

03 | 换个角度解决问题：服务端推送技术

2019-09-16 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 16:50 大小 11.57M



你好，我是四火。

今天我们继续和 HTTP “过不去”。在上一讲，我们讲到了 HTTP 在安全传输方面的局限，并介绍了怎样使用经过 TLS 加密的 HTTPS 连接来解决这样的弊端。

今天，我要给你讲讲传统 HTTP 的另一个在交互模式上的局限，就是只能由客户端主动发起消息传递，而服务端只能被动响应消息的局限，并介绍它的解决办法。

Pull 模型的问题

让我们来思考这样一个场景，假设你设计了一款网页版的即时聊天工具，现在你使用浏览器打开了聊天页面，正在和朋友愉快地聊天。这时有朋友给你发送了一条消息，可是由于

HTTP 本身机制的限制，服务端无法主动推送消息，告知浏览器上的聊天页面“你有一条消息”，进而影响到了消息的即时送达。那么，这个问题怎么解决？

你可能会立即想到**轮询 (Poll)**，比如浏览器每隔十秒钟去问一下服务端是不是有新消息不就完了嘛。这看起来是个好思路，但明显存在这样两个问题：

消息还是不够即时。换言之，假如正好在某次询问之后服务器收到了消息，那么这条消息的获取延迟可能达到至少十秒。

大量的请求 - 响应，带宽和服务器资源浪费。如果你开着聊天工具页面一个小时，除了这一条消息，却没有进一步的聊天行为，于是按照每十秒发送一次请求计算，一共发起了 360 次请求，而其中居然只有 1 次返回了聊天消息是有实际意义的。

显然，轮询这个方案不好。说到底，其实我们并没有抛开对 HTTP 的已有印象，从问题本身出发去思考解决问题的最佳方式，而是潜意识地受限于 HTTP 的传统交互模式，考虑其中的变通方法。

在进一步分析之前，我们先来看两个容易弄混的概念：Pull 和 Poll。

“Pull”指的是去主动发起行为获取消息，一般在客户端 / 服务器 (C/S, Client/Server) 或浏览器 / 服务器 (B/S, Browser/Server) 交互中，客户端或浏览器主动发起的网络请求数据的行为。

而“Poll”，尽管在某些场景下也和 Pull 通用了，但在计算机网络的领域里，通常把它解释为“轮询”，或者“周期性查询”，在 Pull 的基础上增加了“周期性”的概念，这也是它和 Pull 相比最本质的区别。

相应地，和 Pull 行为相对的，从服务端主动发起，发送数据到客户端的行为叫做“Push”。Push 相比 Pull 而言，具备这样两个明显的优势：

高效性。如果没有更新发生，就不会有任何更新消息推送的动作，即每次消息推送都发生在确确实实的更新事件之后，都是有意义的，不会出现请求和响应的资源浪费。

实时性。事件发生后的第一时间即可触发通知操作，理论上不存在任何可能导致通知延迟的硬伤。

可是，有趣的是，事实上 Pull 的应用却远比 Push 更广泛，特别是在分布式系统中。这里有多条原因，其中很重要的一条是：

服务端不需要维护客户端的列表，不需要知晓客户端的情况，不需要了解客户端查询的策略。**这有助于把服务端从对客户端繁重的管理工作中解放出来，而成为无状态的简单服务，变得具备幂等性（idempotent，指执行多次和执行一次的结果一样），更容易横向扩展。**

尤其在分布式系统中，状态经常成为毒药，有了状态，就不得不考虑状态的保存、丢失、一致性问题，因此这种无状态往往可以很大程度地简化系统的设计。

服务端推送技术

有了这些基础知识，我们就可以来谈谈实际的服务端推送技术了，这些都从一定程度上解决了 HTTP 传统方式 Pull 的弊端。

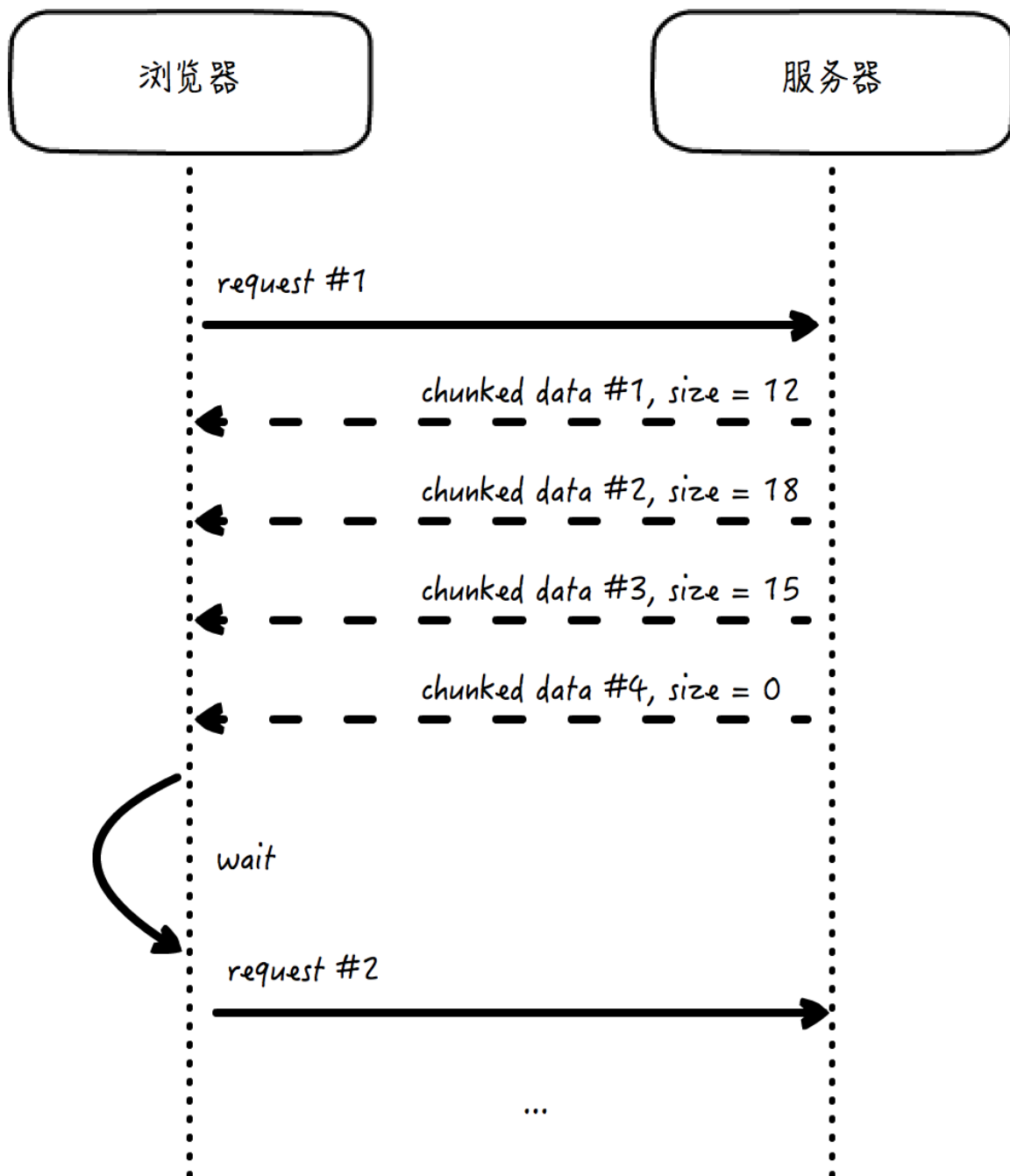
1. Comet

严格说，Comet 是一种 Web 应用客户端和服务端交互的模型，它有几种服务端推送的具体实现，但是，它们的大致原理是一样的：**客户端发送一个普通的 HTTP 请求到服务端以后，服务端不像以往一样在处理后立即返回数据，而是保持住连接不释放，每当有更新事件发生，就使用分块传输的方式返回数据**（如果你忘记了块传输的方式，请回看 [\[第 1 讲\]](#)）。

若干次数据返回以后可以完成此次请求响应过程（分块传输返回长度为 0 的块，表示传输结束），等待客户端下一次请求发送。这种过程看起来也属于轮询，但是每个周期可包含多次服务端数据返回，因而也被形象地称为“长轮询”（Long Polling）。

在服务端推送技术中，Comet 最大的好处是，它 100% 由 HTTP 协议实现，当然，分块传输要求 HTTP 至少是 1.1 版本。但也正因为这点，它也存在一些弊端，比如，客户端必须在服务端结束当次传输后才能向服务端发送消息；HTTP 协议限制了它在每次请求和响应中必须携带完整的头部，这在一定程度上也造成了浪费（这种为了传输实际数据而使用的额外开销叫做 overhead）。

下面我给出了一个 Comet 实现的示例图。浏览器在发出 1 号请求要求数据，连接保持，接着陆续收到几个不同大小的响应数据，并且最后一个大小为 0，浏览器被告知此次传输完成。过了一会儿，浏览器又发出 2 号请求，开始第二轮的类似交互。



在 Comet 方式下，**看起来服务端有了推送行为，其实只是对于客户端请求有条件、讲时机的多次返回**，因此我们把它称为服务端“假 Push”。

2. WebSocket

HTML 5 规范定义了 WebSocket 协议，它可以通过 HTTP 的端口（或者 HTTPS 的端口）来完成，从而最大程度上对 HTTP 协议通透的防火墙保持友好。但是，**它是真正的双向、全双工协议，也就是说，客户端和服务端都可以主动发起请求，回复响应，而且两边的传输都互相独立。**

和上文的 Comet 不同，WebSocket 的服务端推送是完全可以由服务端独立、主动发起的，因此它是服务端的“真 Push”。

WebSocket 是一个可谓“科班出身”的二进制协议，也没有那么大的头部开销，因此它的传输效率更高。同时，和 HTTP 不一样的是，它是一个带有状态的协议，双方可以约定好一些状态，而不用在传输的过程中带来带去。而且，WebSocket 相比于 HTTP，它没有同源的限制，服务端的地址可以完全和源页面地址无关，即不会出现臭名昭著的浏览器“跨域问题”。

另外，它和我们之前学习的加密传输也丝毫不冲突，由于它在网络分层模型中位于 TLS 上方，因此他可以使用和 HTTP 一样的加密方式传输：

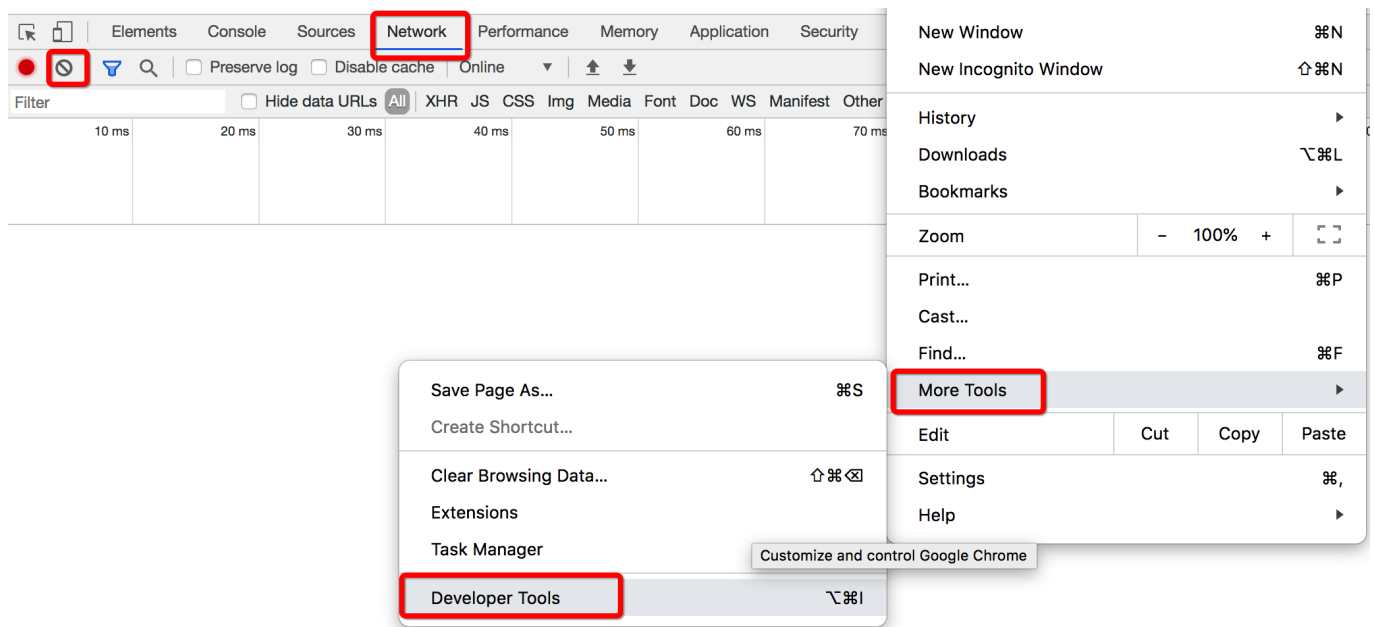
HTTP → WS

HTTPS → WSS


最后，最有意思的事情在于，和我们之前的认识不同，WebSocket 是使用 HTTP 协议“升级”的方法来帮助建立连接的，下面我们动手来试一试。

首先，我们需要找到一个可以支持 WebSocket 测试的网站，比如 websocket.org，然后我们将使用 Chrome 的网络工具来捕获和显示通过浏览器发送和接收的消息。如果这是你第一次使用 Chrome 的开发者工具，那么你需要好好熟悉它了，因为它将在你今后全栈的道路上派上大用场。

使用 Chrome 打开 [Echo Test](#) 页面，在这里你可以发送建立一个 WebSocket 连接。但是别急，我们先打开 Chrome 的开发者工具，并选中 Network 标签，接着点击左上角的清除按钮，把已有页面加载的网络消息清除掉，以获得一个清爽的网络报文监视界面：




接着，确保页面上建立 WebSocket 连接的对端地址和传递的信息都已经填写，比如：

 复制代码

```
1 Location:
2 wss://echo.websocket.org
3 Message:
4 Rock it with HTML5 WebSocket
```


于是就可以点击 “Connect” 按钮了，旁边的日志框将出现 “CONNECTED” 字样，同时，Chrome 开发者工具将捕获这样的请求（如果在开发者工具中网络监视界面上，选中消息的消息头处于 “parsed” 展示模式，你需要点击 Request Headers 右侧的 “view source” 链接来查看原始消息头）：

 复制代码

```
1 GET wss://echo.websocket.org/?encoding=text HTTP/1.1
2 Host: echo.websocket.org
3 Origin: https://www.websocket.org
4 Connection: Upgrade
5 Upgrade: websocket
6 Sec-WebSocket-Version: 13
7 Sec-WebSocket-Key: xxx
8 ... (省略其它 HTTP 头)
```

好，你可以看到，这是一个普普通通的 HTTP GET 请求，但是 URL 是以加密连接 “wss” 开头的，并且有几个特殊的 HTTP 头：Origin 指出了请求是从哪个页面发起的，Connection: Upgrade 和 Upgrade: websocket 这两个表示客户端要求升级 HTTP 协议为 WebSocket。

好，再来看响应，消息的头部为：

 复制代码

```
1 HTTP/1.1 101 Web Socket Protocol Handshake
2 Connection: Upgrade
3 Sec-WebSocket-Accept: xxx
4 Upgrade: websocket
5 ... (省略其它 HTTP 头)
```

嗯，返回码是 101，描述是 “Web Socket Protocol Handshake”，并且，它确认了连接升级为 “websocket” 的事实。

3. 更多推送技术

到这里，我已经介绍了几种服务端的推送技术，事实上还有更多，但是，**如果你依次了解以后认真思考，就会发现，这些原理居然都在某种程度上和我介绍的 Comet 和 WebSocket 这两种类似，有的甚至来自于它们。**

这些技术包括：

SSE，即 Server-Sent Events，又叫 EventSource，是一种已被写入 HTML 5 标准的服务端事件推送技术，它允许客户端和服务端之间建立一个单向通道，以让服务端向客户端单方向持续推送事件消息；

为了提高性能，HTTP/2 规范中新添加的服务端推送机制，我们在 [\[第 01 讲\]](#) 中提到过，并在该讲的扩展阅读中有它的原理介绍；

WebRTC，即 Web Real-Time Communication，它是一个支持网页进行视频、语音通信的协议标准，不久前已被加入 W3C 标准，最新的 Chrome 和 Firefox 等主流浏览器都支持；

还有一些利用浏览器插件和扩展达成的服务端推送技术，比如使用 Flash 的 XMLSocket，比如使用 Java 的 Applet，但这些随着 HTML 5 的普及，正慢慢被淘汰。

你看，通过学习一两个典型的技术，再拓展开，去类比了解和分析思考同一领域内的其它技术，就能掌握到最核心的东西，这就是我推荐的一种学习全栈技术的方式。

总结思考

今天我们从 HTTP 的交互局限性引出了网络交互中 Pull 和 Push 的两大模型，比较了它们的优劣。服务端 Push 的方式具备高效性和实时性的优势，而客户端 Pull 的方式令服务端免去状态的维护，从根本上简化了系统。

之后我们以 Comet 和 WebSocket 为重点，介绍了服务端推送的不同方式，尤其是用了实际抓包分析，介绍了通过 HTTP “升级” 的方式来建立 WebSocket 连接的原理。

今天学习得怎样呢？来看这样两个问题：

文中介绍了 Push 和 Pull 在原理上的不同，在你的实际项目中，是否应用了 Push 或 Pull 的模型呢？

文中介绍了 Push 比 Pull 具备高效性和实时性的优势，而 Pull 比 Push 则具备使得服务变得无状态的优势，除了最重要的这几个，你还能说出更多它们各自的优势吗？

今天的内容就到这里。以 HTTP 协议为核心，介绍网络协议的三讲文章已经更新完毕了，你是否对于全栈技术本身，还有适合自己的学习方法，有了新的理解呢？欢迎留言和我讨论。

扩展阅读

文中提到了跨域问题，如果感兴趣，推荐你阅读 MDN 的 [HTTP 访问控制 \(CORS\)](#) 这篇文章。

TutorialsPoint 的 [WebSocket 系统教程](#)，对于本文介绍的 WebSocket 协议，需要进一步了解的一个好去处。

关于 HTTP Update 头的 [RFC 2616 协议片段](#)和 WebSocket 的 [RFC 6445](#)，你也许对响应和请求中的其它 HTTP 头心存疑问，和之前介绍的 HTTP 的 RFC 协议一样，你通常不需要仔细阅读，但它是对协议有问题时的最终去处。

[Stream Updates with Server-Sent Events](#)，一篇非常好的介绍 SSE 基础，和同类技术比较优劣，并给出代码示例的文章；如果你对 WebRTC 感兴趣，那么可以先看看这个[胶片](#)，再阅读这篇基础知识 [Getting Started with WebRTC](#)。

全栈工程师修炼指南

从全栈入门到技能实战

熊燚

Oracle 首席软件工程师



新版升级：点击「🔗 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 02 | 为HTTP穿上盔甲：HTTPS

精选留言 (3)

写留言



许童童

2019-09-16

四火老师讲得真好，服务端推送模型最后都可以看作是Comet和WebSocket的变形。



饭团

2019-09-16

1)公司一个简单的聊天业务系统！就用了push模型！
2)push模型处理信息比poll稍微复杂一些！存在解包封包！需要定一套消息结构！
展开 ∨



許敲敲

2019-09-16

谢谢老师给这么多资料

展开 ✓

