美团点评技术团队

WebView性能、体验分析与优化

育新 徐宏 嘉洁 · 2017-06-09 20:03

在App开发中，内嵌WebView始终占有着一席之地。它能以较低的成本实现Android、iOS和Web的复用，也可以冠冕堂皇的突破苹果对热更新的封锁。

然而便利性的同时，WebView的性能体验却备受质疑，导致很多客户端中需要动态更新等页面时不得不采用其他方案。

以发展的眼光来看，功能的动态加载以及三端的融合将会是大趋势。那么如何克服WebView固有的问题呢？我们将从性能、内存消耗、体验、安全几个维度，来系统的分析客户端默认WebView的问题，以及对应的优化方案。

性能

对于WebView的性能，给人最直观的莫过于：打开速度比native慢。

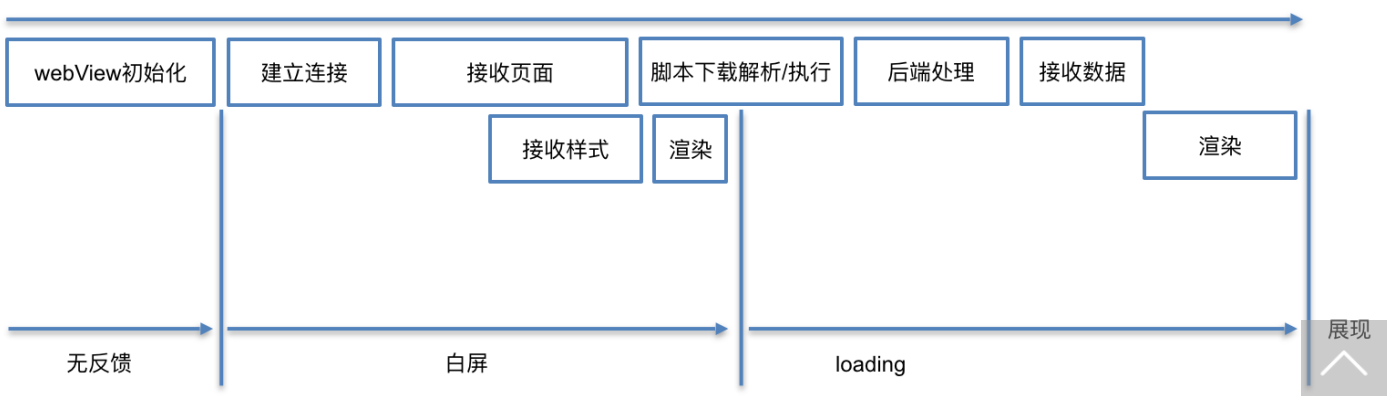
是的，当我们打开一个WebView页面，页面往往会慢吞吞的loading很久，若干秒后才出现你所需要看到的页面。

这是为什么呢？

对于一个普通用户来讲，打开一个WebView通常会经历以下几个阶段：

1. 交互无反馈
2. 到达新的页面，页面白屏
3. 页面基本框架出现，但是没有数据；页面处于loading状态
4. 出现所需的数据

如果从程序上观察，WebView启动过程大概分为以下几个阶段：



如何缩短这些过程的时间，就成了优化WebView性能的关键。

接下来我们逐一分析各个阶段的耗时情况，以及需要注意的优化点。

WebView初始化

当App首次打开时，默认是并不初始化浏览器内核的；只有当创建WebView实例的时候，才会创建WebView的基础框架。

所以与浏览器不同，**App中打开WebView的第一步并不是建立连接，而是启动浏览器内核。**

我们来分析一下这段耗时到底需要多久。

分析

针对WebView的初始化时间，我们可以定义两个指标：

- 首次初始化时间：客户端冷启动后，第一次打开WebView，从开始创建WebView到开始建立网络连接之间的时间。
- 二次初始化时间：在打开过WebView后，退出WebView，再重新打开WebView，从开始创建WebView到开始建立网络连接之间的时间。

测试数据：

> 测试系统1： iOS模拟器，Titans 10.0.7
>
> 测试系统2： OPPO R829T Android 4.2.2
>
> 测试方式：测试10次取平均值
>
> 测试App：美团外卖
>
> 单位：ms

	首次初始化时间	二次初始化时间
iOS（UIWebView）	306.56	76.43
iOS（WKWebView）	763.26	457.25
Android	192.79 *	142.53

> *Android外卖客户端启动后会在后台开启WebView进程，故并不是完全新建WebView时间。

这意味着什么呢？

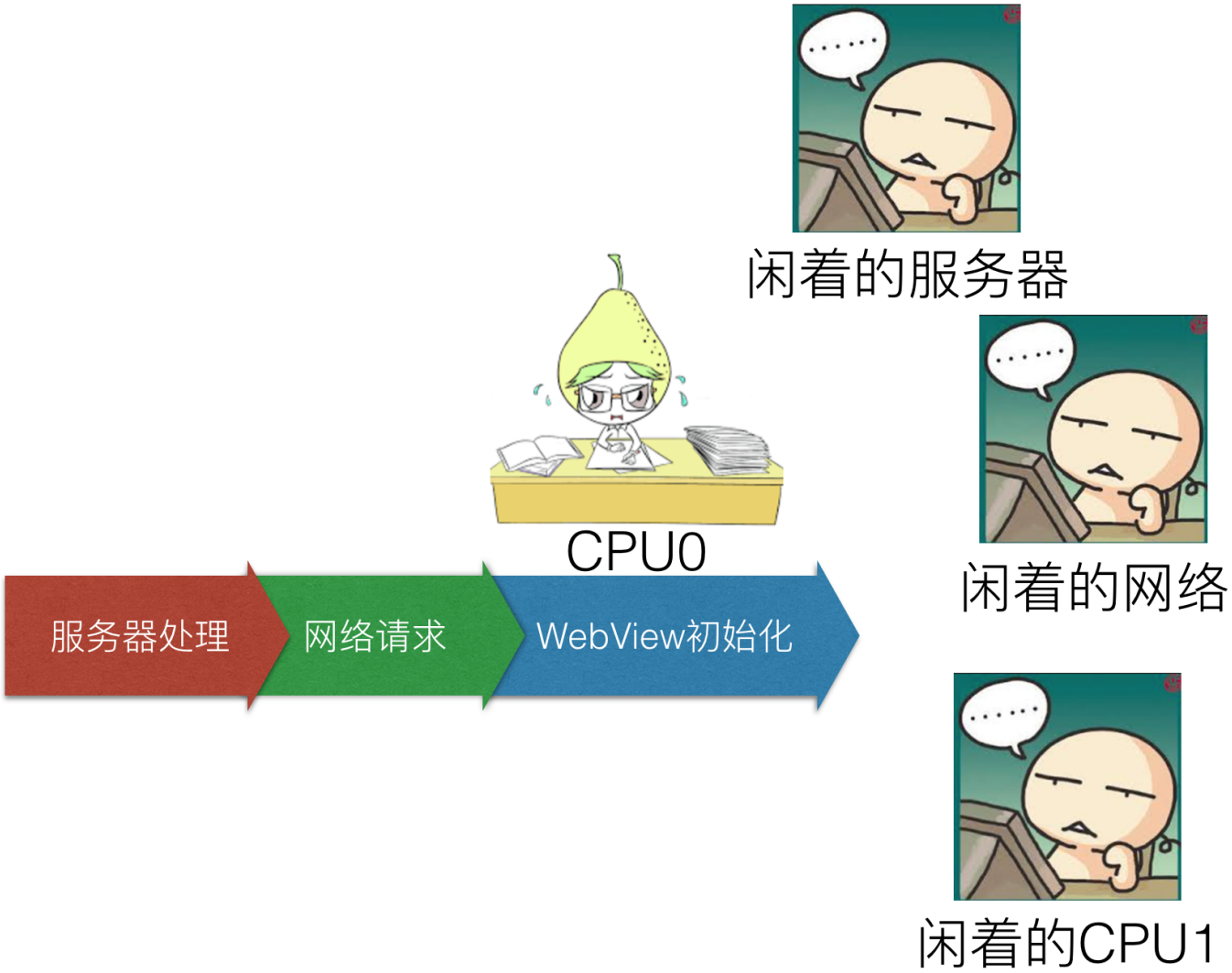
作为前端工程师，统计了无数次的页面打开时间，都是以网络连接开始作为起点的。

很遗憾的通知您：WebView中用户体验到的打开时间需要再增加70~700ms。

于是我们找到了“为什么WebView总是很慢”的原因之一：

- 在浏览器中，我们输入地址时（甚至在之前），浏览器就可以开始加载页面。
- 而在客户端中，客户端需要先花费时间初始化WebView完成后，才开始加载。

而这段时间，由于WebView还不存在，所有后续的过程是完全阻塞的。可以这样形容WebView初始化过程：



那么有哪些解决办法呢？

怎么优化

由于这段过程发生在native的代码中，单纯靠前端代码是无法优化的；大部分的方案都是前端和客户端协作完成，以下是几个业界采用过的方案。

全局WebView

方法：

- 在客户端刚启动时，就初始化一个全局的WebView待用，并隐藏；
- 当用户访问了WebView时，直接使用这个WebView加载对应网页，并展示。

这种方法可以比较有效的减少WebView在App中的首次打开时间。当用户访问页面时，不需要初始化WebView的时间。

当然这也带来了一些问题，包括：

- 额外的内存消耗。
- 页面间跳转需要清空上一个页面的痕迹，更容易内存泄露。

> 【参考东软专利 - 加载网页的方法及装置 CN106250434A (https://www.google.com/patents/CN106250434A?cl=zh)】

客户端代理数据请求

方法：

- 在客户端初始化WebView的同时，直接由native开始网络请求数据；
- 当页面初始化完成后，向native获取其代理请求的数据。

此方法虽然不能减小WebView初始化时间，但数据请求和WebView初始化可以并行进行，总体的页面加载时间就缩短了；缩短总体的页面加载时间：

> 【参考腾讯分享：70%以上业务由H5开发，手机QQ Hybrid 的架构如何优化演进? (http://mp.weixin.qq.com/s/evzDnTsHrAr2b9jcevwBzA)】

还有其他各种优化的方式，不再一一列举，总结起来都是围绕两点：

1. 在使用前预先初始化好WebView，从而减小耗时。
2. 在初始化的同时，通过Native来完成一些网络请求等过程，使得WebView初始化不是完全的阻塞后续过程。

建立连接/服务器处理

在页面请求的数据返回之前，主要有以下过程耗费时间。

- DNS
- connection
- 服务器处理

分析

以下为美团中活动页面的链接时间统计：

> 统计： 美团的活动页面
>
> 内容值： n%分位值（ms）

内容值：11%分位值（ms）

	DNS	connection	获取首字节
50%	1.3	71	172
90%	60	360	541

优化

这些时间都是发生在网页加载之前，但这并不意味着无法优化，有以下几种方法。

DNS采用和客户端API相同的域名

DNS会在系统级别进行缓存，对于WebView的地址，如果使用的域名与native的API相同，则可以直接使用缓存的DNS而不用再发起请求图片。

以美团为例，美团的客户端请求域名主要位于api.meituan.com，然而内嵌的WebView主要位于i.meituan.com。

当我们初次打开App时：

- 客户端首次打开都会请求api.meituan.com，其DNS将会被系统缓存。
- 然而当打开WebView的时候，由于请求了不同的域名，需要重新获取i.meituan.com的IP。

根据上面的统计，至少10%的用户打开WebView时耗费了60ms在DNS上面，如果WebView的域名与App的API域名统一，则**可以让WebView的DNS时间全部达到1.3ms的量级**。

静态资源同理，最好与客户端的资源域名保持一致。

同步渲染采用chunk编码

同步渲染时如果后端请求时间过长，可以考虑采用chunk编码，将数据放在最后，并优先将静态内容flush。对于传统的后端渲染页面，往往都是使用的【浏览器】--> 【Web API】 --> 【业务 API】的加载模式，其中后端时间就指的是Web API的处理时间了。在这里Web API一般有两个作用：

1. 确定静态资源的版本。
2. 根据用户的请求，去业务API获取数据。

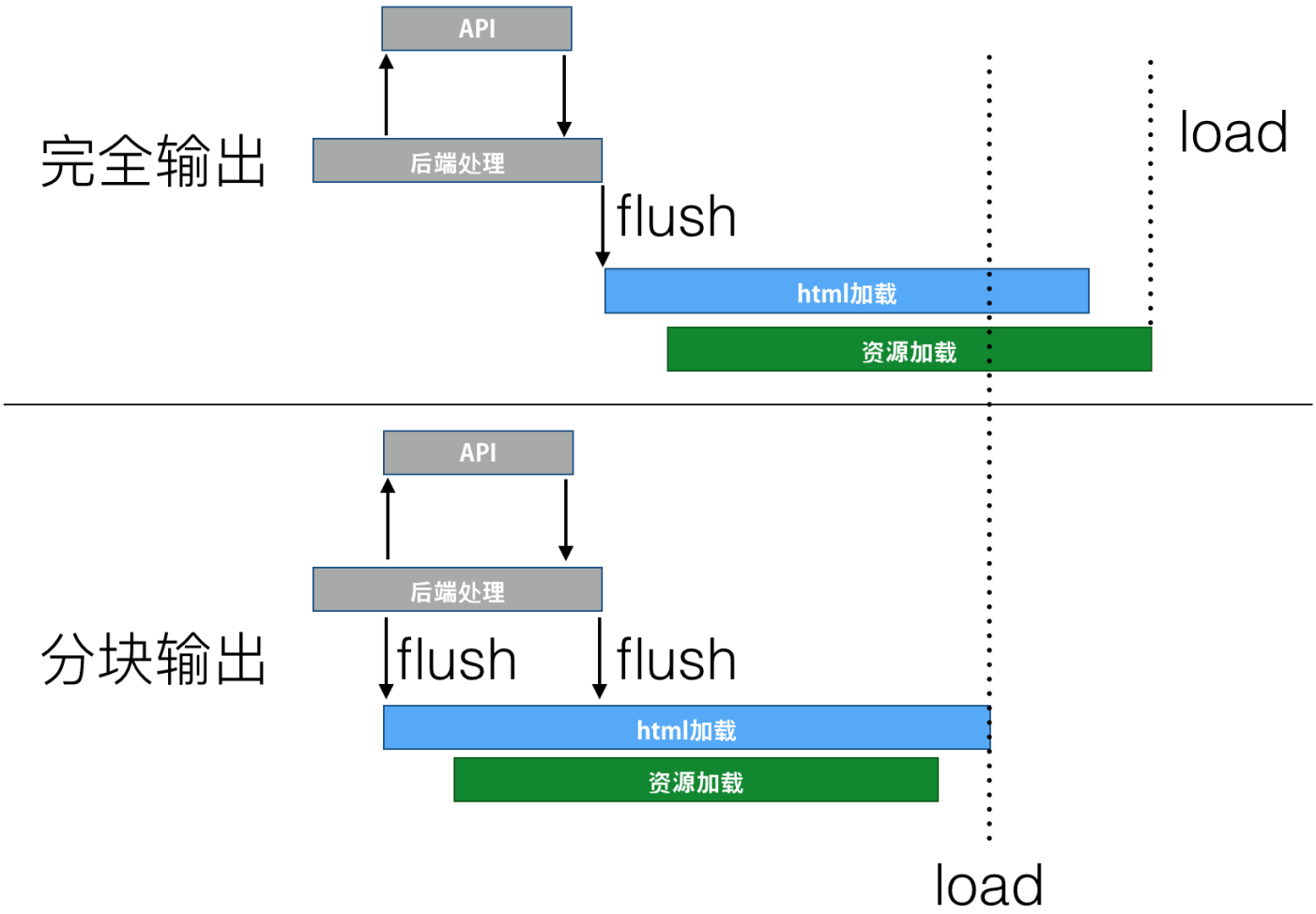
而一般确定静态资源的版本往往是直接读取代码版本，基本无耗时；而主要的后端时间都花费在了业务API请求上面。

那么怎么优化利用这段时间呢？

在HTTP协议中，我们可以在header中设置 `transfer-encoding:chunked` 使得页面可以分块输出。如果合理设计页面，让head部分都是确定的静态资源版本相关内容，而body部分是业务数据相关内容，那么我们可以在用户请求的时候，首先将Web API可以确定的部分先输出给浏览器，然后等API完全获取后，再将API数据传输给浏览器。

下图可以直观的看出分chunk输出和一起输出的区别：

下图可以直观的看出分chunk输出和一起输出的区别：



- 如果采用普通方式输出页面，则页面会在服务器请求完所有API并处理完成后开始传输。浏览器要在后端所有API都加载完成后才能开始解析。
- 如果采用chunk-encoding: chunked，并优先将页面的静态部分输出；然后处理API请求，并最终返回页面，可以让后端的API请求和前端的资源加载同时进行。
- 两者的总共后端时间并没有区别，但是可以提升首字节速度，从而让前端加载资源和后端加载API不互相阻塞。

页面框架渲染

页面在解析到足够多的节点，且所有CSS都加载完成后进行首屏渲染。在此之前，页面保持白屏；在页面完全下载并解析完成之前，页面处于不完整展示状态。

分析

我们以一个美团的活动页面作为样例：

> 测试页面：http://i.meituan.com/firework/meituanxianshifengqiang (http://i.meituan.com/firework/meituanxianshifengqiang)
>
> 在Mac上面，模拟4G情况
>
> 页面样式：

亲子游乐 | 亲子摄影 | 幼儿教育 | 更多优惠

“亲子游乐”

玛尔比恩婴幼儿游泳馆

[价值396元]单人双次婴儿游泳套餐

¥90 减10

¥80抢

鱼乐贝贝婴幼儿游泳馆...

[价值168元]单人婴幼儿游泳单次体验

¥58 减8

¥50抢

鱼乐贝贝婴幼儿水育馆

测试得到的时间耗费如下：

表1

	阶段	时间	大小	备注
DOM下载	58ms	29.5 KB	4G网络	
DOM解析	12.5ms	198 KB	根据估算，在手机上慢2~5倍不等	
CSS请求+下载	58ms	11.7 KB	4G网络（包含链接时间，CDN）	
CSS解析	2.89ms	54.1 KB	根据估算，在手机上慢2~5倍不等	
渲染	23ms	1361节点	根据估算，在手机上慢2~5倍不等	
绘制	4.1ms		根据估算，在手机上慢2~5倍不等	
合成	0.23ms		GPU处理	

同时，对HTML的加载时间进行分析，可以得到如下时间点。

表2

	指标	时间	计算方法
HTML加载完成时间	218	performance.timing.responseEnd - performance.timing.fetchStart	
HTML解析完成时间	330	performance.timing.domInteractive - performance.timing.fetchStart	

这意味着什么呢？

对于表1

可以看到，随着在网络优良的情况下，Dom的解析所占耗时比例还是不算低的，对于低端机器更甚。Layout时间也是首屏前耗时的大头，据猜测这与页面使用了rem作为单位有关（待进一步分析）。

对于表2，我们可以发现一个问题

一般来说HTML在开始接收到返回数据的时候就开始解析HTML并构建DOM树。如果没有JS（JavaScript）阻塞的话一般会相继完成。然而，在这里时间相差了90ms……也就是说，解析被阻塞了。

进一步分析可以发现，页面的header部分有这样的代码：

```
.....
<link href="//ms0.meituan.net/css/eve.9d9eee71.css" rel="stylesheet" onload="MT.pageData.ev
<script>
window.fk = function (callback) {
require(['util/native/risk.js'], function (risk) {
    risk.getFk(callback);
});
}
</script>
</head>
....
```

通常情况下，上面代码的link部分和script部分如果单独出现，都不会阻塞页面的解析：

- CSS不会阻止页面继续向下继续。
- 内联的JS很快执行完成，然后继续解析文档。

然而，当这两部分同时出现的时候，问题就来了。

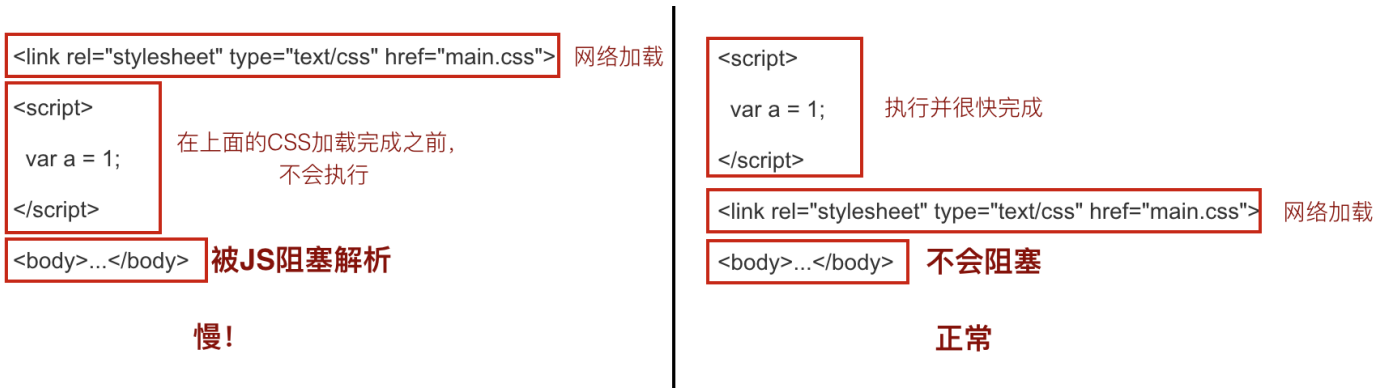
- CSS加载阻塞了下面的一段内联JS的执行，而被阻塞的内联JS则阻塞了HTML的解析。

通常情况下，CSS不会阻塞HTML的解析，但如果CSS后面有JS，则会阻塞JS的执行直到CSS加载完成（即便JS是内联的脚本），从而间接阻塞HTML的解析。

优化

在页面框架加载这一部分，能够优化的点参照雅虎14条 (https://stevesouders.com/hpws/rules.php)就够了；但注意不要犯错，一个小小的内联JS放错位置也会让性能下降很多。

1. CSS的加载会在HTML解析到CSS的标签时开始，所以CSS的标签要尽量靠前。
2. 但是，CSS链接下面不能有任何的JS标签（包括很简单的内联JS），否则会阻塞HTML的解析。
3. 如果必须要在头部增加内联脚本，一定要放在CSS标签之前。



JS加载

对于大型的网站来说，在此我们先提出几个问题：

- 将全部JS代码打成一个包，造成首次执行代码过大怎么办？

- 将JS以细粒度打包，造成请求过多怎么办?

- 将JS按 "基础库" + "页面代码" 分别打包，要怎么界定什么是基础代码，什么是页面代码；不同页面用的基础代码不一致怎么办?
- 单一文件的少量代码改的是否会导致缓存失效?
- 代码模块间有动态依赖，怎样合并请求。

关于这些问题的解决方案数量可能会比问题还多，而它们也各有优劣。

具体分析太过复杂，鉴于篇幅原因在这里不做具体分析了。您可以期待我们的后续计划：BPM（浏览器包管理）。

JS解析、编译、执行

在PC互联网时代，人们似乎都快忘记了JS的解析和执行还需要消耗时间。确实，在几年前网速还在用kb衡量的时代里，JS的解析时间在整个页面的打开时间里只能算是九牛一毛。

然而，随着网速越来越快，而CPU的速度反而没有提升（从PC到手机），JS的时间开销就成为问题了。那么JS的编译和解析，在当今的页面上要消耗多少时间呢?

分析

我们用以下方式来检验JS代码的解析/编译和执行时间：

```
<script>
    window.t1 = performance.now()
</script>
<script>
    window.test = function () {
        // test code
    }
</script>
<script>
    window.t2 = performance.now()
    test();
    window.t3 = performance.now();

    alert("编译耗时：" + (t2 - t1));
    alert("执行耗时：" + (t3 - t2));
</script>
```

将测试代码放入 【test code】 位置，然后在手机中执行；

- 在t1~t2期间，JS代码仅仅声明了一个函数，主要时间会集中在解析和编译过程；
- 在t2~t3时间段内，执行test时时间主要为代码的执行时间

在首次启动客户端后，打开WebView的测试页面，我们可以得到如下的结果：

> 测试系统： iPhone6 iOS 10.2.1
>
> 测试系统： OPPO R829T Android 4.2.2
>
> 内容值： 编译时间（ms）/执行时间（ms）

系统	Zepto.js	Vue.js	React.js + ReactDOM.js
iOS	5.2 / 8	12.8 / 16.1	13.7 / 43.3
Android	13 / 40	43 / 127	26 / 353

当保持客户端进行不关闭情况下，关闭WebView并重新访问测试页面，再次测试得到如下结果：

系统	Zepto.js	Vue.js	React.js + ReactDom.js
iOS	0.9 / 1.9	5 / 7.4	3.5 / 23
Android	5 / 9	17 / 12	25 / 60

执行时间指的是框架代码加载的页面的初始化时间，没有任何业务的调用。

这意味着什么

经过测试可以得出以下结论：

- 偏重的框架，例如React，仅仅初始化的时间就会达到50ms ~ 350ms，这在对性能敏感的业务中时比较不利的。
- 在App的启动周期内，统一域名下的代码会被缓存编辑和初始化结果，重复调用性能较好。

所以，在移动浏览器上，JS的解析和执行时间并不是不可忽略的。

在低端安卓机上，（框架的初始化+异步数据请求+业务代码执行）会远高于几KB网络请求时间；高性能的Web网站需要仔细斟酌前端渲染带来的性能问题。

优化

- 高性能要求页面还是需要后端渲染。
- React还是太重了，面向用户写系统需要谨慎考虑。
- JS代码的编译和执行会有缓存，同App中网页尽量统一框架。

WebView性能优化总结

一个加载网页的过程中，native、网络、后端处理、CPU都会参与，各自都有必要的工作和依赖关系；让他们相互并行处理而不是相互阻塞才可以让网页加载更快：

- WebView初始化慢，可以在初始化同时先请求数据，让后端和网络不要闲着。
- 后端处理慢，可以让服务器分trunk输出，在后端计算的同时前端也加载网络静态资源。
- 脚本执行慢，就让脚本在最后运行，不阻塞页面解析。

- 同时，合理的预加载、预缓存可以让加载速度的瓶颈更小。

- WebView初始化慢，就随时初始化好一个WebView待用。
- DNS和链接慢，想办法复用客户端使用的域名和链接。
- 脚本执行慢，可以把框架代码拆分出来，在请求页面之前就执行好。

WebView内存消耗

分析

为了测试WebView会消耗多少内存，我们设计了如下的测试方案：

1. 客户端启动后，记录消耗的内存。
2. 打开空页面，记录内存的上涨。
3. 退出。
4. 打开空页面，记录内存上涨。
5. 退出。
6. 打开加载了代码的页面，记录内存的额外增加。

得到如下测试结果：

> 测试系统： iOS模拟器，Titans 10.0.7
>
> 测试系统： OPPO R829T Android 4.2.2
>
> 测试方式：测试10次取平均值

	首次打开增加内存	二次打开增加内存	加载KNB+VUE+灵犀
iOS UIWebView	31.1M	5.52M	2M
iOS WKWebView	1.95M	1.6M	2M
Android	32.2M	6.62M	1.7M

WKWebView的内存消耗相比其他低了一个数量级，在此方面相当占优。

UIWebView和Android的WebView在首次初始化时都要消耗大量内存，之后每次新建WebView会额外增加一些。

UIWebView的内存占用不会在关闭WebView时主动回收，每次新开WebView都会消耗额外内存。

相比于性能，对于内存的优化可以做的还是比较有限的。

- WKWebView的内存占用优势比较大（代价是初始化比较慢）。
- 页面内代码消耗的内存相比与WebView系统的内存消耗相比可以说是很低。

WebView体验

除了打开的速度，WebView通常体验也没有native的实现更好，我们可以找到以下几个例子：

除了打开的速度，WebView通常体验也没有native的实现更好，我们可以找到以下几个例子：

长按选择

在WebView中，长按文字会使得WebView默认开始选择文字；长按链接会弹出提示是否在新页面打开。

解决方法：可以通过给body增加CSS来禁止这些默认规则。

点击延迟

在WebView中，click通常会有大约300ms的延迟（同时包括链接的点击，表单的提交，控件的交互等任何用户点击行为）。

唯一的例外是设置的meta：viewpoint为禁止缩放的Chrome（然而并不是Android默认的浏览器）。

解决方法：使用fastclick一般可以解决这个问题。

页面滑动期间不渲染/执行

在很多需求中会有一些吸顶的元素，例如导航条，购买按钮等；当页面滚动超出元素高度后，元素吸附在屏幕顶部。

这个功能在PC和native中都能够实现，然而在WebView中却成了难题：

```
在页面滚动期间，Scroll Event不触发
```

不仅如此，WebView在滚动期间还有各种限定：

- setTimeout和setInterval不触发。
- GIF动画不播放。
- 很多回调会延迟到页面停止滚动之后。
- background-position: fixed不支持。
- 这些限制让WebView在滚动期间很难有较好的体验。

这些限制大部分是不可突破的，但至少对于吸顶功能还是可以做一些支持：

解决方法：

- 在iOS上，使用position: sticky可以做到元素吸顶。
- 在Android上，监听touchmove事件可以在滑动期间做元素的position切换（惯性运动期间就无效了）。

crash

通常WebView并不能直接接触到底层的API，因此比较稳定；但仍然有使用不当造成整个App崩溃的情况。

目前发现的案例包括：

- 使用过大的图片（2M）
- 不正常使用WebGL

WebView安全

WebView被运营商劫持、注入问题

由于WebView加载的页面代码是从服务器动态获取的，这些代码将会很容易被中间环节所窃取或者修改，其中最主要的问题出自地方运营商（浙江尤其明显）和一些WiFi。

我们监测到的问题包括：

- 无视通信规则强制缓存页面。
- header被篡改。
- 页面被注入广告。
- 页面被重定向。
- 页面被重定向并重新iframe到新页面，框架嵌入广告。
- HTTPS请求被拦截。
- DNS劫持。

这些问题轻则影响用户体验，重则泄露数据，或影响公司信誉。

针对页面注入的行为，有一些解决方案：

使用CSP（Content Security Policy）

CSP可以有效的拦截页面中的非白名单资源，而且兼容性较好。在美团移动版的使用中，能够阻止大部分的页面内容注入。

但在使用中还是存在以下问题：

- 由于业务的需要，通常inline脚本还是在白名单中，会导致完全依赖内联的页面代码注入可以通过检测。
- 如果注入的内容是纯HTML+CSS的内容，则CSP无能为力。
- 无法解决页面被劫持的问题。
- 会带来额外的一些维护成本。

总体来说CSP是一个行之有效的防注入方案，但是如果对于安全要求更高的网站，这些还不够。

HTTPS

HTTPS可以防止页面被劫持或者注入，然而其副作用也是明显的，网络传输的性能和成功率都会下降，而且HTTPS的页面会要求页面内所有引用的资源也是HTTPS的，对于大型网站其迁移成本并不算低。

HTTPS的一个问题在于：一旦底层想要篡改或者劫持，会导致整个链接失效，页面无法展示。这会带来一个问题：本来页面只是会被注入广告，而且广告会被CSP拦截，而采用了HTTPS后，整个网页由于受到劫持完全无法展示。

对于安全要求不高的静态页面，就需要权衡HTTPS带来的利与弊了。