唯快不破：Web 应用的 13 个优化步骤

2017-06-09 前端大全

（点击上方公众号，可快速关注）

> 作者： 立青油菜花
> zhuanlan.zhihu.com/p/21417465
> 如有好文章投稿，请点击 → 这里了解详情

时过境迁，Web 应用比以往任何时候都更具交互性。搞定性能可以帮助你极大地改善终端用户的体验。阅读以下的技巧并学以致用，看看哪些可以用来改善延迟，渲染时间以及整体性能吧！

更快的 Web 应用

优化 Web 应用是一项费劲的工作。Web 应用不仅处于客户端和服务器端的两部分组件当中，通常来说也是由多种多样的技术栈构建而成：数据库，后端组件（一般也是搭建在不同技术架构之上的），以及前端（HTML + JavaScript + CSS + 转译器）。运行时也是变化多端的：iOS，Android，Chrome，Firefox，Edge。如果你曾经工作在一个不同的单一庞大的平台之上，通常情况下性能优化只针对于单一目标（甚至只是目标的单一版本而已），但是现在的话你就可能会意识到任务复杂度要远超于此。这就对了。但这儿也有一些通用的优化指南可以大大优化一个应用。我们将会在接下来的章节中探讨这些指南的内容。

> 一份 Bing 的研究表明，页面加载时间每增加 10ms，网站的年收入就会减少 25 万美元。 —— Rob Trace 和 David Walp，微软高级程序经理

过早优化?

优化最难的地方就是如何在开发生命周期中最适当的时候去做优化。Donald Knuth 有一句名言：「过早优化乃万恶之源」。这句话背后的原因非常简单：因为一不小心就会浪费时间去优化某个 1% 的地方，但是结果却并不会对性能造成什么重大影响。与此同时，一些优化还妨碍了可读性或者是可维护性，甚至还会引入新的 Bug。换句话说，优化不应当被认为是「意味着得到应用程序的最佳性能」，而是「探索优化应用的正确的方式，并得到最大的效益」。再换句话说，盲目的优化可能会导致效率的丢失，而收益却很小。在你应用以下技巧的时候请将此铭记在心。你最好的朋友就是分析工具：找到你可以进行通过优化获得最大程度改善的性能点，而不用损害应用开发的进程或者可维护性。

> 程序员们浪费了大量时间来思考，或者说是担忧，他们的程序中非关键部分的运行速度。并且他们对于性能的这些尝试，实际上却对代码的调试和维护有着非常消极的影响。我们应当忘记那些不重要的性能影响，在 97% 的时间里都可以这么说：过早优化乃万恶之源。当然我们也不应当在那关键的 3% 上放弃我们的机会。—— Donald Knuth

1. JavaScript 压缩和模块打包

JavaScript 应用是以源码形式进行分发的，而源码解析的效率是要比字节码低的。对于一小段脚本来说，区别可以忽略不计。但是对于更大型的应用，脚本的大小会对应用启动时间有着负面的影响。事实上，寄期望于使用 WebAssembly 而获得最大程度的改善，其中之一就是可以得到更快的启动时间。

另一方面，模块打包则用于将不同脚本打包在一起并放进同一文件。更少的 HTTP 请求和单个文件解析都可以减少加载时间。通常情况下，单独一种工具就可以处理打包和压缩。Webpack 就是其中之一。

示例代码：

```
function insert(i) {
    document.write("Sample " + i);
}

for(var i = 0; i < 30; ++i) {
    insert(i);
}
```

结果如下：

```
!function(r){function t(o){if(e[o])return e[o].exports;var n=e[o]={exports:
{},id:o,loaded:!1};returnr[o].call(n.exports,n,n.exports,t),n.loaded=!0,n.exports}var e={};return t.m=r,t.c=e,t.p="",t(0)}
([function(r,t){function e(r){document.write("Sample "+r)}for(var o=0;30>o;++o)e(o)}]);
//# sourceMappingURL=bundle.min.js.map
```

进一步打包

你也可以使用 Webpack 打包 CSS 文件以及合并图片。这些特性都可以有助于改善启动时间。研究一下 Webpack 文档来做些测试吧！

2. 按需加载资源

资源（特别是图片）的按需加载或者说惰性加载，可以有助于你的 Web 应用在整体上获得更好的性能。对于使用大量图片的页面来说惰性加载有着显著的三个好处：

- 减少向服务器发出的并发请求数量（这就使得页面的其他部分获得更快的加载时间）
- 减少浏览器的内存使用率（更少的图片，更少的内存）

- 减少服务器端的负载

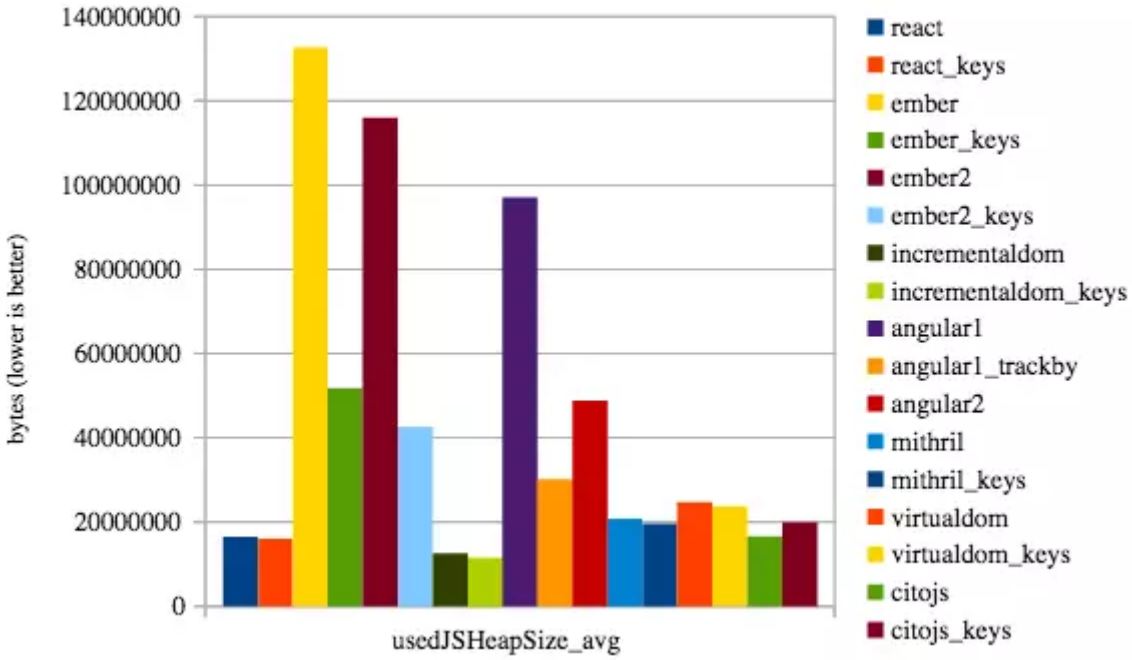
大体上的理念就是只在必要的时候才去加载图片或资源（如视频），比如在第一次被显示的时候，或者是在将要显示的时候对其进行加载。由于这种方式跟你建站的方式密切相关，惰性加载的解决方案通常需要借助其他库的插件或者扩展来实现。举个例子，react-lazy-load 就是一个用于处理 React 惰性加载图片的插件：

```
const MyComponent = () => (
  <div>
    Scroll to load images.
    <div className="filler" />
    <LazyLoad height={762} offsetVertical={300}>
      <img src='http://apod.nasa.gov/apod/image/1502/HDR_MVMQ20Feb2015ouellet1024.jpg' />
    </LazyLoad>
    (...)
```

一个非常好的实践范例就像 Goggle Images 的搜索工具一样。点击前面的链接并且滑动页面滚动条就可以看到效果了。

3. 在使用 DOM 操作库时用上 array-ids

如果你正在使用 React，Ember，Angular 或者其他 DOM 操作库，使用 array-ids（或者 Angular 1.x 中的 track-by 特性）非常有助于实现高性能，对于动态网页尤其如此。我们已经在上一篇程序衡量标准的文章中看到这个特性的效果了： More Benchmarks: Virtual DOM vs Angular 1 & 2 vs Mithril.js vs cito.js vs The Rest (Updated and Improved!)。



此特性背后的主要概念就是尽可能多地重用已有的节点。Array ids 使得 DOM 操作引擎可以「知道」在什么时候某个节点可以被映射到数组当中的某个元素。没有 array-ids 或者 track-by 的话，大部分库都会进行重新排序而摧毁已有的节点并重新创建新的。这就非常损耗性能了。

4. 缓存

Caches 是用于存储那些被频繁存取的静态数据的组件，便于随后对于这个数据的请求可以更快地被响应，或者说请求方式更加高效。由于 Web 应用是由很多可拆卸的部件组合而成，缓存就可以存在于架构中的很多部分。举例来说，缓存可以被放在动态内容服务器和客户端之间，就可以避免公共请求以减少服务器的负载，与此同时改善响应时间。其他缓存可能被放置在代码里，以优化某些用于脚本存取的通用模式，还有些缓存可能被放置在数据库或者是长运行进程之前。

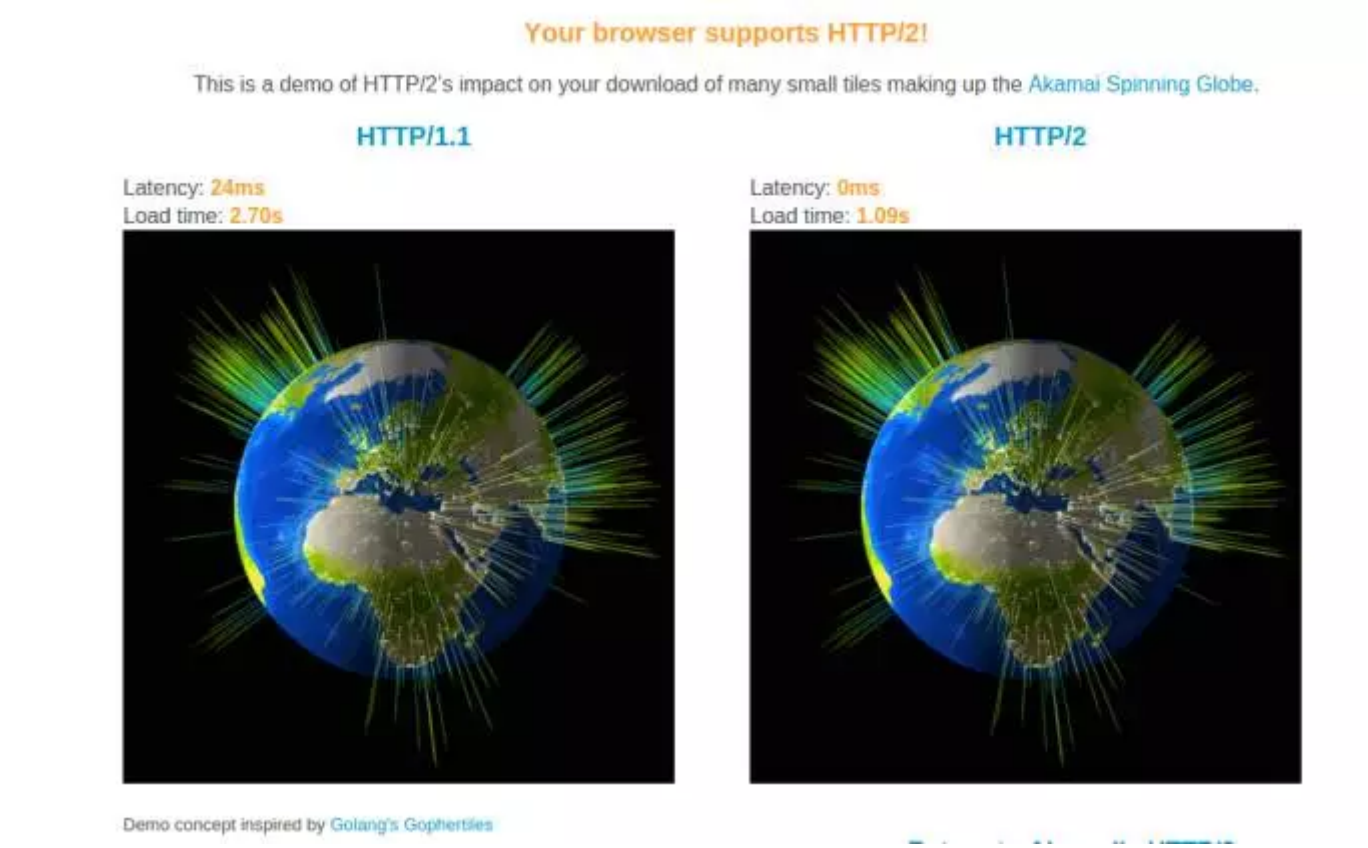
简而言之，在 Web 应用中使用缓存是一种改善响应时间和减少 CPU 使用的绝佳方式。难点就在于搞清楚哪里才是在架构中存放缓存的地方。再一次，答案就是性能分析：常见的瓶颈在哪里？数据或者结果可缓存吗？他们都太容易失效吗？这都是一些棘手的问题，需要从原理上来一点一点回答。

缓存的使用在 Web 环境中富有创造性。比如，basket.js 就是一个使用Local Storage 来缓存应用脚本的库。所以你的 Web 应用在第二次运行脚本的时候就可以几乎瞬间加载了。

如今一个广受欢迎的缓存服务就是亚马逊的 CloudFront。CloudFront 就跟通常的内容分发网络（CDN）用途一样，可以被设置作为动态内容的缓存。

5. 启用 HTTP/2

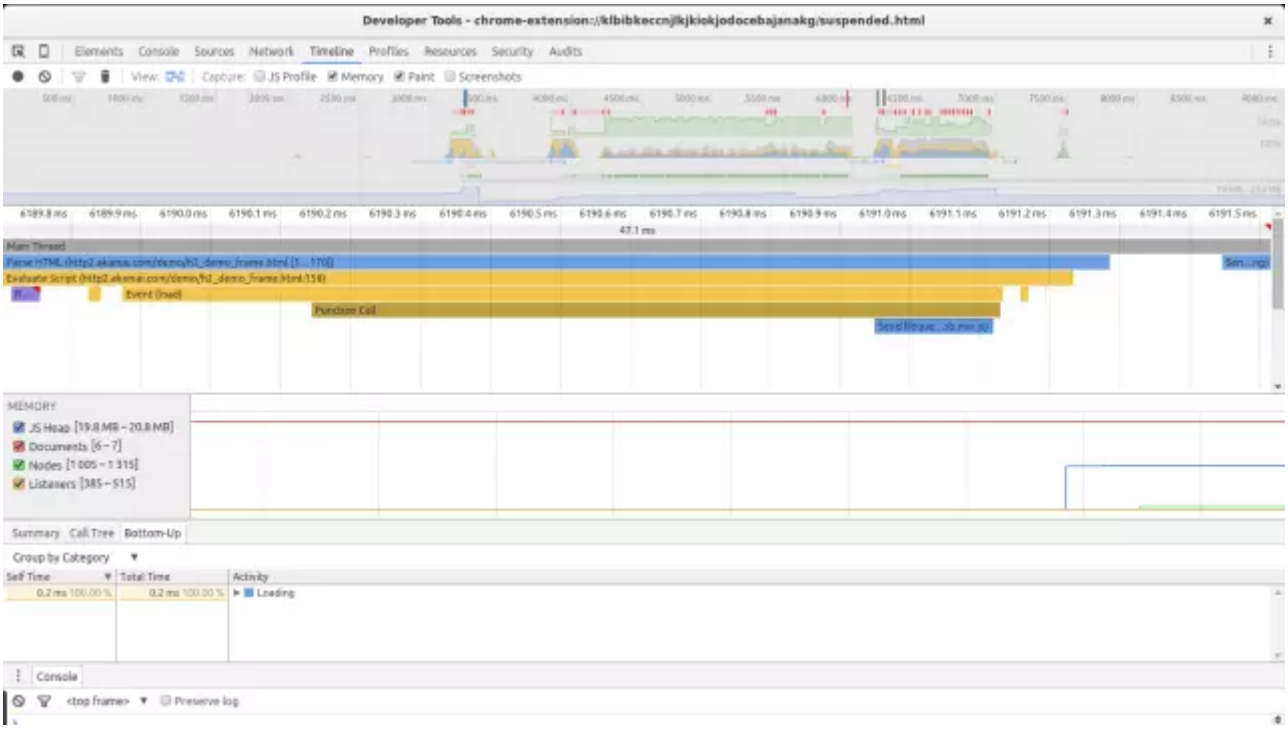
越来越多的浏览器都开始支持 HTTP/2。这可能听起来没有必要，但是 HTTP/2 为同一服务器的并发连接问题带来了很多好处。换句话说，如果有很多小型资源需要加载（如果你打包过的话就没有必要了），在延迟和性能方面 HTTP/2 秒杀 HTTP/1。试试 Akamai 的 HTTP/2 demo，可以在最新的浏览器中看到区别。



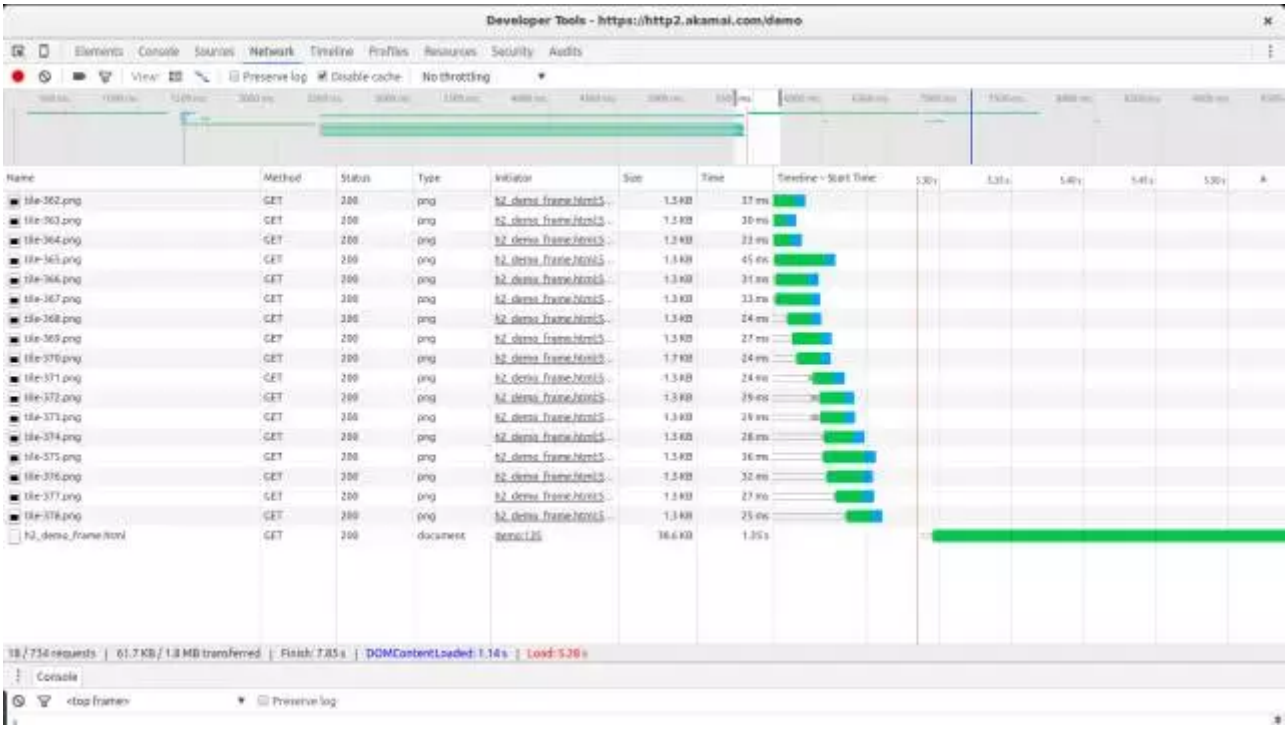
6. 应用性能分析

性能分析是优化任何应用程序时的重要一步。就像介绍中所提到的那样，盲目尝试优化应用经常会导致效率的浪费，微不足道的收益和更差的可维护性。执行性能分析是识别你的应用问题所在的一个重要步骤。

对于 Web 应用来说，延迟时间是最大的抱怨之一，所以你需要确保数据的加载和显示都尽可能得快。Chrome 提供了非常棒的性能分析工具。特别是 Chrome Dev Tools 中的时间线和网络视图都对于定位延迟问题有着很大的帮助：



时间线视图可以帮忙找到运行时间较长的操作。



网络视图可以帮助识别出额外的由缓慢请求导致的延迟或对于某一端点的串行访问。

正确分析的话，内存则是另一块可能获得收益的部分。如果你正在运行着一个拥有很多虚拟元素的页面（庞大的动态表格）或者可交互式的元素（比如游戏），内存优化可以获得更少的卡顿和更高的帧率。从我们最近的文章 4 Types of Memory Leaks in JavaScript and How to Get Rid Of Them 中，对于如何使用 Chrome 的开发工具有着进一步的深度理解。

CPU 性能分析也可以在 Chrome Dev Tools 中找到。看看这篇来自 Google 官方文档中的文章 Profiling JavaScript Performance。

找到性能损耗的中心可以让你有效率地达到优化的目标。

对后端的性能分析会更加困难。通常情况下，确认一个耗费较多时间的请求可以让你明确应该优先分析哪一个服务。对于后端的分析工具来说，则取决于所构建的技术栈。

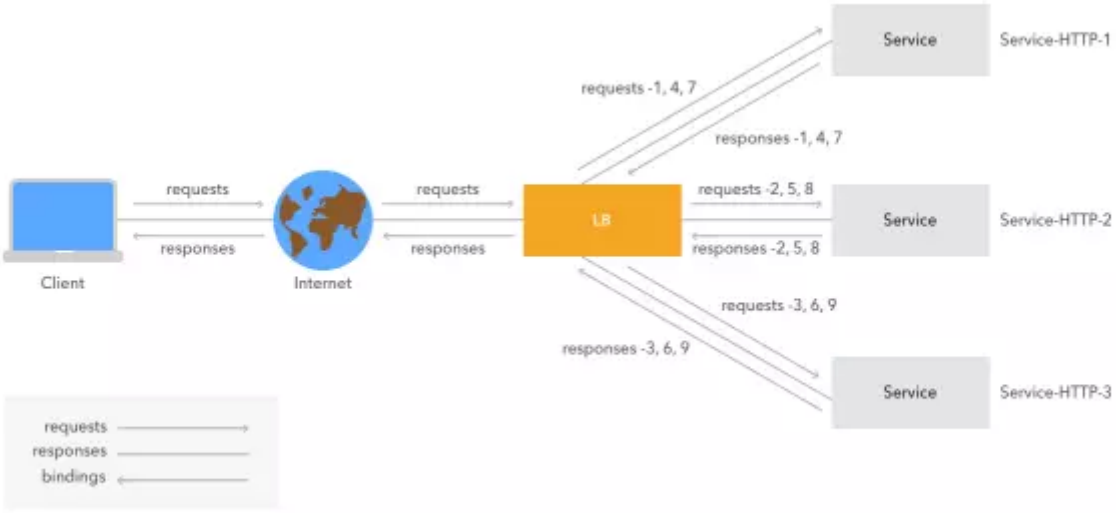
一个关于算法的注意事项

在大多数情况下，选择一个更优的算法，比围绕着小成本中心所实现的具体优化策略能够获得更大的收益。在某种程度上，CPU 和内存分析应该可以帮你找到大的性能瓶颈。当这些瓶颈跟编码问题并不相关时，则是时候考虑考虑不同的算法了。

7. 使用负载均衡方案

我们在之前讨论缓存的时候简要提到了内容分发网络（CDNs）。把负载分配到不同的服务器（甚至于不同的地理区域）可以给你的用户提供更好的延迟时间，但是这条路还很漫长，特别是在处理很多的并发连接的时候。

负载均衡就跟使用某个 round-robin（循环）解决方案一样简单，可以基于一个 nginx 反向代理，或者基于一个成熟的分布式网络，比如 Cloudflare 或者 Amazon CloudFront。



> 以上的图来自于 Citrix。 为了使负载均衡真正有效，动态内容和静态内容都应该被拆分成易于并发访问的。换句话说，元素的串形访问会削弱负载均衡器以最佳形式进行分流的能力。与此同时，对于资源的并发访问可以改善启动时间。

虽然负载均衡可能会很复杂。对最终一致性算法不友好的数据模型，或者缓存都会让事情更加困难。幸运的是，大多数应用对于已简化的数据集都只需要保证高层次的一致性即可。如果你的应用程序没有这样设计的话，就有必要重构一下了。

8. 为了更快的启动时间考虑一下同构 JavaScript

改善 Web 应用程序观感的方式之一，就是减少启动时间或者减少首页渲染时间。这对于新兴的单页面应用尤为重要，其需要在客户端执行大量任务。在客户端做更多事情通常就意味着，在第一次渲染被执行之前就需要下载更多的信息。同构 JavaScript 可以解决这个问题：自从 JavaScript 可以同时运行在客户端和服务器端，这就让在服务器端来执行页面的首次渲染成为可能，先把已渲染的页面发送出去然后再由客户端的脚本接管。这限制了所使用的后端（必须使用支持该特性的 JavaScript 框架），但却能获得更好的用户体验。举例来说，React 就很适合于做这个，就像以下代码所示：

```
var React = require('react/addons');
var ReactApp = React.createFactory(require('../components/ReactApp').ReactApp);

module.exports = function(app) {

  app.get('/', function(req, res){
    // React.renderToString takes your component
    // and generates the markup
    var reactHtml = React.renderToString(ReactApp({}));
    // Output html rendered by react
    // console.log(myAppHtml);
    res.render('index.ejs', {reactOutput: reactHtml});
  });

};
```

Meteor.js 对于客户端和服务器端的 JavaScript 混用有着非常棒的支持。

```
if (Meteor.isClient) {
 Template.hello.greeting = function () {
  return "Welcome to myapp.";
 };

 Template.hello.events({
  'click input': function () {
   // template data, if any, is available in 'this'
```

```
    if (typeof console !== 'undefined')
      console.log("You pressed the button");
  }
 });
}

if (Meteor.isServer) {
 Meteor.startup(function () {
  // code to run on server at startup
 });
}
```

但是，为了支持服务器端渲染，需要像 meteor-ssr 这样的插件。

> 谢谢 gabrielpoca 在评论中指出这一点。如果你有复杂的或者中等大小的应用需要支持同构部署，试试这个，你可能会感到惊讶的。

9. 使用索引加速数据库查询

如果你需要解决数据库查询耗费大量时间的问题（分析你的应用看看是否是这种情况！），是时候找出加速数据库的方法了。每个数据库和数据模型都有自己的权衡。数据库优化在每一方面都是一个主题：数据模型，数据库类型，具体实现方案，等等。提速可能不是那么的简单。但是这儿有个建议，可能可以对某些数据库有所帮助：索引。索引是一个过程，即数据库所创建的快速访问数据结构，从内部映射到键（在关系数据库中的列），可以提高检索相关数据的速度。大多数现代数据库都支持索引。索引并不是文档型数据库（比如 MongoDB）所独有的，也包括关系型数据库（比如PostgreSQL）。

为了使用索引来优化你的查询，你将需要研究一下应用程序的访问模式：什么是最常见的查询，在哪个键或列中执行搜索，等等。

10. 使用更快的转译方案

JavaScript 软件技术栈一如既往的复杂。而改善语言本身的需求则又增加了复杂度。不幸地是，JavaScript 作为目标平台又会被用户的运行时所限制。尽管很多改进已经以 ECMAScript 2015（2016正在进行）的形式实现了，但是通常情况下，对客户端代码来说又不可能依赖于这个版本。这种趋势促使了一系列的转译器：用于处理 ECMAScript 2015 代码的工具和只使用 ECMAScript 5 结构实现其中所缺失的特性。与此同时，模块绑定和压缩处理也已经被集成到这个生产过程中，被称为为发布而构建的代码版本。这些工具可以转化代码，并且能够以有限的方式影响到最终代码的性能。Google 开发者 Paul Irish 花了一些时间来寻找这些转译方案会如何影响性能和最终代码的大小。尽管大多数情况下收益会很小，但也值得在正式采用某个工具栈之前看看这些数据。对于大型应用程序来说，这种区别可能会影响重大。

11. 避免或最小化 JavaScript 和 CSS 的使用而阻塞渲染

JavaScript 和 CSS 资源都会阻塞页面的渲染。通过采取某些的规则，你可以保证你的脚本和 CSS 被尽可能快速地处理，以便于浏览器能够显示你的网站内容。

在 CSS 的情况下这是非常重要的，所有的 CSS 规则都不能与特定媒体直接相关，规则只用于处理你准备在页面上所显示内容的优先级。这可以通过使用 CSS 媒体查询来实现。媒体查询告诉浏览器，哪些 CSS 样式表应用在某个特定的显示媒体上。举个例子，用于打印的某些规则可以被赋予比用于屏幕显示更低的优先级。

媒体查询可以被设置成 <link> 标签属性：

```
<link rel="stylesheet" type="text/css" media="only screen and (max-device-width: 480px)" href="mobile-device.css" />
```

轮到 JavaScript 了，关键就在于遵循某些用于内联 JavaScript 的规则（比如内联在 HTML 文件当中的代码）。内联 JavaScript 应该尽可能短，并将其放在不会阻塞页面剩余部分解析的地方。换句话说，被放在 HTML 树中间的内联 JavaScript 将会在这个地方阻塞解析器，并强制其等待直到脚本被执行完毕。如果在 HTML 文件中随意放了一些大的代码块或者很多小的代码块，对于性能来说这会成为性能杀手。内联可以有效减少额外对于某些特定脚本的网络请求。但是对于重复使用的脚本或者大的代码块来说，这个好处就可以忽略不计了。

防止 JavaScript 阻塞解析器和渲染器的一种方法就是将 <script> 标签标记为异步的。这限制了我们对于 DOM 的访问但是可以让浏览器不管脚本的执行状态而继续解析和渲染页面。换句话说，为了获得最佳的启动时间，确保那些对于渲染不重要的脚本已经通过异步属性的方式标记成异步的了。

```
<script src="async.js" async></script>
```

12. 用于未来的一个建议：使用 service workers + 流

Jake Archibald 最近的一篇博文详细描述了一种有趣的技术，可以用于加速渲染时间：将 service workers 和流结合起来。结果非常令人叹服：

不幸的是这个技术所需要的 APIs 都还不稳定，这也是为什么这是一种有趣的概念但现在还没有真正被应用的原因。这个想法的主旨就是在网站和客户端之间放置一个 service worker。这个 service worker 可以在获取缺失信息的同时缓存某些数据（比如 header 和一些不会经常改变的东西）。缺失的内容就可以尽可能快速地流向被渲染的页面。

https://www.youtube.com/watch?v=Cjo9iq8k-bc

13. 更新：图片编码优化

我们的一个读者指出了一个非常重要的遗漏：图片编码优化。PNGs 和 JPGs 在 Web 发布时都会使用次优的设置进行编码。通过改变编码器和它的设置，对于需要大量图片的网站来说可以获得有效的改善。流行的解决方案包括 OptiPNG 和jpegtran。

A guide to PNG optimization（http://optipng.sourceforge.net/pngtech/optipng.html） 详细描述了 OptiPNG 可以如何用于优化 PNGs。

The man page for jpegtran（http://linux.die.net/man/1/jpegtran） 对它的一些特性提供了很好的介绍。

如果你发现这些指南相对于你的要求来说都太复杂了的话，这儿有一些在线网站可以提供优化服务。也有一些像 RIOT 一样的图形化界面，非常有助于批量操作和结果检查。

悄悄话：Auth0 中常见的优化

我们是一个 Web 公司。就以这种身份来说，我们为我们的基础设施的某些部分部署了一些特定的优化。举例来说，在登录页面你可以发现，在我们域名的 /learn 路径下（比如，登录页面的单点登录），我们采用了一种特别的优化：为了方便我们使用 CMS 来创建每篇文章。因为文章都没有中心索引，但是为了能够被搜索引擎发现，使用了 webtask 的爬虫来预渲染每个页面并生成了一个静态版本然后上传到我们 CDN。这减少了我们在服务器端上的压力，因为无须为每个访客都生成动态的服务器端内容。与此同时还改善了延迟（并且隔离了我们发现与 CMS 相关的安全问题）。

对于文档部分，我们正在使用同构 JavaScript，这让我们获得了非常棒的启动时间，并且使我们的后端和前端团队能够轻松集成。

结论

由于应用程序变得越来越大和越来越复杂，性能优化对于 Web 开发来说正在变得越来越重要。在做出任何值得的时间和潜在的未来成本的优化尝试时，有针对性的改进都是必不可少的。Web 应用程序早已突破了大多数静态内容的边界，学习常见模式进行优化则是令人愉悦的应用和完全不可用的应用之间最大的区别（这是让你的访客留下来的长远之计！）。没有什么规则是绝对的，但是：性能分析和研究特定软件技术栈的错综复杂之处，是找出如何优化它的唯一方式。你曾经发现过对你的应用产生巨大影响的其他建议吗？请留言让我们知道。Hack on!