第一章 绪论

    1.1 选题背景及意义

    1.2 与课题有关的国内外研究现状

    1.3 课题综述及主要工作

    1.4 论文的组织结构

第二章 相关理论与技术基础

    2.1 浏览器相关技术

        2.1.1 浏览器的对比

        2.1.2 浏览器基本原理

        2.1.3 浏览器内核架构

    2.2 Chromium开源浏览器Media的架构分析

    2.3 Gstreamer的基础

2.3.1 Gstreamer是什么

2.3.2 Gstreamer的整体框架

2.3.3 Gstreamer控件介绍

2.3.4 Gstreamer的使用方法

    2.4 流媒体标准

    2.5 本章小结

第三章 分析与设计

    3.1 功能性需求分析

    3.2 非功能性需求分析

    3.3 总体设计

    3.4 本章小结

第四章 实现与优化

    4.1 音频详细设计与实现

4.2 优化（多render改成被Browser管理，挖洞）

第五章 测试

第六章 总结与展望

致谢

参考文献

基于Gstreamer的Chromium音视频播放系统的设计与实现

摘 要

此处应该写摘要，摘要正摘要正文文摘要正文的格式摘要正摘要正文文摘要正文的格式摘要正摘要正文文摘要正文的格式要求是：四号宋体，首行缩进二个字，1.25倍行距。正文1000字左右。此处应该写摘要，摘要正摘要正文文摘要正文的格式摘要正摘要正文文摘要正文的格式摘要正摘要正文文摘要正文的格式要求是：四号宋体，首行缩进二个字，1.25倍行距。正文1000字左右。文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正要正文的格文摘要正。

关键词 关键词1，关键词2，关键词3，最后一个关键词无标点

THE DESIGN AND IMPLEMENTATION

OF CHROMIUM AUDIO AND VIDEO PLAYER

BASED ON GSTREAMER

ABSTRACT

ab (Homogenous Charge Compression Ignition) combustion has advantages in terms of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high. of efficiency and reduced emission. HCCI combustion can not only ensure both the high.

KeywordsHCCI, chemical kinetics, numerical simulation, DME

目录

1 绪论

1.1 选题背景及意义

此处是正文：中文字体是小四号宋体，英文是Times New Roman，英文是Times New Ro，英文是Times New Ro，英文是Times New Ro，英文是Times New Ro，英文是Times New Ro，英文是Times New Ro，英文是Times New Ro.

1.2 与课题有关的国内外研究现状

此处是正文：中文字体是小四号宋体，英文是Times New Roman，英文是Times New Ro，英文是Times New Ro，英

1.3 本文的研究内容及论文组织结构

此处是正文：中文字体是小四号宋体，英文是Times New Roman，英文是Times New Ro，英文是Times New Ro，英

1.4 本章小结

此处是正文：中文字体是小四号宋体，英文是Times New Roman，英文是Times New Ro，英文是Times New Ro，英

2 相关理论与技术基础

2.1 浏览器

网页浏览器（英语：Web Browser，常被称为浏览器（Browser）,之后将简称为浏览器）是一种用于检索并展示万维网信息资源的应用程序。这些信息资源可为网页、图片、影音、或其他内容，他们由统一资源标志符（Uniform Resource Identifier，缩写URI）标志。信息资源中的超链接可使用户方便地浏览相关信息。浏览器虽然主要用于使用万维网，但也可用于获取专用网络中网页服务器之信息或文件系统内之文件[1]。

2.1.1浏览器简介

互联网的革命浪潮带动着众多技术的发展，其中，浏览器作为互联网最重要的终端接口之一在短短二十多年时间里日新月异，特别是在进入21世纪后，越来越多的技术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，更是更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，他在上世纪80年代后期90年代初发明了世界上第一个浏览器，最初的名字叫WorldWideWeb，后来改名为Nexus，1991年的时候源代码被公布。它支持早期的HTML标记语言。当然它的功能很简单，只是支持文本、简单的样式表、声音、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。

1993年 Marc Andreessen领导的团队开发出Mosaic，这就是后来大名鼎鼎的网景（Netscape）浏览器。同样的，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IE）浏览器，自此第一次浏览器大战打响。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，开始凤凰涅槃。在该基金会的推动下，网景公司住到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），这拉开了一个新的序幕。同时，值的一提的是，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，实机上也有很多不同）。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，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上占有重要的一席之地。

至此，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和安卓浏览器变得非常流行。有趣的是，他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[2]。

2.1.2 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、多网页管理、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。下面是对它们之中一些重要功能的简要介绍。

* 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：从网络上下载或者本地获取资源，并将他们管理起来，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。
* 多网页管理：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，他们不仅能显示网页，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，扩展则跟浏览器由密切关系，常见的有Firefox扩展和Chromium扩展。
* 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。
* 开发者工具：这对普通用户来说用处不大，但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。

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。目前主流浏览器说支持的操作系统情况，如表2-1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2-1 操作系统支持

Table2-1 Operating system support

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | IE | Firefox | Chrome | Safari |
| Windows | 是 | 是 | 是 | 是（5.17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 |

2.1.3 浏览器基本工作原理

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，它被称为渲染引擎。所谓渲染，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。在介绍浏览器的基本工作原理之前，先介绍一下与网页相关的HTML、JavaScript和CSS。

* HTML：HyperText Markup Language，超文本标记语言，是一种用于创建[网页](https://zh.wikipedia.org/wiki/%E7%BD%91%E9%A1%B5)的标准[标记语言](https://zh.wikipedia.org/wiki/%E6%A0%87%E8%AE%B0%E8%AF%AD%E8%A8%80)，用于网页的创建和其他信息在浏览器中的显示。它是一种基础技术，常与[JavaScript](https://zh.wikipedia.org/wiki/JavaScript)、[CSS](https://zh.wikipedia.org/wiki/CSS)一起被众多网站用于设计令人赏心悦目的网页、网页应用程序以及移动应用程序的用户界面。其语法比较简单，基本上是一系列的标签（也成为元素），这些标签可以用来表示文字、图片、多媒体等，通常成对出现。[网页浏览器](https://zh.wikipedia.org/wiki/%E7%BD%91%E9%A1%B5%E6%B5%8F%E8%A7%88%E5%99%A8)读取HTML文件后，将其渲染成可视化网页。图2-1是一个简单的HTML文件被解释和渲染后的结果，其中左边是HTML文件，右边是用Chrome浏览器展示的结果。



图2-1 一个经典的HelloWorld程序示例

Fig2-1 an example of the classic [Hello world program](https://en.wikipedia.org/wiki/Hello_world_program)

* CSS：Cascading Style Sheets，层叠样式表，是一种用来为结构化文档（如HTML文档或XML应用）添加样式（字体、间距和颜色等）的计算机语言，由W3C定义和维护。CSS最重要的目标是将文件的内容与它的显示分隔开来。在CSS出现前，几乎所有的HTML文件内都包含文件显示的信息，比如字体的颜色、背景应该是怎样的、如何排列、边缘、连线等等都必须一一在HTML文件内列出，有时重复列出。CSS使开发者可以将这些信息中的大部分隔离出来，简化HTML文件，这些信息被放在一个辅助的，用CSS语言写的文件中。HTML文件中只包含结构和内容的信息，CSS文件中只包含样式的信息。
* JavaScript：一种直译式脚本语言，是一种动态类型、弱类型、基于原型的语言，内置支持类型。它的解释器被称为JavaScript引擎，为浏览器的一部分。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Jscrip。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，收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（HTML最新的修订版本，2014年10月由万维网联盟完成标准制定）中很多规范都是基于JavaScript语言来定义的。

我们可以暂时可以把渲染引擎看做一个黑盒，渲染引擎的作用可以用图2-2表示，左侧HTML/CSS/JavaScript是输入，图像是输出。

HTML/CSS/JavaScript

浏览器渲染引擎

图像

图2-2 浏览器渲染引擎的作用

Fig 2-2 The role of browser rendering engine

目前主流的渲染引擎包括Trident、Gecko和WebKit，他们分别是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，后面再介绍）。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，同一个渲染引擎可以被多个浏览器采用。表2-2显示了三个主流渲染引擎和采用它们的浏览器和Web平台。

表2-2 浏览器和Web平台及其渲染引擎

Table2-2 Browser/Web platform and Rendeing engine

|  |  |  |  |
| --- | --- | --- | --- |
|  | 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 |

前面我们把渲染引擎看做是一个黑盒子，黑盒子中包含什么以及有什么作用，我们还没有探究，接下来让我们一窥究竟。

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2-3中所描述的众多功能模块。图中主要分为三层，最上层使用虚框线框住的是渲染引擎说提供的功能。

渲染引擎

HTML解释器

CSS解释器

布局

JavaScript引擎

网络

存储

2D/3D图形

音视频

图片解码器

操作系统支持

图2-3 渲染引擎模块及其依赖的模块

Fig 2-3 Rendering engine modules and its dependent modules

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，下面依次来描述他们。

* HTML解释器：解释HTML文本的解释器，主要作用就是将HTML文本解释成DOM（Document Object Model，文档对象模型）树，DOM是一种文档的表示方法。
* CSS解释器：级联式样表的解释器，它的作用是为DOM中各个元素对象计算出式样信息，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，计算他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（CSS Object Model ，CSS对象模型）接口来修改网页内容和式样信息，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等。实际上，渲染引擎中还应该包括如何使用这些依赖模块部分。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，当然在最下层当然少不了操作系统的支持，例如线程支持、文件支持等。

在了解这些模块之后，下面先整体介绍一下这些模块是如何一起工作以完成网页渲染的。一般的，一个典型的渲染过程如图2-4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2-4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解释器在解析它后构建成一棵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，这交给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2-4 渲染引擎的一般渲染过程及各阶段依赖的其他模块

DOM

网页内容

CSS解释器

HTML解释器

JavaScript引擎

内部表示

布局和绘图

图像

网络

存储

2D/3D图形

音频和视频

图片解码器

Fig 2-3 The process of Rendering and dependent modules

最后解释图2-4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。

以浏览器内核WebKit为例，根据数据的流向，可以将渲染分为三个阶段，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树到构建完成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。为了藐视这个过程，下面会将WebKit中的一些细节展示给大家。图2-5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，因为整个过程可能重复并且可能交叉。

图2-5 从网页URL到DOM树

资源加载器

URL

网页内容、JS、CSS等

DOM树

网络模块

HTML解释器

JavaScript引擎

1

2

3

4

5

6

7

8

Fig 2-5 The process of generating DOM tree

具体的过程如下。

1. 当用户在地址栏输入URL的时候，WebKit调用资源加载器加载该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引擎解释并运行。
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；如果是加载JavaScript资源URL（没有标记异步方式），则需要暂停当前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“DOMContent”事件和DOM的“onload”事件，分别在DOM树构建完成后以及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树的构建，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2-6所示的过程。

DOM树

CSS解释器

CSS

RenderObject树

RenderLayer树

绘图上下文

布局计算

1

2

3

图2-6 从CSS和DOM到绘图上下文

Fig 2-6 From CSS and DOM to Drawing context

这一阶段的具体过程如下。

1. CSS文件被CSS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。
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，这里不再赘述。

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2-7所示。

绘图上下文

2D图形库

最终的图像

3D图形库

绘图具体实现类

3

1

2

图2-7 从绘图上下文到最终的图像

Fig 2-7 From Drawing final image

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，也有可能复杂的实现。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。

上边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。

2.1.4 浏览器内核架构及Chromium浏览器总体架构

前面曾介绍过目前主流的渲染引擎包括Trident、Gecko和WebKit，其中WebKit目前作为苹果公司及谷歌等公司浏览器的主要内核，从2012年11月开始，市场占有超过40%，成为拥有最大市场份额的排版引擎，超越了IE所使用的Trident内核（数据来自浏览器市场份额调查），具有快速、稳定、安全的特点。

说到WebKit，还得从KHTML（HTML网页排版引擎之一，由KDE所开发。）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他们。两者的分歧越来越大，最终在2001年，苹果宣布从KHTML的源代码树种复制代码出来，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。

当我们谈到“WebKit”的时候，其实可以表示两种含义，这里姑且成为广义WebKit和狭义WebKit。广义的WebKit指的就是WebKit项目。为了解释狭义的WebKit，让我们在更高层次上俯视一下这个开源项目。图2-8显示的是该项目的大模块。图中“WebKit嵌入式接口”就是指的狭义WebKit，它指的是在WebCore（包含前面提到的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。以后若无特殊说明，所引用的WebKit均是指广义的概念。

WebKit项目

WebKit嵌入式接口

WebCore

JavaScript引擎

调用系统或依赖库接口的桥接层

图2-8 WebKit项目的大模块

Fig 2-8 Brief Modules for WebKit project

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）。前边我们介绍过一张简单的WebKit结构图，图中只有2~3个模块，那是故意隐去了其中的细节，下图（图2-9）是更为详尽的的WebKit架构图。

WebKit

WebCore

WebKit嵌入式接口

WebKit2嵌入式接口

WebKit绑定

WebKit2绑定

WebKit Ports

网络栈

音视频

文字

硬件加速

图片解码

. . .

JavaScriptCore

CSS

SVG

布局

渲染树

HTML

DOM

Inspector

. . .

2D图形库

t

3D图形库

t

网络库

t

存储

t

音频库

t

视频库库

t

. . .

t

操作系统

图2-9 WebKit架构

Fig 2-9 The Architecture of WebKit

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。之所以没有说的那么绝对，是因为它们中的一些特征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。这里面有很多的不同，所以，很多使用WebKit的浏览器可能会表现出不同的行为，就不令人惊奇了。

下面，我们开始依次从下向上分析。

图中最下面是“操作系统”，WebKit可以在不同的操作系统上工作。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、MacOS、Linux、Android等系统。

在“操作系统”之上的就是WebKit赖以工作的第三方库，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大，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，每层包含很多个模块，由于图的大小限制，略去了其中一些次要模块。

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的网页的基础部分，他们必不可少，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。举个例子来说，“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。WebCore这些部分主要被前面介绍过的加载和渲染过程的第一、二阶段说使用。

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，她的性能不是很好，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。之所以说它还是默认的，是因为它不是唯一并且是可以替换的。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的原因，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。

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（WebKit2 is a new API layer for WebKit to support a split process model, where the web content lives in a separate process from the application UI[3].）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由是统一接口。

WebKit还有一个部分没有展示出来，那就是测试用例，包括布局测试用例和性能测试用例，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。

以上，便是对WebKit架构的介绍。

前面我们提到WebKit是从KHTML分裂出来的。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，Google宣布了从WebKit复制出来并独立运作的Blink项目，其中原因也是一言难尽，主要还是Google和苹果公司有了一些分歧。据一些第三方的 WebKit 开发者透露，苹果和 Google 的开发者在交流时没有一般开源开发者的那种相互支持，反而更像两头相互打量的狮子，气氛比较紧张。而苹果团队则称Google不愿意将多进程架构的支持整合到 WebKit 中，这些代码一直在 Chromium 自己的目录中。最初Blink和WebKit并没有特别大的不同，因为刚刚从WebKit复制过来。

在二者“分手”后，WebKit将于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。二者的差别在以后会越来越大，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。

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MacOS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。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。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，包括但不限于图形、JavaScript引擎、内存的使用，编译二进制文件的大小，等。

之后还有很多其他技术会被逐渐引入，我们持续关注WebKit和Blink的发展。

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也是很多技术的创新者，它将很多先进的理念引入到浏览器领域。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。为了方便理解，我们从架构和米快、多进程模型和多线程模型角度一一剖析。

首先要熟悉的是Chromium的架构及其包含的模块。

Chromium浏览器

Content Shell

Android WebView

Content接口

Content模块

Blink(WebKit)

GPU/Command Buffer

V8

沙箱模型

CC(Chromium合成器)

IPC、

UI、

PPAPI、

. . .

图2-10 Chromium模块结构图

Fig 2-10 Modules of Chromium

图2-10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）。

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的模块。这里或许有个疑问，WebKit不就是渲染网页内容的吗？是的，没有Content模块，浏览器开发者也可以在WebKit的Chromium移植上渲染网页内容，但是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，其一可以用来检测Content模块很多功能的正确性，例如渲染、硬件加速等；其二是一个参考，可以被很多外部的项目参考来开发基于“Content API”的浏览器或者各种类型项目。

前面多次提到Chromium的多进程模块架构，下面谈谈它的好处。相信你一定用这样的经历：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。

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。

多进程模型现在不可避免的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。该模型至少带来三个好处：其一是避免因单个网页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？

Browser进程

NPAPI Plugin

NPAPI Plugin

. . .

Render

Render

. . .

Pepper Plugin

Pepper Plugin

. . .

GPU

图2-11 Chromium多进程模型

Fig 2-11 Mutil-process of Chromium

图2-11给出了最常用的Chromium浏览器多进程模型，Chromium架构的设计是非常灵活的，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2-11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，负责他们的创建和销毁等工作，它有且仅有一个。
* Render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Render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。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，Render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。

通过上面的讨论，对于桌面系统（Windows、Linux、MacOS）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进程，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。

经过这些分析我们可以看到，多进程模型多使用比较多的资源，但换来的好处是浏览器更稳定了。

上面说到Chromium允许用户配置Render进程被创建的方式，下面简单介绍下模型的类型。

* 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。例如，用户访问了gmail个人账户主页，然后从主页分别打开多封邮件，每封邮件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gmail个人账户的主页，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烦人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。好处是相对于相同的域，进程可以共享，内存消耗较小，坏处是可能会有特别大的Render进程。
* 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，所有渲染工作都放在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游泳，可以在命令行加入参数“--single-process”来尝试它。

因为Browser进程和Render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2-12给出。

浏览器的用户界面

(src/chrome)

Web Contents页面内容

(src/content/browser/web\_contents)

RendererHost

(src/content/renderer\_host)

Renderer

(src/content/renderer)

WebKit黏附层

(src/webkit/glue)

WebKit接口层

(WebKit/Source/WebKit)

Browser

进程

Renderer

进程

图2-12 从WebKit接口层到用户界面的路径

Fig 2-12 The path from WebKit interface layer to user UI

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。

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Render进程之间的通信。不过RendererHost是给Render进程发送请求并接收来自Render进程结果。

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，这个子窗口最后被嵌入到浏览器的用户界面，作为它的一个标签页。

通过上面的介绍，这里的关系已经基本理顺。那么，进程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。

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相应。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。既然文件读写会阻碍其他操作，那好，把它们放在单独的线程里自己忙或者等待去吧。而在Render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2-13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。

Browser进程

UI线程

IO线程

Render进程

渲染线程

IO线程

GPU进程

IO线程

GL线程

图2-13 Chromium的多线程模型

Fig 2-13 The multi-thread model of Chromium

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Render进程。
2. Render进程的IO线程经过简单的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Render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。

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。

以上，是我们对浏览器基本技术的简单介绍。当然，浏览器所涉及的技术不仅仅只有我们介绍的这些，它是一个庞大而复杂的项目。考虑到篇幅所限，我们只针对本论文可能涉及的部分进行了简单的描述，以便于后文的展开。

2.3 Gstreamer基础

2.3.1 Gstreamer是什么

Gstreamer是一个开源的多媒体框架库。利用它，可以构建一系列的媒体处理模块，包括从简单的Ogg/Vorbis回放、到音视频流媒体以及复杂的音频（混音）和视频（非线性编辑）处理。应用程序可以透明地利用解码和过滤技术，开发者可以使用简洁通用的接口来编写一个简单的插件来添加新的解码器或滤镜[4]。它的基本设计思想来自于俄勒冈（Oregon）研究生学院有关视频管道的创意，并且借鉴了微软DirectShow的设计思想[5]。

2.3.2 Gstreamer特点

Gstreamer框架是基于插件的，有些插件中提供了各种各样的多媒体数字信号编解码器，也有些提供了其他的功能。所有的插件都能够被链接到任意的已经定义了的数据流管道中。它的核心库函数是一个处理插件、数据流和媒体操作的框架。另外，核心库还提供了一个API，这个API是开放给开发者使用的，当开发者需要使用其他的插件来编写他所需要的应用程序的时候可以使用它。

Gstreamer的程序开发框架使得编写任意类型的流媒体应用程序成了可能。在编写处理音频、视频或者两者皆有的应用程序时，Gstreamer可以让工作变得简单。它并不受限于音频和视频处理，理论上能够处理任意类型的数据流。管道设计的方法对于实际应用的滤波器几乎没有负荷，它甚至可以用来设计出对延时有很高要求的高端音频应用程序。

2.3.3 基本概念

在Gstreamer中有一些基本的概念和术语，了解和理解这些概念和术语是使用Gstreamer的前提和基础，后续章节会进场提到这些概念和术语。

最重要的一个概念就是元件（elements）。要想使数据流顺利的传输，需要创建一系列的元件，并将这些元件按照一定规则连接起来（当然连接元件涉及到例外一个概念，后边会介绍。），然后数据流就可以在这些被连接起来的元件间传输。想象一下一段段的自来水管，为了把洁净的水输送到每家每户，工作人员从供水中心铺设管道，利用大小不一各种各样的接口、水管和其他元件一段衔接一段接入到每个家庭。元件的类型有多种，一个元件它的类型可能是sources、filters、muxers、demux­ers、codecs或者sinks。每个元件都有一个特殊的函数接口，有些元件的函数接口是用于能够读取文件的数据、解码文件数据的，而有些元件的函数接口只是输出相应的数据到具体的设备上（例如：声卡设备、显卡设备，等）。我们将若干个元件连接在一起，从而创建一个管道来完成一个特殊的任务，例如音频的播放和录制。Gstreamer默认已经安装了很多有用的元件，通过使用这些元件你能够构建一个具有多种功能的应用程序。在某些情况下，开发人员可以根据项目的需要编写一些特别的新的元件。一般来说，元件像是一个黑盒，数据从黑盒的一端流入，在黑盒内部进行一些加工处理，然后从另外一端流出被处理过的数据。例如：对于一个mux­er元件，流入的是视频文件、音频文件、可能还会有字幕文件，流出的是某一格式的视频文件。如，可将1.avi、1.mp3、1.srt用muxer合并为mkv格式的视频文件。

插件（plugins）是一块可以加载的代码。通常被称为共享对象文件或动态链接库。一个插件中可以包含一个或多个元件，因此可以把多个元件封装进一个插件中，使之支持新的功能。Gstreamer充斥着插件的概念，即使你只用到一些标准的包。核心库只有少量的基本函数，其它所有的功能都由插件来实现。一个XML文件被用来保存所有注册的插件的详细信息。这样，使用Gstreamer的程序可以只在需要时加载插件，而不必实现全部加载。

衬垫（pads）是Gstreamer的另外一个基本概念。在Gstreamer中，衬垫是用来在元件间协商连接和数据流的。衬垫可以看做元件见相互连接的“接口”，数据流通过这些接口流入和流出元件。衬垫具有特殊的数据处理能力：衬垫可以限制通过它的数据类型。只有当两个衬垫允许通过的数据类型兼容时才可以将它们连接起来。打个比方，衬垫类似于物理设备上的a plug or jack。想想一个包含功放、DVD播放器和一个视频投影仪的家庭影院系统。将投影仪和DVD播放器相连是允许的，因为这两个设备具有兼容的video jack。而将投影仪和功放连接起来也许就行不通了，因为他们之间的jack不同。大部分情况下，所有在Gstreamer中流经的数据都遵循一个原则，数据从元件的一个或多个源衬垫流出，从一个或多个sink衬垫流入。源和sink元件分别只有源和sink衬垫。

以上是基本概念和术语的介绍，下面我们介绍几种常见的元件：

1. 源元件（source elements）

源元件是管道数据的产生者，比如从硬盘或者声卡等设备中将数据读取出来。如图2-14所示：源元件有且只有一个源衬垫（source），它只产生数据，并不接收数据。也就是说，它是数据的源头，不能对数据做任何处理。

source element

source

图2-14 源元件示意图

Fig 2-14 source element

1. 过滤器（filters）与类过滤元件（Filter-like elements）

过滤器以及类过滤元件都同时拥有输入和输出衬垫，它们对从输入衬垫接收到的数据进行处理操作，然后将处理后的数据提供给输出衬垫。音量元件、视频转换器、Ogg分流者或者Vorbis解码器都是这种类型的元件。类过滤元件可以拥有任意个的元衬垫或者接收衬垫。像一个视频分流器可能有一个接收衬垫以及多个（1-N）源衬垫，每个接收衬垫对应一种元数据流（elementary stream）。相反地，解码器只有一个源衬垫及一个接收衬垫。图2-15为形象化的过滤元件，这个特殊的元件同时拥有源端和接收端，接收输入数据的接收衬垫在元件的左端，源衬垫在右端。图2-16为形象化的拥有多个输出的过滤元件，它有多个输出衬垫（source pad）。Ogg分流器是个很好的实例，因为Ogg流包含音频元数据流。当一个新的衬垫被创建时，分流器通常会产生一个信号，开发者可以在信号处理事件中处理新的元数据流。

filter

src

sink

图2-15 形象化的过滤元件

Fig 2-15 Figurative filter element

video

demuxer

sink

audio

图2-16 有多个输出的过滤元件

Fig 2-16 filter element with multiple outputs

1. 接收元件（sink elements）

接收元件处在媒体管道的末端，如图2-17所示为接收元件的示意图，它只有一个接收衬垫（sink pad），因而它只接收数据，不产生数据。例如常见的写磁盘、利用声卡播放声音，以及视频输出都是由接收元件实现的。

sink element

sink

图2-17 接收元件

Fig 2-17 sink element

1. 箱柜（bins）和管道（pipelines）

箱柜是一个可以装载元件（element）的容器。管道是箱柜的一个特殊子类型，管道可以操作包含在它自身内部的所有元件。因为箱柜本身又是元件的子集，所以能够像操作普通元件一样地操作箱柜，通过这种方法可以降低应用程序的复杂度。开发者可以改变一个箱柜的状态来改变箱柜内部所有元件的状态。箱柜可以发送总线消息（bus messages）给它的子集元件。管道是高级的箱柜，当设定管道的暂停或者播放状态的时候，数据流将开始流动，并且媒体数据处理也开始进行。一旦开始，管道将在一个单独的线程运行，直到被停止或者数据流播放完毕。箱柜的示意图如图2-18所示。

bin

element3

element2

element1

sink

src

src

sink

图2-18 箱柜示意图

Fig 2-18 bin

1. 元件状态

一个元件在被创建后，它不会执行任何操作。如果想要元件执行某些动作，就需要需要改变元件的状态，使得它能够做某些事情。在Gstreamer中，元件有四种状态，每种状态都有其特定的意义。

* GST\_STATE\_NULL：默认状态。所有新创建的元件的初始默认状态，该状态将回收所有被该元件占用的资源。
* GST\_STATE\_READY：准备状态。这种状态下，元件会得到所有所需的全局资源，这些全局资源将被通过该元件的数据流所使用。例如打开设备、分配缓存。但这种状态下，数据流仍未开始被处理，所以数据流的位置信息应该自动置0。如果数据流先前被打开过，它应该被关闭，并且其位置信息、特性信息应该被重置为初始状态。
* GST\_STATE\_PAUSED：暂停状态。在这种状态下，元件已经对流开始了处理，但此刻暂停了处理。因此，该状态下元件可以修改流的位置信息，读取或者处理数据流，以及一旦状态变为PLAYING，流可以重放数据流。这种情况下，时钟是禁止运行的。总之，PAUSED状态除了不能运行时钟外，其它与PLAYING状态一模一样。处于PAUSED状态的元件会很快变换到PLAYING状态。举例来说，视频或音频输出元件会等待数据的到来，并将它们压入队列。一旦状态改变，元件就会处理收到的数据。同样，视频接收元件能够播放数据的第一帧（因为这并不会影响时钟）。自动加载器（Autopluggers）可以对已经加载进管道的插件进行这种状态转换。其它更多的像codecs或者filters这种元件不需要在这个状态上做任何事情。
* GST\_STATE\_PLAYING：PLAYING状态除了当前运行时钟外，其它与PAUSED状态一模一样。你可以通过函数gst\_element\_set\_state()来改变一个元件的状态。你如果显式地改变一个元件的状态，Gstreamer可能会使它在内部经过一些中间状态。例如，将一个元件从NULL状态设置为PLAYING状态，Gstreamer在其内部会使得元件经历过READY以及PAUSED状态。当处于GST\_STATE\_PLAYING状态，管道会自动处理数据。它们不需要任何形式的迭代，Gstreamer会开启一个新的线程来处理数据。Gstreamer同样可以使用GstBus在管道线程和应用程序线程间交互信息。

2.3.4 总线（Bus）

总线是一个简单的系统，它拥有一套自己的线程机制，采用这套机制可以把一个管道线程的message分发到一个应用程序中。当使用Gstreamer的时候，由于总线机制的存在，应用程序不需要对线程过分关心或者线程识别，尽管Gstreamer可能已经加载了多个线程。

当一个管道创建后，默认是包含有一个总线的，因此应用程序不需要再创建总线，只需要在总线上设置一个消息处理器。当主循环运行的时候，总线会轮询这个消息处理器看是否有新的消息，如果有消息则消息被采集到，总线会调用相应的回调函数完成处理。

使用总线的方法有两种。第一种是运行GLib/Gtk+主循环（也可以自己运行默认的GLib循环），然后使用侦听器对总线进行侦听。如果使用这种方法，Glib的主循环将轮询总线上是否存在新的消息，当存在新的消息的时候，总线会马上通知你。在这种情况下，会用到gst\_bus\_add\_watch()/gst\_bus\_add\_signal\_watch()两个函数。当使用总线时，设置消息处理器到管道的总线上可以使用gst\_bus\_add\_watch()来创建一个消息处理器来侦听管道。每当管道发出一个消息到总线，这个消息处理器就会被触发，消息处理器则开始检测消息信号类型从而决定那些时间将被处理。第二种是自己侦听总线消息，使用gst\_bus\_peek()/gst\_bus\_poll()就可以实现。

Gstreamer有几种由总线传递的预定义消息类型，这些消息都是可扩展的，插件可以定义自己的一些信息。所有的消息都有一个消息源、类型和时间戳。这个消息源能被用来判断由哪个element发出消息。例如在众多的消息中，应用程序只对上层的管道发出的消息感兴趣（例如状态发生了改变）。下面是一些消息的简介。

* 错误、警告和消息提示：它们被各个元件用来在必要的时候通知用户现在管道的状态。错误信息表明有致命的错误并且终止数据传送，错误应该被修复，这样才能继续管道的工作。警告并不是致命的，但是暗示有问题存在。消息提示用来告知非错误的信息。这些消息含有一个带有主要的错误类型和消息的GError，和一个任选的调试字符串。这两项都可以用gst\_message\_parse\_error()/gst\_message\_parse\_warning()以及gst\_message\_parse\_info()三个函数来提取其信息。当使用完毕后，错误和修正字符串都会被释放出来。
* 数据流结束（End of stream）提示：当数据流结束的时候，改消息被发出。管道的状态不会改变，但是之后的媒体操作将会停止。应用程序接收到这一消息后，可以根据具体的情况作出动作，比如跳到下一首歌播放。在数据流结束提示的消息出现之后，任然是可以通过向后搜索来回到以前数据流前面的位置。之后的播放工作将会自动的继续执行。此消息不携带特殊参数。
* 标签（Tags）：当元数据在数据流中被找到的时候，此消息被发送。一个管道可以发出多个Tag（如元数据的描述里有艺术家、歌曲名等）。应用程序应该将元数据存储在缓存里。函数gst\_message\_parse\_tag()被用来解析tag的列表，当该列表不再使用的时候，函数gst\_tag\_list\_free()释放其相应的tag。
* 状态转换（State Changes）：当状态成功的转换时，此消息被发送。函数gst\_message\_parse\_state\_changed()可以用来解析转换中的新旧状态。
* 缓冲（Buffering） ：当对网络数据进行缓冲的时候，此消息被发送。可以通过gst\_message\_get\_structure()的返回值解析“buffer percent”，从而得到缓冲进度的百分比。
* 元件消息（Element messages）：它是一组特殊的消息，用以标识一个特定元件。这样一组特殊的消息通常表述了一些额外的信息。元件的信息应该被详细的描述，因为这样一些元件信息将被作为消息而发送给其他元件。例如：“qtdemux”QuickTime整流器（demuxer）应该把“redirect”信息保存于该元件信息中，以便在某些特殊情况下将“redirect”元件信息发送出去。
* Application-specific消息：我们可以将取得的消息结构解析出来，从而得到有关Application-specific消息的任何信息，通常这些信息是能够被安全地忽略。应用程序消息主要用于内部，以备从一些线程排列信息到主线程应用的需求。

2.3.5 Gstreamer使用方法

下面，我们以一个具体的实例来看看如何利用Gstreamer框架的组件，实现一个简单的MP3播放器。数据源元件负责从磁盘上读取数据，过滤器元件负责对数据进行解码，而接收器元件负责将解码后的数据写入声卡。

要想在程序中应用Gstreamer提供的各种功能，首先必须在主函数中调用gst\_init()来完成初始化工作，以便用户从命令行输入的参数传递给Gstreamer函数库。一个典型的Gstreamer应用程序的初始化如下所示：

#include <gst/gst.h>

int main(int argc, char \*argv[])

{

gst\_init(&argc, &argv);

}

接下去需要创建三个元件并连接成管道，由于所有Gstreamer都具有相同的基类GstElement，所以可以采用如下方式进行定义：

GstElement \*pipeline, \*filesrc, \*decoder, \*audiosink;

管道在Gstreamer框架中是用来容纳和管理元件的，下面的代码将创建一条名为pipeline的新管道：

pipeline = gst\_pipeline\_new(“pipeline”);

数据源元件负责从磁盘文件中读取数据，它具有名为“location”的属性，用来指明文件在磁盘上的位置。使用标准的GObject属性机制可以为元件设置相应的属性：

filesrc = gst\_element\_factory\_make(“filesrc”, “disk\_source”);

g\_object\_set(G\_OBJECT(filesrc), “location”, argv[1], NULL);

过滤器元件负责将解码后的数据利用声卡播放出来：

audiosink = gst\_element\_factory\_make(“audiosink”, “play\_audio”);

已经创建好的三个元件需要全部添加到管道中，并按顺序连接起来：

gst\_bin\_add\_many(GST\_BIN(pipeline), filesrc, decoder, audiosink, NULL);

gst\_element\_link\_many(filesrc, decoder, audiosink, NULL);

所有准备工作都做好之后，就可以通过将管道的状态切换到PLAYING状态，来启动整个管道的数据处理流程：

gst\_element\_set\_state(pipeline, GST\_STATE\_PLAYING);

 这里可以通过gst\_pipeline\_get\_bus(GST\_PIPELINE(pipeline))获取到总线bus，然后加入一个消息处理函数bus\_call来监视产生的消息：

gst\_bus\_add\_watch(bus, bus\_call, loop)

bus\_call可以定义如下：

static gboolean bus\_call(GstBus \*bus, GstMessage \*msg, gpointer data)

{

// do something

}

 当播放结束，终止管道，并解除引用释放资源：

gst\_element\_set\_state(pipeline, GST\_STATE\_NULL);

gst\_object\_unref(GST\_OBJECT(pipeline));

以上是一个简单的使用Gstreamer创建播放器的实例，便于我们理解Gstreamer的使用方法和流程。

2.3.6 Gstreamer音视频同步机制

媒体内容在播放时，最不好控制的就是音视频的同步问题。从技术上说，解决音视频同步问题的首选方案就是时间戳：首先选择一个参考时钟（要求参考时钟上的时间是线性递增的）；生成数据流时依据参考时钟上的时间给每个数据块都打上时间戳（一般包括开始时间和结束时间）；在播放时，读取数据块上的时间戳，同时参考当前参考时钟上的时间来安排播放（如果数据块的开始时间大于当前参考时钟上的时间，则不急于播放该数据块，直到参考时钟达到数据块的开始时间；如果数据块的开始时间小于当前参考时钟上的时间，则尽快播放这块数据或者索性将这块数据丢弃，以便播放进度追上参考时钟）。

可见，避免音视频不同步的现象有两个关键，一是在生成数据流时要打上正确的时间戳，如果数据块上打的时间戳本身有问题，那么播放时怎么调整也于事无补；二是在播放时基于时间戳对数据流控制，也就是对数据块早到或晚到采取不同的处理方法。基于时间戳的播放过程中，仅仅对早到的或晚到的数据块进行等待或快速处理有时候是不够的。

如果想要更加主动并且有效地调节播放性能，需要引入一个反馈机制，也就是要将当前数据流速度太快或太慢的状态反馈给“源”，让源去放慢或加快数据流的速度。熟悉DirectShow的开发者一定知道，DirectShow中的质量控制（Quality Control）就是这么一个反馈机制。

Gstreamer使用时钟来维持管道中的播放同步。时钟是由某些元件（如audio sink）来提供的，因此，其他元件只是从属于该时钟。时钟的主要任务是根据提供时钟元件的播放速率显示时间进度。如果管道中没有元件提供时钟，那么系统时钟会替代元件来提供时钟。

Gstreamer从播放状态中衍生出四个重要的时钟概念，分别是clock time(absolute time)、base time、running time、stream time。clock time是管道维护的一个全局时钟，是一个以纳秒为单位的单调递增时钟，可以通过gst\_clock\_get\_time()函数获取，如果管道中没有元素提供时钟，则使用该系统时钟。base time指媒体从0开始的全局时间值，可以通过\_get\_time()函数获取。running time表示媒体处于PLAYING状态时流逝的时间。stream time是媒体播放的位置（在整个媒体流中）。

管道的同步通过如下三个方面实现：GstClock、Timestamps of GstBuffer、NEW\_SEGMENT event preceding the buffers。running\_time有两种计算方式：第一种是用全局时钟和元素的base\_time计算，running\_time = absolute\_time – base\_time；第二种是用buffer的时间戳和NEW\_SEGMENT事件计算（假设rate为正值），running\_time = (B.timestamp – NS.start) / NS.abs\_rate + NS.accum。同步主要是保证上述两个时间计算值的相同，即absolute\_time – base\_time = (B.timestamp – NS.start)/NS.abs\_rate + Ns.accum，而absolute\_time也就是Buffer的同步时间（B.sync\_time == absolute\_time），因此B.sync\_time = (B.timestamp – NS.start)/NS.abs\_rate + Ns.accum + base\_time

在render之前需要等待，直到时钟到达sync\_time；对于多个流，则是具有相同running\_time的将会同步播放；解复用器（demuxer）则需要保证同时播放的Buffers具有相同的running\_time，因此会给Buffer附上相同的时间戳以保证同步。

管道中元素与时钟的同步仅仅发生在各个sink中，如果其他元素对buffer没有延迟的话，那么延迟就为0。延迟的引入主要是基于这样的考虑，buffer从source推送到sink会花费一定的时间，从而可能导致buffer被丢弃。这个问题一般发生在活动管道，sink被设置为PLAYING并且buffer没有被预送（preroll）至sink。

一般的解决方案是在被预送（preroll）之前所有sink都不能设置为PLAYING状态。为了达到这样的目的，管道需要跟踪所有需要被预送的元素（就是在状态改变后返回ASYNC的元素），这些元素发送一个ASYNC\_START消息，当元素进行预送，便把状态设置为PAUSED，同时发送一个ASYNC\_DONE消息，该消息恰好与之前的ASYNC\_START相对应。当管道收集了所有与ASYNC\_START消息对应的ASYNC\_DONE消息以后便可以开始计算全局延迟了。

Gstreamer的同中步主要在sink中实现，在render之前进行，因此一般在函数GstBaseSink::render中具体实现。同步指的是buffer在进入到每个sink、render之前与时钟的同步。媒体流在解复用后，在其多个流（比如音频流和视频流）的buffer中附加了时间戳，因此在sink进行输出之前分别与时钟进行同步，即可达到A/V的同步输出。

在gstreamer-0.10.3之前，同步在GstBaseSink的函数gst\_base\_sink\_render\_object()中实现，子类对其进行覆盖的很少。在之后版本中，在某些具体的sink子类中进行了覆盖，使得同步的效果达到了最佳。比如在GstBaseAudioSink的函数gst\_base\_audio\_sink\_render()中对Audio的同步进行了覆盖。然而对于食品的同步并没有覆盖，仍然在基类中进行实现。

2.4 音视频编解码压缩标准（《基于Gstreamer的网络媒体播放系统的开发 3.2》）

音视频压缩技术是指采用恰当的数字信号处理技术对原始的音频视频数字信号进行压缩编码。而压缩编码必须要用相应的解压缩或解码的逆变化，只有采取了逆变换过程将信号还原给观众才能进行节目观看。经过一段时间的发展，音视频编解码方案已经出现了很多，目前常见的主要是MPEG系列和H.26X系列，其中H.264X系列只用于视频方面的压缩编码，而MPEG系列在音频视频编码方面都适用。

2.4.1 音频编解码压缩标准

音频压缩主要有无损压缩和有损压缩两种，其下对应的压缩技术又有很多种。目前，音频编码的标准主要有：MPEG-1、Dolby AC-3、AAC以及WMA。而MPEG-1中的III层标准——MPEG Audio Layer3是目前应用最广泛的音频编解码压缩标准。

* MPEG-1与MP3

由运动图像专家组开发的MPEG-1标准（ISO/IEC11172-3）在音频编码方面分为I层、II层和III层三层，每层都在计算复杂度和声音的质量方面较上一层有所提高，且II层和III层又分别被称为MP2和MP3。这三层是分等级的，每一层的比特流格式和解码器被指定，其中I层解码器只能处理本层，II层解码器能处理本层以及I层，而III层解码器三层都能处理。

这三层工作在想进的比特率范围内：I层工作在32~448kbit/s范围；II层工作在8~384kbit/s范围；III层工作在8~320kbit/s范围。虽然三层工作范围相似，但目标比特率不同，从I层到III层分别是每信道192kbits/s、128kbits/s和64kbit/s。这三层编码方案的目的都是尽可能的保证声音质量，都采用了感知方法，利用人类听觉系统中存在的某些特定缺陷去对信号进行处理，从另一个方面保证了人们感知到的声音质量。I层在VCD中得到了应用；II层因为复杂度恰当、声音质量优秀在一些数字节目的制作、传送等方面得到广泛使用；III层是在II层基础上融合了ASPEC思想的混合压缩技术，在低码率情况下拥有较高的声音质量，因而在软解压和网络广播方面如鱼得水。

下面我们来了解一下MPEG-1标准中的III层。

MPEG-1标准在编解码压缩方面的制定是相当成功的，

2.5 本章小结

参考文献

[1] <https://en.wikipedia.org/wiki/Web_browser> This page was last modified on 7 June 2016, at 05:48. Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

[2] WebKit技术内幕

[3] http://trac.webkit.org/wiki/WebKit2

[4] https://gstreamer.freedesktop.org/

[5] 张治忠.基于SEP6200处理器的Gstreamer媒体播放器优化与实现