

本科生学士学位论文

一种云端协同的移动浏览器缓存优化框架实现

Implementation of Cloud-Client Collaborative

Cache Optimization Framework in Mobile Web Browser

姓 名： 王鑫洋

学 号： 1100012752

院 系： 信息科学技术学院

专 业： 计算机科学与技术

指导老师： 刘譞哲 副教授

二〇一五年五月

摘 要

在智能手机和平板电脑等移动设备上，浏览器是用户来访问互联网服务的主要途径之一。然而，众所周知地，现有的移动浏览器用户体验并不理想：网页加载速度慢，流量消耗较多，电量消耗也是一个不容忽视的问题。

浏览器的缓存机制是解决上述问题的有效手段，但是，我们研究发现，处于种种原因，现有浏览器缓存的性能并不理想，存在很大的优化空间。因此，在不改变现有用户浏览习惯与Web应用缓存原有配置的前提下，本文设计并实现了一套云端协同的优化浏览器缓存的解决方案，通过资源校验码比对、数据压缩与网站预取的手段，最大程度减少传输流量并提高加载速度。实验表明，该系统在用户第一次访问网页时，平均可以节省流量17%，提高加载速度29%；立刻再次访问同一网页，平均可以为第二次访问节省流量66%，提高加载速度9%。

本文的主要工作如下：

* 分析了当前浏览器的缓存机制与潜在的缺陷，分析了缓存的理论性能上限与当前性能间的差距及造成的原因，并提出了优化方案。
* 针对提出的优化方案，本文设计并实现了基于代理的移动浏览器缓存优化系统。该系统通过移动设备端的后台服务和云端服务器对浏览器请求资源的记录、选择性推送与优化传输，实现了对现有缓存机制的优化。
* 通过对比缓存优化前后用户加载网站所用的时间与流量，验证了本文的缓存优化系统对提升网页加载速度和节省流量的有效性。

**关键词：移动浏览器，浏览器缓存，浏览器代理**

## Abstract

On the mobile devices like smartphones and tablet computers, Web browser is one of the main entrance of the Internet. However, the existing mobile browsers have not considered user experience very well. The page load speed is slow. The data usage is wasteful. And the energy consumption is too much.

The cache mechanism is an effective way to solve the problems above, but the existing browser’s cache performance is not satisfying. It has a huge space to improve. Therefore, without the change of user browsing habits and the configuration of Web applications, this thesis designs and implements a cache optimization system. With checksum comparison, data compression and website prefetching methods. This system can reduce the data usage and increase the load speed. The experiment shows that this system can save data usage by 17% and load time by 29% on average when users load a Web page at the first time. When users load the same page immediately, this system can save data usage by 66% and load time by 9% for this second load.

The contributions of this thesis are as follows.

* This thesis introduces existing browser’s cache mechanism and potential defects, analyses of the upper bound of cache performance and the cause of the gap between actual and ideal performance, raises the optimization scheme.
* With the optimization scheme, this thesis designs and implements the proxy-based cache optimization system for mobile Web browser. This system consists of a system service on mobile devices and a personal proxy server, which records browser’s requests, selectively pushes resources and optimizes the transition.
* This thesis measures the Web page load time and data usage with and without optimization, and proves that the proposed system can both speed up the page load and reduce the data usage for mobile Web browsing.

**Key Words: Mobile Web Browser, Browser cache, Browser Proxy**

## 目 录

[Abstract 3](#_Toc481765140)

[目 录 4](#_Toc481765141)

[第一章 引言 1](#_Toc481765142)

[1.1 移动浏览器的应用现状 1](#_Toc481765143)

[1.2 现有缓存机制的分析与应用 1](#_Toc481765144)

[1.2.1 浏览器缓存原理 1](#_Toc481765145)

[1.2.2 现有缓存机制的潜在缺陷 1](#_Toc481765146)

[1.3 本文的主要工作与结构 1](#_Toc481765147)

[1.3.1 本文的主要工作 1](#_Toc481765148)

[1.3.2 本文的结构 1](#_Toc481765149)

[第二章 现有缓存机制优化空间的分析 1](#_Toc481765150)

[2.1 缓存的理想性能与现有缓存的改进空间 1](#_Toc481765151)

[2.2 冗余传输的问题 1](#_Toc481765152)

[2.3 冗余传输的来源 1](#_Toc481765153)

[2.4 本文的优化方案 1](#_Toc481765154)

[2.4.1 缓存优化的可行手段 1](#_Toc481765155)

[2.4.2 本文方案的系统结构 1](#_Toc481765156)

[2.4.3 本文的缓存优化方案 1](#_Toc481765157)

[第三章 asd 1](#_Toc481765158)

[第三章 asd 1](#_Toc481765159)

[3.1 面向对象的分析 1](#_Toc481765160)

[3.1.1 用况图 1](#_Toc481765161)

[3.1.2 顺序图 2](#_Toc481765162)

[3.1.3 类图 7](#_Toc481765163)

[3.1.4 状态机图 8](#_Toc481765164)

[3.2 面向对象的设计 9](#_Toc481765165)

[3.2.1 系统环境 9](#_Toc481765166)

[3.2.2 系统架构图 10](#_Toc481765167)

[3.2.3 类图 11](#_Toc481765168)

[3.2.4 数据库设计 12](#_Toc481765169)

[第四章 移动浏览器缓存优化的实现 13](#_Toc481765170)

[4.1 本文使用的相关工具 13](#_Toc481765171)

[4.1.1 SPDY协议 13](#_Toc481765172)

[4.1.2 Jetty 14](#_Toc481765173)

[4.1.3 Chromium Embedded Framework (CEF) 14](#_Toc481765174)

[4.2 手机端代理服务 15](#_Toc481765175)

[4.2.1 RequestHandler 15](#_Toc481765176)

[4.2.2 ServerpushHandler 16](#_Toc481765177)

[4.2.3 MemCacheManager 17](#_Toc481765178)

[4.3 在线访问服务器 18](#_Toc481765179)

[4.3.1 JettySPDYServer 18](#_Toc481765180)

[4.3.2 ResourceHandler 18](#_Toc481765181)

[4.3.3 ResourceRequester 19](#_Toc481765182)

[4.3.4 ResourcePusher 19](#_Toc481765183)

[4.4 离线网站采集程序 21](#_Toc481765184)

[4.4.1 JCEF的配置与使用 21](#_Toc481765185)

[4.4.2 JCEFController 21](#_Toc481765186)

[4.4.3 ServerUpdater 22](#_Toc481765187)

[第五章 移动浏览器缓存优化的实验验证与性能分析 23](#_Toc481765188)

[5.1 实验设计 23](#_Toc481765189)

[5.1.1 实验方案 23](#_Toc481765190)

[5.1.2 网站的选取 23](#_Toc481765191)

[5.2 自动化测试系统的设计 23](#_Toc481765192)

[5.2.1 系统流程图 23](#_Toc481765193)

[5.2.2 实验数据收集方案 24](#_Toc481765194)

[5.2.3 系统实现方案 25](#_Toc481765195)

[5.3 自动化测试系统的实现 26](#_Toc481765196)

[5.3.1 自动化测试用Firefox插件 26](#_Toc481765197)

[5.3.2 自动化测试控制服务器 28](#_Toc481765198)

[5.4 实验结果 29](#_Toc481765199)

[5.4.1 性能评价方法 29](#_Toc481765200)

[5.4.2 实验结果 29](#_Toc481765201)

[5.4.3 实验结果讨论 31](#_Toc481765202)

[5.5 本系统的额外开销 32](#_Toc481765203)

[5.5.1 额外流量开销 32](#_Toc481765204)

[5.5.2 额外性能开销 32](#_Toc481765205)

[第六章 工作总结与展望 34](#_Toc481765206)

[6.1 本文工作总结 34](#_Toc481765207)

[6.2 未来工作展望 34](#_Toc481765208)

[参考文献 36](#_Toc481765209)

[致 谢 37](#_Toc481765210)

## 第一章 引言

1.1 移动浏览器的应用现状

根据中国互联网络信息中心于2015年1月发布的《第35次中国互联网络发展状况统计报告》，截至2014年12月，我国手机网民规模达5.57亿，较2013年增加5672万人。网民中使用手机上网的人群占比由2013年的81.0%提升至85.8%。连续几年来，手机网民规模保持稳定增长，手机也早已成为第一大上网终端。

越来越多的人们正在利用手机或者平板电脑来通过Wi-Fi或3G网络上网，随之而来的，Web浏览器成为了移动设备上最常用的应用之一。根据德克萨斯大学2014年的研究，浏览器占据了移动设备63%的显示时间。但是，现有的这些浏览器带来的用户体验却远不尽如人意。资源加载是影响浏览器性能重要因素之一，有实验表明，网页加载65%的时间都花在了资源加载上。由于移动设备的网络状况通常并不理想，尤其是可能出现网络拥挤或是用户正在移动的状况，这个问题就变得更为严重了。此外，下载资源的过程本身会带来非常严重的电量消耗，而电量恰恰是移动设备上最为稀缺的资源。

为了加速Web资源加载，缓存机制被广泛运用。通过将已访问网站的资源存储到本地（内存或磁盘），当用户再次访问这些网站时，就可以尝试从本地取这些资源，而不用再从远程Web服务器上请求了。通过上述缓存的方法，可以为用户减少资源下载量，从而提高Web浏览性能。这种缓存机制给移动设备带来的好处更大，因为在移动设备上，该机制不单可以减少网页加载时间，还可以减少网络流量和电量消耗。

然而，虽然缓存机制已经被详细定义在了HTTP协议中，在实际运用中，缓存却并未被很好地配置与实现。此前的研究表明，缓存机制的理论优化空间很大，而测得的实际性能却并不如人意。因此当前有这样一个需求，在不改变现有浏览器与服务器实现的前提下，对现有的缓存机制进行优化，提高缓存的利用效果。希望可以以此进一步提高移动设备用户在浏览网页时的体验，减少用户的流量与电量消耗。

1.2 现有缓存机制的分析与应用

1.2.1 浏览器缓存原理

本文只分析HTTP协议中规定的缓存以及浏览器对其的实现。

浏览器为了加载一个网页，需要向服务器请求每一个资源，而请求每一个资源需要向指定服务器发出请求资源的HTTP头，服务器会根据这一请求返回相应的HTTP响应头以及资源体，浏览器接收到资源并进行渲染，同时可能将资源进行缓存。浏览器是否该缓存此资源，以及何时可以调用缓存的信息就包含在HTTP响应头中。下面来用一个HTTP响应头的例子来说明服务器是如何通过HTTP响应头配置缓存的。

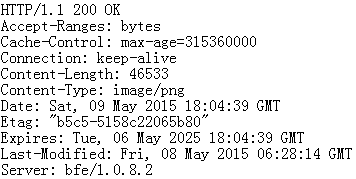
  
图1：2015年5月10日请求百度首页Logo资源的响应头

图1是响应头的示例，此处关注响应头中的以下部分：

**响应状态行：**样例中的第一行。当浏览器有过期的缓存时，可以向服务器验证缓存的有效性，此处为304 Not Modified时表示验证通过，缓存有效，不返回资源本体，浏览器使用缓存的资源渲染网页；此处为200 OK表示验证不通过，服务器会返回完整的资源。

**Cache-Control域：**这里标示了缓存策略，如是否应该缓存。可以用max-age标示了缓存的有效时间。

**Cache-Control的max-age设置与Expires域：**均标示缓存的有效期限，max-age含义为多少秒后过期，Expires标明过期的时间点。（当响应头的Cache-Control中设定了max-age时，优先级高于Expires）浏览器不能直接使用过期的缓存渲染页面。

**Etag域：**此处内容为该资源在服务器上的唯一标示符，通常为文件信息的hash值，不同的资源有不同的Etag信息，在后面提到的缓存验证中有作用。

**Last-Modified域：**此处标示该资源在服务器上最近一次更新的时间，在后面提到的缓存验证中有作用。

需要注意的是，这些缓存相关的信息并不是网站必须提供的，好的网站设计者会通过配置这些参数来提高用户加载网站的速度，当没有这些配置信息时，浏览器可以通过自身的启发式算法来配置缓存的过期时限。上面提到的是服务器对于资源的缓存设置，浏览器在接收到这些信息之后会记录并采用这些设置。浏览器判断是否存在缓存的唯一依据是资源的URL，在第二次访问请求此URL时，浏览器是使用缓存的方法是这样的，如图2所示。

  
图2：第二次请求同一URL，浏览器使用缓存的流程图

1.2.2 现有缓存机制的潜在缺陷

根据上一节的分析，可以发现，有一些情况下，缓存不能被很好地利用：

* **URL变化的相同资源：**由于URL是缓存利用的唯一索引，在某些情况下，当同一资源在服务器上的URL发生改变时，可能此前浏览器曾经缓存过资源本体，当前的缓存机制还不能解决由此带来的额外流量开销。
* **不必要的资源传输：**服务器端会有一些资源没有设置Last-Modified或Etag，在这种情况下，如果缓存过期，浏览器将不得不从服务器请求一份新资源，即使内容与原缓存一致。当缓存有效期较短，用户访问较频繁时，会给用户带来大量不必要的流量开销。
* **错误缓存问题：**当缓存未过期时，浏览器从本地缓存中读取资源并渲染网页，然而，有一些资源已经在服务器上更新了，浏览器却错误地使用了过期的资源，而没有从服务器上重新请求这些新资源，本文称这种情况为错误缓存。错误缓存可能会极大地影响用户的浏览体验，尤其是当如CSS或JavaScript这种功能型资源发生错误缓存时，相关的网站可能会产生完全错误的显示。
* **缓存时限设置不当：**网站的开发者设置一些资源的有效期可能是不准确的，还有些网站的开发者不设置有效期，此时浏览器会通过自身算法“假设”资源的有效期。这种并不可靠的有效期可能会带来冗余传输、错误缓存和不必要的资源验证。

以上是根据缓存机制可以分析出的现有缓存的潜在缺陷。下一章中将分析每一种缺陷的成因和对缓存性能的影响程度，以及改良这些缺陷后缓存性能的提升量。

1.3 本文的主要工作与结构

1.3.1 本文的主要工作

本文的主要工作包括：

* 介绍了当前浏览器的缓存机制与潜在的缺陷，分析了缓存的理论性能上限与当前性能间的差距及造成的原因，验证了实现浏览器的缓存优化是有意义而且可行的，并提出了优化方案。
* 针对提出的优化方案，本文设计并实现了基于代理的移动浏览器缓存优化系统，该系统有三个子系统组成：手机端后台服务、服务器端在线服务、服务器端离线资源抓取程序。该系统通过移动设备端的后台服务和云端服务器对浏览器请求资源的记录、选择性推送与优化传输，实现了对现有缓存机制的优化。
* 设计实验验证了本文的缓存优化系统对提升网页加载速度和节省流量的有效性。实验表明，该系统在用户第一次访问网页时，平均可以节省流量17%，提高加载速度29%；立刻再次访问同一网页，平均可以为第二次访问节省流量66%，提高加载速度9%。

1.3.2 本文的结构

本文的结构如下：

第一章呈现了移动浏览器的应用现状，对现有移动浏览器的缓存机制进行了分析，并指出了现有缓存的潜在缺陷。

第二章分析了缓存的理论上限与当前性能间的差距及造成的原因，验证了实现浏览器的缓存优化是有意义而且可行的，并对可行的缓存优化手段进行了分析，提出了优化方案。

第三章基于面向对象的方法对本文的系统进行了分析与设计。

第四章给出了移动浏览器缓存优化系统的具体实现细节。

第五章对本系统进行了实验设计，并描述了自动化测试系统的设计与实现，展现了实验的结果，最后分析了本系统带来的额外开销。

第六章对全文的工作进行了回顾和总结，并提出了未来工作的三个方向。

## 第二章 现有缓存机制优化空间的分析

基于小组之前的工作[Ma 2015]，本文给出对现有缓存机制问题所在及其潜在的可提升空间的分析。此分析将对设计针对性方案有帮助，且能够确保基于缓存的优化是有意义的。文中进行了相关实验，实验方法简述如下：

* 使用PC版的Chrome内置的模拟器来模拟Android 4.2原生浏览器，并通过Charles代理来收集HTTP记录。
* 选取Alexa前100个网站，并从中去掉无法访问的网站、域名不同的相同网站和只使用HTTPS的网站。最后得到了55个网站作为测试集。
* 在一周的时间里，使用模拟器每隔30分钟访问上述测试集中每个网站一次，最终从Charles收集到了相关的资源请求记录。

本文对文章中的数据进行了整理，对缓存机制的可优化空间进行了讨论，同时分析了可以通过缓存解决的冗余传输问题与造成问题的根源。

2.1 缓存的理想性能与现有缓存的改进空间

为了得到缓存对于浏览器的性能提升程度，可以计算在理想情况下，Web资源的访问中，有多少可以通过缓存的方式节省下来。为此，对测试集中的所有网站，本文计算在相隔一段时间后的第二次访问中，从缓存里可以读出的资源上限，得到了以下数据与结论：

* **可缓存的资源在全部资源中所占的比重很高，因此缓存是有效的：**对于所测网站，通过缓存可以节省56.4%-72.7%的数据流量（具体值取决于两次访问的间隔，间隔约长，该数值越低）。因此，当缓存被正确配置与实现时，缓存对于移动Web浏览是很有帮助的，尤其是对那些经常访问的网站（如新闻、搜索引擎），缓存机制更是有能力显著减少Web浏览带来的网络流量上、计算上和能源上的消耗。
* **不同类型的资源的缓存利用能力是不一样的：**通过测试得到的结论，缓存可以节省HTML类型的资源4.5-9.1%的数据流量，这个原因可能是由于现在的网站为了应对用户需求，变得越来越动态化，PHP等动态Web技术可以动态修改服务器端发送的HTML的内容，从而使HTML变得更难缓存。相比之下，缓存机制最容易发挥在CSS上，可以节省74.6-89.4%的流量，这一点也恰好揭示了CSS的本质，即在网页内容频繁改变时保证网页布局与风格稳定。

上述分析表明了缓存的有效性，本文还需要讨论的是优化现在的缓存机制是否有必要，因此接下来讨论，当前浏览器访问Web资源的过程中，缓存实际为用户节省了多少流量。对测试集中的所有网站，文章计算了在相隔一段时间后的第二次访问中，浏览器实际利用了多少已缓存的资源来访问一个网站，有以下发现：

* **比较小的资源并没有很好地利用缓存：**通过分析相隔24小时访问测试集的数据，可以看出，所测网站的缓存可以节省80%左右的流量，但却只有不到50%的资源数目被缓存利用。这说明在Top网站中，设计者比较重视对大资源的缓存配置，而小资源的缓存也存在一定可利用程度，不应该被完全忽视。

总体来看，缓存的理想运用与实际效果之间还是存在很大差距的。对于所选取的网站，这个差距主要表现在大量的小资源没有得到很好的缓存配置。

2.2 冗余传输的问题

当用户访问网站时，有一些内容可以从缓存中正确地获取，然而浏览器却从原网站上请求并下载了相同的资源。这种通过网络传输的是不必要的，即冗余传输。冗余传输会带来不必要的流量开销、减慢网页加载速度，从而严重影响用户的浏览体验。文章定义了网站访问的流量冗余率，即在相隔一段时间后的第二次访问中，可以从缓存中读取但却从网络传输的数据流量占总网络数据流量的比率。通过对流量冗余率的分析，可以得到以下结论：

* **冗余传输在移动浏览中问题普遍：**考虑相隔时间为24小时的情况，访问网站的流量冗余率为10%。具体而言，在一次相隔24小时的访问中，分别有10%的流量是可以通过缓存节省，但却没有节省下来。由此可见，即使选择的是制作非常专业的流行网站作为测试集，也存在一定的冗余传输，对于比较普通的小网站来说，这个问题就会更加明显。
* **访问时间间隔越短，冗余传输越严重：**当访问时间间隔缩短时，网站的冗余传输率均有所上升。在最差的情况下，相隔0.5小时的访问，有30%的网站的冗余率高于60%，而相隔24小时的访问，这些网站的冗余率为高于30%。因此，在频繁访问网站时，有大量的数据传输都是冗余的，是可以为用户剩下来的。此外，前面曾提到过，访问时间间隔越短，缓存的可利用空间越大。由此可见，现有的缓存机制还有很大的改进空间。
* **有一些网站的传输是完全冗余的：**实验还发现，在间隔24小时的访问中，有10%的网站的冗余率为100%，这些网站中包括百度首页、维基百科和苹果首页等。作为搜索工具与流行网站，用户可能每天都访问它们，而缓存机制本可以让他们在访问这些网站时无需任何网络传输，但事实上用户却始终浪费着大量的网络流量和电量。

2.3 冗余传输的来源

为了研究清楚冗余传输问题的来源，文章将造成资源可以从缓存中读取但却从网络传输的原因分成如下六类：

* Same Content：URL不同但已经缓存过存在相同的内容
* Heuristic：服务器未指定过期时间，但浏览器的启发式算法认为缓存已过期
* No store：服务器认为资源为No-Store，即不应在磁盘上保留
* No cache：服务器认为资源为No-Cache，浏览器需向服务器验证缓存有效性
* Checked：服务器指定过期时间，资源已过期，服务器验证通过
* No check：服务器指定过期时间，资源已过期，服务器验证未通过

当发生上述六种情况之一时，网站并不一定会传给移动设备完整的资源，有的只是进行数据的有效性的验证，但这个过程也需要网络传输。文章将每一次有上述问题的资源都归为这六类之一，分析得出以下结论：

* **内容相同的情况与启发式算法出错的情况是造成冗余传输最多的两个因素：**在所测网站中，发生内容相同但URL不同的情况占比最多，其次是浏览器启发式算法造成缓存未利用的情况。此外，访问间隔时间不会影响这两种情况的比重。
* **缓存过期，但服务器验证通过的情况占到第三位：**当缓存超过了服务器指定的时间，在一些情况下，如上文所述，浏览器可以向服务器请求验证缓存的有效性，即通过发送资源的Last-Modified或Etag值来实现的，此时服务器传回资源未改变的响应头，无需传输资源本体。造成这种情况的原因是对缓存时效性的设置过短。

本文首先来关注内容相同但URL不同的情况。由于HTTP标准规定以URL作为浏览器缓存的索引，因此当URL发生变化时，浏览器缓存是无法使用此前存下来的缓存的。通过查看资源请求的记录，首先发现有以下两个原因可能造成URL改变的情况：

* **资源版本管理：**为了始终给用户提供最新的资源，有一种Web开发的手段，将随机的查询字符串附在资源的URL后面。这样每次浏览器访问这些资源时，都会因URL不同而把它们当成全新的资源，从而从网络上重新下载。还有一种情况，比如说优酷网，在每一次网站更新时会使用一个新的目录来存资源，从而导致所有资源的URL发生变化。由于在一次网站更新中，一般不会所有的资源都发生变化，因此在新的目录下也会存在那些可以从缓存直接使用的老资源。
* **网站层次架构：**目前的主流网站普遍采用内容分发网络（CDN）与反向代理来提升访问速度。为了使用这些中间服务器传输，Web服务器会生成不同的URL或者在URL上额外的信息，从而达到控制路由的目的。由此带来的URL改变就可能会导致缓存的失效。

接下来是缓存过期的问题。有时开发者并不注意缓存的重要性，从而没有给某些资源配置缓存过期时间，此时浏览器只能去靠启发式算法猜测过期时间。在所测网站中，该问题的占比也达到了第二位。文章发现，在这些网站中，有6%的可缓存资源没有配置过期时间。也就是说，这些资源是否被缓存利用，完全取决于浏览器算法，而非网站开发者本身。这将更可能会带来额外流量消耗或缓存错误使用的问题。

对于那些被配置了缓存的资源，配置的参数也未必正确。在所测网站中，有40%的资源被配置了少于一天的过期时间，在这其中，文章却发现有高达89%的资源在其过期时间内其实并没有更新。也就是说，目前的过期时间还有可以延长的空间，然而，延长时间可能会带来更多的缓存错误使用的情况，因此，对于资源过期时间的设置是一个需要开发者权衡的问题。

2.4 本文的优化方案

2.4.1 缓存优化的可行手段

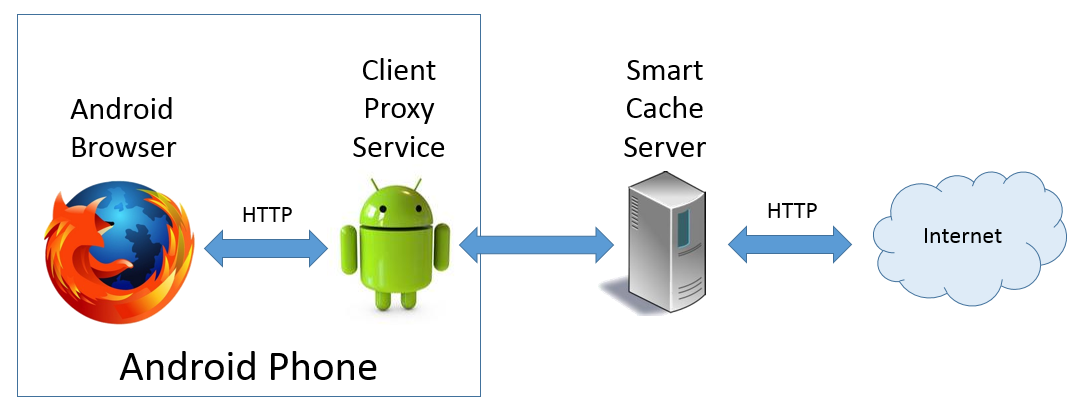
通过上文的分析，可以发现，缓存仍有很大的提升空间。进行缓存优化的技术手段有很多，下面进行简单的介绍，并分析各种手段的利弊：

* **修改浏览器：**通过修改浏览器，可以改变浏览器的资源加载方法与缓存使用方法，从而达到修改浏览器资源加载机制的目的。但是，这种方法仅限于修改现有缓存使用的机制，无法进一步优化。没有云端服务器的帮助，单靠手机端无法检测到不同URL的相同资源。这种方法的好处是优化带来的额外性能开销会比较少。然而，这种方法的缺点很明显，首先需要改变用户的浏览器使用习惯，需要让用户使用指定的浏览器；其次，修改主流浏览器的资源加载与缓存方法涉及到修改浏览器内核，难度很大，而修改基于WebView的浏览器虽然难度小，但浏览器本身效率远不如主流的Firefox或Chrome等浏览器。
* **在手机端搭建代理服务器：**这种方法与修改浏览器的方法类似，本质上也是改变浏览器的资源加载方法，且仍没有云端服务器的帮助。这种方法的优点是无需改变用户习惯，只需要为浏览器配置HTTP代理即可，而且开发上比较容易，只需要在手机上制作后台服务即可。
* **在云端搭建代理服务器：**在不修改浏览器的情况下，直接通过在云端搭建代理服务器来优化缓存是有难度的，因为所有缓存资源只被存放在浏览器中，调用与清理也由浏览器决定。此外，由于浏览器使用缓存的唯一依据是URL，因此该手段无法优化URL发生变化的相同资源。采用这种方法可以起到一定的浏览优化作用，如Google的Data Compression Proxy就是采用了这种方法。
* **在手机端搭建代理服务器，借助云端服务器取资源：**这种方法的优势在于，所有资源都将先通过云端服务器再选择推送给手机，资源判重的过程可以在云端进行，这样最大程度地利用缓存，节省流量。其潜在的问题在于，浏览器、手机代理、服务器与互联网构成一个四层架构，访问可能造成很大的延迟；其次用户需要一个云端服务器作为支持，而且用户需要在手机上启动服务，由此可能会带来额外的计算与电量消耗。

2.4.2 本文方案的系统结构

现有的服务器技术与浏览器技术已相对成熟，为了使用户更容易地使用本文的成果，本文所采用的方案中，用户应当无需改变现有的浏览器或服务器构件，通过简单的配置就可以直接使用本文的成果。其次，本文所采用的方案应能够尽可能解决现有缓存机制当中存在的问题，从而为用户提供最大的流量节省。

正因如此，本文采用了上一节提到的最后一个方法，即在手机端搭建代理服务器，并借助云端服务器向手机端选择性发送资源。根据此方法，本文设计了一个四层架构。该系统首先配置手机端浏览器的HTTP代理，并在手机端搭建HTTP代理服务，此后台服务接收浏览器的请求，并向浏览器返回响应。该后台服务并不直接向互联网请求资源，而向在个人云上搭建的服务器请求资源，由此服务器从互联网上下载资源，并有选择地向手机端进行推送。具体结构见图3。

  
图3：四层架构方案

通过这种架构，本文可以在不需要考虑浏览器的内部实现的情况下，完全控制浏览器资源的加载与对流量的消耗，对该系统的开发比较有利。此外，此系统并不影响浏览器自身的缓存系统。采用该方案也会给用户带来很好的体验，他们可以使用当前的主流浏览器，按照自己的习惯浏览网页，而对浏览器的要求仅限于可以配置代理服务器。此外，个人云服务器加载互联网资源的速度可能快于手机直接加载，因此采用此架构可能会加快资源的访问。

然而此架构的不足正如上文所提到的，四层架构可能带来高延迟，其次手机端可能带来电量消耗，而且，需要一个云端服务器作为支持。

2.4.3 本文的缓存优化方案

根据上面发现的缓存机制当中存在的冗余传输等问题，基于四层架构的系统的方案，本文提出如下的针对性优化方案：

* **以校验码为依据的冗余传输检查**：手机端会缓存所有服务器端传送的资源。在服务器传送资源时，会先通过资源MD5值的方式来确认该资源是否已在手机上缓存，如果待传送资源已在手机上缓存，则不传送资源内容。与传统的通过URL与过期时间的方式判断资源缓存相比，以资源内容作为缓存有效性的更为准确而且合理，这样做既可以利用内容相同但URL不同的资源，也可以避免因缓存有效期设置不当造成的错误使用缓存的问题。为了实现本优化，本文需要实现一套在线访问的系统，通过服务器对MD5值的判断完全避免传输到手机端已存在的资源。
* **数据传输均采取压缩方式**：虽然目前的主流浏览器均可以接收通过gzip、deflate等方式压缩的资源，但服务器并不会把所有资源均压缩后发送给用户。传送未经压缩的资源可能会带来流量上的浪费，因此在本系统中，服务器与手机端之间的资源体在传输之前，均已先通过静态gzip的算法进行压缩。
* **预取用户可能访问的网站，提高访问速度**：在系统预先知道用户可能使用的Web应用时，系统先将访问网站的所需资源预取下来，并在用户实际访问网站时将这些资源推送给用户。这样做可以为用户省去从远程Web服务器加载资源的时间，从而提高网站的加载速度。根据此方案，本文需要实现一个离线资源抓取的系统，在合适的时间将用户可能会访问到的网站提前预取下来，供用户使用。

根据上文给出系统结构与缓存优化方案，下一章将对本系统的进行设计。

## 第三章 理论推导

本章对传感器和视觉结合的定位系统的各个部分进行了详细说明和理论推导

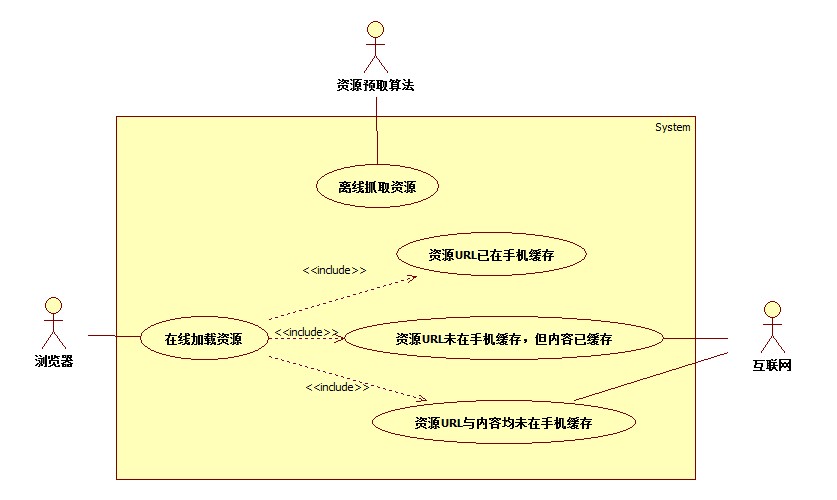
3.1 手机姿态测算

3.1.1 总述

总体而言，本文参照[]中的理论推导，使用四元数作为姿态的状态表示，以避免使用旋转矩阵表示造成的冗余或欧拉角表示产生的奇异点；使用扩展卡尔曼滤波器将角速度传感器数据和外源观测数据融合，消除角速度计的漂移。与[]中不同的是，本文中用以融合的数据为加速度计和地磁传感器数据，而不是

下图给出了系统的用况图，共有两个用况：

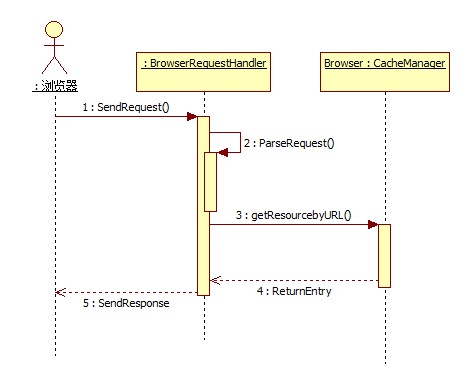
* **离线抓取资源：**由资源预取算法得到在合适的时间对用户可能访问的网站列表进行离线抓取资源。
* **在线加载资源：**浏览器以资源为单位向系统发送请求，并等待系统返回相应的响应。根据请求的不同，可能出现三种情况：URL已在手机缓存、URL未在手机缓存但内容已缓存以及URL和内容均未在手机缓存。

  
图4：面向对象分析用况图

3.1.2 顺序图

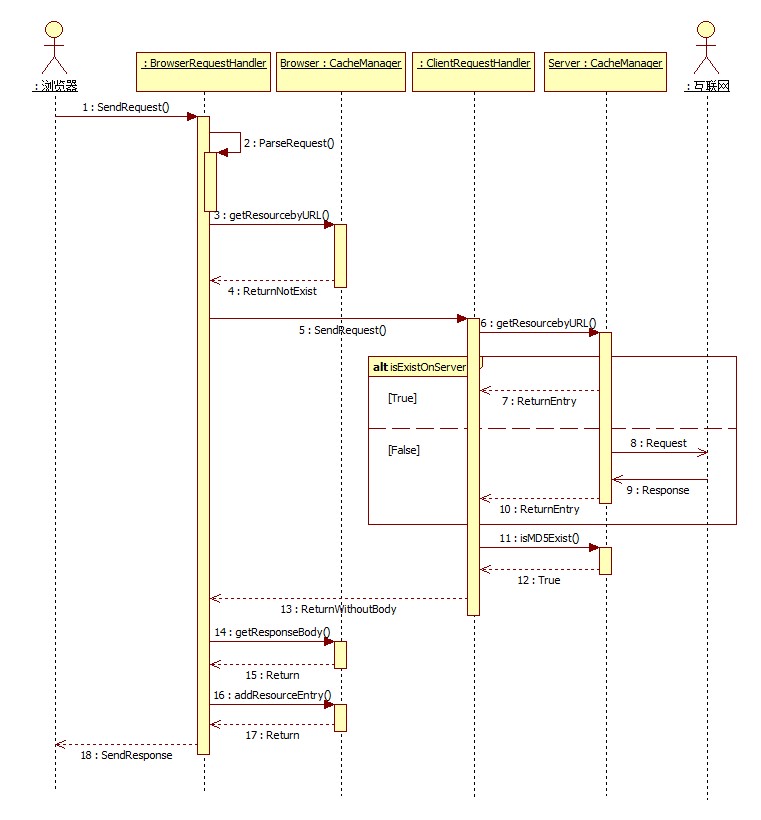
根据前面的分析，共有以下四个用况，下面逐个绘制顺序图：

1. 取URL已在手机缓存的资源

  
图5：OOA顺序图——取URL已在手机缓存的资源

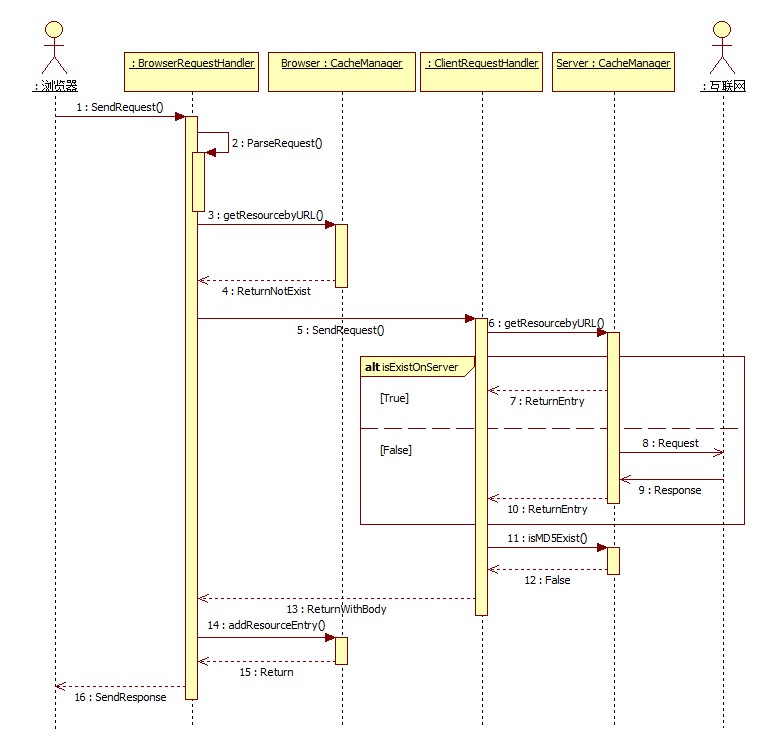
上图表示，当手机端已经有资源的缓存，可以直接从缓存列表中调用相应数据，并组成HTTP响应发回浏览器。

1. 取URL不在手机缓存但内容已缓存的资源

  
图6：OOA顺序图——取URL不在手机缓存但内容已缓存的资源

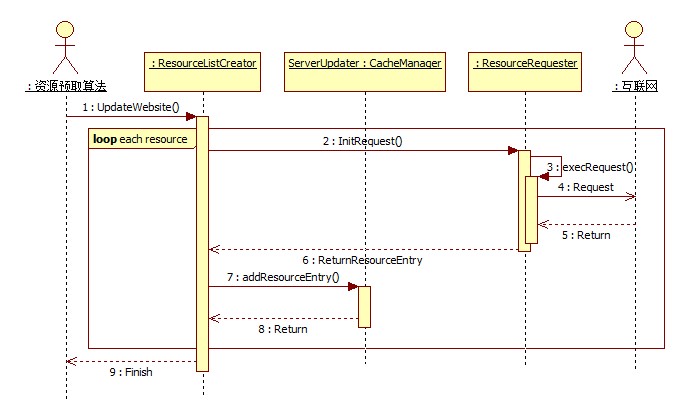
上图表示，手机端无法通过缓存信息独立判断出URL对应的资源内容。此时手机端将请求发给服务器，服务器在其缓存中查找该URL，如果找不到则向互联网请求该资源。此后服务器通过查看该资源的MD5的方式来查询到该资源的内容在手机端已缓存，则服务器端把资源体之外的信息（包括MD5、响应头）发回手机，手机更新缓存，并从缓存中提出资源体之后，组成HTTP响应发回服务器。

1. 取URL和内容均未缓存的资源

  
图7：OOA顺序图——取URL和内容均未缓存的资源

此用况与上一个用况的不同之处在于，服务器通过查看资源的MD5查询到该资源在手机端没有缓存，也就是说这是一个全新的资源。此时，服务器端返回完整的资源，手机端在接收到资源后，将其更新到缓存中，并组成HTTP响应发回浏览器。

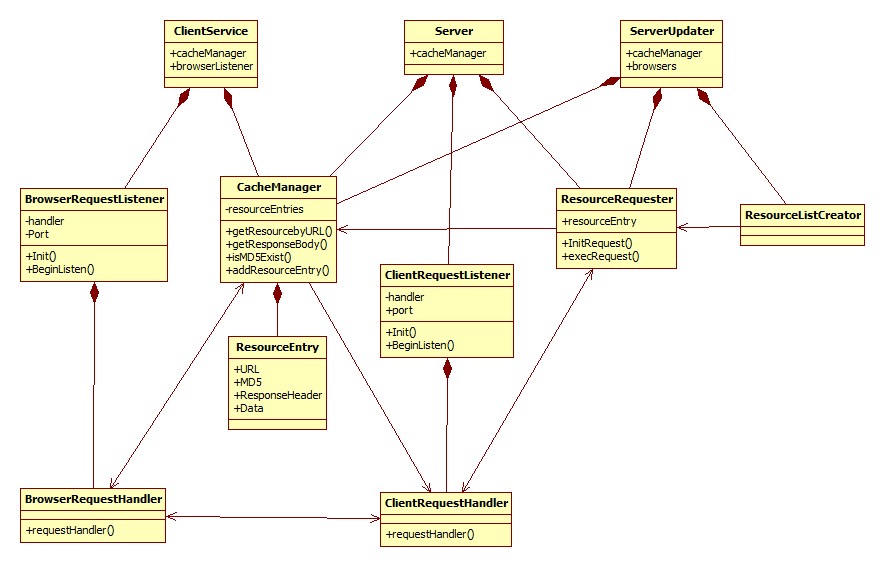
1. 服务器离线抓取资源

  
图8：OOA顺序图——服务器离线抓取资源

上图是服务器离线抓取资源的顺序图，由资源预取算法，服务器端资源预取系统预取相应URL的所有资源。根据网站的地址，服务器可以使用浏览器渲染网站并得到所有的资源列表。对列表中的没一个资源，服务器均从网站上请求该资源，并将其加入到服务器端缓存库中。

3.1.3 类图

根据系统的用况图，本文设计得到了下面的类图

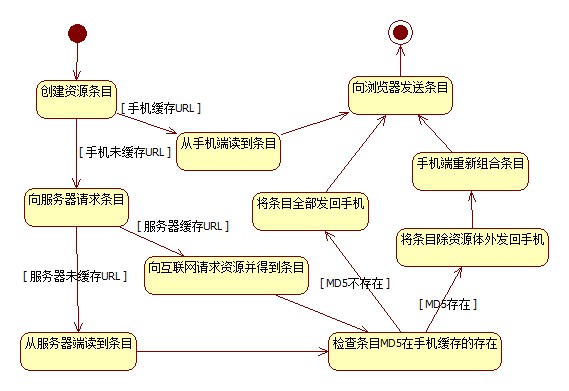
  
图9：面向对象分析类图

根据类图可以看出，整个系统由三个子系统组成，手机端后台服务（ClientService）、在线服务器（Server）、服务器端资源预取（ServerUpdater）。剩余类的描述如下：

* CacheManager：三个子系统均有此类的实例，用于系统中对缓存的存储、修改、读取等操作，缓存存放的位置涉及内存、磁盘、数据库
* ResourceEntry：资源条目，每一个取下来的资源会生成一个此对象
* BrowserRequestListener：用于手机端后台服务监听浏览器HTTP请求的类
* BrowserRequestHandler：BrowserRequestListener在监听到浏览器的请求后，会实例化出此类的对象，用来进行请求的处理
* ClientRequestListener：用于服务器端监听手机端的资源请求连接
* ClientRequestHandler：ClientRequestListener在监听到手机端请求后，会实例化出此类的对象，用来进行请求的处理
* ResourceRequest：当服务器需要请求资源时，会实例化出此类的对象来从互联网上请求资源。
* ResourceListCreator：服务器端资源预取系统中，通过此类来解析出一个网站的所有相关资源，并在此后进行所有资源的请求与记录

3.1.4 状态机图

当浏览器在线加载资源时，资源条目（ResourceEntry）这一对象在其生命周期内有频繁的状态变化，下面画出这个对象的状态机图。

  
图10：OOA状态机图——资源条目（ResourceEntry）

从本图可以看出，资源条目在状态转移中共涉及三个判断，首先在手机端通过URL在缓存库中检索，如果找到条目则直接发送，没有找到时向服务器请求。服务器再在服务器的缓存库中检索，找到时直接准备发回手机，没有找到则向互联网请求资源并得到条目。最后在服务器端的数据库中通过MD5检查资源体是否在手机缓存中存在，存在则不将资源体发回手机，手机从缓存中取出资源体并重组条目，若不存在，则将条目全部发回手机，最后手机端将条目发回浏览器。

3.2 面向对象的设计

3.2.1 系统环境

面向对象的设计是对分析的细化，同时需要结合具体实现的环境对分析的结论进行调整，本系统实现的环境如下：

* **通信协议：**为了提高系统效率，本文需要实现服务器推送（server push），TCP连接复用等功能，因此本文的手机端与服务器端通信采用SPDY协议。
* **手机端：**浏览器使用Android平台可以配置HTTP代理的Firefox浏览器。手机端需要在后台启动服务，为浏览器提供HTTP代理。因此手机端需要实现Android上的App，该应用可以作为后台服务运行。编程语言为Java。
* **服务器在线端：**服务器端需要处理手机端的连接请求，同时通信使用SPDY协议。因此本文将Jetty的SPDY模块内嵌到服务器端的程序中，并由此搭建服务器，编程语言为Java。
* **服务器离线端：**该程序需要能抓取网站的所有资源，此前的研究表明，为了获得网站资源列表，只能使用真实浏览器渲染。本文将JCEF实现的浏览器内嵌到该离线端，用来获得网站资源列表。程序的编程语言为Java。
* **数据库：**为了实现在服务器端实现离线抓取网站、对缓存长久保存、同步手机缓存资源等功能，需要在服务器端搭建数据库，本文使用MySQL数据库。

在第四章中，将对所涉及的具体工具的进行更为详尽的介绍。

3.2.2 系统架构图

  
图11：系统架构图

本文的系统由三个子系统组成：

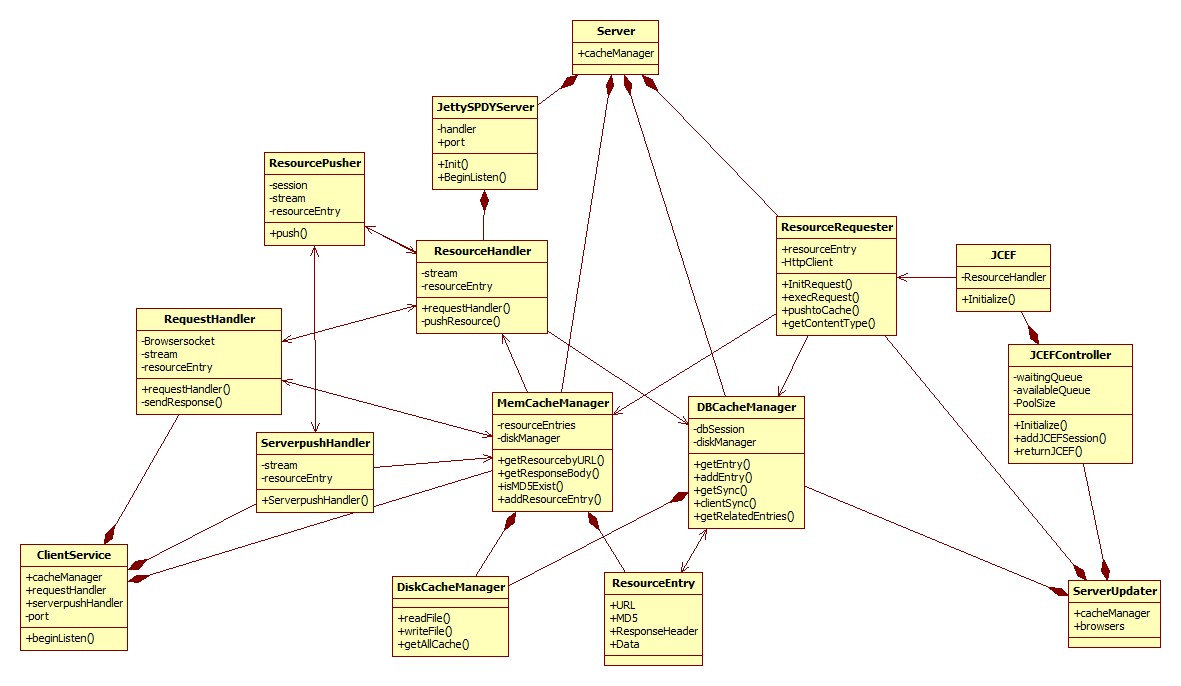
**手机端后台服务：**通过HTTP协议接收Firefox浏览器的请求并发送响应，通过SPDY协议与服务器在线系统进行通信，发送资源请求，并接收服务器推送的资源。系统内部由3个子模块组成，JettySPDYClient是与服务器通信的模块，同时处理服务器推送的数据；MemCacheManager为缓存控制模块，用来处理缓存条目的查询与添改，同时控制磁盘上的缓存存储；RequestHandler用于处理浏览器的请求，通过缓存管理模块检索缓存并通过通信模块向服务器发送资源请求与接受响应。

**服务器端在线程序：**通过SPDY协议与手机端通信，处理手机端发来的资源请求，推送手机端需要的资源。JettySPDYServer是与手机端的通信模块，同时负责开启端口让手机端接入；ResourceHandler用来实际处理手机端的请求，访问数据库检索资源，向手机端返回资源，该模块内含两个子模块，ResourcePusher用于向服务器推送资源体，ResourceRequester基于HTTPClient完成Web资源获取的工作。

**服务器端离线程序：**读取用户可能访问的资源列表，并从网上抓取访问网站相关的所有资源。共由3个子模块组成，ResourceRequester获取Web资源，CEFPool控制浏览器加载网站并驱动ResourceRequester请求资源，Prefetcher负责数据库的读取与离线程序的控制。

3.2.3 类图

根据系统环境与架构的设计，本文在面向对象分析类图的基础之上，对类图进行了修改与细化。

  
图12：面向对象设计类图

相比面向对象分析中的类图，主要有以下三个改动：

* 细化缓存管理类，将原CacheManager拆分为3个类，分别负责内存、磁盘、数据库中的缓存。手机端程序不涉及数据库，因此没有数据库管理类；离线端程序不需要在内存记录缓存，因此没有内存缓存管理。
* 在离线资源抓取系统中，针对JCEF设计控制类，重写JCEF的资源获取方法，从而可以用ResourceRequester取资源并记录数据
* 细化手机端与服务器端的通信事件处理类，手机端细化为手机资源请求类和服务器推送处理类，服务器端细化为服务器推送类和手机资源请求处理类。

3.2.4 数据库设计

本系统共建立了两个数据库表：resource与userpreference。

resource表用于存储服务器的缓存条目与手机端的同步情况，该表共含5个资源，细节如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段 | 类型 | 设置 | 说明 |
| URL | varchar | 主键 | 资源的URL |
| MD5 | varchar | 非空 | 该资源体的MD5，长为32的字符串，资源的实际内容存储在磁盘中，名称为MD5的值 |
| Header | varchar | 非空 | 该资源最新的响应头 |
| Father\_URL | varchar |  | 表示该资源在哪一个URL的资源列表中 |
| Sync | integer | 非空 | 表示该资源体是否在手机上有储存，0表示未储存，1表示已储存 |

userpreference表用来存储用户可能访问的网站，即服务器离线取资源的预取列表，只有一个字段用来保存用户访问网站的URL。

## 第四章 移动浏览器缓存优化的实现

本章的主要内容是缓存优化系统的实现部分，首先介绍实现过程中涉及的相关工具与协议，此后介绍系统的各个子系统与各个模块的具体实现与伪代码。

4.1 本文使用的相关工具

4.1.1 SPDY协议

SPDY是由Google创造的Web内容传输协议，SPDY协议是基于TCP协议的，在应用层上对原HTTP协议进行的优化，目标是降低网络延迟，提高安全性。下一代通信协议HTTP/2正是在SPDY的基础之上修改的。在本文的应用中，由于HTTP/2协议相对较新，目前支持HTTP/2协议的开源项目非常少，而且还远不成熟，而SPDY已经具有3年的历史，相关Web应用与开发资料比较丰富。因此本文使用SPDY作为手机端与服务器端的通信协议。

SPDY在安全套接层（SSL）之上创建了会话层（Session Layer），本文主要利用SPDY会话层的功能来实现客户端对资源的请求与服务器端对资源的推送。这个会话层是允许连接的多路并发的，可以在单个TCP连接上通过流（Stream）的方式进行同时双向同时传输，而且客户端与服务器端均可以在会话层上创建多个流。当会话创建后，可以用每一个流来模拟HTTP短连接，从而解决TCP连接建立慢的问题。此外，在同一会话中，不同的流可以设置不同的优先级，这样可以将不同的资源以不同的优先级进行传输。

在流上，数据以帧（Frame）的方式进行传输，服务器与客户端均可以发送帧，帧可以用来传送控制信息并可以附带数据。帧分为控制帧（Control Frame）和数据帧（Data Frame）两种，常用的控制帧有以下两种：

* SYN\_STREAM帧：用来主动创建流，后文简称SYN帧。本文的系统中，手机端发送此帧，表示手机向服务器请求资源；服务器端发送此帧，表示进行一个资源的Server push。
* SYN\_REPLY 帧：在接收到SYN\_STREAM帧后，发回此帧表示连接成功建立，后文简称REPLY帧。本文的系统中，手机端发送此帧，表示手机端已准备好接收server push；服务器端发送此帧，会附带资源信息，表明这个资源手机是否已有缓存。

数据帧没有控制流的作用，其功能是在流建立完成后，在流上互相发送数据，后文称DATA帧。本文的系统中，只有服务器端可以发送此帧，用来传输Server push的资源内容。

以上SPDY协议在本系统中的具体应用方式详见后文伪代码。

4.1.2 Jetty

Jetty项目为开发者提供了一个易用的Web服务器与Servlet容器，同时还有大量的网络功能，如SPDY、WebSocket等。Jetty的所有构件都是开源的并可以用作商业用途。Jetty是由Java语言编写的，其API以多个Jar包的形式进行发布。Jetty的特点是可以把Web服务器实例化成一个对象，从而可以很容易地嵌入Java应用，来提供Web服务器的功能。

本文在服务器端使用Jetty搭建了基于SPDY协议的服务器，在手机端使用Jetty的SPDYClient库与服务器端进行SPDY协议的通信。使用Jetty开发对JDK的最低版本要求是1.7，嵌入Android时，对Android系统的最低版本要求是4.4 (API Level 19)。这也是本文系统运行的最低Java版本与Android版本。

4.1.3 Chromium Embedded Framework (CEF)

实验证明，静态分析HTML的方式不能分析出加载一个网站所需要的全部资源，如果想要得到加载一个网站对应的资源列表，最好的方式是使用浏览器引擎将其加载一遍。CEF是基于Google Chromium项目的开源浏览器项目，但CEF与Chromium并不相同，Chromium关注的是开发Google Chrome应用，着眼于优化渲染网页的方式；CEF关注浏览器在第三方应用上的嵌入式应用，使开发者可以在不了解浏览器内核的情况下，通过CEF提供的API来开发浏览器应用。大量的软件使用CEF为应用提供内嵌浏览器，如Evernote，QQ等

CEF是C++语言的项目，基于CEF项目，开发小组启动了JCEF项目，即CEF的Java语言项目。本文系统实现的语言为Java，因此本文使用JCEF作为内嵌的浏览器。

本文使用JCEF来在应用中嵌入浏览器并通过接口创建它们。在离线采集系统中，当系统可以预先判断出用户喜好的资源列表时，就可以在服务器上通过让JCEF浏览器通过渲染网页的方式得到加载到用户喜好网站的资源列表，从而完成预取的功能。

4.2 手机端代理服务

4.2.1 RequestHandler

RequestHandler类主要负责在手机端处理浏览器发出的资源加载请求，包括RequestHandler1和RequestHandler2两个方法，分别为接收浏览器请求的处理函数和接收服务器响应的处理函数，下图给出其伪代码。

|  |
| --- |
| // 浏览器通过代理服务器加载资源的事件处理函数  // 客户端新建一个线程执行该函数  function RequestHandler1(BrowserSocket)  // 读取浏览器请求  browserRequest <- getRequestFromSocket(BrowserSocket)  // 对于不支持的协议或方法返回404  if browserRequest not supported  BrowserSocket.return404()  return  // 从请求中提取目标URL与请求头  URL <- parseURL(browserRequest)  // 从CacheManager中尝试检索资源  resourceEntry <- MemCacheManager.getEntry(URL)    // 资源条目如果在缓存中存在  if resourceEntry exists  // 如果资源正在传输过程中，线程挂起  if resourceEntry is transmitting  // 等待在该资源上  resourceEntry.wait()  // 资源准备好后通过按照HTTP响应结构发给浏览器  BrowserSocket.sendResponse(resourceEntry)  else  // 找到当前可用的SPDY会话，如果过期则重新创建  session <- getSPDYSession()  // 创建发送给服务器的请求头  header <- newSPDYHeader()  header.add("URL", URL)  header.add("header", browserRequest)  // 通过session发送SYN帧  // 创建stream并将访问相关信息发服务器，保持流畅通  // 注册RequestHandler2函数作为服务器返回信息的事件处理函数  session.syn(header, STREAM\_NOT\_CLOSE,  RequestHandler2(URL, BrowserSocket))  // 服务器发回响应时的处理函数，服务器返回资源的相关信息  function RequestHandler2(URL, BrowserSocket, ReplyInfo)  // 从服务器的响应中提取响应类型、响应头、MD5  replyType <- getReplyType(ReplyInfo)  responseHeader <- getResponseHeader(ReplyInfo)  MD5 <- getResponseMD5(ReplyInfo)    // 判断响应的内容是否已存在  if replyType is "MD5Exist"  // 向浏览器发送响应  BrowserSocket.sendResponse(responseHeader,  MemCacheManager.getData(MD5))  else  // 等待服务器推送该URL的资源  MemCacheManager.WaitOnURL(URL)  // 等待后线程被重启，向浏览器发送资源  resourceEntry <- MemCacheManager.getEntry(URL)  BrowserSocket.sendResponse(resourceEntry) |

图13：RequestHandler的伪代码

4.2.2 ServerpushHandler

ServerpushHandler类主要负责处理服务器端主动通过SPDY流发起的资源推送，仅包括ServerpushHandler方法，为接收到服务器端主动发起SPDY流后的处理函数，下图给出其伪代码。

|  |
| --- |
| // 用于监听服务器通过SYN帧发起的stream的处理函数  function ServerpushHandler(stream, synInfo)  // 从传来的头文件中得到推送资源的：  // URL, MD5, 在本地是否存在，响应头，数据大小，压缩后大小  URL <- synInfo.getURL()  MD5 <- synInfo.getMD5()  MD5Exist <- synInfo.getMD5Exist()  responseHeader <- synInfo.getResponseHeader()  dataSize <- synInfo.getDataSize()  zipSize <- synInfo.getZipSize()  // 如果该资源已存在，则保留响应头，无需传递资源体  if MD5Exist  MemCacheManager.addNewEntryFromExist(URL,  MD5, responseHeader)  return  // 向服务器使用REPLY帧发回响应，服务器接下来传数据  stream.reply(STREAM\_NOT\_CLOSE)  // 即使资源尚未传输完成，也先在缓存中增加该URL项  // 浏览器请求该资源时，会等待该资源准备完成  resourceEntry <- MemCacheManager.addNewEntry(URL,  MD5, responseHeader, DataSize)  // 监听DATA帧，服务器会通过DATA帧传回资源  stream.addDataListener(DataListener(resourceEntry))  // 用于监听服务器通过DATA帧发送资源的处理函数  // DATA帧的内容已在服务器经过GZip静态压缩  function DataListener(resourceEntry, stream)  //读取服务器通过DATA帧推送的所有资源，并放在zipArray中  while stream is not close  zipArray.append(stream.data)  //对数据解压缩，并写回到MemCacheManager中  resourceEntry.data <- GZipUtils.decompress(zipArray)  //向MemCacheManager发送消息，表明资源准备好  //回执行数据往磁盘的写入并继续所有挂起在该URL上的线程  resourceEntry.dataReady() |

图14：ServerpushHandler的伪代码

4.2.3 MemCacheManager

MemCacheManager主要负责管理内存中的缓存，除此之外，同时也通过DiskCacheManager来管理磁盘中对资源的缓存，该类实现对内存中缓存条目获取与添加修改。需要实现的方法包括：添加资源条目、根据URL获得资源条目、根据MD5查询资源的存在情况、根据MD5获得资源体。具体的实现伪代码在此不再赘述。

4.3 在线访问服务器

4.3.1 JettySPDYServer

该对象的作用是打开端口监听SPDY的会话层连接，默认端口为8181。手机端通过SPDY的会话层连接连上此端口后，当手机端发送SYN帧建立流时，触发事件并调用ResourceHandler处理函数。

4.3.2 ResourceHandler

ResourceHandler类的主要负责在手机发送SYN帧建立流时处理手机的资源请求，其中实现了ResourceHandler方法为接收SYN帧时的处理函数，下图给出其伪代码。

|  |
| --- |
| // 当手机端执行RequestHandler1时，发来SYN帧时触发事件执行该处理函数  // 此函数返回响应时出发手机端的RequestHandler2函数  function ResourceHandler(stream, synInfo)  // 初始化传回手机端的响应  replyHeader <- newSPDYHeader()  // 读取手机端传来的资源请求，包括URL与请求头  URL <- synInfo.getURL()  requestHeader <- synInfo.getResquestHeader()    // 从内存缓存中根据URL读取资源  // 注：服务器端的MemCacheManager的内容与手机是同步的  resourceEntry <- MemCacheManager.getEntry(URL)  // 如果存在则表明手机端请求的资源在传送中  if resourceEntry exists  // 此时应让手机端等待  replyHeader.add("ReplyType", "Waiting")  // 发回回复，并将流关闭  stream.reply(replyHeader, STREAM\_CLOSE)  else  // 从数据库尝试读取URL，查看服务器端已缓存本资源  resourceEntry <- DBCacheManager.getEntry(URL)  // 数据库不存在此URL，从网络在线取资源  if resourceEntry is null  resourceRequest <- newRequest(URL, requestHeader)  resourceEntry <- resourceRequest.execute()  // 查询手机端是否已缓存该资源体  isSync <- DBCacheManager.getSync(resourceEntry.MD5)  // 同步时，让手机端调用本地资源体，否则需要发送资源  if isSync  replyHeader.add("ReplyType", "MD5Exist")  else  replyHeader.add("ReplyType", "New")  ResourcePusher.push(stream, resourceEntry)  // 附上相关响应头与MD5，向手机端返回  replyHeader.add("ResponseHeader",  resourceEntry.header)  replyHeader.add("MD5", resourceEntry.MD5)  stream.reply(replyHeader, STREAM\_CLOSE)    // URL可能是数据库已缓存的网站HTML（即Father\_URL）  // 此时打包返回所有该网站相关的未同步的资源  relatedEntries <- DBCacheManager.  getRelatedEntries(URL)  for each relatedEntry in relatedEntires  ResourcePusher.push(stream, relatedEntry) |

图15：ResourceHandler的伪代码

4.3.3 ResourceRequester

ResourceRequester类的实例负责处理每一个服务器对Web资源的请求。该类基于HTTPClient 3.1对网络资源进行请求，该类可以读取浏览器的请求头并将请求头整合到HTTPClient的请求里面。在HTTPClient默认配置之上，本系统做了以下修改：

* 关闭资源重定向，原样返回状态为302的响应
* 为了提高浏览器的加载速度，将资源连接超时设定为2秒，并关闭资源请求重试的功能

4.3.4 ResourcePusher

ResourcePusher类负责服务器端主动推送资源，由ResourceRequester视需求创建并调用。实现的方法有两个，push用于向手机发送SYN帧并创建流，datapushHandler是接收到手机端接收数据的确认后的调用的处理函数，下图给出其伪代码。

|  |
| --- |
| // 此函数的功能是服务器端的对资源的主动推送  // 注意：此处的流是手机端发起的资源请求流，  // 主动推送需要在会话层上需要重新创建流  function push(stream, resourceEntry)  // 得到SPDY的会话层，可以在会话层上主动发起流  session <- stream.getsession()  // 压缩资源文件  zipData <- GZipUtils.compress(resourceEntry.data)    // 检查资源体是否和手机端同步  isSync <- DBCacheManager.getSync(resourceEntry.MD5)  // 创建推送的头信息，包含信息：  // URL, MD5, 在本地是否存在，响应头，数据大小，压缩后大小  pushHeader <- newSPDYHeader()  pushHeader.add("URL", resourceEntry.URL)  pushHeader.add("MD5", resourceEntry.MD5)  pushHeader.add("ResponseHeader",  resourceEntry.responseHeader)  pushHeader.add("DataSize", resourceEntry.data.getSize())  pushHeader.add("zipSize", zipData.getSize())    // 如果资源体已经在手机上，则只发送推送头信息即可  if isSync  // 向手机端发送头信息后，关闭此流  session.syn(pushHeader, STREAM\_CLOSE)  else  // 向手机端发送头信息，等待手机端REPLY响应，  // 此后调用处理函数向手机端推送数据  session.syn(pushHeader, datapushHandler(zipData, MD5), STREAM\_NOT\_CLOSE)  // 手机端接受到头信息后，会发回REPLY帧，此时的处理函数  function datapushHandler(zipData, MD5, stream)  // 更新数据库，此资源体已同步  DBCacheManager.ClientMD5Sync(MD5)  stream.data(zipData, STREAM\_CLOSE) |

图16：ResourcePusher的伪代码

4.4 离线网站采集程序

4.4.1 JCEF的配置与使用

对于JCEF，本文进行了针对性的配置，首先修改了浏览器的请求头，将User-Agent修改为手机Firefox浏览器的User-Agent，并根据目标手机的浏览器分辨率进行了分辨率的配置，用来模拟手机的访问。其次重写了浏览器资源请求的方法，将所需请求资源的信息发送到ResourceRequester类，并由该类对Web资源进行请求与记录。

4.4.2 JCEFController

JCEFController用于控制JCEF池，系统在启动时会启动多个JCEF实例，同时按照需求给JCEF分配加载任务，当任务过多时排队等待。在有JCEF加载完成后接收新的任务。需要实现的方法有三个，Initialize方法用于初始化多个JCEF实例，addJCEFSession方法用于给JCEF池分配新的任务，returnJCEF方法被JCEF实例在网站加载完成后调用，释放JCEF接受新的任务，下图给出其伪代码。

|  |
| --- |
| // 初始化JCEF池  function Initialize(PoolSize)  // 初始化等待队列与可用队列  WaitingQueue <- newQueue()  AvailableQueue <- newQueue()  // 初始化JCEF浏览器，并加入到可用队列中  for i=1 to PoolSize  AvailableQueue.push(newJCEF())  // 在用户通过JCEF抓取资源时使用此函数加载URL  function addJCEFSession(URL)  // 从池中找到可用的JCEF，否则等待  JCEF <- JCEFAvailableQueue.pop()  if JCEF exists  // 如有JCEF可用，直接加载URL  JCEF.loadnewURL(URL)  else  // 将URL插入等待队列  WaitingQueue.push(URL)  // 这是浏览器加载完成事件的处理函数  // 当JCEF加载资源完毕时，通过此函数返还原先占用的JCEF  function returnJCEF(JCEF)  // 查询有没有等待中的访问  if WaitingQueue is not empty  URL <- WaitingQueue.pop()  JCEF.loadnewURL(URL)  else  AvailableQueue.push(JCEF) |

图17：JCEFController的伪代码

4.4.3 ServerUpdater

ServerUpdater类是离线网站抓取系统中的主动类，其中实现了Prefetch函数用于抓取用户所有可能访问的网站中的所有相关资源。

|  |
| --- |
| // 默认JCEF池大小为10  JCEFPoolSize <- 10  // 从网上抓取资源  function Prefetch()  // 从数据库读取用户的网站列表  websites <- DBManager.getUserPreferences()  // 初始化JCEF池  JCEFController.Initalize(JCEFPoolSize)  // 通过JCEF加载所有资源  for each website in websites  JCEFController.addCEFSession(website) |

图18：Prefetcher的伪代码

## 第五章 移动浏览器缓存优化的实验验证与性能分析

在缓存优化系统实现完成后，本文接下来需要验证通过本文缓存优化系统相比原浏览器直接加载网页存在性能提升。为此，本文设计并进行了如下实验。

5.1 实验设计

5.1.1 实验方案

在本文的实验中，采取以下方案：使用两个相同的手机，连接到同一个无线路由器。一个手机上的浏览器通过配置代理的方式使用本文的系统，另一个手机的浏览器使用默认配置。服务器端首先通过离线网站采集系统采集将要测试的网站，在浏览器缓存清空的情况下，同时使用着两个手机加载测试网站。此后，在手机端有缓存的情况下，立刻再次加载网站。测量两次网站加载的时间与加载网站的手机端传入流量，进行分析与对比。

后文提到的第一次访问的冷启动实验简称为Cold实验，第一次加载后立刻进行得第二次访问的热启动实验简称为Warm实验。

5.1.2 网站的选取

本文在Alexa排名前500网站中选取网站，选取原则如下：

* 网站在实现环境下可以访问，列表中域名对应的内容不重复
* 网站在原浏览器和采用本文系统后的浏览器中均可以顺利访问，标志为浏览器的进度条读满，显示正常
* 访问网站均为专为移动设备设计的移动版网站

遵循以上原则，本文选择出50个网站。为了避免某一次访问的结果有较大的偏差，本文对每一个列表中的网站进行10次试验，采集优化前后的各10个数据。因此，本文共得到使用优化前后共2000条加载网站的加载时间与流量数据。

5.2 自动化测试系统的设计

5.2.1 系统流程图

以下为单个手机进行测试的流程图。

  
图19：自动化测试系统流程图

5.2.2 实验数据收集方案

实验中共需要收集两个数据，加载时间与加载流量，以下是数据收集的方案。

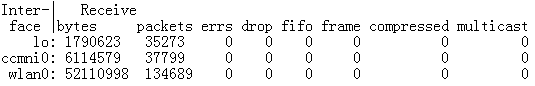
关于网站加载时间，本文定义，网站加载的时间为从网站开始加载到网页触发onload事件的时间间隔。本文的系统中，开始时间为通过Firefox插件控制浏览器加载某一网站的时间，结束时间为该网站触发onload事件，并执行Firefox插件重写的onload处理函数的时间，这两个时间点的差即网页加载所需时间。

关于网站加载流量，本文定义，网站加载的流量为从网站开始加载到触发onload事件5秒之后的网卡传入流量，定一个5秒的空闲时间主要是因为onload的时刻并不一定代表网站所有资源的加载完成，onload事件触发后，有一些需要通过JavaScript加载的资源仍在加载，如果此时就停止加载或者记录流量数据是不符合实际使用情况的，因此本文增加了此延时。

为了获取网卡的传入流量，本文将手机通过USB连接到PC，通过Android Debug Bridge (ADB)的shell命令读取记录Android设备当前网络传输流量的文件，在PC端执行如下命令：

> adb shell cat proc/net/dev

下图为执行该命令后得到的结果示例（对传入部分和没有使用的网络接口进行了省略）：

  
图20：adb shell cat proc/net/dev的结果示例

图中可以看到ADB返回的在执行命令时刻各网络接口的传入数据，其中，lo表示回环接口，ccmni0表示手机移动网络的接口（在未来的3G网络实验中将用到），wlan0表示Wi-Fi网络的接口，是本实验所关心的。在实际应用中，本文在网页加载前后两次执行此命令，对两次wlan0采集到的传入字节数做差，即为加载网页所需流量。

5.2.3 系统实现方案

如果想控制网页的加载并获得onload事件的触发时间，就需要对浏览器进行一定修改，来控制浏览器的行为，本文是通过编写Firefox插件的方式来实现的。

由于需要调用PC端的命令来获取流量信息，此处无法仅通过Firefox插件实现，因此需要在PC端搭建程序，并让Firefox插件与PC端程序协同工作。针对浏览器本身易于使用HTTP方式发送并接受信息的特点，本文设计了以下架构：

在PC端搭建Web服务器，浏览器加载此服务器上的Web网页（即通过HTTP的GET方法加载），以参数的方式向服务器报告网页加载时长与时刻，同时可以从服务器请求下一个加载的网站与是否清除缓存的指令。为了实现多设备同时测试，本文在每个设备发送的参数中添加一个设备编号，即可以在服务器端区分每个设备的测试。

手机端可以传入的参数有以下三种：

* devid: 表示设备编号
* type: 值为begin表示测试开始，为end表示测试结束
* time: 当type=end时，用来表示这一次加载所需的毫秒数

5.3 自动化测试系统的实现

5.3.1 自动化测试用Firefox插件

Firefox插件可以为Firefox浏览器添加新功能，例如管理标签页、更改Web内容、提高安全性等。Firefox插件的开发语言是JavaScript，其实现的效果也相当于是在浏览器里执行JavaScript代码。本文需要开发的是Android平台上Firefox的插件，其开发细节与开发普通Firefox浏览器的插件略有差异。

根据设计，在用户点击开始测试后，浏览器监听每一个页面的onload事件，浏览器总共会加载三种网页：测试网页、开始网页、结束网页。其中开始网页和结束网页是用来和服务器通信，发送测试结果并取得测试控制信息的。以下给出了Firefox插件的伪代码。

|  |
| --- |
| // 获取测试服务器提供的ServletURL  // 如：http://162.105.51.254/TestServlet  ServletURL <- getServletURL()  // 记录加载页面开始的时间戳  StartTime <- null  // 记录当前设备编号  DeviceID <- getDeviceID()  //点击浏览器的“开始测试”触发的函数  function ClickStart()  {  // 监听浏览器的onload事件，事件处理函数为onPageLoad()  Browser.addEventListener("load", onPageLoad)  // 向测试服务器发送开始信息，请求下一个网站  loadBegin()  }  //onload事件处理函数  function onPageLoad()  {  // 获取onload完成的网站URL  URL <- Browser.currentURL  // 判断加载完成的是目标网站还是服务器指令  if URL.contains(ServletAddr)  // 判断载入的是加载开始还是加载结束  if URL.contains("begin")  // 读取网页内容，即下一个要载入的URL  nextURL <- Browser.getDocument.getBody()  // 记录开始加载的时间  StartTime <- getTime()  // 加载下一个网站  Browser.loadURL(nextURL)  else if URL.contains("end")  // 读取是否需要清理缓存，如果需要则清理  CacheControl <- Browser.getDocument.getBody()  if CacheControl = "clear"  Browser.cleanCache()  // 开始下一次测试  loadBegin()  else  // 计算出加载所需时间  loadTime = getTime() - StartTime  // 等待5秒钟  setTimeout(5000)  // 测试结束，传输加载时间  loadEnd(loadTime)  }  // 测试开始  function loadBegin()  {  Browser.loadURL(ServletURL +  "?type=begin&devid=" + DeviceID);  }  // 测试结束  function loadEnd(loadTime)  {  Browser.loadURL(ServletURL + "?type=end&time=" +  loadTime + "&devid=" + DeviceID);  } |

图21：Firefox插件的伪代码

5.3.2 自动化测试控制服务器

本文采用Servlet搭建测试控制Web服务器，Servlet在Tomcat服务器上运行。服务器的主要工作是在测试开始和结束时调用ADB采集流量信息，在测试开始时提供加载的网站，在结束时收集访问时间，计算访问流量并将测试结果写入到文件中。以下给出Servlet的伪代码。

|  |
| --- |
| // 通过GET方法访问Servlet时的调用函数  function doGet(request, reponse)  // 获取设备编号  devID <- request.getParameter("devid")  // 以字符串读取到相应设备的adb传入的全部流量数据  wlanData <- getNetStatFromADB(DevID)  // 对数据进行文本分析，提取出Wi-Fi的传入流量  rx <- Extract\_wlanRx(wlanData)  // 判断此次GET是开始测试或者结束测试  if request.getParameter("type") = "begin"  // 得到该设备下一个访问的URL  nextURL <- getNextURL(devID)  // 向手机端以HTML内容的方式发送URL  response.sendHTMLbody(nextURL)  // 记录设备访问前的传入数据量  RecordRx(devID, rx)  else if request.getParameter("type") = "end"  // 从参数获得加载时间  loadTime <- request.getParameter("time")  // 将此时的数值与开始时的数值做差得到数据量  loadRx <- getLoadRx(devID, rx)  // 向设备发送是否清理缓存的指令  response.sendHTMLbody(isCleanCache(DevID))  // 记录本次访问的数据  output(devID, loadTime, loadRx) |

图22：自动化测试Servlet的伪代码

5.4 实验结果

本节首先给出实验数据的处理方法，定义了优化率作为评价优化程度的指标。之后从各个角度描述实验结果，最后讨论实验见过，对实验中优化程度很低的测试进行更深一步的具体分析

5.4.1 性能评价方法

本文首先定义优化率，用来优化后相比于优化前的性能提升水平。下式表示的是时间优化率的计算，将加载时间变为加载流量即可得到流量优化率。

在实验中，本文测量加载时间与加载流量，这两个数值越低，说明访问效果越好。根据公式，优化后的测试结果越低，优化率越高。因此采用该优化率评价系统的优化能力是合理的。

为了避免外在因素造成偏差较大的结果，本文对于每一个网站的第一二次访问实验均重复了10次，对于这10次测量结果，本文首先计算出在每一次测量中时间与流量上的优化率，再对这10个优化率取中位数得到最终该网站在某一次访问实验中的时间或流量优化率。对于50个网站的实验，可以得到200个优化率结果。

5.4.2 实验结果

通过上一节的方法，本文得到针对每个网站的四个实验的优化率：

* Cold, Time：第一次访问的加载时间优化率
* Cold, Data：第一次访问的传入流量优化率
* Warm, Time：第二次访问的加载时间优化率
* Warm, Data：第二次访问的传入流量优化率

针对每个实验所测的50个网站的优化率，本文绘制了优化率的累积分布函数图，即CDF图，如图23所示。

图23：优化率的累积分布函数（CDF）图

除此之外，本文对数据进行了初步统计，每个实验的50个结果的平均值与中位数见下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 优化率 | 平均值 | | 中位数 | |
| Time | Data | Time | Data |
| Cold | 28.9% | 17.1% | 29.6% | 11.2% |
| Warm | 9.0% | 66.4% | 10.9% | 74.5% |

图24中，直接对实验中所测得网站的消耗流量与加载时间绘制了CDF图，可以更直观的看出，采用了本文的缓存优化系统后，访问网站的加载时间有明显的减少，传输的流量也有明显的减少。

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

图24：加载时间与加载流量的CDF图

5.4.3 实验结果讨论

虽然整体结果表明，本系统整体上对访问网站有明显优化，但在某一些网站加载的测试结果中，出现了负优化率，有一些数值较大，以下给出原因的分析。

第一种情况是本身加载时间非常短。如m.360.cn网站，在第二次访问中，原网站加载的时间为0.18秒，而优化后加载时间为0.34秒。用户无法在访问中感受到这两个时间的任何差异，然而计算出的优化率却为-90%左右，对实验结果有较大影响。

第二种情况是由于HttpClient加载失败引起的。在cn.bing.com网站的第二次访问实验中，原加载方法的时间为0.4秒，优化后加载时间为2.1秒左右，此时，计算得到流量优化率为-400%左右（使第二次访问平均速度优化率降低了8%）。由此对第二次加载的时间平均值造成非常大的影响。通过反复回放发现，原因在于，有一些资源无法通过HttpClient正常获取，而通过浏览器本身的资源加载方式则可以获取。这种情况发生时，HttpClient会在2秒的超时时间经过后返回资源获取失败，而此时计算得到该网站的加载时间就在2秒以上了。但Firefox浏览器的原生请求中，该资源可以在短时间内正常获取，这就造成了实验中的情况。而为什么HttpClient无法正常获取的问题根源目前还没有发现。

5.5 本系统的额外开销

在带来浏览性能优化的同时，本系统也会给手机和服务器带来一定的额外开销，本节对系统造成的额外开销进行分析

5.5.1 额外流量开销

相比与传统方式，通过本系统进行Web浏览，可能会带来额外的手机流量开销。可能带来额外流量开销的共有以下几点：

* **访问预取网站传入的额外资源**：在手机端浏览已预取的网站时，服务器将预取到的资源列表中的所有资源发给手机。由于现在的网站通常使用了大量的AJAX技术，即使是相同的HTML与JavaScript文件，在执行过程中也可能产生不同的资源请求，从而造成资源列表不同。因此，可能会给手机传送部分在访问中不需要的资源。
* **与服务器通信时额外传送的信息：**手机端与服务器的通信包含一些传统HTTP请求之外的信息，如MD5，SPDY流的控制信息等，由此可能造成额外的流量开销
* **不同URL的相同资源造成的额外请求：**不同URL的相同资源理论上应该不需要额外花费流量请求资源，在本系统中手机端需要向服务器端发送请求，虽然服务器端不需要发送资源体，但需要服务器确认该URL所对应的资源后，手机才能利用该资源。这个过程可能造成额外流量消耗。

5.5.2 额外性能开销

本文关注手机端App带来的额外性能开销。通过测试，本文得到了App在手机端的内存与CPU开销：

* **内存：**刚安装好的应用启动后台服务时消耗内存11244KB，浏览50个网站后，内存消耗为58104KB。此时重新启动服务时，程序会将缓存资源读入内存，此时的内存消耗为34432KB。
* **CPU：**本文对比了渲染移动版网页时，Firefox浏览器与本文的后台服务之间的CPU占用之比，该比例的平均值约为40:1，CPU实际占用率约为3%。由此可见，通过本文的后台服务加载网站并不会对CPU造成太大负担。

## 第六章 工作总结与展望

6.1 本文工作总结

本文的主要工作包括：

* 介绍了当前浏览器的缓存机制与潜在的缺陷，分析了缓存的理论性能上限与当前性能间的差距及造成的原因，验证了实现浏览器的缓存优化是有意义而且可行的，并提出了优化方案。
* 针对提出的优化方案，本文设计并实现了基于代理的移动浏览器缓存优化系统，该系统有三个子系统组成：手机端后台服务、服务器端在线服务、服务器端离线资源抓取程序。该系统通过移动设备端的后台服务和云端服务器对浏览器请求资源的记录、选择性推送与优化传输，实现了对现有缓存机制的优化。
* 设计实验验证了本文的缓存优化系统对提升网页加载速度和节省流量的有效性。实验表明，该系统在用户第一次访问网页时，平均可以节省流量17%，提高加载速度29%；立刻再次访问同一网页，平均可以为第二次访问节省流量66%，提高加载速度9%。

6.2 未来工作展望

在本研究的基础之上，未来可以从三个方面继续进行研究：

1. 以此研究成果作为平台，继续优化浏览体验

本文所实现的系统在开发的过程中一直本着高扩展性的原则，该系统今后可以作为基于资源的移动浏览器优化实验平台，可以在工作的基础之上再进一步地丰富功能，比如说：

* **更细粒度缓存的实现：**现有的缓存是以资源为粒度的，研究表明，可以通过更细粒度的缓存来进一步提高流量的节省，如对HTML中的每一行进行缓存。
* **资源推送优先级的实现：**目前系统推送资源为随机顺序推送，然而，良好的推送顺序能使浏览器更快地加载资源。在资源列表的构造过程中，可以构建出访问一个网站的资源树。在浏览器加载的过程中，只有浏览器加载了根节点资源，其子节点的加载才变为可能。由此可以优先推送位于根节点的资源，提高加载速度。
* **网站预取的实现：**当用户处于Wi-Fi环境下，服务器可以将用户可能接下来访问到的资源自动推送到手机端，在用户下一次访问这些资源时可以进行无流量的浏览。

1. 继续进行试验，验证缓存优化的效果

还需要测量不同时间间隔的缓存优化效果来验证系统的性能。目前实验中的第二次访问是在第一次访问后立刻执行的，接下来可以做相隔半小时、一小时、两小时等不同市场的优化效果。这个实验更符合用户的使用行为，更有意义，然而进行此实验存在很大的技术难题尚未解决。

1. 利用此前制作的应用商店App，开展用户实验

此前，作者制作了应用商店App用来开展用户实验，其功能是将常用网站的快捷方式放在手机的桌面上，便于用户快速打开这些网站。本文的系统可以与该App简单地结合到一起，让用户体验到本系统优化浏览的效果，同时收集用户的访问数据，并进一步发现系统的可提升之处。

## 参考文献

[1]中国互联网络信息中心. 第35次中国互联网络发展状况统计报告[R].2015

[2] Y. Zhu, V.J. Reddi. WebCore: Architectural support for mobile Web browsing[C]. ISCA2014

[] N. Trawny, S.I. Roumeliotis: Indirect Kalman Filter for 3D Attitude Estimation. : <http://www-users.cs.umn.edu/~trawny/Publications/Quaternions_3D.pdf>

[]

## 致 谢

感谢杨芙清院士、梅宏教授和北京大学软件工程研究所，为我提供了宽松自由的学习与研究环境，使我有机会接触并参与软件工程领域的研究项目，让我学习到很多书本上学不到的内容，为我未来的工作打下了坚实的理论基础，并积累了极富价值的工程经验。

感谢网构中间件小组的黄罡老师和刘譞哲老师。在黄罡老师开设的软件工程实验班上，我参与设计并实现了第一版缓存优化系统，在此过程中，两位老师给了我悉心的指导，使我对这个项目产生了浓厚的新区与深厚的感情。非常感谢他们给了我机会，让我可以在毕业设计中继续进行移动浏览器改进的项目。特别感谢我的导师刘譞哲老师，在百忙之中为我的研究指引方向，关心我的工作进度，在我感到懈怠时督促并鼓励我。在撰写论文与准备答辩的过程中，刘老师为我提供了大量有价值的写作建议，并亲自为我修改论文，付出了大量的时间与精力。

感谢网构中间件小组的师兄师姐，为我提供了良好的学术氛围与研究环境。特别感谢马郓师兄，他之前所做的研究是本文工作的重要前提与依据。在我毕业设计的工作中，他不遗余力地对我的工作进行指导，帮助我解决学习与生活上的各种问题，为我提供很多宝贵的研究思路和技术思路，每天花费大量时间和我讨论工作，时常忙碌到深夜。即使师兄远赴国外开会也一直挂念着我的工作，和我保持交流。在撰写论文的过程中，他帮我耐心修改论文的整体结构，逐字逐句地找问题。没有师兄的帮助，我的工作将很难达到现有的高度。感谢陈镇鹏师弟，在忙碌的期中考试周，他利用宝贵的时间帮我完成了部分实验。

感谢大学四年来知道的我的所有老师。老师们的言传身教，让我拓宽了眼界，使我在计算机领域打下了良好的基础。

最后，衷心感谢我的家人，尤其是我的父母。你们一直在背后默默支持我、鼓励我，给我无微不至的关怀，是我大学四年工作与生活中最坚实的后盾。