/x-keydom/
http://www.ibm.com/developerworks/cn/xml/x-keydom2/index.html
http://wenku.baidu.com/view/7fa3ad6e58fafab069dc02b8.html

CSS 基础

CSS 初探

概述

先谈谈 HTML 网页的开发者们所遭遇地痛苦和悲惨的经历。在 CSS 出现前或者出现早期，HTML 因为要设计不同风格和样式的元素，所以在不停地加入很多新的元素来表示，例如 p，span。然后，问题还是存在，那就是大量的使用表格（Table）元素来排列网页中的元素，这导致一些不好的问题，其一，Table 经常嵌 Table，导致网页较大，消耗带宽，其二，被搜索引擎解析后，其内容变得杂乱无章。庆幸地是，CSS 的出现极大地解决了这些问题。

CSS 的全称是 CascadingStyle Sheet, 中文名是级联样式表，是用来控制网页显示风格的，被广泛地使用在网页中，现在基本所有的浏览器支持它。其有一个比较重要的特征就是将网页的内容和展示的方式分离开，这很重要。另一个重要的特征是它很强大，不是一般的强大，特别是新的 CSS3 标准，不仅能提供对页面各个元素的精准控制，同时提供丰富多彩的样式。简而言之，CSS 是一种非常出色的文本展示语言。

本章将简单介绍 CSS 的一些基本功能，让你对它有个大概地认识，后面的一章中，我们将会介绍它们如何在 WebKit 和 chormium 中获得支持地。

下面给出一个虽然简单但是却展示了 CSS 众多特征的示例，CSS 的主要部分在包含元素 Style 中，也就是下例中从第 3 行到第 16 行，同时 JavaScript 中也有使用部分对样式的操作，后面的部分会对它们逐一加以解释。

```
1  <html>
2      <head>
3          <style type="text/css">
4              div  /* 选择器 */
5              {
6                  position: absolute; /* 位置 */
7                  top: 200px; /* 坐标 */
8                  left: 200px; /* 坐标 */
9                  width: 200px; /* 宽度 */
10                 height: 20px; /* 高度 */
11                 background-color: #efefef; /* 背景色 */
12                 color: green; /* 颜色 */
13                 border:2px solid black; /* 边框 */
14                 padding: 20px 20px 40px 40px; /* 填充大小 */
15                 opacity: 0.5;
16                 -webkit-transform:rotate2(10deg); /* WebKit内核支持的变换属性 */
17             }
18         </style>
19     </head>
20     <body>
21         <p> This is a css test! </p>
22         <div id="adiv" class="aclass">CSS Style and Transform</div>
23         <script>
24             var rotate = 10;
25             function loop()
26             {
27                 var elem = document.getElementById("adiv");
28                 var value =  "rotate2(" + rotate + "deg)";
29                 elem.style.webkitTransform = value;
30                 window.webkitRequestAnimationFrame(loop);
31                 rotate += 10;
32                 rotate = rotate % 360;
33             }
34             loop();
35
36         </script>
37     </body>
38 </html>
```

CSS 功能

选择器

CSS 的选择器是一组模式，用来匹配相应的 HTML 元素。但选择器匹配相应的元素时候，该选择器包含的各种样式的设置就会作用在选中的元素上。通过选择器，CSS 能够精准地控制 HTML 页面中的任意一个或者多个元素的属性。看上面的例子中第四行。该行中的’div’就是一个选择器，它属于元素选择器，其含义是选择该页面中的所有’div’元素。因为仅有第 20 行包含一个’div’元素，所以，该选择的选择结果就是该元素。那么，div 下面所设置的样式等属性（花括号内）都会作用在该元素，从第 6 行到第 16 行。

示例中的选择器仅是众多选择器类型中的一种，从 CSS1 到 CSS3，规范陆续地加入了多达 42 种选择器，极大地方便了开发者，下面介绍其中一些主要的选择器：

标签选择器：根据元素的名称来选择，例如例子中的选择器，可以选择一个或者多个
类选择器：根据类别信息来选择目标元素，可以选择一个或者多个，例子中选择 div 元素也可以- 使用类选择器，方法是”.aclass”；
ID 选择器：根据 ID 来选择目标元素，仅能选择一个，例子中选择 div 元素也可以根据属性选择器，方法可以是”#adiv”

属性选择器：根据属性来选择目标元素，可以选择一个或者多个，例子中选择 div 元素也可以使用属性选择器，方法是”div[id]”，”div[id=’adiv’]”，”div[id~=’di’]”，”div[id|=’ad’]”；
后代选择器：选择某元素包含的后代元素，可以选择一个或者多个，例子中选择 div 元素也可以使用后代选择器，方法是”body div”；
子女选择器：选择某元素包含的子女元素，可以选择一个或者多个，例子中选择 div 元素也可以使用后代选择器，方法是”body>div”；
相邻同胞选择器：根据相邻同胞信息来确定选择的元素，可以选择一个或者多个，例子中选择 div 元素也可以使用相邻同胞选择器，方法是”p+div”；
还有很多类型的选择器，例如伪类选择器，通用选择器，群组选择器，根选择器等等，这里不再一一作介绍。

介绍了选择器之后，后面还有个重要的问题，那就是优先级。因为多个选择器可能作用于同一个元素，它们设置的样式属性可能不一样，这种情况下，应该怎么确定使用哪种样式。

一般而言，选择器越特殊，它的优先级越高，也就是所选择器指向的越准确，它的优先级就越高。例如，如果用 1 表示标签选择器的优先级，那么类选择器优先级是 10，id 选择器就是 100，数值越大表示优先级别越高。所以，尽量使用精确控制的选择器，使用合理优先级的选择。

各种属性

从第 6 行到第 16 行设置选择的元素的样式属性值，大致把这些属性分成以下类别：

背景：通常有两种方式设置，一个是设置背景颜色（例子中的 background-color）,另外一种设置背景图片。
文本：设置文本缩进，对齐，单词间隔，字母间隔，字符转换，装饰，空白字符等
字体：设置字体属性，可以是内嵌的，也可以是自定义的方式，另外还可以设置加粗，变形等等。
列表：设置列表类型，可以以字母，希腊字母，数字等变好列表
表格：通过设置边框来达到表格的目的，设置是否把表格边框合并为单一的边框，设置分隔单元格边框的距离，设置表格标题的位置，设置是否显示表格中的空单元格，设置显示单元、行和列的算法等。
框模型(box model)：框模型定义了元素框处理元素内容，内边框，边框和外边距处理方式。在示例中包含属性’border’，’padding’分别表示边框和内边距。
定位：CSS 提供相对，绝对地位和浮动定位。示例使用了绝对定位，参见第 6 到第 8 行。 从示例中相信你可以看到，CSS 的基本单元就是选择器加上它所包含的各个属性设置，参看示例中第 4 行到第 17 行，CSS 就是由多个这样的基本单元所组成。在编写 CSS 的时候，通过选择器来精确控制需要选择的元素，然后通过设置属性值来让这些元素展示出不同的显示效果。

CSS3 新增功能

选择器：上面介绍属性选择器就是 CSS3 新加入的，除此之外，还加入了精确控制的选择器用来选择特定位置的子女，特定元素标签的子女等等。
样式：增加了一些比较好的功能，例如自定义字体，圆角属性，边框颜色等等
变换，过渡和动画(transform,transition, animation)：CSS3 提供令人惊奇的变好，转变和动画功能，另其更加的赏心悦目。规范的草案中定义了 2D 的变换，更为吃惊的是 WebKit 提供了 3D 的变换。变换有三种类型，平移，旋转和缩放。同 2D 不同的是，3D 增加了绕 Z 轴的平移旋转和缩放。有一点颇令人遗憾，那就是各个不同的浏览器对这些属性的名字定义不一致，例如标准对变换的定义属性名是’transform’，而 webkit 的是’-webkit-transform’，如例子中第 15 行所示，IE 的’-ms-transform’，firefox 的则是’-moz-transform’,opera 的是’-o-transform’，这不免令人心烦意乱。过渡（transition）描述了属性从一个值过渡到另一个值 的过程，定义了过程的时间，启动过程的延迟时间等等。但是，这些标准草案中的定义还不足以描述更精确的变化过程，所以引入了更为灵活的方式，这就是 CSS 动画(animation)。通过动画，你能够定义不同的 keyframes 来控制中间变化过程而不仅仅是开始和结束。你可以这么理解，过渡是一种较为简单和常见的动画。

CSS 和 JavaScript

在新的浏览器中，Javascript 也可以方便且简单地来操作设置 css 的值，看看例子第 24 行到 34 行的代码。在函数’loop’中，但得到元素’adiv’后，可以通过设置它的 style 属性来设置各个 CSS 的属性值，例如本例中是要设置变换的不同角度，含义是通过不断地改变’webkitTransform’的值来让’adiv’元素绕 Z 轴旋转起来，效果相当酷。

另外一个非常不错的功能是，规范中引入了两个新的 JavaScript 接口： querySelector 和 querySelectorAll。这两个接口让 CSS 定义的所有的选择器都可以作为参数传给这两个接口，从而获取到相应的 HTML 页面中的元素，这非常的有用，你值得试试。chromium，safari 和 Firefox 都支持它。

在这一节结束前，强烈建议你将该示例在浏览器中尝试（最好是 chrome 或者 safari，如果是其它浏览器，可能需要做相应的属性名修改），同时，逐一注释掉每个属性，看看它怎么影响最终的显示效果。

参考文献

http://www.webkit.org/projects/layout/index.html
http://en.wikipedia.org/wiki/Tableless_web_design
http://www.w3schools.com/cssref/css_selectors.asp

WebKit 渲染基础

概述

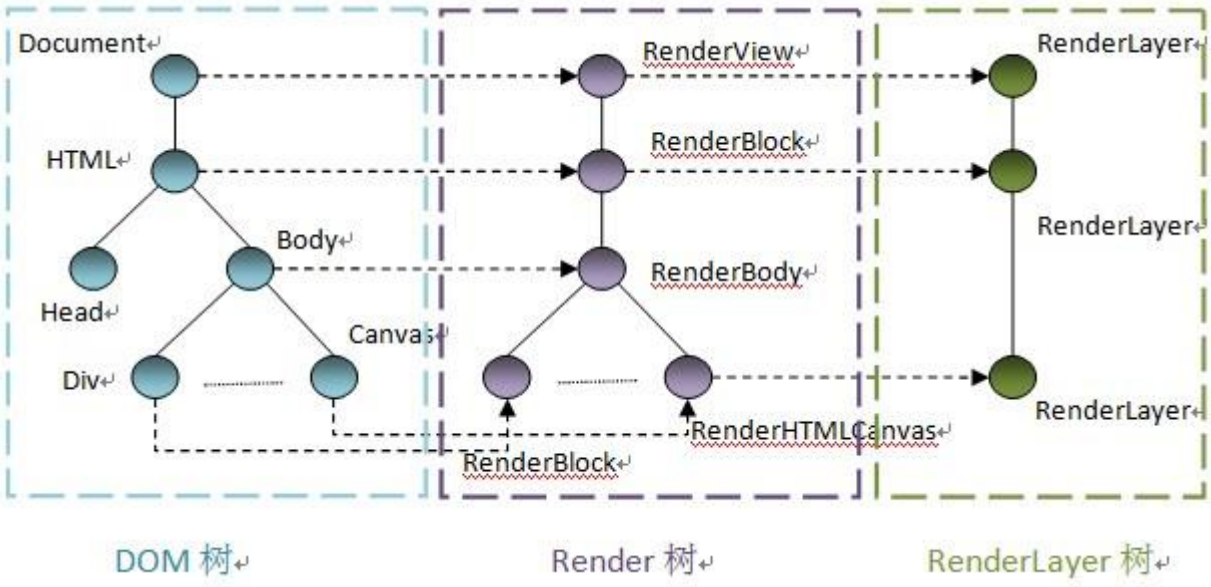
WebKit 是一个渲染引擎，而不是一个浏览器，它专注于网页内容展示，其中渲染是其中核心的部分之一。本章着重于对渲染部分的基础进行一定程度的了解和认识，主要理解基于 DOM 树来介绍 Render 树和 RenderLayer 树的构建由来和方式。

那么什么是 DOM？简单来说，DOM 是对 HTML 或者 XML 等文档的一种结构化表示方法，通过这种方式，用户可以通过提供标准的接口来访问 HTML 页面中的任何元素的相关属性，并可对 DOM 进行相应的添加、删除和更新操作等。相关信息可查阅 W3C 的文档，这里不再赘述。

基于 DOM 树的一些可视（visual）的节点，WebKit 来根据需要来创建相应的 RenderObject 节点，这些节点也构成了一颗树，称之为 Render 树。基于 Render 树，WebKit 也会根据需要来为它们中的某些节点创建新的 RenderLayer 节点，从而形成一棵 RenderLayer 树。

Render 树和 RenderLayer 树是 WebKit 支持渲染所提供的基础但是却非常重要的设施。这是因为 WebKit 的布局计算依赖它们，浏览器的渲染和 GPU 硬件加速也都依赖于它们。幸运地是，得益于它们接口定义的灵活性，不同的浏览器可以很方便地来实现自己的渲染和加速机制。

为了直观了解这三种树，下图给出了这三种树及其它们之间的对应关系，后面会详细介绍里面的细节。



Render 树

Render 树的建立

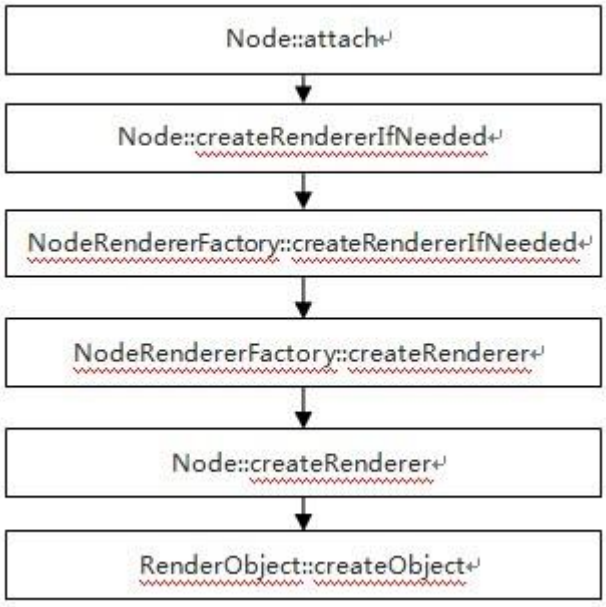
Render 树是基于 DOM 树建立起来的一颗新的树， 是布局和渲染等机制的基础设施。 Render 树节点和 DOM 树节点不是一一对应关系，那么哪些情况下需要建立新的 Render 节点呢？

a) DOM 树的 document 节点；

b) DOM 树中的可视化节点，例如 HTML，BODY，DIV 等，非可视化节点不会建立 Render 树节点，例如 HEAD，META，SCRIPT 等；

c) 某些情况下需要建立匿名的 Render 节点，该节点不对应于 DOM 树中的任何节点；

RenderObject 对象在 DOM 树创建的同时也会被创建，当然，如果 DOM 中有动态加入元素时，也可能会相应地创建 RenderObject 对象。下图示例的是 RenderObject 对象被创建的函数调用过程。

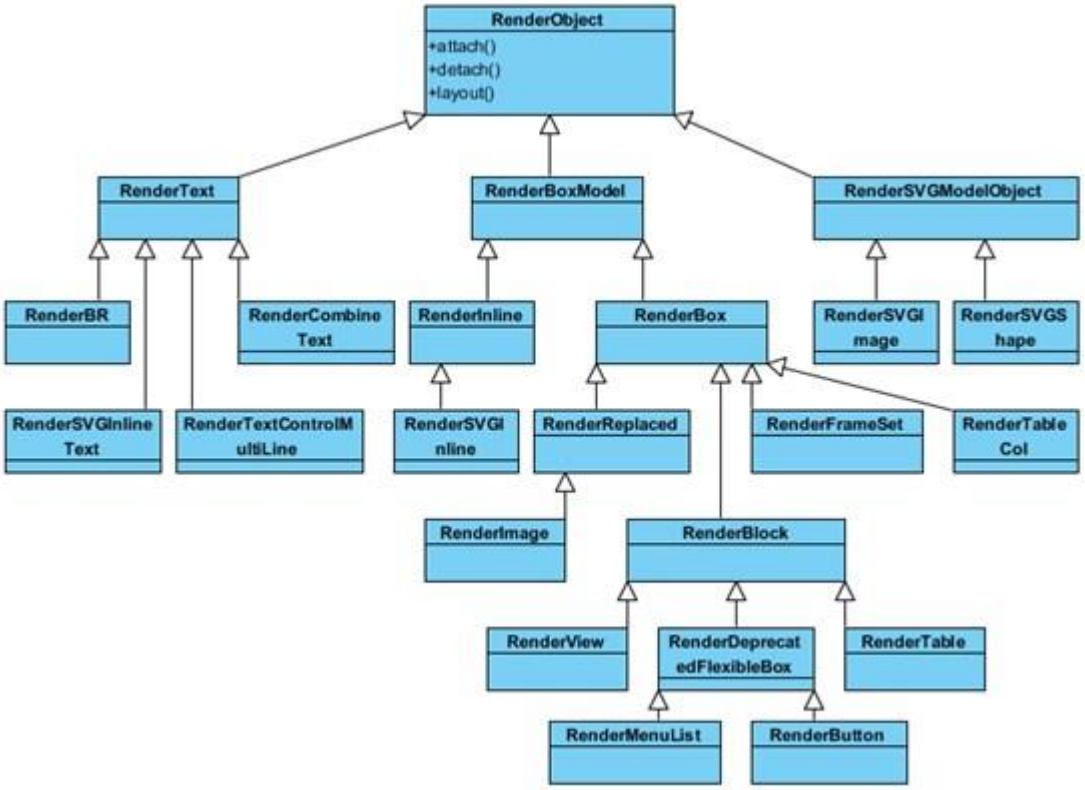


Render 树建立之后，布局运算会计算出相关的属性，这其中有位置，大小，是否浮动等。有了这些信息之后，渲染引擎才只知道在何处以及如何画这些元素。

RenderObject 类及其子类

RenderObject 是 Render 树的节点基础类，提供了一组公共的接口。它有很多的子类，这些子类可能对应一些 DOM 树中的节点，例如 RenderText，有些则是容器类，例如 RenderBlock。下图给出了一些常用的类的继承关系图，这其中

RenderBlock 是一个非常重要的类。



匿名 RenderBlock 对象

CSS 中有块级元素和内嵌(inline)元素之分。内嵌元素表现的是行布局形式，就是说这些元素以行进行显示。以’div’元素为例，如果设置属性’style’为’display:inline’时，则那是内嵌元素，那么它可能与前面的元素在同一行；如果该元素没有设置这个属性时，则是块级元素，那么在新的行里显示。 RenderBlock 用来是用来表示块级元素，为了处理上的方便，某些情况下需要建立匿名的 RenderBlock 对象，因为 RenderBlock 的子女必须都是内嵌的元素或者都是非内嵌的元素。所以，当它包含两种元素的时候，那么它会为相邻的内嵌元素创建一个块级 RenderBlock 节点，然后设置该节点为自己的子女并且设置这些内嵌元素为它的子女。

RenderLayer 树

RenderLayer 树是基于 Render 树建立起来的一颗新的树。同样，RenderLayer 节点和 Render 节点不是一一对应关系，而是一对多的关系。那么哪些情况下的 RenderObject 节点需要建立新的 RenderLayer 节点呢？

a) DOM 树的 document 节点对应的 RenderView 节点

b) DOM 树中的 document 的子女节点，也就是 HTML 节点对应的 RenderBlock 节点

c) 显式的 CSS 位置

d) 有透明效果的对象

e) 节点有溢出（overflow），alpha 或者反射等效果的

f) Canvas 2D 和 3D (WebGL)

g) Video 节点对应的 RenderObject 对象

一个 RenderLayer 被建立之后，其所在的 RenderObject 对象的所有后代均包含在该 RenderLayer，除非这些后代需要建立自己的 RenderLayer。而后代的 RenderLayer 的父亲就是自己最近的祖先所在的不同的 RenderLayer，这样，它们也构成了一颗 RenderLayer 树。

每个 RenderLayer 对应的 Render 节点内容均会绘制在该 RenderLayer 所对应的层次上（或者内部存储结构上）。不同的 RenderLayer 可以共享同一个内部存储结构，也可以有各自的后端存储，这取决于不同的移植。在软件渲染下，通常各个 RenderLayer 的内容都绘制在同一块后端存储上。在 GPU 硬件加速的下，某些 RenderLayer 可能有自己独立的后端存储，而后通过合成器来把这些不同的后端合成在一起，最终形成网页的可视化内容。

RenderLayer 在创建 RenderObject 对象的时候，如果需要，也会同时被创建，当然也有可能在执行 JavaScript 代码时，更新页面的样式从而需要新创建一个 RenderLayer。

一个例子

以上说了那么多，一堆堆的类，一个个的建立规则，听起来太抽象，不太容易理解，那么一个具体的 Render 树和 RenderLayer 树到底是怎么样的呢？为了可视化理解 Render 树和 RenderLayer 树，下面给出一个具体的例子来加以解释和说明。 首先，让我们来看一个简单的 HTML 网页，源代码如下所示。

```
<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>
```

上面的代码结构很简单，就是一些 HTML 的基本元素组成的，例如 HTML，HEAD，DIV，A，IMG， TABLE 等等， 然后包含一个特别的 HTML5 元素 CANVAS，而且还有一小段 JavaScript 代码。照顾到一些没有相关背景的读者，简单解释一下这段代码的含义。这段代码是对 CANVAS 元素创建一个 WebGL 的 Context，有了这个 context，就可以在 canvas 元素上绘制 3D 的内容。这个类似于 OpenGL 或者 OpenGLES 的 context，具体 canvas 和 WebGL 会有另外专门的章节来介绍。

这段 HTML 源代码被 WebKit 解析后会生成一颗 DOM 树。这段代码的 DOM 树主要结构如本章第一幅图中的‘DOM 树’部分所示。当 DOM 树生成时，WebKit 同时建立了一颗 Render 树，如上所说，代码的 Render 树被打印成如下图所示的文本信息。

```
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
```

首先，上图中的“layer at (x, x)”表示不同的 RenderLayer 节点，下面的所有 RenderObject 均属于该 RenderLayer。以第一个 layer 为例，它对应于 DOM 中的 Document 节点。后面的(0, 0)表示该节点在坐标系中的位置，最后的 1028x683 表示该节点的大小。它包含的 RenderView 节点后面的信息也是同样的意思。

其次，看其中最大的部分－第二个 layer，包含了 HTML 中的绝大部分元素。这里有三点需要解释一下：第一，Head 元素没有相应的 RenderObject，如上面所解释的，因为它不是一个可视的元素。第二，Canvas 元素不在其中，而是在第三个 layer 中（RenderHTMLCanvas）。但是它仍然是 RenderBody 节点的子女。第三，匿名的 RenderBlock 节点，它包含了 RenderText， RenderInline 等节点，原因如前所说。

再次，来看一下第三个 layer。因为从 Canvas 创建了一个 WebGL3D Context 对象，这里需要重新生成一个新的 layer。

最后，来说明一下三个 layer 的创建时间。第一和第二个 layer 在创建 DOM 树后，会触发创建；第三个 layer 测试资源加载解析好之后，执行后面的 JavaScript 代码后所创建。

基于上面的描述，相信大家已经对 Render 树和 RenderLayer 树有了一定的了解。现在让我们回忆一下本章的第一幅图。该图其实是给出了示例的 HTML 网页在 WebKit 中三种树的对应关系（仅选取其中重要的部分），相信现在你已经了解它们的含义了。

源代码目录

```
WebKit/Source/WebCore/rendering
        WebKit 为支持渲染所涉及的各个类
WebKit/Source/WebCore/dom/
        DOM 树的相关文件，包括一些基础类及其接口定义
WebKit/Source/WebCore/html/
```

为 HTML 网页的元素所定义的相关的基础类，它们基于 DOM 的 Node 类

参考文献

Google design documents: http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

渲染主循环（main loop）和 requestAnimationFrame

概述

曾经写过一段 JavaScript 代码，因为涉及到需要循环调用某个函数来实现动画的功能，很自然地，我想到了使用 setInterval 函数（或者 setTimeout，大家是否有类似经历呢？），然后心满意足地很快的搞定。结束后，朋友帮忙阅读了一下代码，他提醒我是不是可以考虑使用 requestAnimationFrame。之前一直知道这个函数，也知道一些它的一些优点，问题是为什么呢？本着追究到底的精神，决定还是去阅读一下 WebKit 相关代码和一些相关文档，了解它们背后的故事。好吧，本章我将和大家一起来学习和探讨这背后的故事…

背景

接触过 JavaScript 的读者应该有过了解或者使用 setTimeout 或者 setInterval 的经历，其功能是在每个时间间隔之后一次性或者重复多次执行一段 JavaScript 代码（称为回调函数），以完成特定的动画要求。但是，这里面有还有些疑问：

时间间隔应该设置为多少才合适呢？跟屏幕的分辨率有关系吗？
设置的时间间隔会按照预想的执行吗？动画会被平滑地显示出效果吗？
回调函数是复杂的好还是简单的好呢？应该如何编写才能效率高呢？
与平台和浏览器相关吗？如何适应不同平台呢？

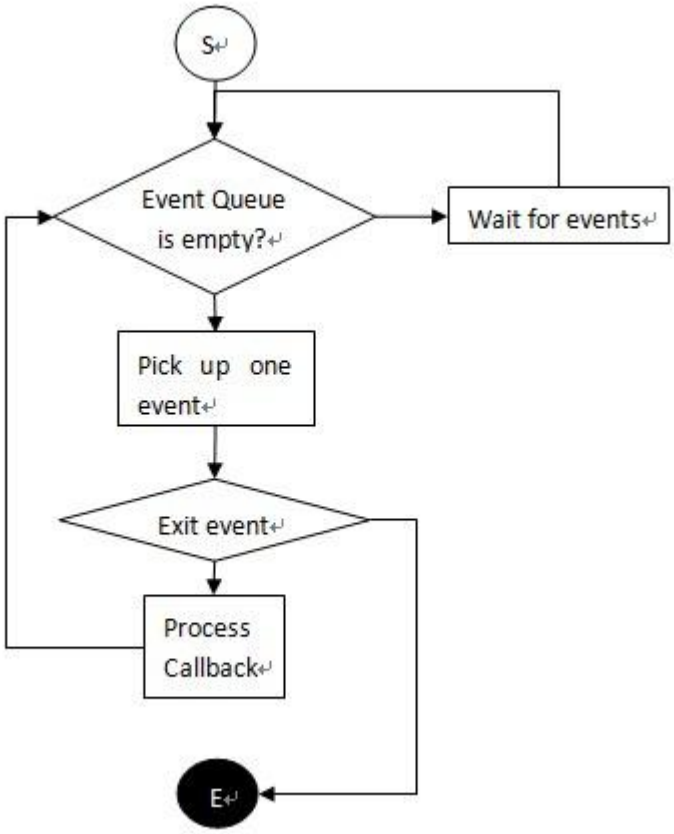
这对 setTimeout 和 setInterval 来说很重要。如果对 mainloop 机制和渲染机制有一定了解的读者来说，上面这几条其实是非常难做到地，哪怕是较为接近理想的结果。

幸运地是，总是有聪明的人来帮助大家解决难题。对问题提出一个漂亮解决方案的是 mozilla 的 Robert O’Callahan。他的灵感和依据来源于 CSS。CSS 是知道动画什么时候发生，所以能够较为准确的知道什么时候刷新 UI。对于 JavaScript 来说，是不是也可以根据类似的机制呢？答案是肯定地。其做法是增加一个新的方法 requestAnimationFrame, 该方法告诉浏览器 JavaScript 想发起一个动画帧，然后在动画帧绘制之前，需要做一些动作，这样浏览器可以根据需要来优化自己的 mainloop 机制和调用时间点，以达到较好地平衡效果。 好吧，下面来看看 mainloop 机制及其工作原理。

渲染 mainloop

因为 chromium 是多进程的结构（参看 Chromium 多进程架构篇），所以，跟一般浏览器不一样的是，Browser 进程 UI 用户界面的 mainloop 和 Renderer 进程的主线程的 mainloop 不是同一个，分别位于两个不同的进程，所以 UI 和渲染可以互相不影响，听起来这好像很不错，是的，但是问题依然存在，那就是 Renderer 进程的渲染工作和 JavaScript 的执行工作都在其主线程中，由 mainloop 来负责调度完成，所以竞争依然存在。

大致过程是一个大的循环加上一个事件队列，具体的过程，如下图所示。当队列中有事件时，从队列中取出第一个事件，设置相应的状态信息，处理该事件及其对应的处理函数，直到该函数处理完后，才重新检查队列中是否有事件。如果有，继续处理；如果没有，则继续等待。这其中可以看出，如果队列中事件多的时候，那么很多事件可能来不及处理，从而造成比较大的延时，因而事件的平均等待时间会比较长。同时，如果事件的处理函数需要的时间很长，就会造成后面的事件一直在等待，同样会增加事件的平均等待时间。而当队列比较空闲时或者事件的处理函数需要的时间比较短，则事件的平均等待时间会相对小很多。



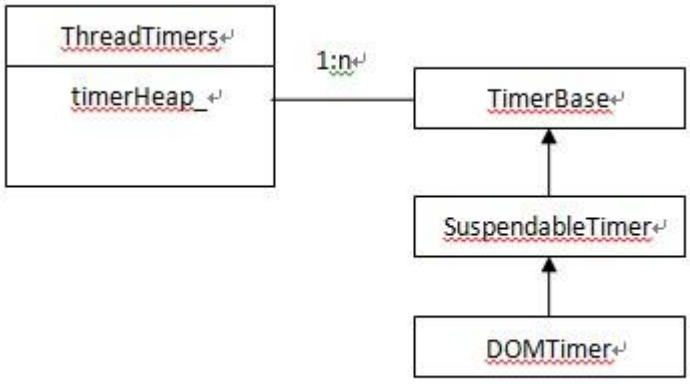
WebKit 和 Chromium 中的实现

理解了 mainloop 之后，下面来看一看 setTimeout 和 setInterval 的实现。

来看一下它们的实现：WebKit 中 setTimeout 和 setInterval 的实现机制是类似的，区别在于后者是重复性的，见下图所示的类图关系。

WebKit 会为 DOM 中的每个 setTimeout 和 setInterval 的调用创建一个 DOMTimer，而后该对象会由存储 TLS（thread localstorage）中的 ThreadTimers 负责管理，其内部其实是一个最小堆，每次取 timeout 时间最小的，同时，时间相同的 Timer 可以合并。

当 Timer 超时后，Chromium 清除该 Timer 对象，同时调用相应的回调函数，回调函数通常会更新页面的样式和布局，这会触发 relayout，从而触发立即重新绘制一个新帧。



结合上面的描述，我们大致地总结 setTimeout 和 setInterval 主要不足就是：

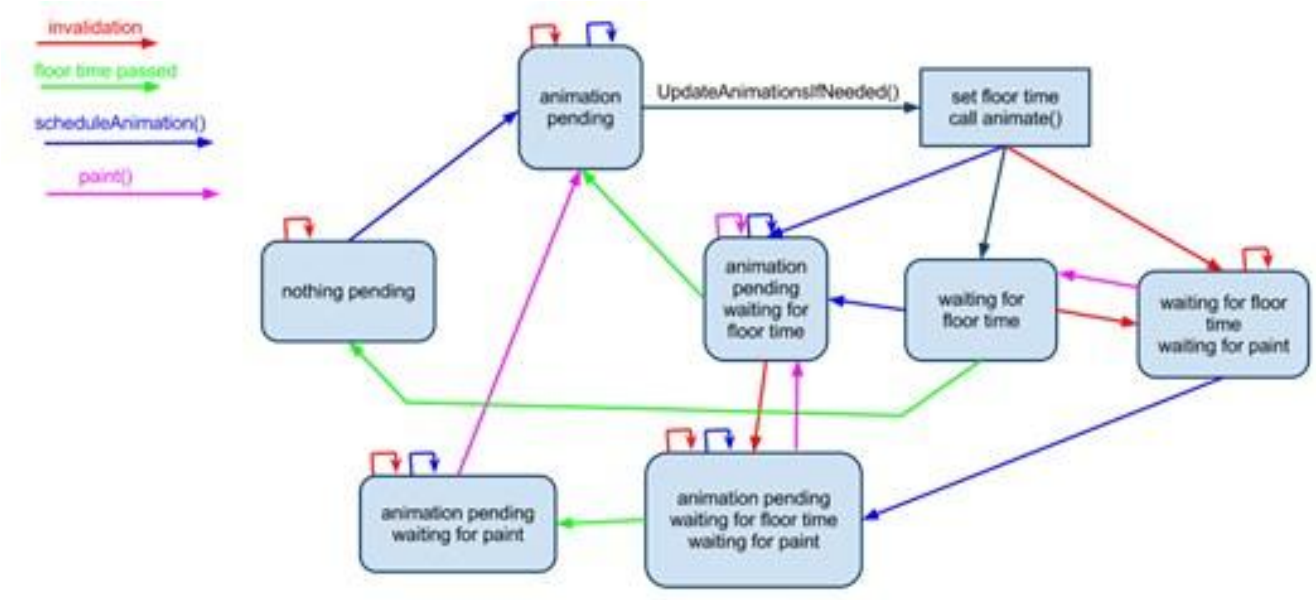
setTimeout 和 setInterval 从不考虑浏览器内部发生了其他什么事，它只要求浏览器在某个时间之后调用它的回调函数，无论浏览器很繁忙或者页面被隐藏（虽然某些浏览器做了这方面的优化，例如 chromium）；
setTimeout 和 setInterval 只要求浏览器做什么，而不管浏览器能不能做到（例如 mainloop 有很多事件需要处理），这有点强人所难，而且会带来极大的资源浪费。举个例子，例如屏幕的刷新率是 60HZ，但是设置的时间间隔是 5ms，其实对用户来说根本看不到这些变化，但是额外需要消耗更多的 CPU 资源，太不环保了…
setTimeout 和 setInterval 可能是编程风格方面的考虑。如果每一帧可能在不同的代码出需要设置回调函数，一个方法是统一到一个地方，但是这有点勉为其难，另一个方法是分别用 setInterval 设置它们，这个方法的问题是，浏览器可能需要计算更多次，刷新更多次的屏幕，唉。

现在再来看看 requestAnimationFrame 的实现，看看其如何解决这些不足之处的。其原理就是其会申请绘制下一帧，至于什么时候不知道，由浏览器决定，只需要浏览器在绘制下一帧前执行其设置的回调函数，完成 JavaScript 对动画所做的设置和逻辑即可。基本过程是这样的：

JavaScript 调用 requestAnimationFrame，因而相应的 webkit 和 chromium 会调度一个需要绘制下一证的事件，该事件会将 requestAnimationFrame 的调用上下文和回调函数记录下来；
上面的请求会触发 Chromium 更新页面内容的事件，该事件被 mainloop 调度处理后，会检查是否需要调用动画的相关处理，因为有动画需要处理，所以会依次调用那些回调函数，JavaScript 引擎会更新相应的 CSS 属性或者 DOM 树修改；Chromium 触发重新计算 layout（参看 layout 章节），更新自己的 Renderer 树（参看 webkit 渲染基础章节），而后绘制，完成一帧的渲染。

下图是一个上述过程对应的状态转换图，来源于 chromium 的官方网站，看着的确比较饶人，可以先理解一下其中几个主要的概念：

Floortime：指的是绘制下一帧之前需要等待的事件间隔
Invalidation：触发重新绘制请求的操作；
scheduleAnimation：JavaScript 调用 requestAnimationFrame 所引起的 WebKit 内部请求调度动画的操作；
这些状态的转换倒是说明了，requestAnimationFrame 可以很好地和 Chromium 内部的绘制过程结合，从而达到比较好的性能。



为了实现更好的性能，chromium 中对 requestAnimationFrame 有三个设计原则 1. 当页面不可见时，其回调函数不会被调用，这可以减少 CPU 和 GPU 的使用率，更环保嘛； 2. 其最大调用频率不会超过 60hz，无论屏幕的刷新率是多少，因而回调函数也不会每秒调用超过 60 次，这是因为 60FPS 已经能够满足 UI 流畅的要求了，更频繁的刷新效果不明显；

3. 只有当页面真正开始渲染时，回调函数才会被调用。 为了对比二者的性能上的差异，我测试了 GuiMark 中 HTML5Charting Test benchmark，修改里面一些代码（其缺省使用的是 setInterval，改为 requestAnimationFrame 作对比），从实际测试的效果上看，在 Google Chrome 中，两者相差不是特别大，使用了 requestAnimationFrame 的 benchmark 的 FPS 大概只好了 1～2FPS，所以 chrome 对 timer 机制的优化做地应该相当不错。如果你遇到了其他差别比较大的例子，欢迎跟我和大家分享。 Google Chrome 对其处理的比较好不代表其他浏览器也是，所以各位还是在编程时候多考虑考虑，多思考思考，为了更好的性能，为了环保…

设计机制带来的编程考虑

最后，结合 mainloop 和 requestAnimationFrame 的设计原理和机制，看一看它们带给我们在编写 JavaScript 代码时有哪些方面的思考和便利： 1. 回调函数不能太大，不能占用太长时间，否则会影响页面的响应和绘制的频率； 2. requestAnimationFrame 不需要设置间隔时间，不同刷新率的间隔时间不一样，这完全由浏览器来控制，而不需要 JavaScript 程序员操心； 3. 回调函数无需合并，程序员可以在任意位置设置回调函数，它们可以被浏览器集中处理，而无需要一个统一的入口。

源文件目录

```
third_party/WebKit/Source/WebCore/page/
  支持 requestAnimationFrame，setTimeout 和 setInterval 的绝大多数基础设施都在这里，建议在该目录下搜索这些关键字即可
third_party/WebKit/Source/WebCore/platform
  Timer 方面的一些支持
```

参考文献

http://dev.chromium.org/developers/design-documents/requestAnimationFrame-implementation
http://www.cnblogs.com/rubylouvre/archive/2011/08/22/2148793.html
http://www.nczonline.net/blog/2011/05/03/better-javascript-animations-with-requestAnimationFrame/
https://developer.mozilla.org/en-US/docs/DOM/window.requestAnimationFrame
http://www.w3.org/TR/animation-timing/#requestAnimationFrame
http://creativejs.com/resources/requestAnimationFrame/
http://www.craftymind.com/factory/guimark2/HTML5ChartingTest.html

高级篇

第二部分是高级话题。

Chromium 软件渲染

概述

本章将介绍 chromium 渲染的最基础部分，同时也是最常见的部分－软件渲染。故名思路，软件渲染就是利用 CPU，根据一定的算法来计算生成网页的内容，其没有那么多复杂难懂的概念和架构。如果了解了那么多地关于硬件加速的知识后，你可能会觉得本章的内容非常的简单明了，是的，你没猜错。 在绝大多数的情况下，也就是没有所谓地那些需要硬件加速内容的时候（包括但不限于 CSS3 3D transformation， CSS3 3D transition， Canvas 2D， WebGL 和 Video），Chromium 都是用软件渲染的技术来完成页面的绘制工作（除非你强行打开硬件加速绘制），基本上你浏览地大多数门户网站，论坛网站，社交网站等等的网页，都是采用这项技术来完成页面的生成。可以这么说，传统的网页都是软件渲染，用到了硬件加速的新网页通常都是加入了很多眩目视觉效果的网页（当然，这不是绝对的）。 本章首先详细了解软件渲染的基础设施和架构，然后介绍渲染的过程，最后并与硬件加速渲染作一比较，介绍一下各自的优点和局限性。

软件渲染基础和架构

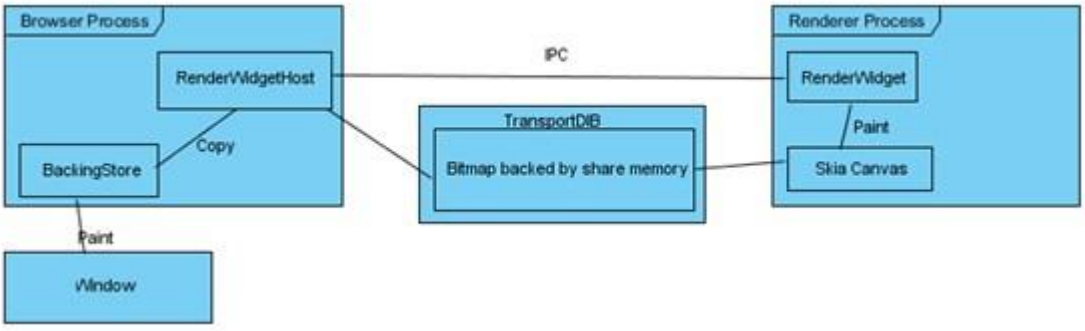
前面介绍了 WebKit 中的渲染基础，这些设施在这里也同样需要，不过，我不打算在这里重新详细介绍它们。下面详细介绍的是 Chromium 中支持软件渲染的基础设施及其架构。

先来看看 Renderer 进程。同硬件加速渲染一样，RenderWidget 对象是必不可少的，它不仅负责调度页面渲染和页面更新等操作，而且负责同 Browser 进程的通信。另一个重要的设施是 PlatformCanvas，也就是 SkiaCanvas，Render 树的绘制操作最后都是调用 Canvas 的元素操作来实现。

再来看看 Browser 进程。第一个设施就是 RenderWidgetHost 对象，一样的必不可少，负责同 Renderer 进程的通信。第二个是 BackingStore，顾名思义，它就是一个后端的存储空间，它的大小通常就是 viewport 也就是网页可视区域的大小，该空间存储的就是页面的绘制结果。

最后来看看这两个进程是如何传递信息和绘制内容地。传递消息还是之前介绍过的消息机制，而绘制的结果则是通过 TransportDIB 类来完成，该类在 Linux 系统下其实是一个共享内存的实现。对 Renderer 进程来说，Skia Canvas 把内容绘制到 bitmap 中，该 bitmap 的后端即是共享内存。对 Browser 进程来说，当绘制完成后，它会把共享内存的内容复制到自己的 backingstore 中。

下图显示的是软件渲染的架构图，其主要来源于 chromium 的官方网站，但这里做了一些扩充。



其组成部分上面已经介绍过，那么一个大致的过程是这样的：RenderWidget 接收到更新请求时，创建一个共享内存区域，然后创建 skia

canvas，之后 RenderWidget 会把实际绘制的工作派发到 Render 树中来完成，具体上来讲，也就是 WebKit 负责遍历 Render 树，每个 RenderObject 节点根据需要来绘制自己和子女的内容到目标存储，也就是 SkiaCanvas 所对应的共享内存的 Bitmap 中。之后，RenderWidgetHost 把 bitmap 复制到 backingstore 的相应区域中，并调用 paint 来把自己绘制到窗口中。下面详细介绍这个过程。

渲染具体过程

依照惯例，我们先来看一个简单的 HTML 例子，里面包含简单的元素和一小段 JavaScript 代码，它会每隔 50ms 请求更新 id 为”content”的元素的内容。

```
<html>
    <head>
        <title>Simple HTML</title>
    </head>
    <body>
        <div id="content">This is a simple html page</div>
        <script type="text/javascript">
            var counter = 0 ;
            function loop() {
                var elem = document.getElementById("content");
                elem.innerHTML = "This is a simple html page: " + counter;
                counter++;
            }
            setInterval(loop, 50);
        </script>
    </body>
</html>
```

后面我们会讲它在哪些时候请求绘制网页内容，这里先了解一下很多情况下会发起重新绘制某些区域的请求，大概可以分为两类： 1） 前端请求：该类包括从 browser 进程发起的请求，可能是 browser 自身的，也有可能是 X（或者其他窗口系统）。 2） 后端请求：由页面自身发起更新部分区域的请求，例如 HTML 元素或者样式的改变，动画等等。上面的例子中， JS 代码每隔 50ms 更新元素，所以它会触发更新部分区域。 下面来解释一下当有绘制或者更新某个区域请求时，Chromium 和 WebKit 如何来处理。过程如下图所示，下面阐述一下大致的步骤。

1） Renderer 进程的 message loop 调用处理 Invalidation 的回调函数，该函数主要调用 RenderWidget::DoDeferredUpdate 来完成绘制请求。

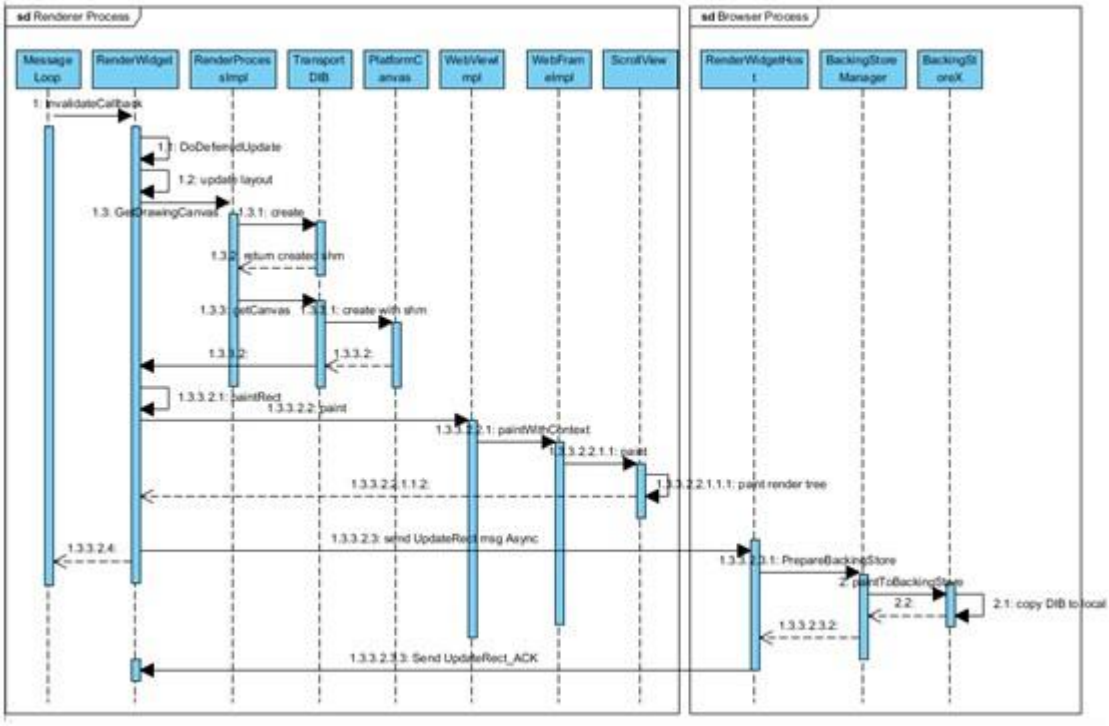
2） RenderWidget::DoDeferredUpdate 首先调用 layout 来触发检查是否有需要重新计算的布局和更新请求。

3） RenderWidget 调用 TransportDIB 来创建共享内存，内存大小为绘制区域的高×宽×4，同时调用 Skia 来创建一个 canvas，它的绘制目标是一个使用共享内存存储的 bitmap。

4） 当渲染该页面的全部或者部分时，ScrollView 请求按照从前到后顺序遍历并绘制所有的 RenderLayer 的内容到目标的 bitmap 中，每个 RenderLayer 的绘制通过以下步骤来完成：首先计算重绘的区域是否和自己有重叠，如果有，则要求该 layer 中的所有 RenderObject 对象绘制自己。这在上图中省略了该部分的具体内容，详情参考代码。

5） 绘制完成后，发送 UpdateRect 的消息给 browser 进程，Renderer 进程同时返回完成绘制。Browser 进程接受到消息后首先由 BackingStoreManager 来获取或者创建 BackingStoreX（在 linux 平台上），BackingStoreX 大小是 Viewport(可视区域)，包含相对于整个网页的坐标信息，它根据 UpdateRect 的更新区域的位置信息将共享内存的内容绘制到自己的对应存储区域中。

6） 最后 Browser 进程发送 UpdateRect 的 ACK 消息给 renderer 进程，完成整个过程。



那么上述的 HTML 例子中大概有哪些绘制请求呢？

1） 当初始打开该 HTML 页面时，首先会由 browser 进程发起要求更新整个可视区域的重绘请求，后面的绘制过程如上所述。

2） 后面，每隔 50ms，chromium 执行 JS 代码，JS 代码修改了 HTML 元素，该动作触发重新计算布局，因而发起更新某块小的区域的请求。

同硬件加速渲染的对比

我们可以看到，软件渲染有一个重要的影响性能的地方来自于共享内存后的复制到 backingstore 的操作，那么为什么需要将共享内存的内容复制到 backingstore 中，而不是直接使用呢？我认为这里面有两点重要的原因：

第一， 共享内存是很宝贵的资源。通常，操作系统对共享内存的大小和总量有一定的限制，因而可以使用的量比较小（相对于虚拟地址空间）。chromium 浏览器可能包含多个 Renderer 进程，这些进程如果都申请这么多的共享内存，这将是一个极大的浪费，鉴于此，在使用完后，因尽快地释放共享内存。

第二， 重新绘制网页内容的需要。当 X 有请求更新浏览器窗口的某个区域时，在此情况下，网页内容可能没有任何的改变，因而如果在浏览器端有一个 backingstore 保存网页的内容，那么不需要 WebKit（Renderer 进程）重新渲染，这将会是一个很费时的工作，这在某种程度上可以提高性能。

软件渲染同硬件加速渲染另外一个很不同的地方就是对更新区域的处理。回忆一下之前介绍的硬件渲染部分，当网页中有一个更新某个区域的请求时（例如动画），硬件渲染可能只需要对其中的一层重新绘制，然后重新合成即可。但是，在软件渲染过程中，因为它没有为每一层提供后端存储，因而它需要将和这个区域有重叠部分的所有层次重新绘制一遍，因而某些情况下开销比较大。

上面说了之后，可能会觉得软件渲染不那么有用，其实不然，大多数情况下，软件渲染是有其优势所在的，

第一， 很多场景适合用软件渲染，而且速度更快。一般的网页，用软件渲染速度更快，用了硬件加速这个“牛刀”可能会造成较大地额外负载，一般网页的文字和图片对现代 CPU 来说不是什么大问题，处理起来那是相当地快。

第二， 相比较硬件加速渲染，它不需要 GPU 内存和其他资源。硬件加速渲染需要很多的 GPU 内存资源，因为它会每个层分配内存对象，很多时候 GPU 资源很紧张。

源文件目录

```
content/renderer/
        Renderer 进程端与 Browser 进程通信的设施
content/browser/
        Browser 进程端与 Renderer 进程通信的设施
ui/gfx/surface
        包含 TransportDIB 类，用来申请和管理共享内存
```

参考文献

GPU 加速合成：http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome
渲染过程图： http://www.chromium.org/developers/design-documents/rendering-architecture-diagrams

Chromium 的 GPU 硬件加速

概述

这里所说的 GPU 硬件加速是指应用 GPU 的图形性能对 chromium 中的一些图形操作交给 GPU 来完成，因为 GPU 是专门为处理图形而设计，所以它在速度和能耗上更有效率。但是，使用 GPU 加速有些额外开销，并且某些图形操作 CPU 完成的会更快，因而不是所有的操作都合适交给 GPU 来做。

Chromium 中，GPU 加速可以不仅应用于 3D，而且也可以应用于 2D。这里，GPU 加速通常包括以下几个部分：Canvas2D，布局合成（Layout Compositing），CSS3 转换（transitions），CSS3 3D 变换（transforms），WebGL 和视频(video)。目前 chromium 对这些的支持已经越来越充分，只是某些方面可能还存在些正确性和性能问题。

Chromium 除了对以上采用 GPU 硬件加速之外，在新的 views 框架中，引入了对浏览器主界面采用硬件加速的机制。它可以把浏览器主界面上的各种工具栏等和网页通过 GPU 合成（compositing）起来。因为之前网页布局也用到合成器，目前两个合成器各自为战，所以 chromium 会实现一个新的合成器叫 chrome compositor，关于它的细节，我会在专门的章节来做介绍。

本章主要介绍 GPU 进程相关的内容，但是不涉及各个部分的加速实现。具体的各个加速实现及 WebKit 对 GPU 硬件加速的支持，后面会有分别的章节来介绍。

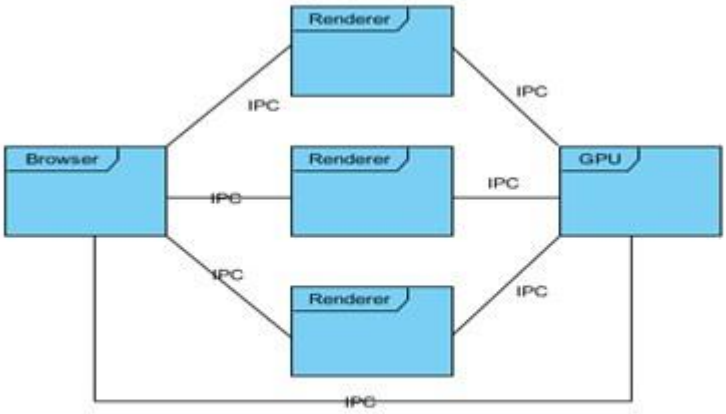
启用 GPU 加速

如果想使用 Chromium 的 GPU 硬件加速功能，首先需要你的 GPU 驱动程序不在 chromium 的黑名单中。因为很多的 GPU 驱动程序存在很多的错误，这些错误可能会比较严重的影响了 chromium 的稳定，例如导致程序崩溃，所以这些 GPU 被列在黑名单中。一个检测的简单办法是在地址栏里输入“about:gpu”查看相关 GPU 的信息，检查相关加速是否已经打开。

如果你的 GPU 驱动不幸在黑名单中，首先的办法是升级你的驱动，然后重新启动 chromium。如果还是解决不了问题，可以通过在 chromium 的启动参数中加入“--ingore-gpu-blacklist”来关闭黑名单功能，这样你就开启了你的 GPU 加速功能。但是，如前所说，这种方法可能会造成 chromium 的不稳定，因而只是权宜之计。

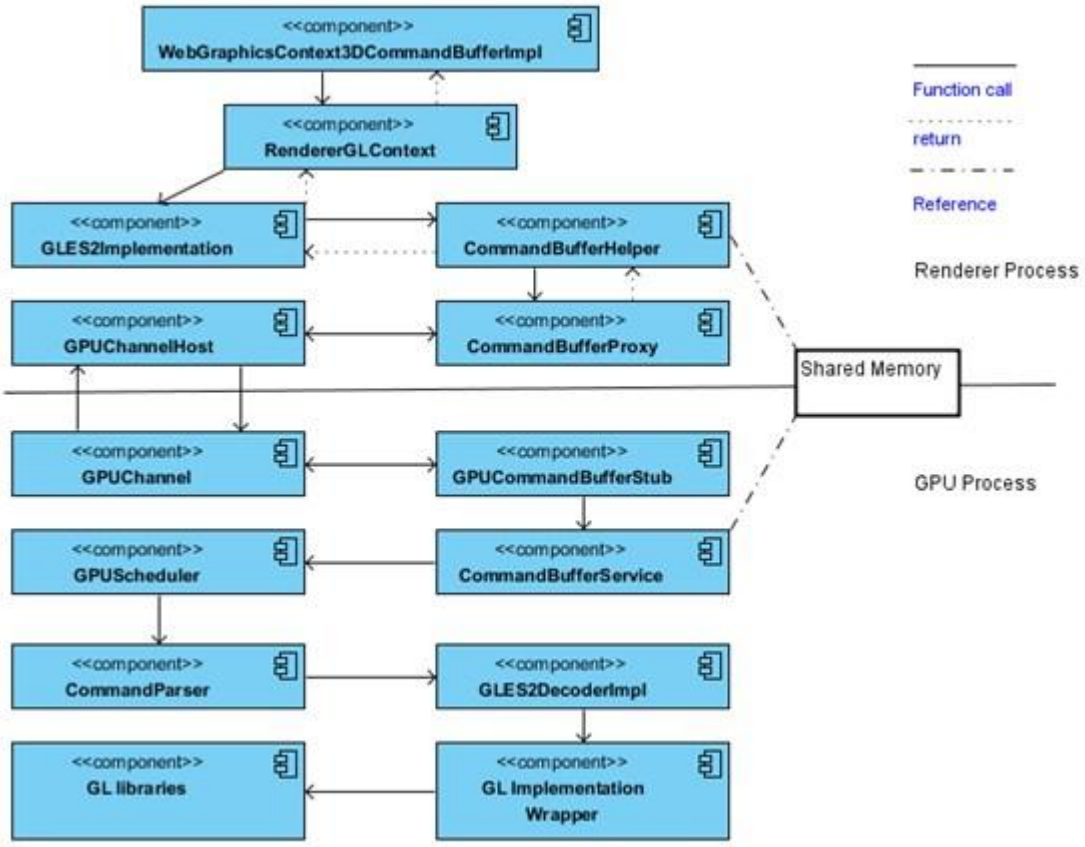
进程模型

由于 chromium 的安全模型和稳定性考虑，GPU 加速也采用多进程模型来实现。所以的 GPU 加速相关的操作均有一个独立的进程来完成，这就是 GPU 进程。详见下图所示 GPU 进程和 Browser，Renderer 进程。



GPU 进程由 Browser 进程来创建和销毁，这类似于对 plugin 进程的管理。它们之间的通信是通过 chromium 的 IPC 消息机制实现的。Chromium 只会创建一个 GPU 进程，该进程被所有的 Renderer 进程和 Pepper plugin 进程所共享，但是 GPU 进程会为不同的进程创建不同的 command buffer 的示例。对于一个 Renderer 进程，GPU 进程也可以为其创建多个示例，这取决于 Renderer 进程的需求。

下面主要介绍以下 GPU 进程和客户进程（以 Renderer 进程示例）之间是如何工作的。下图描述了以 WebGL 为示例的主要模块（类）及它们之间的联系。



首先来看一看 Renderer 进程的模块类：

WebGraphicsContext3DCommandBufferImpl:继承自 WebKit::WebGraphicsContext3D 类， 具体的实现类，主要是转接自 WebKit 的调用到 chromium 的具体实现。主要包括一个 RenderGLContext 对象。

RendererGLContext： Renderer 进程对 GLContext 的一个封装，包括所有用于跟 GPU 进程交互的类，有一个 GLES2Implementation 对象，一个 CommandBufferProxy 对象和一个 GPUChannelHost 对象。

GLES2Implementation:该类模拟 GLES2，但是不直接调用 GLES2 的实现，而是将这些调用转换成特定格式的命令存入 CommandBuffer 中。

CommandBufferHelper： 该类是一个辅助类，包括一个 CommandBuffer 代理类和一个共享内存

CommandBufferProxy:CommandBuffer 的一个代理类，实现 CommandBuffer 的接口，用于和 CommandBufferStub 之间的通信

GPUChannelHost:用于 IPC 消息的通信类

接着再挨个了解一下 GPU 进程中的模块类：

GPUChannel:用于 GPU 进程中的 IPC 消息的通信类

GPUCommandBufferStub:CommandBuffer 的桩，接受来自于 CommandBufferProxy 的消息，将请求交给 CommandBufferService 处理

CommandBufferService:具体的 CommandBuffer 的实现类，但是其实它并不具体解析和执行这些命令，而是当有新的命令时，触发给注册的回调函数来处理

GPUScheduler:负责调度执行 commandbuffer 的命令，它会检查该 commandbuffer 是否应该被执行，并适时将命令交给 CommandParser 的来处理

CommandParser:仅检查 CommandBuffer 中的命令的头部，其余交给具体的 decoder 来做

GLES2DecoderImpl:解析每条具体的命令并执行调用 GL 相应的函数

GLImplementation Wrapper: 一组 GL 相关的函数指针，通过用户的设定来读取相应 gl 库的函数地址来设置自己

GLLibraries: 具体的函数库，可以是 opengl，opengles，mesagl，mock 等

通过上面每个模块类的具体介绍，相信你已经大概知道它们的工作过程。那么，它们是如何同步的呢？因为 Renderer 进程需要等待 GPU 进程做好某些操作之后才继续执行。答案是，GPU 进程处理一些命令后，会向 Renderer 进程报告当前自己当前的状态，Renderer 进程通过检查状态信息和自己的期望结果来确定是否满足自己的条件。

GPU 进程最终绘制的结果不再像软件渲染那样通过共享内存传递给 Browser 进程，而是直接将页面的内容绘制在浏览器的标签窗口内。

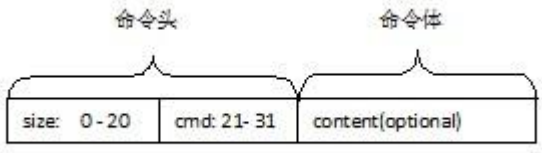
Command Buffer

Command Buffer 主要用于 GPU 进程（且称为 GPU 服务端）和 GPU 的调用者进程（且称 GPU 客户端， 如 Renderer 进程， Pepperplugin 进程）传递 GPU 操作命令。从接口上来，它只提供一些基本的接口来对 buffer 进行管理，它没有对 buffer 的具体方式和命令的格式进行任何限制。

现有实现是基于共享内存的方式来完成，因而命令是基于 gles 编码成特定的格式存储在共享内存中。共享内存方式采用了 ring buffer 的方式来管理，这表示内存可以循环使用，旧的命令会被新的命令所覆盖。基于 gles 的编码格式，因为 WebGL 是基于 gles 的 JavaScript 绑定，所以简单直观。

命令基本格式

一条命令可以分成两个部分：命令头和命令体。命令头包含两个部分，一个是命令的长度，一个是命令的 ID。命令体包含该命令所需要的其他信息，例如命令的立即操作数。命令是可以固定长度的，也可以是变长的，一起取决于该命令。具体的结构如下图所示。



基于 GLES2 的命令

上面说到，本身 Command Buffer 是没有定义具体的命令，所以 GLES2 可以根据需要自己来定义。命令大概可以分成两类，第一类是公共的命令，主要用来操作桶(bucket)，跳转，调用和返回。第二类是跟 GLES2 的函数相关的命令，主要用来操作 GLES2 的函数。

IPC 机制

前面提到，命令本身是保存在共享内存中的，而且每条命令的长度不能超过（1<<21 -1），而且共享内存的大小也是固定的，如果命令太长，可存储的命令很少。那么问题就出来了，如何解决需要传输较大数据的命令呢？对于这样类型的数据，可以对它们利用独立的共享内存来实现，例如 TexImage2D。但是，当共享内存大小超过系统的限制时，这种方式就行不通。chromium 提供了一种新的机制来解决这个问题。

这个机制就是桶（bucket）机制。其解决问题的原理是，通过共享内存机制来分块传输，而后把分块的数据保存在本地的桶内，从而避免了申请大块的共享内存。前面提到的公共的命令就是用来处理桶相关的数据。当数据传输完成之后，对该数据进行操作的命令就可以执行了。

桶机制也可用来传输 string 类型的变长数据。接受端首先获取桶内 string 的长度，然后通过共享内存方式来分块传输，最后合成在自己的桶内。

GPU 加速的后端

GPU 加速虽然是基于 opengles 的接口来设计 commandbuffer 的，但是这不意味着系统的 gl 的后端库必须是 egl/opengles 库，这是因为 chromium 提供了一层转接。

这层转接如何实现的呢？原理其实并不复杂。Chromium 首先定义了一组 gl 相关的函数指针，然后 chromium 根据设置或者命令行参数来加载相应的库，并从这些库中读取相应的函数地址，并把这些地址赋值给这些函数指针。这样一来，GPU 加速部分的代码调用的是这一组函数指针，从而避免直接使用 opengles 的库。详细代码参考源文件目录”ui/gfx/gl”。

源代码目录

```
gpu/
        该目录包含 GPU 相关的文件，主要包括 commandbuffer 的实现，工具，GPU 涉及的 IPC 等
gpu/command_buffer
        该目录存放 CommandBuffer 相关的类，主要包括 client 端，service 端和公共的基础类
gpu/command_buffer/client
        该目录存放 GPU 客户端所使用到的与 CommandBuffer 相关的类
gpu/command_buffer/service
        该目录存放 GPU 进程端所使用到的与 CommandBuffer 相关和 GLES2 具体实现相关的类
content/gpu
        该目录存放 GPU 进程所涉及使用的基础类，包括进程入口函数，IPC，进程和线程管理等相关类
```

content/common/gpu/
该目录存放 GPU 进程使用的与 GPU 相关的类
content/Renderer/gpu/
该目录存放 Renderer 进程使用 GPU 加速所涉及的类，主要包括 IPCchannel, CommandBuffer 代理等类
content/browser/gpu/
该目录存放 Browser 进程和 GPU、Renderer 进程交互相关的类，如 GPUProcessHost。还包括 GPU 黑名单的实现
ui/gfx/gl/
该目录存放与具体的 opengl,opengles 库相关类，包括根据命令行参数读取所需要的 gl 的库类型，库的函数地址的读取和管理等等

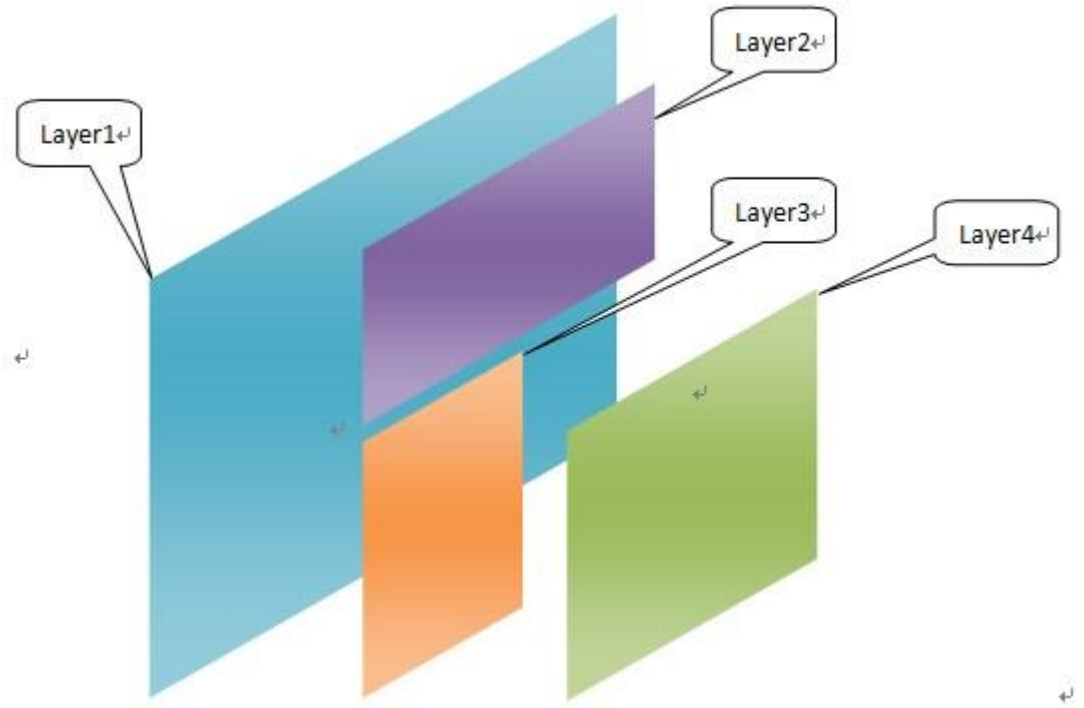
参考文献

http://www.chromium.org/developers/design-documents/gpu-command-buffer
http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome
http://www.chromium.org/developers/design-documents/rendering-architecture-diagrams

Chromium 硬件加速合成

概述

在介绍硬件加速合成(hardware accelerated compositing)之前，让我们先大概了解一下基本的做法是如何渲染生成网页内容的。 一个网页通常可以包含很多层，这个我们在 WebKit 渲染基础中讲过，例如有透明效果的节点，Canvas 节点等，这些节点都可以是页面中的一层，这些层的内容最后组成一个可视化的网页内容，如下图所示。



在没有硬件加速的情况下，浏览器通常是依赖于 CPU 来渲染生成网页的内容，大致的做法是遍历这些层，然后按照顺序把这些层的内容依次绘制在一个内部存储空间上（例如 bitmap），最后把这个内部表示显示出来，这种做法就是软件渲染（software rendering），我们会有专门针对它的介绍。

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，本章要介绍的就是利用 GPU 来实现对页面分层渲染并加速合成网页的可视化结果。

前面我们已经介绍了 WebKit 中对于渲染所做的一些基础设施，包括 Render 树和 RenderLayer 树，对于每一个 RenderLayer，我们可以为其单独创建一块内部存储（有些情况下可以为多个 layer 创建同一块存储），这些存储会被用来保存该层中的内容，浏览器最后会把这些所有的层通过 GPU 合成起来，形成最终网页渲染的内容，这就是硬件加速合成。

本节主要介绍 WebKit 中为硬件加速所做的另外一些基础设施，接着是 chromium 中为硬件加速合成所做的支持，最后介绍 chromium 中最新的合成器：chromium compositor(cc)。

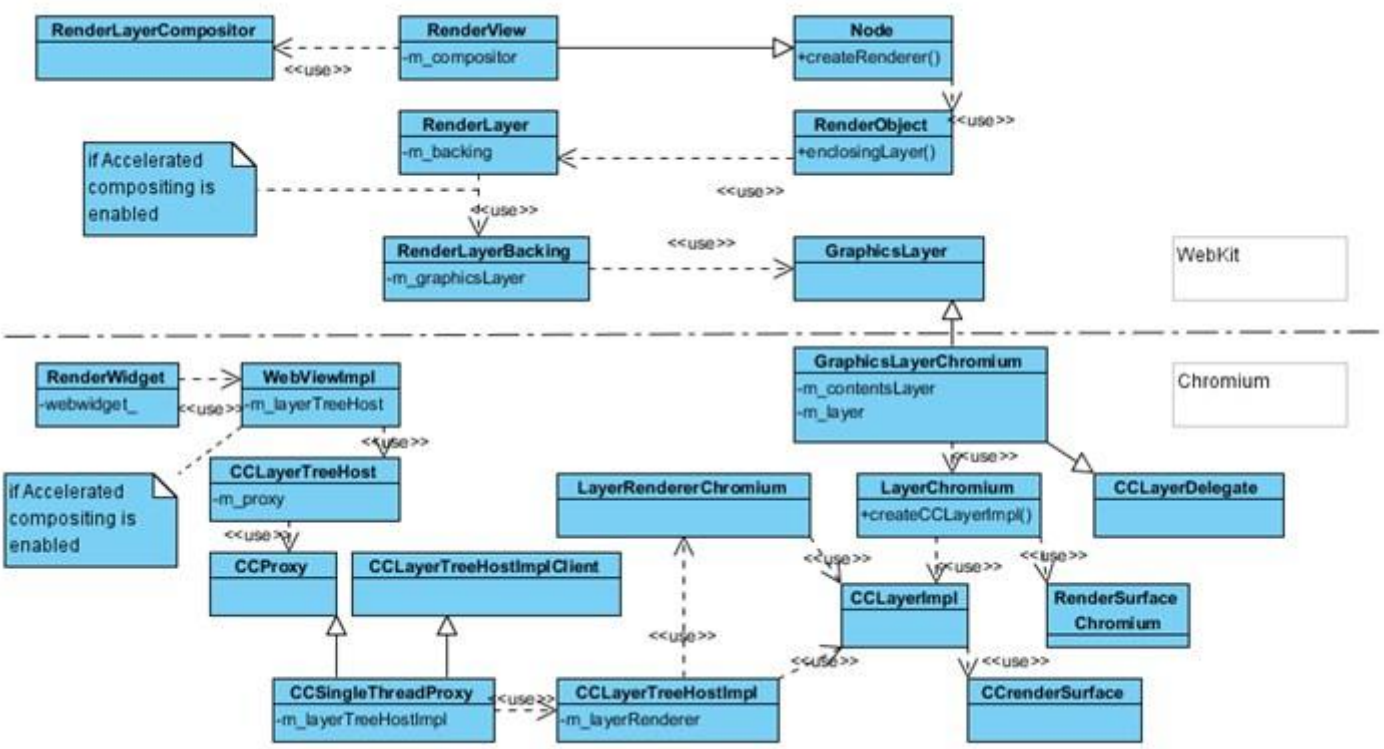
WebKit 的支持

在建立 DOM 树后，如前面所述， WebKit 会创建相应的 Render 树和 RenderLayer 树。如果启用“加速合成(accelerated_compositing)”选项，WebKit 会做一些特别的处理。

它为每个 RenderLayer 创建一个 RenderLayerBacking，这个类非常的有用，它是用来管理和控制 RenderLayer 的合成相关的事情，决定对该层是否做优化，同时包含了很多 GraphicsLayer 的对象。GraphicsLayer 类是一种抽象，用于

表示有后端存储（backing store）的渲染 surface，同时，也包括作用于之上的各种变换（transformation）和动画（animation）。 RenderLayerBacking 类中包含的这些 GraphicsLayer 有内容层，可能还有前景层(Foreground)，剪切层（clipping）等。下图中的 WebKit 部分是 WebKit 中为渲染和硬件加速所涉及的类，在此一一介绍。

RenderLayerCompositor: WebKit 中渲染部分‘掌控大局’的类， 管理 RenderLayer 树结构，它通过浏览器的设置来决定是否创建 RenderLayer 以及是否硬件加速合成，同时也决定是否为 RenderLayer 创建 RenderLayerBacking
RenderLayerBacking:用来管理和控制相对应的 RenderLayer 的合成行为，包含很多 GraphicsLayer 对象，这些对象用于表示层内容，前景（foreground）内容等等
GraphicsLayer: 用于表示有后端存储（backingstore）的渲染 surface 及其相应的变换等，该类是一个抽象类，不同平台的移植有不同的实现



Chromium 的支持

WebKit 提供了基础设施， 但要实现硬件加速合成，各个移植还有很多事要做，chromium 也是如此，见上图中的 Chromium 部分。 大致地，可以把 Chromium 中的关于硬件加速合成部分分成两块，第一块是对 WebKit 接口的具体实现，第二块是 Chromium 合成的管理和实现部分。 第一块包括上图中 Chromium 部分的两个类 GraphicsLayerChromium 和 LayerChromium： GraphicsLayerChromium：继承自 GraphicsLayer，实现了 GraphicsLayer 的虚函数接口，提供了 chromium 自己的移植。 LayerChromium:Chromium 中的 PlatformLayer 类，具体的就是 chromium 的实际的被合成层的基类，被 GraphicsLayerChromium 所创建。LayerChromium 有多个子类，例如对 WebGL 和 video 支持的 WebGLLayerChromium，VideoLayerChromium 类 第二块则是 Chromium 部分的 CC 开头的类和 LayerRenderChromium：
LayerRendererChromium:一个很重要的类，负责实现合成功能，利用 GL 来绘制被合成的各个 CCLayerImpl 层
CCLayerImpl:该类主要是负责一个层的绘制（drawing）工作，由 LayerChromium 创建，同样构成一个树型结构，被 CCLayerTreeHostImpl 来管理。每个对象可以有自己的渲染 surface 对象，也可以使用其祖先的，这取决于具体的策略。 CCRenderSurface:渲染的 surface 类，被 CCLayerImpl 所创建和使用，有自己的存储后端 CCLayerTreeHost:一个 CCLayerImpl 树的客户端类，用于响应 Chromium 的合成请求后，处理 LayerChromium 准备工作，并把请求最后传递给 CCProxy，由其转发给具体的实现者，该类被主线程所调用 CCProxy：一个代理抽象类，用于将主线程的合成操作桥接到具体的合成器实现，合成器的线程可能并非主线程。 CCSingleThreadProxy:CCProxy 的子类，该类表明合成任务没有在单独的线程中执行，该类将客户端的请求发送到合成的具体类 CCLayerTreeHostImpl，并响应合成器调用客户端的回调函数 CCThreadProxy：CCProxy 的子类，将合成任务放在单独的新线程中执行， 目前没有被使用

CCLayerTreeHostImpl:该类主要管理 CCLayerImpl 树及相应的状态，接受来自 CCProxy 的请求，将实际的合成任务交给 LayerRendererChromium 来处理 在早期的版本中，Chromium 中对合成的支持主要就是 LayerRenderChromium，该类负责处理所有关于合成方面的工作。后来， Chromium 新引入了 CC（chromium compositor）的设计。

Chromium Compositor

为了把合成这一功能独立出来，chromium 设计了一套新的合成架构 chromium compositor（后面简称 CC）。

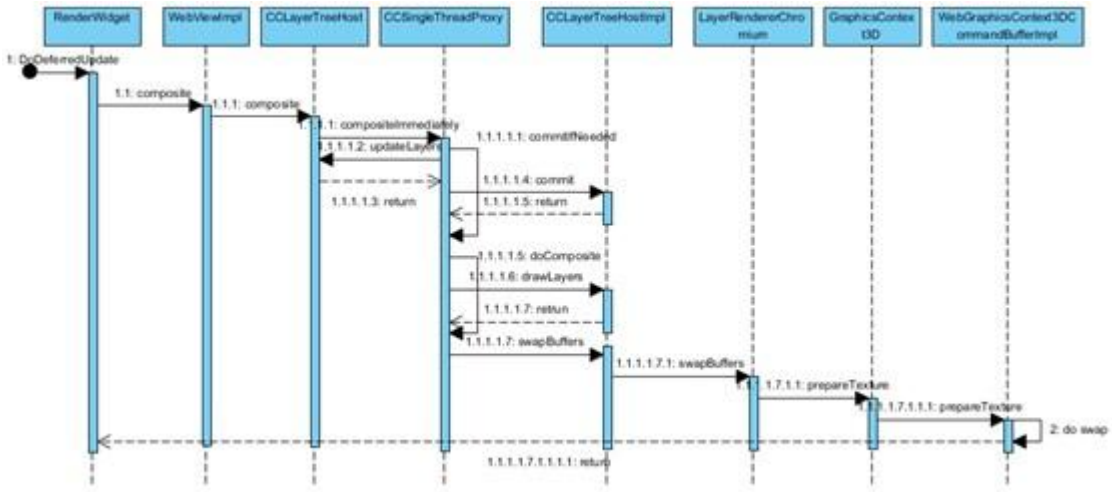
Chromium 定义了自己的层（CCLayerImpl）和层的后端存储方式（CCRenderSurface），所以它的合成任务是基于它们来完成。从架构上来讲，新的 CC 设计了一套更加灵活的机制来实现合成：合成的任务可以放在独立的线程中来做，而不仅仅是当前线程，该机制的好处是合成不会阻碍当前线程的响应，比较灵活。

结合上面的类图，我们来理解一下其内部的机制。简单来讲，可以把合成分成调用端和实现端。调用段由 CCLayerTreeHost 来提供相应的接口，被 render 的主线程调用，当需要合成时，就会调用 CCLayerTreeHost::composite 来完成， 这里它可以被看成调用端。但是实际的实现者却不是该类，而是 CCLayerTreeHostImpl（和 LayerRenderChromium），它们是实现端。它们这两端如何连接起来呢？答案是 CCProxy，它提供了一组抽象的接口，而它的子类（CCSingleThreadProxy 和 CCThreadProxy）来决定具体的实现，它们可以让合成在调用端的线程来完成，也可以创建一个独立的线程来完成。目前，在 Renderer 进程中的合成是在主线程中来完成的。

假以时日，我们有理由相信， chromium compositor 应该会被抽取出来，形成一个独立的库（2012-11-12 更新：该部分已经成为一个独立的库），这样它会被同时用在网页的合成和 chromium 主界面的合成，借助于灵活的架构，合成的工作也不会影响正常的主界面对用户的响应。

GPU 硬件加速合成过程

在了解这些基础模块类之后，下面让我们来看看合成是如何来完成的。下图给出的是一个完整的 Renderer 进程的合成过程。



来解释一下这个过程。Browser 进程或者页面发起一个失效（Invalidate）的请求后，Renderer 进程会调度一个任务来更新相关区域。当该任务被调度执行时，如果硬件加速合成被打开后，Renderer 进程便开始执行对网页的合成。

合成任务由 CC 来管理和执行的。首先，调用 CCSingleThreadProxy::commitIfNeeded，它主要做两件事，第一件，CCLayerTreeHost::updateLayers 为合成做更新 LayerChromium 的准备工作，包括创建 RenderSurface，设置绘制的区域，计算和设置需要更新的层。第二件，如果 CCLayerImpl 树来没有建立的话，那么就根据 LayerChromium 树来建立它。

再次，就是来绘制并合成相关的层，CCLayerTreeHostImpl::drawLayers 基于 CCLayerImpl 树做绘制工作，把需要的绘制的所有层以此重新绘制并合成，该任务由 LayerRenderChromium 类负责处理完成（调用 GPU Process 来完成 GL 的绘制）。

最后，调用 swapbuffers 把命令发给 GPU 进程用来交换前后端 buffer，完成了页面的显示。

源文件目录

```
third_party/WebKit/Source/WebCore/platform/graphics/chromium/
        与硬件加速相关的 chromium 移植的相关类
third_party/WebKit/Source/WebCore/platform/graphics/chromium/cc
        Chromium 新的合成器 CC，包含所有 CC 的基础和管理类
cc/
      最新的 CC 部分的相关代码，已经成为一个新的独立的库
```

参考文献

http://www.chromium.org/developers/design-documents/aura/graphics-architecture
http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome
http://www.chromium.org/developers/design-documents/rendering-architecture-diagrams

Canvas2D 及其实现

概述

Canvas 是 HTML5 新引入的元素，它是一个画布。开发者可以用 JavaScript 脚本在该元素上绘制任意图形（2D 或者 3D）。Canvas 元素有两个属性“width”和“height”，用来设置画布的宽度和高度。Canvas 本身来讲并没有定义绘制图形的动作和行为，只是提供了一个获取绘图的上下文（context）对象的方法－getContext 来获取绘制 2D 或者 3D 上下文。

Canvas 的‘getContext’方法包含一个参数，该参数用来指定创建上下文对象的类型。对于 2d 的图形操作，通过传递参数值’2d’，浏览器会返回一个 2d 的绘图上下文，称为 CanvasRenderingContext2D， 它提供了用于绘制 2d 图形的各种 API， 包括基本图形绘制（例如线，矩形，圆弧），文字绘制，图形变换，图片绘制及合成等。我们把以上这些称之为 Canvas2D。下图是一个使用 Canvas2d 的简单例子，后面的讨论都会基于它来进行。

```
<!DOCTYPE HTML>
<html>
    <body>
        <canvas id="myCanvas" width="80" height="100">
            your browser does not support the canvas tag
        </canvas>
        <script type="text/javascript">
            var canvas=document.getElementById('myCanvas');
            var ctx=canvas.getContext('2d');
            ctx.fillStyle='#FF0000';
            ctx.fillRect(0,0,80,100);
        </script>
</body>
</html>
```

Canvas 2D 是 HTML5 草案的一部分，被众多的浏览器所支持， 包括 chrome, IE， Firefox， Safari 等等。 不仅如此，越来越多的浏览器开始支持在 Canvas 中绘制 3D 图形，该部分规范由 Khronos 组织定义。 也就是，通过传入适当的参数来调用 Canvas 的 getContext 函数以创建 3D 上下文对象用于绘制 3D 图形， 目前不同的浏览器支持不同的参数，例如 chromium 支持“webkit-3d”和”experimental-webgl”。

值得注意的是，2D 和 3D 是互斥的，不能同时在同一个 canvas 中操作它们，也就是说，当创建了一个 2D 的上下文对象后，你不能再为其创建 3D 的上下文，反之亦然。本章将重点介绍 Canvas 2D 方面的只是及在 WebKit 和 chromium 中的实现， Canvas 3D 也就是 WebGL 将在下一章中作介绍。

CanvasRenderingContext2D 介绍

前面说到，CanvasRenderingContext2D 是 2d 图形绘制的上下文对象，其提供了用于绘制 2d 图形的 API，W3C 工作组起草了标准的草案，详细见参考文献 2。该对象由 JavaScript 代码创建后，JavaScript 便可以调用它的 API 在画布上绘制图形了。这些 API 主要的作用就是在画布上绘制点，线，矩形，弧形，图片等等，除此之外，还提供了这些绘制的样式和合成效果等。下面按类简单介绍一下它所包含这些 API。

第一部分，如下两个函数 save 和 restore 用来保存或者恢复 2D 上下文的状态；

```
void save();
void restore();
```

第二部分，用来设置变换参数，例如缩放，旋转，平移等，这些构成了一个变换矩阵，缺省是 Identity 矩阵；

```
void scale(float sx, float sy);
void rotate(float angleInRadians);
void translate(float tx, float ty);
void transform(float m11, float m12, float m21, float m22, float dx, float dy);
void setTransform(float m11, float m12, float m21, float m22, float dx, float dy);
```

第三部分，对象的两个属性 globalAlpha 和 globalCompositeOperation，用来设置 alpha 值和合成模式，alpha 值表示将要绘制图形的透明度，合成模式表示将要绘制的图形和绘制对象如何合成，合成模式有多种，例如 source-over, source-in, source-out, destination-over,destination-in, destination-in 等等；

```
attribute double globalAlpha; // (default 1.0)
attribute DOMString globalCompositeOperation; // (default source-over)
```

第四部分，有关设置颜色和样式，包括两个属性 strokeStyle, fillStyle，gradient, pattern, 用来表示笔画和填充的格式，变化率，和用来填充图片的模式

```
attribute any strokeStyle; // (default black)
attribute any fillStyle; // (default black)
CanvasGradient createLinearGradient(in double x0, in double y0, in double x1, in double y1);
CanvasGradient createRadialGradient(in double x0, in double y0, in double r0, in double x1, ;
CanvasPattern createPattern(in HTMLImageElement image, in DOMString repetition);
CanvasPattern createPattern(in HTMLCanvasElement image, in DOMString repetition);
CanvasPattern createPattern(in HTMLVideoElement image, in DOMString repetition);
```

第五部分，有关线的样式属性，它们包括线宽度，线端样式及其线与线的连接方式等；

```
attribute double lineWidth; // (default 1)
attribute DOMString lineCap; // "butt", "round", "square" (default "butt")
attribute DOMString lineJoin; // "round", "bevel", "miter" (default "miter")
attribute double miterLimit; // (default 10)
```

第六部分，有关阴影方面的属性，这是全局的，影响所有绘制的图形；

```
attribute double shadowOffsetX; // (default 0)
attribute double shadowOffsetY; // (default 0)
attribute double shadowBlur; // (default 0)
attribute DOMString shadowColor; // (default transparent black)
```

第七部分，绘制矩形的相关操作，包括清除矩形区域，填充和绘制矩形边框；

```
void clearRect(float x, float y, float width, float height);
void fillRect(float x, float y, float width, float height);
void strokeRect(float x, float y, float width, float height);
void strokeRect(float x, float y, float width, float height, float lineWidth);
```

第八部分，Path 相关的操作，用来绘制自定义的路径的图形，这可以是图形可以是直线，弧形，曲线等；

```
void beginPath();
void closePath();
void moveTo(in double x, in double y);
void lineTo(in double x, in double y);
void quadraticCurveTo(in double cpx, in double cpy, in double x, in double y);
void bezierCurveTo(in double cp1x, in double cp1y, in double cp2x, in double cp2y, in double x,
void arcTo(in double x1, in double y1, in double x2, in double y2, in double radius);
void rect(in double x, in double y, in double w, in double h);
void arc(in double x, in double y, in double radius, in double startAngle, in double endAngle, in
void fill();
void stroke();
void clip();
boolean isPointInPath(in double x, in double y);
```

第九部分，与文字相关的操作，包括字体，文字对齐方式等属性和绘制文字等操作；

```
attribute DOMString font; // (default 10px sans-serif)
attribute DOMString textAlign; // "start", "end", "left", "right", "center" (default: "start")
attribute DOMString textBaseline; // "top", "hanging", "middle", "alphabetic", "ideographic", "bottom" (default: "alphabetic")
void fillText(in DOMString text, in double x, in double y, in optional double maxWidth);
void strokeText(in DOMString text, in double x, in double y, in optional double maxWidth);
TextMetrics measureText(in DOMString text);
```

第十部分，绘制图像到画布上，图像来源可以 Image 元素，Canvas 元素，或者是 Video 元素。通过设置的样式，可以把待绘制的图像以不同的方式绘制到目标的 Canvas 上；

```
void drawImage(in HTMLImageElement image, in double dx, in double dy, in optional double dw, in dc
void drawImage(in HTMLImageElement image, in double sx, in double sy, in double sw, in double sh,
void drawImage(in HTMLCanvasElement image, in double dx, in double dy, in optional double dw, in c
void drawImage(in HTMLCanvasElement image, in double sx, in double sy, in double sw, in double sh,
void drawImage(in HTMLVideoElement image, in double dx, in double dy, in optional double dw, in dc
void drawImage(in HTMLVideoElement image, in double sx, in double sy, in double sw, in double sh,
```

第十一部分，有关像素方面的操作，包括 ImageData 创建，以及从 canvas 读取内容创建 ImageData 和把 ImageData 内容写到 canvas；

```
ImageData createImageData(in double sw, in double sh);
ImageData createImageData(in ImageData imagedata);
ImageData getImageData(in double sx, in double sy, in double sw, in double sh);
void putImageData(in ImageData imagedata, in double dx, in double dy, in optional
                  double dirtyX, in double dirtyY, in double dirtyWidth, in double dirtyHeight);
```

第十二部分，定义与上面一些 API 相关的类（接口），例如 CanvasGradient，CanvasPattern 等，这些类街头连同上面的 API 都被定义 WebIDL 格式。

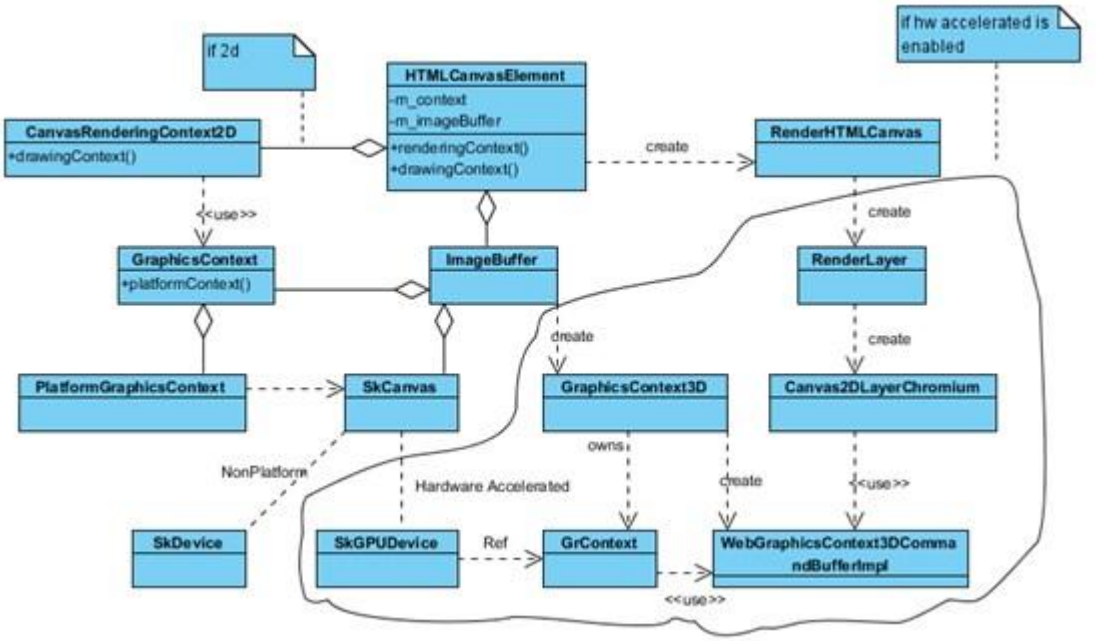
通过以上解释，那么 canvas2d 例子中的代码就比较容易理解了。那段 JS 代码首先创建一个 2D 的上下文对象，然后设置填充的颜色为红色，最后填充一个 80x100 的矩形内部为红色。你可以尝试把例子放在 chromium 里面运行然后看看能得到什么。值得提醒的是，当画布的宽度或者高度发生改变时，该画布上的所有绘制的内容都将被移除。

WebKit 和 Chromium 的支持

W3C 定义了 2D context 标准的草案， 这些接口保存在 IDL 文件中。 WebKit 根据这 IDL 来直接生成相关的 C++类的代码，这些类包括 CanvasRenderingContext2D,CanvasPattern, CanvasGradient, ImageData, TextMetrics 等， 它们和标准中的接口一一对应，你可以在 WebKit/Source/WebCore/html/canvas 中找到它们。那么它们被 JS 中的代如何码调用的呢？答案是生成 JavaScript 绑定。

Canvas2D 上下文类的具体绘制动作由实现平台决定， 其由软件或者硬件来完成取决于移植。下面， 我们来看看 WebKit 如何支持 Canvas2D 以及如何提供合适的接口将实现交给移植来完成的。

首先，大致理解一下 WebKit 和 chromium 中支持 Canvas2D 的重要的类模块和它们之间关系，如下图所示。同以前一样， 包括 WebKit 的基础类和与平台相关的绘图类及支持 canvas2D 硬件加速的类。



让我们简单了解一下这些类的作用：

WebKit 端：HTMLCanvasElement：DOM 中的对应着 HTML5 的 canvas 元素，该类包含有关为 2D 或者 3D context 服务的相关接口，主要的作用创建 JS 使用的 2D 或者 3D 上下文对象，绘图的平台无关的 GraphicsContext 对象，后端存储的 buffer
GraphicsContext：WebKit 的平台无关的上下文类，用于绘制工作，具体调用平台相关的上下文类来实现绘制
PlatformGraphicsContext：平台绘制上下文类，不同的平台有不同的实现，在 chromium 中是 PlatformContextSkia
ImageBuffer：WebKit 平台无关后端存储类，不同平台会定义不同的结构，在 chromium 中会使用 SkCanvas
RenderHTMLCanvas：RenderObject 的子类，为 canvas 而设计的

Chromium 端：

PlatformContextSkia： Chromium 中的 PlatformGraphicsContext 类
SkCanvas：skia 画布，包含所有的绘制状态，使用 SkDevice 来绘制图形
SkDevice：设备类，包含一个 SkBitmap 作为后端，利用光栅扫描算法来生成像素值保存于后端存储中，用于软件绘制方案
SkGpuDevice：设备类，包含一个绘制的目标对象，通过 GrContext 来绘制，其利用硬件加速的 GL 库来绘制 2D 图形
GrContext：GPU 绘制的上下文类，包含一个平台相关的 3D 上下文成员

Canvas2DLayerChromium:LayerChromium 的子类，包含一个硬件加速的 Canvas2D 层 其他类：之前介绍过，不再赘述上图中加速部分相关的 GraphicsLayer, CCLayerImpl 等，因为在之前介绍过，为方便起见，图中省略了它们。

Chromium 中的 Canvas2D 的绘制操作的实现都是由图形库 skia 来完成，这里包括软件和硬件加速实现，chromium 所要做的就是把 WebKit 中的调用交给 skia 来执行并和自己的绘制模型和硬件加速机制集成起来。 那么如何打开或者关闭 Canvas2D 的硬件加速功能呢？ Chromium 中提供了两个选项，分别是“--disable-accelerated-2d-canvas”和“--enable-accelerated-2d-canvas”，用户可以通过查看“about://gpu”来确认。

在介绍了基础设施之后，下面来看看具体的软件和硬件加速如何实现 Canvas2D 的。后面的软件实现和硬件加速实现都基于本章开始的 canvas 例子来说明。

Canvas 2D 的软件实现

这里我们以本章中的 Canvas2D 的例子来说这个过程。

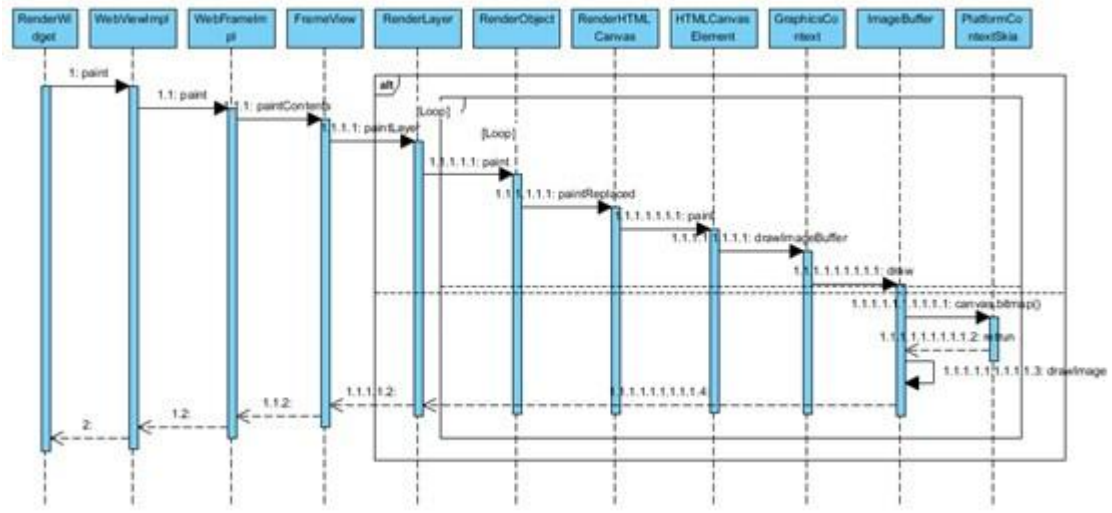
首先，当执行到 JS 代码中的 canvas.getContext 时，WebKit 通过 V8 JS 绑定会调用 HTMLCanvasElement.getContext。该函数根据传入的参数来决定创建 2D 或者 3D 的上下文对象。 在这里， CanvasRenderingContext2D 对象会被创建。此时其他有关的对象例如 ImageBuffer，GraphicsContext 等不会被创建，直到后面使用到时才会被创建，这是 WebKit 的做事原则。

其次，当设置 fillStyle 属性时，WebKit 同样通过 V8 JS 绑定调用 CanvasRenderingContext2D.setFillStyle，在这种情形下，2D 上下文对象会开始创建相关对象（在软件实现情况下，上图中右侧框中的对象不会被创建，包括 ImageBuffer， GraphicsContext， PlatformContextSkia， SkCanvas，SkDevice 和 SkBitmap）。

当执行 JS 的 fillRect 时，CanvasRenderingContext2D 调用 GraphicsContext 来完成绘制。 这两个类是 WebKit 的基础类，GraphicsContext 需要有不同移植来具体实现绘制工作，在 chromium 中，这就是 PlatformContextSkia。该类是一个转接口，调用 skia 来绘制。

SkCanvas 根据之前设定的样式，由 SkDevice 利用光栅扫描法来计算生成相应的像素值，结果保存在一个 SkBitmap 中。

最后，当 fillRect 操作调用完成之后，会安排一个 Invalidate 相关区域的命令。而后，当该命令被执行了，WebKit 会遍历 RenderLayer 依次绘制 RenderObject 的内容，当绘制 Canvas 元素时，会把之前 Canvas 绘制在 SkBitmap 的内容绘制到网页的 Bitmap 上（该 bitmap 通过共享内存机制会被 Browser 进程绘制在 Tab 子窗口中）。下图显示的是当有更新请求时，WebKit 遍历 RenderLayer 树和 Render 树来绘制网页及 Canvas 的详细调用过程。



Canvas 2D 的硬件加速实现

在硬件加速实现 Canvas 情况下，有一些地方是相似的，下面主要介绍它们的不同之处。

首先，硬件加速需要创建更多的对象和设施，主要有两点，一是如前面章节介绍的，会为 Canvas 元素创建一个新的 RenderLayer 及其相应的 GraphicsLayer，Canvas2DLayerChromium，CCLayerImpl 等。二是因为利用 GL 来渲染，所以为 skia 的 SkCanvas 创建一个 SkGpuDevice, GrTexture, GrContext 来使用 GL 绘制 2D 图形，同时，跟合成器一样，它也会创建 3D 的上下文对象- WebGraphicsContext3DCommandBufferImpl， 将 skia 的 GrContext， GrTexture 等对 gl 调用转发给 GPU 进程，具体的创建过程可参考 ImageBufferSkia.cpp 文件中的 createAcceleratedCanvas 函数。

其次，当调用 fillRect 时，canvas 的内容由 SkCanvas 调用 SkGpuDevice 将其绘制在 Texture，当然这些 GL 的操作都通过 WebGraphicsContext3DCommandBufferImpl 交给 GPU 进程来完成绘制。最后，会请求一个更新某个区域的任务。

最后，更新请求会调度合成操作，其首先调用 Canvas2DLayerImpl::paintContentsIfDirty 绘制自己，然后将 Canvas 的 Texture 合成起来生成网页内容。

源文件目录

```
WebKit/Source/WebCore/html/canvas/
          WebKit 支持 Canvas 相关的类，包括支持 2D 和 3D 上下文及其所涉及的其他类
WebKit/Source/WebCore/platform/graphics/skia/
          实现 WebKit 中的平台相关的 2D 图形功能，Canvas2D 在 chromium 中的实现依赖 skia 图形库
```

参考文献

HTML5 canvas element http://www.w3schools.com/html5/html5_canvas.asp
Canvas2D context http://www.w3.org/TR/2dcontext/
WebKit HTMLCanvasElement
http://developer.apple.com/library/safari/#documentation/WebKit/Reference/HTMLCanvasElementRef/HTMLCanvasElementClassReference.html#//apple_ref/doc/uid/TP40000947

WebGL 及其实现

概述

前面章节介绍了 Canvas2D，同时也介绍了在 canvas 中同样也可以绘制 3D 图形，也就是 Canvas3D 或者称为 WebGL。同 Canvas2D 不一样的是，WebGL 标准草案不是由 W3C 来起草的，而是 Khronos 组织来负责的，目前很多浏览器支持 WebGL，例如 Firefox， Chrome， Safari（仅限 Mac 平台）和 Opera 。但是，微软以安全性为由拒绝在 IE 中支持 WebGL，虽然它支持 Canvas2D。

WebGL 很热，自从诞生以来就备受关注，想知道 WebGL 有多酷，你可以查看参考文献 1 看看 Google 为大家制作的一些很棒的示例，看看网页中如何显示 3D 图形，相信你可以借此进一步了解它的魅力。

那么 WebGL 到底是何方神圣呢？根据官方标准中的说法，”WebGL 是一套为 Web 设计的基于立即模型绘制 3D 的 API。”它来源于 OpenGL ES2.0，提供了相似的 3D 绘图功能，只不过它运行在 HTML 网页中，简单上来理解， 你可以把它看成 OpenGLES 2.0 的 JavaScript 绑定，网页的开发者可以利用它来在网页中显示 3D 图形。WebGL 定义的是一个被 HTML Canvas 元素创建（同样通过 getContext）的用来 3D 渲染的上下文接口。

前面一章在介绍 Canvas 2D 的时候，曾经介绍过一个简单的例子，该例子是把 canvas 区域填充满红色。下面看一个 WebGL 的简单例子，功能一样， 把 canvas 区域填充满红色。

```
1 <html>
2     <body>
3         <canvas id="webGLCanvas" width="100" height="100"></canvas>
4         <script type="text/javascript">
5             var canvas = document.getElementById("webGLCanvas");
6             var gl = canvas.getContext("experimental-webgl");
7             gl.clearColor(1.0, 0.0, 0.0, 1.0);
8             gl.clear(gl.COLOR_BUFFER_BIT);
9         </script>
10    </body>
11 </html>
```

上图示例代码包含一个 Canvas 元素和一段 JavaScript 代码。根据以前的介绍，你应该已经知道，第 5 行是获得 canvas 元素的 DOM 节点对象，第 6 行代码通过传入“experimental-webgl”参数让该节点对象创建一个 3D 的上下文对象，之后就可以利用该上下文对象绘制 3D 图形。第 7 行设置用于下一行的颜色，格式为 RGBA，。最后一行将之前设置的颜色填充到颜色缓冲区中（对应于 canvas 的后端缓冲区之一），从而将 canvas 区域设置为红色。

同 Canvas2D 类似地，Canvas 元素提供了绘制的场所，WebGL 则提供了绘制 3D 内容的上下文类－WebGLRenderingContext，下面将对其详细介绍。与 Canvas 2D 不同的是，WebGL 没有软件渲染和硬件加速两种模式，它仅在硬件加速开启的情况下才能运行，因为它依赖于 3D 的图形库。

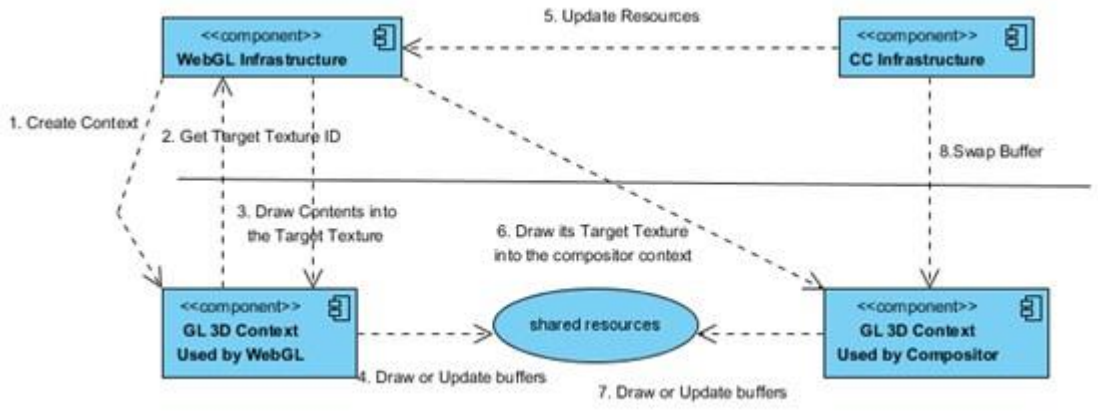
根据 WebGL 规范中的描述，下面来介绍一下它的几个主要部分：

上下文及内容展示：在使用 WebGL 的 API 之前，需要获取 WebGLRenderingContext 和 DrawingBuffer（Chromium 需要它）。对 JavaScript 来说，GL 的操作都是由 WebGLRenderingContext 来负责完成。但是， DrawingBuffer 对用户来说是透明的，它是用来存储渲染的内容并被合成器所合成，它包括帧缓冲器对象（绘制的结果存储）和纹理对象（纹理被合成器所使用）。
WebGL 的资源及其生命周期：textures,buffers (i.e., VBOs), framebuffers, renderbuffers, shaders and programs。它们有对应的 JavaScript 对象 WebGLObject 对应，它们的生命周期是一致的。
安全：为保证安全性，它要求：1，所有的 WebGL 资源必须包含初始化的数据；2，来源安全性，为防止信息泄露，当 canvas 的属性’origin-clean’为 false 时， readPixels 将会抛出安全方面的异常。3，要求所有的着色语言必须

符合 OpenGL ES shading language 1.0; 4, 为防止 DoS 攻击，建议采取适当的措施对花费时间过长的渲染操作过程进行监控和限制。
WebGL 接口：主要包括各种资源类的接口和上下文类的接口，用于绘制 3D 的操作，这些 API 基本上来源于 OpenGL ES 2.0.
WebGL 与 OpenGLES 2.0 的区别：这里不再一一介绍，读者可以自行翻阅和了解相关细节。 想要了解你的浏览器对 WebGL 支持的详细情况，请访问 http://webglreport.sourceforge.net/获取详细参数。

WebGL 的渲染过程

在这一部分将介绍 WebGL 如何被渲染以及如何被合成器合成的，大致的过程如下图所示。先解释以下图中的 5 个模块。WebGL 模块是指 WebKit 和 Chromium 中为支持 WebGL 而引入的基础设施。“GL 3D context used by WebGL”是 GPU 进程中为支持 WebGL 创建的 GLES 的实际的 3D 上下文对象。 CC 模块是 chromiumcompositor，就是合成器。“GL 3D context used by compositor”是 GPU 进程中为网页合成而创建的 3D 上下文对象。



第一步，当 JavaScript 的代码通过 HTML Canvas 对象创建 3D 上下文时，WebGL 模块便为其创建一个绘制 3D 图形用的上下文对象，该对象在 GPU 进程中会有一个实际的 OpenGL ES 的上下文对象对应。同时，DrawingBuffer 对象也会被创建。

第二步，DrawingBuffer 会同时创建了一个帧缓冲区对象，用于存储该 WebGL 绘制的内容，同时获取用于合成的纹理对象，WebGL 模块会获取这些对象的 ID 信息。

第三步，JavaScript 的调用上下文对象的绘制图形代码被 V8 解析后，利用 V8 的绑定机制，会调用相应的 WebGLRenderingContext 对象的 Callback 函数，这些函数把它们转换成内部的 Commands, 通过 IPC 机制传给 GPU 进程，GPU 进程在之前创建好的上下文对象，完成实际的绘制工作。在这过程中，有两点值得关注，第一，很多命令需要同步，这会有很大的开销，第二，很多情况下，JavaScript 需要操作各种图片资源，所以需要将它们共享给 GPU 进程，同时将它们上载到 GPU 的内存，因而开销比较大。

第四步，GPU 进程执行 renderer 进程发送过来的命令，调用 GL 库函数把计算结果内容更新存储到帧缓冲区中，前面这四步在解析 JavaScript 时，同时执行。之后，它会调度一个任务，请求更新一个矩形区域（同 viewport 大小）。

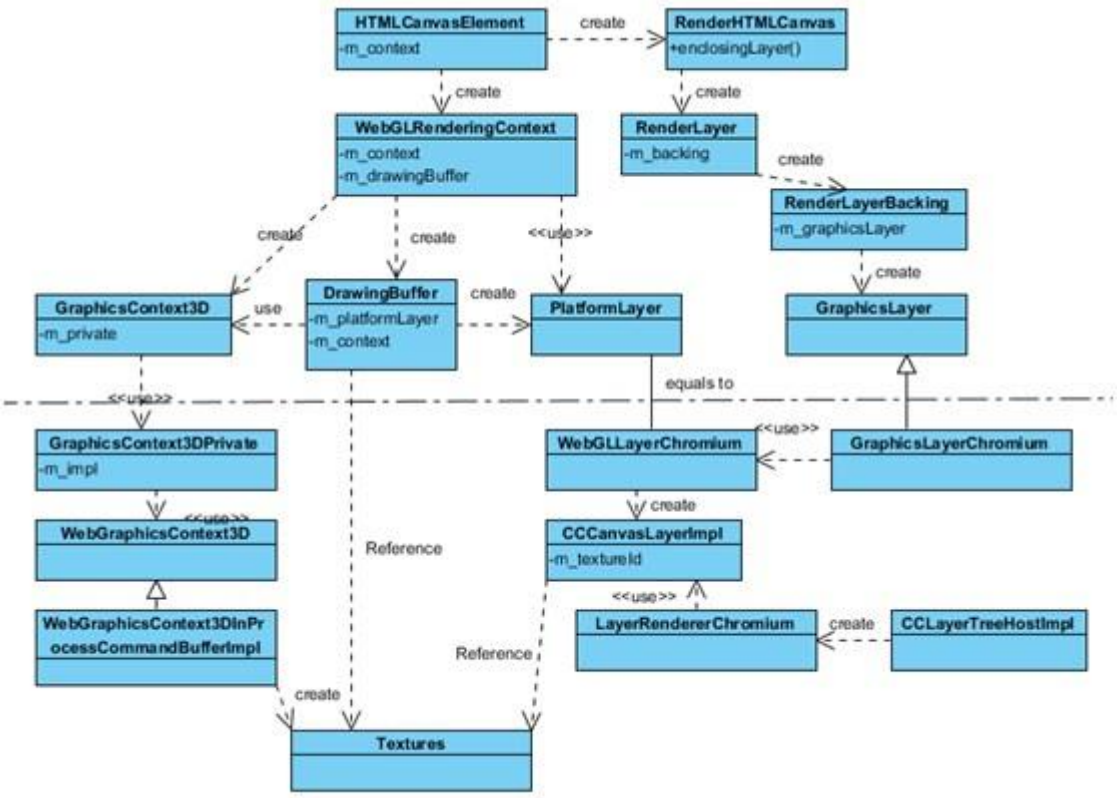
第五步，从这步开始，合成器发起合成，当由任何变化或者更新网页的请求时都会触发它。合成器要求每层去更新自己的发生变化的部分，这需要尽可能的快，以避免大幅地影响一次网页的绘制工作。WebGL 此时会准备绘制内容或者更新内容到帧缓冲区中，同时把内容拷贝到纹理对象中去，以备合成器所使用。

第六步和第七步，因为 WebGL 的纹理对象是在这些 GL 的上下文对象中共享的，WebGL 模块切换到合成器的上下文对象环境中，将它的纹理对象根据变换绘制到该环境中的帧缓冲区中去，完成对该层的合成。

第八步，当所有层次的绘制完成后，最后一步就是交换前后端的缓冲区，来显示合成完的内容。

WebKit 和 Chromium 基础设施

前面介绍了 WebGL 如何渲染的过程，那么 WebGL 模块在 WebKit 和 Chromium 中包含哪些相关的重要类来支持的呢？如下图所示，里面包括了一些主要的 WebGL 模块类。



HTMLCanvasElement：它是起点，是 DOM 中表示 Canvas 元素的类。如 Canvas2D 类似，当传入‘webgl’到它的方法 getContext 时，WebGLRenderingContext 对象就会被创建。 WebGLRenderingContext：JavaScript 中的 3D 操作基本上都是由该类来处理，也就是说该类其实是 JavaScript 和 GL 实现的桥梁。在 WebGLRenderingContext 创建的同时，DrawingBuffer 也同时被创建，但是目前只有 chromium 中会创建它。
DrawingBuffer： 表示规范中的 DrawingBuffer，在 WebKit 的 chromium 移植中，DrawingBuffer 会创建绘制用的 Chromium 的 3D 上下文对象和所需要的 buffer， 这些 buffer 最后会被 CC 的 CCCanvasLayerImpl 使用。
GraphicsContext3D： WebKit 中用于绘制 3D 图形的上下文基础类，其对图形的绘制接口都是对该类的调用，具体需要各个移植的实现。
GraphicsContext3DPrivate,WebGraphicsContext3D： Chromium 中的具体 3D 图形上下文的具体实现的两个辅助类。WebGraphicsContext3D 是一个 chromium 平台的基类，其子类可以是 In-Process 的 3D 绘图上下文实现类，也可以是 out-of-process 的 3D 绘图上下文实现类。
PlatformLayer，WebGLLayerChromium：平台相关的用于表示各个移植中的，其可以将绘制好的 buffer，发布到平台相关的层次上。

CCCanvasLayerImpl：CC 中的用于表示 canvaslayer 的类，其来具体完成平台层的绘制请求，并将它们的结果内容会知道合成器的结果中。

源文件目录

```
WebKit/Source/WebCore/html/canvas/
                    WebKit 支持 Canvas 相关的类，包括支持 2D 和 3D 上下文及其所涉及的其他类
WebKit/Source/WebCore/platform/graphics/chromium
                    Chromium 的 DrawingBuffer 支持，3D 上下文，及其为合成 WebGL 所涉及的类
```

参考文献

WebGL demos: http://www.chromeexperiments.com/webgl
WebGL Spec https://www.khronos.org/registry/webgl/specs/1.0/
WebGL Report for browser: http://webglreport.sourceforge.net/
OpenGL ES2.0 http://www.khronos.org/opengles/sdk/docs/man/

WebKit 的 CSS 实现

概述

前面章节介绍了 CSS 的三种基本要素，大概可以分成选择器，各种基本样式和 CSS3 引入的变形、变换和动画等。本章在此基础上，着重介绍 CSS 是如何在 WebKit 和 Chromium 得到支持的。首先介绍的是 CSS 解析器，而后分别阐述上面三种基本要素如何在 WebKit 和 Chromium 中实现的。

接前面章节，这里仍然以之前的 CSS 例子为基础来介绍本章的内容。为方便起见，依旧包含该例子，如下图所示。

```
1  <html>
2      <head>
3          <style type="text/css">
4              div  /* 选择器 */
5              {
6                  position: absolute; /* 位置 */
7                  top: 200px; /* 坐标 */
8                  left: 200px; /* 坐标 */
9                  width: 200px; /* 宽度 */
10                 height: 20px; /* 高度 */
11                 background-color: #efefef; /* 背景色 */
12                 color: green; /* 颜色 */
13                 border:2px solid black; /* 边框 */
14                 padding: 20px 20px 40px 40px; /* 填充大小 */
15                 opacity: 0.5;
16                 -webkit-transform:rotateZ(10deg); /* WebKit内核支持的变换属性 */
17             }
18         </style>
19     </head>
20     <body>
21         <p> This is a css test! </p>
22         <div id="adiv" class="aclass">CSS Style and Transform</div>
23         <script>
24             var rotate = 10;
25             function loop()
26             {
27                 var elem = document.getElementById("adiv");
28                 var value =  "rotateZ(" + rotate + "deg)";
29                 elem.style.webkitTransform = value;
30                 window.webkitRequestAnimationFrame(loop);
31                 rotate += 10;
32                 rotate = rotate % 360;
33             }
34             loop();
35
36         </script>
37     </body>
38 </html>
```

CSS 解析器

CSS 的词法其实并不复杂，其分析工作由 bison 完成，具体实现可以在文件 CSSGrammar.y 中找到，详细过程这里不在赘述。其主要工作是解析词法并调用 CSSParser 的回调函数来生成结果，例如上例中的第 4 行到第 17 行，第 4 行首先是识别选择器，后面是各个属性的识别，最后，这些全部构成了基本的 StyleRule 类型（关于类型会在后面介绍），所以调用 CSSParser 的 createStyleRule 函数，该函数将选择器和属性列表的中间表示最后处理生成其内部表示保存在一个规则 StyleRule 中。对于其他的类型，作类似的处理。

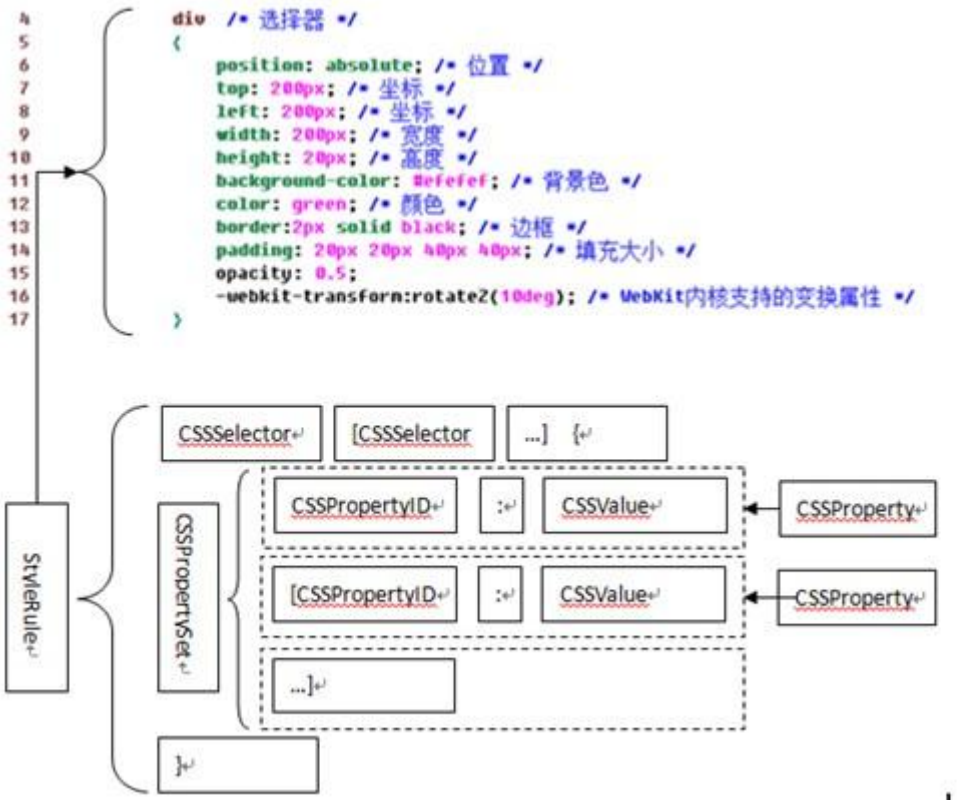
解析器对上层的接口是 CSSParser，所有任务均有其处理。那么上面例子中有哪些场景会需要创建 CSSParser 的实例呢？显而易见地有两处地方，第一个是第 4 行到第 17 行，当 DOM 建立好之后在创建 RenderObject 的时候会调用 CSSParser；第二个是第 29 行，在该段 JavaScript 代码被执行时，JavaScript 引擎会间接调用 CSSParser 为元素的属性 style 解析。但其实还有一个地方会使用到 CSSParser。

实际上 WebKit（其他渲染引擎应该也类似吧 K）会为每个网页设置一个缺省的样式，这决定了你所没有设置的元素及其它们的属性缺省值和将要显示的效果。通常来讲，不同的 WebKit 的移植（port）会有不同的缺省样式。

解析后的结果应该是一系列的样式规则，下面让我们来看看这些规则在 WebKit 中的内部表示。

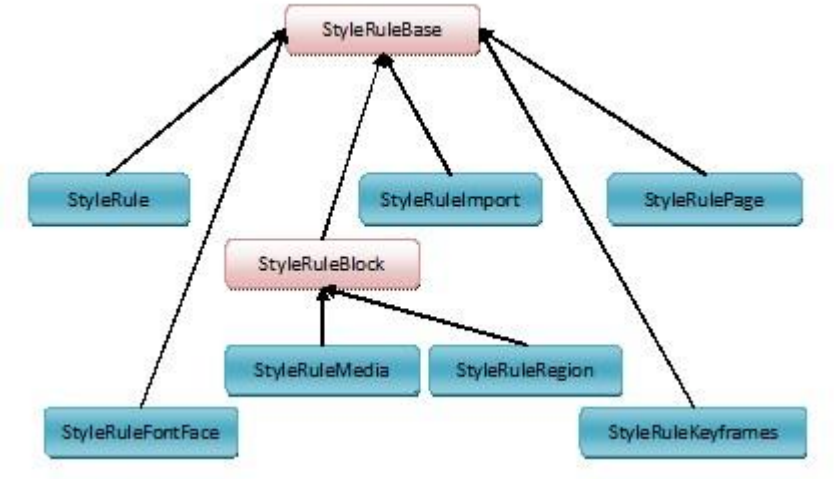
样式的内部表示

被解析后的 CSS 样式其实就是一组样式规则，每一个规则包含一组选择器和一组样式属性。如前面例子中第 4 行到第 17 行所示。这些数据在 WebKit 中都有相应的内部表示，为了便于理解，下图给出了 CSS 的表示和内部结果表示的对应图。



值得一提的是选择器部分，上图是一个选择器列表，这在现实中是比较常见的。举个例子， a[class=abc]其实是两个选择器，第一个是’a’，它一个标签选择器；第二个是’[class=abc]’，它是一个属性选择器。 这里例子中的样式规则是一种基本的样式类型。CSS 的标准包含了多种规则类型：

Style：这个是基本类型，一般大多数规则属于这个类型；
Import：是 WebKit 中为方便引入的，其对应的是一个导入 CSS 文件的 Style 元素
Media：对应于 CSS 标准中的@media 类型
Fontface：CSS3 新引入的自定义字体的规则类型
Page：对于 CSS 标准中的@page 类型
Keyframes：WebKit 的@-webkit-key-frames 类型，可以用来指定特定帧的样式属性信息
Region：对 CSS 标准正在进行中的 Regions 的支持，这方便了开发者对页面进行分区域来排版。详见参考文献 2. 下图给出了这些类型在 WebKit 中的定义及其关系。



这里有必要解释一下 StyleRuleImport 类，这个一个伪类型，CSS 中并没有该类型的定义，只是 WebKit 处理 CSS 文件方便引入的。该类需要两个类来辅助完成，一个是 StyleSheetContents 类，就是 CSS 文件中各个规则的处理和内部表示，这些规则被称为子规则，构成一个列表；另外一个是 CSSStyleSheet，该类是个包装类，代表 CSS 文件，包含一个 StyleSheetContents 对象。

选择器（CSSSelector 类）

CSS 选择器的实现并不复杂，其实现由类 CSSSelector 来完成。CSSSelector 的作用是储存从解析器生成的结果信息以被匹配使用。这里匹配指的是当需要为每个 DOM 中的节点计算样式适合，WebKit 需要根据当前的节点信息来从规则列表中找到能够符合调节的规则，并把规则中的属性列表提取出来生成节点的样式信息。 在 CSS 介绍那一章中，我们知道有多达 42 种选择器类型，例如。在 WebKit 的 CSSSelector 类中，它们被称为伪类型（PseudoType），其含义是表示 CSS 标准中定义的类型，而主要用于匹配算法的类型定义为 Match，它包含 Id, Class, Extract, Set, List, Hyphen，PseudoClass,PseudoElement, Contain, Begin, End, PagePseudoClass。这些类型是通过抽象标准中的类型而得来，例如:first-of-type，:last-of-type，:only-of-typed 等等都属于 PseudoClass。具体的匹配算法参考代码（文件 CSSSelector.cpp），这里不在赘述。

StyleResolver 和 RenderStyle

在样式规则解析完成之后，剩下的问题是如何把这样规则应用到具体的元素上。这就涉及到两个主要的类：StyleResolver 和 RenderStyle。

StyleResolver 是管理类，其负责根据样式规则为每一个 Document 中的元素匹配响应的样式属性，它和 Document 节点是一一对应关系，也就是说 WebKit 为每个 Document 创建一个 StyleResolver 对象，为所有该 Document 中的节点计算样式，并将其结果保存到 RenderStyle 对象中。

RenderStyle 是元素所有样式属性的内部表示。由于其包含了所有样式属性，为了节约空间，WebKit 将属性分为两种类型：常用属性和非常用属性。非常用属性会进行分组合并，并且仅在需要时创建，这相对有效地节约了内存。该对象在被 StyleResolver 创建后由该元素所对应的 RenderObject 所拥有。

那么 StyleResolver 是如何为一个 DOM 元素生成 RenderStyle 对象的呢？大致地有如下几个主要步骤：

首先创建一个新的 RenderStyle 对象
从它的父亲那里继承它的一切可以继承的属性
如果是 link 类元素，设置 link 属性
而后是样式规则的匹配，从已知规则中找到匹配到的属性
将匹配到的属性应用到 RenderStyle 对象中

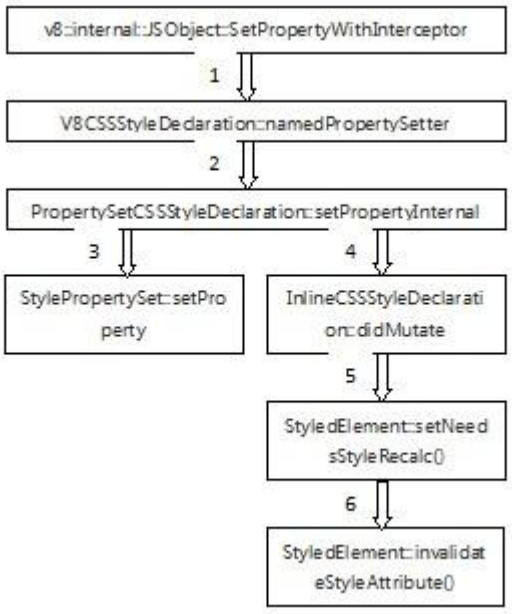
为该 DOM 元素的 RenderStyle 做一些修正工作
清理 StyleResolver，为下次匹配请求做准备

RenderStyle 和 RenderObject

元素在匹配生成其样式属性值之后，RenderStyle 对象被 RenderObject 所获得，这个触发一个重新绘制的动作，WebKit 此时可以根据样式属性值来计算它的布局和显示，这将在下一章作详细介绍。

JavaScript 设置样式

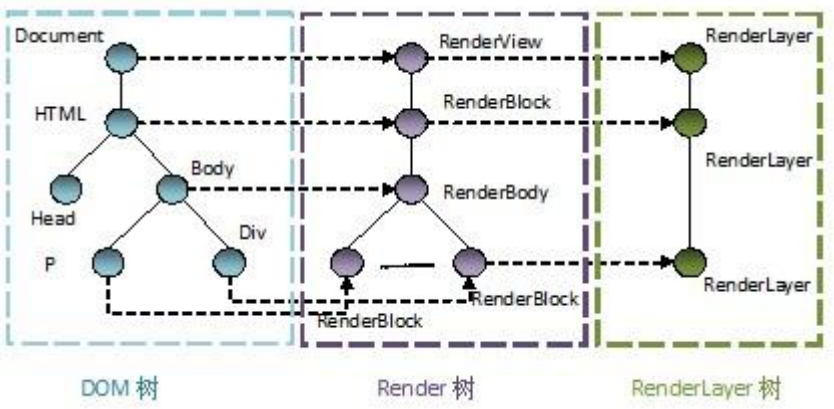
JavaScript 有能力设置任何元素的样式值，如例子中第 29 行所示，其原则是覆盖样式中同一属性的值。大致的过程是，JavaScript 引擎调用设置属性值的公共处理函数，然后该函数调用属性值解析函数，在这个例子中则是 CSS 的 JS 绑定函数，而后将解析后的信息设置到元素的 style 属性的样式 webkitTransform 中，然后设置该元素需要重新计算 style 和 invalidate 它的 style 属性，如下图所示。



在这之后，重新绘制请求被处理时，WebKit 先会重新计算布局，而后在渲染相应的区域。

CSS3 变形（transform）、变换(transition)和动画(animation)

在 WebKit 渲染基础中，我们介绍了 DOM 数，Render 树和 RenderLayer 树，根据 WebKit 的设计原则，可以知道上面 HTML 例子的三种树结构如下图所示。也就是说，元素 P 的内容会被包含在中间的 RenderLayer 中，Div 元素的内容会被会包含单独的最下面的 RenderLayer 中，因为 Div 元素的需要做 CSS 的 3D 变形。



我们知道 RenderLayer 会有一个后端存储空间，Chromium 中是一个 FrameBuffer，该层会转成一个 Texture，那些 CSS3 的变形，变换和动画效果将作用于该 Texture 上，而后绘制在网页的 framebuffer 中。 变形比较简单，WebKit 和 Chromium 内部有一个 Matrix 来表示平移，旋转，缩放和扭曲等变形，因为 OpenGL 有直接对变形的支持，所以只需创建响应的 OpenGL 变换操作即可。

变换和动画的实现大致相同，前面解析的过程跟其他属性没有区别，其结果会保存于 Animation 对象中。在 chromium 的实现中，CC(chromium compositor, 前面介绍过)中有两个主要类对其提供支持，一个是 CCLayerAnimationController，一个是 CCActiveAnimation。前者是控制动画的行为，后者则是包含动画相关参数和状态。这一部分其实很复杂，有兴趣的读者可以自行阅读代码。

源文件目录

```
third_party/WebKit/Source/WebCore/css/
  与 CSS 解析，内部表示生成等一系列相关的类
third_party/WebKit/Source/WebCore/rendering/style
  渲染所需要的样式的支持类，其依赖于 CSS 解析器及其结果
```

参考文献

http://www.w3schools.com/css
http://dev.w3.org/csswg/css3-regions/

WebKit 布局 (Layout)

概述

一个网页从文本信息到最后的渲染结果，要经过很多复杂的过程，前面介绍过 DOM 树、Render 树的创建，也阐述了页面如果被渲染的，其实，这两者中间还有一个非常重要的步骤－－布局计算，这是因为在渲染每个元素之前，渲染引擎必须知道它的位置大小等布局信息，我们把计算这些信息的过程称之为布局。

布局根据其计算的范围大致可以分为两类，第一类是对整个 Render 树进行的计算，第二类是对 Render 树中某个子树的计算，常见于文本元素或者是 overflow:auto 块的计算，这种情况一般是其子树布局的改变不会影响其周围元素的布局，因而不需要重新计算更大范围内的布局。

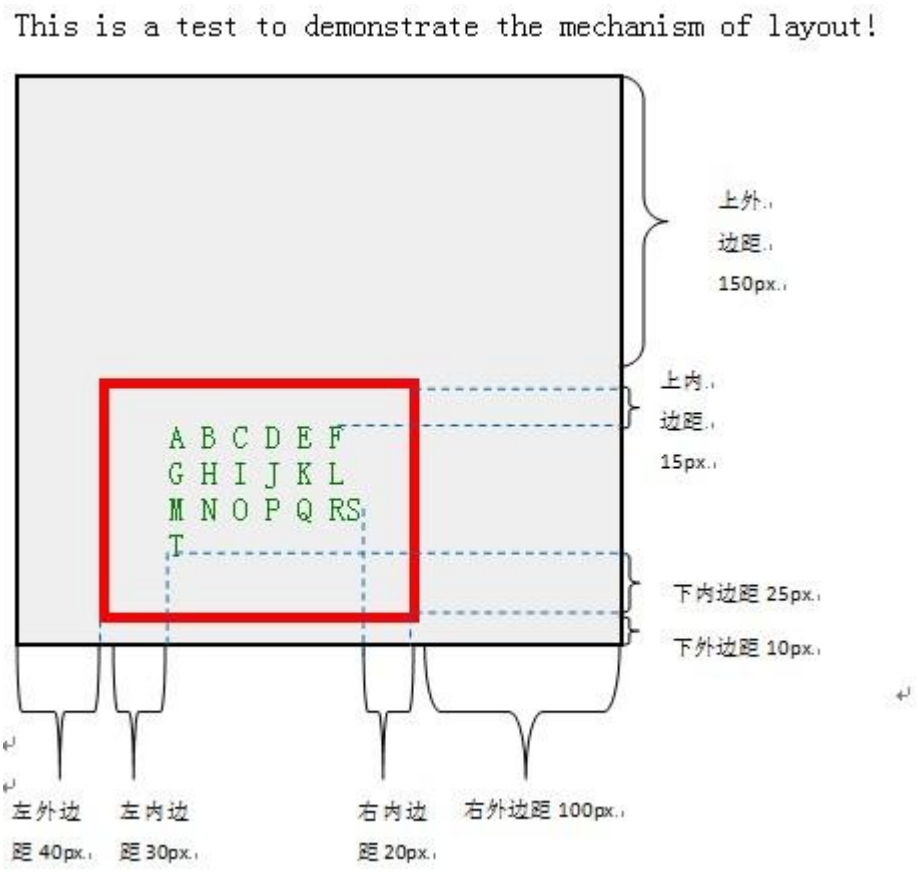
按照惯例，为了便于表述，首先我们来看一个能够帮助理解问题的例子。这个例子不复杂，其重点是包含两个 div 元素，其中 id 为 adiv 的元素是类型为 aclass 元素的父亲。CSS 样式设置了类型 aclass 的样式属性值。

```
<html>
    <head>
        <style type="text/css">
            #adiv /* ID选择器 */
            {
                width: 300px; /* 宽度 */
                background-color: #efefef; /* 背景色 */
                border:2px solid black; /* 边框 */
            }
            .aclass /* 类选择器 */
            {
                border: 5px solid red; /* 边框 */
                margin: 150px 100px 10px 40px; /* 外边距 */
                padding: 15px 20px 25px 30px; /* 内边距 */
                color: green; /* 颜色 */
            }
        </style>
    </head>
    <body>
        <p> This is a test to demonstrate the mechanism of layout! </p>
        <div id="adiv">
              <div class="aclass">A B C D E F G H I J K L M N O P Q RS T</div>
       </div>
    </body>
</html>
```

箱子(box)模型

先介绍箱子模型。CSS 布局计算是基于箱子模型来进行的，其基本构成是一个矩形区域，包含了外边距(margin)，内边距(padding)，边框(border)和内容(content)，也就是说每个元素的布局都是按照箱子模型来排布的，通过设置这些属性，达到特定的布局效果。如下图所示，最外层的虚线以内就是一个 box 模型的实例，这个实例表示的就是一个元素布局表示。

这个图相信大家都看过，这里简单介绍一下。箱子最边缘部分分别是四个方向上的外边距（TM, RM, BM, LM），可以设置不同的大小，其次往内是四个方向上的边框(TB, RB, BB, LB)，再次是四个方向上的内边距(TP, RP, BP, LB)，最后是中间的内容。下面结合实例来解释，给人以直观印象，便于深入了解。下图就是本章开始时候介绍的示例在浏览器中的显示结果，最外边的矩形框就是 ID 为‘adiv’的 div 元素的显示区域，其内部就是对于属于’.aclass’的类型的 div 元素的 box 模型实例。旁边的标注表明了 box 模型的各个属性值，读者可以跟上面的箱子模型进行对照理解。



例子中，文字被设置为‘A B C… T’，其表示的是内容区域，之所以设置成这样，是为了让布局随时可以换行，便于读者更清晰地看到内边距和内容区域。同样地，建议读者修改本章例子的各个参数，亲身体验这些设置带来的布局变化。 介绍了箱子模型后，箱子内部的位置和大小可以被确定了，那么箱子本身的位置和大小是如何确定的？这就需要“包含块”的帮助了。

Containing block(包含块)

当需要计算元素的箱子的位置和大小的时候，需要计算它和另外一个矩形区域的相对位置，这个矩形区域称为该元素的包含块，箱子模型就是在包含块内计算和确定其一系列的属性值的，包含块的具体定义如下：

1) 根元素的包含块称为初始包含块，通常它的大小就是可视区域（viewport）的大小。 2) 对于其他位置属性设置为’static’或者’relative’元素，它的包含块就是的最近的祖先的箱子模型中的内容区域(content)。 3) 如果元素的位置属性为’fixed’，其脱离文档，固定在可视区域的某个特定位置。 4) 如果元素的位置属性为’absolute’，其包含块由最近的包含属性’absolute’，‘relative’或者’fixed’的祖先决定，具体规则如下： a) 如果其是一个 inline 元素，那么元素的包含块是包含该祖先的第一个和最后一个 inline 盒子的内边距的区域。 b) 否则，则是由该祖先的内边距所包围的区域。

这个复杂的定义读起来比较让人头痛，webkit 中简单理解起来就是：

Render 节点的包含块是该节点的负责决定该节点的祖先节点对应的块区域。根节点 RenderView 表示的就是”初始包含块”，其初始大小始终是可视区域的大小。

结合实例来讲，类型为’aclass’的 div 元素的包含块就是父亲的内容区域，其箱子模型就是在该内容区域上进行计算生成得来的。

布局计算

布局计算的相关信息都保存在 RenderStyle 对象中，如之前介绍，该对象属于 Render 节点。 RenderStyle 没有什么特别之处，就是包含各个样式的属性值。同时，Render 节点也包含一个位数组，该数组会保存一些用来表示是否需要重新计算布局等信息。

下面看看如果计算布局的。其主要由 RenderObject 中的 layout 方法来完成： 首先，layout 函数会判断 Render 节点是否需要重新计算，通常这通过检查位数组中的相应标记位，子女是否需要计算布局等来确定。一般来说，初始显示，可视区域变化，样式值变化（例如动画，JavaScript 操作样式值）等都会需要重新计算布局。

其次，它会遍历其每一个子女节点，依次计算它们的布局。

再次，对于每一个元素来说，它会实现自己的 layout 方法，根据特定的算法来计算该类型元素的布局。如果页面元素定义了其自己的宽高，那么 webkit 按照其定义的宽高来确定其大小，而对于象文本节点这样的 Inline 元素则需要结合其字体大小及文字的多少等来确定其对应的宽高。如果页面元素所确定的宽高超过了布局容器包含块所能提供的宽高，同时其 overflow 属性为 visible 或 auto，则会提供滚动条来保证可以显示其所有内容。除非定义了页面元素的宽高，一般说来页面元素的宽高是在布局的时候通过相关计算得出来的。

布局测试(layouttests)

渲染引擎要处理各式各样、越来越复杂的网页，这需要 Layout 测试来保证它的质量。 Layout 测试可以说是 WebKit 中最重要并且最著名的测试了。其基本测试工作方式是，预先准备很多很多的简单网页和期望的渲染结果，然后根据 WebKit 编译出来的 DumpRenderTree（DRT）来测试网页，把得到的结果和期望的结果进行对比，以检查 WebKit 引擎的对网页排版布局等的正确性。每个 WebKit 的移植都会提供一个 DumpRenderTree，通查由于移植的差异性，它们的期望结果也不一样，所以通常每个移植都有自己特殊的期望结果。

每个测试都会有一个或者多个期望结果，一般地，期望结果是一些文本结果，但是，对一些复杂的测试，单纯的文本不能够满足需求，因为测试渲染结果可能需要比较布局，字体，图片等等，所以这时候期望结果其实是一幅图片（还有其他类型），这个图片其实就是网页应该渲染的结果。不幸的是，由于字体，平台的样式等差异性（例如 Qt，GTK 等就不一样），相同的网页渲染出的结果可能不一样，所以，你可以看到布局测试对不同的移植会有不同的期望结果。

一般来讲，当开发者提交新的补丁时候，需要先进行布局测试，只有当该测试通过，才有可能被 WebKit 所接受。如果你提交的是解决了一个新问题，那么，建议你提交一个新的测试用例来保证代码的正确性。

源文件目录

```
third_party/WebKit/Source/WebCore/rendering/style
  渲染所需要的样式的支持类，其依赖于 CSS 解析器及其结果
```

参考文献

http://www.webkit.org/projects/layout/index.html
http://www.w3.org/TR/CSS21/box.html#box-dimensions
http://www.webkit.org/blog/1452/layout-tests-theory/
http://www.webkit.org/blog/1456/layout-tests-practice/

插件机制(NPAPI Plugin)

概述

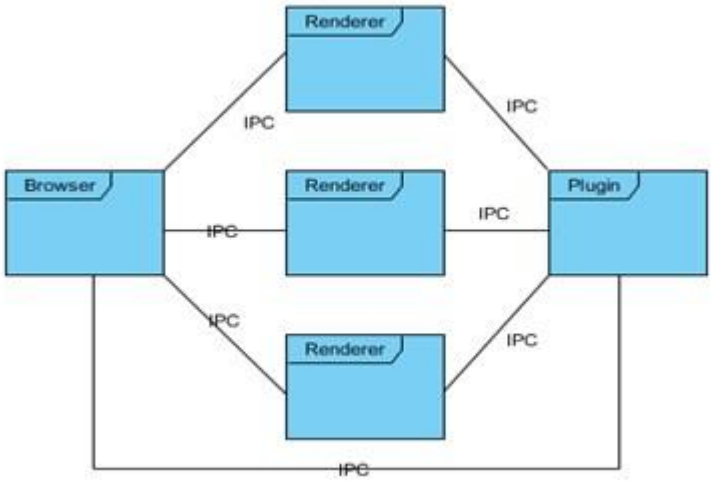
Chromium 中的 NPAPI 插件(plugin)来源于 mozilla 的插件机制。因为它被广泛的应用，很多插件厂商或者开发者基于它编写了数以万计的插件，因而 chromium 对它也提供了支持，不过 chromium 有自己独特的插件架构，后面我们会详细介绍。

NPAPI 提供两组接口，一类以 NPP 打头，由插件来实现，被浏览器调用，主要包括一些插件创建，初始化，关闭，销毁，信息查询及事件处理，数据流，窗口设置，URL 等；另一类以 NPN 打头，由浏览器来实现，被插件所调用，主要包括图形绘制，数据流处理，浏览器信息查询，内存分配和释放，浏览器的插件设置，URL 等。

原始的 NPAPI 的接口使用起来不是很方便，因而有贡献者对其进行了封装以利于其使用。一个比较著名的开源项目是 Firebreath。它将原始的 C 风格的 NPAPI 进行封装成 C++风格的接口，非常方便用户使用，而且有针对 Windows 和 X window 的移植，用户无需对底层特别了解。特别的是，Firebreath 也有对 ActiveX 的封装，因而对于现在主流的两种插件接口，你都可以基于 Firebreath 的接口进行编程，极大地方便了开发者。详情请参考 Firebreath 主页 http://www.firebreath.org/display/documentation/FireBreath+Home

架构

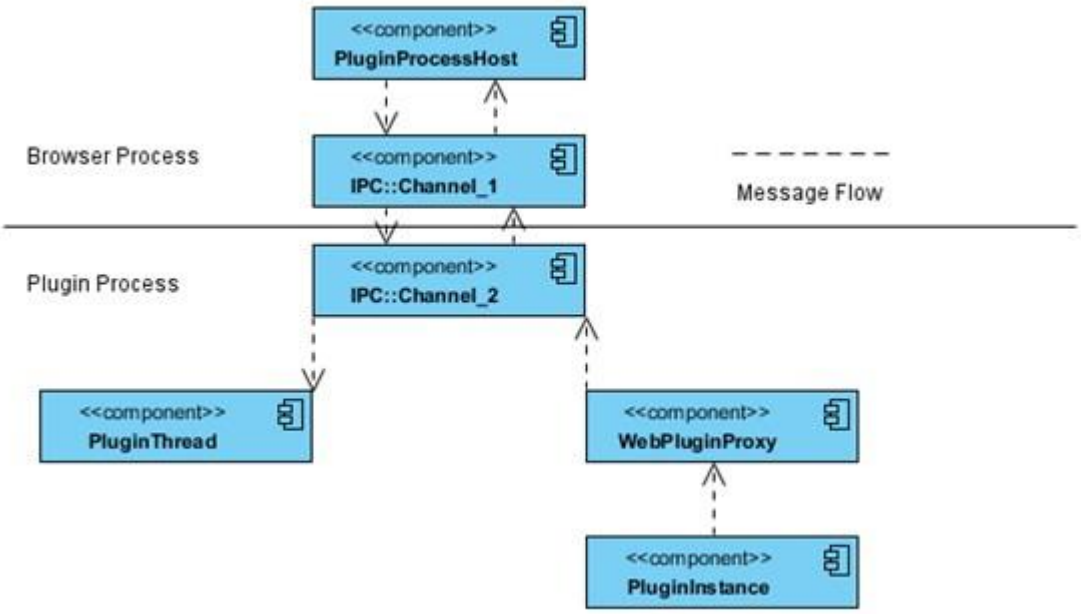
因为 chromium 的安全模型，renderer 进程没有访问除 I/O 读写等之外的权限，因而插件需要有自己的进程，这就是插件的 out-of-process 模型。下图给出 chromium 中插件的架构和进程模型。



每一种类型的 plugin 只有一个进程，这就是说，如果有两个或者多个 renderer 进程同时使用同一个插件，那么该插件会共享同一个进程。因为多个 renderer 进程共享同一种的 plugin 进程，那么 plugin 进程如何为它们服务呢？答案是为每个插件使用点在 plugin 进程中创建一个插件实例(PluginInstance).

值得注意的是，plugin 进程是由 browser 进程来负责创建和销毁，而不是 renderer 进程。 原因在于 renderer 进程没有创建的权限，而且 plugin 进程由 browser 进程来统一管理也更方便。 当 plugin 进程创建成功时，browser 进程会返回 IPCchannel handle 用于创建和 plugin 进程通讯的 PluginChannelHost. 那它什么时候被销毁呢？当没有任何插件实例并且空闲一段事件后，它才会被销毁，这样做的好处是避免频繁的创建和销毁 plugin 进程。

下图描述了 browser 和 plugin 进程间的通讯机制及其所涉及的相关的模块（类）。Browser 进程通过 PluginProcessHost 发送消息调用 Plugin 进程的函数，响应动作由 PluginThread 完成。而 Plugin 进程则是通过 WebPluginProxy 发送消息调用 browser 的函数，响应动作有 PluginProcessHost 完成。

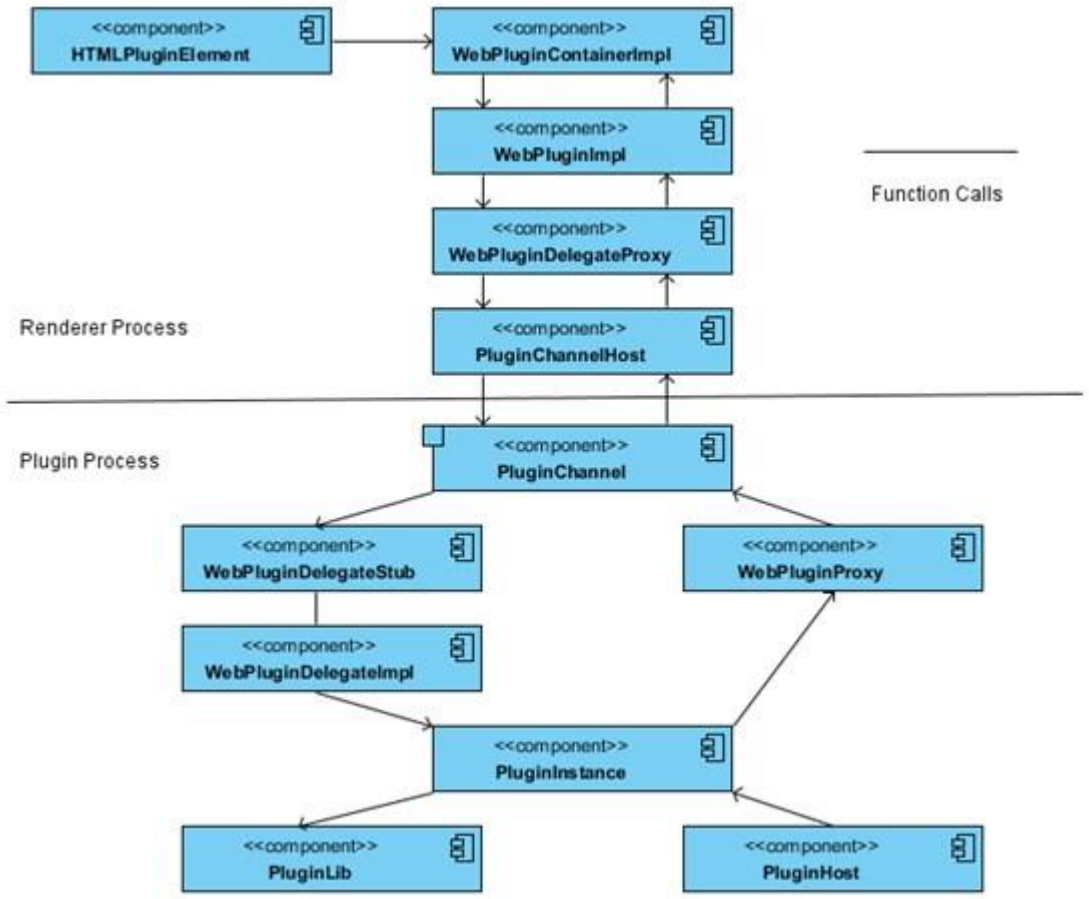


Browser 和 Plugin 仅有较少的消息传递，用于创建等管理工作。主要的部分在 renderer 进程和 plugin 进程之间，机制也相对更复杂一些。HTMLPluginElement 会包含一个 WebPluginContainerImpl，而它包含一个 WebPluginImpl，对 plugin 的调用有 WebPluginDelegateProxy 负责中转。在 Plugin 进程中，由 WebPluginDelegateStub 处理所有 renderer 过来的请求，并由 WebPluginDelegateImpl 调用创建好的 PluginInstance 对象。PluginInstance 最终调用 PluginLib 读取的插件的函数入口地址，最终完成对插件实现的调用。而对插件实现中对 NPN 开头函数的调用，则是通过 PluginHost 来完成。

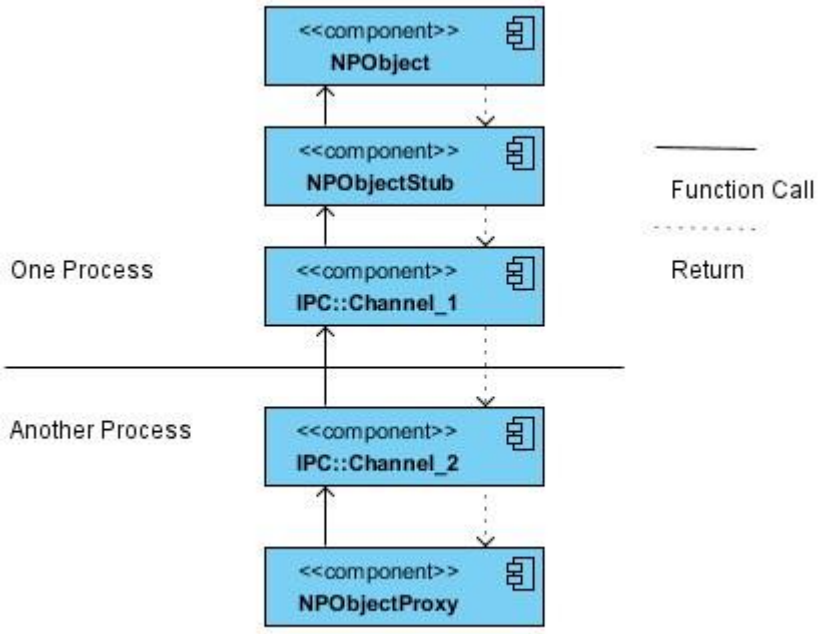
PluginHost 主要负责实现 NPN 开头的函数，如前面所描述，这些函数被 plugin 进程所调用。可以在 plugin 和 renderer 进程被调用。当在 plugin 进程调用这些函数时，chromium 会覆盖 PluginHost 的部分函数，而这些新的 callback 函数会调用 NPObjectProxy 来通过 IPC 发送请求到 renderer 进程。

PluginInstance 实现了 NPP 开头的函数，被 render 所调用（webpluginImpl 通过 webplugindelegateImpl 来调用），PluginInstance 通过 PluginLib 获得了插件库的函数地址，从而把实际的调用桥接到具体的插件中。

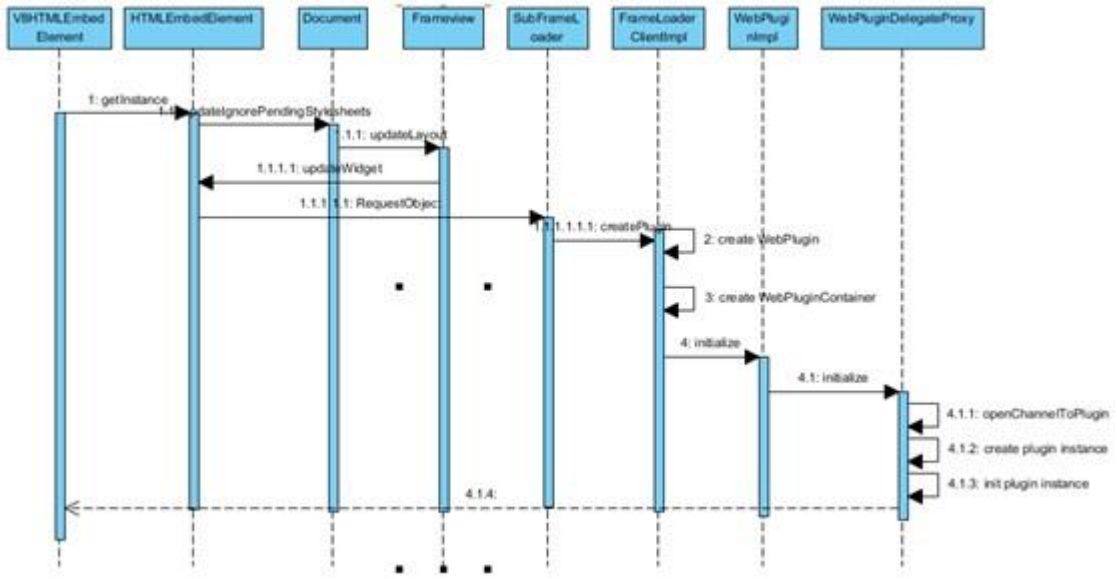
具体的见下图所示，主要结构来源于 chromium 的官网，略有修改。



对于 NPObject 相关的函数调用， 有专门的类来处理。NPObject 的调用或者访问是双向的（renderer 进程<->plugin 进程），他们的具体实现是通过 NPObjectProxy 和 NPObjectStub 来完成。 NPObjectProxy 接受来自对方的访问请求，转发给 NPObjectStub，最后 NPObjectStub 调用真正的 NPObject 并返回结果。见下图所示。



下面示例给出一个插件如何被 renderer 进程触发创建的过程。



当页面中包含一个“embed”或者“object”元素，renderer 进程会创建一个 HTMLEmbedElement 元素，当该元素被访问是，会触发创建相应的插件。HTMLEmbedElement 会请求创建自己对应的 RenderWidget（WebPluginContainerImpl），进而创建 WebPluginImpl 和 WebPluginDelegateProxy。WebPluginDelegateProxy 会发送消息来创建 Plugin 进程。Plugin 进程被 browser 进程创建后，会响应 renderer 的请求来创建 PluginInstance 并初始化它。这样它们之间的联系就建立好了。

Window 和 windowless 插件

可以通过设置”embed”或者”object”元素的属性。两者的区别主要在于绘图的方式不同。Window 插件由 renderer 进程提供一个窗口(window)，插件直接在该窗口上进行绘制；而 windowless 插件则不同，插件将绘制的结构（Pixmap），通过共享内存方式(Transport DIB)传递给 renderer 进程，renderer 绘制该内容到自己内部的存储结构(backing store)上。好处是，可以把插件绘制的结构和网页上的其他内容做各种形式的合成。但是，从上面不能看出，一般来讲，window 模式的性能是要高于 windowless 的。

相关目录和文件

```
third_party/WebKit/Source/WebKit/chromium/public/webplugin.h:
        定义 Webkit::WebPlugin 接口，用于创建和销毁插件，及传送事件给插件
content/common/plugin_messages.h:
        定义 plugin 进程与 renderer 进程和 browser 进程间的消息结构体，包括五类：
PluginProcessMsg 开头的消息－ browser 进程发给 plugin 进程
PluginProcessHostMsg 开头的消息－ plugin 进程发给 browser 进程
PluginMsg 开头的消息－ renderer 进程发给 plugin 进程
PluginHostMsg 开头的消息－ plugin 进程发给 renderer 进程
NPObjectMsg 开头的消息－ plugin 进程与 render 进程相互编码和解码 NPObject
content/common/npobject_stub.(h&cc):
                类 NPObjectStub 的定义和实现
content/common/npobject_Proxy.(h&cc):
                类 NPObjectProxy 的定义和实现
content/plugin:
        该目录用于存放 Plugin 进程使用的 IPC 和 WebPlugin 相关的函数和类
content/plugin/webplugin_proxy.(h&cc):
                实现 WebKit::npapi::WebPlugin 接口，把插件的调用通过 IPC 发送给 renderer 进程
content/plugin/webplugin_delegate_stub.(h&cc):
```

把 WebPluginDelegateProxy 的消息转换为对 WebPluginDelegateImpl 的调用
content/plugin/plugin_channel.(h&cc):
定义 Plugin 进程与 renderer 进程通信的通道
webkit/plugins/npapi/:
该目录用于存放 Plugin 进程使用的对 WebKit 接口实现和插件库处理的相关的函数和类
webkit/plugins/npapi/webplugin.h:
定义 WebPlugin 接口，用于 plugin 端同 web frame 和 webcore 对象的交互
webkit/plugins/npapi/webplugin_impl.(h&cc):
实现 WebKit::WebPlugin 和 WebPlugin 接口，通过 WebPluginPageDelegate 来创建 WebPluginDelegate
webkit/plugins/npapi/plugin_instance.(h&cc)
PluginInstance 类的接口和实现，代表一个 Plugin 实例，对应于 renderer 进程的一个请求创建的 plugin 实例
webkit/plugins/npapi/webplugin_delegate.h:
定义 WebPlugin 代理，用来分离具体的插件的实现方式，例如可以使来实现进程内或者跨进程的插件架构，对 WebPlugin 来说，具体架构是透明的
webkit/plugins/npapi/webplugin_delegate_impl.(h&cc):
响应 renderer 进程实现对 PluginInstance 的调用请求，有 gtk，win 和 aura 三种不同的实现
webkit/plugins/npapi/plugin_host.(h&cc):
实现 NPN 开头的函数，在 plugin 进程和 renderer 进程有不同的实现
webkit/plugins/npapi/plugin_lib.(h&cc):
PluginLib 用来管理实际插件库的生命周期
content/renderer/webplugin_delegate_proxy.(h&cc)
定义和实现 WebPluginDelegateProxy 类，桥接所有来自于 WebPlugin 的请求到 WebPluginDelegateImpl
content/browser/plugin_process_host.(h&cc):
类 PluginProcessHost 的定义和实现，用于 Browser 进程与 plugin 进程的交互
content/browser/plugin_service_impl.(h&cc):
类 PluginServiceImpl 的定义和实现，用于响应 Renderer 进程创建插件请求及其他一些插件管理工作

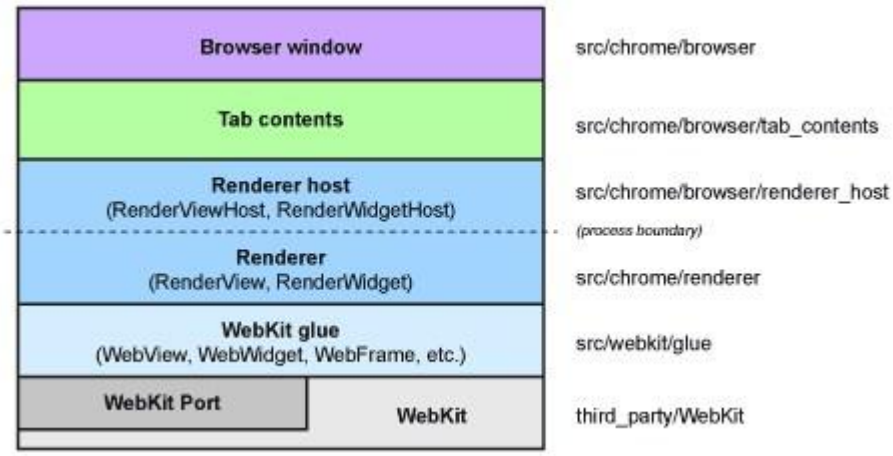
参考文档

http://www.chromium.org/developers/design-documents/plugin-architecture

Content API 和 CEF3

概述

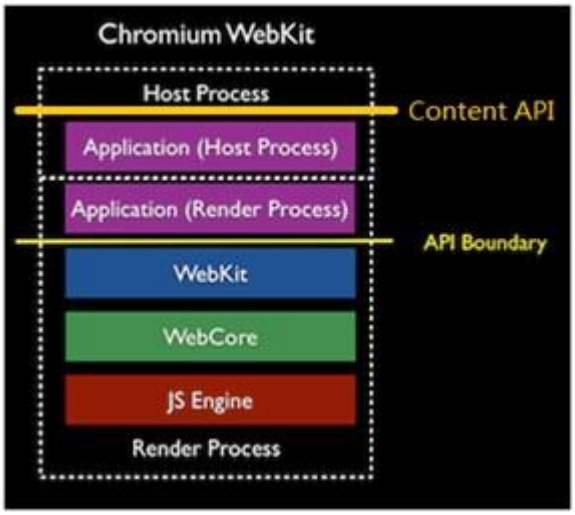
相信你一定看过下面这张图（没看过的话去上官网阅读一下“how chromium displays web pages”）。



这是一幅介绍页面如果被渲染和显示的概括性的层次结构图。Renderer 进程和 Browser 进程通过 IPC 来交换信息，具体的设施就是 RendererHost 和 Renderer 等相关类，其作用是把网页的内容（content）渲染成 Tab 的显示内容。一个 Tab 可能会包含多个页面的内容，因而它会管理 Tab 中的多个页面内容。Tab contents 之上就是浏览器，Tab contents 会把内容绘制在 browser 窗口的一个标签中。

Chromium 把 RendererHost 及其以下部分称为 Content，同时包括还有很多对 HTML5 功能实现的支持，contentAPI 基于此两部分，包装成为一套公开的接口。Tab contents 及以上部分称为 Chrome（chrome 的原意即是包装在网页内容之上的框）。浏览器中相关的功能仅仅在 content API 之上才有，而不存在于 content API 中。

上面的这个架构看起来没什么问题，但是，这对希望把 chromium 渲染网页的功能包装成接口，被内嵌到任何其他应用程序来说，就有着明显地不足。第一，在 WebKit chromium 移植中提供了一套公开的接口，但是它在 renderer 进程，没有很多 chromium 的提供的高级功能；第二，Chromium 在 browser 进程中没有提供任何公开的接口，也就是说，如果基于 RendererHost 等开发一套嵌入式的接口，代价将是非常巨大的，因为这些内部使用的类及其接口是经常变动的（注意，确实变化的非常快）。



上图介绍的是 WebKit 的 chromium 移植接口和 Content API 在整个栈中所在的层次，“API boundary”是原来的 WebKit 的 chromium 移植所提供的公开接口，“Content API”表示的是新的 content API 所在的层次。层次上的向上移动带来了使用者的稳定性，HTML5 功能的支持，GPU 硬件加速功能，沙箱模型等的支持。这里稍微不同的地方是浏览器的功能在 Content API 之上，所以图片中的 Application 应该去除掉这一部分。图片来源于 WebKit 官网，略有修改。

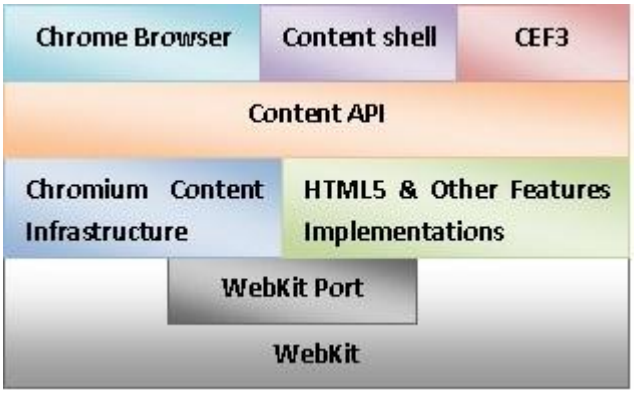
下面我们来看一下 chromium 官网上所宣称引入 Content API 的两条原因：

1） 让 Chrome 的开发者们摆脱 content 内部的复杂工作原理和机制； 2） 给 content 和 chrome 划分清楚的界限来帮助开发者或者嵌入式的使用者们。 在了解了相关背景之后，上面的解释就相当简单明了了。

为了方便测试，chromium 中有一个基于 content api 的简单测试程序 content shell。它非常的简单，仅仅是一个壳，调用了 content API 并实现了部分必需的回调接口，可以用来测试和其他一些简单的功能。 CEF 全称 chromium embeddedframework，其目的是提供一套嵌入式的接口，最初的版本是基于早期的 RendererHost 和 chrome 浏览器的很多内部接口开发而来的，在新的 CEF3 中，其主要依赖于公开的 Content API 来实现的。

为了清晰地了解它们之间的的关系，下图描述了 WebKit， content API，浏览器，content shell 和 CEF3 的层次关系。Chrome 浏览器，content shell 和 CEF3 三者都是基于 content API 开发的，它们只是有不同的实现，服务于不同的应用场景而已。

后面的章节将重点介绍 content API 和 CEF3，因 content shell 比较简单，故略过。



Content API

Content API 不仅提供了公开和稳定的接口，而且它从诞生以来一个重要的目标就是要支持所有的 HTML5 功能和 GPU 硬件加速功能，这可以让它的使用者们不需要很多的工作即可以得到好的 HTML5 支持和硬件加速机制。同时，借助于现有的多进程架构，一些 chromium 中的新功能例如沙箱模型等也在其中得到了支持。 下面详细介绍一下 Content API 包含哪些部分。 Content API 的相关的接口定义文件均在 content/public 目录下，按照功能分成六个部分：每个部分的接口一般也可以分成两类，第一类是嵌入者（embedder，这里可以是 Chrome 浏览器，CEF3 和 content shell）调用的接口，另一类是嵌入者实现的回调接口，被 content API 的内部实现所调用，例如参与实现的逻辑，事件的监听等。 1） App 这部分主要是跟应用程序或者进程的创建和初始化相关。 第一类，主要包括创建进程的初始化函数，content 的初始化和关闭； 第二类，主要是实现回调函数，告诉嵌入者启动完成，进程启动，进程推出，沙盒模型初始化开始和结束等等。 2） Browser 第一类包括，对一些 HTML5 功能和其他一些高级功能实现的参与，例如 resource，sensor，notification， speech recognition， web worker， download， web intents，等等; 第二类包括 ContentBrowserClient，主要是实现部分逻辑，被 Browser 端（或者进程）调用，还有就是一些事件的回调函数. 3） Common: 主要定义一些公共的接口，被 render 和 browser 共享，例如一些进程相关，参数，gpu 相关等等 4） Plugin: 仅有一个接口告诉嵌入者 plugin 进程被创建了 5） Render 第一类包含获取 RenderThread 的消息循环，注册 v8 extension，计算 JavaScript 表达式等等 第二类包括 ContentRendererClient，主要是实现部分逻辑，被 Browser 端（或者进程）调用，还有就是一些事件的回调函数 6） Utility: 工具类接口，主要包括让嵌入者参与 content API 中的线程创建和消息的过滤。

CEF 和 CEF3

早在 content API 出现之前，CEF 便已出现，其目的是提供嵌入式的框架，可以让渲染网页的功能方便地嵌入到应用程序之中。CEF 依赖于 chromium 浏览器，所以 chromium 对 HTML5 的支持和性能上的优势，都得以继续在 CEF 中体现出来。但是，根据实际测试的结果来看，情况可能并非如此。首先，其对 GPU 硬件加速的支持不是很好，这时因为它会把 GPU 内存读回到 CPU 内存，速度非常慢；再次，因为基于 chromium 的内部结构，而它们经常变化，所以 CEF 经常需要发生变化，这对维护来说是件很头痛的事。 得益于 content API 的出现，CEF 的作者也基于它开发了 CEF3。CEF3 在保持其提供的接口基本不变的情况下，借助 content API 的能力，其对 HTML5 和 GPU 硬件加速提供了较好的支持。它的核心变为调用 content API 的接口和实现 content API 的回调接口，来组织和包装成 CEF3 自己的接口以被其他开发者所使用。其好处是，CEF3 的接口相对比较简单，使用起来方便，同时不需要实现很多 content API 的回调接口，但是缺点就是，如果需要使用 content API 的很多功能，CEF3 的接口可能做不到，或者说只能通过直接调用 content API 接口来完成。 下面简单介绍一下 CEF3 的接口。

CefClient：回调管理类，包含 5 个接口用于创建其它的回调类的对象
CefLifeSpanHandler：回调类，用于控制 popup 对话框的创建和关闭等操作
CefLoadHandler：回调类，可以用来监听 frame 的加载开始，完成，错误等信息
CefRequestHandler：回调类，用于监听资源加载，重定向等信息
CefDisplayHandler：回调类，用于监听页面加载状态，地址变化，标题等得信息
CefGeolocationHandler：回调类，用于 CEF3 向嵌入者申请 geolocation 的权限
CefApp：与进程，命令行参数，代理，资源管理相关的回调类，用于让 CEF3 的调用者们定制自己的逻辑
CefBrowser：renderer 进程中执行浏览相关的类，例如前进，后退等
CefBrowserHost：browser 进程中的执行浏览相关的类，其会把请求发送给 CefBrowser
CefFrame：表示的是页面中的一个 Frame，可以加载特定 url，在该运行环境下执行 JavaScript 代码等得。
V8：CEF3 提供支持 V8extension 的接口，但是这有两个限制，第一，v8 extension 仅在 Renderer 进程使用；第二，仅在沙箱模型关闭时使用。

Content API 和 WebKit2

Content API 和 WebKit2 的接口都是基于跨进程的结构的，按我个人的理解，它们是互相影响而又互相吸收对方的优点，它们具有相同点，又有不同点：

1） Content API 提供的是一套基于跨进程的架构模型（当然它也可以不是多进程的），这有利于 browser（UI）进程的稳定，这同样对 WebKit2 适用； 2） Content API 不仅如此，它里面包含 chromium 对于 HTML5，GPU 硬件加速，和沙箱模型等功能的支持，而 WebKit2 可能并非如此； 3） Content API 是 chromium 提出的接口，而 WebKit2 是一个更加开放和公开的接口，被众多的 WebKit 移植所使用。

源文件目录

```
Content/public/app ,Content/public/browser, Content/public/render, Content/public/common,
Content/public/plugin, Content/public/utility
        ContentAPI 所包含的相关的目录，里面包含所有的接口
```

参考文献

http://www.chromium.org/developers/content-module/content-api
http://code.google.com/p/chromiumembedded/
http://trac.webkit.org/wiki/WebKit2

Chromium 移动版(Chromium for Mobile: Android & iOS)

概述

Chromium 的诞生于桌面（desktop)系统之上的，这其中包括 Windows， Linux 和 Mac。这奠定了 Google Chrome 的地位，市场份额不断上升。随着移动操作系统的流行，Chromium 也迫切需要拓展其在移动设备上的市场，所以 Chromium 的 Android 版和 iOS 版应运而生了，这两个背后有不同的故事。

本章讨论一些有关移动版的有趣事情，这里面没有特别细节的东西，主要是一些技术方向性方面的描述。

移动版

chromium 的 iOS 版和 Android 是为两个流行的移动操作系统设计的，其区别与传统的桌面版，特别是 UI 方面进行了较大的重新设计。两者从外观上看颇为相似，但是其内部的渲染引擎的差别非常的大，原因在于 iOS 对应用程序的控制造成的，而非两个操作系统的差异性。

Chromium 的 iOS 版

iOS 是最受欢迎的移动操作系统之一，其设备数量以亿计，所以能在 iOS 上成功就特别重要了。显而易见，这有助于浏览器的市场份额。苹果对 iOS 上的应用程序是严格审核的，浏览器作为非常重要的应用程序，当然也不例外。

不幸地是，苹果不允许浏览器有自己的内核（WebKit 或者其他），理由是安全性。所以，chromium 的 iOS 版始终没法通过审核，其中的争论也是非常激烈。抛开这些争论，Chromium 的 iOS 版已经上线了，不过这是一个没有 chromium 内核的浏览器，其基于的是 iOS 提供的 UIWebView（一个嵌入式的网络渲染模块，该模块提供对网页渲染和 HTML5 的支持），加上 Chromium 风格的 UI。

有趣的是这个 UIWebView 跟 Safari 的所使用的支持 HTML5 的模块是不一样的。相对于 Safari 的浏览器内核，其有两个明显的缺点：

Safari 使用 Nitro JavaScript 引擎，而 UIWebView 使用 WebKit 缺省 JavaScript 引擎，性能上要差很多；
Safari 使用单独的线程而不是主线程来做渲染工作，悲剧的是 UIWebView 渲染是在主线程来完成，这会阻碍事件的响应。

好吧，有了这两点，其他基于 UIWebView 的浏览器被 Safari 抛开一大截了，而且这些浏览器只能使用 UIWebView，而不能有自己的浏览器内核。

Chromium 的 Android 版

我们知道 Chromium 是一个开源的项目，但是，chromium 的 Android 版从开始之初就是一个闭源的项目，其基于一个稳定的 chromium 版本，在 Google 的内部开发，直到发布 Google Chrome 的 Android 1.0 版。 跟 iOS 不一样，Android 版从出生就是亲儿子，其跟 Android 关系非常紧密，怎么说呢，Android 上的应用程序一般都是基于 Android SDK 和 Android NDK 的 API 来开发的，Chromium 可不是这样，它其实是使用了 Linux 平台的 API 和很多第三方的库，可以说是一个有“特权”的应用程序。 目前 Google Chrome 的 Android 版是基于 chromium18 开发的，虽然是闭源的，但是 Google 开放了 c++端的代码（参考文献第一条可以下载这部分代码，是一个基于 chromium18 的 zip 包），而 Java 端的代码开放的很少，与 UI 和浏览器功能相关的则基本没有开放。Google Chrome 的 Android 版本中也加入了一些新的特性和架构，例如 in-process-gpu， threaded compositing，基于 Android UID isolate 的沙箱技术等等，对于 HTML5 的支持也非常的到位，各种主流的 HTML5 功能基本都支持，除了 WebGL，这都会在以后的章节作一一介绍。

无须担心的是 Google 的开放性。Google 正在逐步释放出 Android 版的源代码到 Chromium 的源代码库(upstream trunk)中，目前对开放者来说，很多功能逐步加入，所以很多单元测试已经被打开了(实际上本人也参与贡献了一些 patch)。对用户来说，主要的进展是能够通过一个简单的浏览器（content shell）渲染网页，虽然有很多的 bug，但是基本的渲染功能是可以工作的，而且支持 HTML5 的 canvas2D 和 WebGL 等功能（当然，质量还是个问题）。这部分做过比较深的研究，也贡献给 upstream 一些代码，相对也比较熟悉，会在后面的章节中作专门的介绍。

参考文献

https://developers.google.com/chrome/mobile/docs/faq
http://peter.sh/2012/02/bringing-google-chrome-to-android/

Chromium for Android

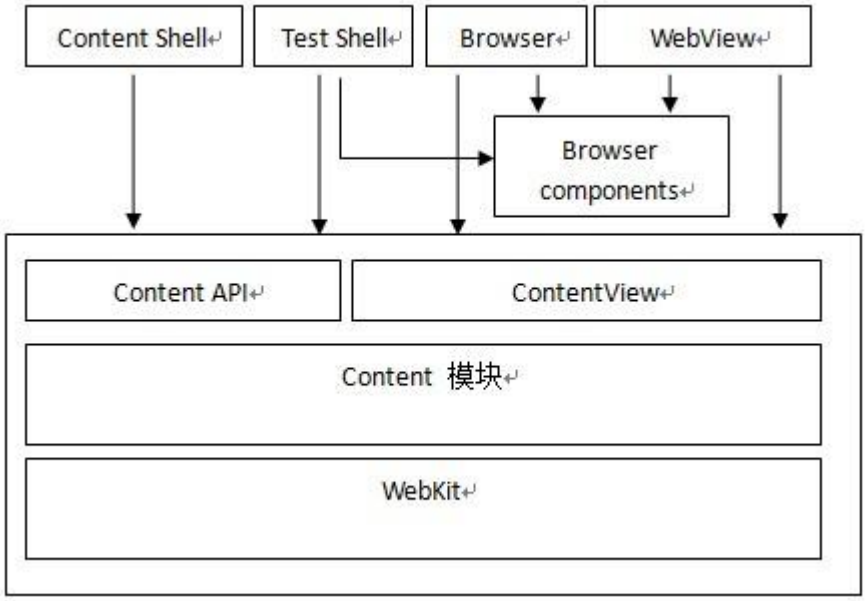
概述

在前面的 Chromium 的移动版章节中，我们对 Chromium 的 Android 版作过简单介绍，本章则进一步阐述其中的细节。

2012 年，Google 发布了第一版的 ChromeFor Android。该产品受到了巨大的关注，但是同桌面版的 Chrome 浏览器不同地是，它的源代码并没有被开源出来，而且也不是跟随最新的 Chromium Trunk 代码，而是基于很早一个 chromium 稳定版－18。

在刚开始的时候，Google 每隔一段时间会放出一个压缩包，里面包含 Chrome For Android 所有的 C++代码，而 Java 相关的代码则没有放出来（其实放出了一部分代码，但是 UI 部分则没有），因而你没有办法编译出来一个类似桌面版 Chromium 的浏览器。不过，你可以利用 Google 释放发出的代码来编译一个 C++动态库（情况有些变换，现在你可以使用最新 trunk 上面的代码来编译一个这样的库了），来替换发布的 APK 包中相应的动态库，这样你就能生成一个自己定制的浏览器 APK。如果你不是很在乎 UI 层和浏览器的丰富功能，还有一种方法可以让你编译一个简单的浏览器，那就是利用 chromium 代码直接编译出两个 APK －Content Shell 和 Test Shell，它们界面非常简单，但是它们都可以渲染网页并拥有基本上所有 Chrome For Android 的 HTML5 功能。

听起来令人迷惑，不是吗？好了，让我们通过下面的图来理解它们吧。



上图的层次结构非常清楚。从图中可以看出，它们都是基于 Content 模块/Content API/ContentView，而这是渲染网页和支持 HTML5 功能的核心，所以很大程度上来讲，在 HTML5 的支持上，它们不仅很类似，而且代码都是相同的，因而分析 Content Shell 或者 Test Shell 可以帮助你理解 Chrome for Android 的网页渲染和 HTML5 功能。

在上图的最上面一层，除了 Browser 是看不到它的代码以外，其它的你都可以在 chromium 的开源社区中找到它们的源代码并可以尝试编译它们。至于如何编译它们，请见参考文献 1，支持编译 Android 的 ARM 版和 IA 版。

Content Shell 和 TestShell

这两者都是在 content 模块上构建的简单浏览器，怎么个简单法呢，简单到界面就是一个 URL 输入框附加一个页面显示框，功能就是显示个网页，至于其它功能，抱歉，它们基本上都没有提供，而且这也不是它们的目标。

从界面上来看，目前两者没有什么区别，但就其目的和功能上来说，我的理解它们还是很不一样的：

Content Shell 是直接在 content 模块之上的，其主要目的是测试 content API/ContentView 等模块的正确性； Test Shell 虽然也依赖 content 模块，但是它还依赖于 chrome 浏览器的很多基础设施和组件（android 平台的 chrome 内核），更像是测试它们的一个简单浏览器，因为 chrome for android 的 UI 部分不开源，所以 Test Shell 应该是最接近它的一个工具，可以帮助理解 chrome for android 的架构和内部原理。

Android 的 WebView

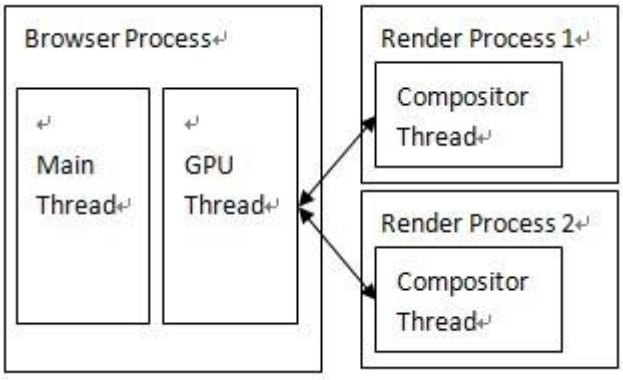
使用 Chromium 的内核来实现 Android 的 WebView 机制，这个后面单独介绍。

同桌面版的不同之处

Chromium 的 Android 版同传统的桌面版有很多不同的地方，具体包括以下几个方面： 首先是工作模式方面，Chromium 的桌面版一直紧跟 trunk，有着不同的发布渠道，包括 dev，beta，stable 等。而 Android 版则是基于某个稳定版在内部开发，然后不定期的释放出一些代码，其并没有同最新的 Chromium 同步（最近才开始保持同步）。

其次是开源方面，Chromium 桌面版都是开源的，但是目前来看，Android 版只是开源了部分，就是 Content 模块和以下，浏览器的一些功能模块，而 UI 部分则没有而且在可以预见的未来也不会开源。

再次是架构方面，首先是 GPU 进程变成为 browser 进程的一个线程，其次是线程化合成已经是 Android 版的一个默认功能。见下图所示，你可以跟“多进程模型”中的进程模型图进行对比一下。



然后是安全机制，对于 Android 平台来说，沙箱模型有些不一样。在 Android4.1 之后，引入了进程的 isolated UID 机制，这就为 Chromium 的 renderer 进程提供独立开来的可能性。每个 Renderer 进程都有唯一的 UID，因而它们之间和 Browser 进程互相不能访问，直接被隔离开来。

之后是渲染部分，特别是 GPU 硬件加速部分也有很多不同，以后会一一介绍他们。 最后是功能方面，同桌面版比较，一些功能如 HTML5，插件，NativeClient，extension 等目前都不支持。

源文件目录

```
content/public/android/
        content 模块中的 android 接口，例如 contentview 等
content/shell/android
        Contentshell 相关类
chrome/android
        TestShell 的相关类
android_webview/
```

该目录包含实现 Android 系统当前 WebView 接口所需的内部实现类，基于 Chrome 的内核

参考资料

https://code.google.com/p/chromium/wiki/AndroidBuildInstructions

基于 Chromium 内核的 Android WebView

概述

熟悉 Android 系统和 HTML 编程的人可能都听说过 Android 提供的一个重要类 android.webkit.WebView，它继承于 View 类，这是它同其它很多控件的相似之处。不同之处在于，它能够用来渲染网页。当前，WebView 的实现是基于现有的缺省 WebKit 内核（Android 缺省浏览器是基于 WebView 构建），它不同于 chromium 所使用的 WebKit 内核，虽然它们都叫 WebKit。

目前，它被广泛的用在众多的 Android 应用程序中，通常我们称之为混合应用程序(Hybridapplications，意思是结合了 HTML5 和传统的应用程序特征)。遗憾的是，它对 HTML5 的支持很差，而且也没有新的功能被加入进来，同时，Chromium 的 Android 版正在积极向前发展，更多的针对该平台的 HTML5 能力和优化逐步被实现和采用，那么是否也可以使用 Chrome 的内核来实现该 WebView 呢？答案当然是肯定的。

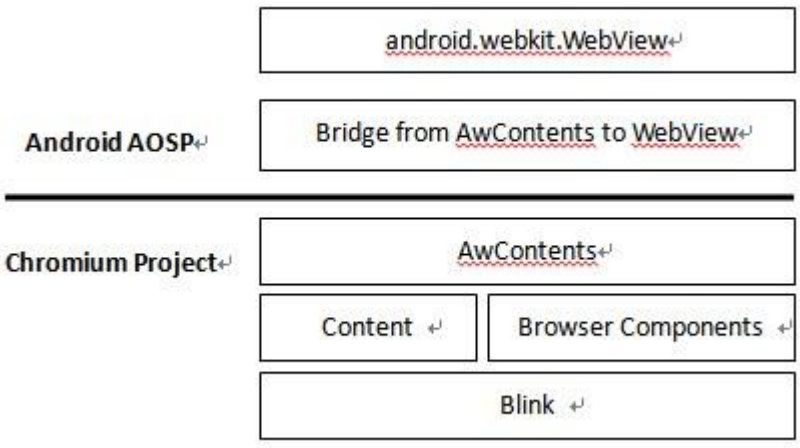
目前，该项目已经启动，其核心思想保持 WebView 的 API 兼容性，也就是说只是将内部的实现从当前缺省 WebKit 内核变了 Chromium 的内核，但是原有的 WebViewAPI 保持不变，这样对于 WebView 的用户来说，不需要做任何改变，便可以使用上功能更多性能更好的渲染内核了。

你可以通过编译目标“android_webview_apk”来尝试一下它的功能，这也是基于 WebView 的一个简单实例应用程序，类比一下就如同 content 模块和 ContentShell 的关系。

结构

在今后的 Android 某个版本之后，毫无疑问，基于 Chromium 内核的 WebView 将会取代现有 WebView 的实现。初看一下，目前的代码结构如下图所示，在 ContentAPI 之上，Chromium 的 WebView 实现封装了一个新的类 AwContents，该类主要基于 ContentViewCore 类的实现，不同的是，AwContents 需要基于一个原来存在于“chrome/”目录下的模块（图中的 BrowserComponents），但是 AwContents 不应该依赖该目录，所以，将 chrome 中的一些所谓的浏览器模块化是 Chromium 的一个方向，目前，一些模块以及从 chrome 中抽取出来了，参见“components/”,具体介绍见参考资料 2。

AwContents 提供的不是 WebView 的 API，所以，需要一层桥接部分，将 AwContents 桥接到 WebView，这就是图中的桥接模块，该模块位于 Android 源代码中，目前没有开源，应该稍后会开放出来。



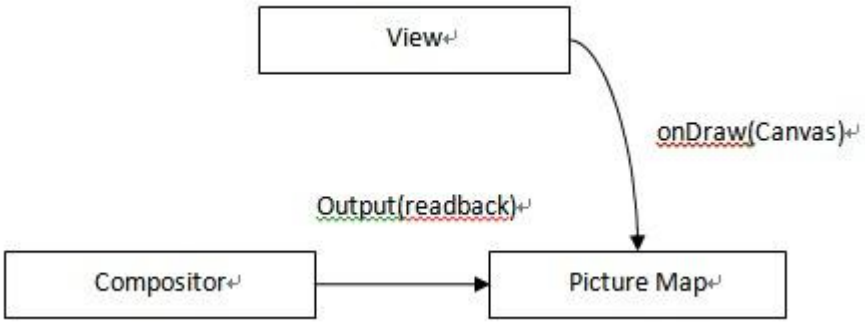
今后，WebView 有什么变化，将会在该篇文章中持续更新，欢迎大家关注。

新变化

WebView 同样也是基于 ContentAPI(web contents, ContentViewCore 等)，在这点上来说，它同 Content Shell 和 Chromium 浏览器没有大的不同，区别在于它们对很多 Delegate 类的实现不同，这是 ContentAPI 用于让使用者参与内部逻辑和实现的过程。具体来说，它主要有以下两个方面的不同：

1.渲染机制

因为 WebView 提供的是一个 View 控件，那么 View 控件的容器可能接受储存在 CPU 中的结构（如 bitmap），也可能是储存在 GPU 内存中的结构（如 surface），所以它需要提供两种不同的输出结果。那么是否意味了 WebView 提供软件渲染和 GPU 硬件渲染两种方式呢？答案是否定的。 目前，Chromiumfor Android 不提供网页软件渲染，只有 GPU 硬件渲染一种方式，其渲染的结果由合成器生成。那么，如何生成 bitmap 呢？目前答案是通过 OpenGL ES 的回读方式。当合成器没合成一帧时候，AwContents 将该帧保存在一个 PictureMap 中，当 UI 需要重新绘制时候，便把当前的 Picture 取出，绘制在当前 View 的 canvas 中，如下图所示，所以它其实什么也不能做，不接受事件输入，不能滚动等等，只能看到一个渲染结果。同时，这样做会导致及其低效的性能。当然，这只是一个临时方案。



当前，对于 Compositor 的结果输出到给定 View 的 GPU 内存这种方式，AwContents 还不支持，工作正在进行中。 为什么？因为 Chromium 即将引入一种新的合成器 UberCompositor++，该合成器支持输出到 GPU 和 CPU 内存两种方式，今后将对其作介绍。

2. 进程

目前 WebView 只支持单进程方式，未来不排除支持多进程方式。单进程意味了没有办法使用 Android 的 isolated UID 机制，因此，某种程度上来讲，安全性降低了，而且页面的渲染崩溃会导致使用 WebView 的应用程序崩溃。

源文件目录

```
android_webview/
        该目录包含了支持 webview 所需的所有相关类
```

参考资料

http://developer.android.com/reference/android/webkit/WebView.html
http://www.chromium.org/developers/design-documents/browser-components