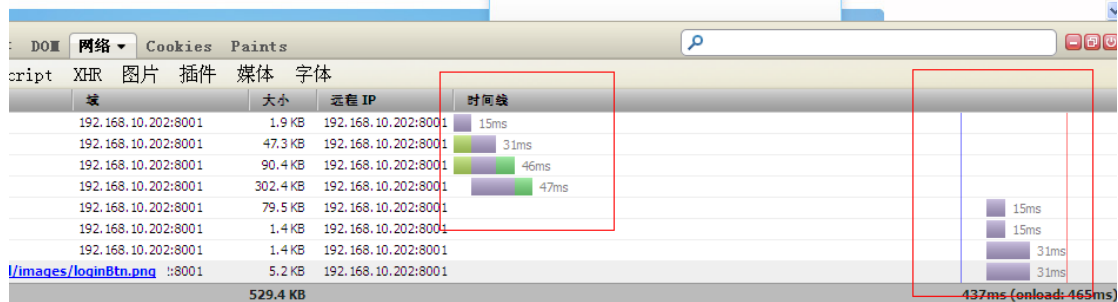


## 从 Timeline 时间线看 Dom 资源的加载

浏览器用 FF25.0, Firebug 的使用, 不想多讲了, 说下 firebug 中时间线的问题。可能正确理解时间线的不多。以下说下时间线的问题。

不废话, 看图

### 华通业务管理系统



可能有人会说。哦, 不就是时间线么。、经常看到, 有什么问题。

你有没有想过这些时间线的颜色, 个数, 出现的先后顺序, 颜色条的长度等信息代表的意义。未必能说的很清楚把。呵呵

在浏览器的工作原理里, 已经概要的说了下, 浏览器请求资源的工作模式。从到到下扫描, 资源请求可并发, 可执行脚本会阻塞后续请求。

详细=>[浏览器工作原理上篇](#), [浏览器工作原理下篇](#)

时间线能带给我们什么信息?

根据官方的描述, 每一个资源请求, 都有一个对应的 **tab** 线条, 先展示资源的加载开始和加载结束, **tab** 线条的长度则代表着持续时间。这种持续时间是相对的。通过这些线条, 你能了解到一些你不曾知道的信息。

通过这些时间线:

- 1.你能看到每个资源请求了多少时间
- 2.你能知道哪些脚本执行, 导致后续资源请求堵塞, 并且能知道这些脚本执行了多长的时间。
- 3.你能知道哪些后台资源文件的请求耗时过长, 这种问题一般就是后台代码的性能问题
- 4.你还能看到什么, 什么都能看到。

心意合一, 只要你懂得怎么看, 你想看的都有。:))))

为了后文减少一些必要的疑问。我需要先讲一些浏览器并发连接数的问题。

一次 **request**, 页面上有上百个资源是很正常的, 甚至会更多。这些资源都会被请求返回。很显然。不可能是单线程同步请求, 不然我们开一个一页面, 岂不是, 所有资源请求完要耗时

很长。

欣慰的是，开发浏览器的人，一定清楚用多线程的技术。是啊，但是这一百多个资源，或者少点说，几十个资源，他为什么不有多少资源就并发多少线程去请求呢？

是啊，为什么呢？

真实情况是，浏览器针对同一域名下的并发请求个数是有限制的(限制是针对 domain 的)。注意，是针对同一域名下的请求个数，不同域名下的请求无限制。

浏览器在请求的时候，请求走 http/https 协议，典型的请求响应式(握手请求模型)，也就是说通信走的是 TCP 连接,发起 TCP 连接->建立连接->建立会话->Server 处理->写回通知->应答结束->连接关闭。

嗯，这里有 http 连接和 Socket 连接的简要说明，电梯=>[Http 连接和 Socket 连接的说明](#)

深入和详细的查阅两本书，一本是 Http 通信协议，一本是网络技术原理。感兴趣的可以自行查下。

回到我们的问题上来，为什么要限制并发连接个数。在我上面降到的，请求响应模型，这场会话，具体内容不是主体，对资源并发个数产生限制的会话环境的维持。浏览器底层需要维护请求的会话环境，比如端口，比如协议栈通信的资源，如内存，其他的还有 cookie。

而这些东西，有些资源是很宝贵的。就向用数据库连接一样，老鸟告诉菜鸟说，要用 using，来保证用完立即释放，端口作为硬件资源也是一样的。很宝贵，因为端口总共才有 65536 个，不可以一直占用或者占用过度。

现在浏览器在并发请求技术上做了很多优化。往往是针对一个域名，创建一个 TCP 连接，后续的资源请求，复用打开的那个 TCP 连接，相当与用，我一个页面，一个 gif，一个 css，一个 js，四个资源挨个请求，实际上，TCP 连接在请求页面的时候被打开，后面的三个资源请求是复用之前打开的那个连接。

试想，复用一个 TCP 连接，css，js，还有图片这些数据传输，传输会消耗多少时间呢，如果要消耗很久，通常是 ms 级。如果要传输很久，则意味着 Server 的 TCP 端口一直处理被占用的状态。那么，如果这样，Server 能应对针对这一个页面的高并发请求么？


显然是不可能的。所以，为了解决这种宝贵资源超时或者过度占用的问题。开发浏览器的人，针对复用 TCP 连接，限制了资源并发请求的个数。当然这是客户端的限制。为了保证 Client 和 Server 端均有比较好的体验。Server 端针对并发请求也做了限制。两边共同限制了这种宝贵资源的超时或者过度占用问题。

所以。浏览器资源请求并发个数是有限制的。当然这只是从端口上来讲的。还有其他更深层次的东西，比如我多个线程请求资源，我接收其他线程返回的资源是要切换线程的，这会有

嗯，说到这里，我觉得有一张图可以登场了。

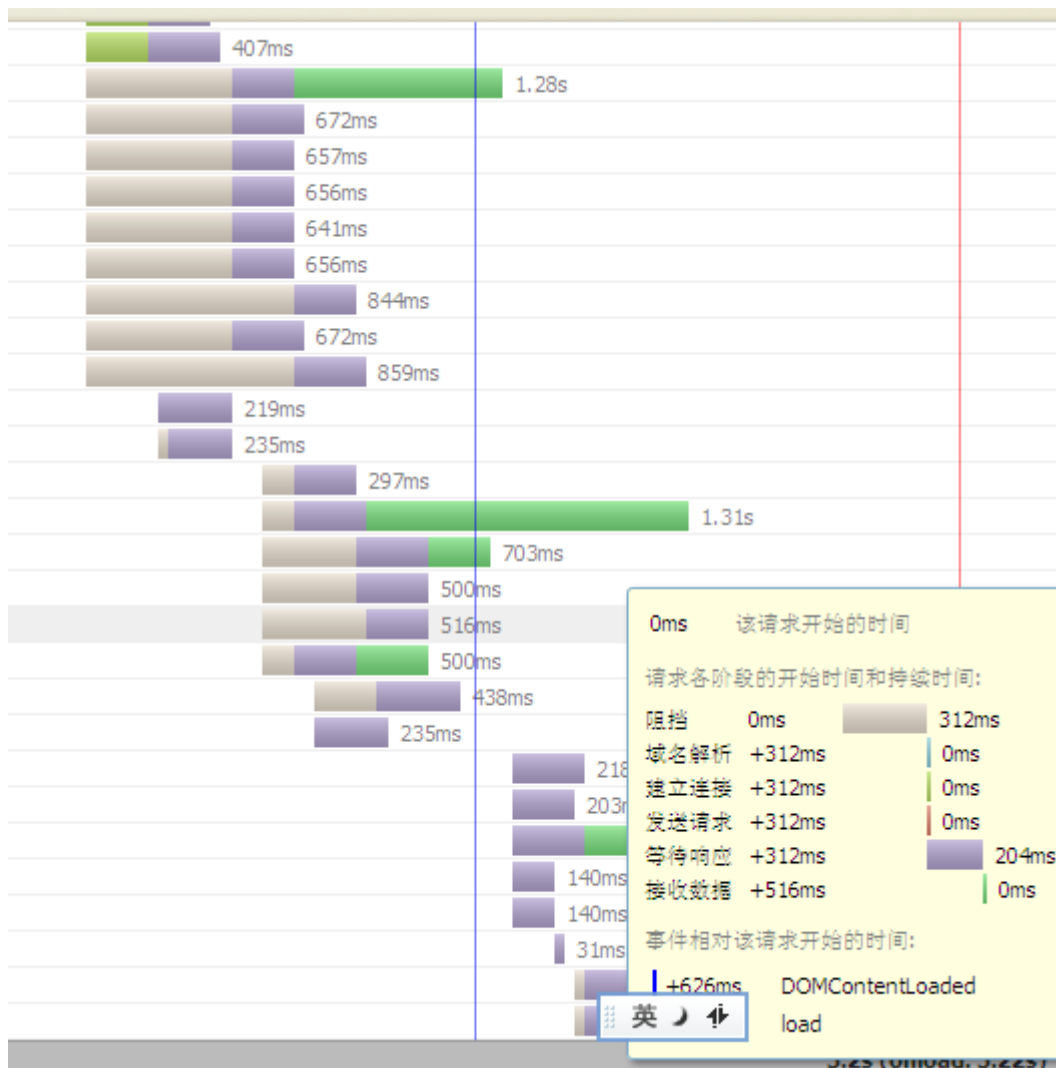
The table below shows the number of connections per server supported by current browsers for HTTP/1.1 as well as HTTP/1.0.

上面这张图，显示的是不同浏览器在不同的 http 协议上，并发请求限制个数。

Top Browsers 		score	Perf	Lim	ing	Connections per Hostname	Max Connections	Script Script	Script Stylesheet	Script Image	Script Iframe	Async Scripts	CSS CSS	CSS + Inline Script	Cache Expires	Cache Redirects	Cache Resource Redirects	Link Prefetch	data: URLs	Headers in trailer	# Tests
<input type="checkbox"/> Chrome 24 →	12/16	yes	6	9	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	no	478
<input type="checkbox"/> Firefox 18 →	13/16	yes	6	11	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	no	458
<input type="checkbox"/> IE 8 →	7/16	no	6	35	yes	yes	yes	no	no	no	yes	no	yes	no	yes	no	no	no	yes	no	3002
<input type="checkbox"/> IE 9 →	12/16	yes	6	35	yes	yes	yes	yes	no	no	yes	yes	yes	yes	yes	yes	yes	no	yes	no	1098
<input type="checkbox"/> IE 10 →	12/16	yes	8	16	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	no	271
<input type="checkbox"/> Opera 12.11 →	10/16	no	6	16	yes	yes	yes	yes	no	no	yes	no	yes	no	yes	yes	yes	no	yes	yes	53
<input type="checkbox"/> Safari 6.0.2 →	11/16	no	6	9	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	no	87
<input type="checkbox"/> Chrome 25 →	12/16	yes	6	9	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	no	545
<input type="checkbox"/> Chrome 26 →	12/16	yes	6	9	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	no	681
<input type="checkbox"/> Firefox 19 →	13/16	yes	6	14	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	no	460
<input type="checkbox"/> Firefox 20 →	11/16	yes	6	15	yes	yes	yes	yes	no	yes	no	no	yes	yes	yes	yes	yes	yes	yes	no	385
<input type="checkbox"/> Firefox 21 →	11/16	yes	6	16	yes	yes	yes	yes	no	yes	no	no	yes	yes	yes	yes	yes	yes	yes	no	503
<input type="checkbox"/> Opera 12.12 →	10/16	no	6	9	yes	yes	yes	yes	no	no	yes	no	yes	yes	yes	yes	yes	no	yes	yes	91
<input type="checkbox"/> Safari 6.0.3 →	11/16	no	6	16	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	yes	yes	no	yes	no	19
<input type="checkbox"/> Android 2.3 →	8/16	no	8	10	yes	yes	yes	yes	no	no	yes	no	yes	no	yes	no	yes	no	yes	no	169

来源: <http://www.browserscope.org/?category=network>

Ok。回到时间线上来。来一张明显点的图



请求时间线以瀑布的形式展现。时间线条的颜色四种，如下

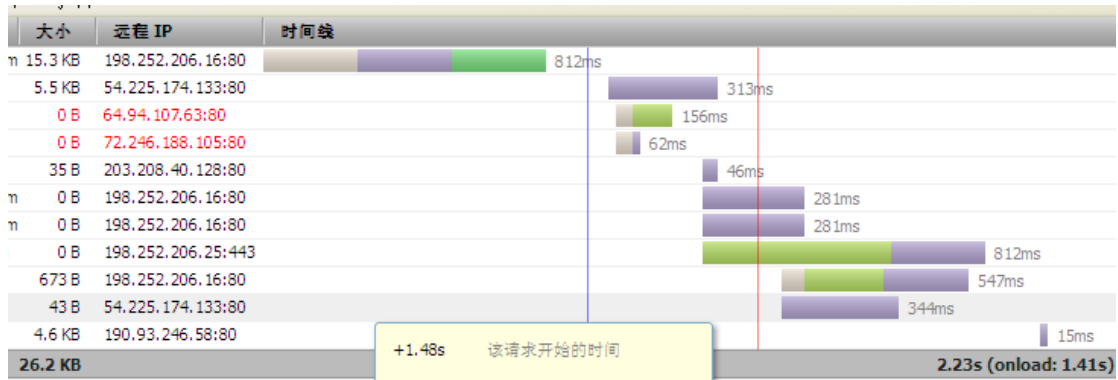
1. 淡蓝色，代表域名解析持续时间
2. 浅灰色，代表线程并发请求堵塞持续时间
3. 黄绿色，代表建立 TCP 连接所持续的时间
4. 浅褐色，代表发送请求所持续的时间
5. 浅紫色，代表等待服务器响应所持续的时间。
6. 草绿色，代表接收 Server 发回的资源所持续的时间

通常情况下 3 和 4 我们是看不到任何数据的，因为他们持续的时间太短了。Timeline 的单位是 ms，而 3, 4 的行为持续时间不在 ms 级别，应该在 us 级别。

我对颜色的感觉不准。大概描述成这个样子，具体参考我发的图把。

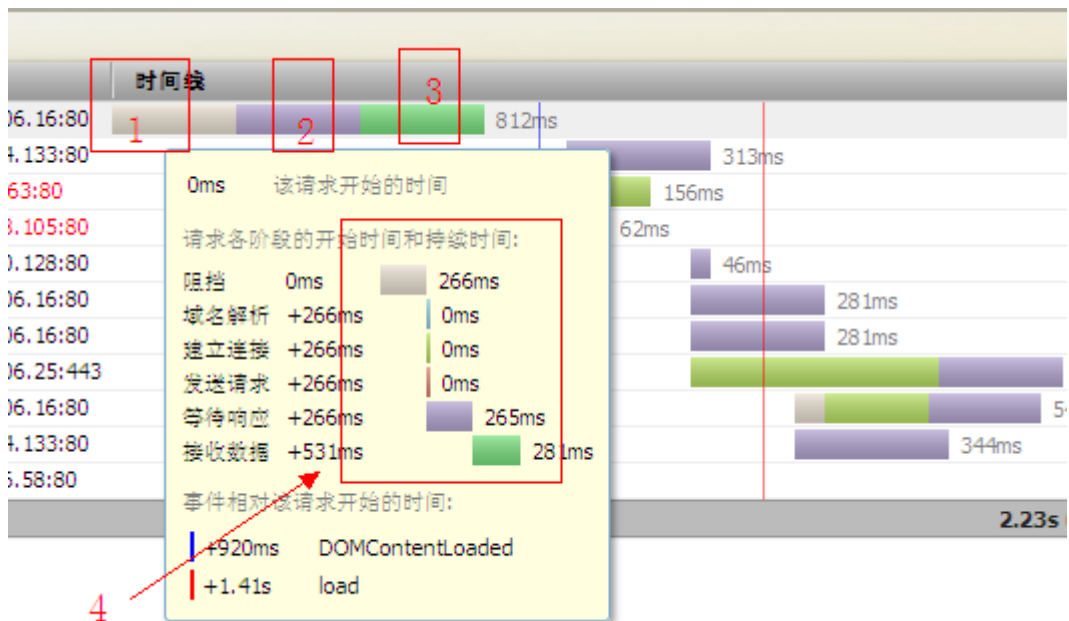
接下来，我们访问

<http://webmasters.stackexchange.com/questions/26753/why-do-big-sites-host-their-images-css-on-external-domains>



这个页面上的资源不多。拿这个来说下。

1. 光标移向第一个请求的 timeline, tip 如下



看下这个请求资源所处的域

域	大小	远程 IP	时间线
webmasters.stackexchange.com	15.3 KB	198.252.206.16:80	812ms

看 1234 处, 这 get 请求,

第一步, 首先就被堵塞了 266ms。为什么会被堵塞? 我已经说过。Client 和 Server 端都会对并发资源请求个数限制。我能解释的可能原因, 在这一时刻, 浏览器的中请求这个资源的线程因为某种原因不能立马正常的投入工作, 因为这个是一个新的请求, 浏览器内部需要初始化请求所需要的所有工作环境, 比如准备 Socket 资源, 初始化线程栈, 协议栈通信资源的分配等等)。因此才产生了一个阻挡时间, 这只是我的猜测。

第二步, 在被堵塞 266ms 以后, 域名解析开始, 时间极短, 0ms,

第三步, 接着打开 Socket 建立连接, 这个时间也极端, 因为工作环境都已经你给初始化

好了，到这里 TCP 连接建立起来

第四步，接着发送具体的资源请求，告诉 Server，要那个文件。这个时间也极短，0ms

第五步，浏览器请求线程处理等待状态，时间持续 265ms，为什么会有这个时间，Server 找到我们需要的文件，做对应的处理(比如 aspx，就会响应后台代码)，处理完毕，将信息全部变为二进制马，组装成通信报文，通过 Socket 写回

第六步，到这一步的时候，请求资源线程结束等待，在发起请求开始后的第 531ms 开始接收 Server 写回来的数据，接收数据的过程持续了 281ms。

进行到第六步的时候，整个资源请求就已经结束。

再看 tip 下面有一行小字，事件相对于该请求的开始时间。

DomContentLoaded +920ms

Loaded +1.41s

是什么意思呢，在本次页面资源的请求结束时，距离 Dom 加载结束还有 920ms，距离页面 Load 结束还有 1.41s

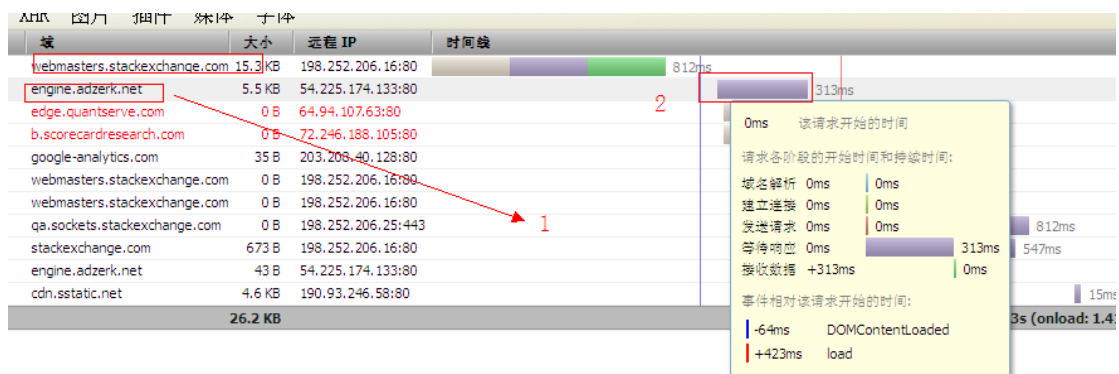
这里的两个值有可能是负数，

如果 DomContentLoaded 是负数，则表达，在本次资源请求结束的时候，Dom 加载已经结束了 X 毫秒或者 X 秒，

如果 Loaded 是负数，表达，在本次资源请求结束的时候，页面 Load 结束已经 X 毫秒或者 X 秒了。

2. 第一个页面请求已经结束了，接下来，浏览器的主解析引擎会解析页面，解析到页面上引入的外部资源，将会再次创建线程去请求所需要的资源。光标移到第二个请求时间线

看图

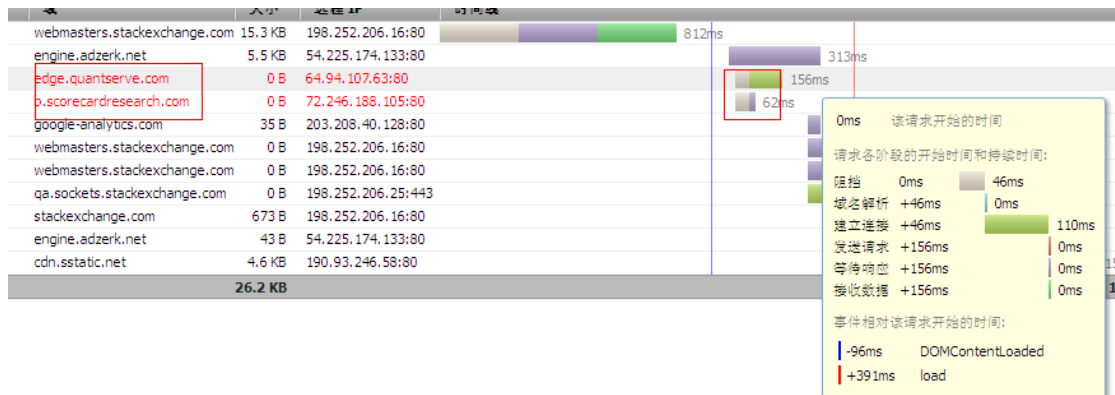


看 1 处，第二个资源请求跟第一个资源不在同一个域名下，ip 也不一样，那么这个时候，会跟前面的一样，重复以上的过程。再看 2 处，没有阻挡时间，为什么呢。我理解的是，可能是第一次请求开始以后，一些访问所需要的公共环境已经建立好了吧。我自己的猜测，如果有人知道，欢迎在此处更正。

第二个请求，在第一个请求之后，这很好理解。不多说。第二个请求也经过了 313ms 的等待，数据接收花的时间很端，us 级别

到第二个请求结束的时候，Dom 加载已经结束了 64ms，但是页面 load 还没结束，距离页面 load 还有 423ms

### 3. 鼠标移到第三个请求，tips 如下

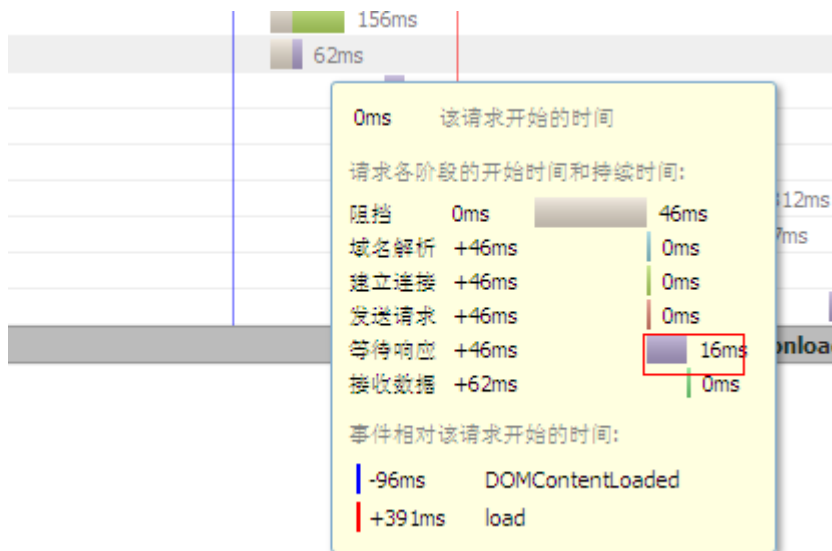


注意，时间线彩色条，注意到第三个和第四个的彩条是同时开始的。我的理解是这俩资源请求是同时发出的。虽然他们所在域不一样。

第三个和第四个资源请求均被堵塞了 46ms

第三个请求在花费了 110ms 建立连接 以后，Server 无响应，请求废弃，报红。

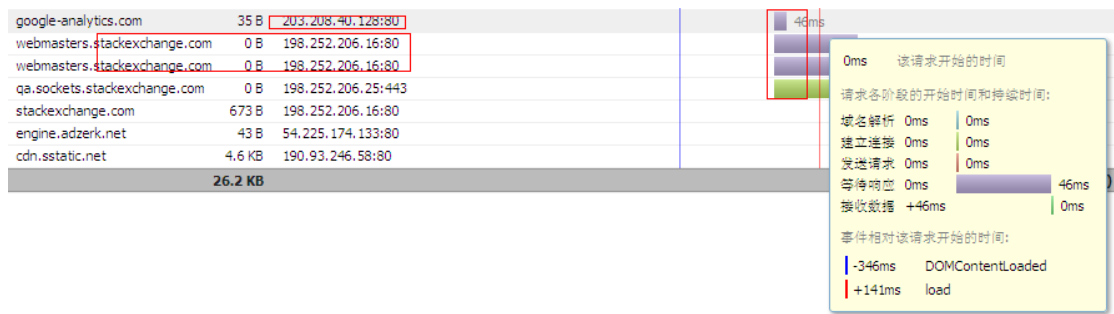
### 4. 鼠标移到第四个请求，看 tips



在建立连接并等待响应 16ms 以后，该请求开始接收数据，可能是网络堵塞也可能是 Server 无响应，请求废弃。爆红。



## 5. 看第五个请求

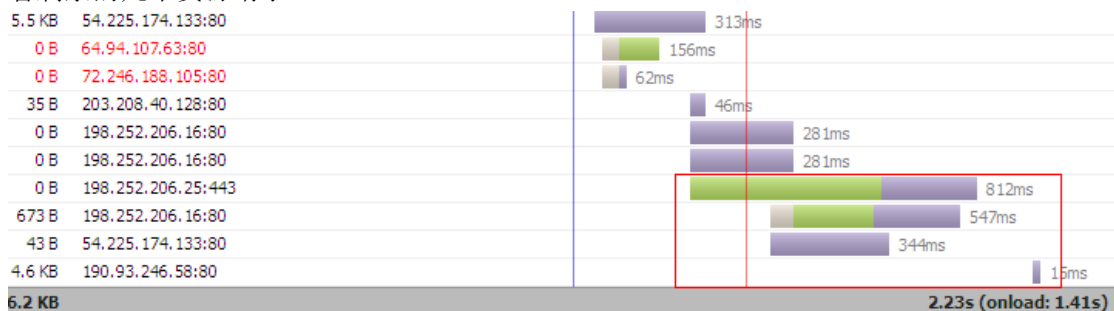


第五个请求开始有点意思了。四个时间线开始是同时的  
是不是意味着这几个资源的线程请求是同时发出的呢？我不敢确定。具体需要考证。  
现在，先暂且视他们是同时发出的吧

第六个和第七个请求均等待了 281ms，Server 端响应，但是未写回任何内容，所以看到了这样



## 6. 看剩余的几个资源请求



仍然是瀑布图。  
经过了请求，并响应成功。

时间线，还有几处可以读到的消息，如下





看 1,2,3 处。打开一个页面，页面的资源部署在 7 个域名下面，对应 7 个 ip  
3 处的两个请求，接收数据持续时间很长，数据量过大

从知乎和 stackoverflow 来看。

用的技术有，CDN 加速，静态资源压缩分离，资源分开部署

奇怪的是知乎那张图，有一个 101 切换协议，我也不知道是啥意思。

再来看腾讯的两个张图

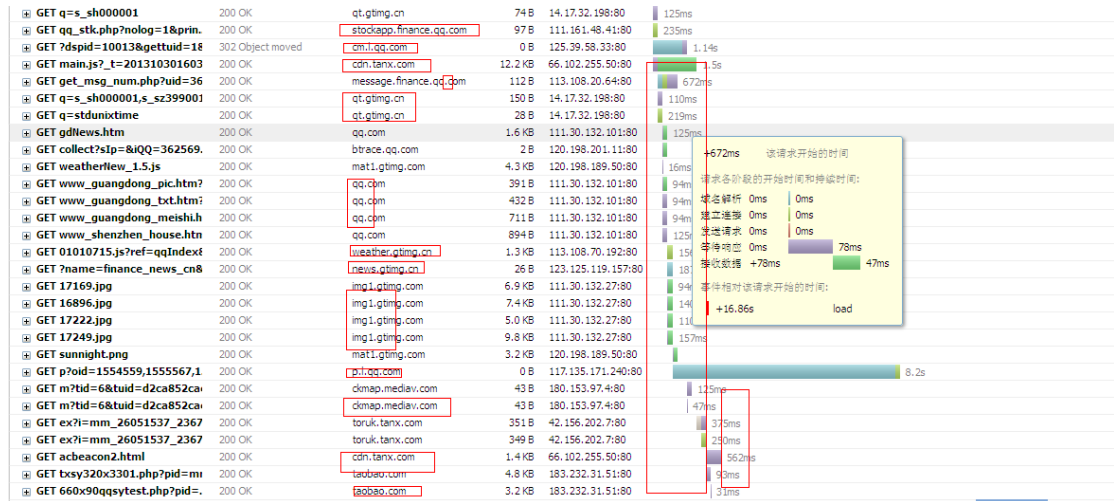


图 1

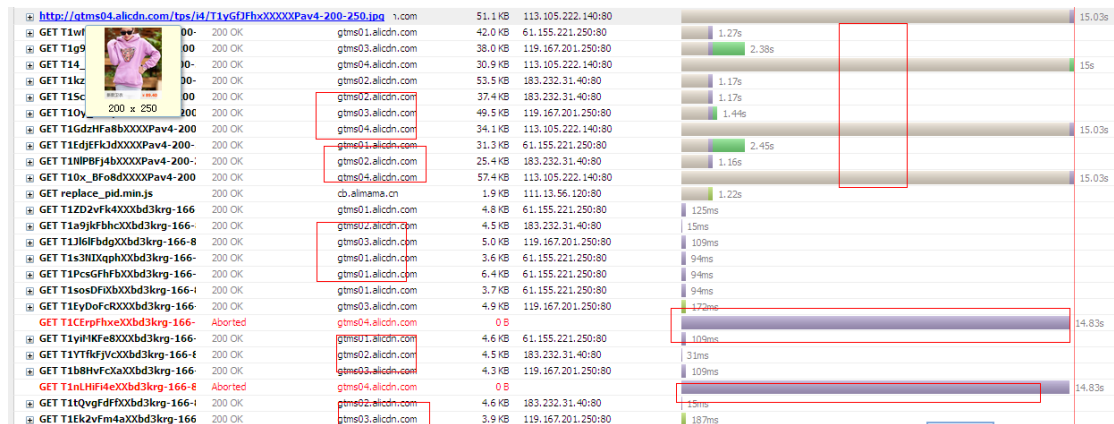


图 2

从腾讯的图可以看出，腾讯也用了 cdn 加速，资源分离和压缩，分布部署技术。  
而且更直观的是，腾讯用的是阿里巴巴的 CDN 加速服务。



图 3



上面说的都是前台的技术方案，后台的也有不少。其中很多技术解决方案，公司到目前为止可能还没有涉及到，但是这是未来必然要有所突破的方向。

另外，说道阿里，听不少圈内的鸟儿说，阿里的技术壁垒是国内几个大佬中比较低的。开源的东西比较多，阿里自己在 [github](#) 上开源很多技术解决方案，后台以 `java` 平台居多，前台的也有不少。提交更新还有 `issues` 跟踪挺活跃的。

消息队列 `RocketMQ`，在双 11 过后的第二天，听他们内部的人透露说。最大消息承载量开源达到百万级别，多节点部署，可以达到亿级别。是否属实不清楚，没研究过。

里面也有 `TFS`，`LVS` 等淘宝正在用的解决方案。在《淘宝技术这 10 年》提到过。想不到开源了。

还有很多 `Nginx` 相关的开源组件和中间件。

阿里官网的 `Github`，=>[alibaba](#)

无论是前台还是后台技术，有精力和时间，功底也够的，可以看源码学习。

`TFS`，`LVS` 这些牛逼方案的产生，凝结了阿里 N 多大犇的心血和智慧。

阅读大神写的代码，一定是技术成长中的必经之路。若能从这些大犇身上学得几成，技术上来讲足以应付很多百万级的 `Project`。那些高级解决方案，会成为项目规模的瓶颈。也会成为高可靠性产品问世的瓶颈。

关于 `timeline` 时间线看 `Dom` 资源加载，到此结束。

有不正确的欢迎纠正，欢迎各路大神吐槽指点，不懂的也欢迎私下来找我。:))))

周末愉快、

2013-11-16