·前沿与综述·

终端 Web 运行环境及其相关优化技术研究综述*

王 昭[1,2] 邓浩江[1] 朱小勇[1] 胡琳琳[1]

([1] 中国科学院声学研究所 国家网络新媒体工程技术研究中心 北京 100190
[2]中国科学院大学 电子电气与通信工程学院 100049)

摘要: 随着 HTML5 标准的发布和不断完善,Web 应用功能越来越丰富的同时,其性能也愈发成为用户关注的焦点,Web 运行环境作为支撑 Web 应用解析执行的核心运行环境,其架构、资源获取、代码解析执行等机制直接影响到 Web 应用的性能。本文对终端主流 Web 运行环境及其架构进行了介绍,并对 Web 运行环境解析执行 Web 应用过程中所涉及不同方面的相关优化技术进行了分析和总结。

关键词: Web 运行环境,Web 应用,HTML5,性能优化

Overview of Terminal Web Runtime and Related Optimization Technology

WANG Zhao[1,2], DENG Haojiang[1], ZHU Xiaoyong[1], HU Linlin[1]

([1]Institute of Acoustics, Chinese Academy of Sciences, National Network New Media Engineering Research Center, Beijing, 100190, China, [2]University of Chinese Academy of Sciences, School of Electronic, Electrical and Communication Engineering, Beijing, 100049, China)

Abstract: With the release and continuous improvement of HTML5 standards and the function of web application becoming more and more plentiful, the performance of web application is even becoming the focus of web users. Web runtime is the core running environment that supports the resolving and executing of web application and its architecture, resource acquisition, code resolving and other mechanisms directly affect the performance of web application. This paper introduced the architecture of terminal mainstream web runtime, analyzed and summarized the related optimization technologies from different perspectives that is concerned in the running process of web application.

Keywords: Web Runtime, Web Application, HTML5, Performance Optimization

0 引言

随着 Web 技术的不断发展,Web 已经从简单的网页扩展到了 Web 应用[1],其支撑运行环境也从浏览器向 Web 运行环境(Web runtime)转变。Web 应用是由 HTML、CSS、JavaScript 等 Web 语言编写的应用,它具有轻量级、跨平台等优势。Web 运行环境作为支撑 Web 应用运行的核心环境,可以实现 Web 应用的运行和全生命周期管理,它可以使 Web 应用像本地应用一样在桌面上启动,并且用户可以随时安装、更新和卸载。Web 运行环境的具体功能有支持 Web 应用运行; Web 应用访问控制; 通过 JS API 向 Web 应用提供传感器、摄像头等设备资源; Web 应用全生命周期管理。

本文于 2019 - 06 - 28 收到。

* 中国科学院战略性科技先导专项课题: SEANET 技术标准化研究与系统研制(编号: XDC02010701)。

按照 Web 运行环境的工作模式,可以将它分成三类,第一类是 Web 操作系统,即为 Web 应用设计的操作系统,它自身支持 Web 应用的运行[2],典型 Web 操作系统有 Chrome OS,Firefox OS 和 Tizen[3] 等;第二类是浏览器自身扩展了支持 Web 应用的能力,它的特性是 Web 应用都是由该运行环境管理,对于操作系统而言,它只支持本地应用,看不到 Web 应用的存在,典型代表有 Chrome、Crosswalk 的 Tizen 版等;第三类 Web 运行环境是以一个独立的框架存在于传统的操作系统,Web 运行环境自身作为本地应用需要操作系统的支持,而且它会将 Web 应用打包成本地应用,打包后的 Web 应用对操作系统来说就是一个本地应用,这是它与第二类 Web 运行环境的主要区别,但是 Web 应用的管理是由 Web 运行环境而不是操作系统管理的,典型的代表有 Crosswalk for Android 和 Cordova。

随着 HTML5 标准的发布和不断完善,Web 应用功能越来越丰富的同时,其性能也成为用户关注的焦点。Web 应用解析执行特性尤其是 JavaScript 语言动态类型、面向原型等特性成为限制 Web 应用性能的主要因素,因此如何提升 Web 应用性能一直是 Web 领域研究的热点。本文对主流的 Web 运行环境及其架构进行了介绍,并对 Web 应用解析执行过程中涉及的资源获取、解析执行等多个方面的相关优化技术进行了分析和总结。

1 主流 Web 运行环境及其架构

目前 PC 端主流 Web 运行环境有 Google Chrome、Microsoft IE 和 Mozilla Firefox,智能终端 Web 运行环境主要有 Tizen 和 Crosswalk。Chrome 浏览器一家独大,占领了超过一半的市场份额。IE 浏览器不开源,而开源项目 Crosswalk 目前也已停止维护,因此本文选择 Chrome、Firefox 和 Tizen 三种 Web 运行环境并对其架构进行介绍。

1.1 Chrome 和 Chrome OS

2008 年,Google 公司以苹果开源项目 Webkit 为内核创建了 Chromium 项目,该项目的目标是创建一个快速、支持众多操作系统的浏览器,并基于 Chromium 发布了自己的浏览器产品 Chrome。Chrome 发布后以其卓越的性能和优异的用户体验迅速占领浏览器市场,和 Microsoft IE、Mozilla Firefox 成为桌面系统上最流行的三款浏览器。目前 Chrome 浏览器已经发展成市场占用率最高的 Web 运行环境,据 2019 年 3 月 NetMarket-Share 数据显示,Chrome 浏览器的市场份额不低于 68.88%,是微软 Edge 的 10 倍以上。Chrome 浏览器的成功得益于其技术上的大胆创新以及对 Web 最新标准的支持。Chromium 作为 Chrome 的开源版本,众多创新技术会议先在 Chromium 上发布,待技术稳定后会应用于 Chrome 浏览器上,当然,Chrome 也会加入一些私有的解码器、音视频支持技术和 Google 服务等 Chromium 没有的功能。图 1 所示为 Chrome 多进程架构及核心模块图。Chrome 主要包含两种进程:主进程和渲染进程。渲染进程包含 JS 引擎、HTML 解释器和 CSS 解释器等模块,负责接收网页内容,对其进行解释执行,并进行布局和渲染,主进程负责将渲染进程的渲染结果进行显示,以及子进程管理、多窗口管理等。

Chrome 支持 Process - per - site - instance、Process - per - site、Process - per - tab 三种多进程策略和 Single process 一种单进程策略,多进程策略适用于 PC 端,不同 Web 应用或网页由多个渲染进程独立渲染,单个 Web 应用或网页的崩溃不会影响到其他进程,多进程策略有利于 Web 运行环境的稳定运行,但这样会导致资源开销大,因此在资源受限的智能终端设备上,一般都采用 Single process 单进程模式,多个 Web 应用或网页的渲染是由单个进程内的不同线程完成的。

Chrome OS 是 Google 开发的 Web 操作系统,它也是基于 Chromium 开发的,它的核心思想是基于 Linux 内核、一些第三库以及 Chromium 浏览器的能力打造一个支持 Web 应用的操作系统。图 2 为 Chrome OS 系统架构图[4],Chrome OS 作为 Web 操作系统,其架构主要包含 3 个部分:基于 Chromium 的浏览器和窗口管理系统;系统和用户级软件;定制化固件。窗口管理器负责用户和多个应用窗口的交互,包括控制窗口的定位,分配输入焦点等;系统和用户级软件主要提供如 NTP、syslog 等标准 Linux 服务和系统更新、连接管理、电

源管理等; 定制化固件主要为 Chrome OS 的安全和快速启动做优化,如去除一些不必要的组件,增加启动验证机制等。

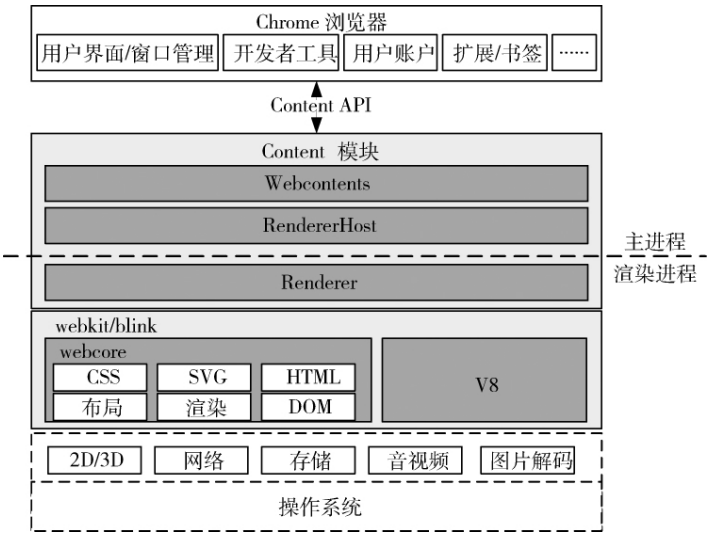

图 1　Chrome 架构图

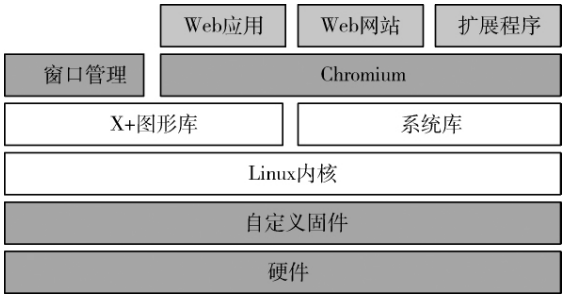

图 2　Chrome OS 架构图

1.2　Firefox OS

Firefox OS 是 Mozilla 在 2011 年为智能手机、平板电脑、智能电视等设备打造的移动 Web 操作系统,它是基于 Linux 内核和 Gecko 渲染引擎开发的。图 3 所示 Firefox OS 系统架构图[5],它主要包含 Gaia、Gecko 和 Gonk 三层。

Gaia 层是 Firefox OS 平台的用户接口层。设备的预装应用和用户自己安装的 Web 应用以及 lock screen、home screen 等都是由 Gaia 基于 Gecko 引擎渲染的,它本身也是一个由 HTML、CSS 和 JavaScript 开发的 Web 应用。Gecko 层是 Firefox OS 中的核心层,它是支持 Web 应用运行的 Web Runtime,它提供了上层 Web 应用需要的各种 Web APIs,其中包含 HTML5 等最新的 Web 标准,还提供了访问平台系统能力的接口,如 GPS、摄像头和传感器。Gonk 包括了 Linux 内核和众多的基础库,它类似于底层操作系统,是支持 Gecko 引擎的基础层。

1.3　Tizen

Tizen 是由英特尔和三星联合开源社区开发的一款 Web 操作系统,它是基于 Linux 内核和 Webkit 引擎开发的。不同于 Chrome OS 和 Firefox OS,Tizen 基于 Linux 内核、各种基础库和框架构建了 Tizen Web Framework 和 Tizen Native Framework,既可以支持 Web 应用也可以支持 native 应用。图 4 为 Tizen 2.0 系统中 Web 运行环境架构[6],其中主要包括安装模块、客户端

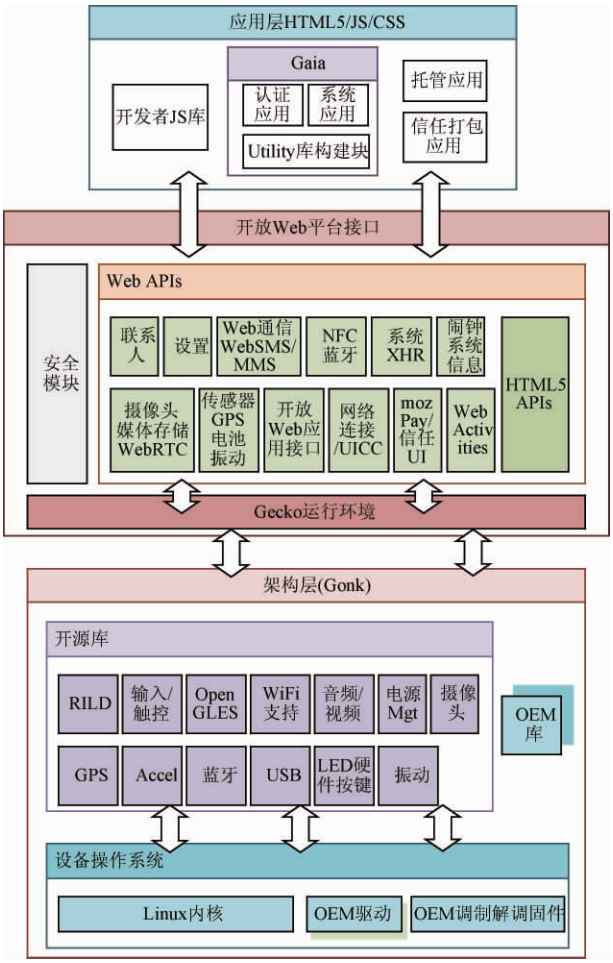

图 3　Firefox OS 架构图

模块和安全模块。安装模块主要负责 Web 应用的安装、卸载,安全模块通过内容安全策略(content security policy, CSP) 管理 Web 应用权限,客户端模块基于核心模块对 Web 应用进行全生命周期管理。2014 年,Tizen 3.0 系统将 Crosswalk 采用为新的 Web Runtime 来增强 Tizen 系统的 HTML5 功能。

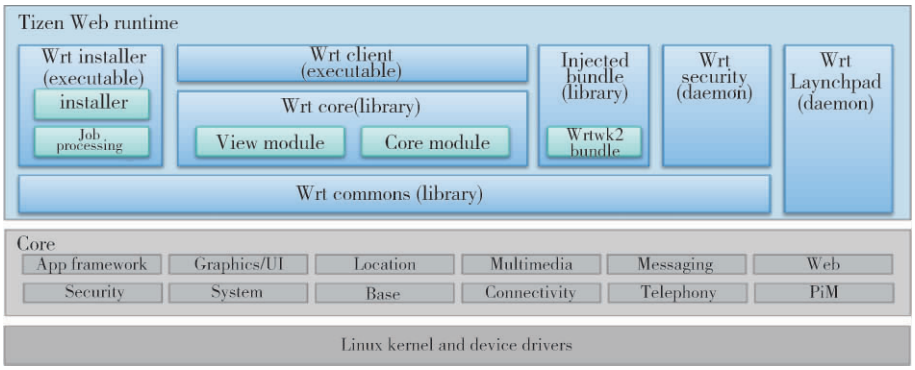

图 4 Tizen 2.0 系统中 Web 运行环境

2 Web 运行环境相关优化技术

Web 应用的执行包括建立网络连接、网页文件获取、网页文件的解析执行、布局渲染,因此现有 Web 应用的优化研究也涵盖了上述几个方面,包括采用高效网络栈、资源缓存、计算迁移、状态保存恢复、JavaScript 引擎加速等,下面分别进行介绍。

2.1 高效协议栈

2.1.1 长连接(Persistent connection) 和 HTTP 管线化(Pipelining)

早期的 HTTP 协议工作机制比较简单,默认情况下 HTTP 协议中每个 TCP 连接只能承载一个 HTTP 请求和响应,浏览器发送一个请求给服务器,服务端回复完成后随即断开连接,当浏览器需要请求新的资源时,需要重新建立 TCP 连接。随着 Web 技术的发展,网页文件中包含越来越多文件,尤其是图片和媒体文件,这会导致浏览器获取服务器资源的效率降低。HTTP1.1 增加了长连接机制,在建立 TCP 连接后,只要一方不显式断开连接,TCP 连接就不会断开。这样后续资源请求均可基于之前的 TCP 连接,减少了 TCP 连接建立和断开的时间,有利于加快网页文件响应速度。

HTTP1.1 还增加了管线化(Pipelining) 支持,引入管线化机制之前,浏览器和服务端采用一问一答的形式,浏览器在收到上一个 HTTP 请求的回复后才能发送 HTTP 下一个请求,基于管线化技术可以将多个 HTTP 请求一次性填充在一个 TCP 包内一次性提交给服务器,在发送过程中不需要等待服务端对上一个请求的回复,减少了网络上传输的 TCP 数据包,减轻了网络负载。HTTP 管线化基于长连接实现,需要浏览器和服务器两者配合才能实现。图 5 描述了 HTTP 管线化的运行机制。

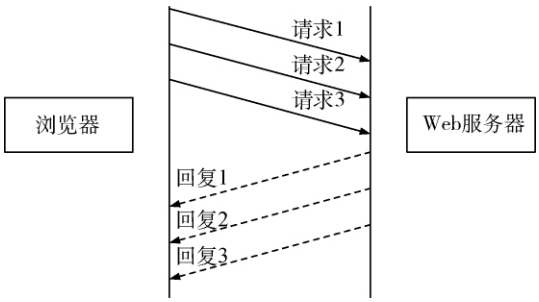

图 5 HTTP 管线化机制

管线化技术可以加快网页文件的加载时间,尤其在具有高延迟的网络连接环境中,在速度较快的网络连接环境中,管线化技术可能提速不明显,首先只有 GET 和 HEAD 等请求可以进行管线化,POST 请求不能进行管线化,使用场景有限; 其次,服务器端要按照请求顺序回复,这有可能会导致队头阻塞(head of line blocking, HOLB) 问题。

2.1.2 SPDY 协议

HTTP 管线化技术存在很大的限制和缺陷,为了解决网络延迟和安全性问题,Google 在 2012 年引入了

SPDY[7,8] 协议。SPDY 是一种基于 TCP 改进的多路复用传输协议。它被定义在网络协议栈的 HTTP 协议和 TCP 协议之间,图 6 描述了 SPDY 在协议栈中所处的层次。

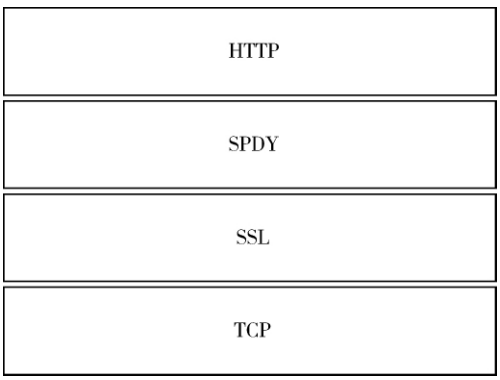

图 6　SPDY 协议层次图

SPDY 协议的核心思想是多路复用,使用单个 TCP 连接承载多个 HTTP 请求,这样一个网页中的众多资源可以基于单个 TCP 连接来传输,解决了 HTTP 管线化面临的队头阻塞问题。SPDY 的实现需要浏览器和服务器端协同合作,浏览器使用 SPDY 层对 HTTP 协议头进行封装,并使用 TCP 发送到服务器端,服务器端通过 SPDY 解释层解析 SPDY 协议并从中取出各个资源的 HTTP 头部。除了多路复用,SPDY 还有允许设置请求优先级、HTTP 头部压缩和服务端主动推送等特性。根据 Google 官方数据,SPDY 可以将网页加载时间减少 64%[2]。

2.1.3　QUIC 协议

为了解决 SPDY 协议底层基于 TCP 连接导致传输效率低的问题,Google 基于 UDP 实现了一种新型可靠、多路复用的网络传输协议 QUIC[9,10,11,12],它的主要目标是提供低延迟、高效率、安全加密的数据传输,提供更好的用户体验,它最初是 Google 于 2012 年开发的用于公司内部减少连接延迟的网络拥塞的协议,于 2015 年 6 月作为标准被提交给了 IETF。与 TCP 协议相比,它减少了延迟,而且它运行于用户空间。它的协议层次如图 7 所示[12],非常类似于在 UDP 上实现的 TCP + TLS + HTTP/2。

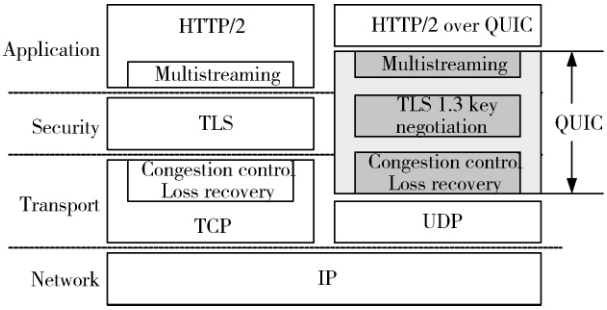

图 7　QUIC 协议层次图

和 TCP + TLS + HTTP2 相比,它的主要有以下特点: 首先,QUIC 通过在首次建立连接时保存 serverConfig 信息用于后续建立连接,实现了后续 0 - RTT 时间建立连接; 其次,QUIC 通过在应用程序层面实现不依赖操作系统的拥塞控制算法,采用单调递增的 Packet Number 解决 TCP 的重传歧义问题,采用 Tail Loss Probes (TLPs) 实现快速重传等方法大大改进了拥塞控制; 最后,QUIC 实现了多个同域名下的请求复用同一连接的多路复用方法,而且同一链路上的请求不再依次等待,可以间隔进行,解决了 TCP 协议队头阻塞问题(Head - Of - Line Blocking,HOLb) ; 然后,QUIC 协议的每个数据包除了包含本身的数据以外,会带有其他数据包的部分数据,在少量丢包的情况下,可以使用其他数据包的冗余数据完成数据组装而无需重传,提高数据的传输速度; 最后,QUIC 改进了传统 TCP 连接由客户端和服务端 IP 和端口组成的 4 元组标识的方法,采用客户机随机生成的 64 位 UUID 标识连接,当 QUIC 客户机更改 IP 地址时,它可以继续使用新 IP 地址中的旧连接 ID,快速重启会话,继续传输数据,实现连接迁移。

2.2　资源缓存

2.2.1　WebKit 内存缓存

资源的缓存机制是提高资源使用效率、减少网络开销、提升 Web 应用性能的有效方法。它会在内存中建立一个资源的缓存池来缓存之前请求过的资源,下次需要请求资源时,首先去缓存池中查找,如果有,WebKit 则取出使用; 如果没有,WebKit 则发送请求给服务器,WebKit 收到资源后将其缓存下来以供下次使用,图 8 显示了这一机制的原理。

Webkit 需要一定的机制来管理缓存的资源。首先是用于缓存资源的空间大小有限,需要有相应的机制来替换其中旧的资源,加入新的缓存资源,LRU 是最常用的缓存替换算法; 另一方面,内存缓存池中缓存的

资源可能是过期的，服务器端可能已经更新了资源，HTTP 协议提供了规范来让浏览器向服务器发送 HTTP 请求确认资源是否已更新。根据服务器端发送状态码，浏览器中 Webkit 可以判定资源是否有效，并决定是重新下载资源还是利用缓存池中的资源。

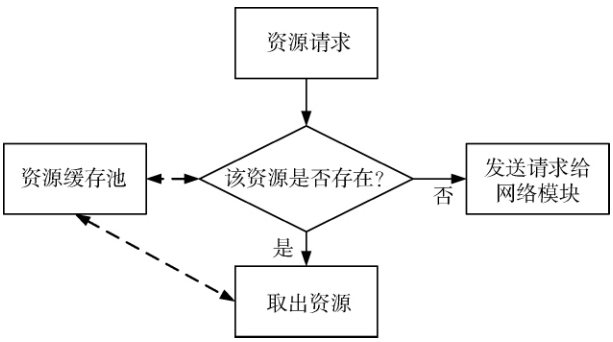

图 8 WebKit 资源缓存机制

2.2.2 磁盘本地缓存

如果浏览器没有磁盘缓存的话，每次浏览器访问网页都需要从网站下载网页文件和图片等资源，既浪费了资源又不利于网页快速加载。目前，绝大多数浏览器都有磁盘缓存机制，将之前访问网页时下载的资源保存下来，放到磁盘中以备后用。

2.3 计算迁移

随着各种智能终端设备的快速发展，智能终端设备功能越来越丰富，其上运行的应用也越来越复杂，但受制于智能终端设备体积、重量和移动性等因素，其上的存储、计算和电量等资源有限，无法很好地满足人们运行复杂应用的需求，解决这一问题的有效方案是计算迁移[13]（computation offloading），将运行于智能终端应用中计算密集型任务迁移到其他资源充足的设备上运行，然后将计算结果返回到智能终端设备。计算迁移的目的主要有 2 个：一是为了提高智能终端应用性能；二是为了节约智能终端资源。

面向 Web 应用的计算迁移[14-18]是计算迁移的重要研究方向，它主要是将 Web 应用中 JavaScript 相关的任务代码迁移到服务端运行，其中涉及到迁移代码的选择、迁移决策、迁移代码状态保存恢复以及迁移后代码和客户端代码的交互等。除了 JavaScript 代码迁移，还有一些研究将网页的部分渲染工作迁移到服务端去做，服务端将网页渲染的中间结果压缩返回到客户端，客户端只需要做简单的显示工作[19]。随着 HTML5 标准的发布，也有一些学者针对 HTML5 标准中 Web Worker、Web Storage 等某些特性进行迁移研究，最典型的是 HTML5 Web Worker[20-22]，它是 HTML5 标准提出的多线程机制，其相对独立而且承担的多是计算密集型任务，非常适合进行计算迁移。然而计算迁移也有代价，需要迁移决策模块参照当前的网络、计算能力等状况决定是否进行计算迁移。在涉及多个无线智能终端向同一个服务端进行计算迁移（mobile edge computing，MEC）时，多个终端的计算迁移会存在无线网络、计算资源等竞争，这时多个终端的迁移策略一般采用博弈论方法来解决。

2.4 状态保存恢复

状态保存恢复也是 Web 应用的性能优化方法之一，Web 应用的代码都是解释执行的，其运行速度比 C/C++ 等编译执行语言要慢，而且大部分 Web 应用启动过程中要加载大量的 JS 框架、图片资源等，这可能会导致 Web 应用启动耗时较大，因此，将 Web 应用启动后的状态保存下来，下次直接从状态文件启动，用状态文件的空间存储换取 Web 应用的启动速度加快。现有研究中文献[23]基于 Snapshot 方法将 Web 应用加载阶段的 JavaScript 代码的状态保存成状态文件以加速 Web 应用启动速度，如图 9 所示。除此之外，Chromium 浏览器中采用的 V8 引擎也采用了保存恢复技术来加速 V8 引擎的启动速度，采用序列化方法将 V8 引擎启动过程中内置 JavaScript 对象的状态进行保存。V8 引擎直接读取堆内存中保存的内置 JavaScript 对象的状态进行恢复，以加快 JavaScript 引擎的启动速度。

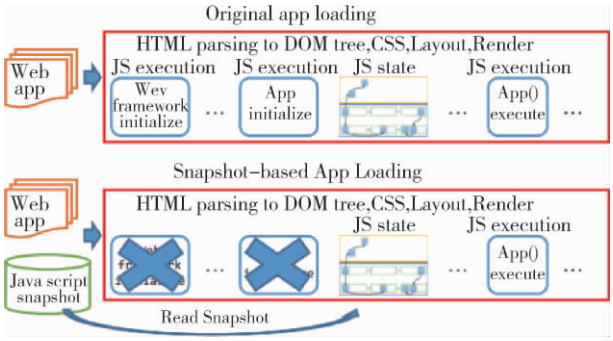

图 9 基于 Snapshot 的 Web 应用状态保存恢复

2.5　JS 引擎及其相关优化技术

目前主流的 JavaScript 引擎有 Google Chrome 浏览器使用的 V8 引擎,Apple Safari 浏览器使用的 JavaScriptCore 引擎和 Mozilla Firefox 浏览器使用的 SpiderMonkey 引擎,它们均采用了 JIT 编译器,但使用 JIT 编译器的思路又有所不同,据此,三种主流的浏览器对应的编译架构可以划分为两种: 第一种是以 JavaScriptCore 和 SpiderMonkey 引擎为代表的字节码解释器 + JIT 编译器编译架构,第二种是以 V8 引擎为代表的 JIT 全代码生成器编译架构。下面分别以 JavaScriptCore 引擎和 V8 引擎为例介绍两种主流编译架构。

JavaScriptCore 引擎采用的字节码解释器和多层 JIT 编译器的编译架构如图 10 所示。图 10 中第 1 层 LLINT interpreter 用于对字节码进行解释执行,第 2 层基础 JIT 编译器将某些执行频繁的热点函数的字节码编译成基础本地代码(Baseline Native code) ,对于更热点的函数采用第 3 层的 DFG Speculative JIT 编译器生成更优化的本地代码(DFG Native code) ,第 4 层 LLVM JIT 编译器还可以对代码进一步优化,生成执行效率更高的 LLVM 本地代码。但是 JavaScript 引擎中 JIT 编译器在对代码进行优化时需要进行类型特化,并在执行时进行 deoptimization check,如果检查失败,则会进行代码回退。图 10 中 OSREntry 表示代码优化,OSRExit 表示代码回退。

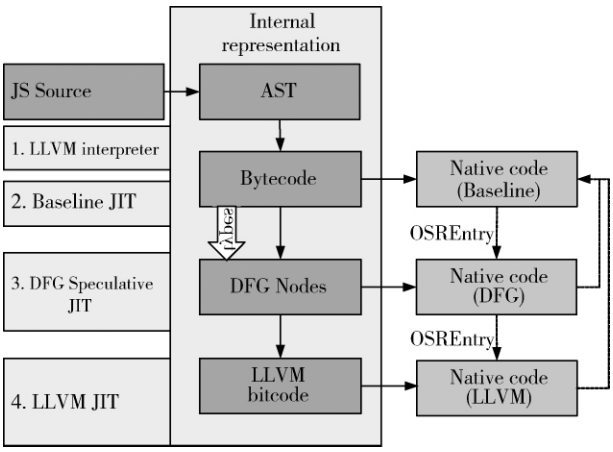

图 10　JSC 多层 JITC 架构

不同于 JavaScriptCore 引擎,V8 引擎采用了 JIT 全代码生成器架构,如图 11 所示,先将源码解释为抽象语法树(AST) ,然后将 AST 使用 JIT 编译器的全代码生成器生成本地代码,并辅以优化回滚、隐藏类、内嵌缓存等机制,很大地提升了 V8 引擎对 JavaScript 代码的执行速度。后来为了优化性能加入了 CrankShaft 编译器,它可以对全代码生成器生成的本地代码进行优化,但 V8 引擎 JIT 全代码生成器编译架构也带来了内存占用过大,缺少中间表示导致应用启动过程慢等缺点,因此 V8 引擎也在不断改进。2017 年 4 月 V8 发布的 5.9 版本中又引入了 Ignition 解释器和 TurboFan JIT 编译器[24] ,形成混合编译架构。在后来的 V8 引擎版本中,全代码生成器和 Crankshaft 编译器逐渐被去掉,形成了如图 12 所示的 Ignition + TurboFan 的全新编译架构。Ignition 解释器用于将源码解释为可执行的字节码,TurboFan 编译器对热点函数进行 JIT 编译,生成优化的本地代码,Ignition 解释器的引入宣告 V8 引擎也投入了字节码的怀抱。

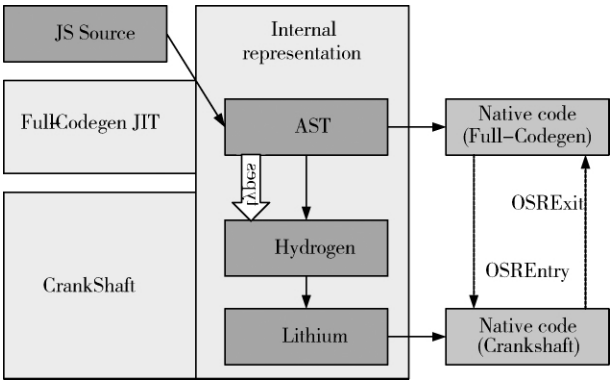

图 11　V8 引擎全代码生成器编译架构

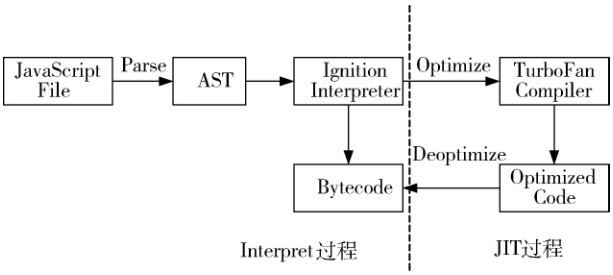

图 12　V8 引擎最新编译架构

目前 JavaScript 引擎优化相关技术可以分类为三类。第一类是对 JavaScript 引擎的即时编译(just - in - time compilation, JITC) [25,26] 过程进行优化; 第二类是基于 JIT 的预先编译(ahead - of - time compilation, AOTC) 方法[27-31] ; 第三类是 JavaScript 引擎并行化研究。

主流的 JavaScript 引擎均采用了即时编译机制,将热点函数和代码片段的中间表示(抽象语法树或字节

码) 编译为执行效率更高的机器码,下次热点代码再被调用时直接运行机器码。即时编译机制主要涉及 3 个方面的内容,首先,因为即时编译的代价较高,所以一般只对调用次数比较多的热点函数进行,通过热点函数的多次执行抵消掉一次即时编译的代价; 其次是在对热点函数的机器码进行调用时需要做类型检查,由于 JavaScript 是动态类型语言,所以在即时编译时需要确定代码中变量的类型,对代码做类型特化,即假定下次该代码运行时类型不会改变,但是 JavaScript 语言的动态类型特性随时可能改变变量类型,导致代码本次运行时变量的类型可能和之前机器码中特化的类型不一样,所以在调用机器码前需要做类型检查,即 deoptimization check,若类型没有发生变化,则直接调用机器码,或变量类型发生了变化导致 deoptimization check 失败,则转到相应的字节码处重新执行; 最后,即时编译也是一个不断优化的过程,所以很多 JavaScript 引擎采用了多层即时编译机制,在代码运行过程中不断收集信息,生成优化代码,对于执行频率一般的热点函数,采用基本的即时编译器生成一般的机器码,对于执行频率更高的热点函数,采用优化即时编译器生成更优化的机器码。JavaScriptCore 引擎便采用了典型的多层即时编译架构。

目前关于 JavaScript 引擎 JITC 的优化方法又可以分为两类,一类是硬件相关的优化,在使用 JIT 产生的机器码过程中每次都要进行 deoptimization check,平均情况下 12.9% [32] 的 JavaScript 运行时间花在了 deoptimization check 上,但实际上 deoptimization check 失败的次数相对较少,因此文献 [33] 针对不同类型的 deoptimization check 分别引入了不同的机器指令来替代原来的 check 指令以减少 deoptimization check 的时间代价,文献提出了低复杂度的硬件架构扩展 CheckedLoad[32],它引入了新的 ISA 指令替代软件 deoptimization check,同时对变量类型预测提出了硬件支持,减小 deoptimization check 失败的几率; 另一类是软件相关的优化,文献 [34,35] 分析了 JITC 过程中脱优化的代价,并提出了相应的建议以减少脱优化发生的次数。文献 [36 – 38] 从不同方面改进 JITC 过程中类型特化的机制。文献 [39] 对 JITC 过程中的热点探测算法进行改进,以更早更精确地发现热点代码,从而尽早对其优化,而且考虑到 deoptimization 的次数来对热点探测算法进行改进。文献 [40] 针对资源受限智能终端设备对 JITC 编译器进行了改进,生成尺寸更小的机器码指令以减少 JITC 的内存占用。文献 [41] 对 JITC 的过程中字节码的内存缓存机制进行了改进。

第二类 JavaScript 引擎优化技术主要是基于即时编译的预先编译方法。它的主要原理是将上一次网页或应用执行过程中生成的特定函数的字节码或机器码保存下来。下一次相同的 Web 应用再次运行时直接执行已保存的特定函数的字节码或机器码。预先编译和即时编译的不同之处在于即时编译是发生在单次 Web 应用或网页运行过程中的,是在网页和 Web 应用每次加载或运行的时候都要进行的,而预先编译面向的是 Web 应用多次运行过程,将上次网页和 Web 应用加载或运行的时候保存的机器码或字节码用于下次网页的加载或 Web 应用的运行。依据保存的代码类型不同,预先编译方法可以分为基于字节码的预先编译方法、基于机器码的预先编译和自适应预先编译方法。基于字节码的预先编译方法保存热点函数的字节码供下次使用,基于机器码的预先编译方法将热点函数的机器码或优化的机器码保存下来供下次使用,自适应预先编译方法综合基于字节码和基于机器码的预先编译方法,基于一定的代价模型对热点函数的字节码或机器码进行自适应选择。

第三类 JavaScript 引擎优化方法是充分利用多核处理器的优势对 JavaScript 引擎并行化研究[42,43],提升 JavaScript 引擎和 Web 应用性能。文献 [42] 针对串行 JavaScript 引擎提出了数学模型来检测串行 JavaScript 任务之间的依赖性并基于此设计了并行执行算法,并在 SquirrelFish Extreme 引擎上进行了实现。文献 [43] 使用线程级推测技术(thread – level speculation, TLS) 增强 Web 应用性能,它是一种能够动态对线性执行的程序进行并行性扩展的技术,其能够运用在硬件和软件环境下,线程级推测技术的一个主要方法就是将每个循环迭代方法分配给每个线程。

3 结束语

本文对主流 Web 运行环境架构及其在解析执行 Web 应用过程中的多种相关优化技术进行了介绍和分

析,对终端 Web 应用性能优化有重要的参考意义。

参 考 文 献

[1] 王鑫. Native App 与 Web App 移动应用发展[J]. 计算机系统应用, 2016(9): 250 – 253.

[2] 朱永盛. Webkit 技术内幕[M],北京: 电子工业出版社,2014.

[3] Web 应用和 Web 运行环境[OL]. http://www.ituring.com.cn/book/miniarticle/56260.

[4] The Chromium Projects. Software Architecture of Chrome OS[OL]. http://www.chromium.org/chromium – os/chromiumos – design – docs/software – architecture.

[5] Mozilla. B2G OS architecture[OL]. https://developer.mozilla.org/en – US/docs/Archive/B2G_OS/Architecture. (2019.3.18).

[6] Ming Jin. Tizen Web Runtime Update[OL]. http://download.tizen.org/misc/media/conferen 2013/slides TDC 2013 – Tizen_Web_Rontime_Update.pdf,2013.

[7] Mineki G, Uemura S, Hasegawa T. SPDY accelerator for improving Web access speed[C]// Advanced Communication Technology (ICACT), 2013 15th International Conference on. IEEE, 2013: 540 – 544.

[8] Jana, Rittwik, Ramakrishnan, et al. Towards a SPDY'ier Mobile Web? [J]. IEEE/ACM Transactions on Networking: A Joint Publication of the IEEE Communications Soceity, the IEEE Computer Society, and the ACM with Its Special Interest Group on Data Communication, 2013.

[9] Langley A, Iyengar J, Bailey J, et al. The QUIC Transport Protocol[C]//Unknown. Proceedings of the Conference of the ACM Special Interest Group on Data Communication – SIGCOMM'17. New York, New York, USA: ACM Press, 2017: 183 – 196.

[10] Y. Wang, K. Zhao, W. Li, et al. Performance Evaluation of QUIC with BBR in Satellite Internet[C]//2018: 195 – 199.

[11] S. Yang, H. Li, Q. Wu. Performance Analysis of QUIC Protocol in Integrated Satellites and Terrestrial Networks[C]//2018: 1425 – 1430.

[12] The Chromium Projects. QUIC, a multiplexed stream transport over UDP[OL]. https://dev.chromium.org/quic.

[13] Mach P, Becvar Z. Mobile Edge Computing: A Survey on Architecture and Computation Offloading[J]. IEEE Communications Surveys & Tutorials, 2017, 19(3): 1628 – 1656.

[14] Wang X S, Shen H, Wetherall D. Accelerating the mobile web with selective offloading[C]//Gerla, M; Huang, D. Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing – MCC'13. New York, New York, USA: ACM Press, 2013: 45.

[15] Yu M, Huang G, Wang X, et al. JavaScript Offloading for Web Applications in Mobile – Cloud Computing[C]//IEEE International Conference on Mobile Services. IEEE, 2015: 269 – 276.

[16] Li W YS, Wu S, Chan W K, et al. JSCloud: Toward Remote Execution of JavaScript Code on Handheld Devices[C]//12th International Conference on Quality Software. IEEE, 2012: 240 – 245.

[17] Hwang I, Ham J. Cloud Offloading Method for Web Applications[C]//2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering. IEEE, 2014: 246 – 247.

[18] Jeong H – J, Moon S – M. Offloading of Web Application Computations: A Snapshot – Based Approach[C]//IEEE 13th International Conference on Embedded and Ubiquitous Computing. IEEE, 2015: 90 – 97.

[19] SG N K, Madugundu P K, Bose J, et al. A Hybrid Web Rendering Framework on Cloud[C]//IEEE International Conference on Web Services (ICWS). IEEE, 2016: 602 – 608.

[20] Zhang X, Jeon W, Gibbs S, et al. Elastic HTML5: Workload Offloading Using Cloud – Based Web Workers and Storages for Mobile Devices[C]//International Workshop on Mobile Computing and Clouds (MobiClloud) 2010.

[21] Zbierski M, Makosiej P. Bring the Cloud to Your Mobile: Transparent Offloading of HTML5 Web Workers[C]//IEEE 6th International Conference on Cloud Computing Technology and Science. IEEE, 2014: 198 – 203.

[22] S. Kurumatani, M. Toyama, E. Y. Chen. Executing Client – Side Web Workers in the Cloud[C]//2012: 1 – 6.

[23] Oh J, Moon S – M. Snapshot – based loading – time acceleration for web applications[C]//IEEE/ACM International Symposium on Code Generation and Optimization (CGO). IEEE, 2015: 179 – 189.

[24] Ross McIlroy. Firing up the Ignition interpreter[OL]. https://v8.dev/blog/ignition – interpreter.

[25] Roo A de, Sözer H, Akşit M. Reuse of JIT compiled code based on binary code patching in JavaScript engine [J]. Information and Software Technology, 2012, 54(12): 1432 - 1453.
[26] Lee S - W, Moon S - M. Selective just - in - time compilation for client - side mobile javascript engine [C]//Gupta, R; Mooney, V. Proceedings of the 14th international conference on Compilers, architectures and synthesis for embedded systems - CASES'11. New York, New York, USA: ACM Press, 2011: 5.
[27] Park H, Kim S, Moon S - M. Advanced ahead - of - time compilation for Javascript engine [C]//Unknown. Proceedings of the 2017 International Conference on Compilers, Architectures and Synthesis for Embedded Systems Companion - CASES'17. New York, New York, USA: ACM Press, 2017: 1 - 2.
[28] Park H, Kim S, Park J - G, et al. Reusing the Optimized Code for JavaScript Ahead - of - Time Compilation [J]. ACM Transactions on Architecture and Code Optimization, 2019, 15(4): 1 - 20.
[29] Zhuykov R, Sharygin E. Ahead - of - time compilation of JavaScript programs [J]. Programming and Computer Software, 2017, 43(1): 51 - 59.
[30] Zhuykov R, Vardanyan V, Melnik D, et al. Augmenting JavaScript JIT with ahead - of - time compilation [C]//Computer Science and Information Technologies (CSIT). IEEE, 2015: 116 - 120.
[31] Park H, Jung W, Moon S - M. Javascript ahead - of - time compilation for embedded web platform [C]//13th IEEE Symposium on Embedded Systems For Real - time Multimedia (ESTIMedia). IEEE, 2015: 1 - 9.
[32] Anderson O, Fortuna E, Ceze L, et al. Checked Load: Architectural support for JavaScript type - checking on mobile processors [C]//IEEE 17th International Symposium on High Performance Computer Architecture. IEEE, 2011: 419 - 430.
[33] Dot G, Martinez A, Gonzalez A. Analysis and Optimization of Engines for Dynamically Typed Languages [C]//27th International Symposium on Computer Architecture and High Performance Computing (SBAC - PAD). IEEE, 2015: 41 - 48.
[34] Southern G, Renau J. Overhead of deoptimization checks in the V8 javascript engine [C]//IEEE International Symposium on Workload Characterization (IISWC). IEEE, 2016: 1 - 10.
[35] 黎遇军,张昱. JavaScript 引擎中 Deoptimization 模式分析与改善 [J]. 电子技术, 2017(7): 23 - 29.
[36] Kedlaya M N, Roesch J, Robatmili B, et al. Improved type specialization for dynamic scripting languages [C]//Hosking, A; Eugster, P; Bolz, C F. Proceedings of the 9th symposium on Dynamic languages - DLS'13. New York, New York, USA: ACM Press, 2013: 37 - 48.
[37] Rafael de Assis Costa I, Nazaré Santos H, Rafael Alves P, et al. Just - in - time value specialization [J]. Computer Languages, Systems & Structures, 2014, 40(2): 37 - 52.
[38] Gal A, Orendorff J, Ruderman J, et al. Trace - based just - in - time type specialization for dynamic languages [J]. ACM SIGPLAN Notices, 2009, 44(6): 465.
[39] Duan H, Ni H, Deng F, et al. A hybrid hot spot detection algorithm for JavaScript just - in - time compiler [J]. Journal of Computational Information Systems, 2014, 10(15): 6615 - 6624.
[40] Lee S - W, Moon S - M, Jung W - K, et al. Code size and performance optimization for mobile JavaScript just - in - time compiler [C]//Jones, A K; Yuan, X. Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture - INTERACT - 14. New York, New York, USA: ACM Press, 2010: 1.
[41] Heo J, Woo S, Jang H, et al. Improving JavaScript performance via efficient in - memory bytecode caching [C]//IEEE International Conference on Consumer Electronics - Asia (ICCE - Asia). IEEE, 2016: 1 - 4.
[42] Hucai D, Hong N, Feng D, et al. A Parallelization Design of JavaScript Execution Engine [J]. International Journal of Multimedia and Ubiquitous Engineering, 2014, 9(7): 171 - 184.
[43] Martinsen J K, Grahn H, Isberg A. Using speculation to enhance javascript performance in web applications [J]. IEEE Internet Computing, 2013, 17(2): 10 - 19.

作者简介

王昭,(1992 -),男,博士,主要研究方向为嵌入式 Web 引擎,Docker 虚拟化技术。
邓浩江,(1971 -),男,博士,研究员,主要研究方向为多媒数字信号处理,多媒体通信。
朱小勇,(1982 -),男,博士,研究员,主要研究方向为轻量级虚拟化技术,嵌入式系统 Web 引擎。
胡琳琳,(1980 -),女,硕士,副研究员,主要研究方向为 Web 引擎,Web 操作系统,容器技术。