计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2017,26(7):130−136 [doi: 10.15888/j.cnki.csa.005839]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

Hybrid App开发框架的实现及性能优化①

贾军营[1], 张大成[1,2], 高　春[3]

[1](中国科学院 沈阳计算技术研究所，沈阳 110168)
[2](中国科学院大学，北京 100049)
[3](辽宁大学，沈阳 110036)

摘　要: Hybrid App融合了Native开发和Web开发的优势, 其开发方式在移动应用开发和桌面应用开发中所占的比重越来越大. 本文实现了WebView和Native的双向通信机制, 建立了Hybrid App的开发框架, 并通过对Web数据进行本地离线存储, 对Web文件采用基于字节流的差量更新的方式对框架进行优化. 最后对开发框架和优化机制进行了实际测试, 采集数据并进行分析, 实验结果表明开发框架和优化机制具有实用性和可行性.

关键词: Hybrid App; 离线缓存; 差量更新; 优化机制

引用格式: 贾军营,张大成,高春.Hybrid App开发框架的实现及性能优化.计算机系统应用,2017,26(7):130–136. http://www.c-s-a.org.cn/1003-3254/5839.html

Implementation and Performance Optimization of Hybrid App Development Framework

JIA Jun-Ying[1], ZHANG Da-Cheng[1,2], GAO Chun[3]

[1](Shenyang Institute of Computing Technology, Chinese Academy of Sciences, Shenyang 110168, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)
[3](Liaoning University, Shenyang 110036, China)

Abstract: The Hybrid App combines the advantages of Native development and Web development, which takes more and more proportions in the mobile application development and desktop application development. This paper realizes bidirectional communication mechanism between WebView and Native, set up development framework for Hybrid App, and optimizes the performance of Hybrid App by means of off-line storage for Web data and delta update based on byte stream for Web files. Finally, the development framework and optimization mechanism are actually tested and the collected data are analyzed, and the experimental results reveal that development framework and optimization mechanism have good feasibility and practicability.

Key words: Hybrid App; offline cache; delta update; optimization mechanism

Hybrid App兼具Web App跨平台的优势和Native原生应用拥有良好的交互体验的优势. 目前Hybrid App开发方式有两类, 第一类通过第三方中间件平台实现, 比如借助PhoneGap或者Ionic[1,2]. 第二类是在移动端或者PC端软件中嵌入一个或者多个WebView, 并实现WebView和Native的双向通信, 使得WebView具有访问本地硬件和本地代码的能力, 同时Native也具有访问WebView中JavaScript代码的能力. 本文实现的Hybrid App框架和优化方案基于课题组的融合通信项目, 该项目具有Andorid, IOS和Windows三个终端. 为了提高各终端开发效率, 在已有框架基础上引入Hybrid App开发方式, 最终使得三个终端的界面用一套Web文件来展现. 由于在PhoneGap等平台中提供了大量的本地能力封装机制, 既增加软件的复杂度, 又降低了引入后软件的效率, 而在融合通信系统Web和Native交互的过程中, Native提供的API主要由融合通信系统中已经成熟的业

① 收稿时间: 2016-11-01; 收到修改稿时间: 2017-01-04

务逻辑模块提供, 所以在项目改进过程中只需实现一个轻量级的Hybrid App框架[3,4], 而不需要引入第三方平台.

正文中首先介绍了Native和WebView的双向交互机制, 在此基础上实现了Hybrid开发基本框架, 并针对该框架设计和实现了优化方案, 最后对框架和优化方案进行测试. 在设计优化机制的过程中参考了HTML5技术. 目前HTML5的manifest机制提供了Web文件的离线访问, HTML5的Localstorage机制提供了数据缓存的功能. 但其均存在不足, 当Web文件被改改动, manifest发生改变, 其中所有定义的缓存文件全部重新以全量的方式获取, 消耗流量, 而且控制不够灵活. Localstorage提供了非常易用的API, 通过setItem, getItem, removeItem, clear四个接口可以实现数据离线存储, 但是按照目前标准, 存储空间太小, 浏览器只给每个独立的域名提供5M的存储空间. 虽然HTML5还提供了Web SQL Database, 它拥有一套使用SQL语句操作客户端数据库的API, 在本地建立轻量级的数据库, 但此规范工作已经停止. 综上, 借鉴manifest和Localstorage机制, 利用Hybrid app的Native端的优势, 在Hybrid App开发框架的基础上设计了一套数据离线缓存和Web文件增量更新方案, 提高了hybrid App应用的性能和效率.

1 Hybrid App开发框架

1.1 在软件中嵌入WebView

Hybrid App开发框架需要引入WebView层. Chromium Embedded Framework (CEF)是基于Google Chromium项目实现的一个Web控件, 下面以Windows端嵌入CEF为例. 首先设计一个CWebClient类, 这个类主要完成对浏览器事件的回调处理. 该类需要继承CefClient和各种消息处理接口, 消息处理主要包括CefKeyboardHandler类提供的键盘输入相关的回调处理, CefLoadHandler提供的浏览器页面加载状态的回调处理, CefFocusHandler提供的焦点相关的回调处理, CefLifeSpanHandler提供的页面周期回调接口等. 然后将CWebClient类对象作为参数传入CefBrowser::CreateBrowser生成CEF实例并显示. 浏览器创建后, CefBrowser类用于对WebView进行控制, CefBrowser对象可以在cef窗口创建完成后由CefLifeSpanHandler接口中的OnAfterCreated函数的参数中获取.

1.2 Web和native交互通信机制

Hybrid App开发框架的核心是native和Web的交互通信机制. 通信机制可以使得WebView中的Web页面通过javascript函数访问native的中封装的接口API, 同样native也可以访问WebView中的javascript接口API.

1.2.1 Native调用Web

Windows下基于cef的WebView调用方式如下:

frame->ExecuteJavaScript(“alert(‘ok’)”, “”, 0); 其中frame为CefFrame实例. CefFrame实例可以通过CefBrowser对象实例获取.

Adroid平台和IOS平台对于Native调用Web端的javascript代码都有很好的原生支持.

Adroid中调用的方式如下, WebView.loadUrl ("javascript:(function(){alert('ok');})()");其中Webview是WebView的实例.

IOS中的调用方式如下,

{WebView stringByEvaluatingJavaScriptFromString: @"alert('ok')"}; 其中WebView是UIWebview的实例.

1.2.2 Web调用Native

Windows中CEF可以通过重写CefV8Handler接口的方式实现本地的回调.

Android中可以通过重写WebChromeClient.onJsPrompt, onJsConfirm, 或onJsAler来实现.

IOS中可以通过监控WebView的URL变化实现Web端调用Native.

1.3 Web和native通信的Bridge

综上, 针对windows, android, ios三个平台分别设计三个通信Bridge[11], 通过Bridge连接Web端和native端, 实现Web与Native的双向通信[5]. Bridge提供四个接口, 如表1所示, native的原生代码和Web端的javascript代码可以通过这四个接口进行双向通信, 在通信过程中, 函数名和参数被封装在JSON字符串中. 当Web向native发送请求时, Web端的javascript代码通过调用SendToNative来向native发送请求的JSON字符串, native得到请求之后解析JSON串, 得到函数名和参数, 并执行本地响应的API操作, 完成后natvie端通过调用JsCallback将结果返回给Web端. Natvie请求Web端的过程相类似. Web和Native通过Bridge进行通信的时序图如图1所示.

表1 Bridge的四个函数接口

接口名称	描述	调用方
SendToNative	向native推送一个消息	Webview
NativeCallback	触发native的callback	Webview
SendToJs	向Web侧推送一个消息	native
JsCallback	触发Web侧的callback	native

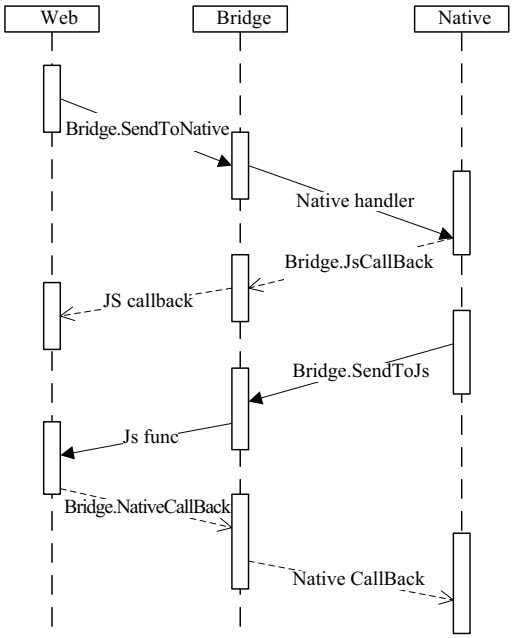

图1 Web和native双向通信机制

2 数据离线存储机制

通讯录信息, 群组成员信息, 通话记录等信息的获取会增加带宽开销, 为了节省带宽, 可以将以上数据进行本地缓存[7]. 基于Web和native通信, 可以实现以上功能.

离线数据以key/value键值对的形式进行存储, 为了实现这样的功能, 设计八个接口, 如表2所示. 同时, 在native端封装对数据库的操作, 当Web端将数据通过json传递到本地后, native解析出key/value, 并将其写入数据库. 以存储离线数据接口CacheStore为例, Web侧实现如下代码即可实现存储数据的功能:

```
CacheStore:function(keys, data){
    var op_cacheStore ={
    "name": "CacheStore",
    "callback": "OnCacheStoreCb",
      "params": {
            "key": keys,
            "data": data
        }
    };
    Bridge.SendToNative(op_cacheStore);
}
```

表2 离线存储的八个接口

接口名称	描述	回调接口
CacheStore	储存离线数据	OnCacheStoreCb
CacheFetch	获取离线数据	CacheFetchCb
CacheClear	清空离线数据	CacheClearCb
CacheDelete	删除离线数据	CacheDeleteCb

3 Web文件差量更新机制

3.1 Rsync算法

Web文件主要包括html, css, javascript文件, 为了节省带宽, 提高加载速度, 将这些文件进行离线存储, 当Web文件发生改变时, 采用差量更新的方式去得到新文件. 本文实现的Web文件差量更新机制基于rsync算法[6-8], 该同步算法可以将两台计算机中的内容不同的文件进行同步, 使之相同. 使用此算法的原因在于其滚动校验和及三级匹配校验和机制能保证在服务器端快速的生成差量信息包. Rsync算法的主要流程是: 机器A将Old文件分块并计算校验和序列, 将序列发送给机器B, 机器B通过New文件和A发送过来的校验和序列, 产生差异包, 再将差异信息包发回给A, A通过差异信息包和Old文件生成New文件. 机器A和机器B进行文件同步主要步骤如下[9]:

1) 机器A将文件Fold进行分割, 得到一系列的字节块B_i, 假设块大小为m节. 则$B_i=F_{old}[im, (i+1)m-1]$, 其中最后一个块可能会小于m个字节.

2) 机器A对每个块计算两个校验和: $U_i = H_u B_i$和$R_i = H_r(B_i)$, H_u使用滚动32位校验和生成函数, 是若校验和生成函数. H_r使用128位的MD5校验和生成函数, 是强校验和函数.

3) 机器A将校验和序列发送给机器B.

4) 机器B建立一个哈希值为16位, 表长度为2^16的哈希表. 将每个块生成的校验序列(U_i, R_i)插入到哈希表中, 其中每个块的R_i映射为16位并作为主键. 此时哈希表中的每一项指向的是拥有相同hash值的校验和列表的第一个元素.

5) 机器B从Fnew第一个字节开始, 通过三级匹配机制进行检索[10], 依次计算长度为S字节的块的32位滚动若校验和并映射为16位的哈希值, 如果哈希值对应的哈希表项非空, 且在Fold生成的哈希字典对应的列

表项存在相同的弱校验和, 则再比较MD5强校验和, 如果相等, 则认为找到了匹配块.

6) 如果找到了匹配块, 差量信息字段中将插入两部分内容, 第一部分为上一次匹配位置和当前位置之间的未匹配的数据内容, 第二部分为当前匹配成功的b块的索引信息. 跳转步骤5, 搜索匹配块的工作会从当前匹配块重新启动.

7) 如果某个位置开始没有找到匹配, 就会前进一个字节, 跳转步骤5继续搜索匹配的块.

8) 如果Fnew整个文件搜索完毕, 机器B将差量信息发送给机器A.

9) 机器A根据差量信息和Fold生成Fnew.

3.2 算法的重要细节

在rsync算法中使用的弱滚动校验和算法基于Mark Adler的Adler32校验算法: 此算法根据$X_1 \ldots X_n$的校验和和X_1, X_{n+1}的字节流值可以简单快速地计算出$X_2 \ldots X_{n+1}$的校验和. 校验算法如公式(1)(2)(3)所示.

$$a(k,l)=\left(\sum_{i=k}^{l} Xi\right) \bmod M \tag{1}$$

$$b(k,l)=\left(\sum_{i=k}^{l}(l-i+1)Xi\right) \bmod M \tag{2}$$

$$s(k,l)=a(k,l)+2^{16}b(k,l) \tag{3}$$

其中$s(k,l)$就是字节块的滚动校验和. 公式(4)(5)利用了递推关系, 可以快速的计算出连续值:

$$a(k+1,l+1)=(a(k,l)-X_k+X_{l+1}) \bmod M \tag{4}$$

$$b(k+1,l+1)=(b(k,l)-(l-k+1)X_k+a(k+1,l+1)) \bmod M \tag{5}$$

3.3 对rsync算法进行改进:

由于Rsync算法用于同步两台机器中的一个文件. 本文对Rsync算法从两个方面进行进行改进.

1) 由于服务器端保存着与客户端相同的旧版本文件, 所以在服务器端分割旧版本文件并生成校验和序列, 这样避免了客户端计算和传输校验和序列的开销.

2) 由于Web文件为多个文件, 为了对不同版本的多个Web文件进行管理, 引入版本号机制. 将同一时间的多个文件设定为相同版本.

当某个或者多个Web文件被修改, Web文件版本号增1, 并对过去版本的Web文件在服务器端生成对应版本的差量更新文件deltaXtoY, 其中X为低版本号, Y为高版本号, deltaXtoY包括更新文件列表和每个列表项对应的文件的增量更新信息, 文件列表对应全部的Web文件, 每个列表项对应一个Web文件. 服务器端生成每个列表项的增量信息的过程如图2所示.

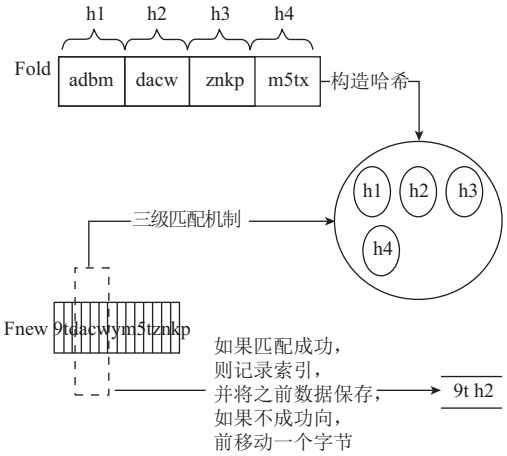

图2 生成每个列表项的增量信息的过程

当客户端启动, 获取最新的Web文件版本号, 检测自己当前版本是否为最新, 如果不是, 则下载对应的差量更新包, 得到差量deltaXtoY后与本地的低版本的Web文件一起生成最新版本的Web文件. 当服务器没有对应版本的差量更新包, 则进行全量更新. 对于每个文件的增量更新过程如图3所示. 旧文件根据对应的差量更新列表项的差量信息, 生成新的文件, 合并过程会读取增量更新信息中的索引信息和数据信息, 并将根据索引信息在Fold中提取数据块, 生成新文件的JavaScipt代码如下:

```
Deltafunction:function(source, blockSize, deltaInfo)
  {
  var strResult="";
  for(var i=0;i<deltaInfo.length;i++)
{
    var info=deltaInfo[i];
    if(typeof (info)=='string')
    {
    strResult+=info;
  }
  else
  {
    Var start=info[0]*blockSize;
    var len=info[1]*blockSize;
var oldcode=source.substr(start, len);
```

```
            strResult+=oldcode;
        }
    }
    return strResult;
}
```

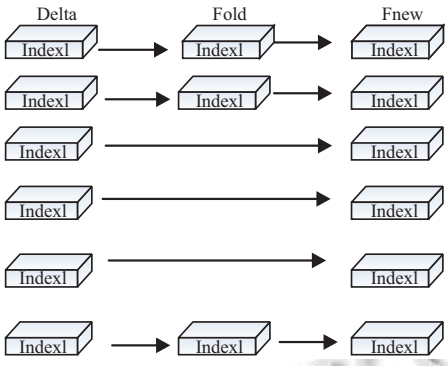

图3 客户端差量更新过程

假设服务器Web文件版本为6, 其中有一个js文件test.js, 其仅有一行数据if(jsonObj.hisTag == ‘end’), 将这行数据修改为if(jsonObj.hisTag != ‘end’). 设块长度为4, 则对文件版本6可以分为6块, 如表3所示.

表3 旧版本数据分块

1	2	3	4	5	6
json	Obj.	hisT	g ==	'en	d')

在服务器端, 通过对test.js, 进行三级匹配机制查找, 最终增量文件表示为数组: [1, 2, 3, “ag !=”, 5, 6], 对应的数据块如表4所示. 进一步简化, 可用一个数组表示为[[1, 3], “ag !=”, [5, 2]]. 由于只有这一个文件被修改, 最终得到的差量信息delta6to7中的文件列表只有一项, 这一项的文件名为test.js, 对应文件的差量信息为[[1, 3], “ag !=”, [5, 2]]. 在客户端, 假设当前Web文件版本为6, 则获取到delta6to7更新信息, 通过与test.js合并, 从test.js就文件中提取出block1, block2, block3, block5, block6, 5个数据块, 并与新数据“ag !=”合并, 得到新的test.js为: block1+block2+block3+“ag !=”+block5+block6.

4 测试及结果分析

4.1 测试Hybrid开发框架

在Hybrid开发框架的基础上, 将软件的界面设计通过Web方式实现, 三个平台的应用采用一套Web文件进行描述, 实现了软件界面的集中开发和集中调试, 软件界面如图4和图5所示. 当用用户通过语音通话界面和消息发送界面与应用交互, Web端会调用Native的业务逻辑, 本地在执行业务逻辑的过程中实时的将状态回调给WebView, WebView更新界面. 同时客户端中将通话记录进行缓存, 每次从本地加载通话记录, 如图6所示.

表4 新版本数据分块

1	2	3	新数据	5	6
json	Obj.	hisT	ag !=	'en	d')

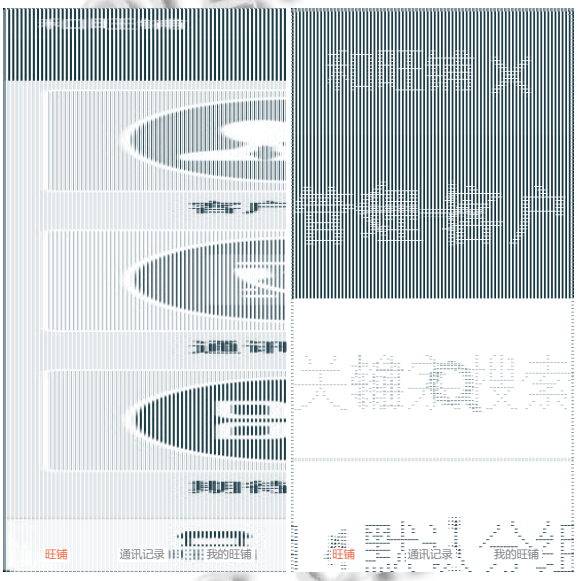

图4 PC端软件界面

4.2 测试离线缓存机制[12]

本地缓存主要节省了客户端与服务器建立连接和传输数据的时间, 试验中在带宽为200KB/s的带宽环境下采集了13次数据, 每次分别从本地和服务器重复三次获取相同字节大小的数据, 获取时间时为从发起请求到数据接收完毕, 单位为毫秒, 计算平均时间并记录. 其中从服务器获取数据通过jquery的AJAX方法获取信息流, 服务器端采用mysql数据库, 客户端缓存数据从sqlite数据库中读取. 实验数据中可以分析出以下趋势: 如图7所示, 从服务器获取数据需要一定建立连接的时间, 随着获取数据量的增多, 获取时间逐渐受到网速的限制, 同时网络状况也会影响获取数据的时间. 从本地数据库获取数据所需的时间基本是随着数据量增多, 呈线性增长的, 相比服务器时间更短.

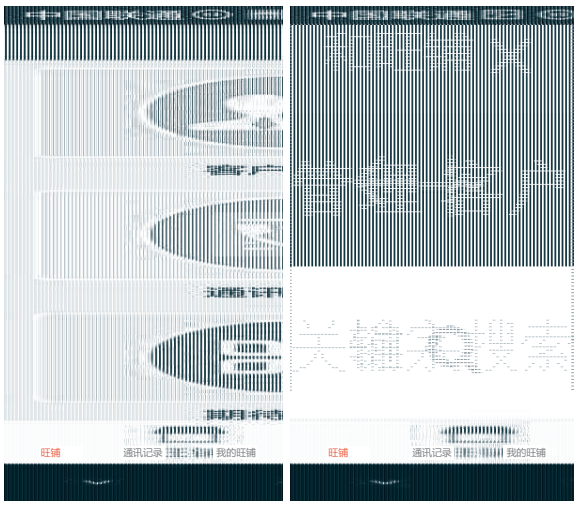

图5 Android端软件界面

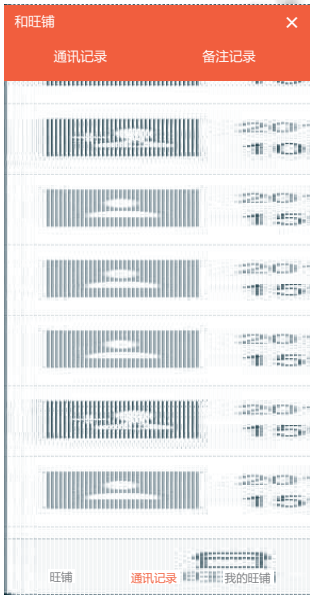

图6 通话记录界面

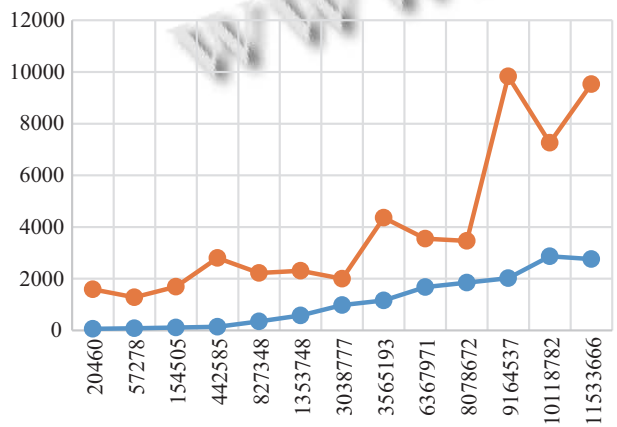

图7 数据的不同获取方式的获取时间对比

表5 数据的获取时间

数据量大小(byte)	本地获取时间(ms)	服务器获取时间(ms)
20460	63	1594
57278	83	1283
154505	112	1694
442585	143	2805
827348	348	2225
1353748	578	2307
3038777	978	2002
3565193	1159	4361
6367971	1678	3551
8078672	1847	3463
9164537	2025	9834
10118782	2869	7261
11533666	2761	9532

4.3 测试差量更新机制

通过对服务器端test.js进行10次修改, 产生了10个不同的版本. 每次的具体修改信息如表格所示. 全量更新所需的传输流量为新文件大小, 差量更新为差量信息包的大小, 服务器端rsync算法划分数据块的大小为700字节. 则每次传输的数据量如图7所示. 横轴表示文件的版本, 纵轴标识需要传输的数据大小. 如图8所示, 位于上方的折线标识每次全量更新需要传输的字节, 位于下方的折线标识差量更新需要传输的字节. 可以通过分析得出如下趋势: 增量信息包的大小与修改文件的位置, 修改文件的内容, 块划分大小有一定的关系, 比如增加已有的内容, 增量信息包体积小一些, 因为新文件中的数据块会在旧文件中的数据块中找到匹配. 通过10次信息对比, 每次增量包的大小基本都与全量数据大小相差100倍. 相对全量更新节省99%的流量消耗, 同时提高传输时间.

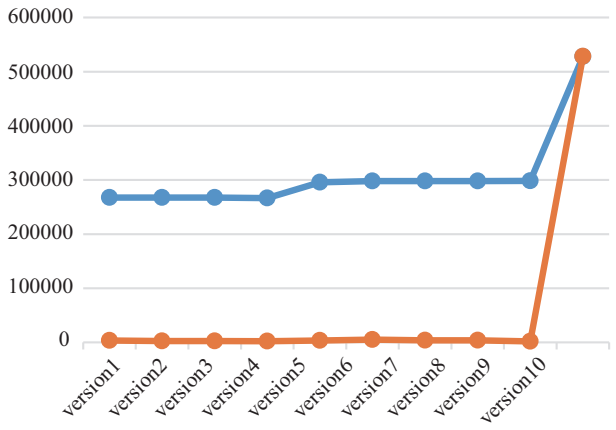

图8 Web文件全量更新和差量更新需要传输的数据量对比

表6 test.js的各个版本的具体信息

版本号	上一个版本文件大小	文件大小	改动字节(bytes)	差异包大小
Version1	266,941	267,440	499(增加已有的数据)	3,548
Version2	267,440	267,446	6(增加新数据)	2,365
Version3	267,446	267,425	21(减少)	2,338
Version4	267,425	266,617	708(减少)	2,247
Version5	266,617	295,503	28,886(增加已有的数据)	3,409
Version6	295,503	298,011	2,508(增加全新数据)	5,027
Version7	298,015	298,015	调整数据块位置	3,935
Version8	298,015	298,015	调整数据块位置	3,754
Version9	298,015	298,111	在开头增加96字节新数据	1,931
Version10	298,111	527,968	替换为全新文件	528,163

5 结束语

本文通过设计Hybrid开发框架, 屏蔽了各终端的差异, 将客户端中的原生界面设计工作转移到Web界面的设计, 提高了开发效率. 同时对Hybrid开发框架进行优化, 提高了Hybrid应用的页面加载速度, 节省了带宽. 综上, 本文设计的Hybrid开发框架具有实用性和可行性, 同时优化方案对其他Hybrid开发方案具有一定的参考价值.

参考文献

1 顾学海, 胡牧, 蒋厚明, 等. 基于HTML5的混合移动应用开发. 计算机系统应用, 2016, 25(5): 236–239.

2 潘春华, 李俊杰, 向花, 等. 基于PhoneGap的智能手机跨平台应用. 计算机系统应用, 2014, 23(7): 106–109.

3 李张永, 陈和平, 顾进广. 跨平台移动Web开发框架与数据交互方法. 计算机工程与设计, 2014, 35(5): 1827–1832.

4 徐隆龙, 李莹, 白静. 移动混合开发框架. 计算机系统应用, 2014, 23(12): 53–59. [doi: 10.3969/j.issn.1003-3254.2014.12.009]

5 施伟, 王硕苹, 郭鸣, 等. 跨平台移动应用中间适配层设计与实现. 计算机工程与应用, 2014, 50(16): 39–44. [doi: 10.3778/j.issn.1002-8331.1208-0481]

6 吕瀛, 刘杰, 马志柔, 等. 一种云存储服务客户端增量同步算法. 计算机系统应用, 2014, 23(10): 152–157. [doi: 10.3969/j.issn.1003-3254.2014.10.026]

7 童丽霞, 何加铭, 陈恳, 等. 基于HTML5技术的Widget引擎内容缓存模型及实现. 计算机应用研究, 2011, 28(12): 4625–4628. [doi: 10.3969/j.issn.1001-3695.2011.12.058]

8 Tridgell A, Mackerras P. The rsync algorithm. Canberra, Australia: Australian National University, 1996.

9 Irmak U, Mihaylov S, Suel T. Improved single-round protocols for remote file synchronization. Proc. IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Miami, FL, USA. 2005. 1665–1676.

10 Gupta D, Sagar K. Remote file synchronization single-round algorithms. International Journal of Computer Applications, 2010, 4(1): 32–36. [doi: 10.5120/ijca]

11 徐凯. 跨终端Web. 北京: 电子工业出版社, 2014.

12 罗圣美, 王蔚, 任文慧. 两种移动应用开发框架的性能测试比较——基于PhoneGap和Titanium. 中兴通讯技术, 2013, 19(3): 44–47.