Webkit 分析报告 II

侯　炯

二零零九年二月十日

导言

相信大家读完《WebKit 分析报告》对浏览器的现状和发展有了一定的了解，并对 WebKit 有了初步印象，更能编译运行 webkit。这里为了更好的对 webkit 进行进行深入的了解，我们将对 WebKit 的框架和流程进一步分析，并对 WebKit 部分代码进行分析。

WebKit 框架比较简单，但内容很复杂，而其在不断的修改完善。并且很多技术有待讨论，新的技术也在不断引进。希望该文章对你的有用。欢迎来信讨论（hou_jiong@163.com）。

目　录

I.WebKit 介绍

一.WebKit 是什么？

WebKit 的前身是 KDE 小组的 KHTML。Apple 将 KHTML 发扬光大，推出了装备 KHTML 改进型的 WebKit 引擎的浏览器 Safari，获得了非常好的反响。WebKit 引擎比 Gecko 引擎更受程序员欢迎的原因，除了其引擎的高效稳定，兼容性好外，其源码结构清晰，易于维护，是一个重要的原因。

现在浏览器的内核引擎，基本上是三分天下:

Trident: IE 以 Trident 作为内核引擎。

Gecko: Firefox 是基于 Gecko 开发。

WebKit: Safari, Google Chrome 基于 Webkit 开发。

WebKit 内核在手机上的应用十分广泛，例如 Google 的手机 Gphone、 Apple 的 iPhone, Nokia’s Series 60 browser 等所使用的 Browser 内核引擎，都是基于 WebKit。

二.WebKit 主要特点和目标

引擎

该项目的主要重点是内容部署在万维网上的，基于标准的技术，如 HTML ，CSS ， JavaScript 和 DOM 中。并能够嵌入 WebKit 在其他应用程序中，并用它作为一般用途的显示和交互引擎。

开源

WebKit 应继续自由使用的两个开源和专有应用。

高性能

维持和改善的速度和内存使用是一个重要的目标。随着网页内容越来越丰富，越来越复杂，作为网络浏览器上运行的更有限的设备，需要提升性能提升，加快浏览速度。

可移植性性

该 WebKit 项目力求解决各种需要。能把 WebKit 移植到各种各样的台式机，移动，嵌入式和其他平台。WebKit 将提供必要的基础设施做到这一点紧张平台集成，重用本地平台服务，并酌情提供友好嵌入的 API 。

兼容性

为用户浏览网页，兼容现有的网站是必不可少的。我们致力于维护和改善兼容现有的网络内容，有时甚至不惜牺牲标准。我们利用回归测试，以保持我们的兼容性收益。

遵守标准

WebKit 的目标是遵守有关的 Web 标准，并支持新的标准

安全

保护用户安全的行为是至关重要的。迅速修复安全问题，以保护用户和维护他们的信任。

专注

WebKit 是一个引擎，而不是浏览器。

WebKit 小组不打算开发或举办一个全功能的网络浏览器基于

WebKit 。而只是一个引擎， WebKit 小组专注于网站的内容，而不是全面的解决方案，需要每一个想象的技术。

三.WebKit 主要组成

WebKit 主要包括三个部分 WebCore、JavascriptCore 及 Ports 部分。WebKit 专注的核心部分主要是: 分析 Html,Javascript 的解析和布局渲染技术。分别在 WebCore/html， JavascriptCore 和 WebCore/rendering 里面。

II.WebKit 之 WebCore 介绍

一.WebCore 所包含的主要内容

1.目录结构

从源代码目录结构来看 WebCore 目录主要包括如下目录:

bindings

将 Dom Binding 给 JavascriptCore 方面的代码，同时包含依据 idl 接口描述文件，自动生成对应于 JavascriptCore 的 Binding 实现的脚本等内容;

bridge

主要包含NPPlugin方面的接口访问等内容;

css

主要包括与css方面相关的内容如解析、不同css规则的定义与实现、css Binding给JS的接口定义等内容;

dom

主要包括dom方面相关的内容如不同dom元素的定义与实现、dom Binding给JS的接口定义等内容;

html

关于 html 方面相关的内容，如不同 html 元素的定义与实现、HTMLTokenizer及HTMLParser等内容。

loader

主要包括装载资源如html页面、css、js及image等方面内容;

page

主要包括描述一个 Web 页面所涉及的内容如 page、frame、frameview、frametree、setting、history、chrome、chromeclient等内容;

rendering

主要包括如何使用样式，组织布局、显示html元素等方面内容;

plugins

主要包括浏览端如何实现NPPlugin方面的内容;

svg

主要包括与svg方面相关的内容;

xml

主要包括与 xml 方面相关的内容如 xml parser、XPath、XSLT 等;

platform

主要包括与不同平台或外部库相关的内容如graphics(图形输出方面)、network(网络处理方面)、image-decoders(解析不同图片格式方面)等。

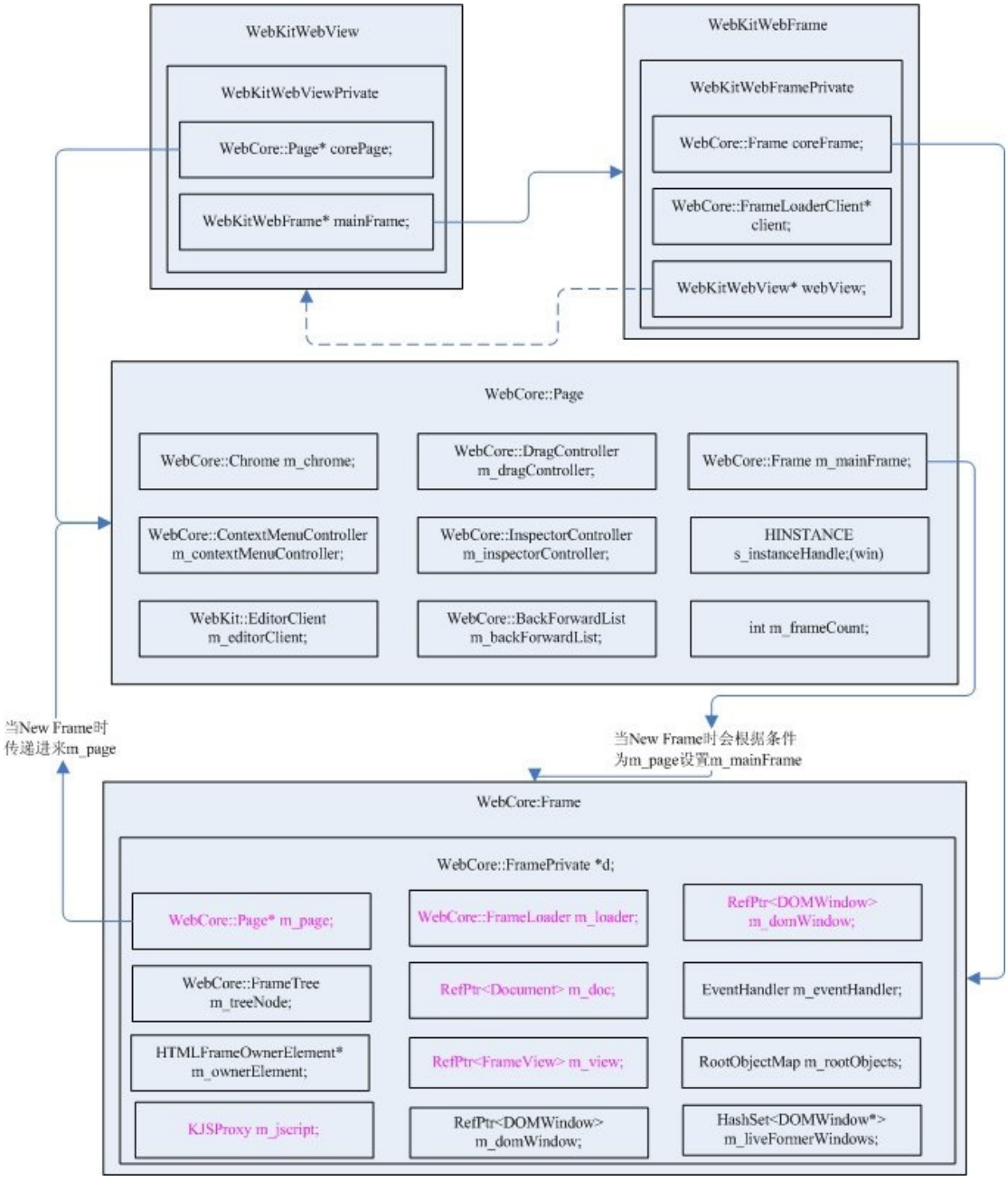
2.主要数据结构

为了更加简单有效的描述浏览网页的内容及过程，WebKit 为了明显区分不同方面的内容，采取了不同的 namespace 如 webcore、javascriptcore、webkit 等，webcore 方面的主要数据结构有:

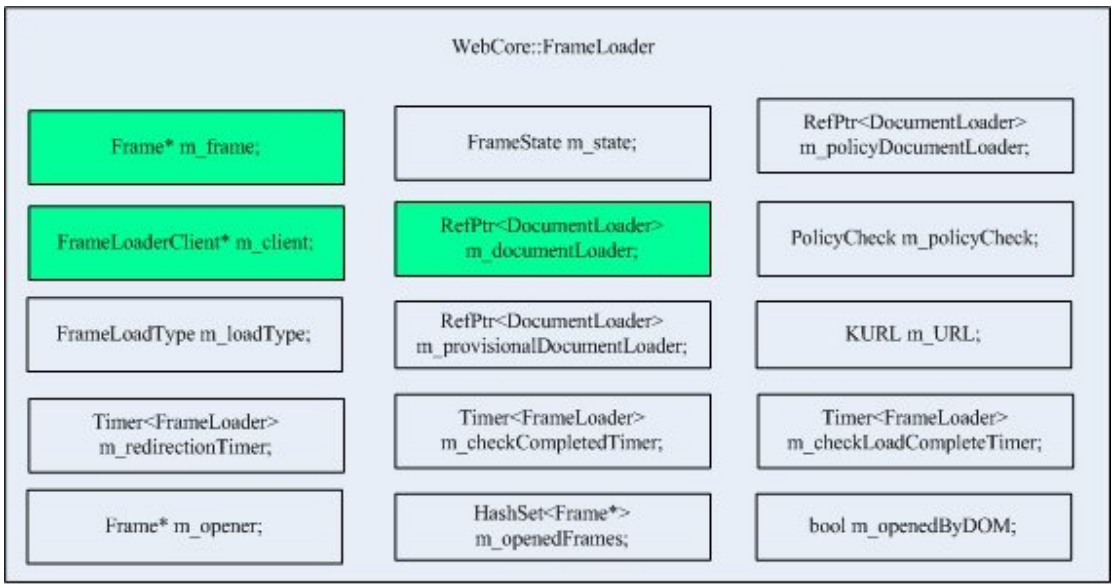
webcore::page 、 webcore::frame 、 webcore::FrameLoader 、 webcore::FrameView 、 Document 、 DOMWindow 、 KJSProxy 、 DocumentLoader 、 ResourceHandle 、 ResourceRequest 、 ResouceResponse、MainResourceLoader、RenderObject、RenderView 等;

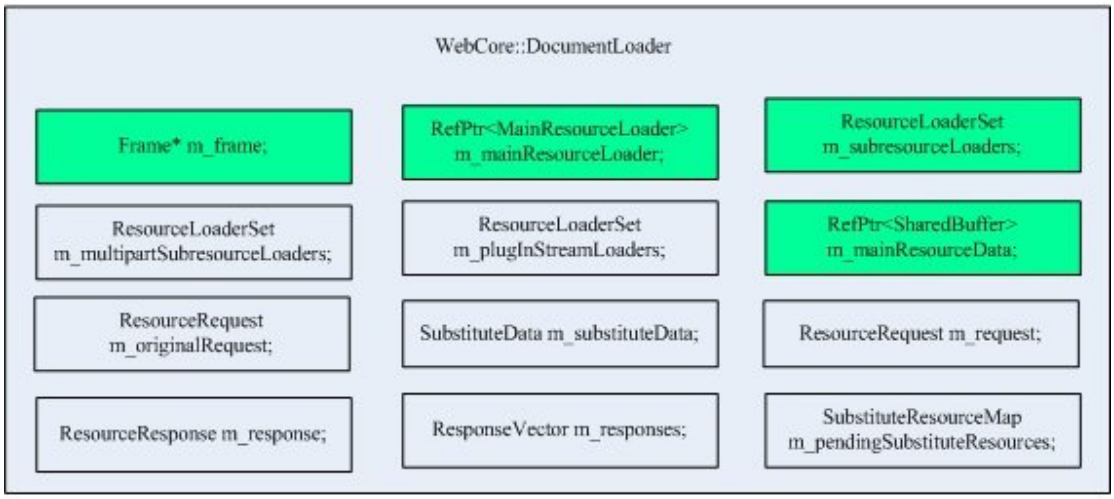
主要数据结构描述如下:

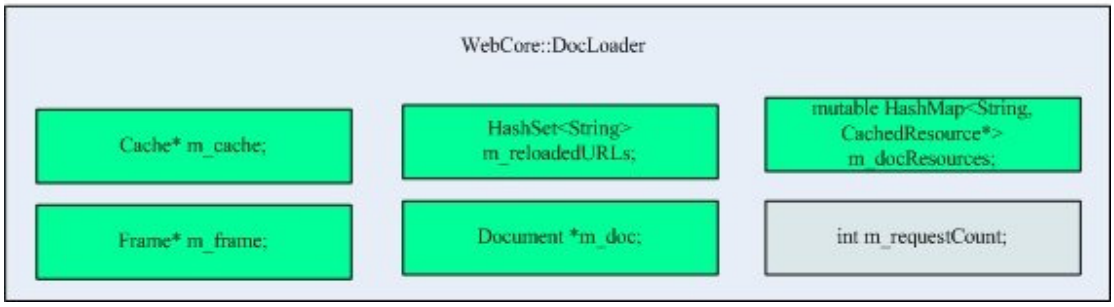
WebView 及 WebFrame 与 page、frame 之间的关系



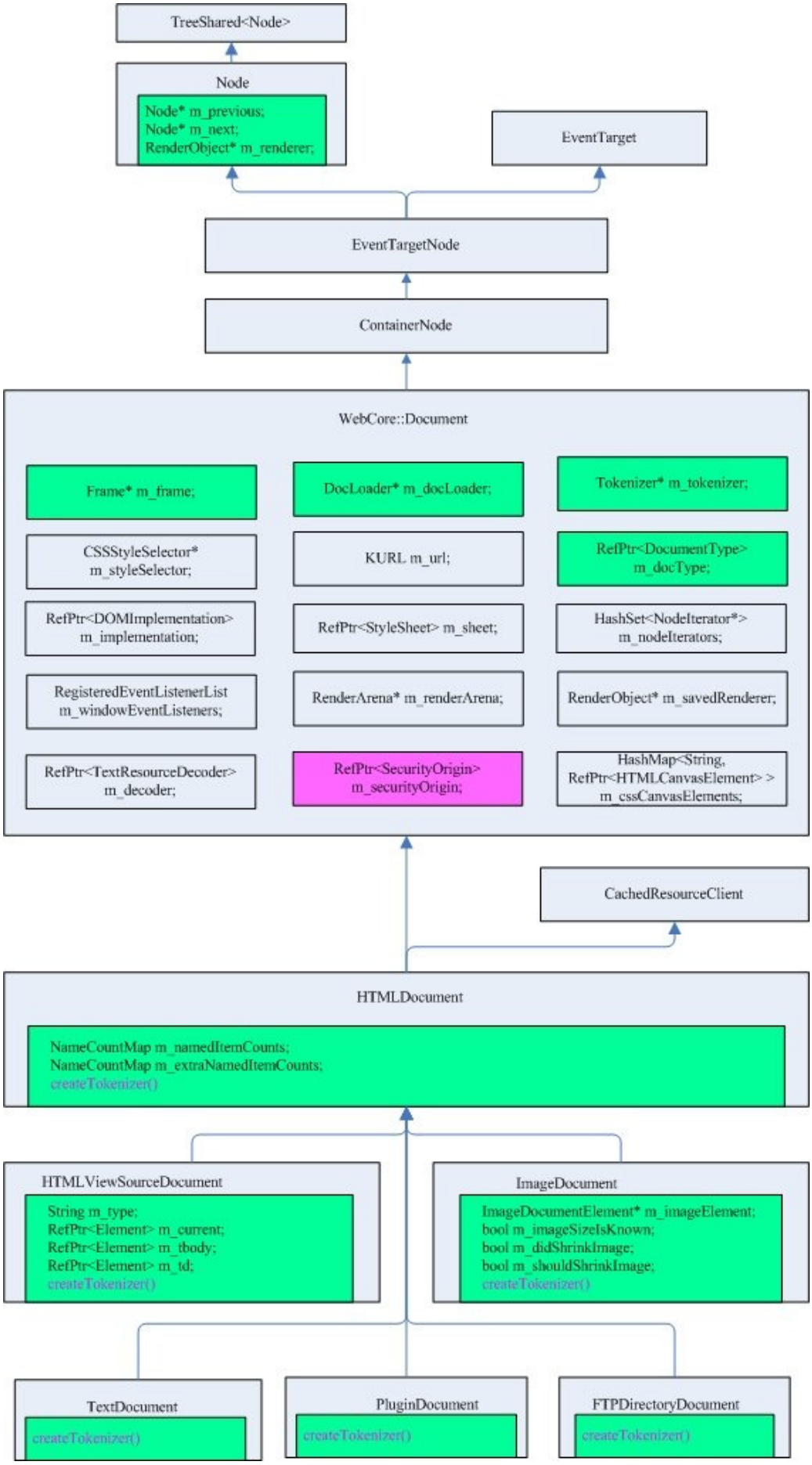
FrameLoader、DocumentLoader、DocLoader 类结构



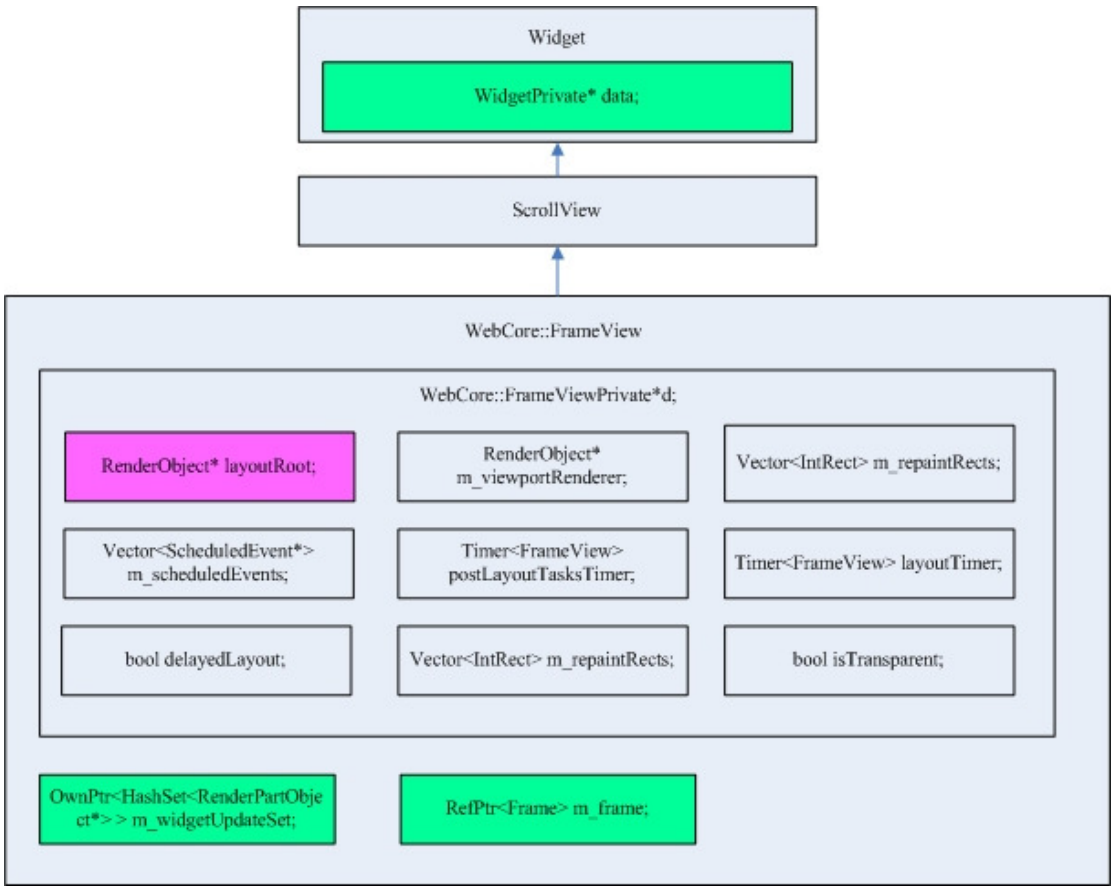




主要 Document 类结构


TreeShared<Node>
Node
Node* m_previous;
Node* m_next;
RenderObject* m_renderer;
EventTarget
EventTargetNode
ContainerNode
WebCore::Document
Frame* m_frame;
DocLoader* m_docLoader;
Tokenizer* m_tokenizer;
CSSStyleSelector* m_styleSelector;
KURL m_url;
RefPtr<DocumentType> m_docType;
RefPtr<DOMImplementation> m_implementation;
RefPtr<StyleSheet> m_sheet;
HashSet<NodeIterator*> m_nodeIterators;
RegisteredEventListenerList m_windowEventListeners;
RenderArena* m_renderArena;
RenderObject* m_savedRenderer;
RefPtr<TextResourceDecoder> m_decoder;
RefPtr<SecurityOrigin> m_securityOrigin;
HashMap<String, RefPtr<HTMLCanvasElement> > m_cssCanvasElements;
CachedResourceClient
HTMLDocument
NameCountMap m_namedItemCounts;
NameCountMap m_extraNamedItemCounts;
createTokenizer()
HTMLViewSourceDocument
String m_type;
RefPtr<Element> m_current;
RefPtr<Element> m_tbody;
RefPtr<Element> m_td;
createTokenizer()
ImageDocument
ImageDocumentElement* m_imageElement;
bool m_imageSizeIsKnown;
bool m_didShrinkImage;
bool m_shouldShrinkImage;
createTokenizer()
TextDocument
createTokenizer()
PluginDocument
createTokenizer()
FTPDirectoryDocument
createTokenizer()

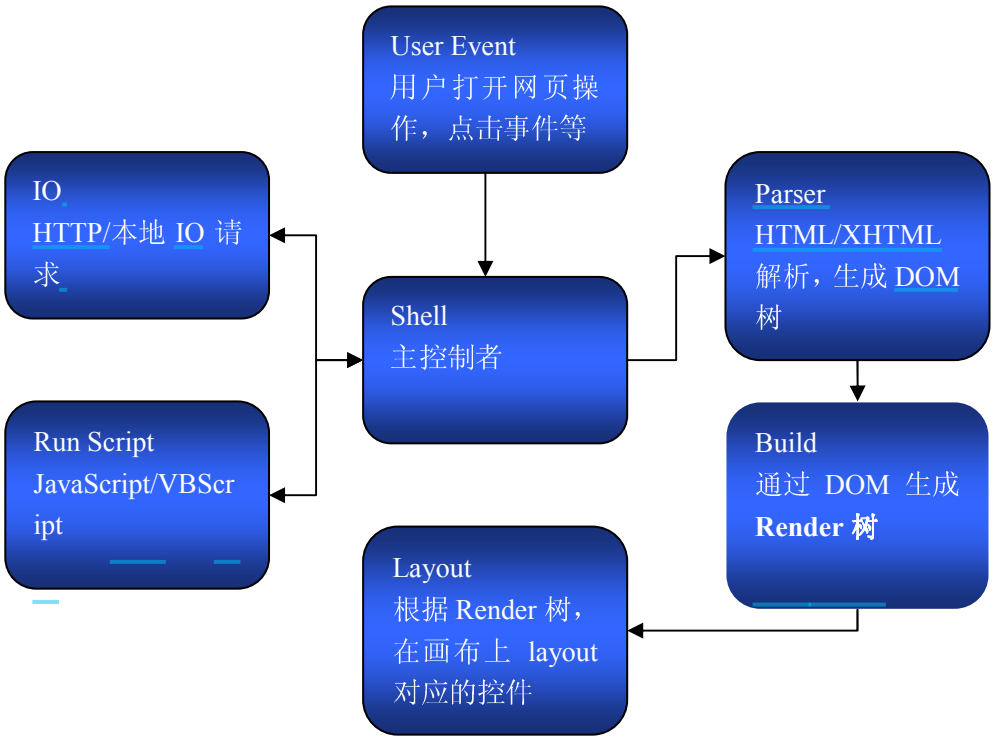
FrameView 类主要结构



总的说来，WebCore 包含了浏览器引擎的核心部分如处理 html、dom、css、svg、获取资源、渲染页面过程控制、回调/通知外壳程序以及与 Javascript 实现的 Binding 等等;

二.一个 Http 请求在 WebCore 中的主要流程

1.WebKit 工作流程



流程:

1. 用户向 Shell 发出页面请求后，页面的 URL 或本地文件名被发送到 Shell;

2. Shell 调用 IO 组件，把 URL 传达到 IO 组件;

3. IO 组件使用 HTTP 协议或再调用本地 IO 获取 HTML/XHTML 源数据，返回 Shell;

4. Shell 把 IO 返回的 HTML/XHTML source 提交 HTML/XHTML 分析器;

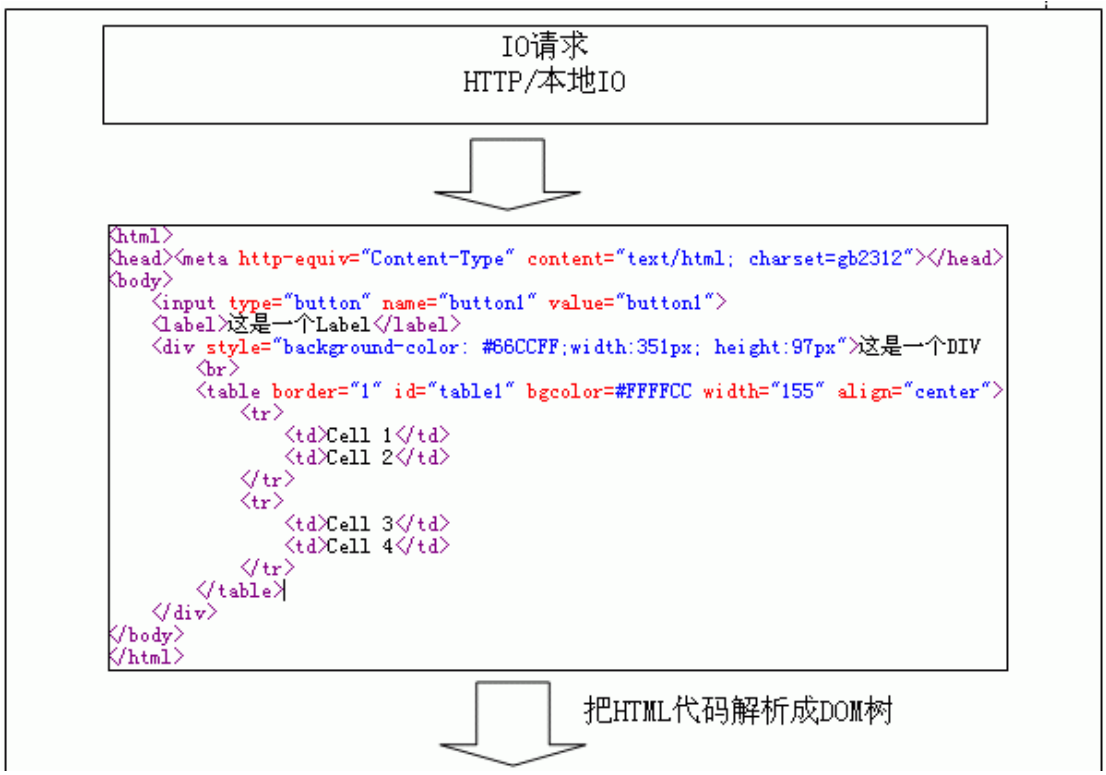
5. HTML/XHTML 分析器分析 HTML/XHTML 代码，构建一棵 DOM 树，树根为 HTMLDocument;

6. 通过 DOM 树，生成 Render 树.也许对于 Render 树大家不那么了解了，简单的说来，它是对 DOM 树更进一步的描述，其描述的内容主要与布局渲染等 CSS 相关属性如 left、 top、width、height、color、

font 等有关，因为不同的 DOM 树结点可能会有不同的布局渲染属性，甚至布局时会按照标准动态生成一些 匿名节点，所以为了更加方便的描述布局及渲染，WebKit 内核又生成一颗 Render 树来描述 DOM 树的布局渲染等特性，当然 DOM 树与 Render 树 不是一一对应，但可以相互关联。

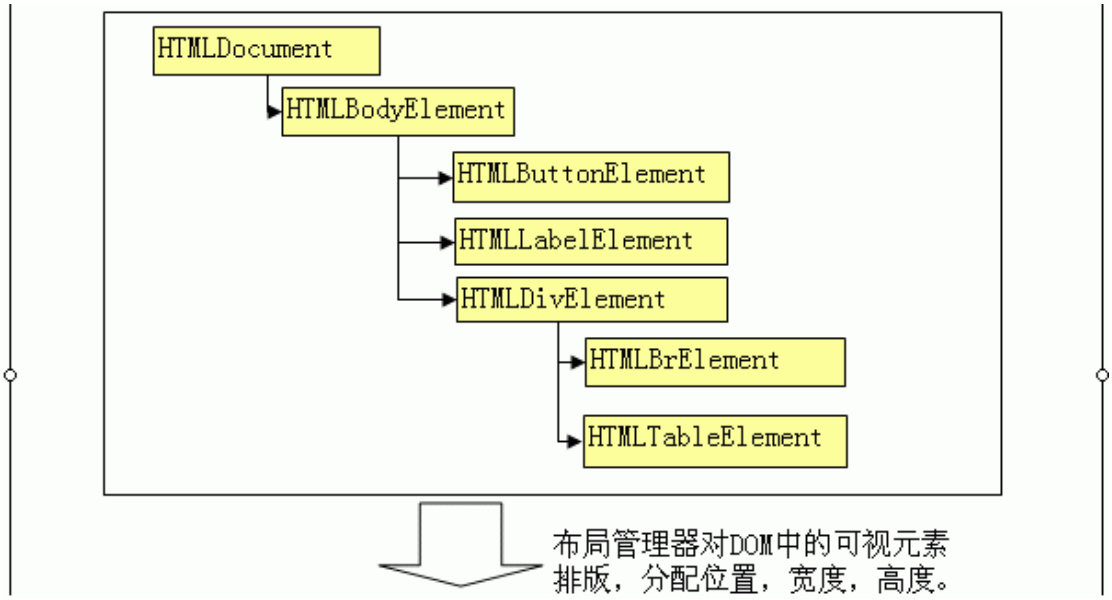
7.布局器布局现实出来。当布局管理器对可视化元素指派好位置，大小后，它就就知道了它的安身之处，也记住了它的大小，它必须要严格遵守布局管理器给它分配的位置，大小，不能擅自更改。既然知道了自己的位置，大小，剩下的就是控件根据自己的属性进行表现自己了，背景，外形等。

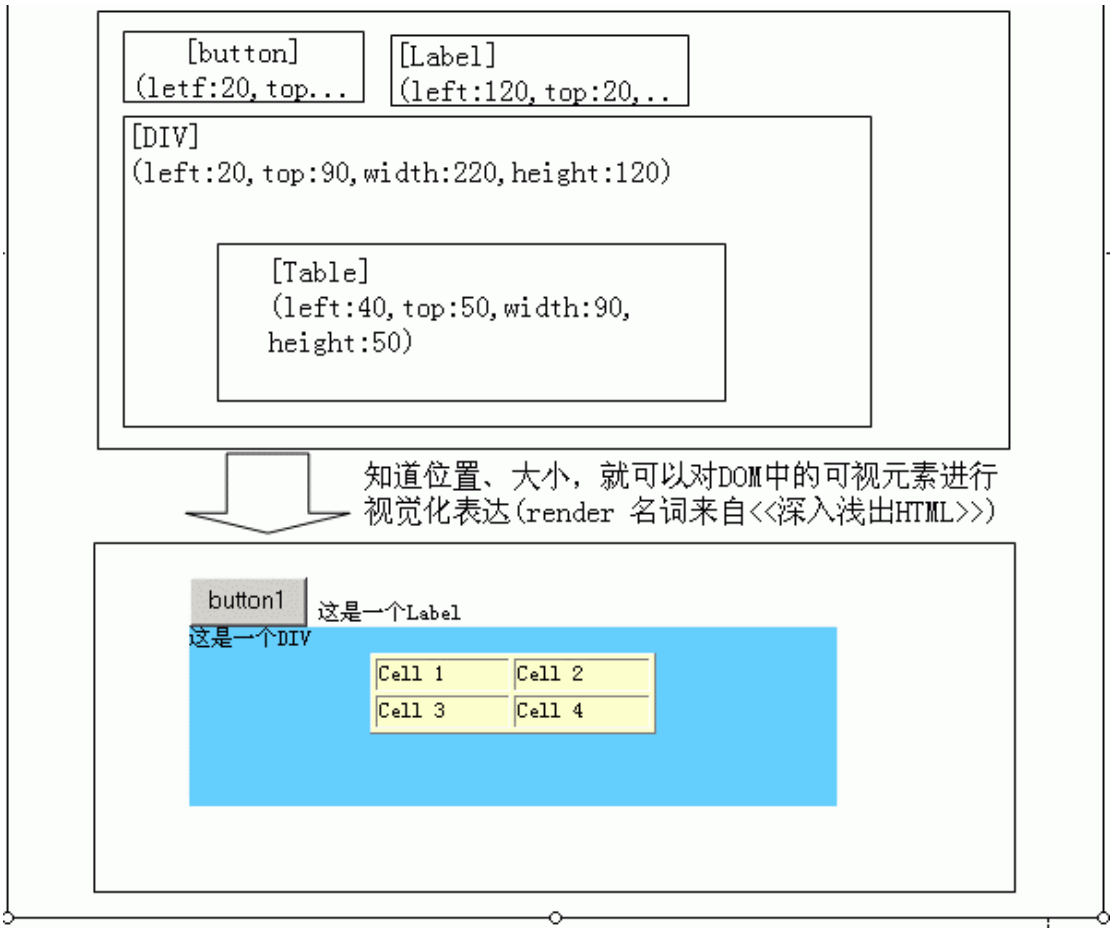
2.处理流程


IO请求
HTTP/本地IO
<html>
<head><meta http-equiv="Content-Type" content="text/html; charset=gb2312"></head>
<body>
<input type="button" name="button1" value="button1">
<label>这是一个Label</label>
这是一个DIV

<table border="1" id="table1" bgcolor=#FFFFCC width="155" align="center">
<tr>
<td>Cell 1</td>
<td>Cell 2</td>
</tr>
<tr>
<td>Cell 3</td>
<td>Cell 4</td>
</tr>
</table>

</body>
</html>
把HTML代码解析成DOM树


HTMLDocument
HTMLBodyElement
HTMLButtonElement
HTMLLabelElement
HTMLDivElement
HTMLBrElement
HTMLTableElement
布局管理器对DOM中的可视元素
排版，分配位置，宽度，高度。



流程:

通过向服务器发送请求

服务器通过请求，发给客户端 html 的内容

浏览器通过 W3C 规范，把接受到的内容解析成 DOM 树，在解析 DOM 树的同时会生成对应的 Render 树。

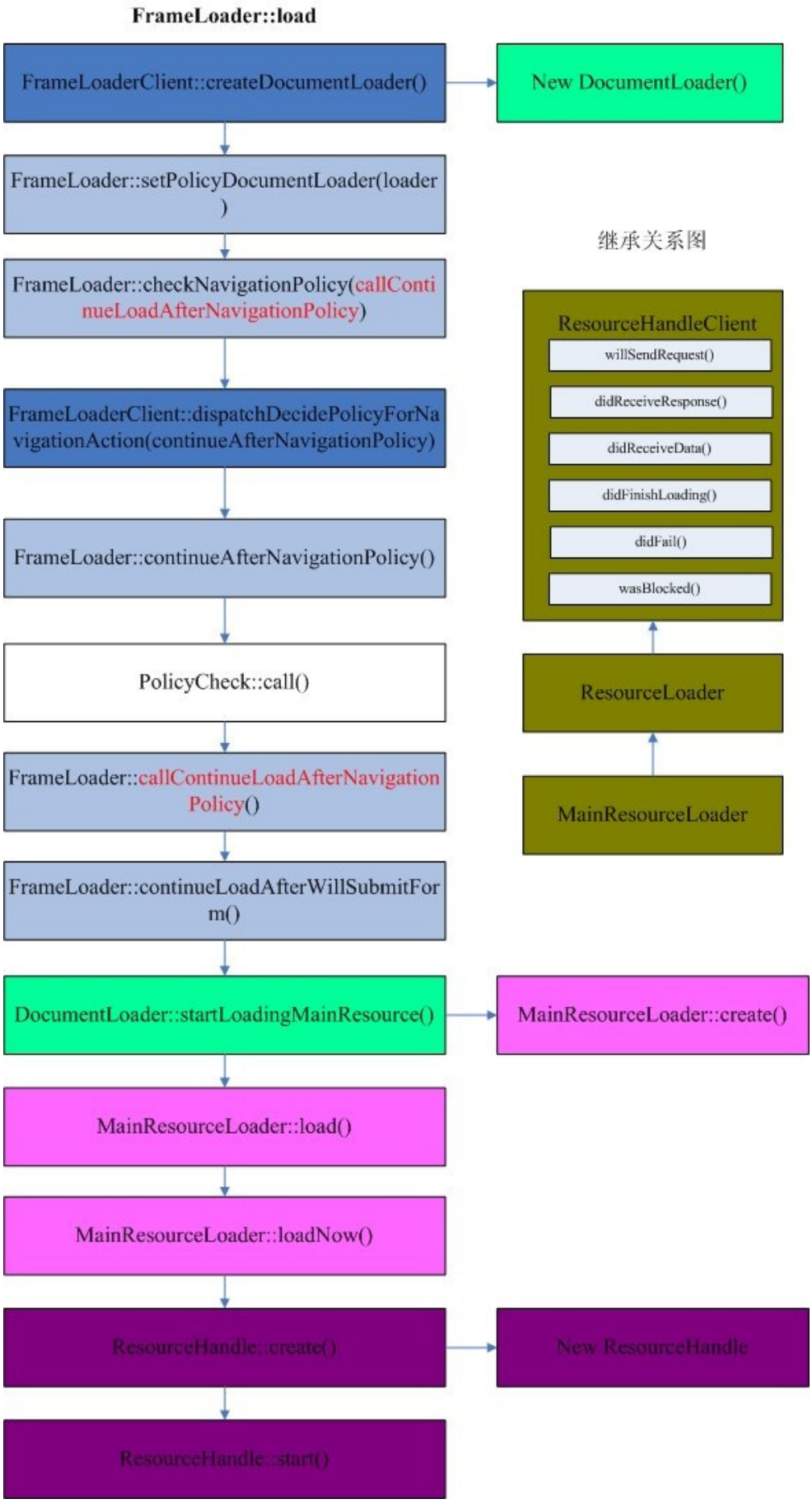
布局管理器通过 Render 树，开始布局。这个是一个动态的过程，DOM 在这个时候会继续向服务器申请自己需要的东西，比如 CSS,JavaScript，图片等。然后布局器动态的加载或布局。这样可以改善用户的使用。

最后把整个网页的 render 出来。

3.代码流程

1、当调用webkit_web_view_open(url)时会触发core(webView)->mainFrame()->loader()->load(uri)(即调用FrameLoader.load)来发起一个Http页面请求;

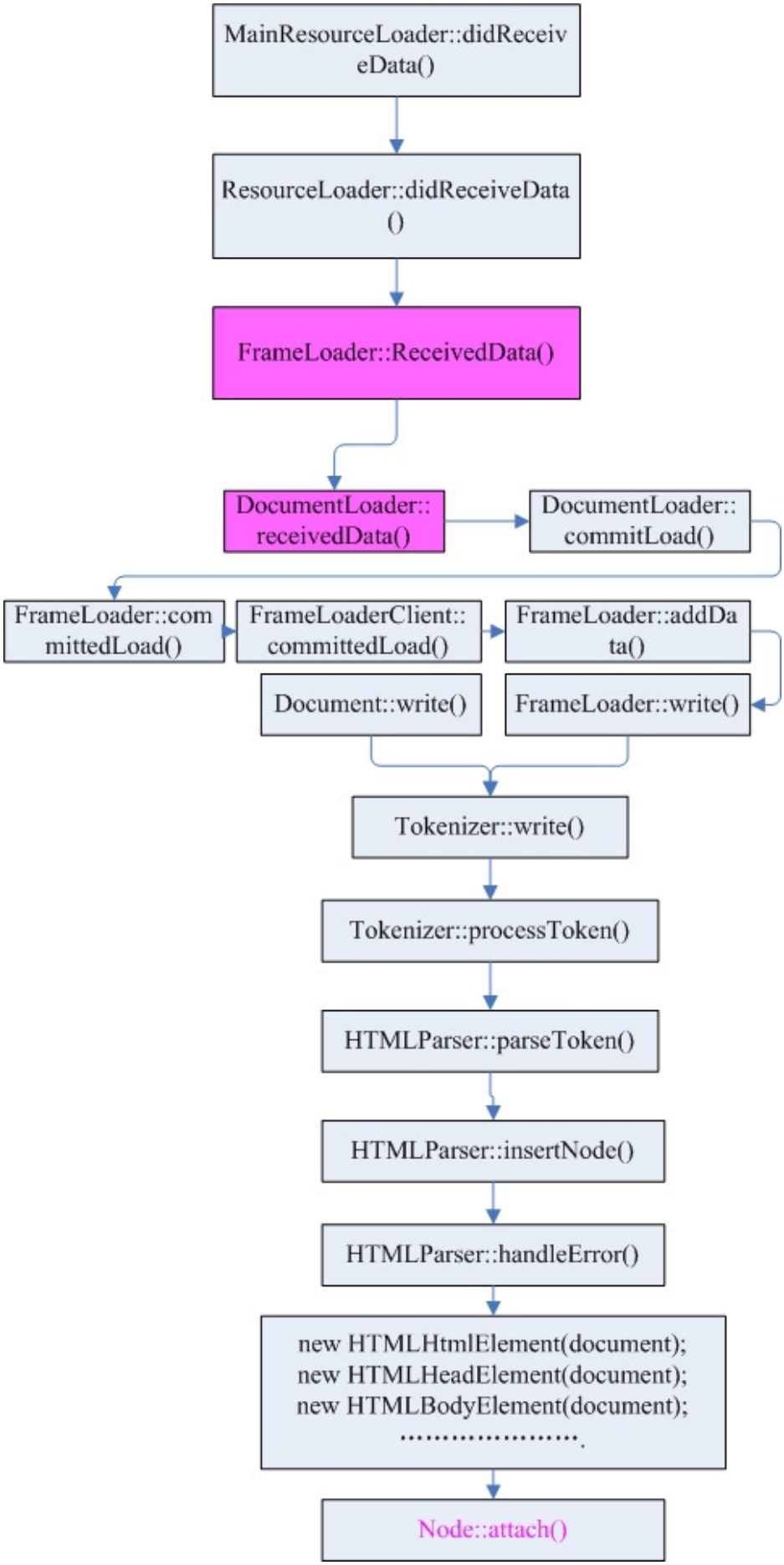
FrameLoader.load方法的主要处理过程如图:


FrameLoader::load
FrameLoaderClient::createDocumentLoader()
New DocumentLoader()
FrameLoader::setPolicyDocumentLoader(loader)
FrameLoader::checkNavigationPolicy(callContinueLoadAfterNavigationPolicy)
FrameLoaderClient::dispatchDecidePolicyForNavigationAction(continueAfterNavigationPolicy)
FrameLoader::continueAfterNavigationPolicy()
PolicyCheck::call()
FrameLoader::callContinueLoadAfterNavigationPolicy()
FrameLoader::continueLoadAfterWillSubmitForm()
DocumentLoader::startLoadingMainResource()
MainResourceLoader::create()
MainResourceLoader::load()
MainResourceLoader::loadNow()
ResourceHandle::create()
New ResourceHandle
ResourceHandle::start()
继承关系图
ResourceHandleClient
willSendRequest()
didReceiveResponse()
didReceiveData()
didFinishLoading()
didFail()
wasBlocked()
ResourceLoader
MainResourceLoader

2、一旦发起 ResourceHandle::start，就会由网络库向 web 服务器发起一个 http 请求;

3、 而 MainResourceLoader 作为一个 ResouceHandleClient，提供了诸如 didReceiveData()、 didReceiveResponse()等回调接口以供网络库调用，一旦从 web 服务器获得相关数据后网络库部分则会调用相关接口如 didReceiveData 等;

4、MainResouceLoader::didReceiveData 的主要回调处理过程如下图:


MainResourceLoader::didReceiveData()
ResourceLoader::didReceiveData()
FrameLoader::ReceivedData()
DocumentLoader::receivedData()
DocumentLoader::commitLoad()
FrameLoader::committedLoad()
FrameLoaderClient::committedLoad()
FrameLoader::addData()
Document::write()
FrameLoader::write()
Tokenizer::write()
Tokenizer::processToken()
HTMLParser::parseToken()
HTMLParser::insertNode()
HTMLParser::handleError()
new HTMLHtmlElement(document);
new HTMLHeadElement(document);
new HTMLBodyElement(document);
.....
Node::attach()

5、通过回调 didReceiveData()方法，进而调用 Node.attach()方法，这样就会解析生成 document，同时会创建 frameview、domwindow 等；

6、创建的 frameview 会触发 layoutTimerFired 时间 Timer，进而调用 layout()方法，从而触发 RenderObject 的创建、布局等，同时或许会 invalidateRect，进而触发操作系统图形库的 paint 消息事件；

7、 由程序主消息处理循环接收 paint 消息事件，进而获取对应 frame ， 获 取 或 创 建 GraphicContext ， 然 后 调 用 frame->view()->paint(&ctx,...)，从而触发对应 RenderObject 树进行重画处理，这样一个完 整的页面就会逐步的显示出来。

三. 网络库、图形库、Javascript 实现与 WebCore 的集成

为方便扩展及模块化，WebCore 在处理浏览页面的过程中，往往使用了类似 java 或 gecko 中接口的概念，一般先定义一组公共接口或基类，然后由不同模块来实现。

如网络处理部分由 WebCore 提供一个 ResourceHandle 类，而在不同的目录如 cf、curl、qt、soup、win 等中在不同网络库的支持 下对 ResourceHandle 类提供不同的实现，待编译时择机选择对应目录下的实现，这种方式从架构的角度看比较简单，但往往不能让程序同时使用多个 网络库，进而由程序动态切换使用不同网络库实现，而 gecko 在 xpcom 的基础上提供了对于这种扩展形式的支持；其中 Chrome 对 ResouceHanle 类的实现基于 WinHttp 网络库。

同样 WebCore 对图形库的集成，也是采取这种 方式来实现，如

由 WebCore 提供一个 GraphicsContext 类，然后在不同的目录如 cairo、cg、qt、win、wx 中在不同的图形库支 持下对 GraphicsContext 提供不同的实现。其中 Chrome 对 GraphicsContext 类的实现基于 Skia 图形库。

WebCore 中实现的 dom、html、svg、css 等，往往需要通过一定的方式输出给 Javascript 的实现如 JavascriptCore、V8，以便 JS Engineer 能认识这些 dom 元素等，并且能调用其中的方法，这种方式叫做 Binding，为了便于将 WebCore 中相对固定的 dom、html、svg、css 接口等极其方便的 Binding 出去，WebKit 使用了极其高效及神奇的方式来实现。

III. WebKit 之 Port 介绍

一. 有关 Port 方面的概述

WebKit Port 方面的内容是可以很广的，例如可将不同的图形库、网络库与 WebCore 集成，提供不同的 Port 接口供外部程序使用等，例如同样在 windows 平台上可以运行的 Google Chrome 和 Safari 就是针对 WebKit 的不同移植。

我们想了解有关 Port 方面的主要内容在于提供不同的 Port 接口供外部程序使用以及如何与外部程序交互，因为 WebKit 中的其它两部分 WebCore、 Javascript 实现，从逻辑上讲是不直接提供接口给外部程序使用的。同时为了完成浏览器的核心功能，WebKit 也需

要从外部程序中通过 Port 接 口的方式获取一些支持。

从这个角度讲 WebKit 作为一个相对独立的整体，它与外部程序之间的交互也就有一组相对固定的接口来定义及维护它们之间的关系，它们之间的关系与插件跟浏览器引擎之间的关系完全类似，接口相当一组协议，有的是由 WebKit 来实现，而供外部程序调用，有的的正好相反。

通过前面的了解我们知道 WebKit 的主要功能集中在分析 Html、渲染布局 Web 内容以及 Javascript 实现方面等，而这些 Web 内容显示在哪个窗口及消息处理的启动循环等都需要由外部程序来提供。

二. WebKit Port 移植实现分析

1. WebCore 交互接口

在 WebKit 源代码目录结构中 WebKit 目录下分别包含 gtk、mac、qt、win、wx 目录，其分别对应不同的 Port 移植方式，在每一个目录下面 都包括 WebCoreSupport 目录，而在不同的 WebCoreSupport 目录下分别包含有对类接口 WebCore::ChromeClient 、WebCore::ContextMenuClient 、 WebCore::DragClient 、WebCore::EditorClient 、 WebCore::FrameLoaderClient 、WebCore::InspectorClient 等的实现，它们代表外部程序提供给 WebKit 内部使用的接口实现，其中 WebCore::ChromeClient、WebCore::FrameLoaderClient 非常重要。

初步了解其接口定义能基本了解其对应的含义，这些接口往往需

要由 Port 移植部分来提供实现，往往由 WebKit 内部根据一定的条件来调用。下面初步来了解几个主要接口：

WebCore::ChromeClient 接口：

```
//往往在运行 window.open 脚本时调用，以便由外部程序决定如何打开一个新页面如新建、
//一个窗口、新建一个 Tab 页签等；
virtual WebCore::Page* createWindow(WebCore::Frame*, const WebCore::FrameLoadRequest&, const WebCore::WindowFeatures&);
//通知外部程序显示页面；
virtual void show();
virtual bool canRunModal();
//通知外部程序以 Modal 的方式显示页面；
virtual void runModal();
//通知外部程序显示 JS 警告提示窗口；
virtual void runJavaScriptAlert(WebCore::Frame*, const WebCore::String&);
//通知外部程序显示 JS 警告确认窗口；
virtual bool runJavaScriptConfirm(WebCore::Frame*, const WebCore::String&);
WebCore::FrameLoaderClient 接口：
//检查是否拥有主页面窗口；
virtual bool hasWebView() const;
//检查是否拥有页面窗口；
virtual bool hasFrameView() const;
//通知外部程序有关 http 请求开始、结束、获取数据等，如通常浏览器状态栏显示的信息；
virtual void dispatchDidReceiveResponse(WebCore::DocumentLoader*, unsigned long identifier, const WebCore::ResourceResponse&);
virtual void dispatchDidReceiveContentLength(WebCore::DocumentLoader*, unsigned long identifier, int lengthReceived);
virtual void dispatchDidFinishLoading(WebCore::DocumentLoader*, unsigned long identifier);
virtual void dispatchDidFailLoading(WebCore::DocumentLoader*, unsigned long identifier, const WebCore::ResourceError&);
//通知外部程序 WebKit 内部主要事件处理，以便外部程序及时响应或创建维护数据等
virtual void dispatchDidHandleOnloadEvents();
virtual void dispatchDidReceiveServerRedirectForProvisionalLoad();
virtual void dispatchDidCancelClientRedirect();
virtual void dispatchWillPerformClientRedirect(const WebCore::KURL&, double interval, double fireDate);
virtual void dispatchDidChangeLocationWithinPage();
virtual void dispatchWillClose();
virtual void dispatchDidReceiveIcon();
virtual void dispatchDidStartProvisionalLoad();
virtual void dispatchDidReceiveTitle(const WebCore::String&);
```

```
virtual void dispatchDidCommitLoad();
virtual void dispatchDidFinishDocumentLoad();
virtual void dispatchDidFinishLoad();
virtual void dispatchDidFirstLayout();
//告诉外部程序需要提供切换到一个新页面状态。此时外部程序往往会新建 FrameView,
并将 FrameView 与 Frame 关联, 设置原生窗口句柄及其消息处理机制等等；
virtual void transitionToCommittedForNewPage();
//告诉外部程序创建一个新的 Frame, 如遇到 html 中 iframe 标签时, 需要外部程序创建一
个新的 Frame 及原生窗口句柄等；
virtual PassRefPtr createFrame(const WebCore::KURL& url, const WebCore::String&
name, WebCore::HTMLFrameOwnerElement* ownerElement,
const WebCore::String& referrer, bool allowsScrolling, int marginWidth, int
marginHeight);
//告诉外部程序需要创建一个 Plugin 实例, 从而创建其原生窗口等等；
virtual WebCore::Widget* createPlugin(const WebCore::IntSize&, WebCore::Element*,
const WebCore::KURL&, const Vector&, const Vector&, const WebCore::String&,
bool loadManually);
```

2.连接模块 loader

对 WebCore 中的 page/loader 等方面的类提供对应 Port 的实现支持如 EventHandlerWin.cpp 、 FrameLoaderWin.cpp 、 DocumentLoaderWin.cpp、DocumentLoaderWin.cpp、WidgetWin.cpp、KeyEventWin.cpp 等.

Loader 是在 WebKit 里面一个很重要的连接器，通过 loader 发起 IO 下载网页，再通过 loader 发起解析，已经最后的渲染功能。

3.显示模块 WebView 和 WebFrame

WebView 及 WebFrame 主要功能是方便外部程序嵌入 WebKit 不同的 Port 移植对 WebView 及 WebFrame 的定义及实现有所不同，但其与 WebCore 中的 Page、Frame 之间的关系图描述相一致。具体关于 WebView、WebFrame 的定义与实现，特别是初始化时的动作可根据不

同的 Port 移植而有所不同，同时初始化时会将上面提到的 WebCore Port 接口实现告诉 WebKit 内部。主要示例代码如下：

```
static void webkit_web_view_init(WebKitWebView* webView)
{
WebKitWebViewPrivate* priv = WEBKIT_WEB_VIEW_GET_PRIVATE(webView);
webView->priv = priv;
priv->corePage = new Page(new WebKit::ChromeClient(webView), new WebKit::ContextMenuClient(webView), new WebKit::EditorClient(webView), new WebKit::DragClient, new WebKit::InspectorClient);
priv->mainFrame = WEBKIT_WEB_FRAME(webkit_web_frame_new(webView));
priv->lastPopupXPosition = priv->lastPopupYPosition = -1;
priv->editable = false;
.....

priv->webSettings = webkit_web_settings_new();
webkit_web_view_update_settings(webView);
.....
}

WebKitWebFrame* webkit_web_frame_new(WebKitWebView* webView)
{
g_return_val_if_fail(WEBKIT_IS_WEB_VIEW(webView), NULL);

WebKitWebFrame* frame = WEBKIT_WEB_FRAME(g_object_new(WEBKIT_TYPE_WEB_FRAME, NULL));
WebKitWebFramePrivate* priv = frame->priv;
WebKitWebViewPrivate* viewPriv = WEBKIT_WEB_VIEW_GET_PRIVATE(webView);

priv->webView = webView;
priv->client = new WebKit::FrameLoaderClient(frame);
priv->coreFrame = Frame::create(viewPriv->corePage, 0, priv->client).get();
priv->coreFrame->init();

return frame;
}
```

4. Chrome 中对 Port 移植方面的实现

其基本上与其他 Port 移植类似，其主要代码在 webkit\glue 目录中，可重点关注带 client_impl.cc 后缀的文件、webview_impl.cc、webwidget_impl.cc 等。但是其究竟如何创建原生 windows 窗口、如何创建 Render 进程、 Render 进程与创建的原生 windows 窗口的关系如何等需要更进一步深入研究 Chrome，如果能从上面提到的 Port 部分入手也许很快就可得到答案。

5. Android 中对 Port 移植方面的实现

其实现有点特殊，由 于 Andriod 将 WebKit 以一个 Java 类接口的方式提供给 Java 环境使用(不像上面提到的 Chrome、Safari 等都是将 WebKit 以 一个 C++动态或静态库的方式供 C/C++外部程序调用)，这样 WebKit 内部与外部即 JavaVM 的交互(如上面提到的 ChromeClient、FrameLoaderClient 接口实现)需要一个 Bridge 类来协调处理，同时 WebView、WebFrame 接口绑定给 JavaVM 的 jni 接口实现也需要通过这个 Bridge 来支持协调处理。具体可详细参考 android 源码代码中 WebCore\platform\android 目录下的源文件。

6. 结论

通过进一步了解 WebCore Port 接口及其实现，可以加深这样一个认识。如果从 MVC 的角度来看整个基于 WebKit 的浏览器(当然不尽合理)，WebKit 的 Port 部分相当于 V 部分，它提供显示页面内容及其辅助信息(如提示状态)的场所(即原生窗口)以及控制该显示场所

的状态变化及消息响应(如改变大小、 鼠标移动等)；而 M 部分往往由 WebCore 来实现,至于 WebCore 如何组织 DOM 则往往由 htmlparser 部分根据 DOM 定义来组织，如何在提 供的显示场所显示 Web 内容则往往由 WebCore 中的 layout 部分来实现，其中充分利用了 Css 定义来布局显示该显示的内容。一旦涉及控制或动态处理往往由 Port 部分发起而由 Javascript 脚本来实现处理，其任务由 JavascriptCore 或 V8 来完成。

一般说来新打开一个页面，Port 部分需要提供一个主显示场所(即原生窗口)，如果页面中含有 iframe 标签，则需要在主显示场所内创建一个子显示场所，以显示 iframe 标签对应 src 的内容。如果页面中含有 embed/object 等插件标签同样往往也需要在主显示场所内创建一个子显示场所(除非 windowless)，以交由插件实现在提供的显示场所中显示内容。

特别需要说明的是我们通常看到的页面表单元素 input text field、textArea、button、radiobutton 等往往不像 window 图形库中的按钮、菜单、输入框等会对应一个原生窗口，页面中的表单元素在一个显示场所(即原生窗口)中完全是利用 Css 等通过 layout 方式来达到我们所看到的类似原生按钮、输入框、列表框、滚动条等效果，其中 特别是能准确定位元素大小、设置 focus、光标显示、响应事件等，这充分的说明了浏览器引擎内部布局部分的威力所在。

从另外一个角度来看一个页面一般说来(除非遇到 iframe 或插件需要另外提供一块子画布)相当于一块画布，浏览器引擎能在其精

确的位置绘制不同颜色的文字、图片、图标 等，同时根据当前的鼠标及一个模拟的输入提示光标位置，接收键盘输入操作。页面中的绝大多数元素与原生的窗口元素几乎没有关联，完全通过组合、布局、准确定位来处理一切。

三.如何利用 WebKit?

了解 WebKit Port 部分，对我们如何利用 WebKit 有非常现实的意义，目前已经将 WebKit 移植到多种平台如 windows、qt、gtk、mac、wx、 java、framebuffer 等，甚至移植到 python、ruby 及 3D 等环境中去。通过借鉴或利用这些已有的 WebKitPort 实现，完全可以将 WebKit 发扬广大。

1.利用 WebKit 实现搜索引擎

试想目前移植利用 WebKit 基本都用来显示页面，往往涉及图形显示方面，但随着 ajax 及动态页面的广泛使用，未来动态生成的页面越来越多，传统的搜索引擎仅仅抓取静态的页面内容显然是不够的，现代化的搜索引擎应该能抓取动态的页面内容，这样它从某种意义讲相当于一个能获取对应的动态页面但不真正显示出其内容的浏览器,这样一个搜索引擎不仅能分析 DOM 树,同时能运行 Javascript 脚本(如运行 ajax)，以真正完整获取页面内容，其实这样一个搜索引擎如果利用 WebKit 来实现的话，应该是个不错的选择，在我们了解 WebKit Port 部分之后，我们是否可以来模拟一个不真正具备图形

显示方面的 Port，进而充分利用 WebKit 中的 WebCore 及 Javascript 实现方面的功能呢？

2. 利用 WebKit 实现平台功能

浏览器未来可望取代目前的操作平台将成为趋势。随着计算机、手机及连网装置也普及，未来终端运算都会在云端执行。目前计算机用户有 9 成的行为是在网络或靠着浏览器就可以完成，未来可能会再进一步提升到 95％或更高。

人们拥有一个强大功能的浏览器，就能满足平时工作生活的需要。在此情况下，浏览器就是未来的操作平台系统。WebKit 将能在其中扮演重要的引擎角色。

3. 高性能的渲染工具

WebKit 的 Rendering 技术是其核心技术，其优秀的 Render 技术得到肯定. 其渲染不仅仅只用于浏览器，也可以单独出来用于其他的渲染。但具体用于什么还有待我们想象。

IV.WebKit之图形库介绍

一.WebKit与图形库

WebKit 为了能高效美观的显示页面的内容，选择适当的图形库非常重要。如果图形库选择不当，往往会导致页面上显示的文字、图片不美观，看起来总让人觉得别扭，更为糟糕的是排列布局出现紊乱，简直无法阅览。

从浏览器发展的历史来看，IE系列浏览器的网页布局、文字图片显示的美观程度还是相当高的，也许这与Microsoft图形显示方面的功力相关，到目前为止linux桌面显示还是与传统的windows桌面显示有相当的差距。

相比较Firefox1.5，Firefox3.0图形显示方面也有相当大的进步，这应该归功于完全采取Cario图形库来显示页面，目前应当完全达到了 IE6的显示效果。可见图形显示的好与坏，直接决定了一款浏览器的质量以及用户接受程度。那究竟什么是图形库？其主要功能是什么？目前WebKit可使用哪些图形库？它们在浏览器中是如何发挥其应有的作用呢?

二.图形库概述及其主要功能

图形库是程序设计，旨在帮助计算机图形渲染和监测。这通常包括提供优化的版本的功能，处理共同绘制任务。

图形库可以做到纯粹的软件在CPU上运行在，能在嵌入式系统，

或由硬件加速的 GPU （更常见的电脑）共同使用。运用这些功能，可以组装程序的形象将输出到显示器。这减轻了程序员的任务，创造和优化了图形功能。

目前主要的图形库有：

windows: GDI/GDI+、DirectX、OpenGL;

支持 X: Cario、GTK、QT、OpenGL;

其他: Skia(google 提供)、Quartz 2D(apple 提供)、wxWidget 等。

一般说来图形库只提供绘画图形，渲染文字、图片等，不管是 2D 还是 3D，其往往不提供消息处理，简单的说来就是如何高效的在一块指定的画布上将线条、文字、图片显示出来，其中往往涉及字体、颜色等。典型的图形库如 GDI/GDI+、Cario、DirectX、Quartz 2D 等。

而按钮、菜单、窗口等图形组件往往是基于图形库的基础上绘画出来的并有相对固定形状，同时一般具有消息处理功能。相关实现有 GTK、QT、wxWidget、windows 组件等。

其中 GTK、QT、wxWidget、Skia 等不仅提供图形组件，同时提供图形库的功能。而 Cario 则是一个纯粹的图形库，类似与 Quartz 2D，目前 GTK2 则完全基于 Cario 来实现。

由于浏览器页面元素的数量存在不确定性，将页面上的一些元素对应成图形组件可能导致一个页面使用组件过多的问题(早期的 IE 就曾出现使用 GDI 对象过多的现象)。因此尽可能的将一个页面的所有元素显示在一个图形组件上，至于其显示交给图形库来处理，其消息响应交互交给 DOM 及原生窗口消息循环来完成。

从这里我们可以进一步的确认图形库在浏览器中的重要性,以及随着用户需求的增加及硬件的提升，浏览器中使用 3D 效果应该是一个大的方向。

三.WebKit 与 Cario

Cario 是一个 2D 图形库，支持多种输出设备。目前支持的输出目标包括 X Window 系统，Quartz， Win32 ， image.buffers, PostScript ， PDF,和 SVG 文件输出。后续版本支持 OpenGL ， XCB ， BeOS ， OS / 2 操作系统，和 DirectFB 。

Cario 的目的是统一输出产物在不同的输出媒体上，同时能利用硬件加速器（例如，通过 X 渲染扩展部分）。

其主要优点在于其在 X、Win32、Quartz 的基础上统一了图形库的操作方式，同时支持 PS、PDF、SVG、PNG/JPEG 等图像格式的输出，极大的方便页面的再次利用，在 glitz 的支持下支持部分 3D 效果。

目前 WebKit 支持的图形库包括 Cairo、Gtk、Qt、Wx、Cg、Mac、Skia 等，虽然不同的图形库能支持不同的平台.但其在不同平台上的显示效果也不尽相同。至于在一个指定的平台上究竟使用何种库，则显示出很大的灵活性。就目前来看，在 windows 平台上可选的图形库有 Cairo、Qt、Wx、Cg、Skia 等。

其实从 WebKit 的角度来看，它通过提供一组公共图形接口，而不同的图形库则根据自身的不同实现了这些公共图形接口，以提供给 WebCore 元素使用，从而可以让 WebKit 支持不同的图形库。

四.WebKit 如何支持不同图形库的实现

在 WebKit 中提供了一个 GraphicsContext 类，其中包括所有的图形接口，针对不同平台的特性，其定义中包含一些不同平台特有的宏及元素定义。

在目录 webcore\platform\graphics\下的子目录 Cairo、Cg、Gtk、Mac、Qt、Win、Wx 分别提供了 GraphicsContext 类部分方法的实现，而公共的实现则在 webcore\platform\graphics\GraphicsContext.cpp 中提供。

其中我们非常值得关注的方法有 drawText 与 drawImage，其实现如下:

```
void GraphicsContext::drawText(const TextRun& run, const IntPoint& point, int from, int to)
{
if (paintingDisabled())
return;
font().drawText(this, run, point, from, to);
}

void GraphicsContext::drawImage(Image* image, const FloatRect& dest, const FloatRect& src, CompositeOperator op, bool useLowQualityScale)
{
if (paintingDisabled() || !image)
return;

float tsw = src.width();
float tsh = src.height();
float tw = dest.width();
float th = dest.height();

if (tsw == -1)
tsw = image->width();
if (tsh == -1)
```

```
tsh = image->height();

if (tw == -1)
tw = image->width();
if (th == -1)
th = image->height();

if (useLowQualityScale) {
save();
setUseLowQualityImageInterpolation(true);
}
image->draw(this, FloatRect(dest.location(), FloatSize(tw, th)),
FloatRect(src.location(), FloatSize(tsw, tsh)), op);
if (useLowQualityScale)
restore();
}
```

最终的实现转交给类 Font、Image 的方法 drawText、draw 来实现，而不同实现如 Cairo、Cg、Gtk、Mac、Qt、Win、Wx 则会 针对类 Font、Image 分别提供部分对应的实现，而公共的实现则在 webcore\platform\graphics\Font.cpp 及 Image.cpp 中提供。不同平台 GraphicsContext 实例创建及使用。

GraphicsContext 创建的时机往往在对应平台的 WebView 获得 Paint 消息事件时，进而将该 GraphicsContext 类实例传递给 FrameView 及其不同的 RenderObject 实例，由不同的 RenderObject 实例来决定究竟如何来显示自身的内容，而 GraphicsContext 类实例提供了各种的显示文字、图形、图像的方法以供 RenderObject 实例调用。其调用关系基本上与 Gecko 中的不同 Frame 对象使用 nsIRenderingContext 接口方法类似。

创建 GraphicsContext 实例的示例如下:

```
//Gtk
static gboolean webkit_web_view_expose_event(GtkWidget* widget,
```

```
GdkEventExpose* event)
{
WebKitWebView* webView = WEBKIT_WEB_VIEW(widget);
WebKitWebViewPrivate* priv = webView->priv;

Frame* frame = core(webView)->mainFrame();
GdkRectangle clip;
gdk_region_get_clipbox(event->region, &clip);
cairo_t* cr = gdk_cairo_create(event->window);
GraphicsContext ctx(cr);
ctx.setGdkExposeEvent(event);
if (frame->contentRenderer() && frame->view()) {
frame->view()->layoutIfNeededRecursive();

if (priv->transparent) {
cairo_save(cr);
cairo_set_operator(cr, CAIRO_OPERATOR_CLEAR);
cairo_paint(cr);
cairo_restore(cr);
}

frame->view()->paint(&ctx, clip);
}
cairo_destroy(cr);
return FALSE;
}

//win
void WebView::paintIntoBackingStore(FrameView* frameView, HDC bitmapDC, const
IntRect& dirtyRect)
{
LOCAL_GDI_COUNTER(0, __FUNCTION__);

RECT rect = dirtyRect;

#if FLASH_BACKING_STORE_REDRAW
HDC dc = ::GetDC(m_viewWindow);
OwnPtr yellowBrush = CreateSolidBrush(RGB(255, 255, 0));
FillRect(dc, &rect, yellowBrush.get());
GdiFlush();
Sleep(50);
paintIntoWindow(bitmapDC, dc, dirtyRect);
::ReleaseDC(m_viewWindow, dc);
#endif
```

```
FillRect(bitmapDC, &rect, (HBRUSH)GetStockObject(WHITE_BRUSH));
if (frameView && frameView->frame() && frameView->frame()->contentRenderer())
{
GraphicsContext gc(bitmapDC);
gc.save();
gc.clip(dirtyRect);
frameView->paint(&gc, dirtyRect);
gc.restore();
}
}

//wx
void wxWebView::OnPaint(wxPaintEvent& event)
{
if (m_beingDestroyed || !m_impl->frame->view() || !m_impl->frame)
return;

wxAutoBufferedPaintDC dc(this);

if (IsShown() && m_impl->frame && m_impl->frame->document()) {
#if USE(WXGC)
wxGCDC gcdc(dc);
#endif

if (dc.IsOk()) {
wxRect paintRect = GetUpdateRegion().GetBox();

WebCore::IntSize offset = m_impl->frame->view()->scrollOffset();
#if USE(WXGC)
gcdc.SetDeviceOrigin(-offset.width(), -offset.height());
#endif
dc.SetDeviceOrigin(-offset.width(), -offset.height());
paintRect.Offset(offset.width(), offset.height());

#if USE(WXGC)
WebCore::GraphicsContext* gc = new WebCore::GraphicsContext(&gcdc);
#else
WebCore::GraphicsContext*                gc                =                new
WebCore::GraphicsContext((wxWindowDC*)&dc);
#endif
if (gc && m_impl->frame->contentRenderer()) {
if (m_impl->frame->view()->needsLayout())
m_impl->frame->view()->layout();
```

```
m_impl->frame->paint(gc, paintRect);
}
}
}
}

//Qt
void QWebFrame::render(QPainter *painter, const QRegion &clip)
{
if (!d->frame->view() || !d->frame->contentRenderer())
return;

d->frame->view()->layoutIfNeededRecursive();

GraphicsContext ctx(painter);
QVector vector = clip.rects();
WebCore::FrameView* view = d->frame->view();
for (int i = 0; i <>paint(&ctx, vector.at(i));
}

/*!
Render the frame into ₩a painter.
*/
void QWebFrame::render(QPainter *painter)
{
if (!d->frame->view() || !d->frame->contentRenderer())
return;

d->frame->view()->layoutIfNeededRecursive();

GraphicsContext ctx(painter);
WebCore::FrameView* view = d->frame->view();
view->paint(&ctx, view->frameGeometry());
}
```

五.WebKit 3D Port 实现

WebKit 对 3D Port 的支持与实现，其实现类似于 Gtk+/Cairo 图形库的实现，但其 3D 效果仅实现在 Port 层，没有对页面上的元素如文字、图像实现 3D 效果支持。

这里只是简单的了解 WebKit 中如何整合不同的图形库及其与 WebCore 的交互。要想更加深入的了解页面上的文字、图形、图像究竟是如何显示出来的，则需要更进一步的针对不同平台库进行学习与了解。

六. 总结

其实关于图形库及其使用的内容非常的多，而对浏览器内核来讲能对图形库进行高效使用也是非常重要的一部分。目前图形库也在不断的更新，一些商业图形库正在兴起，WebKit 专注其引擎，而把图形显示独立出来这点是比 Gecko 好得多。

V. WebKit 之网络库介绍

一. 网络原理

1. 超文本传输协议

超文本传输协议（Hypertext Transfer Protocol，HTTP）是 Web 的基础协议。HTTP 是一个简单的协议。客户进程建立一条同服务器进程的 TCP 连接，然后发出请求并读取服务器进程的应答。服务器进程关闭连接表示本次响应结束。服务器进程返回的内容包含两个部分，一个“应答头”（response header），一个“应答体”（response

body)，后者通常是一个 HTML 文件，我们称之为“网页”。

我们传给服务器 GET 加需要的网页 ，服务器却返回了很多字节。这样，从该 Web 服务器的 根目录下取得了它的主页。一个完整的 HTML 文档以<HTML>开始，以</HTML>结束。大部分的 HTML 命令都像这样成对出现。HTML 文档含有以<HEAD>开始、以</HEAD>结束的首 部和以<BODY>开始、以</BODY>结束的主体部分。标题通常由客户程序显示在 窗口的顶部。

2. URL 类

用户需要打开一个 URL，浏览器将通过该 URL 发送给服务器，服务器返回网页内容，浏览器把内容现实在一个 Page 上。

为了发送 URL 指向的服务器。一般网络模块会定义 Url 类。

```
enum url_scheme{
SCHEME_HTTP,
SCHEME_FTP,
SCHEME_INVALID
};

class CUrl
{
public:
string    m_sUrl; // URL 字串
enum url_scheme m_eScheme;   // 协议名
string    m_sHost;    // 主机名
int    m_nPort;   // 端口号
string m_sPath; // 请求资源
public:
CUrl();
~CUrl();
bool ParseUrl( string strUrl );
private:
void ParseScheme ( const char *url );
```

};

URL 可以是 HTTP，FTP 等协议开始的字符串，在 url_scheme 中定义了 SCHEME_HTTP，SCHEME_FTP， SCHEME_INVALID，分别对应 HTTP 协议，FTP 协议和其他协议。一个 URL 由 6 个部分组成，除了 scheme 部分，其他部分可以不在 URL 中同时出现。

<scheme>://<net_loc>/<path>;<params>?<query>#<fragment>

Scheme

表示协议名称，对应于 URL 类中的 m_eScheme.

Net_loc

表示网络位置，包括主机名和端口号，对应于 URL 类中的 m_sHost 和 m_nPort.

Path

表示 URL 路径.

Params

表示对象参数.

Query

表示查询信息，也经常记为 request.

Fragment

表示片断标识.

3.Page 类

有了 URL,搜集系统就可以按照 URL 标识抓取其所对应的网页，网页信息 保存在 Page 类中。

下面是 Page 类的定义，对应文件 Page.h。

```
class CPage
{
public:
string m_sUrl;
// 网页头信息
string m_sHeader;
int m_nLenHeader;

int m_nStatusCode;
int m_nContentLength;
string m_sLocation;

bool m_bConnectionState;
string m_sContentEncoding; string m_sContentType; string m_sCharset;
// 如果连接关闭，是 false，否则为 true
string m_sTransferEncoding; //页体信息
string m_sContent;
int m_nLenContent;
string m_sContentNoTags;

// link, in a lash-up state
string m_sContentLinkInfo;
// 为搜索引擎准备的链接，in a lash-up state string m_sLinkInfo4SE;
int m_nLenLinkInfo4SE;

// 为历史存档准备的链接，  in a lash-up state string m_sLinkInfo4History;
int m_nLenLinkInfo4History;

//为搜索准备的链接，in a good state
RefLink4SE m_RefLink4SE[MAX_URL_REFERENCES]; int m_nRefLink4SENum;
//为历史存档准备的链接, in a good state
RefLink4History          m_RefLink4History[MAX_URL_REFERENCES/2];          int
m_nRefLink4HistoryNum;

map<string,string> m_mapLink4SE;
vector<string > m_vecLink4History;

enum page_type m_eType;   // 网页类型
public:
CPage();
CPage::CPage(string strUrl, string strLocation, char* header, char* body,
    int nLenBody);
```

```
~CPage();

void ParseHeaderInfo(string header);
bool ParseHyperLinks();

bool NormalizeUrl(string& strUrl);
bool IsFilterLink(string plink);
private:
// 解析网页头信息
void GetStatusCode(string header);
void GetContentLength(string header);
// 解析网页头信息
// 从网页体中解析链接信息

void GetConnectionState(string header);
void GetLocation(string header);
void GetCharset(string header);
void GetContentEncoding(string header);
void GetContentType(string header);
void GetTransferEncoding(string header);

// 从网页体中解析链接信息
bool GetContentLinkInfo();
    bool GetLinkInfo4SE();
    bool GetLinkInfo4History();
    bool FindRefLink4SE();
    bool FindRefLink4History();
 };
```

一个网页是以 URL 作为标识的，所以 Page 类的第一个成员变量是 m_sUrl。Page 类主要完成两个任务：解析网页头信息和提取链接信息。

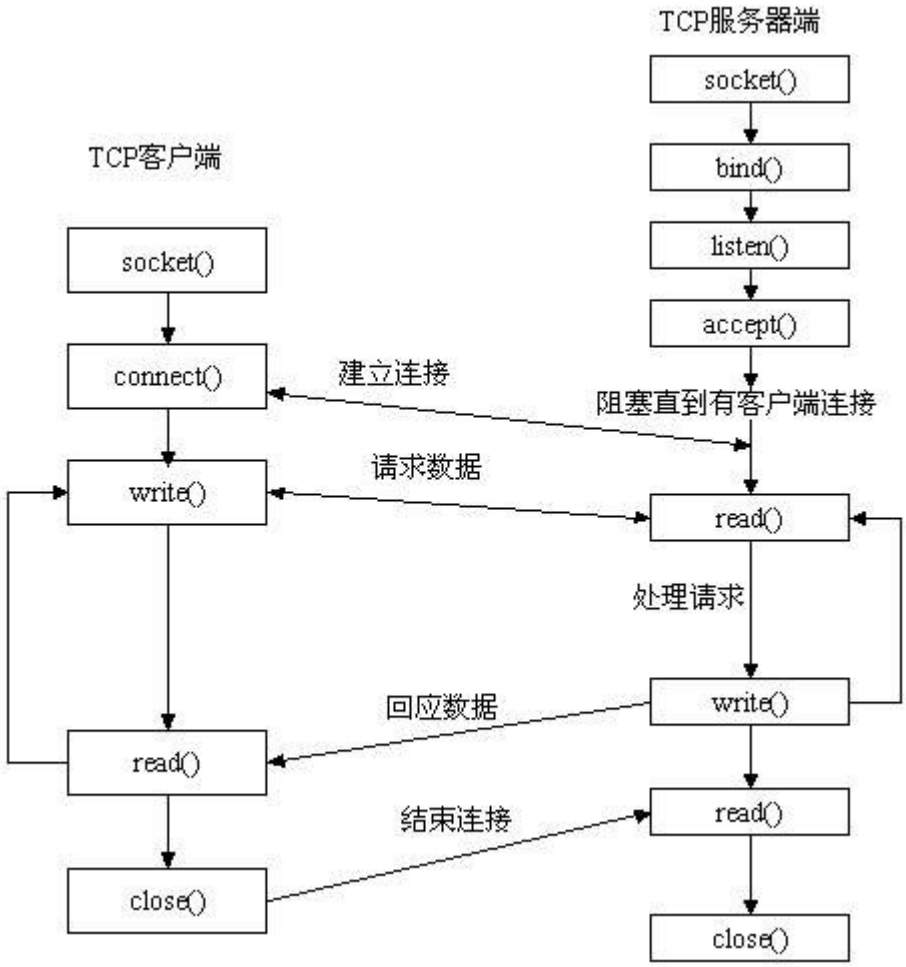
解析网页头信息包括获得状态码 m_nStatusCode ，网页体长度 m_nContentLength（内容字节数），转向信息 m_sLocation，连接状态（如果没有 关闭，下次请求同一个网站可以重新利用已经建立好的 socket，节约资源），网页体编码 m_sContentEncoding（如果是 gzip 编码，要解压缩，然后提取链接信息。现在门户网站的首页有

增大趋势，为了加快传输速度，通常采用 gzip 编码压缩后 传输），网页类型 m_sContentType，网页体字符集 m_sCharset，和传输编码方式 m_sTransferEncoding。提取链接信息，是从获得的网页体中，根据 HTML 的规定，提取出链接信息 .

4.与服务器的连接

已经从 URL 中获得了服务器的主机名，要能够从服务器上获取网页内容,还需要客户端进程与服务端进程建立连接。UDP 和 TCP 的通信采用 Socket 方法 实现，Socket 为进程间通信提供了端点。通信由消息组成，消息是在一个进程的 Socket 与另一个进程 Socket 之间传送的。一个进程要能够接收消息，它的 Socket 必须绑定到一个本地端口和本地地址上。发送到指定 Internet 地址和端口上的消息，只能被绑定到该地址和端口的 Socket 所属进程接收。

连接的建立过程是异步的，一方在监听建立连接的请求，一 方将发起建立连接的请求。连接一旦被接受，操作系统（例如 UNIX）自动创建 新的 socket 使之与客户端连接成通信的通道，这样服务端就可以在原来的 Socket 上继续监听其他客户的请求了。连接建立后，双方进程可以通过建立好的连接进 行读写操作。



二.WebKit 与 CURL 网络库

cURL 是一个利用 URL 语法在命令行下工作的文件传输工具。它支持文件的上传和下载，所以是综合传输工具，但按传统，习惯称 cURL 为下载工具。

cURL 支持 SSL 证书时， HTTP POST 方法，超文本传输协定说，的 FTP 上载，网址形式的基础上载，代理，用户+密码验证功能等。

优异的表现，再加上开源，目前很多浏览器，彩信，E-mail 都通过 cURL 进行传输数据。他的本质其实是对 socket 的封装，提供用

户简单的 API。知己对一些默认的，或通用功能进行封装。WebKit 在 linux 下一般都用 cURL 作为其网络模块，可见其优异性。

VI.WebKit 之 DOM 分析

一.DOM 原理

1.DocView 模型

DocView 模型包括: 网页标识、网页类型、内容类别、标题、关键词、摘要、正文、相关链接等要素。其中正文和相关链接要素属于网页的内容数据，而其他 6 项则属于网页的元数据。下面将对模型中的各个要素作详细描述。

网页标识

是对 Web 上网页的唯一性标识，在 DocView 模型中使用网页的 URL 作为网页标识。

网页类型

是根据网页内容的表现形式进行划分的，在本节中将网页分为三类: 有主题网页（topic）、Hub 网页（hub）、图片网页（pic）。其中，有主题网页是指网页中通过文字描述了一件或多件事物，是有一定主题的。如一张具体的新 闻网页就是典型的有主题网页。Hub 网页是

指专门用来提供网页导向的网页，因 而是超链聚集的网页。如门户网站的首页就是典型的 Hub 网页。图片网页是指网页的内容是通过图片的形式体现的，其中文字很少，仅仅是对图片的一个说明。如某个机构包含图片的人员介绍网页就是典型的图片网页。

将网页分为上述三个类型是因为三类网页在用途和处理方法上存在较大的差别。其中 Hub 网页与其它两类网页的区别在于网页在 Web 上发挥的作用不同，Hub 网页通常不会具体的讲述一件事物，而是提供关于相关信息的链接集。而图片网页与其它两类网页的区别在于处理的方法不同，由于图片网页的内容是通过图片表达的而不是通过文字，因而传统信息处理领域的方法对图片网页是不够有效的。三类网页间的区别导致很多应用领域都会对它们作适当的区别。

内容类别

是从语义上对网页的内容进行分类，它是计算机获取网页语义信息的一个直接手段，在 Web 上的研究领域中有着广泛的使用。它是通过特定的分类器对网页内容分类得到的，依赖于一定的分类体系。

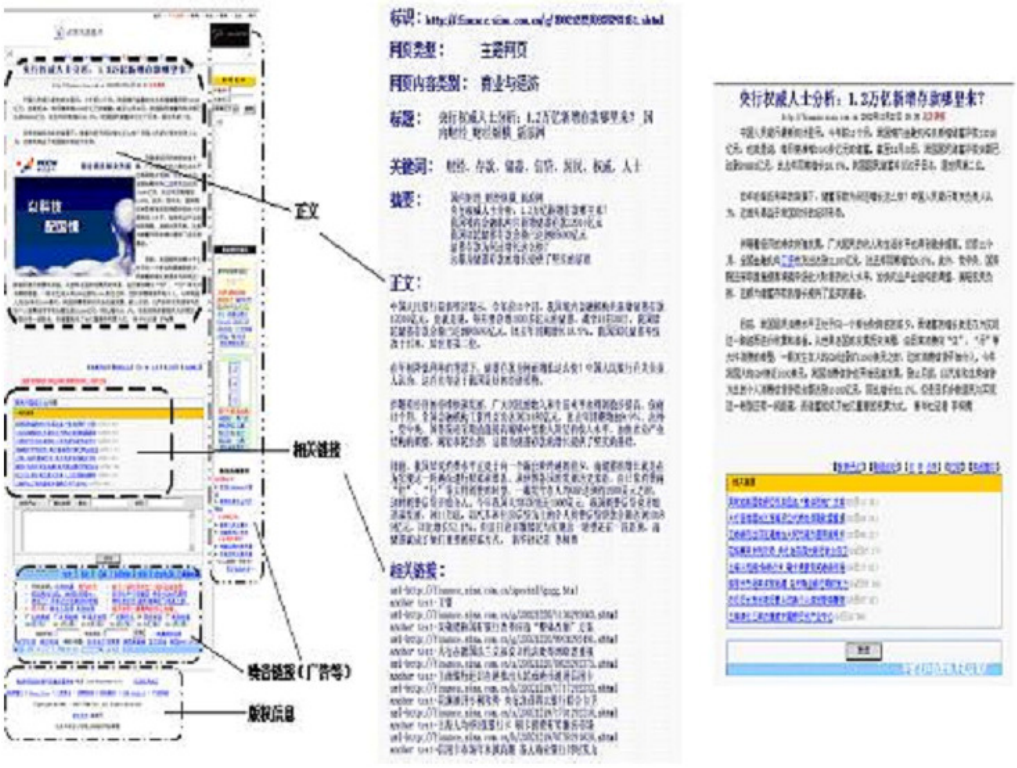
标题、关键词和摘要

是概括描述 Web 文档内容的重要的元数据，对于 Web 信息检索等领域的工作有非常重要的作用。

正文

是原始网页中真正描述主题的部分，因此，在某些具体应用中用正文代替原始网页更为合理。相关链接是指在本网页中指向与正文内容相关的网页的链接，而非广告等噪音链接。将正文和相关超链重新

组合就得到了净化后的网页。



用 DocView 模型提取的网页要素　　净化后的网页

2. 抽象网页表示

网页的表示是网页内容分析的基础，在网页内容分析过程中通常需要两个层次的表示，抽象表示和量化表示。抽象表示是以网页制作规范（如 HTML 规范）为依据和出发点，构造出能体现网页内容结构和内容重要性等信息的表示模型，其目的是充分利用网页制作规范，挖掘出网页中隐含的信息，为后续量化表示提供更多可利用信息。对于 HTML 网页，常用的方法是构造网页的标签树。量化表示则是从计算机处理的角度出发，利用信息检索领域的技术和从网页中挖掘的隐含信息，生成计算机可以直接用于计算的表示模型，如向量空间模型等，这里我们主要了解抽象网页的表示。

HTML 通过定义一套标签来刻画网页显示时的页面。因此，对于 HTML 网页常用的抽象表示方法是构造网页的标签树。依据标签的作用可以将 HTML 的标签分为三类:

规划网页布局的标签

在视觉上，网页是由若干提供内容信息的区域（我们称之为内容块）组成的，而内容块是由特定的标签规划出的（称之为容器标签），而且容器标签是允许嵌套的。常用的容器标签有<table>、<tr>、<td>、<p>、 等。因此，依据容器标签可以将网页表示成树状结构，虽然该树状结构描述的是网页内容的布局结构，但布局信息中隐含着网页内部各部分内容的相关性信息。

描述显示特点的标签

在 HTML 标准中定义了一套标签来规范其包含的内容的显示方式（比如: 字体变大、粗体、斜体），我们称之为重要信息标签。常用的重要信息标签有<b>、<I>、、<h1>、<h2>等十几种。这类标签中的内容通常是网页作者希望引起读者注意的，因此隐含着一定的内容重要性信息。

超链相关的标签

超链是 HTML 网页区别于传统文本的明显的特点之一，表示着网页间的关系，因此整理出超链标签并作合理的分析可以挖掘出网页间的 内容相关性信息。

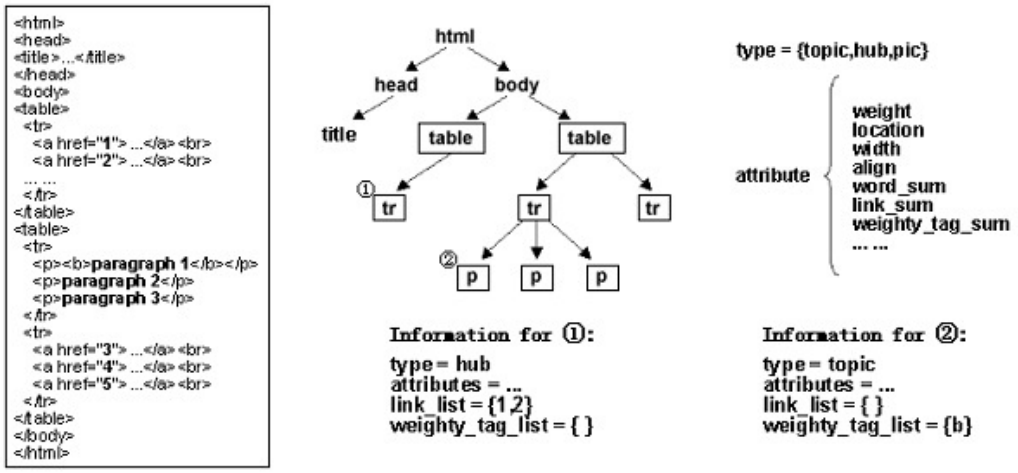
目前，有很多构造标签树的工具（如: W3C HTML lexical analyzer[W3C,1997]），它们各有特点，W3C HTML lexical analyzer

有很强的通用性，适合各种标识语言。HTML Tidy 则能够自动发现并修正标签的错误。由于内容分析需要在网页内部计算各个部分之间的相关性以及确定各部分内容的重要性，因此，用传统的顺序整理各种标签的方法构造出的标签树在用于内容分析时并不方便。适合内容分析的标签树强调内容块的概念，倾向于以内容块为单位的内容组织方式。另外，内容分析过程中经常会关心这样一些信息：标签树的规模（结点个数）、每个内容块包含的各种类型信息（如：文本、超链或图片）及其数量等。鉴于此，我们自行开发了更适合内容分析的标签树构造工具。

下面简要的描述标签树。给定一篇 HTML 网页，顺序整理出容器标签就得到了对应的标签树的框架。而后，整理每个内容块（对应标签树的一个结点）中的超链标签、图片标签和重要信息标签，并在标签树中对应的结点中记录下来。这样就构造了一棵基本的标签树。对上述基本标签树信息作适当的分析、整理就可以得到内容分析过程中需要的一些描述信息。譬如，依据内容块中词项数与图片数和超链数的比值可以为每个内容块设定一个类型，分为 topic、hub、pic 三种。如果内容块中词项数与图片数的比值小于某个阈值，该内容块就是 pic 类型，如内容块中作为 anchor text 出现的词项数与该块中总词项数的比值小于某个阈值，该内容块就是 hub 类型，否则为 topic 类型。这样，标签树中每个结点都有类型和属性集两组描述性信息，以及超链集和重要标签集等数据信息。下图是一个标签树的图例，其中 link_list 表示该内容块中超链集合；weighty_tag_list

表示该内容块中重要标签集合。

.



HTML Tree 结构

3. DOM 解析基本算法

基本算法思想:

1. Html 中每个 tag 都是都将作为树中的一个节点存在的，每个 tag 都属于树中的某一层。

2. 辅助数据结构：栈（stack）、List、HashTable。其中 HashTable[i]（i 属于 int 类型）是一个 List，用于临时存储第 i 层子 Tag。

3. 顺序扫描 Html 文本，当遇到”<A¯Z”这样的标志，表示可能是一个 Tag，调用 GetTag()函数对此段代码进行解析，解析出 Tag 名，Tag 属性等等。如果返回值不为空，那么将返回值入栈。并且记录次 tag 的开始位置。

4.遇到</A¯Z>这样的标志，表示可能是某个 Tag 的结束。解析出此结束标志的 Tag 名。如果在栈中找到与此结束标志名同名的元素（此元素属于栈中第 iLevel 层），那么表示找到匹配的 Tag。则 Tag 出栈，将 HashTable[iLevel+1]到 HashTable[maxLevel]中的所有元素取出作为此 Tag 的子节点。放入第 HashTable [iLevel]中。并记录 Tag 的结束位置。

5.对于<Tag>xxx</Tag>之间的字符串 xxx，将其作为特殊的 HtmlTextTag 处理。出栈，和入栈操作与普通 Tag 类似。

6.当栈为空的时候表示最后一次出栈的 Tag 给根节点。

伪代码:

```
1. public void Parse()
2.
3. {
4.
5.     char ch = GetCurrentChar();   //取第一个字符
6.
7.     while (!Eof())
8.
9.     {
10.
11.         if (ch == '<')
12.
13.         {
14.
15.             ch = MoveNext();        //取下一个字符
16.
17.             if ((ch >= 'A') && (ch <= 'Z') || (ch == '!'))
18.
19.             {
20.
21.                 iBeginPos = Index;          //记录开始位置
22.
23.                 //表示可能是一个标签
24.
25.                 HtmlTag tag = GetTag();   //解析此 Tag
26.
27.                 if (tag != null)
28.
29.                 {
30.
31.                     //首先判断是否有文本
32.
33.                     if (m_CurrentText.Lenght > 0)
```

```
34.
35.                         {
36.
37.                             //将文本作为一个普通 Tag 入栈
38.
39.                             Stack.Push(new HtmlTextTag(m_CurrentText));
40.
41.                         }
42.
43.                         tag.BeginPos = iBeginPos;     //记录此 Tag 的开始位置
44.
45.                         Stack.Push(tag);                  //把 Tag 入栈
46.
47.                     }
48.
49.                 }
50.
51.
52.
53.                 ch = GetCurrentChar();
54.
55.                 if (ch == '/')
56.
57.                 {
58.
59.                     //可能是结束标签
60.
61.                     tagName = GetTagName();
62.
63.                     //从上到下查看 Stack，如果 Tag 中存在
64.
65.                     if (FindInStack(tagName))
66.
67.                     {
68.
69.                         //在栈中找到名为 tagName 的元素，则把找到的元素出栈
70.
71.                         PopTag(tagName);
72.
73.                     }
74.
75.                 }
76.
77.             }
78.
79.             else
80.
81.             {
82.
83.                 //对于<AAA>xxx</AAA>之间的文本 xxx，这里将作为 TextTag 来处理
84.
85.                 m_CurrentText.Append(GetCurrentChar());
86.
87.             }
88.
89.             //继续处理下一个字符
90.
91.             ch = MoveNext();
92.
93.         }
94.
95.
96.
97.         //解析完成以后，如果栈不空，那么把元素出栈，并把最后一次出栈的元素作为根
98.
99.         if (Stack.Count > 0)
```

```
100.
101.    {
102.
103.        HtmlTag tag = null;
104.
105.        while (Stack.Count > 0)
106.
107.        {
108.
109.            tag = Stack.Pop();
110.
111.            PopTag(tag);
112.
113.        }
114.
115.
116.
117.        //最后一个元素作为根元素
118.
119.        if (tag != null)
120.
121.        {
122.
123.            m_listRoot.Add(tag);
124.
125.        }
126.
127.    }
128.
129. }
130.
131.
132.
133. private void PopTag(HtmlTag tag)
134.
135. {
136.
137.    int iLevel = Stack.Count;
138.
139.
140.
141.    //找到了元素，把 iLevel 到 m_IMaxLevel 中所有的元素按照全部作为 tag 的子元素
142.
143.    for (int i = iLevel + 1; i < m_iMaxLevel; i++)
144.
145.    {
146.
147.        for (j = 0; j < HashTable[i].Count; j++)
148.
149.        {
150.
151.            tag.Children.Add(HashTable[i][j]);
152.
153.        }
154.
155.    }
156.
157.
158.
159.    //表示栈已经为空，那么最后一次出栈的 tag 将作为根
160.
161.    if (Stack.Count == 0)
162.
163.    {
164.
165.        m_listRoot.Add(tag);
166.
```

```
167.         }
168.
169. }
170.
171.
172.
173. private void PopTag(string tagName)
174.
175. {
176.
177.         /*
178.
179.          * 元素出栈的时候，首先需要把当前已经存在了的 HtmlTextTag 入栈
180.
181.          * 比如：<A>文本段 1<B>文本段 2</B>文本段 3</A>
182.
183.          * 在 Parse 中，当解析出<B>入栈前，需要先把"文本段 1"入栈
184.
185.          * 在这里，解析出了</B>结束标志
186.
187.          * 那么首先需要把"文本段 2"入栈。
188.
189.          * 解析出</A>则需要把"文本段 3'入栈。
190.
191.          * 这样才能够保证"文本段 1"和"文本段 3"成为<A>的子节点，而"文本段 2"作为<B>的子节点
192.
193.          */
194.
195.         if (m_CurrentText.Lenght > 0)
196.
197.         {
198.
199.             //将文本作为一个普通 Tag 入栈
200.
201.             Stack.Push(new HtmlTextTag(m_CurrentText));
202.
203.         }
204.
205.
206.
207.         HtmlTag tag = Stack.Pop();    //元素出栈
208.
209.         int iLevel = Stack.Count;     //记录栈元素数
210.
211.
212.
213.         while (tag.Name != tagName)
214.
215.         {
216.
217.             //将 tag 放入第 iLevel 层的 List 中
218.
219.             HashTable[iLevel].Add(tag);
220.
221.             tag = Stack.Pop();
222.
223.             iLevel = Stack.Count;
224.
225.         }
226.
227.
228.
229.         //元素出栈后续处理
230.
231.         PopTag(tag);
```

```
232.
233. }
234.
235.
236.
237. private HtmlTag GetTag()
238.
239. {
240.
241.     if ("如果发现是 < !--开头的元素") //则表示是注释
242.
243.     {
244.
245.         SkipComment();
246.
247.     }
248.
249.
250.
251.     HtmlTag tag = new HtmlTag();
252.
253.     tag.Name = GetTagName();
254.
255.     //这里的 Attribute 我将其作为 HashTable 类型，Hash[属性名]=属性值
256.
257.     tag.Attribute = GetTagAttribute();
258.
259.     return tag;
260.
261. }
```

二.WebKit DOM 代码解析流程

分析中。。。。。

VII.WebKit 之布局分析

一.基本概念

作为一个广受好评的浏览器引擎，其网页布局的质量(包括速度、效率、符合标准度等)往往是其关键，那么 WebKit 究竟是如何布局网页上的所有元素(包括滚动条、文字、图片、按钮、下拉框等)呢？其

主要数据结构及流程都包括哪些呢？其布局的基本概念及标准都有哪些呢？下面分别介绍 WebKit 对其实现及运 用。我们首先从关于布局的基本概念及标准的认识开始。

1.CSS 布局相关标准介绍

其实我们对要素的布局都有不同程度的了解如我们使用 Office 时经常使用对一段文字的居中、靠左等操作，复杂一点有设置编号及文字与图片的环绕对应关系等，其实布局的关键在于确定页面元素的显示位置及大小，而页面中主要包括有文字、图片、按钮等页面元素，为了有效的组织布局这些页面元素，一些专家学者经过多年的摸索，总结并设计了布局这些元素所涉及的一些规则及标准，这就是 CSS 标准。

其中 W3C 对其主要规则进行过具体描述，通过下面相关总结和汇总希望能对其主要要点有一定的认识与理解。

2.布局页面的基本概念

要在一块指定的画布(或窗口)上布局一些要素，往往需要按从上到下或从左到右(或从右到左)的规则来布局这些元素，而有些元素则可以包含其他元素，当作布局容器来使用。其中浏览网页的原生窗口就可看作一个布局容器的根。

由于页面内容的大小可能超过原生窗口提供的显示区域的大小，CSS 中称页面上当前显示出来的区域为 ViewPort，这个 ViewPort 相

对页面的原始位置可通过滚动条来调整。

CSS 标准中定义了 html 中的一些标签所对应的元素可当成容器使用的，以建立坐标定位所包含的元素如 p、div，CSS 中称这样的元素为 block-level 元素，相邻的 block-level 元素往往从上到下垂直排列。

而其他象 i、a、b、span 等标签及 text node 对应的元素则缺省为 inline-level 元素，inline-level 元素不能用来定位其他元素，但可以包含其他同为 inline- level 元素，相邻的 inline-level 元素，往往按照从左到右或从右到左的水平方向排列。

block-level 元素所包含的元素往往要么全为 block-level 元素要么全为 inline-level 元素，在一定条件下布局时可能会产生匿名 block-level 元素。

而页面上的每一个元素必须对应一个布局容器称之为 Containing Block，只有 block-level 元素可以成为 Containing Block。

一个 Containing Block 元素究竟包含哪些子元素或者某一元素的 Containing Block 元素究竟是谁，由其自身 position 属性及其在文档层次结构中所处的位置所确定，下一节会描述相关内容。

而每一个元素至少包含一个 Box 模型即由 margin、border、padding、content width/height 等属性所能描述的矩形区域。而这块区域相对于布局容器的坐标 top、left，往往由布局容器按照 block-flow、 inline-flow 等规则布局该元素时确定。

CSS 中将布局 block-level 元素的过程称为 block-flow; 将布局 inline-level 相关元素的过程称为 line-flow。

而 CSS 对 html 中诸如标签 frame、image、object、embed、form 等对应的元素称为 replaced 元素，它表示这些元素的内部布 局不由 Css 来定义，而由浏览器来实现，而这些元素从外部来看相当于 block-level 元素，但可通过设置 display: inline 将其从外部看 设为 inline-level 元素。

不同的 html 标签元素可以通过 display: inline 、display: block 、 display: inline-block 等方式来调整其缺省 block-level 或 inline-level 属性。

3.如何确定页面元素显示位置

一个 html 标签元素的 position 属性可以设置为 static、relative、fixed、absolute、inherit 等，所有元素缺省为 static，其 Containing Block 布局容器元素为最近的祖先 block-level 元素，其属性 left、top、right、bottom 不起作用。

position 属性为 relative 的元素同 static 属性元素一样，但其 left、top 等属性可以有效，其坐标相对于布局容器而言。

position 属性为 absolute 的元素的布局容器元素是最近的除了其属性不为 static 的祖先 block-level 元素。

position 属性为 fiexed 的元素的布局容器元素是往往是根布局容器，但其定位坐标需要根据 ViewPort 的位置作相应调整。

一旦确定了其Containing Block布局容器，同时结合其自身的block-level或inline-level特性,布局时根据block flow和inline flow规则就可确定其起始位置，其中 inline-level 元素可在其布局容器提供的区域内自动换行; 而 block-level 元素可在其布局容器提供的区域内自动换一个段落。

另外 float 属性为 left 或 right 元素较为特殊，则不遵守上面的规则，该元素让在其高度范围内的其他元素始终在其左边或右边。

4.如何确定页面元素大小

对于有定义其宽高的页面元素，则按照其定义的宽高来确定其大小，而对于象 text node 这样的 inline-level 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 Containing Block 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。

5.如何理解 z-index 的使用

页面元素 z-index 属性的出现，引入了页面元素三维布局的思路，提出分层的概念，具有同一 z-index 属性的所有元素按照上面提到的二维布局方式(确 定其位置及大小)来布局，而不同 z-index 所

代表的层的元素有可能被其他层的元素所覆盖。每一个页面元素只能处在一个 z-index 所对应的层中，所 有元素缺省 z-index 为 0。

6.总结

CSS 布局标准的内容相当多，有的还相当复杂，这里只是初步的了解其基本原则及要素，也未必在各种条件下都成立，希望能为我们能从 WebKit 代码去了解 WebKit 究竟是如何布局页面元素作一定准备而已，如果要想对 CSS 标准有更深入的具体理解，只有不断的练习及阅读理解 CSS 布局标准文档。

二.WebKit 主要布局框架

在有了对 CSS 网页布局标准及相关概念的认识之后，我们可以更加深入的理解 WebKit 究竟是如何实现其网页布局，同时实现对 CSS 布局标准的支持。

毕竟标准归标准，要高效的实现这些标准，不同的实现肯定有其不同的实现方式，就像不同的 Web 服务器对 HTTP 协议标准的实现有所不同一样，当然不同的实现也会增加一些自身特有的属性。

下面我们从数据结构的角度来了解 WebKit 中为实现网页布局所设计的主要类结构及其主要方法。

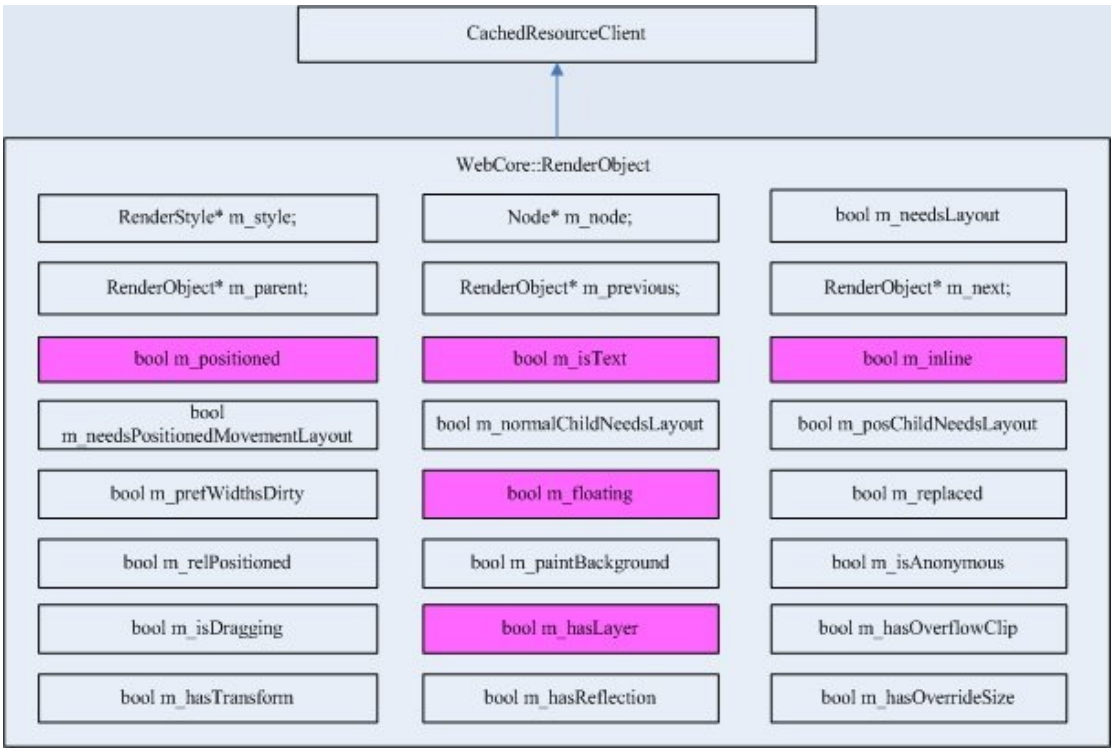
在我们编写网页及使用 JS 的时候，大概都知道 DOM 树及其主要构成，了解到 DOM 树的构建其实质是对一个 html 或 xml 文件的内容采取树结构的方式来组织及描述，不同的标签及其在文档中的位置决

定了其在整颗 DOM 树的地位及属性，针对具体 DOM 树的构成及不同树节点的描述，可以参考有关 DOM 的相关标准等。

也许对于 Render 树大家就不那么了解了，简单的说来，它是对 DOM 树更进一步的描述，其描述的内容主要与布局渲染等 CSS 相关属性如 left、top、width、height、color、font 等有关，因为不同的 DOM 树结点可能会有不同的布 局渲染属性，甚至布局时会按照标准动态生成一些匿名节点，所以为了更加方便的描述布局及渲染，WebKit 内核又生成一颗 Render 树来描述 DOM 树的 布局渲染等特性，当然 DOM 树与 Render 树不是一一对应，但可以相互关联，下面分别描述其主要节点:

1.基类 RenderObject

RenderObject 作为所有 Render 树节点的基类，完全类似与 DOM 树中的 Node 基类，它是构成 Render 树的基础，作用非比寻常，其中包含了构成 Render 树所可能涉及到的一些基本属性及方法，内容相当多，其主要数据成员及方法分别如下:



RenderObject 主要数据成员

其中成员 m_parent、m_previous、m_next 为构建 Render 树设置好关联基础；m_Node 则为 DOM 树中对应的节点；m_style 成员则描述该节点对应的各种 CSS 基本属性数据，下面会单独介绍；至于其他的诸如 m_positioned、m_isText、m_inline、m_floating、m_replaced 等则描述其特性，就像 CSS 标准对不同元素的属性分类定义一样，从字面上我们就可以上一节 WebKit 网页布局实现之基本概念及标准篇中可以找到它们这么定义的踪影。

成员 m_needsPositionedMovementLayout、m_normalChildNeedsLayout、m_posChildNeedsLayout、m_needsLayout 等主要用来描述该 RenderObject 是否确实需要重新布局；当一个新的 RenderObject 对象插入到 Render 树的时候，它

会设置其m_needsLayout属性为true，同时会根据该RenderObject对象在祖先RenderObject看来是一个positioned(拥有positiong:absolute或fixed属性)状态的孩子，如是则将相应祖先RenderObject对象的属性m_posChildNeedsLayout设置为true;如果是一个in-flow(positon:static或relative)状态的孩子，则将相应祖先RenderObject对象的属性m_normalChildNeedsLayout设置为true;

主要方法:

```
//与是否需要layout相关
bool needsLayout() const { return m_needsLayout || m_normalChildNeedsLayout
||m_posChildNeedsLayout; }
bool selfNeedsLayout() const { return m_needsLayout; }
bool posChildNeedsLayout() const { return m_posChildNeedsLayout; }
bool normalChildNeedsLayout() const { return m_normalChildNeedsLayout; }

//与基本属性相关
bool isFloating() const { return m_floating; }
bool isPositioned() const { return m_positioned; } // absolute or fixed positioning
bool isRelPositioned() const { return m_relPositioned; } // relative positioning
bool isText() const { return m_isText; }
bool isInline() const { return m_inline; } // inline object
bool isCompact() const { return style()->display() == COMPACT; } // compact object
bool isRunIn() const { return style()->display() == RUN_IN; } // run-in object
bool isDragging() const { return m_isDragging; }
bool isReplaced() const { return m_replaced; } // a "replaced" element (see CSS)

//与外部DOM关联相关
RenderView* view() const;
// don't even think about making this method virtual!
Node* element() const { return m_isAnonymous ? 0 : m_node; }
Document* document() const { return m_node->document(); }
void setNode(Node* node) { m_node = node; }
Node* node() const { return m_node; }

// RenderObject tree manipulation
//////////////////////////////////////////
```

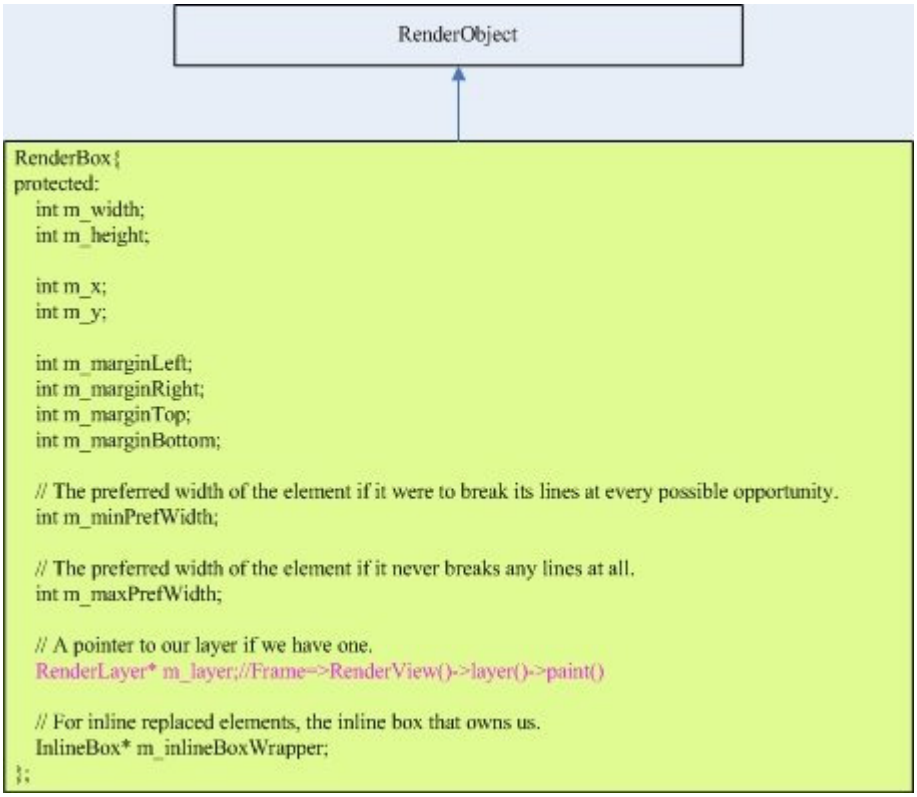
```
virtual bool canHaveChildren() const;
virtual bool isChildAllowed(RenderObject*, RenderStyle*) const { return true; }
virtual void addChild(RenderObject* newChild, RenderObject* beforeChild = 0);
virtual void removeChild(RenderObject*);
virtual bool createsAnonymousWrapper() const { return false; }

// raw tree manipulation
virtual RenderObject* removeChildNode(RenderObject*, bool fullRemove = true);
virtual void appendChildNode(RenderObject*, bool fullAppend = true);
virtual void insertChildNode(RenderObject* child, RenderObject* before, bool fullInsert = true);
// Designed for speed. Don't waste time doing a bunch of work like layer updating and repainting when we know that our
// change in parentage is not going to affect anything.
virtual void moveChildNode(RenderObject*);

virtual void paint(PaintInfo&, int tx, int ty);

/*
* This function should cause the Element to calculate its
* width and height and the layout of its content
*
* when the Element calls setNeedsLayout(false), layout() is no
* longer called during relayouts, as long as there is no
* style sheet change. When that occurs, m_needsLayout will be
* set to true and the Element receives layout() calls
* again.
*/
virtual void layout() = 0;
```

其中很多方法如 paint()、layout()等是虚拟的，不同的子类可以重载它；其中方法 container() 、containingBlock()、paint()、layout()很值得大家深入研究；总的说来 RenderObject 基类定义一些通用属性、方法，以便维护、布局、渲染 Render 树。

2. 子类 RenderBox

RenderBox 代表描述 CSS 标准中的 Box Model，它继承自

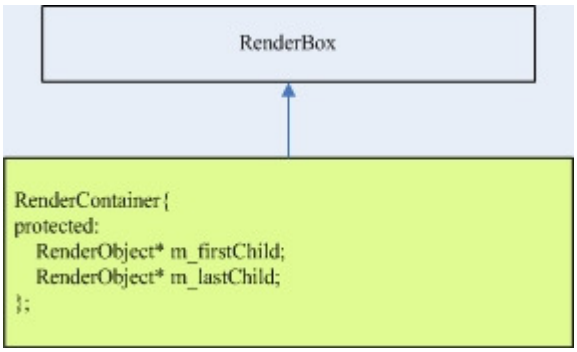
RenderObject，其主要重载了部分继承而来的方法。



RenderBox 主要数据成员

3. 子类 RenderContainer

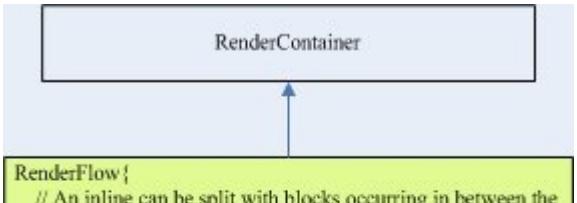
RenderContainer 类用来描述可以拥有子 RenderObject 成员的容器类，它继承自 RenderBox。其主要重载了 RenderObject 提供的维护 Render 树新增、删除树节点等方面的方法。



RenderContainer 主要数据成员

4.子类 RenderFlow

RenderFlow 主要用来描述 CSS 标准中提到的能进行 inline-flow、block-flow 相关处理的 Render 树结点，它继承自 RenderContainer；其主要方法包括在 flow 的过程中创建、关联匿名对象等。

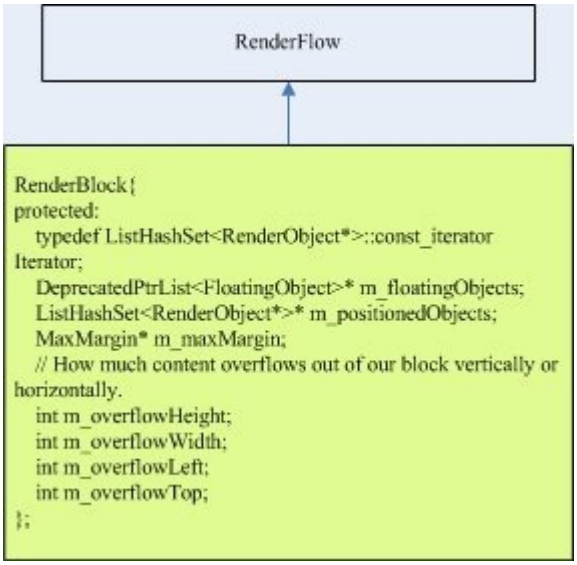



```
RenderFlow{
   // An inline can be split with blocks occurring in between the
inline content.
   // When this occurs we need a pointer to our next object.  We
can basically be
   // split into a sequence of inlines and blocks.  The
continuation will either be
   // an anonymous block (that houses other blocks) or it will be
an inline flow.
   RenderFlow* m_continuation;

protected:
   // For block flows, each box represents the root inline box for
a line in the
   // paragraph.
   // For inline flows, each box represents a portion of that
inline.
   InlineFlowBox* m_firstLineBox;
   InlineFlowBox* m_lastLineBox;

   mutable short m_lineHeight;
   // These bitfields are moved here from subclasses to pack
them together
   // from RenderBlock
   bool m_childrenInline : 1;
   bool m_firstLine : 1;
   bool m_topMarginQuirk : 1;
   bool m_bottomMarginQuirk : 1;
   bool m_hasMarkupTruncation : 1;
   unsigned m_selectionState : 3; // SelectionState
   bool m_hasColumns : 1;

   // from RenderInline
   bool m_isContinuation : 1; // Whether or not we're a
continuation of an inline.
};
```

RenderFlow 主要数据成员

5.子类 RenderBlock

RenderBlock 代表 CSS 标准中的 block-level 元素，它继承自 RenderFlow。



RenderBlock 主要数据成员

它维护了一组由它定位的 positioned 树节点，以及有关 overflow 方面的设置。其主要重载了 RenderObject 继承下来的 layout、paint 等方法。

因为 html 中的 body、div、p 等标签对应 RenderBlock 类对象，其在 Render 树具有非常重要的地位，其 layout、paint 等方法的实现，往往是 WebKit 整个布局、渲染处理的发起中心，内容比较多并且复杂，以后有机会详解。

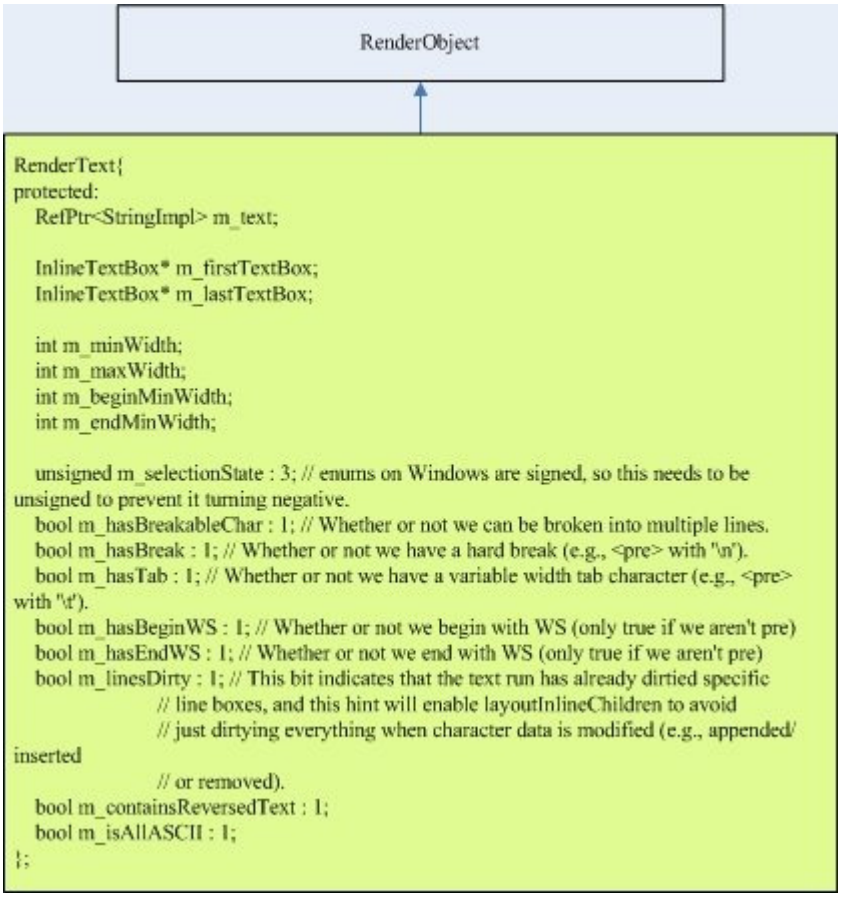
6.子类 RenderInline

RenderInline 代表 inline-level 元素，其继承自 RenderFlow，主要重载了 RenderObject 关于 inline-flow 方面处理

的方法，提供了 splitFlow、splitInlines 等处理自动换行的方法。

7.子类 RenderText

RenderText 代表对 html 中 Text node 对应的 Render 树节点，它直接继承自 RenderObject。

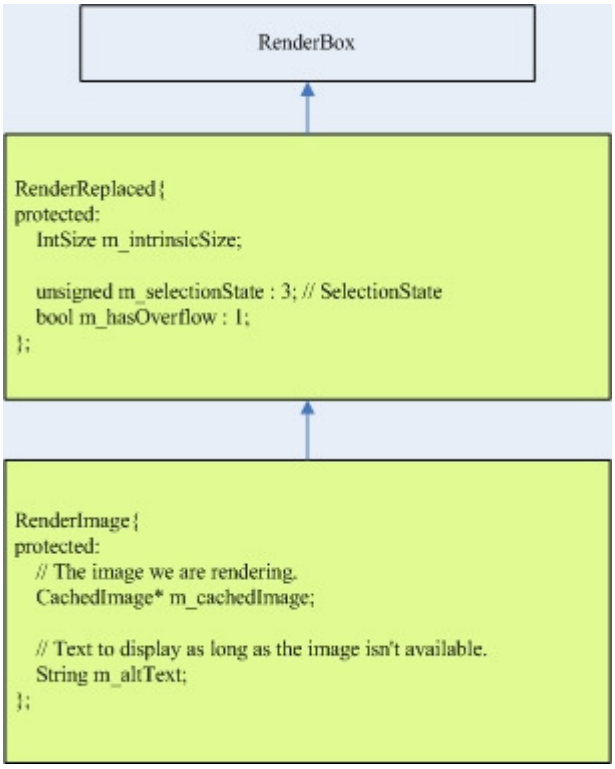


RenderText 主要数据成员

它提供关于处理文字方面如显示文字、行高计算、整个 Text node 对应的宽度等；它没有重载 layout 方法，因为它自身的定位往往由 RenderBlock、RenderInline 父对象来处理；

8.子类 RenderImage

RenderImage 代表 html 中 img 标签对应的树节点，它继承自 RenderBox。

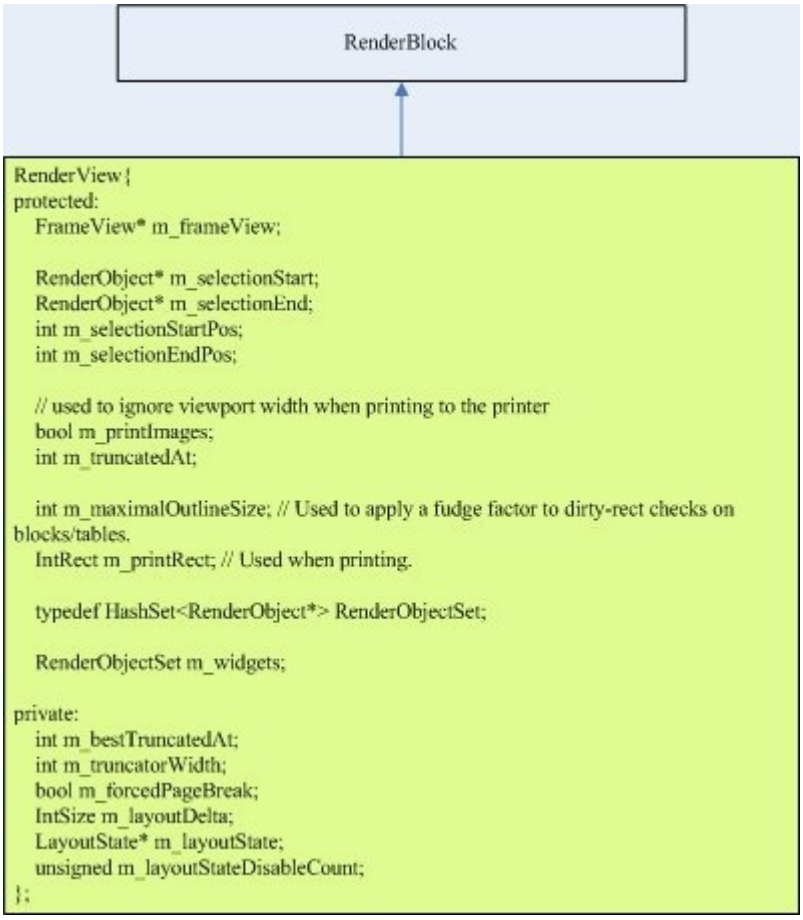


RenderImage 继承关系及主要数据成员

其主要提供关于图片显示、大小设置等方面的处理，其中 paintReplaced 方法将其图片显示出来;

9.子类 RenderView

RenderView 对应整个 html 文档对象的树节点，可看成是 Render 树的根，它继承自 RenderBlock。



RenderView 主要数据成员

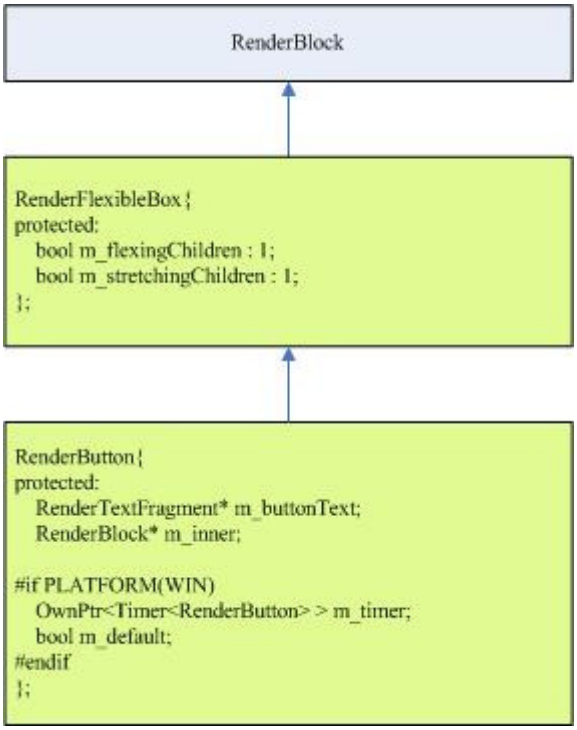
其中 m_frameview 成员对应整个文档对应的 FrameView，而 m_widgets 则包括了该文档可能包含的 plugin 插件等对应的 Render 树节点。

RenderView 对象作为 Render 树的根，它往往随着 Document 对象的创建而创建，它的 layout、paint 方法的发起往往是整颗 Render 树布局、渲染处理的开始。其中也包含了对选择处理。

10. 子类 RenderButton

RenderButton 代表 html 中 input 标签 type 为 button 时对应的 Render 树节点，它直接继承自 RenderFlexibleBox。

RenderFlexibleBox 代表能按居中、左对齐、右对齐等水平或垂直方向布局子节点的树节点。



RenderButton 主要数据成员

其中 m_buttonText 为 button 上的文字对应的树节点，而 m_inner 为添加 m_buttonText 时创建的匿名对象，以便于居中等处理等。这些成员的创建来自于方法 updateFromElement;

```
void RenderButton::updateFromElement()
{
// If we're an input element, we may need to change our button text.
if (element()->hasTagName(inputTag)) {
HTMLInputElement* input = static_cast(element());
String value = input->valueWithDefault();
setText(value);
}
}
void RenderButton::setText(const String& str)
{
.....
m_buttonText = new (renderArena()) RenderTextFragment(document(), str.impl());
m_buttonText->setStyle(style());
```

```
addChild(m_buttonText);
.....................
}
void RenderButton::addChild(RenderObject* newChild, RenderObject* beforeChild)
{
if (!m_inner) {
// Create an anonymous block.
m_inner = createAnonymousBlock();
m_inner->style()->setBoxFlex(1.0f);
RenderFlexibleBox::addChild(m_inner);
}
m_inner->addChild(newChild, beforeChild);
}
```

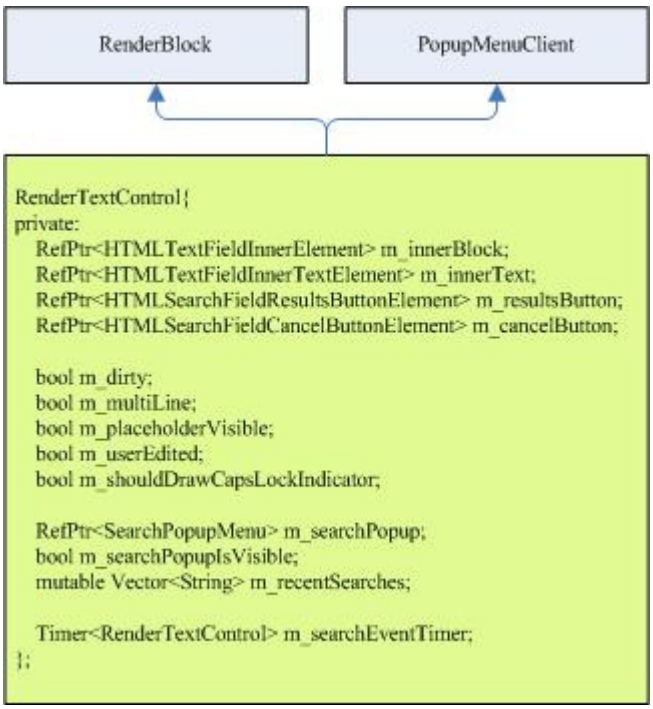
在缺省的 html.css 中对应 button 的 css 属性如下：

```
input[type="button"] {
-webkit-appearance: push-button;
white-space: pre
}
input[type="button"]{
-webkit-box-align: center;
text-align: center;
cursor: default;
color: ButtonText;
padding: 2px 6px 3px 6px;
border: 2px outset ButtonFace;
background-color: ButtonFace;
-webkit-box-sizing: border-box
}
```

这些css属性通过CSSStyleSelector::applyProperty方法来设定其成员 m_RenderStyle 对应的值，其中包含 m_style->setAppearance(PushButtonAppearance)。尤其值得关注，其初步决定了 button 是如何画出来的。

11.子类 RenderTextControl

RenderTextControl 代表 html 中 input 标签 type 为 text 或 textarea 标签对应的 Render 树节点，它直接继承自 RenderBlock;



RenderTextControl 主要数据成员

其中成员 m_multiLine 以描述是 textarea 或 text input; m_innerText 为其中包括的文字对应的树节点; 当作搜索按钮时 m_cancelButton/m_resultsButton 为对应的树节点; 这些成员的创建来自于方法 updateFromElement;

```
void RenderTextControl::updateFromElement()
{
HTMLFormControlElement* element = static_cast(node());

createSubtreeIfNeeded();

.....

m_innerText->renderer()->style()->setUserModify(element->isReadOnlyControl() ||
element->disabled() ? READ_ONLY : READ_WRITE_PLAINTEXT_ONLY);

if ((!element->valueMatchesRenderer() || m_multiLine) && !m_placeholderVisible) {
String value;
if (m_multiLine)
value = static_cast(element)->value();
```

```
else
value = static_cast(element)->value();
if (value.isNull())
value = "";
else
value = value.replace('₩₩', backslashAsCurrencySymbol());
if (value != text() || !m_innerText->hasChildNodes()) {
if (value != text()) {
if (Frame* frame = document()->frame())
frame->editor()->clearUndoRedoOperations();
}
ExceptionCode ec = 0;
m_innerText->setInnerText(value, ec);
if (value.endsWith("₩n") || value.endsWith("₩r"))
m_innerText->appendChild(new HTMLBRElement(document()), ec);
m_dirty = false;
m_userEdited = false;
}
.....
}
.....
}
void RenderTextControl::createSubtreeIfNeeded()
{
.....
if (!m_innerText) {
m_innerText = new HTMLTextFieldInnerTextElement(document(), m_innerBlock ? 0 :
node());
RenderTextControlInnerBlock* textBlockRenderer = new (renderArena())
RenderTextControlInnerBlock(m_innerText.get());
m_innerText->setRenderer(textBlockRenderer);
m_innerText->setAttached();
m_innerText->setInDocument(true);

RenderStyle* parentStyle = style();
if (m_innerBlock)
parentStyle = m_innerBlock->renderer()->style();
RenderStyle* textBlockStyle = createInnerTextStyle(parentStyle);
textBlockRenderer->setStyle(textBlockStyle);

// Add text block renderer to Render tree
if (m_innerBlock) {
m_innerBlock->renderer()->addChild(textBlockRenderer);
ExceptionCode ec = 0;
```

```
// Add text block to the DOM
m_innerBlock->appendChild(m_innerText, ec);
} else
RenderBlock::addChild(textBlockRenderer);
}
.....
}
```

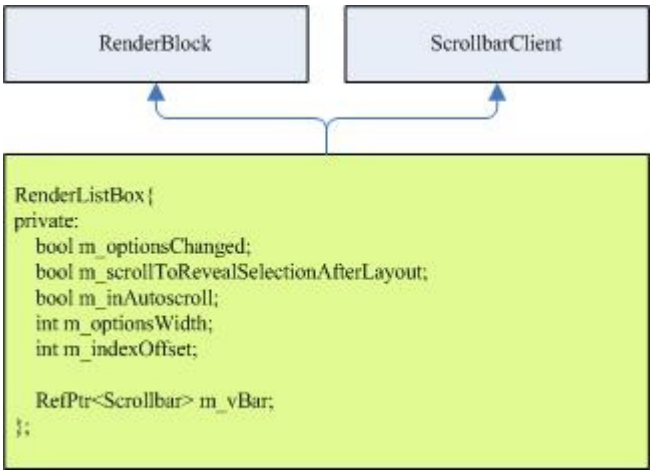
在缺省的 html.css 中对应标签的 css 属性如下：

```
textarea {
-webkit-appearance: textarea;
background-color: white;
border: 1px solid;
-webkit-rtl-ordering: logical;
-webkit-user-select: text;
-webkit-box-orient: vertical;
resize: auto;
cursor: auto;
}
input, input[type="password"], input[type="search"], isindex {
-webkit-appearance: textfield;
padding: 1px;
background-color: white;
border: 2px inset;
-webkit-rtl-ordering: logical;
-webkit-user-select: text;
cursor: auto;
}
```

其中-webkit-appearance 属性分别为 textarea、textfield;

12.子类 RenderListBox

RenderListBox 代表 html 中 select 标签对应的 Render 树节点，它直接继承自 RenderBlock;



RenderListBox 主要数据成员

其相关成员同样通过方法 updateFromElement 来设置初值;

在缺省的 html.css 中对应标签的 css 属性如下:

```
select {
-webkit-appearance: menulist;
-webkit-box-sizing: border-box;
-webkit-box-align: center;
border: 1px solid;
.....................................................
}
```

13.子类 RenderTheme

RenderTheme 在 html 标签中没有对应的页面元素，其作用主要用于如何渲染按钮、输入框、列表框等，其实现往往有一定平台相关性。

```
RenderTheme{
Public:
.....
   bool paint(RenderObject*, const RenderObject::PaintInfo&, const
IntRect&);
.....
protected:
   // Methods for each appearance value.
   virtual void adjustCheckboxStyle(CSSStyleSelector*, RenderStyle*,
Element*) const;
   virtual bool paintCheckbox(RenderObject*, const
RenderObject::PaintInfo&, const IntRect&) { return true; }
   virtual void setCheckboxSize(RenderStyle*) const { }

   virtual void adjustRadioStyle(CSSStyleSelector*, RenderStyle*,
Element*) const;
   virtual bool paintRadio(RenderObject*, const
RenderObject::PaintInfo&, const IntRect&) { return true; }
   virtual void setRadioSize(RenderStyle*) const { }

   virtual void adjustButtonStyle(CSSStyleSelector*, RenderStyle*,
Element*) const;
   virtual bool paintButton(RenderObject*, const
RenderObject::PaintInfo&, const IntRect&) { return true; }
   virtual void setButtonSize(RenderStyle*) const { }

   virtual void adjustTextFieldStyle(CSSStyleSelector*, RenderStyle*,
Element*) const;
   virtual bool paintTextField(RenderObject*, const
RenderObject::PaintInfo&, const IntRect&) { return true; }

   virtual void adjustTextAreaStyle(CSSStyleSelector*, RenderStyle*,
Element*) const;
   virtual bool paintTextArea(RenderObject*, const
RenderObject::PaintInfo&, const IntRect&) { return true; }

   virtual void adjustMenuListStyle(CSSStyleSelector*, RenderStyle*,
Element*) const;
   virtual bool paintMenuList(RenderObject*, const
RenderObject::PaintInfo&, const IntRect&) { return true; }
.....
private:
   mutable Color m_activeSelectionColor;
   mutable Color m_inactiveSelectionColor;
};
```

RenderTheme 主要数据成员及方法

RenderTheme 往往提供一个接口，不同的图形库对其中不同的方法如 paintbutton、paintcheckbox、painttextfield 等进行了实现；其中 paint 方法则根据 appearance 属性的不同以分别调用不同的 paintxxx 方法，其示例代码如下：

```
bool RenderTheme::paint(RenderObject* o, const RenderObject::PaintInfo& paintInfo,
const IntRect& r)
```

```
{
.....
if (paintInfo.context->paintingDisabled())
return false;
// Call the appropriate paint method based off the appearance value.
switch (o->style()->appearance()) {
case CheckboxAppearance:
return paintCheckbox(o, paintInfo, r);
case RadioAppearance:
return paintRadio(o, paintInfo, r);
case PushButtonAppearance:
case SquareButtonAppearance:
case DefaultButtonAppearance:
case ButtonAppearance:
return paintButton(o, paintInfo, r);
case MenulistAppearance:
return paintMenuList(o, paintInfo, r);
break;
.....
default:
break;
}
return true; // We don't support the appearance, so let the normal background/border
paint.
}
```

其中的 appearance 就是上面 RenderButton 、RenderTextControl、RenderListBox 中提到的属性，至于 html 中涉及到的类似标签或属性如 radio、checkbox 等等，其相关代码基本类似，至于不同的平台如 Qt、Gtk、Win、Mac 等究竟是如何 画按钮、下拉框、列表框、多选框、单选框等等，则需详细参考 RenderThemeQt/RenderThemeGtk/RenderThemeWin /RenderThemeMac 等中的实现。

通过上述的了解我们应该对 html 中 form 标签内的输入框、按钮、下拉框等实现有了一定的认识。

14. 子类 RenderTable、RenderTableRow、RenderTableCol、RenderTableCell

这一组子类主要对应与 html 中 table 标签相关的树节点;

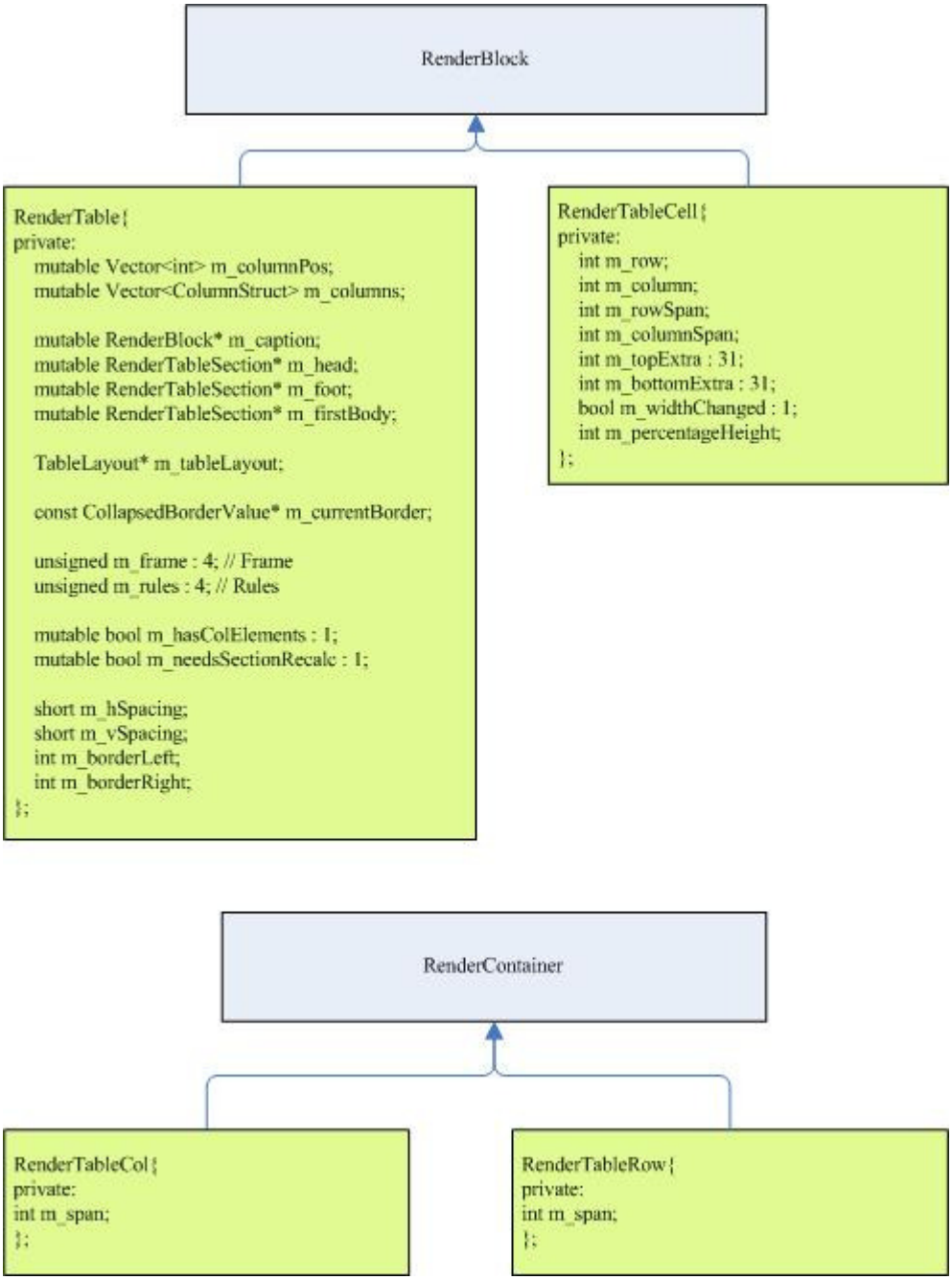


Table 标签相关类主要数据成员

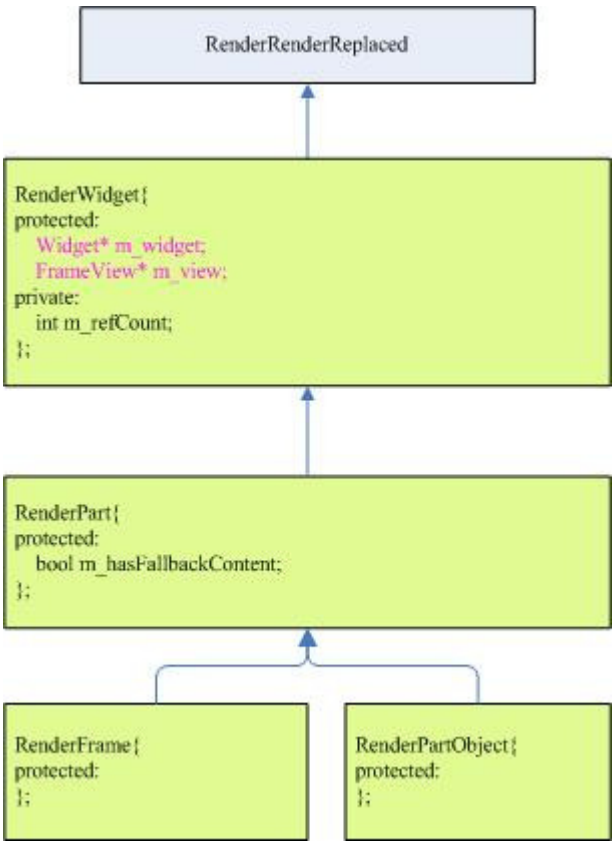
RenderTable 主要通过 addChild 方法来维护对 RenderTableCell、RenderTableCol、RenderTableRow 等对象的管理及维护。

```
void RenderTableCell::updateFromElement()
{
Node* node = element();
if (node && (node->hasTagName(tdTag) || node->hasTagName(thTag))) {
HTMLTableCellElement* tc = static_cast(node);
int oldRSpan = m_rowSpan;
int oldCSpan = m_columnSpan;

m_columnSpan = tc->colSpan();
m_rowSpan = tc->rowSpan();
if ((oldRSpan != m_rowSpan || oldCSpan != m_columnSpan) && style() && parent())
{
setNeedsLayoutAndPrefWidthsRecalc();
if (section())
section()->setNeedsCellRecalc();
}
}
}

void RenderTableCol::updateFromElement()
{
int oldSpan = m_span;
Node* node = element();
if (node && (node->hasTagName(colTag) || node->hasTagName(colgroupTag))) {
HTMLTableColElement* tc = static_cast(node);
m_span = tc->span();
} else
m_span = !(style() && style()->display() == TABLE_COLUMN_GROUP);
if (m_span != oldSpan && style() && parent())
setNeedsLayoutAndPrefWidthsRecalc();
}
```

RenderTableRow 通过方法 layout 和 paint 方法来布局管理 RenderTableCell 对象。这一组子类主要实现人们熟知的表格布局，具体的实现可具体参考相关类实现。

15.子类 RenderFrame

RenderFrame 代表 html 中标签 frame 对应的 Render 树节点，其继承关系如下:



RenderFrame 类继承关系

其中属性 m_widget、m_view 代表 frame 对应的 widget 及 frameview，通过其中 setwidget 方法来设置 m_widget 属性，m_view 属性则在对象创建的时候设置为当前 document 对应的 frameview。

其中 html 中的 embed/object 插件标签对应的 Render 树节点为 RenderPartObject 对象。

三.CSS 属性的描述

1.RenderStyle 类

RenderObject 对象的 m_style 成员为 RenderStyle 类对象，它往往用来描述一个 RenderObject 所可能涉及的 CSS 属性数据(如 left、 top、align、color、font 等等)，其数据成员往往对应于 CSS 中定义的所有属性项，内容非常的庞杂，简单的说来就是将 CSS 标准中的所有 属性按照一定分类定义到一个数据结构中。

2.RenderStyle 类主要方法

为了获取、设置 CSS 属性所对应的值，RenderStyle 类提供了所有的获取、设置 CSS 属性的方法如:

```
void setDisplay(EDisplay v) { noninherited_flags._effectiveDisplay = v; }
void setOriginalDisplay(EDisplay v) { noninherited_flags._originalDisplay = v; }
void setPosition(EPosition v) { noninherited_flags._position = v; }
void setFloating(EFloat v) { noninherited_flags._floating = v; }

void setLeft(Length v) { SET_VAR(surround,offset.left,v) }
void setRight(Length v) { SET_VAR(surround,offset.right,v) }
void setTop(Length v) { SET_VAR(surround,offset.top,v) }
void setBottom(Length v){ SET_VAR(surround,offset.bottom,v) }

void setWidth(Length v) { SET_VAR(box,width,v) }
void setHeight(Length v) { SET_VAR(box,height,v) }
```

等等。。。。

四.RenderObject 及子类对象的生成

1.CSSParser

CSSParser 类顾名思义，主要用来解析文本中各种 CSS 属性，并

且有效的组织在一个 RenderStyle 对象中。

其主要方法 parseValue、applyProperty 的部分代码示例如下:

```
bool CSSParser::parseValue(int propId, bool important)
{
.....
case CSSPropertyFloat:
// left | right | none | inherit + center for buggy CSS
if (id == CSSValueLeft || id == CSSValueRight ||
id == CSSValueNone || id == CSSValueCenter)
valid_primitive = true;
break;

case CSSPropertyClear: // none | left | right | both | inherit
if (id == CSSValueNone || id == CSSValueLeft ||
id == CSSValueRight|| id == CSSValueBoth)
valid_primitive = true;
break;

case CSSPropertyWebkitBoxAlign:
if (id == CSSValueStretch || id == CSSValueStart || id == CSSValueEnd ||
id == CSSValueCenter || id == CSSValueBaseline)
valid_primitive = true;
break;
.....
case CSSPropertyWebkitBoxPack:
if (id == CSSValueStart || id == CSSValueEnd ||
id == CSSValueCenter || id == CSSValueJustify)
valid_primitive = true;
break;
.....
}

void CSSStyleSelector::applyProperty(int id, CSSValue *value)
{
case CSSPropertyOpacity:
HANDLE_INHERIT_AND_INITIAL(opacity, Opacity)
if (!primitiveValue || primitiveValue->primitiveType() !=
CSSPrimitiveValue::CSS_NUMBER)
return; // Error case.
// Clamp opacity to the range 0-1
m_style->setOpacity(min(1.0f, max(0.0f, primitiveValue->getFloatValue())));
return;
```

```
case CSSPropertyWebkitBoxAlign:
{
HANDLE_INHERIT_AND_INITIAL(boxAlign, BoxAlign)
if (!primitiveValue)
return;
EBoxAlignment boxAlignment = *primitiveValue;
if (boxAlignment != BJUSTIFY)
m_style->setBoxAlign(boxAlignment);
return;
}
.....
}
```

2. CSSStyleSelector 类

CSSStyleSelector 类其作用是基于所有用户的 stylesheets 集合为一个给定的 DOM Element 创建出其对应的 RenderStyle 对象。其主要功能由方法 RenderStyle* styleForElement(Element*, RenderStyle* parentStyle = 0, bool allowSharing = true, bool resolveForRootDefault = false)来实现。

3. 构建 Render 树

在构建 DOM 树的过程中，Dom Element 对象创建完后，往往通过 attach 方法来创建 RenderObject 对象，进而构建 Render 树。

其基本实现流程如下:

```
void Element::attach()=>createRendererIfNeeded()=>createRenderer ;

RenderObject* Element::createRenderer(RenderArena* arena, RenderStyle* style)
{
if (document()->documentElement() == this && style->display() == NONE) {
// Ignore display: none on root elements. Force a display of block in that case.
RenderBlock* result = new (arena) RenderBlock(this);
if (result)
result->setAnimatableStyle(style);
```

```
return result;
}
return RenderObject::createObject(this, style);
}

RenderObject* RenderObject::createObject(Node* node, RenderStyle* style)
{
Document* doc = node->document();
RenderArena* arena = doc->renderArena();

const ContentData* contentData = style->contentData();
if (contentData && !contentData->m_next && contentData->m_type ==
CONTENT_OBJECT && doc != node) {
RenderImageGeneratedContent* image = new (arena)
RenderImageGeneratedContent(node);
image->setStyle(style);
if (StyleImage* styleImage = contentData->m_content.m_image)
image->setStyleImage(styleImage);
return image;
}

RenderObject* o = 0;

switch (style->display()) {// 往往在 CSSStyleSelector::styleForElement 或
CSSStyleSelector::adjustRenderStyle 时//调用 setDisplay()以确定其 display 属性。
case NONE:
break;
case INLINE:
o = new (arena) RenderInline(node);
break;
case BLOCK:
o = new (arena) RenderBlock(node);
break;
case INLINE_BLOCK:
o = new (arena) RenderBlock(node);
break;
case LIST_ITEM:
o = new (arena) RenderListItem(node);
break;
case RUN_IN:
case COMPACT:
o = new (arena) RenderBlock(node);
break;
case TABLE:
```

```
case INLINE_TABLE:
o = new (arena) RenderTable(node);
break;
case TABLE_ROW_GROUP:
case TABLE_HEADER_GROUP:
case TABLE_FOOTER_GROUP:
o = new (arena) RenderTableSection(node);
break;
case TABLE_ROW:
o = new (arena) RenderTableRow(node);
break;
case TABLE_COLUMN_GROUP:
case TABLE_COLUMN:
o = new (arena) RenderTableCol(node);
break;
case TABLE_CELL:
o = new (arena) RenderTableCell(node);
break;
case TABLE_CAPTION:
o = new (arena) RenderBlock(node);
break;
case BOX:
case INLINE_BOX:
o = new (arena) RenderFlexibleBox(node);
break;
}
return o;
}
```

这样就不同的 DOM 树节点结合不同的显示属性，创建出不同的 RenderObject 子类对象，进而形成一个 Render 树。

五. Render 树与 RenderLayer 树

1. 构建 Render 树的基本实现流程

实现函数:

```
void Element::attach()=>createRendererIfNeeded()=>createRenderer
```

让我们看看 createRendererIfNeeded()，以更深入的了解是如何构建 Render 树。

```
void Node::createRendererIfNeeded()
{
if (!document()->shouldCreateRenderers())
return;
Node *parent = parentNode();
RenderObject *parentRenderer = parent->renderer();
if (parentRenderer && parentRenderer->canHaveChildren()
#if ENABLE(SVG)
&& parent->childShouldCreateRenderer(this)
#endif
) {
RenderStyle* style = styleForRenderer(parentRenderer);
if (rendererIsNeeded(style)) {
if (RenderObject* r = createRenderer(document()->renderArena(), style)) {
if (!parentRenderer->isChildAllowed(r, style))
r->destroy();
else {
setRenderer(r);
renderer()->setAnimatableStyle(style);
parentRenderer->addChild(renderer(), nextRenderer());
}
}
}
style->deref(document()->renderArena());
}
}

RenderStyle* Element::styleForRenderer(RenderObject* parentRenderer)
{
return document()->styleSelector()->styleForElement(this);
}

void RenderObject::setAnimatableStyle(RenderStyle* style)
{
if (!isText() && m_style && style)
style = animation()->updateImplicitAnimations(this, style);
setStyle(style);
}
```

从 createRendererIfNeeded 中我们可以了解到创建完 RenderObject 子类对象后，会为其设置 RenderStyle 属性，然后在

该 DOM Node 的父节点对应的 RenderObject 中添加刚新建的 RenderObject，这样以构建Render 树。

但是在 setStyle 的过程中可能会调用 createAnonymousFlow 或 createAnonymousBlock 来创建匿名对象,其中 RenderBox 的 setStyle 方法如下:

```
void RenderBox::setStyle(RenderStyle* newStyle)
{
bool wasFloating = isFloating();
bool hadOverflowClip = hasOverflowClip();
RenderStyle* oldStyle = style();
if (oldStyle)
oldStyle->ref();

RenderObject::setStyle(newStyle);
.....
setInline(newStyle->isDisplayInlineType());

switch (newStyle->position()) {
case AbsolutePosition:
case FixedPosition:
setPositioned(true);
break;
default:
setPositioned(false);

if (newStyle->isFloating())
setFloating(true);

if (newStyle->position() == RelativePosition)
setRelPositioned(true);
}

// We also handle and , whose overflow applies to the viewport.
if (!isRoot() && (!isBody() || !document()->isHTMLDocument()) && (isRenderBlock()
|| isTableRow() || isTableSection())) {
// Check for overflow clip.
if (newStyle->overflowX() != OVISIBLE) {
if (!hadOverflowClip)
// Erase the overflow
```

```
repaint();
setHasOverflowClip();
}
}
.....
if (requiresLayer()) {
if (!m_layer) {
if (wasFloating && isFloating())
setChildNeedsLayout(true);
m_layer = new (renderArena()) RenderLayer(this);
setHasLayer(true);
m_layer->insertOnlyThisLayer();
if (parent() && !needsLayout() && containingBlock())
m_layer->updateLayerPositions();
}
} else if (m_layer && !isRoot() && !isRenderView()) {
.....
}
.....
}

bool RenderObject::requiresLayer()
{
return isRoot() || isPositioned() || isRelPositioned() || isTransparent() ||
hasOverflowClip() || hasTransform() || hasMask() || hasReflection();
}

bool RenderObject::isRoot() const { return document()->documentElement() ==
node(); }
bool RenderObject::isPositioned() const { return m_positioned; } // absolute or fixed
positioning
bool RenderObject::isRelPositioned() const { return m_relPositioned; } // relative
positioning
bool RenderObject::isTransparent() const { return style()->opacity()
<>RenderObject::hasOverflowClip() const { return m_hasOverflowClip; }
bool RenderObject::hasTransform() const { return m_hasTransform; }
bool RenderObject::hasMask() const { return style() && style()->hasMask(); }
bool RenderObject::hasReflection() const { return m_hasReflection; }
```

通过上面的了解我们知道在 setStyle 时符合一定条件的 RenderObject 会创建 RenderLayer 对象，那么究竟什么是

RenderLayer 类，其有什么作用，下面作初步的介绍。

2.RenderLayer 类分析

RenderLayer 类其实是一个非常复杂并且很重要的类，其主要数据成员如下:

```
RenderLayer{
protected:
    RenderObject* m_object;

    RenderLayer* m_parent;
    RenderLayer* m_previous;
    RenderLayer* m_next;
    RenderLayer* m_first;
    RenderLayer* m_last;

    IntRect m_repaintRect; // Cached repaint rects. Used by layout.
    IntRect m_outlineBox;
    // Our current relative position offset.
    int m_relX;int m_relY;

    // Our (x,y) coordinates are in our parent layer's coordinate space.
    int m_x;    int m_y;

    // The layer's width/height
    int m_width;
    int m_height;

    // Our scroll offsets if the view is scrolled.
    int m_scrollX;
    int m_scrollY;
    int m_scrollOriginX;
    int m_scrollLeftOverflow;
    // The width/height of our scrolled area.
    int m_scrollWidth;
    int m_scrollHeight;
// For layers with overflow, we have a pair of scrollbars.
    RefPtr<Scrollbar> m_hBar;
    RefPtr<Scrollbar> m_vBar;

    // Keeps track of whether the layer is currently resizing, so events can
cause resizing to start and stop.
    bool m_inResizeMode;

    // For layers that establish stacking contexts, m_posZOrderList holds a
sorted list of all the
    // descendant layers within the stacking context that have z-indices of 0 or
greater
    // (auto will count as 0).  m_negZOrderList holds descendants within our
stacking context with negative
    // z-indices.
    Vector<RenderLayer*>* m_posZOrderList;
    Vector<RenderLayer*>* m_negZOrderList;

    // This list contains child layers that cannot create stacking contexts.  For
now it is just
    // overflow layers, but that may change in the future.
    Vector<RenderLayer*>* m_overflowList

.....................................................
}
```

RenderLayer 类主要数据成员

RenderLayer 类主要与处理分层布局、渲染页面元素等相关如处理 z-index、opacity 等，只有符合一个条件的 RenderObject 才会创建 RenderLayer 对象，并且将这些 RenderLayer 对象组织成一颗树。

3. 构建 RenderLayer 树

通过方法 insertOnlyThisLayer 来组织这颗 RenderLayer 树。

```
void RenderLayer::insertOnlyThisLayer()
{
if (!m_parent && renderer()->parent()) {
// We need to connect ourselves when our renderer() has a parent.
// Find our enclosingLayer and add ourselves.
RenderLayer* parentLayer = renderer()->parent()->enclosingLayer();
RenderLayer* beforeChild = parentLayer->reflectionLayer() != this ?
renderer()->parent()->findNextLayer(parentLayer, renderer()) : 0;
if (parentLayer)
parentLayer->addChild(this, beforeChild);
}

// Remove all descendant layers from the hierarchy and add them to the new position.
for (RenderObject* curr = renderer()->firstChild(); curr; curr = curr->nextSibling())
curr->moveLayers(m_parent, this);

// Clear out all the clip rects.
clearClipRects();
}

void RenderLayer::addChild(RenderLayer* child, RenderLayer* beforeChild)
{
RenderLayer* prevSibling = beforeChild ? beforeChild->previousSibling() : lastChild();
if (prevSibling) {
child->setPreviousSibling(prevSibling); prevSibling->setNextSibling(child);
} else
setFirstChild(child);

if (beforeChild) { beforeChild->setPreviousSibling(child);
child->setNextSibling(beforeChild); } else setLastChild(child); child->setParent(this);

if (child->isOverflowOnly())
dirtyOverflowList();

if (!child->isOverflowOnly() || child->firstChild()) {
// Dirty the z-order list in which we are contained. The stackingContext() can be null in
the
// case where we're building up generated content layers. This is ok, since the lists will
```

```
start
// off dirty in that case anyway.
RenderLayer* stackingContext = child->stackingContext();
if (stackingContext)
stackingContext->dirtyZOrderLists();
}

child->updateVisibilityStatus();
if (child->m_hasVisibleContent || child->m_hasVisibleDescendant)
childVisibilityChanged(true);
}

static void addLayers(RenderObject* obj, RenderLayer* parentLayer, RenderObject*&
newObject,
RenderLayer*& beforeChild)
{
if (obj->hasLayer()) {
if (!beforeChild && newObject) {
// We need to figure out the layer that follows newObject. We only do
// this the first time we find a child layer, and then we update the
// pointer values for newObject and beforeChild used by everyone else.
beforeChild = newObject->parent()->findNextLayer(parentLayer, newObject);
newObject = 0;
}
parentLayer->addChild(obj->layer(), beforeChild);
return;
}

for (RenderObject* curr = obj->firstChild(); curr; curr = curr->nextSibling())
addLayers(curr, parentLayer, newObject, beforeChild);
}

void RenderObject::addLayers(RenderLayer* parentLayer, RenderObject* newObject)
{
if (!parentLayer)
return;

RenderObject* object = newObject;
RenderLayer* beforeChild = 0;
WebCore::addLayers(this, parentLayer, object, beforeChild);
}

void RenderObject::removeLayers(RenderLayer* parentLayer)
{
```

```
if (!parentLayer)
return;

if (hasLayer()) {
parentLayer->removeChild(layer());
return;
}

for (RenderObject* curr = firstChild(); curr; curr = curr->nextSibling())
curr->removeLayers(parentLayer);
}

void RenderObject::moveLayers(RenderLayer* oldParent, RenderLayer* newParent)
{
if (!newParent)
return;

if (hasLayer()) {
if (oldParent)
oldParent->removeChild(layer());
newParent->addChild(layer());
return;
}

for (RenderObject* curr = firstChild(); curr; curr = curr->nextSibling())
curr->moveLayers(oldParent, newParent);
}
```

通过上面一组方法我们可以了解到拥有 RenderLayer 对象的 RenderObject 对象，按照 Render 树中最近的原则将含有的 RenderLayer 对象依 Render 树对应的父子关系组织 RenderLayer 树，RenderLayer 对象的存在是依附于 RenderObject 对象而存在。

RenderView 对象拥有对应的 RenderLayer 对象，同时其作为 RenderLayer 树根。

4. RenderLayer 树与 Render 树的关系

通过 RenderContainer::addChild 方法回过头再来具体看看 Render 树自身的构成。

```
void RenderContainer::addChild(RenderObject* newChild, RenderObject* beforeChild)
{
bool needsTable = false;
//检查是否为 Table 的情况
if (needsTable) {
.....
} else {
// just add it...
insertChildNode(newChild, beforeChild);
}
.....
}

void RenderContainer::insertChildNode(RenderObject* child, RenderObject*
beforeChild, bool fullInsert)
{
if (!beforeChild) {
appendChildNode(child);
return;
}

while (beforeChild->parent() != this && beforeChild->parent()->isAnonymousBlock())
beforeChild = beforeChild->parent();

if (beforeChild == m_firstChild)
m_firstChild = child;

RenderObject* prev = beforeChild->previousSibling();
child->setNextSibling(beforeChild);
beforeChild->setPreviousSibling(child);
if(prev) prev->setNextSibling(child);
child->setPreviousSibling(prev);

child->setParent(this);

if (fullInsert) {
// Keep our layer hierarchy updated. Optimize for the common case where we don't
have any children
// and don't have a layer attached to ourselves.
RenderLayer* layer = 0;
```

```
if (child->firstChild() || child->hasLayer()) {
layer = enclosingLayer(); child->addLayers(layer, child);
}

// if the new child is visible but this object was not, tell the layer it has some visible content
// that needs to be drawn and layer visibility optimization can't be used
if (style()->visibility() != VISIBLE && child->style()->visibility() == VISIBLE && !child->hasLayer()) {
if (!layer)
layer = enclosingLayer();
if (layer)
layer->setHasVisibleContent(true);
}
.....................................................
}

child->setNeedsLayoutAndPrefWidthsRecalc();
if (!normalChildNeedsLayout())
setChildNeedsLayout(true); // We may supply the static position for an absolute positioned child.
.....................................................
}
```

通过上面代码的了解我们知道通过addChild不仅维护Render树的构成，同时会将拥有的RenderLayer树构建起来。

5. RenderLayer树的作用

RenderLayer 树的构建为渲染阶段处理 z-index、opacity、overflow、scrollbar等打下一定的基础。在这里我们初步的了解到在构建Render树的同时会维护一颗RenderLayer树，为分层布局、渲染作准备。