

復旦大學

本科畢業論文



论文题目：	基于用户所处环境主动发现与推送 的移动应用软件新范式：架构探索 与技术实现
院 系：	计算机科学技术学院
专 业：	计算机科学与技术
姓 名：	王轲
学 号：	14307130048
指导教师：	周扬帆

2018 年 5 月 30 日

目 录

第一章 引言与介绍	1
第 1 节 引言	1
第 2 节 相关工作	2
第 3 节 设计目标、与现有方案的对比	3
第二章 应用运行框架	5
第 1 节 基于 WebView	5
第 2 节 基于 JavaScript 引擎，但使用原生 View	5
第 3 节 直接运行原生代码	6
第 4 节 本论文的选择	6
第三章 应用分发优化	7
第 1 节 数据层面优化——业务代码和依赖库的拆分与动态链接	7
第 2 节 网络层面优化——分布式镜像和就近分发	10
第四章 网络总架构设计	13
第 1 节 总架构图	13
第 2 节 主数据管理服务器	13
第 3 节 应用仓库服务器	14
第 4 节 镜像自动同步	15
第 5 节 利用 Docker 容器进行部署	15
第五章 基于地理位置的服务发现	16
第 1 节 概述	16
第 2 节 安卓端客户端	16
第 3 节 服务器端	16
第 4 节 优化	16
第六章 WIFI 网络内的服务发现	18
第 1 节 实现原理	18
第 2 节 路由器 DNS 解析的配置	18
第 3 节 可能问题与解决方案	19
第七章 基于数字签名的用户鉴权支付	21
第 1 节 总设计思路	21
第 2 节 用户鉴权流程	21
第 3 节 用户支付流程	22

第 4 节	用户信息请求	23
第八章	安卓客户端的实现	25
第 1 节	Cordova 代码的改造	25
第 2 节	用户数据的管理	26
第 3 节	APP 发现的实现	27
第 4 节	本地 APP 代码缓存	27
第 5 节	管理公钥私钥	28
第 6 节	鉴权和支付的 API 的实现	28
第 7 节	桌面小工具	29
第 8 节	总结	29
第九章	应用场景探索	30
第 1 节	停车场缴费	30
第 2 节	餐厅点菜	32
第 3 节	在线考试	33
第 4 节	会议时刻表	36
第 5 节	活动报名	37
第 6 节	应用场景的总结、畅想与商业可能	38
第十章	完全去中心化可能性的探讨	39
第十一章	总结	41
附录：PoC 版本代码链接		42
致谢		43

基于用户所处环境主动发现与推送的移动 应用软件新范式：架构探索与技术实现

姓名： 王轲
学号： 14307130048
专业： 计算机科学

摘要：本论文探索了一种新手机 APP 范式：无需安装注册，根据环境主动推送，即点即用。本论文实现了可以支持这样一种新 APP 范式的框架和网络结构，挖掘了这种新范式可带来的便利性与机会，探索了这种范式技术上的挑战，并针对性地提出了创新的解决方法。

关键字：小程序，LBS，前端构建，前端动态链接，分布式内容分发，数字签名鉴权与支付，HTML5，Hybrid App

Abstract

This essay explores a new mobile app paradigm where users of the app do not need to install the app or sign up in the app. Instead, apps are actively pushed to the user's mobile phone according to the environment they are in, and users can use the apps instantly by clicking on them. This essay designed and implemented an app framework and a network architecture to make it possible, it also tackled some key technological difficulties in an innovative way. Lastly, it demonstrated the convenience and commercial opportunities brought about by this new paradigm.

Keywords: mini apps, location based service, front end building, dynamic module splitting and linking, distributed content delivery, digital signatures, HTML5, hybrid app

第一章 引言与介绍

第 1 节 引言

手机 APP 在人的日常生活中扮演越来越重要的角色，随着 APP 开发成本的降低，APP 数量激增。人们的手机里往往好几屏的 APP，但真正经常使用的 APP 却并没有几个。许多 APP 是“一次性”的，或只在某特定场合、地点使用。现有的技术方案并不能很好地分发这些 APP：如果让用户下载，则费时费力，如果扫描二维码打开，扫码动作仍然较麻烦，且下载耗费流量和时间、体验一般。

本论文设计了一种新的基于用户所处环境主动发现与推送 APP 的技术，把手机视为一个与周围环境产生交互的接口，让用户能够轻松使用和当前所处环境相关的 APP，例如：用户身处地下车库时，自动给手机推送寻车导航和在线缴费的 APP；进饭店后，手机屏幕上自动显示菜单与在线点菜 APP；回到家后自动显示智能家电 APP；坐到车里自动显示车况与导航信息……

本论文中，主动推送要做到的效果是：用户无需人为下载，APP 图标就出现在主屏幕上，即点即开没有等待。如何做到 APP 的快速、省流量传输，是本论文要攻克的一大难点。本论文用一种综合而又创新的方案试图解决本问题，设计了编译时的开发者插件和一个分布式网络来支撑 APP 分发。

同时，本论文在用户体验上更进一步，设计方案以免去用户每次使用新 APP 时都需要重新注册、填写个人信息的麻烦，甚至这些 APP 中经常进行的在线支付操作，本论文也设计了一套方案来代理完成。如何合理又安全地设计这些方案，最大程度地方便用户和开发者，也是本论文的重要研究课题。

最后，本论文尝试把这个新系统投入使用，制作了能运行它的安卓客户端，并对 5 种使用情景做了样例 APP、深入探索其应用情景及潜在的商业价值。

第 2 节 相关工作

2.1 微信扫码打开小程序

微信通过扫描二维码可以启动第三方程序，效果和本论文很相似，微信的方式已经在各大商业广场被广泛使用。本毕业设计和微信扫码的对比请看下一章节“和微信扫码打开小程序的对比”。

2.2 IOS 基于地理位置的 APP 建议

从 iOS 10 开始，iPhone 锁屏的左下角可以显示该地理位置常用的 APP，在锁屏界面可以快速启动它。然而，这个功能却不是基于地理位置的 APP 发现，因为它通过分析用户过往的行为达成的。如果一个用户在某个 GPS 位置多次打开某个 APP，那么这个位置会和 APP 产生关联。此外，只有已经安装好的 APP 会被显示，并不具备发现新 APP 的能力。

2.3 运营商基站定位

手机用户到一个新环境时，往往可以收到一条欢迎短信。这是因为手机连接了当地的新基站，通过基站连接，用户被定位。然而，这种技术并不能拿来做 APP 发现，因为它依赖基站硬件，且设备需要有 SIM 卡才能连接（即 iPad、PC 无法连接）。此外，它的定位精度也较低。

第 3 节 设计目标、与现有方案的对比

3.1 目标概述

本论文的设计目标可以概括为：让手机成为和周围环境交互的入口、APP 无需安装，即点即用。

预期达到的效果是：用户手机上自动跳出和用户所处环境有关的手机 APP，并以最小的数据传输量自动下载到本地，用户点击即可使用。

APP 只在用户身处某个环境时显示，在用户离开那个环境后就隐藏。即手机上只显示会被用到的 APP，不显示无关的 APP。

最终要达到的目标是：让手机真正成为一个与周围环境发生交互的平台，成为用户能力的延伸。

3.2 和微信扫码打开小程序的对比

3.2.1 两者的类似之处

1. 两者都能在无需安装的情况下，启动一个新第三方 APP。
2. 两者都可以以某些途径建立用户和周围环境的连接，达到如餐厅在线点菜的效果。
3. 两者都有机制允许用户一键授权登入，无需重新注册。

3.2.2 两者使用上的不同

本论文旨在把整个 APP 启动过程加速到最快，“快”和“主动推送”是和微信扫码打开小程序的区别。

拿停车场缴费为例，要在微信上实现停车场缴费，你需要：

1. 点开微信。
2. 打开扫一扫。
3. 对准二维码，对焦，调整（光线差的环境下，扫二维码并不快）。
4. 关注微信公众号。
5. 在微信公众号菜单中找到停车场缴费功能。
6. 等待网页加载（取决于网络环境，通常要好几秒）。
7. 输入自己的车牌号。
8. 缴费。

而使用本论文中的方案，你只需要：

1. 点开早已在手机主屏等候的停车场 APP。
2. 如果是第一次运行类似程序，则输入自己的车牌号。
(否则可以直接通过用户信息授权的方式自动输入)。
3. 缴费。

微信扫码的方法相比较而言，过程繁琐、耗时长，用户如果不主动扫码的话是不会看到服务的，即用户主动而服务被动。

本论文的方法中，相反，服务会主动去触及它的目标用户，便利度有所提高，让用户更有可能使用商家开发的服务性 APP。

3.2.3 目标、定位上的不同

目标定位上，本论文也和微信有很大不同。

微信的关注点是连接，更注重的是把每一位客户和商户连接在一起。所以微信做公众号和基于公众号上的小程序。而本论文的关注点是服务、体验，只在意用户是否可以在最短的时间内实现他的意愿。

微信的目标是把商户作为整个庞大微信生态圈的一环，让商户依赖微信、互利共赢，在给商户提供便利的同时巩固微信对使用者的地位。而本毕业设计做的是一种开放式的标准，就像一种协议一样，给所有人带来便利却不被某一个机构所掌控。

第二章 应用运行框架

手机操作系统本身是一个应用运行的框架，本论文计划做出的产品是一个手机操作系统之上的应用运行框架（一种虚拟框架）。它的本质是另一个手机 APP，但它要能运行别的程序，即充当框架的角色。

让一个手机 APP 动态运行别的虚拟 APP，大致有以下三种方法：

第 1 节 基于 WebView

这种方法使用最为广泛，是“hybrid app”最早的含义，代表解决方案有 Cordova、PhoneGap。（Dalmaso, 2013）

它的本质就是把网页包在一个原生 APP 的容器里（APP 只有一个内嵌的网页，显示被嵌入的 HTML5 网页），达到 HTML5 APP 看起来是一个原生 APP 的效果。

除此之外，这种方案还能解决一个额外的问题：WebView 的局限性。WebView 是不能读本地文件的、也没有接口让 WebView 打开闪光灯等，但这种方式做的 APP 可以在 WebApp 和原生容器之间通过类似 ipc 的技术建立起一个通道。WebView 把调用原生接口的意愿告诉原生容器，原生容器进行调用并把结果通过 ipc 返回给 WebView。这样就相当于 WebView 有了调用任何原生接口的能力，在能力上能做到和原生 APP 等同。

这种方案有很多优点：跨平台（一次开发，各种手机操作系统使用）、开发便捷（使用 HTML5 的技术）、健壮稳定（是历史最悠久的混合 APP 开发技术）。

缺点主要在于：性能较低，WebView 中运行的是 HTML、CSS，和渲染原生组件相比，WebView 执行渲染时复杂很多。因此尤其在低端机型上，会略微卡顿、无法达到 60fps。

第 2 节 基于 JavaScript 引擎，但使用原生 View

这种方法近几年来比较流行，代表解决方案有 React Native 和 NativeScript。阿里巴巴的 Weex 和腾讯的微信小程序也均是使用这种方案。

它同样使用 JavaScript 运行时作为代码的执行器，不同的是，它不再使用 WebView 来做界面的呈现，而是使用 ipc 的方法，让 JavaScript 动态地告诉原生宿主 APP，要渲染什么原生组件、放在什么位置。

好处是解决了 WebView 的性能问题，因为真正使用原生组件，所以真正有原

生 APP 的流畅体验和交互风格，也能到达 60 fps。

坏处是这种方案和 WebView 比稳定性不那么好（因为是新方案、和原生打交道更多），可定制性没那么强（界面是原生组件+属性，和 WebView 强大的 CSS 比起来弱一些），有些效果难以找到跨平台一致的原生组件，必须为不同的操作系统写不同的代码。

第 3 节 直接运行原生代码

以上两种方法都是使用到 JavaScript 来支持自定义程序的，因为基于 JavaScript 引擎的方案不涉及到原生代码的编译，比较稳定可行。直接运行原生代码的方案也有，如直接动态下载和修改安卓的 .dex，腾讯的 tinker 可以做这样的热修复。

但直接运行一个崭新 APP 的原生代码，这种方案应当还不存在。这样的话相当于一个 APP 替代了操作系统一部分的功能，需要所有的 APP 权限。很有可能操作系统会在 APP 权限、资源分离、元数据定义等方面给这种方案设置阻碍。

第 4 节 本论文的选择

本论文还是选择最成熟、使用广泛的技术，来作为应用运行框架的基石。

本论文选用 Cordova 作为基础，在修改 Cordova 的代码之上添加各种新特性，来实现所有的功能。

具体实现请看后文“安卓客户端的实现”一章。

第三章 应用分发优化

要做到瞬间打开 APP、主动推送,APP 的传输是一个要仔细考虑的技术难点。要开辟一个新的专用信道来做这件事情,并部署到全世界,是不可能的,所以还是要使用因特网(TCP/IP)。但如果只是简单地下载 APP,因为每个 APP 动辄几十兆,所以用户的流量很快就跑光了,这样也不可行。

如何做到主动推送、秒开、节省流量,这是本毕业设计最大的难点。

第 1 节 数据层面优化——业务代码和依赖库的拆分与动态链接

1.1 概述

这是本论文最有创意的一项研究成果,在小程序传输方面尤其奏效。网上查不到任何类似做法的记载,很有可能是首次探索。

这种做法的核心就是:把业务代码(APP 程序员为了实现 APP 服务写的代码)和依赖库代码在构建和发布时拆分开。下载 APP 时,只下载业务代码和本地没有缓存过的依赖库代码,把它们进行链接后运行。

要知道这种方法为何可以大大减少传输流量,请看以下分析:

一个 HTML5 APP 中,除去图片(图片一般运行时按需在线下载,亦无需考虑),剩下的体积基本都是 JS 代码。负责业务逻辑的代码和依赖库代码比起来少之又少。拿停车缴费程序为例,程序员自己可能只写 3000 行代码,但它依赖的 ionic 有 10000 行代码、Angular 框架有 30000 行代码、lodash 有 20000 行代码、Bootstrap 有 2000 行代码。可见,APP 虽然体积大,但是和业务真正相关的代码可能只占总体积的 10%。

所以,如果能把 APP 的业务代码和依赖库代码分拆开,就可以让所有依赖库代码只传输一次,之后,只传输业务代码而不传输依赖库代码。这种分拆、运行时动态组装的方式,尤其在长远看来(运行一段时间后各种常见依赖库都缓存过了),可以大大减少传输体积。

APP 初次安装时也可以预置最常用的依赖项代码包,在有 WIFI 连接的闲时,也可以自动更新新的、被广泛使用的依赖项代码包,这些操作可以进一步节省初次传输依赖包的流量。

接下来,本论文叙述具体的实现方法。

1.2 编译时拆分依赖库和代码主体

JS 的包管理系统已经相当完善,比较现代的 JS 前端都会使用 npm 包管理

(类似 maven)。

npm 中，一个项目会有一个 `package.json`，其中会定义它的所有依赖项（也是 npm 包的形式），和它的主入口（如 `index.js`）（npm, 2018）。

因此，HTML5 APP 在发布时，可以参考 `package.json` 中的信息进行分拆操作，即抽取所有依赖项的代码分开打包，留下主入口的代码（业务代码）和依赖项说明的元数据。

具体实现如果从零开始做会十分困难，好在有成熟的构建工具 `webpack`，提供各种配置选项和 API 接口。本文将借助 `webpack` 打包工具提供的 API 来进行业务代码和依赖库代码的分开构建（Webpack, 2018）。

1. 默认的 `webpack` 构建中，依赖包和所有的业务代码会被打包成一个 `.js` 文件，涵盖所有代码。而现在要达到的效果是，这个 `.js` 文件将只含有业务代码，不含有依赖包代码。这个 `.js` 文件中，所有对依赖包代码的引用会被替换成对 `window.$deps.[依赖包名_依赖包版本]` 这样一个全局变量的引用。要达到这样效果，可以使用 `webpack` 的 `external` 配置项和一些别的参数（具体配置文件可以参考“代码实现”一节中的“`app-builder`”）。
2. 然后，需要对 `package.json` 中定义的 npm 依赖项进行分析。由于 `package.json` 中记录的都是依赖项的 `semver`（版本范围而非具体的依赖项版本），分析的第一步是把这些 `semver` 确切化成 APP 实际使用到的版本。`npm shrinkwrap` 命令可以用来做这件事情。
3. 读取 `shrinkwrap` 的输出，输出中的每一个依赖项转换成 `webpack` 的一个 `entry`，并调用 `webpack` 的 API 对它们分别进行构建。使用 `output.library` 配置项，可以把构建出来的 `umd` 模块注册到 `window.$deps.[依赖包名_依赖包版本]` 全局变量中。
4. 业务代码和依赖库代码都构建完成后，生成一个元数据文件，说明这个 APP 的所有依赖项，并打包成 `.tar.gz`。

经历完以上四步，一个业务代码和依赖库代码分离的前端 APP 构建就完成了。把构建出来的业务代码和依赖代码都上传到应用仓库服务器，就完成了 APP 发布的过程。

1.3 运行时动态拼装与链接 APP

运行时，首先是下载 APP 的元数据，然后根据哈希值下载所有的文件。

下载时注意查本地缓存，如果缓存里已经有某一个哈希代表的文件那么就不

重复下载（哈希相同一定能保证文件内容相同）。

文件全部准备就绪之后，生成一个 HTML 文件来依次载入依赖和业务代码、同时注入 `getUserInfo`、`getUserIdentity` 等额外的 API 供应用使用。

这一步很简单，只要通过字符串拼接，在 HTML 源码中插入各条 `<script src="...">` 即可。

插入的顺序为：注入代码最先、依赖库代码其次、业务代码最后。

Web 模块化的原理保证了依赖库之间代码顺序可以随意。因为每个依赖库都是自包含的，不会出现要处理依赖链路的问题，这大大简化了问题。

HTML 代码生成完之后，写入一个本地的 HTML 文件，然后开启 Cordova 的 WebView 界面并加载那个本地的 HTML 文件，就可以成功启动 APP。

1.4 优化效果分析

本优化方法的具体效果严重取决于缓存命中率。在 HTML5 手机 APP 的领域，有许多体积大的库非常流行，被广泛使用（jQuery、Bootstrap、React、Angular、Vue、Ionic）。jQuery 甚至被 73.2% 的网站所使用（Usage of JavaScript libraries for websites, 2018）。这些库都可以通过预装、WIFI 闲时更新，或缓存未命中时动态下载的方法，缓存到用户的手机中。运营机构也可以给开发者提出建议，让开发者在开发新应用时首选使用一个列表中的一些库，让优化效益更加明显。

在运行一段时间后，本地缓存应该可以覆盖绝大多数常用的重型依赖库。一些特殊 APP 用到的小众依赖库还是没法覆盖到，需要在线下载，但这些库往往体积不会很大。

本论文共有 6 个样例 APP（“应用场景探索”的 5 个和 API 测试的 1 个），技术栈相仿，用到的大型依赖库是 React、Lodash 和 Bulma，部分 APP 引用了小型依赖库（如 `react-syntax-highlighter`）。以这 6 个 APP 为例：

APP 名	普通打包方式	业务代码大小
car-park	571,348 bytes	18,878 bytes
exam	575,449 bytes	22,979 bytes
menu	414,430 bytes	20,953 bytes
schedule	410,178 bytes	16,701 bytes
signup	569,683 bytes	17,213 bytes
auth-test	1,158,739 bytes	16,793 bytes

可见，只传输业务代码而不传输依赖项，可以节省 96.93% 的流量。

初次传输时，假设本地没有任何依赖项的缓存，需要从远程下载所有的依赖项。

依赖项包大小如下：

依赖项包名	大小
bulma	158,991 bytes
axios	14,660 bytes
react	25,133 bytes
react-dom	313,221 bytes
react-syntax-highlighter	623,400 bytes
lodash	72,850 bytes

可见，在这种情况下，算上依赖项大小，仍然可以节省 64.27% 的流量。

1.5 额外说明

本次测试中，APP 使用的依赖项包高度近似。实际生产环境中，如前文所述，问题会比较复杂。

之前的分析可以大致得出结论：生产环境虽然复杂，压缩率仍然会很可观。而且即使在最坏的情况下，使用本技术和不使用本技术比，使用本技术也不会造成任何额外的开销（最差的情况是持平）。

如果使用预装依赖项、WIFI 内闲时下载依赖项等技术以增加缓存命中率，会对存储有额外开销。开销可以由用户决定，如自己设置磁盘配额（如 1GB）。1GB 的磁盘配额可以缓存至少 5000 个依赖项（使用率从高到低排序），可以非常可观地提高缓存命中率（我的估计是可以命中 95% 以上）。

第 2 节 网络层面优化——分布式镜像和就近分发

本节只阐述概念上的设计，具体实现请参见后文“网络总架构设计”一节。

2.1 概述

前文所叙述的方法可以大大减少下载 APP 时要传输的数据量。剩下的那些数据量如何做到更有效率的传输（甚至不损耗任何移动数据流量地传输），是本章节的内容。要做到更有效的传输，基本思路就是就近下载。数据离用户越近越好，越近速度越快、中央服务器负载越小。如果数据离用户近到在同一个内网中，这

是最理想的情况，用户甚至不用耗费任何移动数据流量。

本论文就以就近下载为指导思想，设计并实现了支持多级多镜像就近分发的网络架构。

2.2 在 NAT 内部部署内容分发节点

如果内容分发被部署在 NAT 内部，那么就像内网传文件一样，速度会很快、很稳定，而且不会产生任何费用。

本论文的设计里，通过对 WIFI 网关的 DNS 做修改、在 LAN 内部署 APP 仓库镜像，这一效果可以被达成。这样，用户可以从局域网下载 APP 而不是从远程服务器下载，传输速度更能显著加快。

具体方案和实现请见“WIFI 网络内的服务发现”一章。

2.3 在同城部署内容分发节点并通过 DNS 进行就近解析

如果做不到在 NAT 内部部署节点，也可以在同城部署节点，并使用 DNS 智能线路解析来把用户引向同城节点。

DNS 智能线路解析是一种配置在 DNS 服务器上的功能，能够按照运营商线路细分，在不同地域把相同的域名解析成不同的 IP 地址。

以阿里云提供的 DNS 解析服务为例，智能解析可以做到 4 运营商、31 省份解析 (Alibaba Cloud, 2018)。有了这种解析能力，就可以设置一个统一的域名，在不同地域将它解析到那个地域下的服务器镜像上。这种改进对 APP 代码和用户都是透明的，却能优先使用同城内的服务器，加快速度、减少消耗。

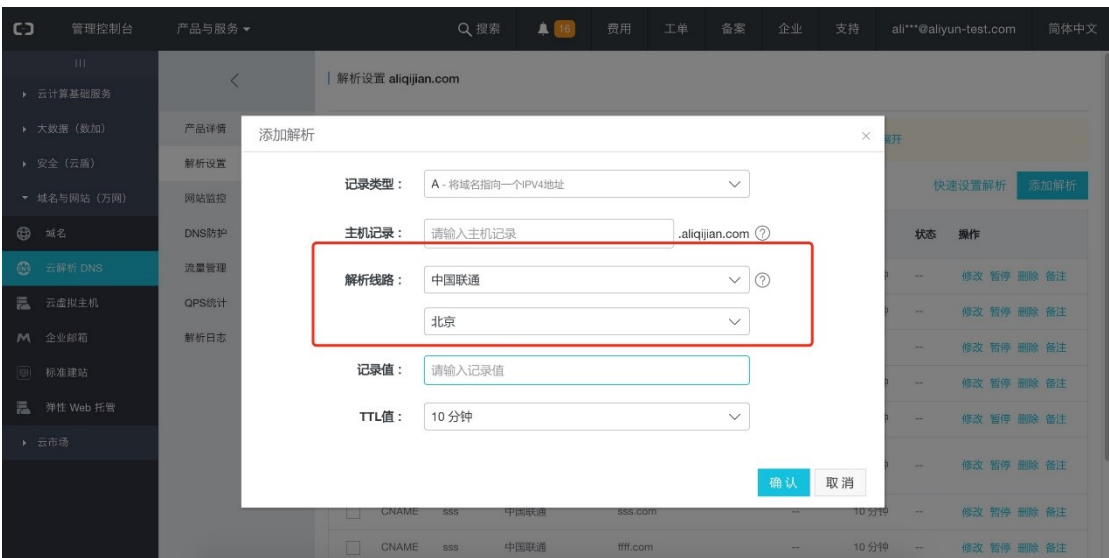


图 2.3 (a) 阿里云 DNS 智能线路解析

由于 DNS 解析服务和多省份镜像部署都很昂贵，所以本毕业设计并没有真正做这方面的实验。

2.4 根据哈希值存储依赖库或业务代码

考虑到 CDN 实现上的方便，本论文的设计是：不存储任何目录结构和文件之间的关联，文件只用一个哈希值唯一确定。

即构建一个 Key-Value 库，Key 是文件的哈希值，Value 是文件的内容。文件内容是业务代码、依赖库代码还是图片，这些均不关心。

APP 仓库中会在 MySQL 数据库里维护元数据，元数据中要引用文件时，直接通过指定文件的哈希值来进行。如 APP A 的业务代码哈希值为 a2fe71...，有 3 个依赖，哈希值分别为 743916...、c433b8...和 71f6c3...。

这样做的好处是易于管理，利于内容分发节点的同步，当一个镜像中没有某个哈希代表的文件时（缓存不命中），可以根据哈希值，精确从源拉取需要的文件。

2.5 传输时压缩

JavaScript 以可见字符而非二进制的形式存在，如果直接传输这些可见字符，传输效率会很低（1 个 Byte，0-255 中只有 95 个可见字符），因此，需要在传输时把 JavaScript 压缩成二进制的。二进制打包本身就可以节约很多流量，如果加上基于字典的压缩，流量能更进一步被节省。大多数网站已经这样做，他们使用 GZip (deflate) 压缩，压缩率一般可以达到 70%-90% (Grigorik, 2018)。Brotli 是一种新的压缩算法，在压缩率和压缩速度上均好于普通的 deflate 算法 (Alakuijala, 2015)。手机端的浏览器并不一定支持 Brotli，但在安卓客户端中，可以使用 Brotli 的 Java 实现，来对服务器传输来的数据进行解压缩。这样，就可以利用这种新压缩技术带来的好处。另外，普通的 Web 服务器是在请求时动态压缩被请求的内容。如果换成静态压缩（即预先把内容压缩好），每次请求速度还能快接近 100ms (WAGNER, 2017)。使用 HTTP2，可以把对多个文件（业务代码包文件和多个依赖库文件）的请求合并成一个请求，节省了多次 TCP 连接建立的过程，实测发现也可以节约 10%-20% 的加载时间 (ATTARD, 2017)。

第四章 网络总架构设计

本节阐述网络架构方面的设计，这是本论文对前文所述“网络层面优化”的具体实现。

第 1 节 总架构图

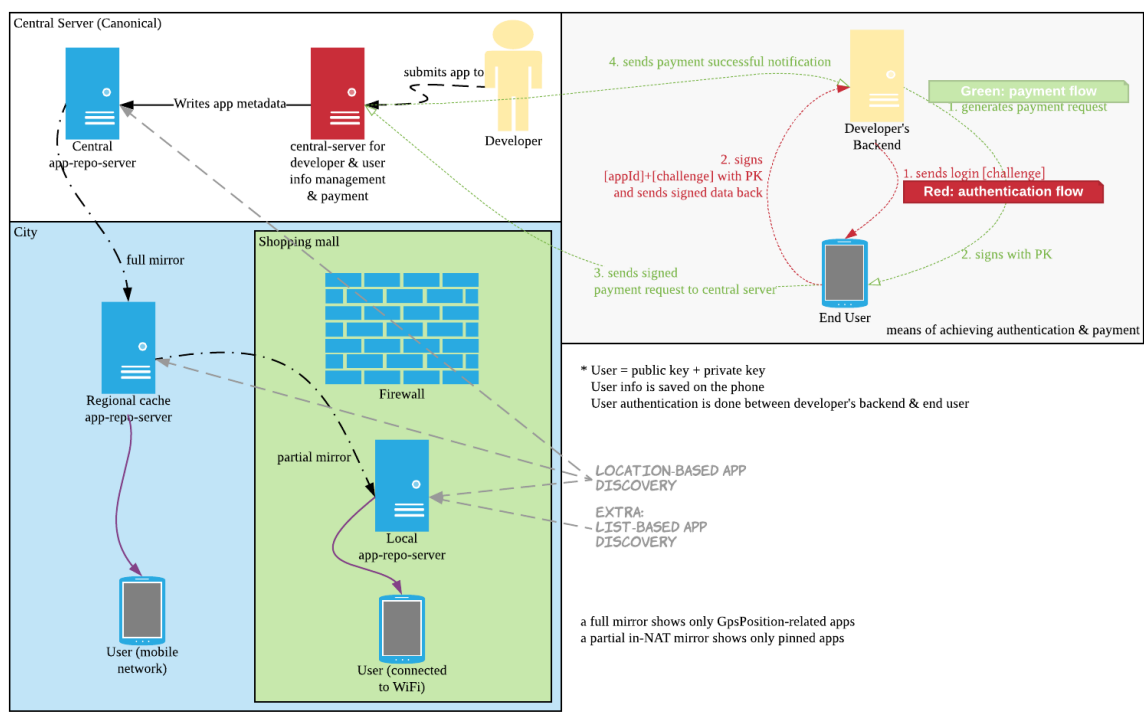


图 4.1 (a) 网络总架构示意图

如图所示，网络方面有两种服务器：数据管理服务器和应用仓库服务器。

数据管理服务器是中心化的（现阶段的设计，还是用了一些中心化服务器，更多讨论请见“完全去中心化可能性的探讨”），负责 APP 元数据管理、用户信息管理。

应用仓库服务器是分布式的，多副本分布在多个不同的地域（甚至在 NAT 内部，对于大商场而言），负责 APP 代码的镜像和分发。

第 2 节 主数据管理服务器

主数据管理服务器是一个中心化服务器，用来管理用户数据、APP 数据、维护根据地理位置发现 APP 时需要使用的元数据、管理 APP 内支付。

开发者把构建好的应用程序包上传到主数据管理服务器，服务器会对它进行

解包分析处理。会读取压缩包内的 `meta.json` 文件，以辨别 APP 的名称、版本、依赖项信息。随后，服务器会分别对业务代码和依赖库计算哈希并存入文件库，并在数据库中插入相应的实体数据、关联数据。同时，开发者可以设置 APP 发现的具体参数：在什么经度和纬度、多少米范围内可见该 APP、在这个条件下启动该 APP 所携带的参数。这些数据会被插入到中央应用仓库服务器中，镜像自动同步工具会把这些数据同步到世界上所有别的节点（见后文），这样，APP 发布就完成了。

主数据管理服务服务器的另一个作用是 APP 内支付代扣。为了方便与统一 APP 内的支付，本架构中不建议开发者自己接入支付服务，而是另外开发了一套基于数字签名的用户支付授权体系（见 基于数字签名的用户鉴权支付）。开发者可以在 APP 内调用一个统一的接口，请求用户支付，用户同意之后开发者会得到一个带有用户数字签名的支付单，需经由自己的后端转发给主服务器。主服务器就担任了验证用户数字签名、支付信息留档、联系银行转款给开发者的角色。

主服务器在现在的架构中，全世界只有一台，由一个机构专门管理。而主服务器却也有做成去中心化的可能性（见完全去中心化可能性的探讨）。如果真正上线运行，主服务器的负载是会非常大的，好在主服务器的业务都可以横向拓展。真正部署时，需使用多实例、负载均衡的方式部署主服务器。

第 3 节 应用仓库服务器

APP 仓库服务器，现在的架构中有三种类型：主服务器附属应用仓库服务器（权威、唯一真相）、分布在各城市的仓库服务器、分布在各 NAT 内的镜像服务器。

主服务器附属应用仓库服务器在前文已经介绍过，是作为唯一真相存储和管理所有的元数据和文件，并等候别的服务器的数据同步请求（见后文）。

分布在各个城市中的服务器，相当于 CDN 的作用，它们的数据和主服务器完全一样（使用了镜像自动同步工具）。它们的功能就是就近分发 APP、分散负载。

NAT 内的镜像服务器，类似分布在各个城市中的服务器，他们把就近分发做得更进一步：只会使用内网流量。但还有一个额外的功能，就是可以通过配置的方法，提供基于 LAN 的 APP 发现的列表（即：当用户连接进这个 WIFI 后，主动弹出哪些 APP）。

总体来说，应用仓库服务器是客户端会直接联系的服务器。客户端会把当前 GPS 坐标发送给服务器来进行查询，以完成基于地理位置的 GPS 发现。服务器的响应会是 APP 元数据列表，每一项都包含了 APP 名称、图标、主体业务代码哈

希、依赖项列表和它们的哈希、启动参数等等。客户端收到这些信息后，会找出自己本地不含有的哈希，并联系应用仓库服务器进行下载。三种应用仓库服务器略有区别，却相互补充，一起完成 APP 发现和分发的任务。

第 4 节 镜像自动同步

分布在各城市/商场内的应用仓库服务器镜像需要定期和主应用仓库服务器进行数据同步。

数据同步主要有两方面的内容：元数据的同步，文件的同步。

元数据存储在 MySQL 数据库中，要实现增量更新，可以通过查询 `updated_at` 大于上次更新时间的所有记录，并把这些记录应用到本地数据库的方法来达成。这些记录中会携带文件哈希（如一个新依赖包的元数据记录会携带这个依赖包的代码哈希），如果这个哈希值对应的文件在本地不存在，则需要把它从服务器下载到本地，这就是文件的同步。以上两个步骤完成后，可以保证镜像的内容和主服务器的内容是完全一致的。

本论文使用了 Node.js 来编写一个概念验证式的增量镜像同步工具，请参考“代码实现”的“mirror-tool”章节。

第 5 节 利用 Docker 容器进行部署

要支撑起这一章节所述的庞大网络架构，方便廉价的部署方案非常重要。

因为要部署的点多，所以要求要有一键部署的能力。

因为在部署 NAT 内应用仓库服务器时，各商场不一定愿意购置新的设备，更愿意复用旧的设备，旧设备操作系统、运行时五花八门，所以虚拟化很重要。

Docker 的容器级虚拟化是解决这些问题的良好途径 (Docker, 2017)。我为主数据管理服务器和应用仓库服务器都制作了 Docker 镜像，来允许一键安装部署（见“附录：POC 版本代码链接”）。

主数据管理服务器的 Docker 镜像中，包含主数据管理服务器和从属的应用仓库服务器，它们在安装后会自动相互协作。应用仓库服务器的 Docker 镜像中，包含其代码本身和镜像自动同步工具，同步工具已设置好，会定时进行增量更新。如果是部署在 NAT 内的，也可以改变它的配置，来达到 LAN 内应用发现的能力。

Docker 的虚拟化屏蔽了硬件和运行时差异、提供了一键部署的能力，它的额外开销又非常小。这是一种理想的部署技术。

第五章 基于地理位置的服务发现

第 1 节 概述

基于地理位置的服务发现是本论文核心主动发现方法之一，但技术上却没有太多的难度。如前文所述，主数据管理服务器已经管理好了 APP 发现元数据（经度、纬度、发现半径、启动参数），剩余要做的就是 在安卓客户端中汇报客户的位置，与在应用仓库服务器上根据位置筛选出符合发现条件的应用。

第 2 节 安卓端客户端

安卓客户端上，使用到了安卓原生的 `LocationManager` 来检测用户位置的变化。通过设置 `requestLocationUpdates` 参数，可以让安卓系统在用户每移动 10 米、每 5 秒通知一次 APP 用户最新的 GPS 坐标。获得坐标后就可以向服务器请求附近区域的 APP 信息，并下载到本地，等待用户点开。

第 3 节 服务器端

服务器端上，收到用户 GPS 坐标后，可以使用球面距离公式计算用户位置到 APP 位置之间的距离，并返回距离小于发现半径的 APP。

第 4 节 优化

以上是最原始的想法，但对计算能力和带宽而言，都是一种浪费。以下是几种优化方法：

4.1 安卓端休眠优化

如果用户在行驶的地铁上，他的地理位置会快速变化。此时，如果他把地铁沿途所有的 APP 都下载到手机上，会造成巨大的浪费（用户移动数据流量的浪费和中央服务器计算能力的浪费、应用仓库服务器流量的浪费）。解决方法是引入休眠的概念，当 APP 发现程序在后台运行（即桌面小工具对用户不可见时），进入“休眠模式”，减少请求。“休眠模式”具体的做法是：当地理位置稳定时（2

分钟内改动小于 100 米)，才发送发现 APP 请求。

4.2 应用仓库服务器数据库 APP 分区

在有很多 APP 的情况下，将用户发来的 GPS 坐标和每一个 APP 的 GPS 坐标做比较，会造成巨大运算量，是不现实的。解决方法是进行“APP 分区”。因为 APP 的发现半径不会太长（小于 1km 才有意义），所以可以把所有空间坐标分成 1km*1km 的区。具体操作方法是：定义函数

$$f(lat, lng) = floor((lat + 90) * 100) * 36000 + floor(lng * 100)$$

为区号。1 纬度大约 111km，所以把经度、纬度都乘以 100 后，对应的就是 1.11km*1.11km 的一个区域。把经度、纬度当成两个基，相加后，即可得到地球上一片区域的标志。然后，处理请求时，只从 $f(lat - 0.01, lng - 0.01)$ 、 $f(lat, lng - 0.01)$... $f(lat + 0.01, lng + 0.01)$ 这九个区域中找 APP 来比较（或者通过判断其值，一般可以决定只从四个区域来寻找），可以大大缩小查找范围。

4.3 使用曼哈顿距离代替球面距离

进一步减少服务器端运算量的方法是使用曼哈顿距离代替球面距离。虽然这种方法会造成一定误差，但是它可以把每次循环中的 2 次平方运算、1 次开平方运算，替换成一次加法运算，大大节省计算力。

第六章 WIFI 网络内的服务发现

除了基于地理位置的服务发现，还应做 WIFI 网络内的服务发现。

这种情景适用于移动的物体（如列车、飞机、私家汽车）。

同时，做 WIFI 网络内的服务发现，更加是为了实现就近分发。当发现 WIFI 网络内有应用仓库服务器时，就无需走任何外网流量，而是可以直接通过局域网，把所需的 APP 传送到用户的手机上。

第 1 节 实现原理

DNS 查询是访问任何服务的重要一环，全球只有 13 台根服务器，而真正做 DNS 查询时，不可能直接访问任何一台根服务器，因为 DNS 服务器有层层缓存。

DNS 缓存的特性，却反而可以用来做有意义的事情：即智能 DNS、根据不同的线路给出不同的解析结果。事实上，商业上已经有很多类似的使用了，如图片 CDN，虽然是同样的域名，在上海和在北京解析出的 IP 地址确是不同的。在上海会解析出上海的某台机器的 IP 地址，在北京会解析出北京的。这样可以做到透明的就近传输，加快速度、减少不必要的线路流量。

基于 DNS 的服务发现，有一套标准，称之为 DNS-SD (S. Cheshire, 2013)。

本毕业设计使用一个自创的简单版本的 DNS-SD，即在路由器的 DNS 解析上做一些修改，让路由器能够把某个特殊的域名解析到一个内网的 IP 地址，达到发现的效果。

第 2 节 路由器 DNS 解析的配置

本论文尝试使用了一台装有 OpenWrt (LEDE) 的路由器做配置 (MT7621A)，



图 6.2 (a) 路由器 SSH 截图

根据 OpenWrt 的文档 (OpenWrt, 2016)，我们可以修改 `/etc/config/dhcp` 的配置，增加一行 `config domain` 的记录，即把 `lan-app-repo-server.appd` 解

析到内网某台主机（比如 192.168.1.217）。

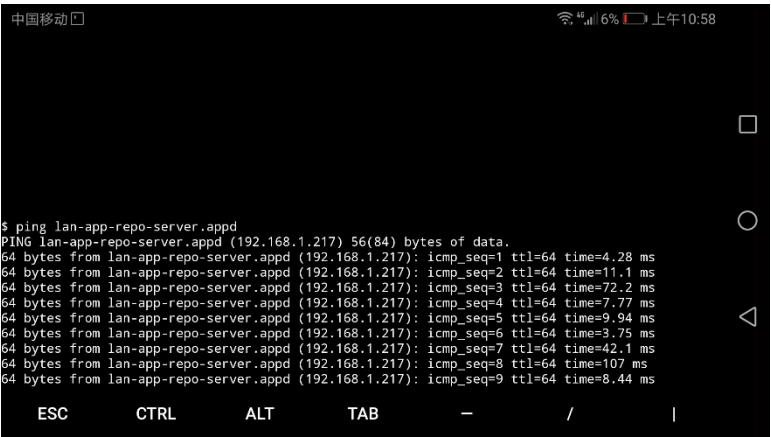
```
option leasetrigger '/usr/sbin/odhcpd-update'

config domain
    option name 'lan-app-repo-server.appd'
    option ip '192.168.1.217'

- /etc/config/dhcp 39/39 100%
```

图 6.2 (b) 路由器配置图

应用后，连入该无线网络的手机就可以访问到专门设置的内网服务器网址。



```
中国移动 6% 上午10:58

$ ping lan-app-repo-server.appd
PING lan-app-repo-server.appd (192.168.1.217) 56(84) bytes of data:
64 bytes from lan-app-repo-server.appd (192.168.1.217): icmp_seq=1 ttl=64 time=4.28 ms
64 bytes from lan-app-repo-server.appd (192.168.1.217): icmp_seq=2 ttl=64 time=11.1 ms
64 bytes from lan-app-repo-server.appd (192.168.1.217): icmp_seq=3 ttl=64 time=72.2 ms
64 bytes from lan-app-repo-server.appd (192.168.1.217): icmp_seq=4 ttl=64 time=7.77 ms
64 bytes from lan-app-repo-server.appd (192.168.1.217): icmp_seq=5 ttl=64 time=9.94 ms
64 bytes from lan-app-repo-server.appd (192.168.1.217): icmp_seq=6 ttl=64 time=3.75 ms
64 bytes from lan-app-repo-server.appd (192.168.1.217): icmp_seq=7 ttl=64 time=42.1 ms
64 bytes from lan-app-repo-server.appd (192.168.1.217): icmp_seq=8 ttl=64 time=107 ms
64 bytes from lan-app-repo-server.appd (192.168.1.217): icmp_seq=9 ttl=64 time=8.44 ms

ESC CTRL ALT TAB - / |
```

图 6.2 (c) DNS 解析效果图

如图所示，安卓手机成功通过 DNS 发现了内网服务器。发现成功后，框架会首先向该内网服务器发送 LAN APP 发现请求（即获得 APP 列表），然后，所有的文件传输请求都会先尝试经过内网服务器，当内网服务器无法提供某文件时才通过 DNS 解析走最近的全量服务器。通过内网传输，不仅快速，而且能帮助用户节省移动流量。

第 3 节 可能问题与解决方案

1. 从有 LAN 服务器的网络切换到无 LAN 服务器的网络，或反向切换时，如何通知程序就近下载所需文件。

解决方案：如果等要下载文件时才去判断是否为有 LAN 服务器的网络，势必会造成效率的降低，因此可以设计一个后台服务定期监控 LAN 服务器的可用性。PoC 代码中，实现了 LanServerAvailabilityMonitor，每隔 5 秒（或在安卓网络情况变化广播时）检查 LAN 服务器的可用性并设置标志。在要进行下载时，会根据这个标志来决定是访问 LAN 服务器还是访问公网服务器。如

果访问 LAN 服务器失败,会重新从公网服务器进行下载并把标志设置为 false。

2. DNS 缓存会导致从一个有 LAN 服务器的网络切换到另一个有 LAN 服务器网络时,不能及时重新解析 IP 地址(新的网络环境下还是解析出旧网络环境下的内网 IP)。

解决方案: 首先, 设置一个比较小的 DNS TTL (比如 10 分钟)。局域网的 DNS 查询应当不会造成太高的服务器负载。然后, 可以设置 n 个不同的域名, `dns0.lan.app-discovery`、`dns1.lan.app-discovery`、`dns2.lan.app-discovery`……依次访问, 如果访问超时(即缓存了错误的 IP 地址), 就访问下一个。总会到某一个域名, 没有 DNS 缓存(比如 `dns5.lan.app-discovery`), 这时一个 dns 查询会发向路由器。路由器上, 只要符合 `/^dns(\d+).lan.app-discovery$/`, 都解析到同一个 IP 地址。

第七章 基于数字签名的用户鉴权支付

以前所有 APP 和网站都要分开注册登入，一人管理很多个账号，很麻烦。

OAuth 的出现较好地解决了这个问题，现在许多 APP 都支持“微信授权登入”、“使用 QQ 账号登入”，或“Sign in with Facebook / Google”。要把体验做到极致，这种一个账号处处登入（Single Sign-On）的功能也必不可少。

我希望把系统做得尽可能的去中心化，所以我尝试了一种新的方案。数字签名可以用来鉴定用户的身份（Fischer, 1991），基于数字签名，我可以做用户鉴权登入，我甚至可以做到在线支付。

第 1 节 总设计思路

用户身份的鉴别依托于非对称密钥的数字签名。这样，每个用户的身份可以由其公钥代表。

需要授权登入时，只要在本地对一个登入请求字符串（载荷）进行签名，把载荷、公钥、签名一同发送给第三方后端，在第三方后端处进行签名校验，即可以完成授权登入的行为。

这样的授权登入，无需经过中央服务器，为以后可能的进一步去中心化改造打下基础。

第 2 节 用户鉴权流程

2.1 前端流程

初次运行程序时，自动生成代表本用户的公钥私钥对（存储在本地私有存储区域——WebView 无法访问到）。

APP 内调用接口 `sys.getUserIdentity` 来进行授权请求，会弹出对话框询问用户是否答应授权。

当用户点击答应授权，系统自动生成一个登入字符串“APP:【APP 名】:【时间戳】”，如“APP:car-park:1526356193580”，并使用私钥对本字符串进行签名（`RSASh1WithSHA1`）。

把私钥、登入字符串、签名一同发送给第三方后端，做身份验证。

2.2 后端流程

后端收到后，先验证登入字符串的内容是否合法，“APP:car-

park:1526356193580”是否符合正确的格式，APP 名是否正确，时间戳和当前系统时间比是否在 1 分钟之内。这些检查可以避免攻击者拿给别的 APP 的登入授权信息欺骗自己的 APP，或截取到用户很多天前的一次登入授权信息后，无限次使用。

第二步要做的检查就是签名是否正确。因为只有真正的用户才有私钥，所以如果签名正确，就可以判定请求者一定是私钥的持有人——用户。

这两部判断完之后，用户的身份被成功鉴定为公钥的内容，在数据库里加一张公钥到 id 的转换表，就可以像普通的授权登入一样执行后续操作了。

第 3 节 用户支付流程

实现授权登入之外，本论文还要能实现便捷的付款功能，即计费支付服务也由系统代理，一键付费。付款同样可以使用数字签名的原理来进行，只是为了保证安全性、逻辑严密性，步骤会多一些。

3.1 支付时序图

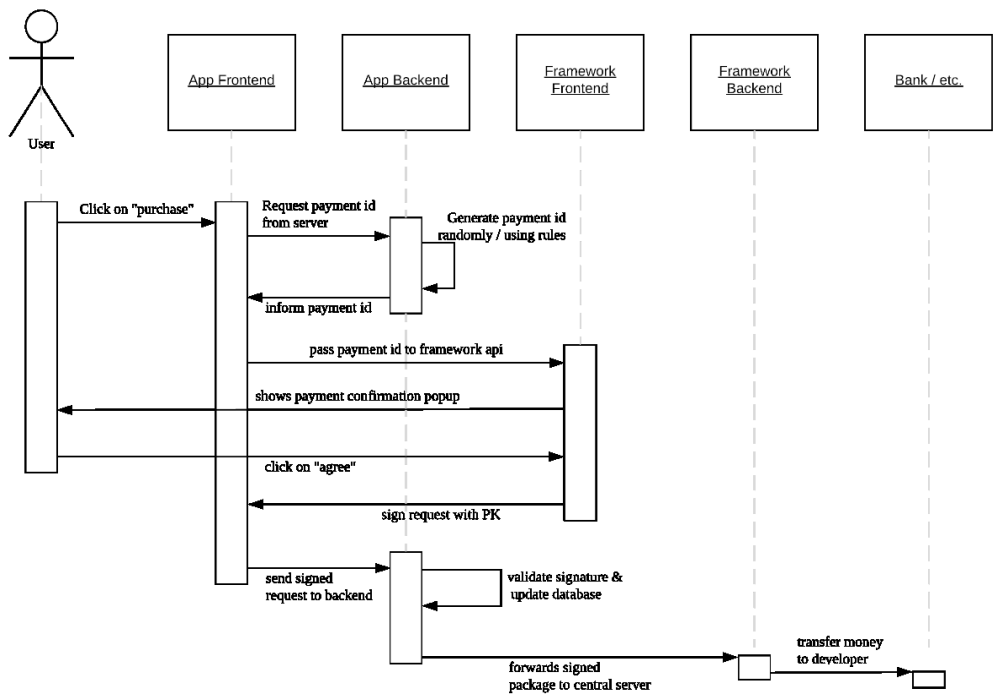


图 7.3 (a) 支付时序图

这是付款操作的时序图，可以看到每次付款操作会有用户、APP 前端、APP 后端、框架前端、框架后端（本网络的中央服务器）、银行，这六方共同完成。

时序图的具体解释如下：

1. 用户点击 APP 内的“支付”按钮。
2. APP 前端向 APP 后端发送订单内容、请求支付 ID。
3. APP 后端记录下订单内容、随机或以某方法生成支付 ID，并告知 APP 前端支付 ID。
4. APP 前端把支付 ID、支付金额和支付说明传给框架前端。
5. 框架前端弹出一个支付确认框（WebView 之外，因此 APP 前端无法操纵这个框）。
6. 用户点击“同意支付”。
7. 框架前端使用用户私钥签署支付请求，并把签名返回给 APP 前端。
8. APP 前端把签名发送给 APP 后端。
9. APP 后端验证签名、更新数据库并发货。
10. APP 后端把签名转发给中央服务器。
11. 中央服务器验证签名通过后，联系银行进行代扣，转账给开发者。

3.2 分析

这种支付方式是基于数字签名进行的，可以保证其安全性。其最大的优势在于使用便利。对用户来说，用户无需进行支付方式选择；对开发者来说，无论是前端还是后端，都可以用非常少的代码量来完成（验证数字签名的函数可以打包在 SDK 中直接让开发者调用）、无需接入普通支付服务繁琐且各不相同的业务。

还有一个有趣的点是：验证支付是否合法，可以只在客户端和 APP 后端两者之间进行，无需牵扯到中央服务器。虽然验证扣费是否成功等还是需要联系中央服务器。

第 4 节 用户信息请求

SSO 做到的是用户登入，只能用来把用户的信息确定为一个公钥。但这样仍然不方便，因为很多 APP 里会需要用到用户姓名、邮箱等常用信息，用户还是得每次都填写。

为了解决这个问题，引入一个补充性的机制：用户信息请求。

这一部分其实和数字签名没什么关系，只是统一管理用户数据。比如用户姓名、用户手机号、用户邮箱地址。

在 APP 里可以通过一个 API（`sys.getUserInfo`）来获得全部或部分用户数据。当然，最终用户可以选择允许或者不允许，这也是对隐私的保护。

技术实现上非常简单，只要在本地进行 Key-Value 存储，用户同意请求之后，读取 Value 发送过去就行，不再赘述。

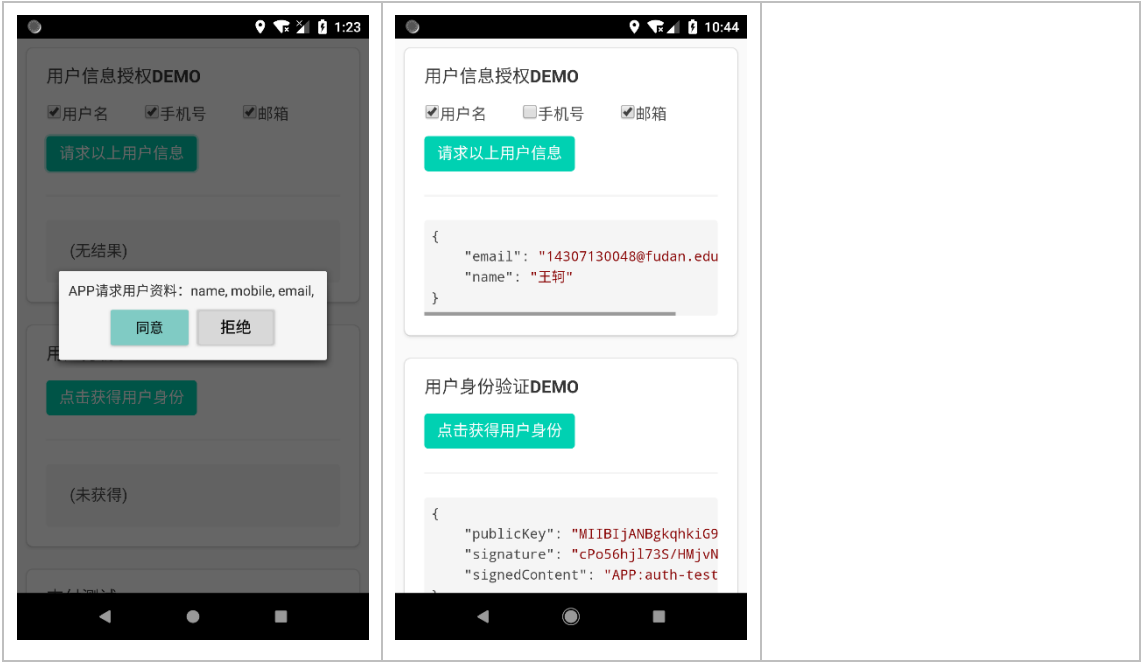


图 7.4 (a) (b) 用户信息请求效果图

第八章 安卓客户端的实现

第 1 节 Cordova 代码的改造

如前文所述，安卓客户端会基于 Cordova 来制作。Cordova 是 Apache 基金会下的开源项目，用来制作跨平台的 HTML5 混合 APP。

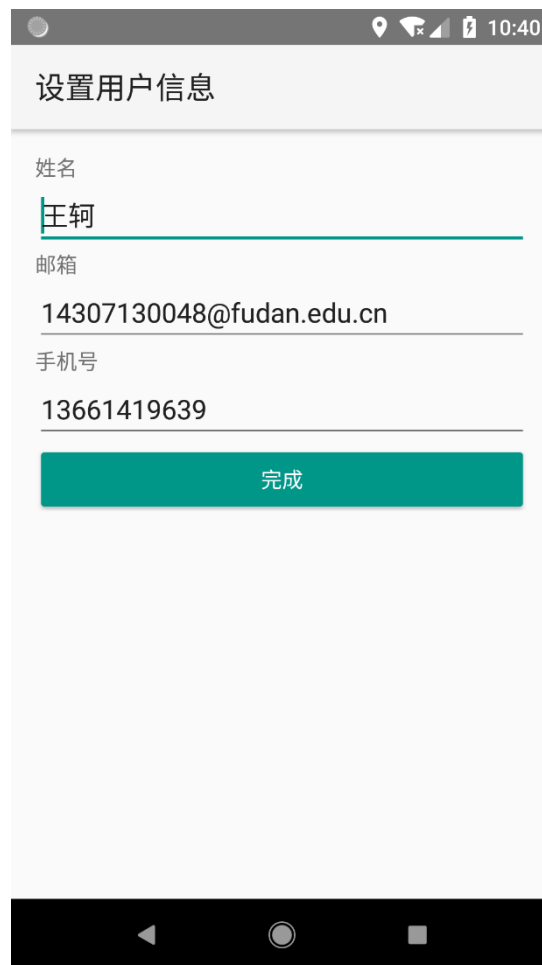
Cordova 的核心功能是打通了 JavaScript 运行时和原生代码之间的壁垒，让它们可以相互沟通，还支持了一套插件体系，把原生功能抽象成 JavaScript 接口（统一的 JS 接口下，安卓、iOS 等有不同的原生代码实现），让开发者来按需安装和调用（Cordova, 2013）。这是本论文需要的功能。除此之外，Cordova 可以给 HTML5 制作的 APP 加上一层外壳，让它有主屏幕 logo、启动界面等，运行起来和普通 APP 一样。这是本论文不需要的功能。

本论文的代码基于 Cordova 的安卓版本实现，保留需要用的、去掉不需要用的、添加额外功能。

具体改造后的代码可以参见附录中 `android-app` 的链接，改造点大致有如下：

1. 去掉默认打开 `assets/` 下 HTML5 APP 的行为。
2. 添加 APP 列表界面，发现并显示 APP。
3. 改造 `WebViewActivity`，使得 Cordova 可以运行多个不同路径的 APP。
4. 添加用户信息设置、用户公钥私钥对生成的功能。
5. 添加本地文件缓存功能（根据哈希缓存文件）。
6. 提供额外 JavaScript API 来进行鉴权、支付等操作。
7. 添加后台服务监测内网服务器可用性，并优先进行内网传输。
8. 提供桌面小工具，允许用户快速启动应用。

第 2 节 用户数据的管理



The screenshot shows a mobile application interface titled "设置用户信息" (Set User Information). It features three input fields: "姓名" (Name) with the value "王轲", "邮箱" (Email) with the value "14307130048@fudan.edu.cn", and "手机号" (Mobile Number) with the value "13661419639". A green "完成" (Finish) button is located below the input fields. The top status bar shows the time as 10:40 and various system icons. The bottom navigation bar is visible at the very bottom.

图 8.2 (a) 用户数据管理效果图

目前只做了三个用户信息字段，用户可以自行设置。APP 可以自由地请求这些字段中的若干个，用户也可以自由地允许/拒绝请求。

一个更加完善的用户数据管理系统，应该能支持更多类型的值设定（如车牌号等），还应该支持反向设定（如用户起初没有设置过车牌号，但是在一家停车场输入了自己的车牌号，那么这个输入的数据应自动存入用户信息字段，以后别的停车场 APP 就可以请求这个字段，用户无需重新输入）。这种高级的自动补全本毕业设计还没有实现。

第 3 节 APP 发现的实现



图 8.3 (a) APP 列表效果图

这是 APP 发现的主界面，一个列表，包含当前环境下的所有被发现的 APP，点击即可使用。这个列表会综合显示基于地理位置发现的 APP 和基于 LAN 发现的 APP，具体发现方法请参见对应章节。

APP 被发现后，会立刻进入下载队列，由于之前做过的 LAN 内分发和业务代码、依赖库分离，下载不会消耗多少时间，也不会消耗用户的数据流量。通常，当用户掏出手机时，APP 早已下载好，可以使用。但如果 APP 还没有下载好，用户会得到提示，APP 会在下载好的瞬间被打开。

第 4 节 本地 APP 代码缓存

本地 APP 代码的缓存非常容易，是基于哈希值的（请参考“根据哈希值存储依赖库或业务代码”章节），直接以哈希值为文件名，写入文件系统。安卓客户端中有一个统一的获取文件方法，传入哈希值，得到文件内容。它的逻辑是：先

判断本地文件系统中是否有这个哈希值命名的文件，如果有，直接返回。如果没有，则从最近的网络节点根据哈希值拉取文件，写入文件系统并返回。这个过程已经达到了缓存的效果，而且缓存的时间是无限久（哈希可以直接表征文件内容）。

第 5 节 管理公钥私钥

在本毕业设计中，公钥私钥对至关重要，是用户身份的象征，支付等重要操作也需要用私钥进行数字签名后才可以完成。

已经完成的安卓客户端中，是这样完成这件事情的：启动 APP 后，从 `SharedPreferences` 中读取公钥私钥对，如果没有，则用 `KeyPairGenerator` 自动生成一个并保存。

因为公钥私钥数据是和应用运行时完全隔离的，所以可以保证 APP 无法在未经用户允许的情况下，访问到这些数据。

第 6 节 鉴权和支付的 API 的实现

鉴权和支付的具体流程已经在前文说过。现在，在安卓客户端上，我通过 Cordova 为 APP 提供额外的接口来请求鉴权和支付。

所有这类接口都被放到了 `window.sys` 全局变量下，这个全局变量会在运行时被注入到 JavaScript 引擎中（PC 端也进行了兼容处理，即使用 JavaScript 的方式，实现了同样的 API 接口，以提高开发体验），有 `getUserInfo`、`getUserIdentity`、`requestPayment` 方法。这些方法的参数可以由调用者指定（如要请求哪些用户信息、要支付多少元）。这些方法的内部，会调用 `cordova.exec` 方法，随后，一个原生类的原生方法会被触发（`class Auth extends CordovaPlugin`）。这个类的逻辑中，会弹出原生 `Dialog`，询问用户是否同意。这个对话框是 `WebView` 之外的，因此基于 `WebView` 的 APP 不能操纵它，可以保证安全性。

当用户同意之后，原生类方法会继续执行逻辑，读取用户的私钥、对支付请求进行签名，并把签名发回给 APP。

如此一来一回，便安全地实现了鉴权和支付 API。

第 7 节 桌面小工具

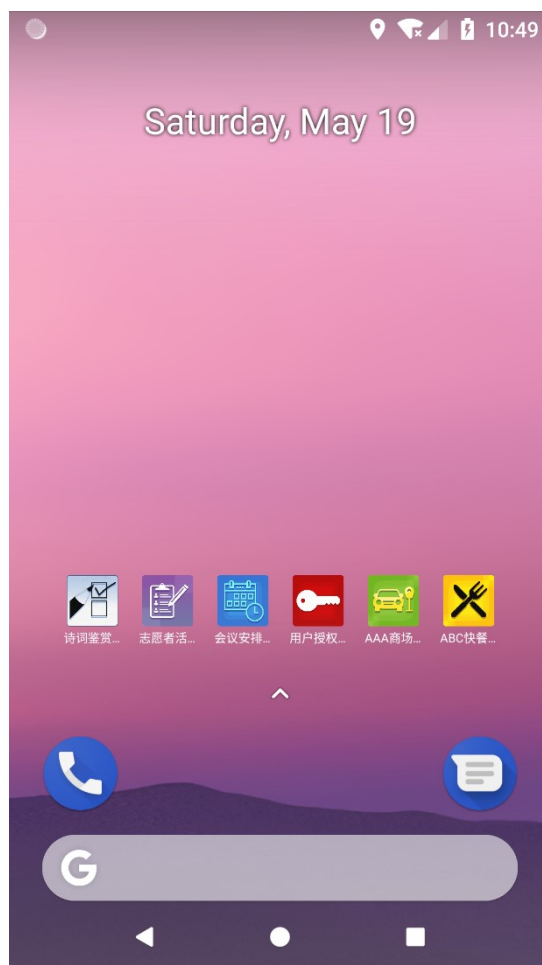


图 8.7 (a) 桌面小工具效果图

如图所示，我制作了安卓“桌面小工具”来让这些动态发现的 APP 看起来像原生 APP 一样。其实这些 APP 图标是桌面小工具画出来的，只是 APP 列表的另一种呈现方式。用户点击图标就可以像点击原生 APP 图标一样，弹出 APP 的界面。

第 8 节 总结

以上几点是安卓 APP 中用到的技术亮点或实现框架的重点。许多安卓开发细节在此隐去不说，具体可以参见附录中的源代码链接。本安卓 APP 是实验性质的，有许多和操作系统紧密相连的优化并没有做（如桌面小工具的更新可能导致耗电量剧增），日后如果投入生产，需要做深入优化。此外，如果要投入生产，一个 iOS 版本的客户端也必不可少。iOS 的客户端由于苹果公司的各种限制，一定更加难做，有可能会需要 iOS 系统工程师自己去做，并成为 iOS 系统的一部分。无论如何，一个可以看到效果的安卓客户端已经完成，这也是本论文的目的。

第九章 应用场景探索

这个章节里，我精选了 5 个典型又容易实现的场景，制作了演示应用。这些演示应用虽然简陋，却前后端兼备，用到了各个我设计的接口。它们不仅能给用户一个更直观的认识，更在我分析框架性能时给出了一定的参考。

第 1 节 停车场缴费

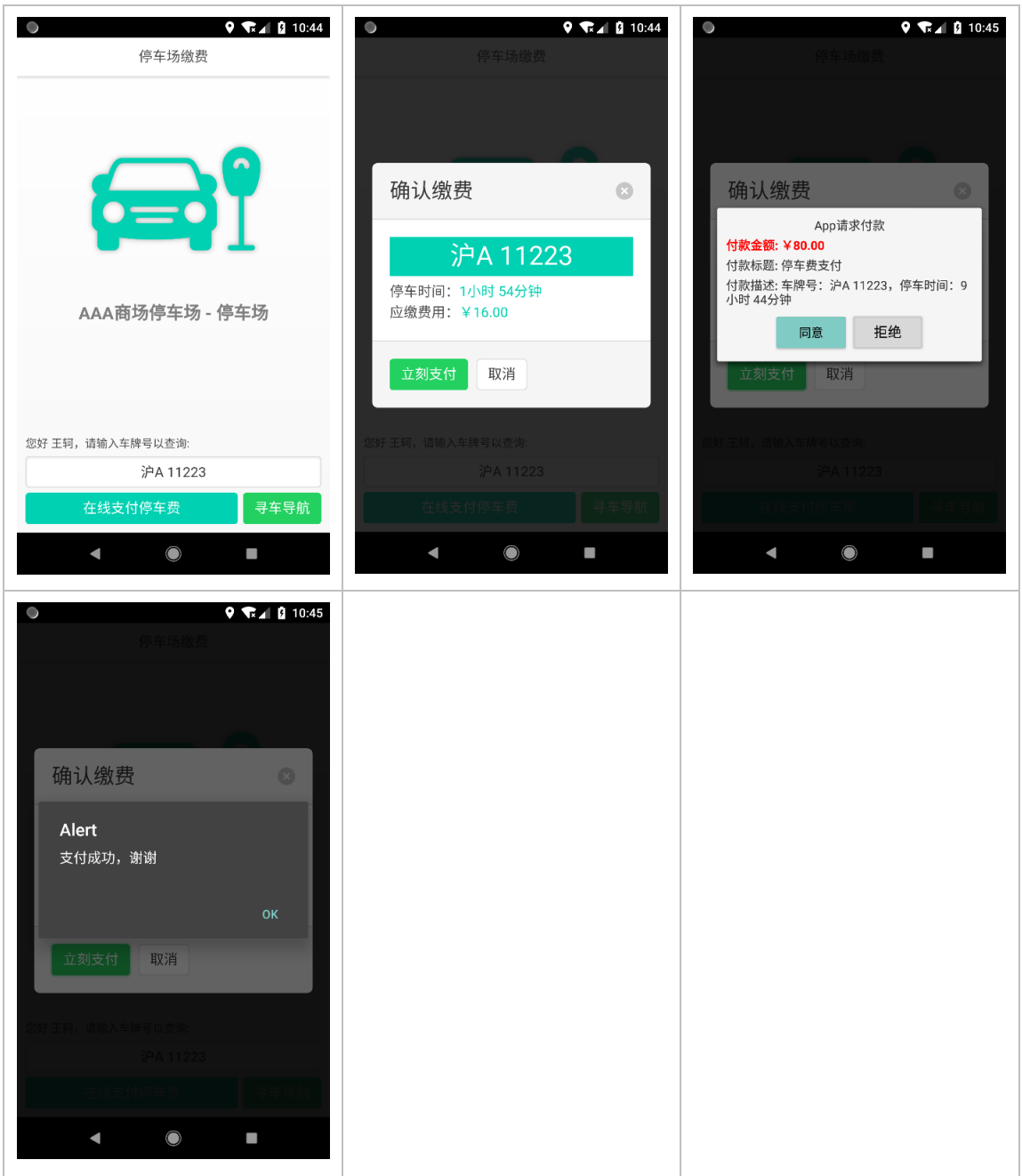


图 9.1 (a) – (d) 停车场缴费 APP 效果图

这个 DEMO 同时涉及到读取启动参数（获得停车场 ID）、请求用户身份、请求 APP 内支付等功能，有一个独立的后端，实现了支付时序图的完整流程。

支付框弹出之前，会请求停车场缴费 APP 的后端生成订单号（数据库会记录订单应付金额、订单内容，并把“已付款”设置为 false）。

前端会根据这个后端返回的订单号请求支付，用户同意后进行签名，发回给后端，后端随即把“已付款”设置为 true。

car_park_payments @demo-apps (@localhost) - 表

id	user_public_key	license_plate_number	amount_paid	created_at	updated_at	paid	parking_minutes
10	MiIBjANBgqhkiG9w...	沪A 11223	7200	2018-05-12 08:12:16	2018-05-12 08:12:19	1	494
11	MiIBjANBgqhkiG9w...	沪A 11223	7200	2018-05-12 08:15:28	2018-05-12 08:15:31	1	494
12	MiIBjANBgqhkiG9w...	沪A 11223	7200	2018-05-12 08:19:10	2018-05-12 08:19:14	1	494
13	← 预先生成订单id，不一定支付成功	aaaa	0	2018-05-12 08:30:31	2018-05-12 08:30:31	0	254
14	MiIBjANBgqhkiG9w...	aaaa	4000	2018-05-12 08:30:37	2018-05-12 08:30:39	1	254
15	MiIBjANBgqhkiG9w...	aaaa	2400	2018-05-12 08:30:50	2018-05-12 08:30:53	1	173
16	MiIBjANBgqhkiG9w...	沪A 11223	7200	2018-05-12 08:49:07	2018-05-12 08:49:30	1	524

图 9.1 (e) 停车场缴费 APP 数据库

如图，这是停车场缴费程序后端的数据库，有些记录“已付款”为 true，有些记录“已付款”为 false（虽然向后端请求了支付，但并没有发支付授权签名给后端。是因为用户拒绝了支付请求）。

后端验证完支付授权签名之后，会把支付签名转发给中央服务器。中央服务器随即会验证支付授权签名，在验证通过之后，调用银行转账的接口把费用转给开发者。

ID	WebApp	User	Amount	Title	Description	Time
1	auth-test (3)	MiIBjANBg...	100.00	支付测试	测试APP内支付功能	2018-05-11 10:47:10
2	car-park (4)	MiIBjANBg...	72.00	停车费支付	车牌号: 沪A 11223, 停车时间: 8小时 14分钟	2018-05-12 08:12:20
3	car-park (4)	MiIBjANBg...	72.00	停车费支付	车牌号: 沪A 11223, 停车时间: 8小时 14分钟	2018-05-12 08:15:31
4	car-park (4)	MiIBjANBg...	72.00	停车费支付	车牌号: 沪A 11223, 停车时间: 8小时 14分钟	2018-05-12 08:19:14

图 9.1 (f) 中央服务器支付记录

如图，这个界面是中央服务器统计的支付记录，各种 APP 的成功支付都会被中央服务器记录。

第 2 节 餐厅点菜

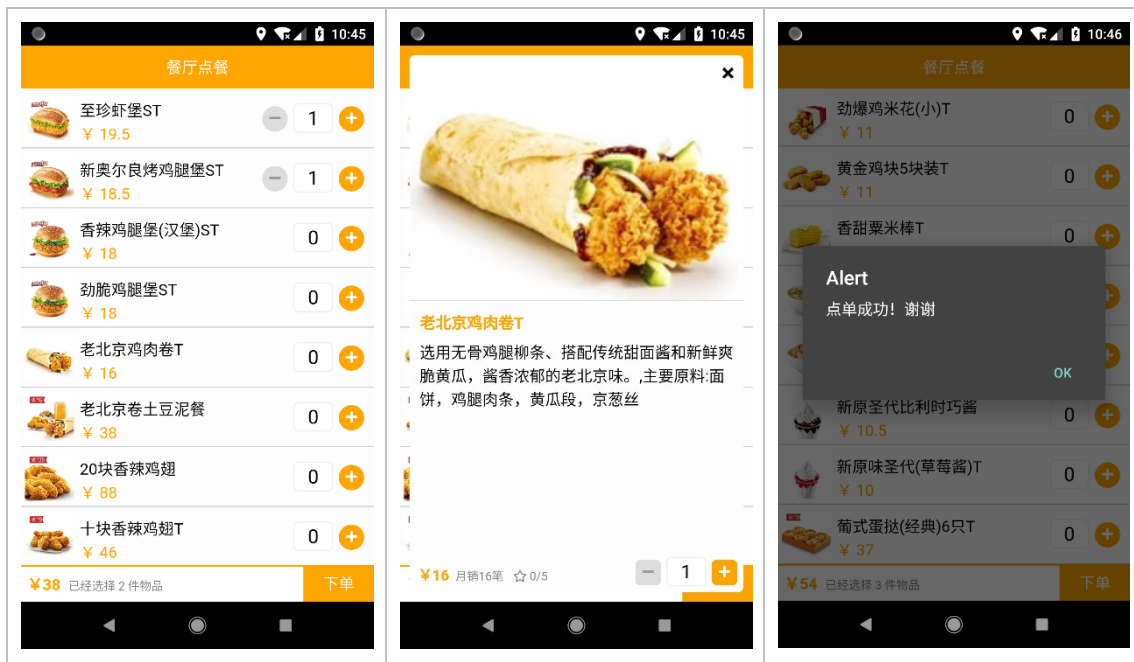


图 9.2 (a) – (c) 餐厅点菜 APP 效果图

这是一个很有商业价值的场景。实现的内容和现在已经存在的扫码点菜比较类似。

用户点击 APP 即可点菜、下单、在线支付。这个 DEMO 在用到的技术上和“停车场缴费”类似。

用户同意支付之后，支付授权签名同样会被发给点菜应用的后端（后端可以通知服务员做菜），后端同样会把它转发给中央服务器，以真正实现收款。

menu_orders @demo-apps (@localhost) - 表					
文件 编辑 查看 窗口 帮助					
开始事务 备注 筛选 排序 导入 导出					
id	user_public_key	amount_paid	content	created_at	updated_at
1	MIIBjANBgkqhkiG9w...	14450	[{"quantity":1,"item_id":"200000209065437889"}, {"quantity":1,"item_id":"200000209075161793"}]	2018-05-14 15:2	2018-05-14 15:24
2	MIIBjANBgkqhkiG9w...	14800	[{"quantity":2,"item_id":"200000209065437889"}]	2018-05-14 15:3	2018-05-14 15:30
3	MIIBjANBgkqhkiG9w...	5400	[{"quantity":1,"item_id":"200000174629666497"}, {"quantity":1,"item_id":"200000174626292417"}]	2018-05-19 02:4	2018-05-19 02:46

图 9.2 (d) 餐厅点菜 APP 数据库

如图，这是餐厅点单的数据库，可记录用户已付款的信息和用户的订单内容。

12	menu (5)	MIIBjANBg...	(1)	74.00	餐厅点餐	共1样菜品	2018-05-14 14:41:12
13	menu (5)	MIIBjANBg...	(1)	144.50	餐厅点餐	共2样菜品	2018-05-14 15:24:52
14	menu (5)	MIIBjANBg...	(1)	148.00	餐厅点餐	共2样菜品	2018-05-14 15:30:02

图 9.2 (e) 中央服务器支付记录

这是中央服务器收到的支付记录，记录里的金额是最终可以结算给商户的。

第 3 节 在线考试



图 9.3 (a) – (e) 在线考试 APP 效果图

这个应用场景可以允许老师给教室里的学生推送随堂测试考卷。学生点击桌面上的 APP 进入考试界面。

第一次进入考试界面后，需要完善个人信息（即姓名、学号），这些信息会被发送到后端，和用户的 `publicKey` 做关联，那么以后每次 API 请求，都可以知道是哪个学生发送的请求。

随后，有一个等待考试开始的环节，考试可以在老师的操控下同时开始，并

设置持续时间。时间到后自动收卷、在线批改统计分数、并记录到数据库。

因为参与考试者的身份是公钥+数字签名，APP 也是根据环境推送的，所以在一定程度上，可以加大代考的技术难度。（若要代考，则必须获得对方的私钥 / 修改客户端代码后请对方人工做数字签名 / 或直接把对方的手机拿到教室）

exam_papers @demo-apps (@localhost) - 表

文件	编辑	查看	窗口	帮助
开始事务	备注	筛选	排序	导入 导出
id	1			
name	古诗词测试			
content_json	[{ "text": "科举制在中国影响深远，乡试录取者称为“举人”，会试录取者称为“贡生”，那么殿试录取者称为()。", "options": ["大元", "解元", "进士", "榜眼"], "key": 3 }			
allowed_time_seconds	300			
is_enabled	1			

图 9.3 (f) 在线考试 APP 试卷数据库设计

这是在线考试后端的试卷数据库，题目和答案使用 JSON 来定义（JSON 被发送给学生时，会被抹去“答案”字段——“答案”只在在线批改时被使用）。`is_enabled` 是个布尔值，代表试卷是否被推送给学生。考试开始时，会从 `false` 被设置为 `true`，客户端那边会轮询，当变为 `true` 时，后端会从返回“等待考试开始”变为返回试卷内容。`allowed_time_seconds` 是考试持续时间（秒），超过这个时间后前端会强制自动交卷（后端也可以设置为：超过这个时间若干秒后，不再接受交卷）。

exam_students @demo-apps (@localhost) - 表

文件 编辑 查看 窗口 帮助

开始事务 备注 筛选 排序 导入 导出

id	public_key	name	student_no	created_at	updated_at
1	MIIBjANBgqh	王轲	14307130048	2018-05-14 08:52:07	2018-05-14 08:52:07

图 9.3 (g) 在线考试 APP 考生数据库设计

这是数据库里存储学生信息的表，公钥和学生信息通过这张表建立关联。

exam_submissions @demo-apps (@localhost) - 表

文件 编辑 查看 窗口 帮助						
开始事务		备注	筛选	排序	导入	导出
id	exam_paper_id	exam_student_id	solutions	score	created_at	updated_at
1	1	1	[4,3,3,1,3]	1	2018-05-14 10:5	2018-05-14 10:51
2	1	1	[2,3,3,2,4]	1	2018-05-14 10:5	2018-05-14 10:54
3	1	1	[2,4,3,3,4]	0	2018-05-14 10:5	2018-05-14 10:54
4	1	1	[3,3,4,1,2]	2	2018-05-14 10:5	2018-05-14 10:55
5	1	1	[2,3,2,1,3]	2	2018-05-14 10:5	2018-05-14 10:56
6	1	1	[2,3,2,1,3]	2	2018-05-14 11:0	2018-05-14 11:00
7	1	2	[3,3,2,3,3]	3	2018-05-19 02:4	2018-05-19 02:42

图 9.3 (h) 在线考试 APP 交卷记录数据库设计

这是考试提交记录的表，可以看到学生的回答和系统自动计算出的得分。

第 4 节 会议时刻表

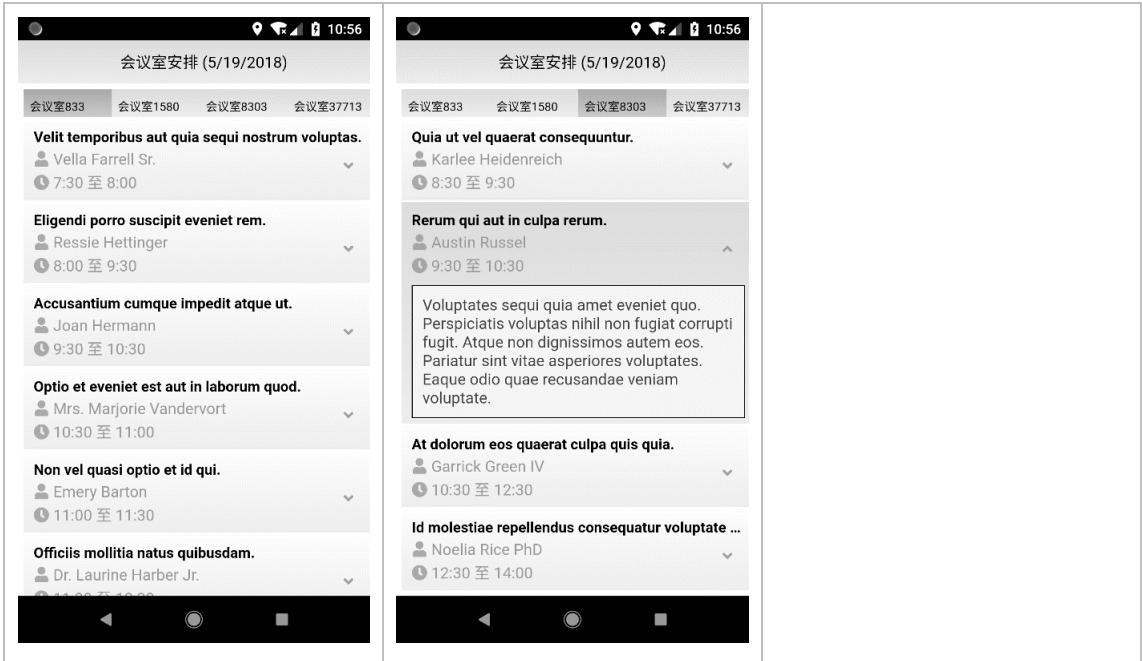


图 9.4 (a) (b) 会议时刻表 APP 效果图

这个 DEMO APP 比较简单，是会议室应用场景的探索。

用户进入会议室之后，手机上可以直接弹出会议安排 APP，点击即可查询每个子会议室的日程安排（讲座时间、讲座内容、主讲人）。

更多的带有和参会人交互的功能也可以被加入本 APP。

第 5 节 活动报名

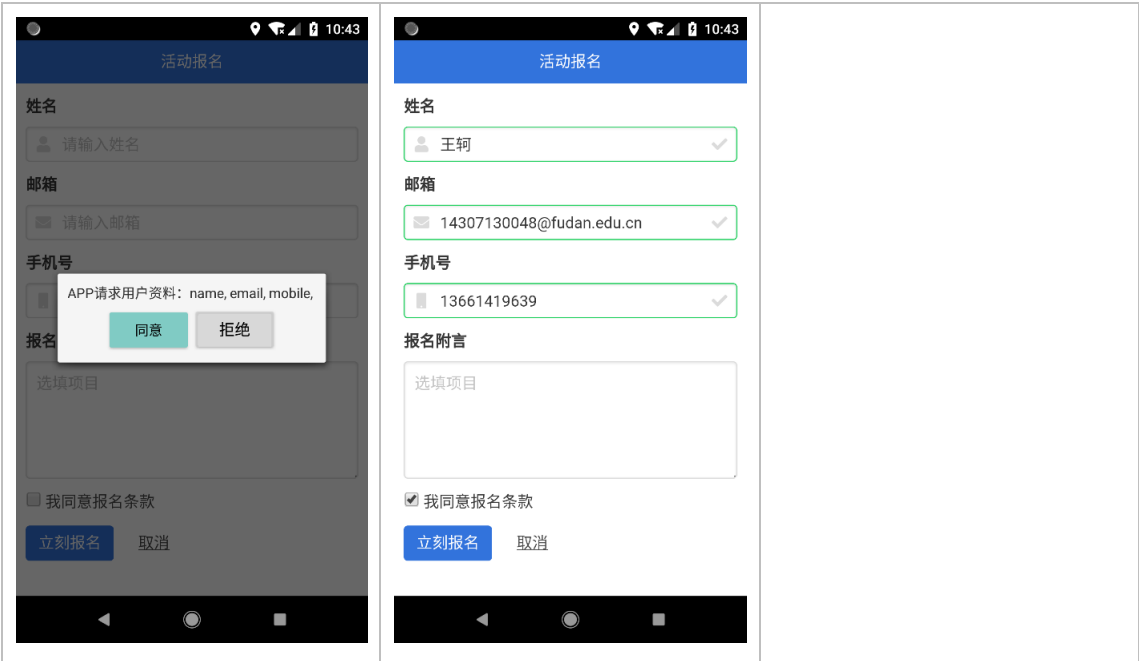


图 9.5 (a) (b) 活动报名 APP 效果图

这也是一个简单的 DEMO APP、用来下发活动报名表。

用户点击 APP 即可报名活动。表单的部分字段使用用户信息请求的机制自动帮用户填写，以省力。

这种场景尤其合适社团摆摊招募。同学希望参加社团活动时，可以直接掏出手机报名活动。

sign_up_records @demo-apps (@localhost) - 表

文件 编辑 查看 窗口 帮助

开始事务 备注 筛选 排序 导入 导出

id	name	mobile	email	comments	extra	created_at	updated_at
1	王轲	1055927733	135692857@abccbc		{"name": "王轲"}	2018-05-14 07:00	2018-05-14 07:00

图 9.5 (c) 活动报名 APP 数据库

报名记录直接被保存进后端数据库。

第 6 节 应用场景的总结、畅想与商业可能

以上是我为了挖掘潜在应用场景，自己制作的 5 个迷你 APP。之前还设想博物馆导览、智能家电控制面板、智能私家车车载系统等等。但时间和硬件方面的因素并未完成。

应用场景初步看来是两方面的，一方面是为商户提供服务（广义的商户也包括会议举办方、社团等），另一方面是为用户本身提供服务（智能家电、远程控制）。

为商户提供服务方面，许多和环境有关的 APP，业务都高度相似。它们有许多共同流程、共同的组件，所以开发者（或开源社区/项目外包公司）可以制作一个高度发达的通用组件库。之后写常见的程序，只要配置组件就行。

这样就可以以极其低廉的价格给每个停车场、商铺定制 APP。客户不花很多钱就得到品牌形象的提升、业务更加便捷；定制方不费很多力就得到了报酬。这种模式有良好的潜在商机。

为用户本身提供服务方面，也有很多深度可以挖掘。比如，当用户进入自己的私家车后，手机自动出现私家车 APP，可以在 APP 里实时监控车的状态、油耗，调节座椅、天窗、空调，甚至导航、播放音乐；回家之后，可以点击自动弹出的智能家电 APP，开灯关灯、开电视机调频道…具备这些智能功能的硬件，能更让用户享受到科技带来的便捷，得到用户青睐。这成就了一种连接一切的可能性，更棒的是，因为所有设备都遵循统一开放的 APP 发现协议，所以一切都是免配置的——进房即用。

为商户提供服务主要会用到基于地理位置的 APP 发现；而为用户本身提供服务主要会用到局域网内 APP 发现。两种发现原理不同，互为补充，都可以照料到。

第十章 完全去中心化可能性的探讨

本毕业设计的架构里，还是用到了中央服务器这一个概念，没有完全去中心化。即还是会有一个机构（类似 ICANN）来管理这一整套系统。如果能做到完全去中心化，那么这套技术架构和协议会更加开放、对所有人更加公平。

目前中央服务器存在的主要意义在于：APP 元数据的管理、根据地理位置发现 APP 所用数据的维护、管理 APP 内支付、抑制诈骗类 APP。可以认为，在目前的这个架构下，只有一份真相（就像 13 台根 DNS 服务器一样），所有的子节点都是从根服务器同步数据（或在 LAN 发现 APP 的情况下，轻微添加一些额外数据）。

如果要去掉中央服务器，需要解决上述所说的几个问题。

APP 内支付问题应该是最容易解决的，现在的模式是中央服务器代扣代收，但完全可以和银行、支付宝等进行合作，添加一种基于数字签名的支付。即：把用户公钥和银行卡/支付宝进行关联，用户签署一笔支付请求后，直接把签名发给银行/支付宝，商户直接收到来自银行的付款通知。

APP 元数据管理、地理位置数据维护，会变得有一些困难，因为有了中央服务器，信息的发布成为了一个问题。但我们可以参考已有的纯 P2P 网络，比如：比特币、freenet、ipfs (Benet, 2014)，用类似的方法解决问题。这些解决方案主要是基于广播，发布数据时传播给自己认识的节点，那些节点再进一步传播给别的它们认识的节点，逐级覆盖全球。理论上这种网络是可以支撑整个方案，代替现在的中心化 APP 仓库服务器的，但是这对每个节点的硬盘容量、网络质量、节点密布数产生了比较大的要求。发布新版本 APP 的速度可能会变慢（新数据可能要很长时间才会被发布到全球各个节点），APP 初次启动时间也会非常长（初次启动时需要通过盲目的搜索，搜到一台节点，才可以用它来进行 APP 发现——或许也可以参考 BT 下载中 PEX 节点交换的技术，即始终跟路由器交换已知节点列表，来加速初次启动的发现过程）。

信用问题（诈骗 APP 防范），看似难解，实际却也有解决思路。可以参考 https 证书（EV）的思路，交给可信第三方来解决。即一个第三方机构（如 VeriSign）调查你的资质（上海 XX 商业广场），调查成功后，用他们的公钥给你的 APP 签名（上海 XX 商业广场地下车库缴费程序）。因为可信第三方的公钥是预先录在系统里的，所以通过这个签名，用户就可以得知：即将打开的 APP 的确是上海 XX 商业广场所颁发的，至少，可信第三方已经帮你调查过 APP 发布者的身份了。

可能最难解决的问题，反而是 APP 膨胀的问题。即什么样的人到最后都想发 APP，然后广告类型的 APP 遍地都是。这个问题需要采用综合手段来解决，一种可以探索的思路是：引入用户评价体系。即用户使用后可以询问该 APP 是什么类

型的（广告/商店/便利服务/咨询），顶还是踩。用户的反馈同样会被签名，分发到各个节点中。节点会在客户端请求发现 APP 的同时把投票的统计数据一并返回给客户端，客户端上可以设置过滤和筛选条件。

但如何保证不法分子不会生成几十万个密钥对，对想要推广的 APP 进行投票，来混淆视听呢？一种方法是网络实名制，即像身份证一样，每个人只能拥有一个公钥私钥对（或者说他的公钥本身就是被签过名的）。

更可行的方法是“自由选择信任对象”，建立起多个非盈利组织，这些组织里的人帮助最终用户甄别是非、维护判断未知 APP 属性的数据。这些组织各有自己的公钥，他们对 APP 进行的投票才是客户端真正关心的投票。用户到底信任哪些组织（哪些公钥），这是可以让用户自己设置的。如果用户发现应用 A 明明是个广告，组织 a 却投票说 A 是个咨询 APP，那么用户可以把对组织 a 的信任给移除，组织 b、组织 c 会给正确的投票。久而久之每个信任组织会有自己的口碑评价，也会珍惜自己的权力。

另一种简单的方法，增加可信第三方认证的收费即可。如果发布一个带签名的 APP 需要投入较高的成本，那么广告类 APP 就不会发得那么猖狂。对于学校等不能支付高昂费用的机构，可以引入“可选择相信的第三方”，即默认不相信学校公钥签发的 APP，但学生可以手动勾选相信。勾选完之后，所有学校公钥签发的 APP 和支付高昂费用后请专业第三方信任机构签发的 APP 有同等地位。

通过以上的分析，我初步认为一个完全去中心化的版本是有可能可以做的。但有许多细节需要进一步探讨、各种协议的制定和实现也将是一个工程上的巨大挑战。

第十一章 总结

本毕业设计探索了一种新的 APP 范式，对它进行了详细的构思，并深度挖掘了技术上可能遇到的各种挑战，创新性地给出了解决方式。此外，本毕业设计还从商业应用的角度对这种新范式可以带来的可能性进行了探索。

在实现层面，本 APP 完成了一个概念验证：构建起了 APP 中央服务器和分布式 APP 仓库服务器网络，实现了数据同步、NAT 内分发；实现了基于地理位置的 APP 发现和基于 LAN 的 APP 发现；实现了构建时依赖库与内核代码拆分、运行时动态拼装以节省流量；实现了一个安卓客户端、以桌面小工具的方式模拟了主动发现后弹出的 APP；实现了用户信息授权和支付流程…

本毕业设计还探讨了完全去中心化的可能性，它涉及到一种分布式信任系统的设计。完全去中心化，是这类系统的最终目标、终极升华，是做开放标准架构的证明。这也是本毕业设计和微信小程序的不同点。

在本毕业设计的最终可能形态里，这种技术可以被植入安卓和 iOS 系统本身，并成为人们使用手机的一种习惯，而世界各个角落的骨干网络、商场 WIFI 里，都有分布式的分发节点（已经被做成了完全去中心化的架构）。它在商业上中性独立，却能给使用者和商户同等地创造价值。

这个目标非常宏大，当然不是我的毕业设计可以企及的，但我非常希望我毕业设计的探索可以启发一些标准制定机构做这件事情，并把人们使用手机 APP 的方式推向下一个阶段。

附录：PoC 版本代码链接

一个可以运行的 PoC 版本已经发布到 GitHub 上。

<https://github.com/environment-based-app-discovery-KW/>

有以下几个 git 仓库：

design

设计相关文档

demo-apps

DEMO APPS，包含停车场缴费、餐厅点单、在线考试、会议日程表、活动报名五个 APP，旨在探索本技术适用的场景、并提供本框架上的前后端开发 DEMO。

app-builder

基于 webpack 的 APP 前端构建器，能够打包出 vendor 分离的代码包。

app-repo-server

APP 仓库服务器，分布式多地部署（可在 NAT 内部署），存储与管理 APP 代码、元数据、部署信息。

android-app

安卓客户端：本地拼装 APP、缓存依赖、提供用户信息获得/鉴权/支付的 JavaScript 接口、桌面小工具快速启动 APP。

app-auth-demo

用户鉴权、获得用户信息、支付的前端 API 的 DEMO。

central-server

中央服务器，用来管理用户数据、上传新 APP（写入 app-repo-server）、管理 APP 内支付。

mirror-tool

app-repo-server 的同步工具，可以用来定时同步最新依赖库和 APP 的代码。

致谢

我于 17 年 6 月就与实验室导师、师兄师姐们聊过本毕业设计的想法，1 年间，陆陆续续进行了许多讨论，想法也一步步变化、迭代和成熟，从最初的畅想到初步的落实想法到网络架构的设计到创新性的代码拆分优化。

由衷地感谢周扬帆副教授、康昱博士，和保障楼 302 的所有同学。感谢大家给我的思路上的启迪、技术上的指导和精神上的鼓励。起初我对 APP 发现的认识简单肤浅，在一次次和大家的探讨中我慢慢发现了可以做文章的点；起初我并不知道如何做学术研究，文字和方法都很不专业，多亏大家倾情传授经验。没有各位的帮助，我并不能完成这样一篇毕业设计，非常感谢！

此外，我也希望向 Apache 软件基金会、webpack 开源项目贡献者和其他开源项目贡献者们表示感谢。站在这些巨人的肩膀上，我才能有这样的技术能力完成我的毕业设计。

最后，也感谢我的家人与朋友们，在我完成本论文过程中，提供了许多支持。

参考文献

- [1] Alakuijala and Kliuchnikov, Evgenii and Szabadka, Zoltan and Vandevenne, Lode Jyrki. (2015). Comparison of brotli, deflate, zopfli, lzma, lzham and bzip2 compression algorithms. Google Inc.
- [2] Alibaba Cloud. (2018 年 5 月 3 日). 智能解析线路. 检索来源: 阿里云: https://help.aliyun.com/document_detail/49060.html
- [3] ATTARDDAVID. (2017 年 2 月 21 日). HTTP/2 - A Real-World Performance Test and Analysis. 检索来源: CSS-TRICKS: <https://css-tricks.com/http2-real-world-performance-test-analysis/>
- [4] BenetJuan. (2014). IPFS-content addressed, versioned, P2P file system. arXiv preprint arXiv:1407.3561.
- [5] CordovaApache. (2013). About Apache Cordova. Apache Software Foundation, accessed.
- [6] Dalmasso and Datta, Soumya Kanti and Bonnet, Christian and Nikaein, Navid Isabelle. (2013). Survey, comparison and evaluation of cross platform mobile application development tools. Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International, 323--328.
- [7] Docker. (2017 年 4 月 9 日). What can I use Docker for. 检索来源: Docker Documentation: <https://docs.docker.com/engine/docker-overview/#what-can-i-use-docker-for>

- [8] FischerMAddison. (1991). Public key/signature cryptosystem with enhanced digital signature certification. US Patent.
- [9] GrigorikIlya. (2018 年 1 月 3 日). Optimizing Encoding and Transfer Size of Text-Based Assets. 检索来源: Google Developers:
https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/optimize-encoding-and-transfer#text_compression_with_gzip
- [10] npm. (2018 年 5 月 16 日). package.json. 检索来源: npm Documentation:
<https://docs.npmjs.com/files/package.json>
- [11] OpenWrt. (2016 年 11 月 02 日). DNS and DHCP configuration. 检索来源: OpenWrt Wiki:
<https://wiki.openwrt.org/doc/uci/dhcp>
- [12] S. CheshireKrochmalM. (2013 年 2 月). DNS-Based Service Discovery. 检索来源: IETF:
<https://www.ietf.org/rfc/rfc6763.txt>
- [13] Usage of JavaScript libraries for websites. (2018 年 5 月 27 日). 检索来源: W3Techs:
https://w3techs.com/technologies/overview/javascript_library/all
- [14] WAGNERJEREMY. (2017 年 4 月 12 日). Brotli and Static Compression. 检索来源: CSS-TRICKS: <https://css-tricks.com/brotli-static-compression/>
- [15] Webpack. (2018 年 5 月 27 日). Configuration. 检索来源: Webpack:
<https://webpack.js.org/configuration/>