携程移动端性能优化

2017-01-03 南志文 CSDN

引言

在《程序员》杂志 10 月刊作者发布了文章《携程移动端架构演进与优化之路》文章第一部分：架构演进，11 月份作者在 CSDN 上发布了《携程移动端 UI 界面性能优化实践》。此文章是在这两篇文章基础上进行总结的关于携程移动端性能优化的其他内容，包括 Hybrid 框架优化、网络优化、通信数据格式传输优化、内存优化、启动时间优化、React Native 优化等内容，接下来作者将从这几个角度进行详细展开。

Hybrid 框架优化

携程 App 的 Hybrid 框架经过多个版本的迭代，支持强大的插件功能，已经做到凡是可用，通通使用 Native 组件来优化 Hybrid 业务的体验。携程 Hybrid 框架在设计之初即采用了离线包功能：Hybrid 业务整体打包在 App 中，节省了用户打开页面时的资源加载时间；同时离线包支持差分增量更新，并通过 7z 压缩方式进一步降低了增量更新包的大小，相对 Zip 压缩减少 30%。

➤离线组件包

传统 HTML5 页面的静态资源常常需要从远程服务器下载，造成移动环境下最昂贵的网络开销。不同于 HTML5，组件化 Hybrid 方案引入了组件包的概念，将业务需要的所有静态资源打包成一个组件包文件，并进行离线化版本管理，而且关于离线包的更新是采取增量差分升级更新的方式。

在用户访问组件页面之前，组件框架通常默默地将最新业务组件包下载至客户端，用户访问组件页面时，WebView 加载的实际上均是本地页面和资源，大幅提升了页面加载性能。不仅如此，关键业务组件在携程客户端版本发布时会预置到 App 中，减少首次下载安装的时间。

➤WebView 预加载

影响页面加载速度的因素非常多，我们在对 WebView 加载一个网页的过程进行调试发现，每次加载的过程中都会有较多的网络请求。除了 Web 页面自身的 URL 请求，还会有 Web 页面外部引用的 JS、CSS、字体、图片等都是个独立的 HTTP 请求。这些请求都是串行的，加上浏览器的解析、渲染时间就会导致 WebView 整体加载时间变长，消耗的流量也对应地增多。所以为了加快 WebView 的整体加载时间，我们使用了预加载策略，将 JS、CSS、字体、图片等资源提前加载出来。Android 通过 WebView 自带的 API 即

setWebViewClient 中的回调接口 ShouldInterceptRequest 去实现资源的预加载。类似事例代码如图 1 所示。

```
public WebResourceResponse shouldInterceptRequest(WebView webView, final String url)
    WebResourceResponse response = null;
    boolean resDown = JSHelper.isURLDownValid(url);
    if (resDown) {
        jsStr = JsjjJSHelper.getResInputStream(url);
        if (url.endsWith(".png")) {
            response = getWebResourceResponse(url, "image/png", ".png");
        } else if (url.endsWith(".gif")) {
            response = getWebResourceResponse(url, "image/gif", ".gif");
        } else if (url.endsWith(".jpg")) {
            response = getWebResourceResponse(url, "image/jepg", ".jpg");
        } else if (url.endsWith(".jepg")) {
            response = getWebResourceResponse(url, "image/jepg", ".jepg");
        } else if (url.endsWith(".js") && jsStr != null) {
            response = getWebResourceResponse("text/javascript", "UTF-8", ".js");
        } else if (url.endsWith(".css") && jsStr != null) {
            response = getWebResourceResponse("text/css", "UTF-8", ".css");
        } else if (url.endsWith(".html") && jsStr != null) {
            response = getWebResourceResponse("text/html", "UTF-8", ".html");
        }
    }
    // 若 response 返回为 null , WebView 会自行请求网络加载资源。
    return response;
}
});
```

图 1 WebView 预加载资源

iOS 通过 setNeedsLayout 机制提前初始化下一个 UIwebView，将 HTML 文件初始化，然后捕获点击事件去动态刷新数据。

网络优化

携程 App 网络请求是基于 TCP 和 HTTP 的混合请求，比如 Hybrid 页面就是通过直接发送 HTTP 请求。由于携程用户环境的复杂情况下 Hybrid 页面 HTTP 失败率在 3%左右，对于携程来说 4 个 9 的高可行性架构要求显然是不符合的，同时在 HTML5 页面使用 HTTP 服务容易发生 DNS 劫持现象。基于这些原因，提高服务的成功率是当时必须要推进而且是重中之重要推进的事情。

我们 Hybrid 网络性能优化通过 Hybrid 接口发送 Native 网络服务的方案替代 HTTP，其一是因为 Native 端的 TCP 长连接可以提高服务成功率，其二从安全角度可以避免 DNS 劫持。此外，针对携程海外用户的特点，我们也进行了海外网络性能优化，主要是通过 TCP 海外加速产品实现链路优化。

由于携程 App 大部分网络服务主要基于 TCP 连接，为了将 DNS 时间降至最低，我们采取了动态 IP 优化策略算法，即内置了 Server IP 列表，该列表可以在 App 启动服务中下发更新。App 启动后的首次网络服务会从 Server IP 列表中取一个 IP 地址进行 TCP 连接，同时 DNS 解析会并行进行，DNS 成功后，会返回最适合用户网络的 Server IP，那么这个 Server IP 会被加入到 Server IP 列表中被优先使用。

此外，Server IP 列表是有权重机制的，DNS 解析返回的 IP 很明显具有最高的权重，每次从 Server IP 列表中取 IP 会取权重最高的 IP。列表中 IP 权重也是动态更新的，根据连接或服务的成功失败来动态调整，这样即使 DNS 解析失败，用户在使用一段时间后也会选取到适合的 Server IP。

除了动态 IP 优化策略，携程还使用 TCP 替换 HTTP 请求服务，原因在于：

- 携程用户有时会在网络环境非常差的景区使用，需要针对弱网进行特别的优化，单纯 HTTP 应用层协议很难实现。
- HTTP 请求首次需要进行 DNS 域名解析，我们发现国内环境下针对携程域名的失败率在 2-3%（包含域名劫持和解析失败的情况），严重影响用户体验。
- HTTP 虽然是基于 TCP 协议实现的应用层协议，优势是封装性好，客户端和服务端解决方案成熟。劣势是可控性小，无法针对网络连接、发送请求和接收响应做定制性的优化，即使是 HTTP 的特性如保持长连接 Keep Alive 或管道 Pipeline 等都会受制于网络环境中的 Proxy 或者服务端实现，很难充分发挥作用。

➤Hybrid 网络服务优化

携程 App 中有相当比例的业务是使用 Hybrid 技术实现的，运行在 WebView 环境中，其中的所有网络服务（HTTP 请求）都是由系统控制的，我们无法掌控，也就无法进行优化，其端到端服务成功率也仅有 97%左右（注：这里指页面中业务逻辑发送的网络服务请求，而非静态资源请求）。我们采用了名为“TCP Tunnel for Hybrid”的技术方案来优化 Hybrid 网络服务，和传统 HTTP 加速产品的方法不同，我们没有采用拦截 HTTP 请求再转发的方式，而是在携程 Hybrid 框架中的网络服务层进行自动切换。具体的架构图如图 2 所示。

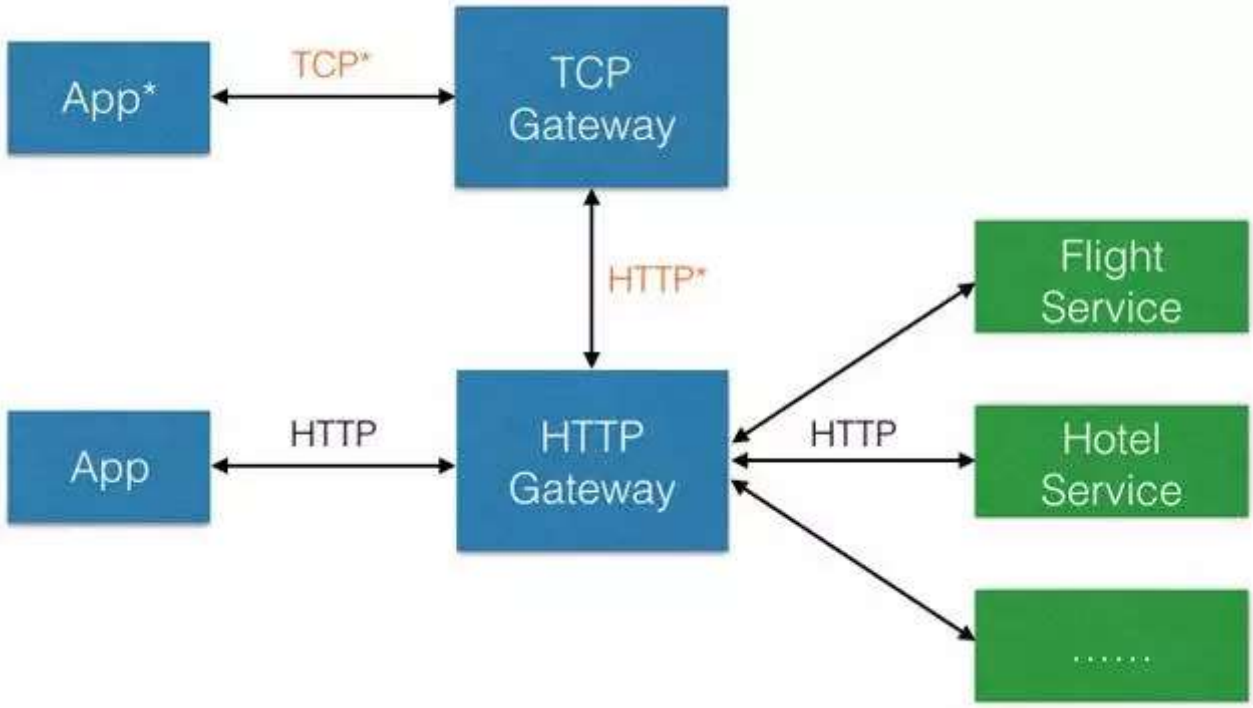


图 2 携程 Hybrid 网络请求转发图

从图 2 可见，该技术方案的流程如下：

如果 App 支持 TCP Tunnel for Hybrid，Hybrid 业务在发网络服务时，会通过 Hybrid 接口转发至 App Native 层的 TCP 网络通讯层。该模块会封装这个 HTTP 请求，作为 TCP 网络服务的 Payload 转发到 TCP Gateway。

TCP Gateway 会根据服务号判断出是 Hybrid 转发服务，解包后将 Payload 直接转发至 HTTP Gateway，此 HTTP 请求对 HTTP Gateway 是透明的，HTTP Gateway 无需区分是 App 直接发来的还是 TCP Gateway 转发来的 HTTP 请求。

后端业务服务处理完成后，HTTP 响应会经 HTTP Gateway 返回给 TCP Gateway，TCP Gateway 将此 HTTP 响应作为 Payload 返回给 App 的 TCP 网络通讯层。 TCP 网络通讯层会再将该 Payload 反序列化后返回给 Hybrid 框架，最终异步回调给 Hybrid 业务调用方。整个过程对于 Hybrid 业务调用方也是透明的，它并不知道 TCP Tunnel 的存在。

采用该技术方案后，携程 App 中 Hybrid 业务的网络服务成功率提升至 99%以上，平均耗时下降 30%，如图 3 所示。

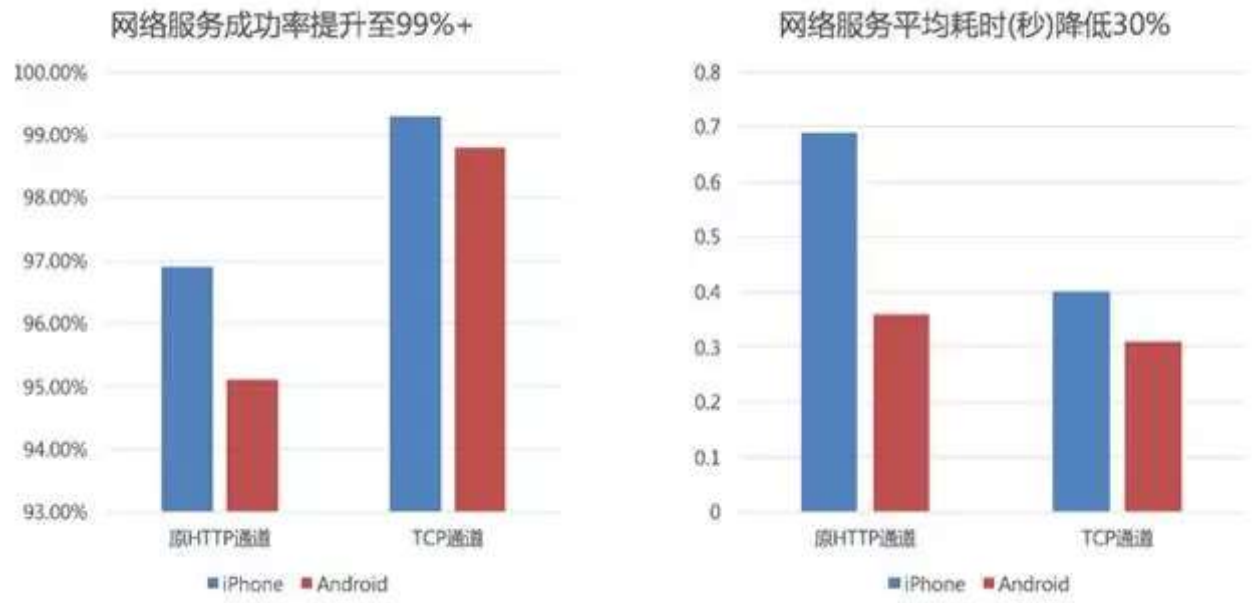


图 3 携程 App TCP 和 HTTP 效果对比图

➤携程 App 海外网络服务优化

携程目前没有部署海外 IDC，海外用户在使用 App 时需要访问位于国内的 IDC，服务平均耗时明显高于国内用户。我们采用了名为“TCP Bypass for Oversea”的技术方案来优化主要是使用了 Akamai 的海外专属网络通道，同时在携程国内 IDC 部署了局端设备，使用专用加速通道的方式来提升海外用户体验，如图 4 所示。

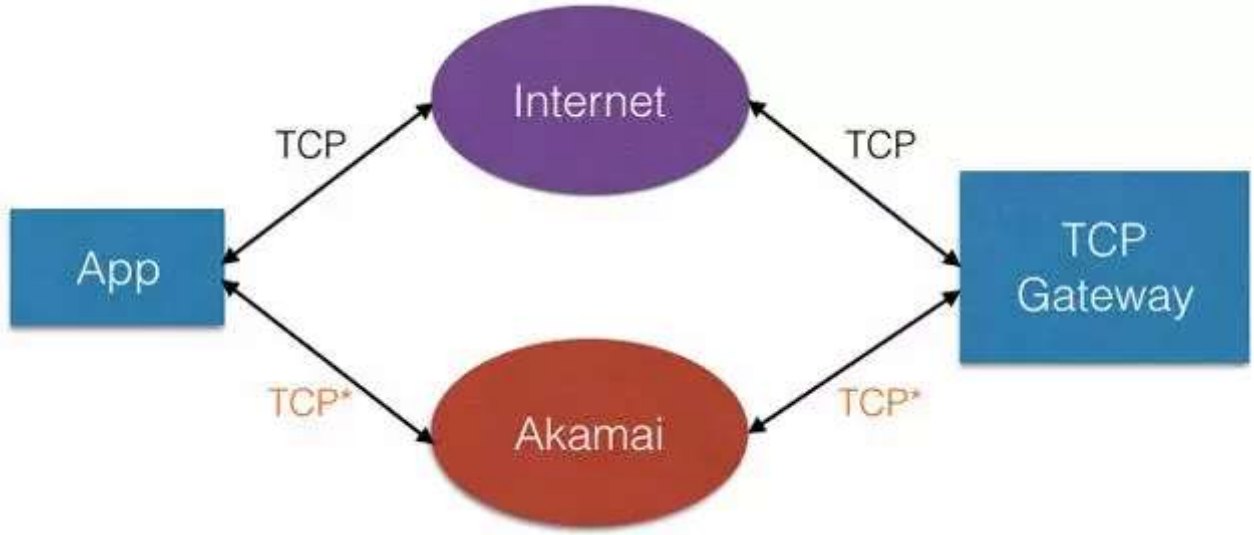


图 4 携程海外网络通道署图

海外用户启动 App 后先通过 Akamai 定制域名获取 Server IP，所有网络服务优先走 Akamai 通道；如果 Akamai 通道的网络服务失败并且重试机制生效时，会改走传统 Internet 通道进行重试。相比只用传统 Internet 通道，在保持网络服务成功率不变的情况下，使用 Akamai 通道 Bypass 技术后平均服务耗时下降了 33%，如图 5 所示。

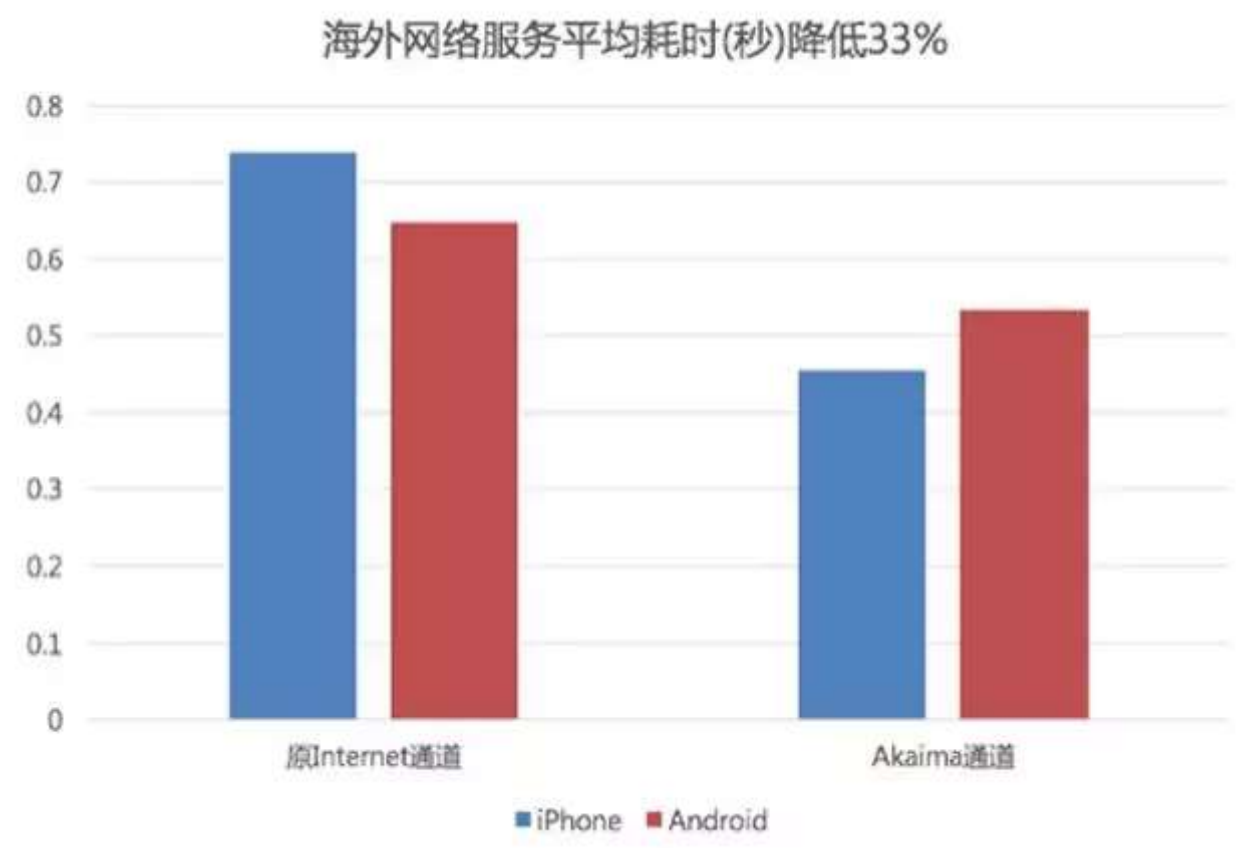


图 5 海外网络服务优化前后对比图

➤携程 App 通信数据格式优化

携程 App 原来使用了自定义的 SOTP 协议格式，底层采用 Socket TCP 协议，为了提高服务响应加载速度，我们优化了 TCP 服务 Payload 数据的格式和序列化/反序列化算法，从自定义 SOTP 格式转换到了 Protocol Buffer 数据格式。此外对数据格式进行了 Gzip 压缩，提升效果非常明显，我们使用 PB + Gzip 后，数据大小下降了 76%，如图 6 和表 1 所示。

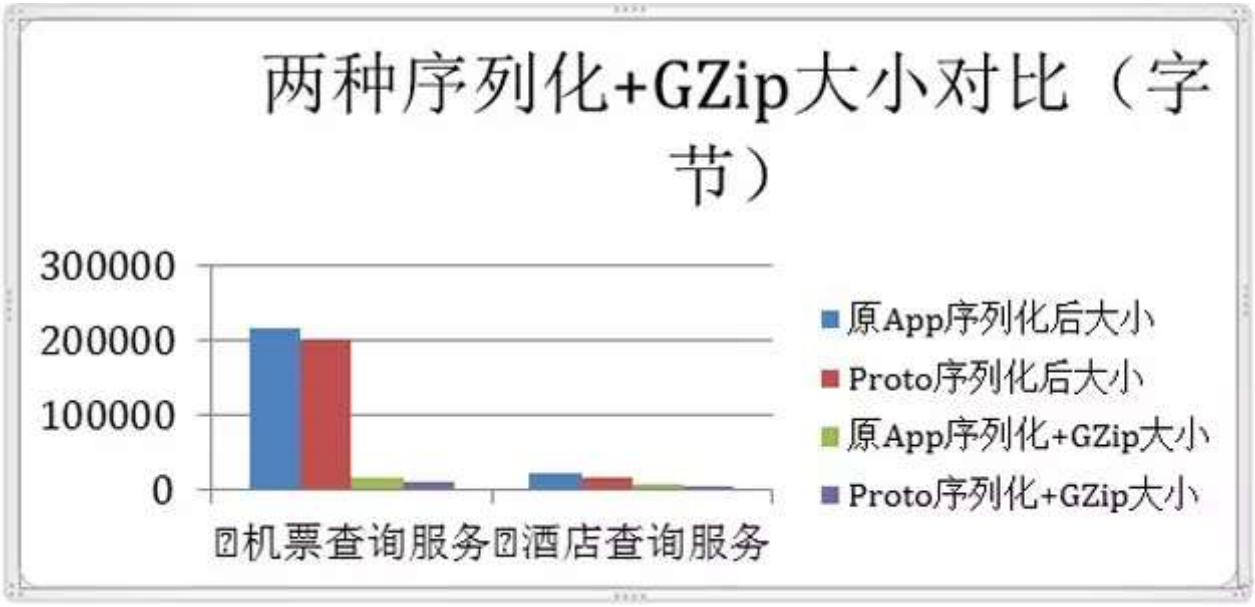


图 6 数据序列化大小对比效果图

	原格式 序列化后大小	PB 序列化后大小	原格式序列化 加Gzip 大小	PB 序列化 加Gzip大小
机票服务	217，169	203，178	16，227	9352
酒店服务	20，582	15，352	6219	4756

表 1 数据序列化大小对比效果

基于 PB+Gzip 的优势，我们目前基本上大部分服务都已经迁移到 PB 上来。另外 Facebook 曾分享过他们使用 FlatBuffer 提高性能的实践，可以参考 Facebook 开源的 Blog。

FlatBuffer 是 Google 为游戏平台开放的一个开源项目，它是 Protocol Buffer 的一种进化方案与实现，具有不需要反序列化就能够获取到任意子元素、保持元数据的优点，不需要打包/解包。它的结构化数据都以二进制形式保存，不需要数据解析过程，数据也可以方便传递。所以在性能上很有优势，但是后续我们分析后不太适合携程的业务场景因而最终没有使用。

➤其他网络相关优化

- 弱网和网络抖动优化

携程 App 引入了网络质量参数，通过网络类型和端到端 Ping 值进行计算，根据不同的网络质量改变网络服务策略：

  - 调整长连接池个数：例如在 2G/2.5G Egde 网络下，会减少长连接池个数为 1（运营商会限制单个目标 IP 的 TCP 连接个数）；Wi-Fi 网络下可以增加长连接池个数等机制。
  - 动态调整 TCP Connection、Write、Read 的超时时间。
  - 不同网络类型状态切换连接机制：例如 Wi-Fi 和移动网络、4G/3G 切换至 2G 时，客户端 IP 地址会发生变化，已经连接上的 TCP Socket 注定已经失效（每个 Socket 对应一个四元组：源 IP、源 Port、目标 IP、目标 Port），此时会自动关闭所有空闲长连接，现有网络服务也会根据状态自动重试。

- 引入重试机制，提升服务成功率

受 TCP 协议重传机制来保证可靠传输的机制启发，在应用层面也引入了重试机制来提高网络服务成功率。我们发现 90%以上的网络服务失败都是由于连接失败，此时再次重试是有机会连接成功并完成服务的；同时我们发现前面提到的网络服务生命周期处于建立连接、序列化网络请求报文、发送网络请求这三个阶段失败时，都可以自动重试，因为我们可以确信请求还没有到达服务端进行处理，不会产生幂等性问题（如果存在幂等性问题，会出现重复订单等情况）。当网络服务需要重试时，会使用短连接进行补偿，而不再使用长连接。

实现了上述机制后，携程 App 网络服务成功率由原先的 95.3%+提升为如今的 99.5%+（这里的服务成功率是指端到端服务成功率，即客户端采集的服务成功数除以请求总量计算的，且不区分当前网络状况），效果显著。

➤其他网络服务机制和技巧

携程 App 也实现了其他一些网络服务机制方便业务开发，如网络服务优先级机制，高优先级服务优先使用长连接，低优先级服务默认使用短连接；网络服务依赖机制，根据依赖关系自动发起或取消网络服务，例如主服务失败时，子服务自动取消。

开发过程中我们也发现一些移动平台上的 TCP Socket 开发技巧：

- iOS 平台上的原生 Socket 接口创建连接并不会激活移动网络，这里原生 Socket 接口是指 POSIX Socket 接口，必须使用 CFSocket 或再上层的网络接口尝试连接时才会激活网络。因此携程 App 启动时会优先激活注册一些第三方 SDK 以及发送 HTTP 请求来激活移动网络。
- 合理设置 Socket 的几个参数：SO_KEEPALIVE 参数确保 TCP 连接保持（注：此 KeepAlive 是 TCP 中的属性，和 HTTP 的 KeepAlive 是两个场景概念），SO_NOSIGPIPE 参数关闭 SIGPIPE 事件，TCP_NODELAY 参数关闭 TCP Nagle 算法的影响。
- 由于 iOS 要求支持 IPv6-Only 网络，因此使用原生 Socket 必须支持 IPv6。
- 如果使用 select 来处理 Non-blocking/IO 操作，确保正确处理不同的返回值和超时参数。
- 保持 TCP 长连接可用性的心跳机制：对于非 IM 类应用而言，心跳机制的作用不大，因为用户会不断触发请求去使用 TCP 连接。尤其在携程业务场景下，通过数据统计发现使用心跳与否对服务耗时和成功率影响极小，因此目前已经关闭心跳机制。原先的心跳机制是 TCP 长连接池中的空闲 TCP 连接每 60 秒发送一个心跳包到 Gateway，Gateway 返回一个心跳响应包，从而让双方确认 TCP 连接有效。

➤关于 SPDY 和 HTTP/2 协议的探讨

过去两年我们的网络服务优化工作都是基于 TCP 协议实现的，基本达到了优化目标。不过这两年来新的应用层网络协议 SPDY 和 HTTP/2 逐步迈入主流，基于 UDP 的 QUIC 协议看起来也非常有趣，值得跟进调研。

SPDY 是 Google 基于 TCP 开发的网络应用层协议，目前已经停止开发，转向支持基于 SPDY 成果设计的 HTTP/2 协议。HTTP/2 协议的核心改进其实就是针对 HTTP/1.x 中影响延迟性能的痛点进行优化：

- Header 压缩：压缩冗余的 HTTP 请求和响应 Header。
- 支持 Multiplexing：在 HTTP/1.1 协议中，浏览器客户端在同一时间，针对同一域名下的请求有一定数量限制，超过限制数目的请求会被阻塞，而 HTTP 2.0 就是为了解决这个限制问题，采取多路复用技术，即支持一个 TCP 连接上同时实现多个请求和响应。
- 二进制分帧提高传输性能：在不改动 HTTP/1.x 的语义、方法、状态码、URI 以及首部字段等因素的情况下，应用层（HTTP/2）和传输层（TCP or UDP）之间增加一个二进制分帧层去提高传输效率。
- 保持长连接（比 HTTP/1.x 更彻底）：减少网络连接时间。
- 支持服务端推送：由服务端主动推送数据到客户端，同时可以缓存，也让在遵循同源的情况下，不同页面之间共享缓存资源成为可能。

官方性能测试结果显示使用 SPDY 或 HTTP/2 的页面加载时间减少 30% 左右，不过这是针对网页的测试结果。对于 App 中的网络服务，具体优化效果我们还在进行内部测试，不过其优化手段和目前我们使用 TCP 协议的相类似，因此性能优化效果可能不会很显著。

不过 HTTP2.0 正处于逐步应用到线上产品和服务的阶段，可以预见未来会有不少新的坑产生和与之对应的优化技巧，HTTP1.x 和 SPDY 也将在一段时间内继续发挥余热。作为工程师，需要了解这些协议背后的技术细节，才能打造高性能的网络框架，从而提升我们的产品体验。

携程图片相关优化

在 Android 系统中，当我们使用资源 ID 来去引用一张图片时，Android 会使用一些规则来帮我们匹配最适合的图片。什么叫最适合的图片？比如我的手机屏幕密度是 xxhdpi，那么 drawable-xxhdpi 文件夹下的图片就是最适合的。因此，当我引用这张图时，如果 drawable-xxhdpi 文件夹下有这张图就会优先被使用，在这种情况下，图片是不会被缩放的。系统就会自动去其它文件夹下找这张图，优先会去更高密度的文件夹下寻找。

我们当前的场景就是 drawable-xxhdpi 文件夹，然后发现这里也没有这张图，接下来会尝试再找更高密度的文件夹，发现没有更高密度的了，就会去 drawable-nodpi 文件夹找，发现也没有，那么就会去更低密度的文件夹下面找，依次是 drawable-xhdpi→drawable-hdpi→drawable-mdpi→ drawable→ldpi。

总体匹配规则就是如上所示，那么比如说我手机是 1080P，但是资源图片放在 drawable-xhdpi 文件夹下面，系统首先去 xxhpi 文件下去寻找发现没有找到，然后跑到更高密度夹 xxxhdpi 还是没找到，最后发现低密度文件夹 xhdpi 下有资源，于是系统自动帮我们做了这样一个 3/2 放大操作。同样的道理，如果系统是在 drawable-xxxhdpi 文件夹下面找到这张图的话，它会认为这张图是为更高密度的设备所设计的，如果直接将这张图在当前设备上使用就有可能会出现像素过高的情况，于是会自动帮我们做一个缩小的操作。

目前市场上主流机型是 1080P，所以美工切图以 1080P 为基准切一套图，直接将这一套图放在 xxhdpi 文件夹下面即可，即如果你的机子是主流机型 1080P 分辨率，则图片资源不会做任何缩放，但如果你的机子是 720P（xhdpi），则图片资源会做 2/3 倍的扩大，其实际占用内存会变小，同时资源不会被放大变形；同理如果是 480P（hdpi），则图片资源会做 2/4 倍的扩大，即图片大小缩放为原来的 1/2，实际占用内存同上所示。

将资源图片放到正确的位置，能帮 App 去优化节省内存，提升用户体验，所以在开发过程中，资源图片不要任意放置。

携程做了一件事情，就是将图片资源位置进行了正确整理，如原来放错位置的 xhdpi/hdpi/mdpi 资源全部迁移到 xxhpi 文件夹下面，并且删除了原来的多套图资源，只是在特殊的界面适配比如 Pad 适配，才会特殊处理。这样做的结果是直接减少内存 20%，Apk Size 减少了 20% 左右。

此外，为了减少 App Size，除了删除多套图资源和无用的资源，我们在图片优化上，还使用了 SVG 矢量图、WebP 格式的图片替换原始的 PNG。

图片是携程 App 使用场景中较多的元素之一，如何快速节省流量的下载和渲染图片是我们非常关注的性能问题。除了上面提到的网络优化长连接，携程还实现了 App 的动态切图技术，动态下发 WebP 格式图片：主要是基于分辨率、质量、锐化、格式四个纬度，对同一张图片生成了不同组合的文件，设置了一系列匹配规则，针对

不同屏幕、机型处理能力、网络环境，配置出适合当前情况的图片大小质量，保证图片大小既节省又确保用户视觉体验。其图片加载流程如图 7 所示，最终携程的优化效果：图片下载大小减少 58%，下载耗时降低 50%。

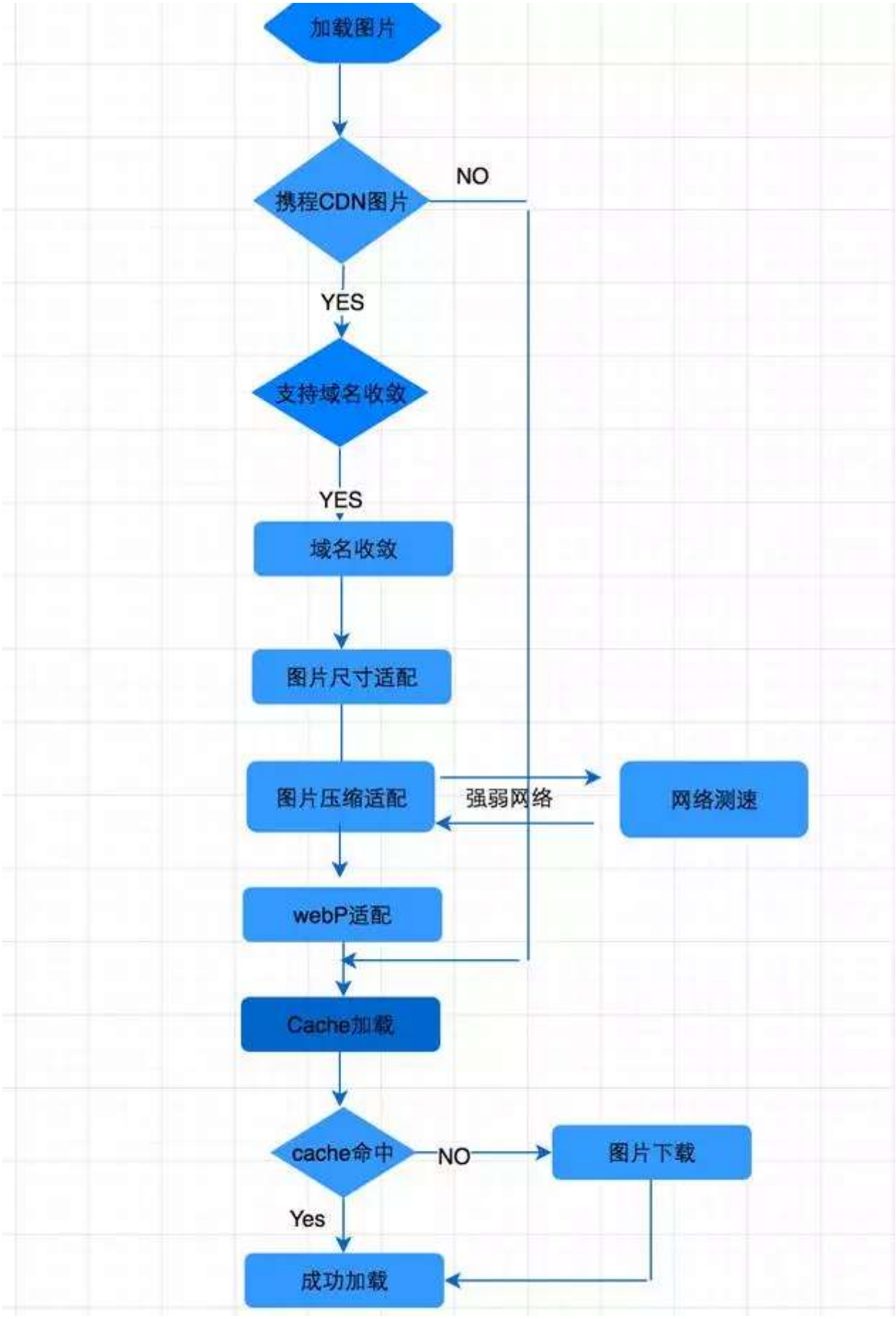
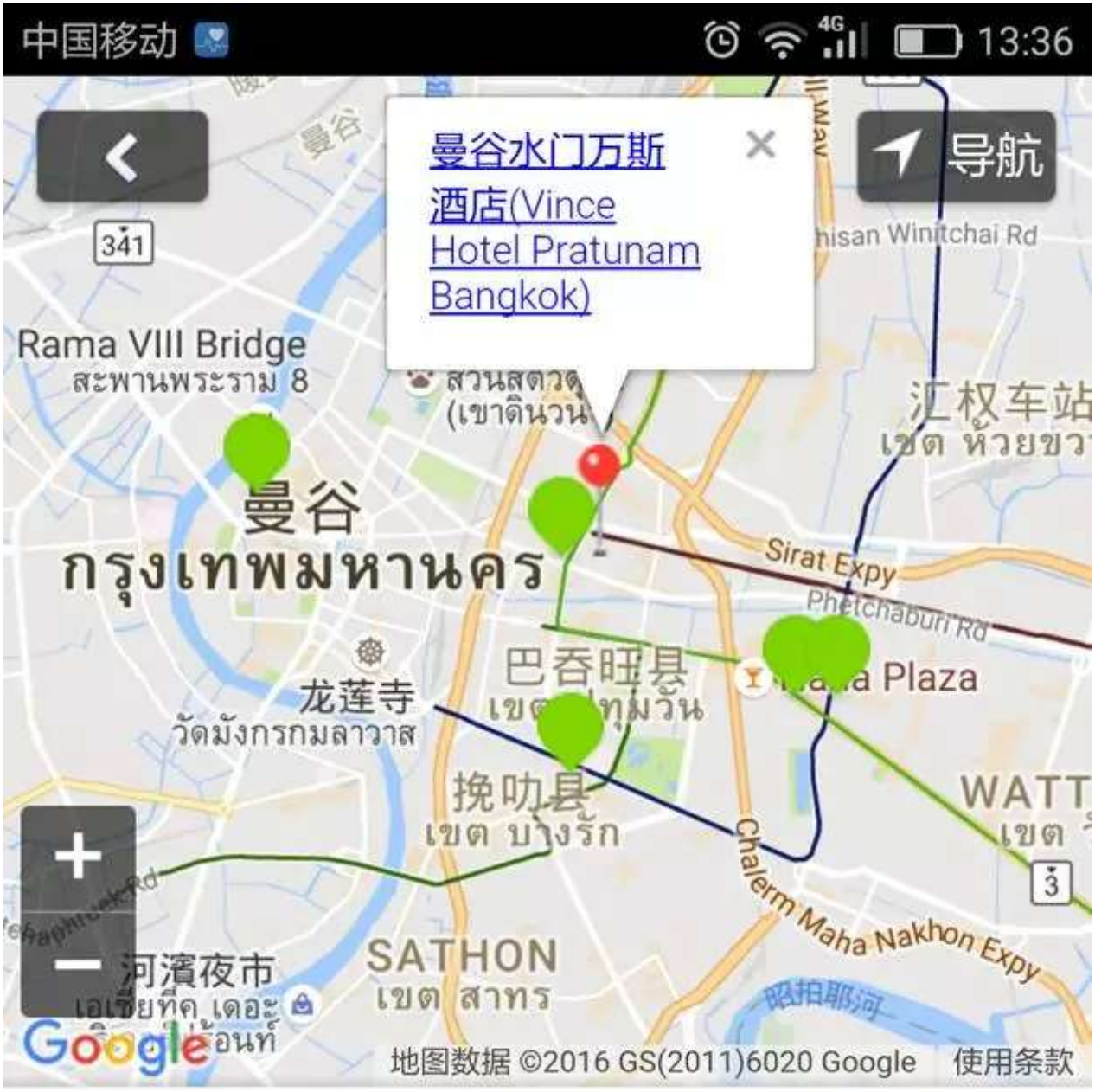


图 7 携程图片加载流程图

在图 7 中，图片加载流程有个域名收敛机制，为什么需要域名收敛？首先它是相对于域名发散技术的，是因为浏览器对同一域名做了最大连接数限制，所以为了让浏览器并发加载而做发散，缺点是域名过多会引起 DNS 解析耗时。域名收敛就是为了解决 DNS 解析耗时问题，以加快图片下载速度从而提升用户体验。

携程地图优化

携程国内地图最早接入的是高德地图，后来才接入百度地图，但是它们海外支持较差，很多城市没有数据支持，iOS 用户相对还好解决，因为苹果在国内的合作厂商是高德，所以当时 iOS App 无论国内外都是直接使用苹果自带的地图。但是 Android App 海外地图相对没有好的解决方案，本来 Android 海外地图直接集成 Google 地图 Native SDK 即可。但是因为国内用户无法访问 Google 的服务，所以这种方案直接就舍弃，后来就直接采用 Google 海外版的 JS 方案，基于 Google JS 地图 API，开发了 Hybrid 版的 Google 地图，最终效果图如图 8 和图 9 所示。



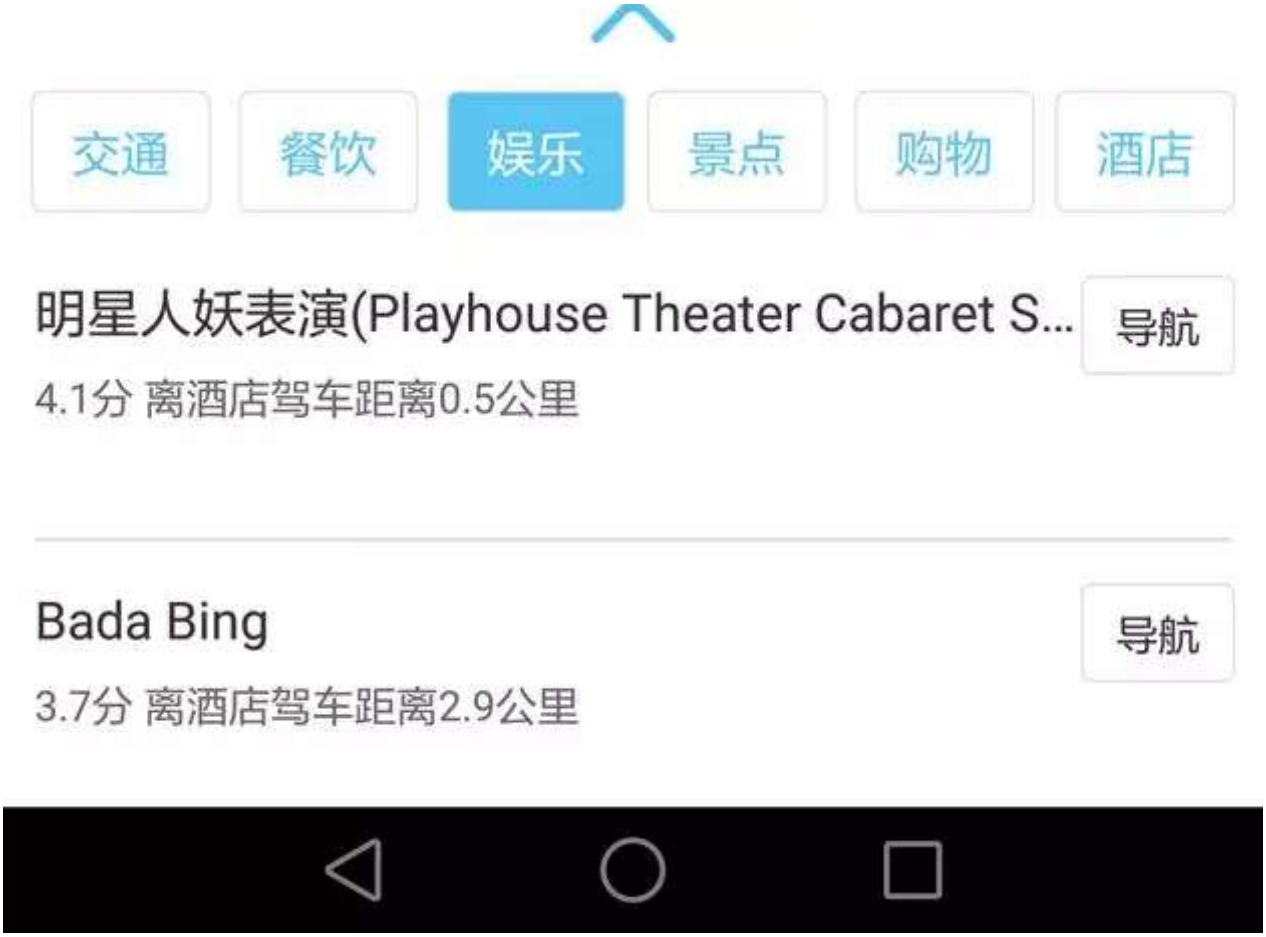

交通
餐饮
娱乐
景点
购物
酒店
明星人妖表演(Playhouse Theater Cabaret S...
导航
4.1分 离酒店驾车距离0.5公里
Bada Bing
导航
3.7分 离酒店驾车距离2.9公里

图 8 携程海外详情地图


中国移动
13:12
共2282家酒店，展示前20家。
MILITARY
หมู่บ้าน
บ้านพัก
นายทหาร
แลนาสิบ
มหาวิทยาลัย
หอการค้าไทย
Mit Maitri Rd
THANA THIP
VILLAGE
หมู่บ้าน
ธนาทิพย์
DIN DAENG
เขต ดินแดง
Yu Charoen
Central Plaza
Grand Rama 9
Black Canyon เซ็นทรัล
พลาซา พระราม 9

图 9 携程海外列表地图

Google JS 地图的域名服务因为大陆不稳定，所以携程对于 JS 域名服务做了一个简单优化，在国内的优先使用香港域名服务，如果访问不通，再使用海外域名服务；在海外的用户则优先使用 Google 海外域名服务，保证了海外地图稳定的服务以及良好的用户体验。此外地图的其他方面优化，比如数据做分段加载，以及数据的缓存机制，WebView 单独 WebCore 的优化，提前预加载数据和 JS 资源等优化。

携程 App 启动内存优化

这个章节主要是针对携程 App Android 平台上做的插件内存优化。携程目前有多个不同的事业部，主要由酒店、机票、火车票、汽车票、旅游等组成，每个事业部都独立开发。其开发的 Bundle 就是一个独立插件，开发过程中不相互依赖，Android 最终集成阶段会单独生产一个 APK，目前携程 App 由 30 多个 Bundle 业务插件组成。

启动 App 时，为了优化内存，不直接将所有的 Bundle 插件而是将每个事业部的核心 Bundle dex 文件加载进来，其他插件实现 Lazy-loading 即按需加载机制。为了实现 Lazy-loading 加载速度，直接将 Bundle 优化到

更加地细粒度（其加载时间尽量不超过 500ms），这样直接加快 App 启动速度，并节省 App 内存，其优化的效果如图 10 所示。

图 10 Android App 安装启动时间对比效果

经过优化后，App 启动时间提升明显。关于内存优化，携程 App 还从图片占用内存、轻量级数据结构、内存泄露等角度进行了优化。

携程 React Native 优化

携程 App 从 12 年开始开发，至今已有 4 年多时间，随着各项业务功能的全面移动化，以及公司“Mobile First”策略的指引下，App 功能愈加丰富和复杂，从而造成 App iOS Size 达到将近 100MB。而同样功能，使用 RN 开发，Size 远远小于 Native 开发，RN 的引入，可以支持我们 App 的可持续健康发展。此外由于 RN 通过 Java Core 解析 Java 模块转换成原生 Native 组件渲染，相比 H5 页面不再局限于 WebView、渲染性能长足提升，运行用户体验比 H5 提升明显可以媲美 Native，且 RN 支持跨平台和动态更新。

基于上述原因，携程从 16 年 4 月份开始，在 App 内部开始小范围使用 React Native 技术，主要是基于 RN 0.28 定制 CRN，App V6.17 版本中实现站内信、机票低价订阅，之后开始较大规模地在各个 BU 事业部推广，有 15 个业务模块在使用，涉及到的页面在 50 个以上。

ListView 是我们大量使用的控件，所以需要重点优化，原生控件超出屏幕的条目，依然被渲染了。没有实现 cell 重用，导致数据量大时会出现卡顿现象。为了提高用户体验，具体优化思路是：

我们基于原生控件开发了可重用 cell 的 CRNListView，iOS 借鉴了第三方的 ReactNativeTableView 的实现，开发了可重用 cell 的 ListView，接口和官方原生的基本一致。Android 借鉴 iOS 的方案，采用 RecyclerView

实现了类似的可重用 cell 的 ListView，同时我们还做了一些扩展，把常用的下拉刷新，载入更多，右侧字母索引栏等功能，都增加了进去。

实际测试下来，当数据量少时，和 RN 官方提供的 ListView 性能基本一致，但数据量大时，CRNListView 优势明显，如图 11 是我们在 Android 上的测试数据。

图 11 前后对比效果图

RN 页面加载最大的瓶颈在 JS Init + Require，这块就是 JSBundle 的执行时间。为了提升页面加载速度，要想办法优化。基于此原因，我们对 RN 官方的打包脚本做改造，将框架代码拆分出来，让所有业务使用一份框架代码，即拆分成 Common.js 和 Business.js。

一般 RN 框架部分文件（Common.js）大小比较大，占用了绝大部分的 JS 执行时间。如果这部分执行加载时间能放到后台预先做完，进入业务也只需执行业务页面的几个 JS 文件，将可以大大提升页面加载速度。

按照这个思路，能后台加载的 JS 文件, 实际上就是一个 RN App。因此我们设计了一个空白页面的 Fake App，用来监听要显示的真实业务 JS 模块，收到监听后，渲染业务模块最终显示页面。

携程 App 其他角度优化

从减少频繁 I/O 角度，及自身业务出发，去除若干初始化阶段不必要的文件操作，以及将若干非实时性要求的文件操作延后处理。

Android 上对于频繁读写数据库 SharedPreference 以及文件的模块，通过增加缓存和降低采样率等手段减少对 I/O 的读写。对于 SharedPreference 进行了专门的优化，减少单个文件的大小，将毫无联系的存储键值分开到不同文件中，并且防止将大数据块存储到 SharedPreference 中，这样既不利于性能也不利于内存，因为 SharedPreference 会有额外的一份缓存长期存在。

关于多线程治理：我们分析了各个模块的线程数量，检查线程池的合理性。通过去掉不必要的线程和线程池，再控制线程池的并发数和优先级。进一步通过框架层的线程池来接管业务方的线程使用，以减少线程太多的问题。

检测超时方法，优化主线程：在早期版本性能优化前，初始化代码都在主线程中执行，为了优化性能，我们已经将部分比较耗时的初始化任务放入后台线程或者异步执行。但是随着携程业务的不断发展和人员变更，还是出现了在主线程中执行很重的初始化任务。

Android 平台通过开启 StrictMode 去检测优化，StrictMode 主要检测两大问题，一个是线程策略，即 TreadPolicy，另一个是 VM 策略，即 VmPolicy。

ThreadPolicy 线程策略检测的内容有：

- 自定义的耗时调用使用 detectCustomSlowCalls() 开启。
- 磁盘读取操作使用 detectDiskReads() 开启。
- 磁盘写入操作使用 detectDiskWrites() 开启。
- 网络操作使用 detectNetwork() 开启。

VmPolicy 虚拟机策略检测的内容有：

- Activity 泄露使用 detectActivityLeaks() 开启。
- 未关闭的 Closable 对象泄露使用 detectLeaked ClosableObjects() 开启。
- 泄露的 SQLite 对象使用 detectLeakedSqlLiteObjects() 开启。
- 检测实例数量使用 setClassInstanceLimit() 开启。

开启 StrictMode 之后，我们将日志输入到 SD card 面，在开发 Debug 阶段导出来可以通过日志分析找出对应的问题和原因。

iOS 平台我们实现了一套应用运行时方法耗时检测机制，能够对应用中所有类的方法调用做耗时统计。方便地找到超时的方法调用后，就可以有针对性地做出修改，或删除或异步化。这种方法调用耗时检测机制同样适用于 App 运行过程中，从而找到导致应用卡顿的根本原因，最后做出对应修改。

携程 App 性能衡量考察

衡量一个 App 性能和质量，首先需要了解 App 性能的现状，即 App 端性能的采集。常规的技术方案有两种：

- 使用第三方性能采集 SDK，例如 OneAPM、听云等；
- 自主研发：携程为了完整掌握用户数据采用了自己研发的方式：App 通过日志 SDK 采集日志包括行为埋点和性能埋点，上传至日志服务端，日志消息经 Kafka 消息队列存入 HDFS（RCFile 格式）分布式文件存储系统，后期的数据查询均基于 Hive 系统来实现。

App 端的性能数据会在多种纬度（操作系统、App 版本、网络状况、位置）下采集网络（网络服务成功率、平均耗时、耗时分布等）、定位（经纬度成功率、城市定位成功率等，见图 12）、启动时间、内存流量等各类指标。其中像网络服务性能是对于用户体验至关重要的端到端性能，也是优化的核心依据。

图 12 携程城市定位成功率

性能数据采集后需要采用简单直观的 Portal 进行展示，携程为无线业务开发了 Web 端和 App 端性能展示 Portal，图 13 和图 14 是网络性能监控的截屏，数据会每小时进行更新聚合并展示。

图 13 网络服务性能监控工具展示界面

图 14 网络服务成功率展示界面

➤携程 App 其他相关

基于 3.x 架构的基础上，携程插件 Bundle 自动化打包平台也应运而生，各 BU 只需要关注自己 BU 的 Bundle 即可，同时开发直接将最新的代码集成到 Bundle 中，打一个最新 Bundle，然后测试也可以自己随时打包，携程无线为我们开发了 MCD 打包平台，可以让整个打包流程自动化，同时这个平台还提供了 HotFix 发布平台、App 应用性能管理、App Crash 数据采集等工具，同时可以实现 Daily/Hourly Build，无缝集成自动化测试，集成了 Sonar 代码扫描去检测重复代码等和 infer null pointer，然后能生成一个代码健康度扫描报告，并能帮助分析和解决问题。

携程 App Size 优化相关

- 通过脚本或第三方工具检测删除无用的资源、类、函数；
- 通过 Sonar 检测整个 App 中的重复代码，将重复代码合并或 App 删除；
- 清理第三方引入的库，删除不用的多套库，比如百度和高德地图 SDK 等，多套图片库等；
- 不轻易引进第三方的库，比如注解框架等，慎重使用第三方定义的控件，自己自定义控件去实现；
- 关于资源图片统一经过 tinpng 压缩；

- 资源图片统一整理，功能和资源统一，不重复造轮子，删除原来的多套资源；
- 资源图片使用 SVG 或者 WebP 格式图片去替换；
- 清理高清资源大图片，特别是超过 10K 以上的图片；
- 能用代码去实现的 UI，尽量不用去图片代替；
- Native 转 H5；
- Native 或者 H5 转 React Native；
- Hybrid 页面离线的 JS 资源，直接转换成 RN，减少包大小比较明显。

总结

随着移动端技术的不断成熟发展，以及各公司业务的成熟稳定发展，App 性能优化成为各大公司重点关注的问题，目的就是为了提升用户的使用体验。性能优化是一个持续发展的实践课题，可以持续贯穿于我们日常的开发工作中，即随着手机机型的日益碎片化，程序功能的复杂化多样化，总之移动端技术的性能优化是没有尽头的，我们会继续持续关注移动端的性能优化，并融入新的技术进行迭代更新。

移动端技术发展很快，携程也正在积极尝试新技术，例如 React Native（已在线上很多模块的信息页面使用，如机票和酒店的部分页面），同时携程基于 RN 推出了自己的 RN 框架 Moles，并且做了不少相关 RN 实践性能优化。同时关于新的网络协议 SPDY、HTTP/2.0，Apple/Huawei/Samsung Watch App 等都做了大量尝试，以期能够提升产品品质。在过去几年时间里，我们已经实现了不少新技术与业务的融合，在这简单列举几点：

- 推出了基于 React Native 的 Moles 框架，并且做了很多 RN 相关优化；
- Bundle 差分升级更新；
- 基于 MVP 和 AOP 的框架设计；
- 率先支持 Apple/Huawei/Samsung Watch App；
- 基于 FreelLine 和 LayoutCast 的热部署方案；
- 支持 SPDY，HTTP2.0 的探讨实现。

未来我们会不断推进新技术的研究并且融合到业务需求中，希望有更多优质内容分享给行业同行。

从之前持续不断的性能优化中，我们虽然积累了不少优化经验，但是在 Android 平台部分，较低端低配置机型上携程 App 性能问题依然不容乐观，所以接下来会继续努力通过更多更细致的优化方案来提升用户体验。

未来我们基于之前积累的历史优化经验会形成一套性能优化的经验闭环，由观察问题现象到分析原因、建立监控，定下量化目标，执行优化方案，验证结果数据再回到观察新问题。每一次闭环只能解决部分问题，不积硅步无以至千里，不积小溪无以成江海，只有不断抓住细微的优化点持续“啃”下去，才能得到螺旋上升的良好结果。

文章最后，感谢陈浩然、赵辛贵之前提供给我的相关框架资料。

作者简介： 南志文，时任携程研发经理，曾负责携程 App 整体技术框架的架构研发与实践，酒店业务的迭代更新及 App 架构、性能优化，先后就职于阿里巴巴、巨人网络等。
责任编辑： 唐小引，技术之路，共同进步，欢迎技术投稿、约稿、给文章纠错，请发送邮件至 tangxy@csdn.net。

阅读原文