深入浅出浏览器渲染原理

对于HTTP协议和浏览器渲染原理都是理解容易但不好讲明白，那么为什么不采用Node.js来阐述呢？以实践的方式、最简单的方式来向你展示不好讲的东西，对于Node.js开发者和大前端开发来说都是非常实用的，以上是本文主要内容概述。

今天的前端

- JavaScript变成了1等公民，发展速度特别快
- Web开发愈加现代化
- 开源项目指数级增长

而Web开发的核心还是浏览器，无论如何都绕不开的，只是场景增加了而已。

- 浏览器端b/s架构的应用
- PC端，将web应用打包成exe或dmg、deb等安装文件
- 移动端，将web应用打包成ipa或apk

那么我们所谓的浏览器机制你又了解多少呢？

访问一个网站，浏览器是如何展示的

大部分人都是这样做的：

- 打开浏览器
- 输入网址
- “啪”，回车
- 等着…
- 有网页或者报错

感觉像是这样，我想要一个网页，浏览器你过去取，取回来给我。事实上，也确实是这样简单。

- 往返，获得html
- 将获得的结果，展示出来

这样说，太通俗了，加点属于：

- 通过 http协议 获得html
- 浏览器 渲染 html

所以，对于开发来说，能够关心的也就只有这2个要点了： http协议 和 浏览器渲染原理 ，当然，这也是本文要讨论的重点，而且要借助于Node.js来理解。

> 浏览器就是一个跑腿的，请善待他。

架构发展

计算机发展这么多年，也是一部大块头历史，这里我们就讨论2个经典的架构： b/s 架构和 c/s 架构。

c/s 架构，即 客户端/服务端 架构：

- c = client = 客户端，指的是PC端的需要安装的软件，比如QQ
- s = server = 服务器端
- 开发比较容易(vb\vf\vc\delphi等等)，操作简便，但应用程序的升级和客户端程序的维护较为困难
- 在2005年以前，跨平台不多，像java这种还不争气…

b/s 架构，即 浏览器/服务端 架构：

- b = browser = 浏览器端，指的是在浏览器里运行的应用
- s = server = 服务器端
- 有浏览器就能运行，不需要安装，听起来就很好
- 网页被认为是跨平台的…

总结：公司都是盈利组织，肯定是趋利避害的， b/s 架构远远比 c/s 架构轻量，这体现在开发、使用、成本，一般我们管 b/s 架构的软件称为“瘦应用”，至于浏览器兼容如何坑死前端er们那是后话！

三层结构

架构变革，相当于给了浏览器开发一个名分，随着web 2.0中ajax对交互体验的升级，慢慢的b/s架构就变成了主流，一切都要从ajax开始，页面无刷新，即可获得数据。妈妈再也不用担心页面跳转问题了！

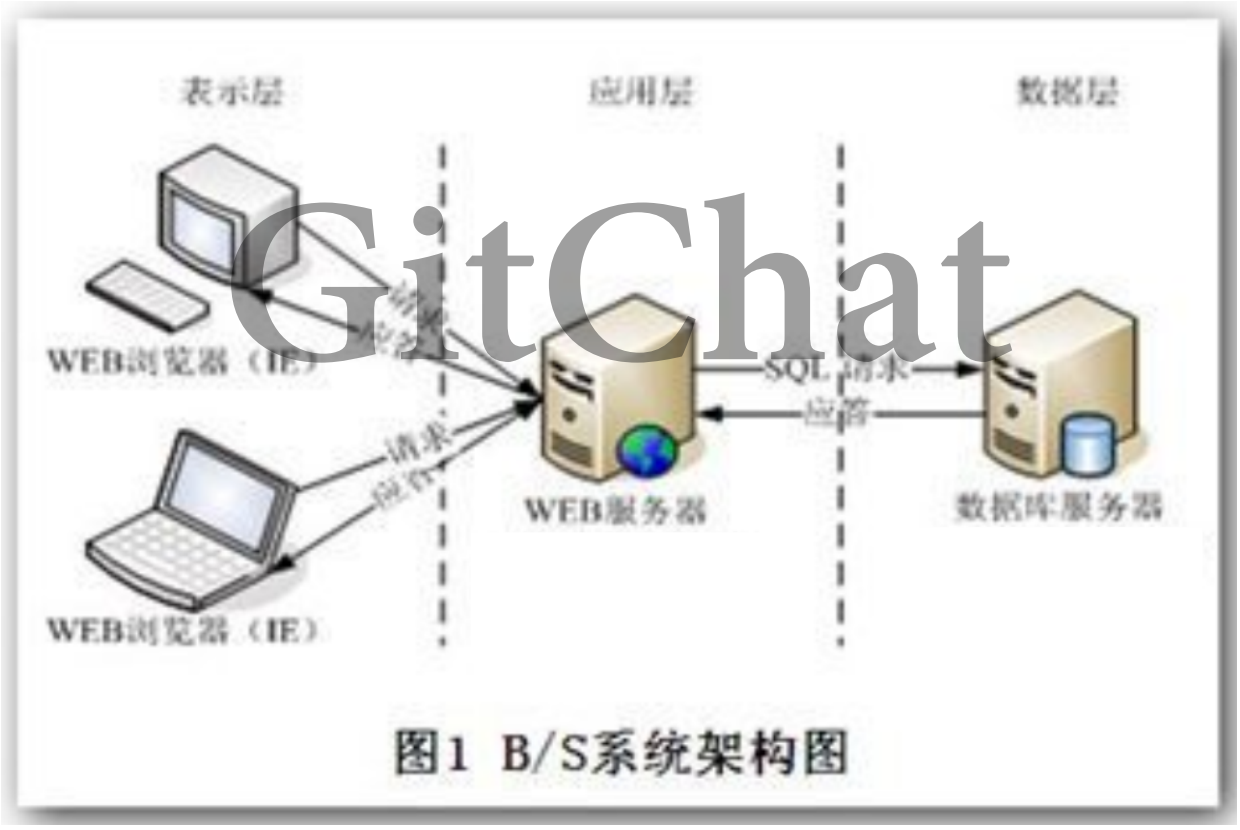


图1 B/S系统架构图

在c/s架构就应用做了拆分，所谓的三层结构就是由逻辑上相互分离的表示层、业务层和数据层构成。在随后的b/s架构里也得以继承。

尽管有3层结构，逻辑上看起来清晰了，最早的代码比如asp、jsp等都像今天php一页，所有的东西在一个文件，页面渲染、数据库连接操作等等。真是不想碰啊！

说一下前端定义问题：

- 那时写网页是前端
- 不包括web server相关的
- 更不要提数据库

今天的前端：

- 主要是写web开发
- 或多或少包括web server相关的
- 浏览器内置数据库，关系型和NoSQL都有…

HTTP

请求/响应模型

一般我们用 Node.js 做的最多是 web server 开发，就来个最简单的 http server 吧。

创建 hello_node.js 文件，写下下面的代码。

```
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello Node.js\n');
}).listen(3000, "127.0.0.1");

console.log('Server running at http://127.0.0.1:3000/');
```

说明：

- 1）引用了 Node.js SDK内置的 http 模块
- 2）通过 http.createServer 创建了一个 http 服务
- 3）通过 listen 方法，指定端口和 ip ，启动该服务
- 4） res 是 http 协议里的 response （响应）的别名，通过res控制对浏览器的操作，设置返回状态码是200，设置header里的'Content-Type'是'text/plain'类型，最后返回'Hello Node.js\n'文本。

通过短短几行代码就可以创建一个 http 服务，是不是非常简单啊？

执行如下：

```
> node hello_node.js
Server running at http://127.0.0.1:3000/
```

通过 node 解释器，执行 hello_node.js 文件里的 Node.js 代码，这是典型的 Node.js 执行过程，脚本不需要编译，整体来说还是比较简单的。

访问浏览器，结果如图。

例子不是最重要，通过node的http.createServer方法来理解请求/应答模型会更好。

```
http.createServer(function (req, res) {
    。。。
})
```

你能够处理的只有req和res。每次请求都要这样处理，这也就非常容易理解图中的“HTTP协议是无状态协议”了。

> 无状态: 每次的请求和响应都是新的。

Content-Type

前面渲染的是文本。

```
var http = require('http');

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('<h1>Hello Node.js<h1>');
}).listen(3000, "127.0.0.1");

console.log('Server running at http://127.0.0.1:3000/');
```

设置’Content-Type’对应的值为’text/html’，这样浏览器就知道按照html来解析了。不然呢？你后面讲什么浏览器渲染啊。

如果，我们：

- 想渲染一个json
- 想渲染一个gif图片

Content-Type 为是headers里比较特殊的一个，是MIME类型（ Multipurpose Internet Mail Extensions type)的设置项。是描述消息内容类型的因特网标准，包含文本、图像、音频、视频以及其他应用程序专用的数据。

举例：

类型	后缀	content-type	浏览器渲染
普通文本	.txt	text/plain	文本
HTML	.html	text/html	网页
JSON	.json	application/json	json文本
Gif图片	.gif	image/gif	图片

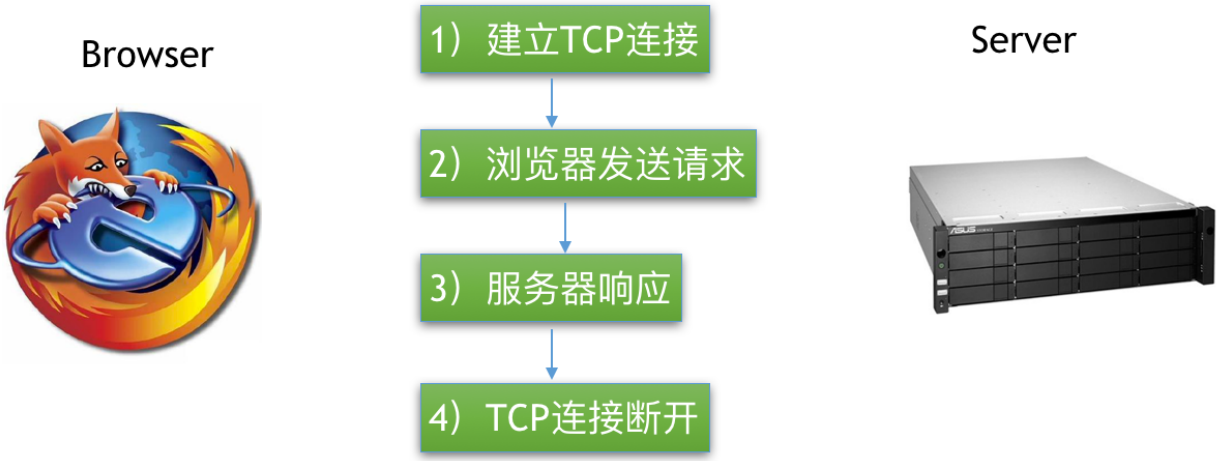
为啥要建立TCP连接呢？

前面说了，HTTP是无状态协议啊，为啥又扯到了tcp呢？

> WEB使用HTTP协议作应用层协议，以封装HTTP文本信息，然后使用TCP/IP做传输层协议将它发到网络上。

TPC/IP协议是传输层协议，主要解决数据如何在网络中传输，而HTTP是应用层协议，主要解决如何包装数据。HTTP是TCP之上的，它当然也需要靠TCP来做传输啦，所以这一步是躲不过去的，只是我们看不到而已。

我们只说Web的核心是HTTP协议，忽略了TCP，具体步骤看图：



为啥要建立TCP连接呢?

如果大家有兴趣，可以看看1）OSI七层模型，2）TCP/IP四层模型，绝对会有非常大的收获！

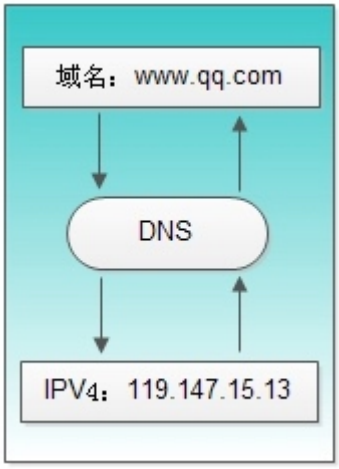
> 一般说来，枯燥的才是最本质的！

比如通过3次握手建立TCP连接：

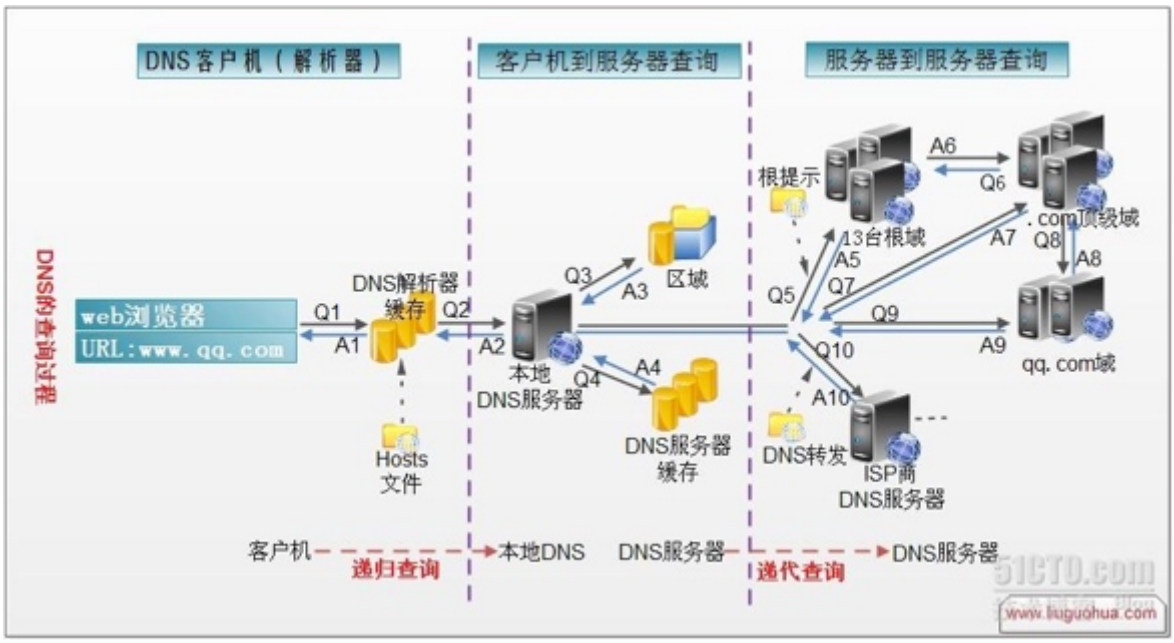
具体如何握手，这里就不细讲了，唯一的一个问题是，TCP建立连接只能通过IP？

DNS解析

浏览器大部分访问的是网址，不是ip，咋办？这就是DNS要做的事儿，域名最终还是要变成ip才能访问的，不然TCP/IP里的ip是干嘛的？哈哈。



具体DNS查询过程原理：



要点：

- 先找本地host，所以这里是可以干很多坏事，拦截、模拟，前端都要做的
- 然后本地DNS解析器
- 然后域名解析

我们关注的问题：

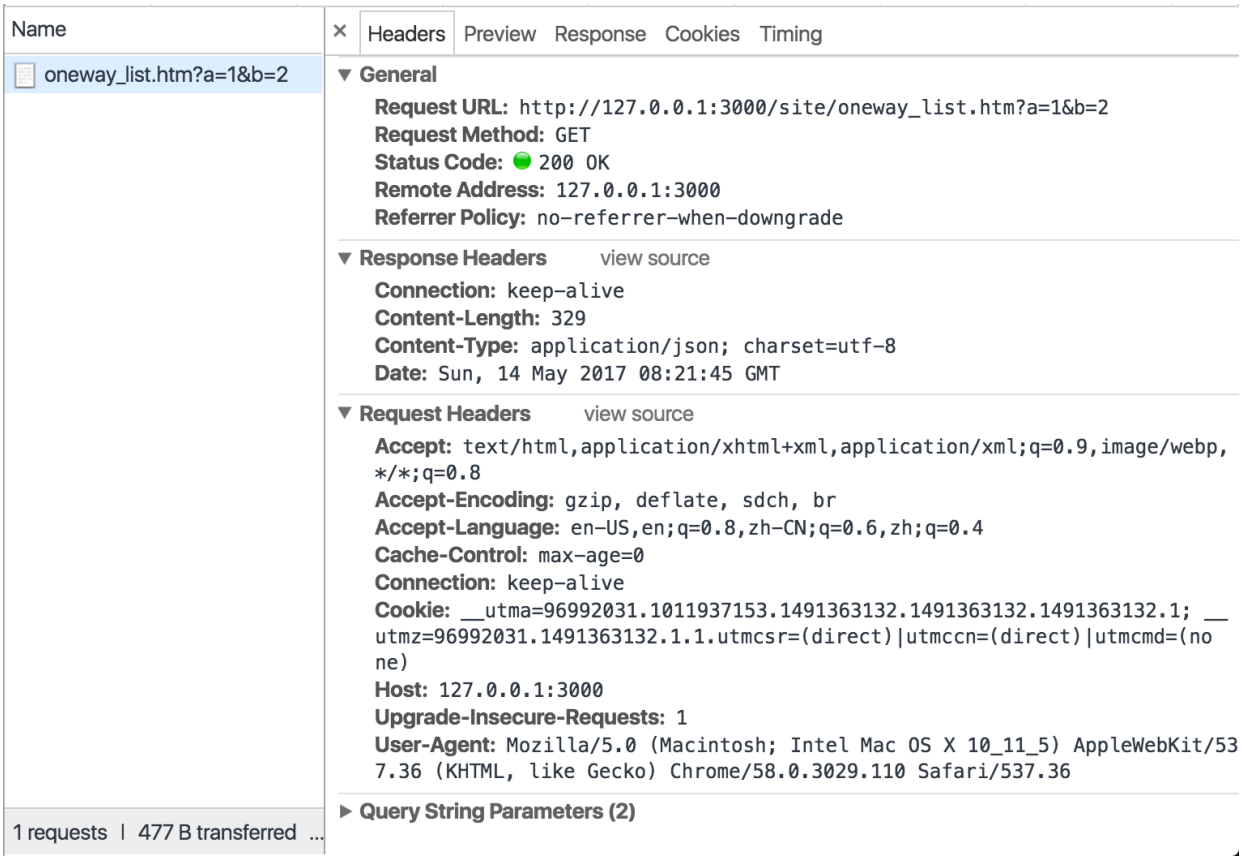
- 查询就要花时间，如果有缓存呢？
- 利用DNS查询原理，可以做很多对开发友好的事儿，比如浏览器代理，是不是就不用走本地dns了？

Headers

HTTP协议采用了请求/响应模型，浏览器或其他客户端发出请求，服务器给与响应。就整个网络资源传输而言，包括message-header和message-body两部分。首先传递的是message- header，即http header消息。在rfc2616中， http header 消息通常被分为4个部分：general header, request header, response header, entity header。

对于开发者而言，无论是使用，还是性能调优，都必须掌握Header的相关内容。

我们在chrome里看一下上面例子的网络请求：



浏览器缓存

浏览器缓存相关的headers，举例：

- cache-control
- Expires
- etag

谁把它们放到浏览器里的呢？回想一下之前的代码。

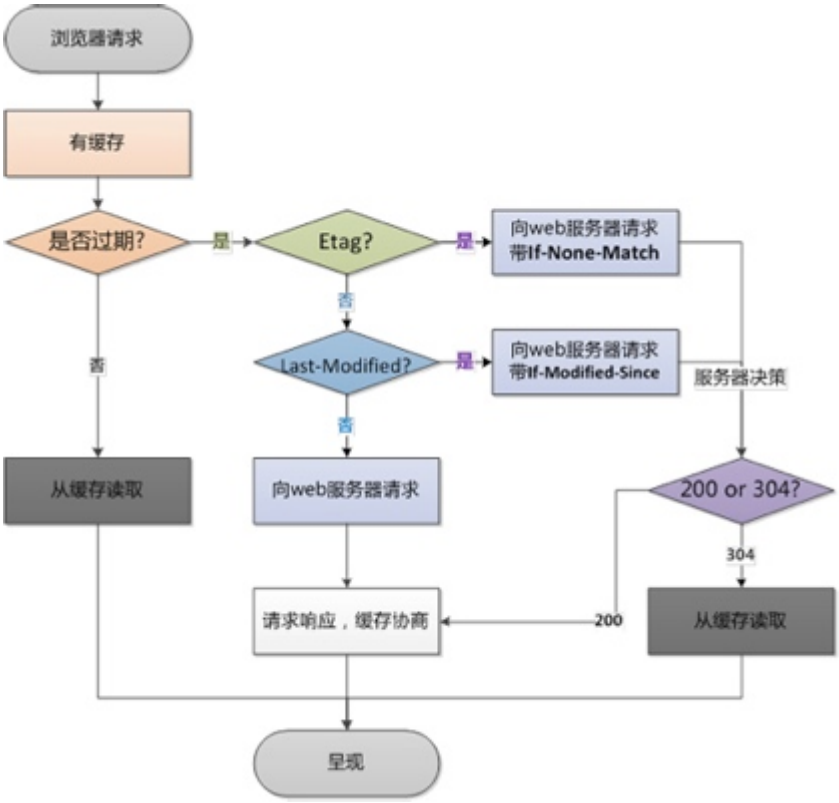
```
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('<h1>Hello Node.js<h1>');
})
```

`res.writeHead` 就是服务器向浏览器写入headers的方法，当然不同的框架里会有些许差异，底层都是这个。

比如，不开启缓存：

```
res.writeHead(200, {
    'Cache-Control': 'no-cache'
});
```

原理就这点，剩下的就是http协议里的内容了，想怎么优化，听你的，你的地盘你做主！细节不讲，但流程要看。



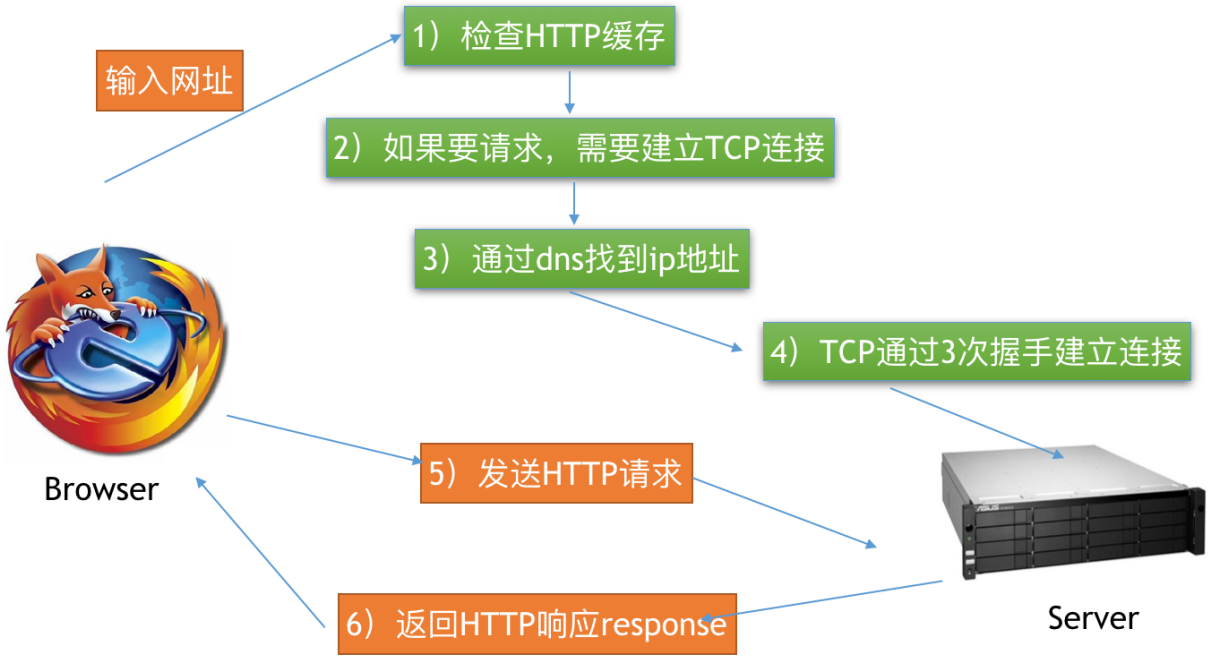
浏览器缓存策略：

- 浏览器会优先检查本地缓存
- 然后是Etag，etag如果状态码是302，依然走本地
- 实在不行，才走服务器

大家都不傻，走一次http请求的代价还是比较大的。

> 缓存是优化性能最好最快的方式！

完整http的请求过程



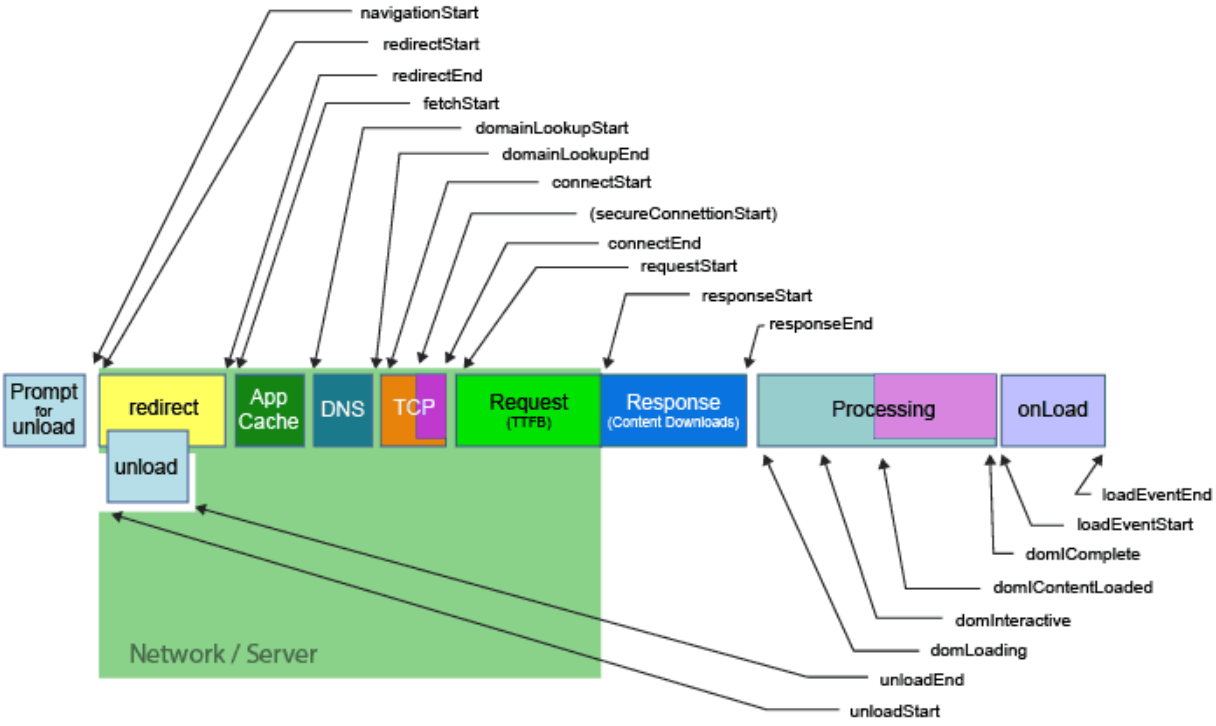
一次完整的请求过程就是这么麻烦：

- 浏览器自身缓存策略开始
- dns查询
- tcp三次握手
- 然后才是http里面的内容

为啥要了解这些弯弯道道呢？

> 如果不掌握原理，你的优化就是上坟烧报纸。

浏览器性能优化



一次完整的请求过程：

- 浏览器自身缓存策略开始（App Cache）
- dns查询（DNS）
- tcp三次握手（TCP）
- 然后才是http里面的内容
    - Request
    - Response
- 剩下的就是浏览器处理了

那么对应的优化呢?

1）App Cache

- 浏览器会优先检查本地缓存
- 然后是Etag，etag如果状态码是302，依然走本地
- 实在不行，才走服务器

能不走服务器就尽量不走服务器。

2）DNS

- 缓存，减少查询时间
- 开发代理工具、提高工作效率

3）TCP

能做不多，简单说，传输的内容越少越好，所以，各种压缩，tree-shaking、DCE(dead code elimination)等。

4）HTTP协议

理解req和res，剩下的就是各种和浏览器交互的headers，你在chrome devtools看到的几乎都可以算进来，比如：

- headers：比如bigpipe，分块传输，保证首屏和加载速度，微博，facebook，qunar、京东、淘宝都大量应用，比如Last-Modified和Etags
- 应用缓存，cookie、session

最好的学习办法就是自己一个一个的敲，没有捷径，好在Node最适合做http server这种io密集型的，而且还是js语法。

> practice makes perfect!

其他

- 抓包工具，Charles、fiddler、wireshark
- https，自己通过acme.sh搭建一套，理解的更深
- http2，使用node-spdy搭建
- pwa里的service-worker实现的非常有意思，那套缓存思路也是绝了
- ssr目前实现最好的vue的ssr，做了很多优化

当下最潮的web开发技术，你其实都可以玩的更帅！

浏览器渲染

讲浏览器渲染，大都是渲染引擎基本流程4步：

- 解析dom树
- 渲染树构建
- 对渲染树进行布局
- 展示

其中dom树包括dom和cssom解析，再有就是js加载优化。

关键点：

- 1）缩短白屏时间
- 2）加快首屏显示
- 3）尽快监听主要操作的事件
- 4）优化关键呈现路径

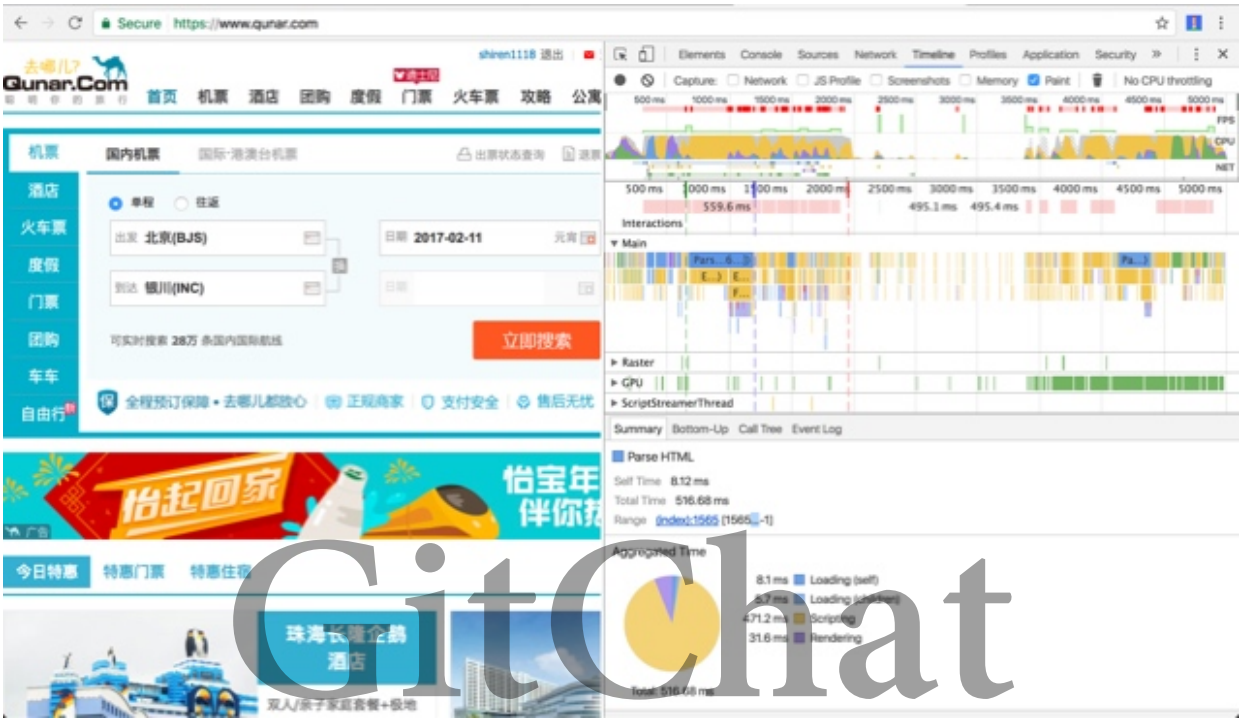
我们要关注的2点：

- 尽快展示，这是http部分，前面主要讲的
- 优化浏览器里面效率

工具的使用

无论是firebug还是Chrome DevTools，必须要会的技能。

举个例子：



对于dom和cssom的性能查看，继而根据代码进行调整就好。

首屏加载时间

一般性能监控都会看的ttfb指标：

> TTFB (Time To First Byte)，是最初的网络请求被发起到从服务器接收到第一个字节这段时间，它包含了 TCP连接时间，发送HTTP请求时间和获得响应消息第一个字节的时间。

你想让用户尽快看到内容，有很多处理方式，比如bigpipe：

> HTTP 1.1引入分块传输编码。

HTTP分块传输编码格式：

> Transfer-Encoding: chunked 如果一个HTTP消息（请求消息或应答消息）的Transfer-Encoding消息头的值为chunked，那么，消息体由数量未定的块组成，并以最后一个大小为0的块为结束。

Nodejs自动开启 chunked encoding：

> 除非通过sendHeader()设置Content-Length头。

举例：

```
'use strict'

var http = require('http')

const sleep = ms => new Promise(r => setTimeout(r, ms))

var app = http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/html', 'charset':
'utf-8' })

  res.write('loading...<br>')

  return sleep(2000).then(function() {
    res.write(`timer: 2000ms<br>`)
    return sleep(5000)
  })
  .then(function() {
    res.write(`timer: 5000ms<br>`)
  }).then(function() {
    res.end()
  })
})
```

```
app.listen(3000)
```

更多，参见 https://github.com/i5ting/nodejs-bigpipe-demo。

我们来反思一下模板引擎的使用，我们真的用对了么？

- 布局（加载js、css）
- 具体html实现
- 一次性写入

有木有更好的方式？

优化关键呈现路径

为了在首次渲染时尽可能快，我们需要优化以下三个变量：

- 最小化关键资源数
- 最小化关键字节数
- 最小化关键路径长度

常规步骤：

1. 分析并描述关键路径：资源数、字节数和长度；
2. 减少关键资源的数量：删掉、延迟下载或标记为异步等等；
3. 优化剩余关键资源的加载顺序：尽早下载所有关键资源以缩短关键路径长度；
4. 优化关键字节数以减少下载时间（往返次数）。

你可以做的更多

通过contentType赋值，可以渲染html，可是写html复用性太差，我们都喜欢模板，各种模板，好像不用模板就掉价一样。

模板的原理：

> 编译 (模板 + data) => html字符串

```
var nunjucks = require('nunjucks')
var compiledData = nunjucks.renderString('Hello {{ username }}',
```

```
{ username: 'James' });
console.log(compiledData)
```

就这么简单！在express或者koa里，直接调用res.render或ctx.render就可以了。

如果再延伸呢？定义模板，给不同的数据，生产不同的代码，这不是代码生成器么？俗称脚手架。

- 初始化模块
- cli二进制模块
- 模板引擎使用
- 解析cli参数和路径
- npm发布

具体参见《零基础十分钟教你用Node.js写生成器（scaffold）：你只需要5步》

采用化用的思想，深入的发现各种技奇技淫巧，让开发变得更快乐，这才是我们的终极目标！

总结

浏览器性能优化就只有2个点：

- http
- 浏览器的渲染原理（有很多也是http辅助完成的）

掌握这些够了，剩下的就是实践！编程没有捷径，优化也没有捷径，对我们而言，node能够辅助我们队http协议以及浏览器优化有更好的理解，能够真正的动手去练习、尝试，这才是它最大的好处。

把原本枯燥的东西变得有趣，你才会热爱它！加油，要相信自己一定可以做的！

> 狼叔说：“少抱怨，多思考，未来更美好”。

本来计划再讲讲Node HTTP三大部分：EventEmitters \ Streams \ HTTP，也有很多有意思的要点，限于篇幅，以后再讲吧，如果有疑问可以在问题里提问！