

01 | 网络互联的昨天、今天和明天：HTTP 协议的演化

2019-09-11 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 16:33 大小 15.16M

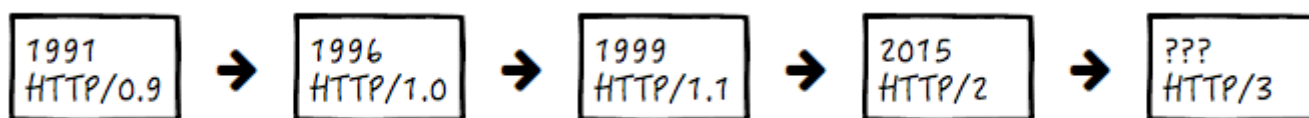


你好，我是四火。

HTTP 协议是互联网基础中的基础，和很多技术谈具体应用场景不同的是，几乎所有的互联网服务都是它的应用，没有它，互联网的“互联”将无从谈起，因此我们把它作为正式学习的开篇。

说到其原理和协议本身，我相信大多数人都能说出个大概来，比如，有哪些常见的方法，常见 HTTP 头，返回码的含义等等。但你是否想过，这个古老而富有生命力的互联网“基石”是怎样发展演化过来的呢？从它身上，我们能否管中窥豹，一叶知秋，找到互联网成长和演进的影子？


今天，我想带你从实践的角度，亲身感受下这个过程，相信除了 HTTP 本身，你还可以发现网络协议发展过程中的一些通用和具有共性的东西。



HTTP/0.9


和很多其它协议一样，1991 年，HTTP 在最开始的 0.9 版就定义了协议最核心的内容，虽说从功能上看只是具备了如今内容的一个小小的子集。比如，确定了客户端、服务端的这种基本结构，使用域名 /IP 加端口号来确定目标地址的方式，还有换行回车作为基本的分隔符。

它非常简单，不支持请求正文，不支持除了 GET 以外的其它方法，不支持头部，甚至没有版本号的显式指定，而且整个请求只有一行，因而也被称为 “The One-line Protocol”。比如：

 复制代码


```
1 GET /target.html
```

虽说 0.9 版本如今已经极少见到了，但幸运的是 Google 还依然支持（Bing 和 Baidu 不支持）。我们不妨自己动手，实践一下！虽然不能使用浏览器，但别忘了，我们还有一个更古老的工具 telnet。在命令行下建立连接：

 复制代码

```
1 telnet www.google.com 80
```


你会看到类似这样的提示：

 复制代码

```
1 Trying 2607:f8b0:400a:803::2004...
2 Connected to www.google.com.
3 Escape character is '^]'.

```


好，现在输入以下请求：

 复制代码

```
1 GET /
```

(请注意这里没有版本号，并不代表 HTTP 协议没有版本号，而是 0.9 版本的协议定义请求中就是不带有版本号，这其实是该版本的一个缺陷)

接着，你会看到 Google 把首页 HTML 返回了：

 复制代码

```
1 HTTP/1.0 200 OK
2 ... (此处省略多行 HTTP 头)
3
4 ... (此处省略正文)
```

HTTP/1.0

到了 1996 年，HTTP 1.0 版本就稳定而成熟了，也是如今浏览器广泛支持的最低版本 HTTP 协议。引入了返回码，引入了 header，引入了多字符集，也终于支持多行请求了。

当然，它的问题也还有很多，支持的特性也远没有后来的 1.1 版本多样。比如，方法只支持 GET、HEAD、POST 这几个。但是，麻雀虽小五脏俱全，这是第一个具备广泛实际应用价值的协议版本。

你一样可以用和前面类似的方法来亲自动手实践一下，不过，HTTP 1.0 因为支持多行文本的请求，单纯使用 telnet 已经无法很好地一次发送它们了，其中一个解决办法就是使用 [netcat](#)。

好，我们先手写一份 HTTP/1.0 的多行请求，并保存到一个文件 request.txt 中：

```
1 GET / HTTP/1.0
2 User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 3.1)
3 Accept: text/html
4
```

(根据协议，无论请求还是响应，在 HTTP 的头部结束后，必须增加一个额外的换行回车，因此上述代码最后这个空行是必须的，如果是 POST 请求，那么通常在这个空行之后会有正文)

你看上面的 User-Agent，我写入了一个[假的浏览器和操作系统版本](#)，假装我穿越来自 Window 3.1 的年代，并且用的是 IE 2.0，这样一来，我想不会有人比我更“老”了吧。

好，接着用类似的方法，使用 netcat 来发送这个请求：

```
1 netcat www.google.com 80 < ~/Downloads/request.txt
```

一样从 Google 收到了成功的报文。

不知这样的几次动手是否能给你一个启示：懂一点特定的协议，使用简单的命令行和文本编辑工具，我们就已经可以做很多事情了。比如上面这样改变 UA 头的办法，可以模拟不同的浏览器，就是用来分析浏览器适配（指根据不同浏览器的兼容性返回不同的页面数据）的常用方法。

HTTP/1.1

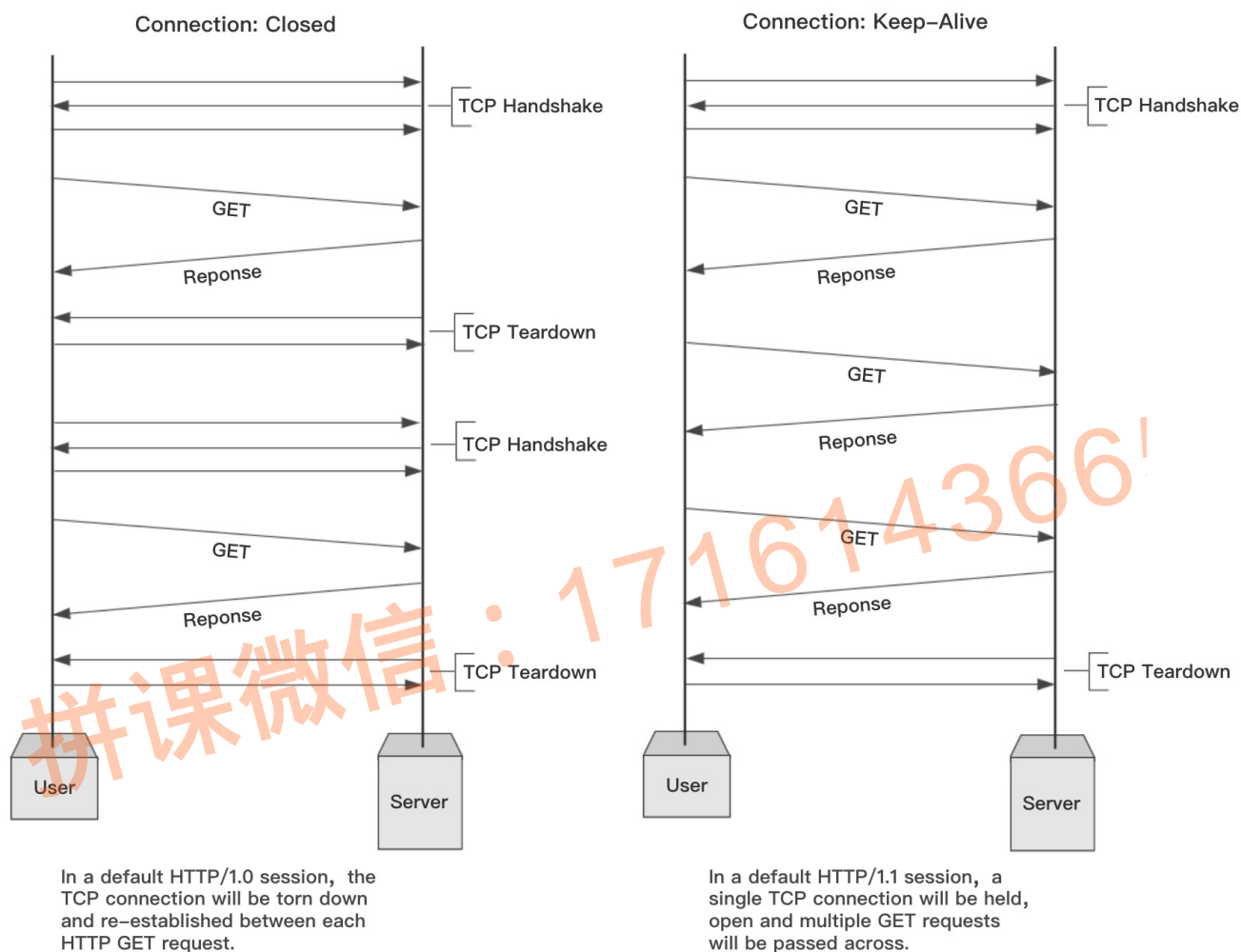
1999 年，著名的 RFC2616，在 1.0 的基础上，大量帮助传输效率提升的特性被加入。

你可能知道，从网络协议分层上看，TCP 协议在 HTTP 协议的下方（TCP 是在 OSI 7 层协议的第 4 层，而 HTTP 则是在最高的第 7 层应用层，因此，前者更加“底层”一点）。

在 HTTP 1.0 版本时，每一组请求和响应的交互，都要完成一次 TCP 的连接和关闭操作，这在曾经的互联网资源比较贫瘠的时候并没有暴露出多大的问题，但随着互联网的迅速发展

展，这种通讯模式显然过于低效了。

于是这个问题的解决方案——HTTP 的长连接，就自然而然地出现了，它指的是打开一次 TCP 连接，可以被连续几次报文传输重用，这样一来，我们就不需要给每次请求和响应都创建专门的连接了：



(上图来自 [Evolution of HTTP — HTTP/0.9, HTTP/1.0, HTTP/1.1, Keep-Alive, Upgrade, and HTTPS](#))

可以看到，通过建立长连接，中间的几次 TCP 连接开始和结束的握手都省掉了。

那好，我们还是使用 netcat，这次把版本号改成 1.1，同时打开长连接：


复制代码

```
1 GET / HTTP/1.1
2 Host: www.google.com
3 User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 3.1)
4 Connection: keep-alive
```

```
5 Accept: text/html
6
```


(别忘了上面那个空行)

相信你也注意到了上面客户端要求开启长连接的 HTTP 头：

 复制代码


```
1 Connection: keep-alive
```

再按老办法运行：

 复制代码

```
1 netcat www.google.com 80 < ~/Downloads/request.txt
```


我们果然得到了 Google 的响应：

 复制代码

```
1 HTTP/1.1 200 OK
2 Date: ...
3 Expires: -1
4 Cache-Control: private, max-age=0
5 Content-Type: text/html; charset=ISO-8859-1
6 Transfer-Encoding: chunked
7 ... (此处省略多行 HTTP 头)
8
9 127a
10 ... (此处省略 HTML)
11 0
12
```


但是在响应中，值得注意的有两点：

1. 在 HTTP 头部，有这样一行：

 复制代码

```
1 Transfer-Encoding: chunked
```

1. 正文的内容是这样的：

 复制代码

```
1 127a
2 ...
3 0
4
```

同时，之前我们见到过头部的 Content-Length 不见了。这是怎么回事呢？

事实上，如果协议头中存在上述的 chunked 头，表示将采用分块传输编码，响应的消息将由若干个块分次传输，而不是一次传回。刚才的 127a，指的是接下去这一块的大小，在这些有意义的块传输完毕后，会紧跟上一个长度为 0 的块和一个空行，表示传输结束了，这也是最后的那个 0 的含义。

值得注意的是，实际上在这个 0 之后，协议还允许放一些额外的信息，这部分会被称作 **“Trailer”**，这个额外的信息可以是用来校验正确性的 checksum，可以是数字签名，或者传输完成的状态等等。

在长连接开启的情况下，使用 Content-Length 还是 chunked 头，必须具备其中一种。**分块传输编码大大地提高了 HTTP 交互的灵活性**，服务端可以在还不知道最终将传递多少数据的时候，就可以一块一块将数据传回来。在 [第 03 讲] 中，你还会看到藉由分块传输，可以实现一些模拟服务端推送的技术，比如 [Comet](#)。

事实上 HTTP/1.1 还增加了很多其它的特性，比如更全面的方法，以及更全面的返回码，对指定客户端缓存策略的支持，对 content negotiation 的支持（即通过客户端请求的以 Accept 开头的头部来告知服务端它能接受的内容类型），等等。

HTTP/2

现在最广泛使用的 HTTP 协议还是 1.1，但是 HTTP/2 已经提出，在保持兼容性的基础上，包含了这样几个重要改进：

设计了一种机制，允许客户端来选择使用的 HTTP 版本，这个机制被命名为 ALPN；

HTTP 头的压缩，在 HTTP/2 以前，HTTP 正文支持多种方式的压缩，但是 HTTP 头部却不能；

多路复用，允许客户端同时在一个连接中同时传输多组请求响应的方法；

服务端的 push 机制，比方说客户端去获取一个网页的时候，下载网页，分析网页内容，得知还需要一个 js 文件和一个 css 文件，于是再分别下载，而服务端的 push 机制可以提前就把这些资源推送到客户端，而不需要客户端来索取，从而节约网页加载总时间。

在 HTTP/2 之后，我们展望未来，HTTP/3 已经箭在弦上。如同前面的版本更新一样，依旧围绕传输效率这个协议核心来做进一步改进，其承载协议将从 TCP 转移到基于 UDP 的 [QUIC](#) 上面来。

最后，我想说的是，**HTTP 协议的进化史，恰恰是互联网进化史的一个绝佳缩影**，从中你可以看到互联网发展的数个特质。比方说，长连接和分块传输很大程度上增强了 HTTP 交互模型上的灵活性，使得 B/S 架构下的消息即时推送成为可能。

总结思考

今天我们了解了 HTTP 协议的进化史，并且用了动手操作的方法来帮助你理解内容，还分析了其中两个重要的特性，长连接和分块传输。希望经过今天的实践，除了知识本身的学习，你还能够在**快速的动手验证中，强化自己的主观认识，并将这种学习知识的方式培养成一种习惯，这是学习全栈技能的一大法宝。**

现在，让我们来进一步思考这样两个问题：

文中介绍了分块传输的 HTTP 特性，你觉得它可以应用到哪些具体场景？

如果让你去设计一个新的网络协议，你能否举例设计的过程中需要遵循哪些原则？


好，今天的分享就到这里，欢迎提出你的疑问，也期待你留言与我交流！

选修课堂：抓一段 HTTP 的包

如果你对于使用 tcpdump 进行网络抓包这个技能已经了解了，就可以跳过下面的内容。反之，推荐你动手。因为在学习任何网络协议的时候，网络抓包是一个非常基本的实践前置技能；而在实际定位问题的时候，也时不时需要抓包分析。这也是我在第一讲就放上这堂选修课的原因。


俗话说，耳听为虚，眼见为实，下面让我们继续动手实践。你当然可以尝试抓访问某个网站的包，但也可以在本机自己启动一个 web 服务，抓一段 HTTP GET 请求的报文。

利用 Python，在任意目录，一行命令就可以在端口 8080 上启动一个完备的 HTTP 服务（这大概是世界上最简单的启动一个 HTTP 服务的方式了）：

 复制代码


```
1 python -m SimpleHTTPServer 8080
```

启动成功后，你应该能看到：

 复制代码

```
1 Serving HTTP on 0.0.0.0 port 8080 ...
```


接着使用 tcpdump 来抓包，注意抓的是 loopback 的包（本地发送到本地），因此执行：

 复制代码

```
1 sudo tcpdump -i lo0 -v 'port 8080' -w http.cap
```

这里的 -i 参数表示指定 interface，而因为客户端和服务端都在本地，因此使用 lo0（我使用的是 Mac，在某些 Linux 操作系统下可能是 lo，具体可以通过 ifconfig 查看）指定 loopback 的接口，这里我们只想捕获发往 8080 端口的数据包，结果汇总成 http.cap 文件。

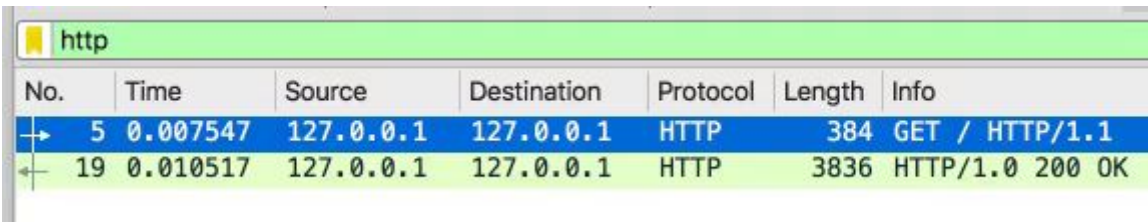
打开浏览器敲入 <http://localhost:8080> 并回车，应该能看到启动 HTTP 服务路径下的文件（夹）列表。这时候你也应该能看到类似下面这样的文字，标志着多少包被捕获，多少包被过滤掉了：

 复制代码

```
1 24 packets captured
2 232 packets received by filter
```

好，现在我们使用 CTRL + C 结束这个抓包过程。

抓包后使用 [Wireshark](#) 打开该 http.cap 文件，在 filter 里面输入 http 以过滤掉别的我们不关心的数据包，我们应该能看到请求和响应至少两条数据。于是接下去的内容就是我们非常关心的了。



No.	Time	Source	Destination	Protocol	Length	Info
5	0.007547	127.0.0.1	127.0.0.1	HTTP	384	GET / HTTP/1.1
19	0.010517	127.0.0.1	127.0.0.1	HTTP	3836	HTTP/1.0 200 OK

如果你看到这里，我想请你再思考下，在不设置上面的 http filter 的时候，我们会看到比这多得多的报文，它们不是 HTTP 的请求响应所以才被过滤掉了，那么，它们都有什么呢？

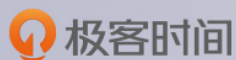
扩展阅读

【基础】如果你对 HTTP 还不熟悉的话，推荐你阅读一篇系统性介绍 HTTP 的教程，比如 [MDN 的这篇教程](#)。

【基础】[The OSI model explained: How to understand \(and remember\) the 7 layer network model](#)：如果你对网络的 OSI 7 层模型还不清楚的话，建议阅读。如果你想知道那些鼎鼎大名的网络协议在这个模型中的哪个位置，那么请从 [List of network protocols \(OSI model\)](#) 里面找。基于聚焦主题的关系，我们在这个专栏中不会详细介绍呈现层（Presentation Layer）之下的网络协议。

HTTP [1.0](#)、[1.1](#) 和 [2.0](#)：它们是 RFC 文档，看起来似乎枯燥乏味，通常我们不需要去仔细阅读它们，但是当我们想知道对协议的理解是否正确，它们是我们最终的参考依据。

[Key differences between HTTP 1.0 and HTTP 1.1](#): 文中总结了从 HTTP 1.0 到 1.1 的 9 大改进; 而 [HTTP/2 Complete Tutorial](#) 是一篇比较系统的 HTTP/2 的介绍。



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 导读 | 如何学习这个专栏?

下一篇 02 | 为HTTP穿上盔甲: HTTPS

精选留言 (14)

写留言



William 置顶

2019-09-11

【📝笔记-HTTP发展史】

- + HTTP/0.9 确立了C/S架构, 域名、IP、端口。换行回车作为基本分隔符。
- + HTTP/1.0 返回码、header、多字符集、多行请求支持
- + HTTP/1.1 长连接keep-alive。分块传输chunked。方法、返回码更全面, 缓存控制策略, content negotiation。...

展开 ∨

作者回复: 笔记部分:

优秀, 说得太棒了。做一点小小的说明: 文中说的客户端和服务端的概念完全是从网络和协议的角度来进行的, 和我们平时提到的选择 C/S 还是 B/S 的“应用模式”有所区别。HTTP 本身和选用 C/S (Client/Server) 还是 B/S (Browser/Server) 并没有必然关系。也就是说, 无论你使用客户端还是浏览器, 都可以使用 HTTP 方式和服务端交流。

思考题部分:

1. 分块传输和 Ajax 并没有直接联系, 也就是说, 分块传输可以使用、也可以不使用 Ajax 来完成。Ajax 的要点是使用异步 JavaScript 的方式来请求和处理网页上的数据。因此, 这个问题, 你可以再想想。:) (顺便预告一下, 我们在 03 篇就会介绍其中的一个使用场景)

2. 正确。当然, 还有其他的原则, 比如数据传输的效率, 可靠性, 协议本身的向后兼容性, 等等等等。

抓包的问题: 正确, 其它的基本都是 TCP 层的报文 (当然, 不只有连接建立握手的报文)。



5



seamoontime

2019-09-12

老师例子能不能用百度, bing之类的, 谷歌国内不可用啊

作者回复: 专栏几乎所有的例子都可以使用别的 HTTPS 网站来完成。这一篇里面除了 HTTP/0.9 那个小部分不能以外, 其它全部都可以使用百度或者 Bing 完成。:)



2



pyhhou

2019-09-11

思考题:

1. 当需要传输一个大文件, 不能一次传完, 可以使用分块传输, 这里的分块传输和 UDP 中的分块传输类似吗?

2. 罗列了下, 大概有几点:

1) 不存在歧义, 计算机最难做的事就是做选择...

展开 ∨

作者回复: 1. 对, “一次传不完” 是其中一个应用场景。它和 TCP 或 UDP 的 packet 的概念有些像, 但是他们是在不同的层次, 一个是在应用层, 一个是在传输层。

2. 嗯, 这些都是很好的方面。

选修课堂问题: 正确



1

**Kada**

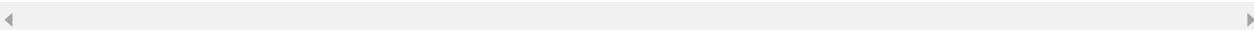
2019-09-14

因为有墙的原因，所以需要海外节点才能telnet到google的80端口。

建议使用<https://labs.play-with-docker.com>，可以一份中之内拉起一个海外linux实例。

展开 ▾

作者回复: 嗯，了解。很好的建议。

**infrared628**

2019-09-13

我可以access google，但是使用telnet www.google.com 80后cmd中啥也没有，按ctrl + c退出后cmd中看到出现以下错误：

HTTP/1.0 400 Bad Request

Content-Length: 54...

展开 ▾

作者回复: 关于 Google 那个，不清楚你的网络环境是否有特别的限制，否则应该是能够使用 telnet 的。你可以试试别的服务，看看telnet是否能访问。另外，telnet 只是建立了 tcp 连接，你需要发送消息才能得到响应。比如，在我的Mac上如下：telnet www.google.com 80

Trying 2607:f8b0:400a:809::2004...

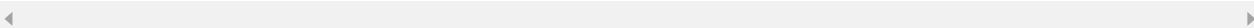
Connected to www.google.com.

Escape character is '^]'

GET /

...

netcat 的话，那个例子不是使用交互模式，而是直接运行：netcat www.google.com 80 < 文件路径

**谷径**

2019-09-12

在mac终端bash中，直接用python -m SimpleHTTPServer 8080 提示语法有错，不知道什么原因

作者回复: 你要是能贴出问题来大家可能能一起看一下。另外, 你看看是不是这个原因: 由于某些原因我的文章中没有使用 python 3, 如果你用的是 python3, 命令是 python3 -m http.server 8080 (当然, 如果你默认的 python 版本就是 3.xx 的, 那么使用 python 替换 python3)

当然, 你也完全可以抓访问其他网站的包, tcpdump 的命令需要稍微改一改 (不是抓本机的 loopback 的包了)。



Franklin.du

2019-09-12

刚看到标题以为是一篇枯燥的介绍http协议历史的文章, 看了以后发现这种和实践相结合的内容很有意思, 这个专栏应该会有很多收获。感谢四火老师。另外其它学员的留言也很有启发, 希望自己以后也能有高质量都留言。

展开 ∨

作者回复: 感谢你的认可!



小伟

2019-09-12

问题一: 在线视频播放, 先传输完整视频的部分包, 让视频缓冲播放, 提升用户体验。当然, 流媒体是不走http协议的, 走http的类似场景都是比较适合的。

问题二: 如果是应用层协议, 那报文的格式化和解析、请求应答的规则是重点。通用简明的报文格式易于格式化和解析, 适合粒度的应答码便于标示请求状态。

展开 ∨

作者回复: 其实流媒体有使用 HTTP 也有不使用 HTTP 的, 其它方面理解正确。对于协议的问题, 理解挺不错的, 当然, 这两个都是开放的问题, 你也可以看看其他人的回答 :)



joker

2019-09-11

抓包的应用场景都有哪些呀, 老师

展开 ∨

作者回复: 在和网络、Web 接口、性能等问题打交道的时候，很常用。



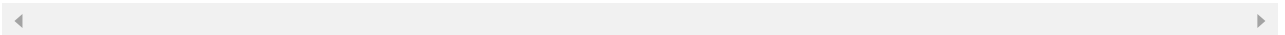
xcoder

2019-09-11

完了，这课不好学啊，好多不懂，看着一头雾水，缺少必要的知识去联系起来，工具还得自己去查查学起来，还要阅读英文文章。。。。。

展开 ∨

作者回复: 你好 xcoder，别害怕，每个人不同领域的知识储备都不同。你要是能具体谈谈那些不懂，我可以想想办法帮你



Geek_63377c

2019-09-11

老师，请问netcat在mac电脑上怎么安装呢？

展开 ∨

作者回复: 你可以安装 Homebrew <https://brew.sh/> 这个包管理工具，这样以后这些工具大多可以用它来安装，包括 netcat:

```
brew install netcat
```



leslie

2019-09-11

可能因为职业的特性吧：网络协议的设计基本都是由专业的网络工程师去做的；老师今天的第二个问题其实同样希望老师能够再后续课程做个思路的讲解吧。

今天课程中的tcpdump和wireshark之前都有使用过和强化过：课程中就略过了。

课后习题做个简单解答吧：第二个问题还是希望老师后面用个篇幅或者在某节课中讲解一下核心思路吧，麻烦老师了，给老师添加工作量了。...

展开 ∨

作者回复: 既然你特别说到第二个问题，你的回答中“内外网的隔离”其实是它下层的协议，例如第三层的网络层中的 IP 协议关心的，而不是第七层应用层的 HTTP 所关心的。建议你阅读以下扩展阅读 OSI 相关的内容。

设计方面，本专栏中不会详细解读协议设计，但会有很多应用层协议的使用，以及特别强化 Web 接口的设计的介绍。:)



tt

2019-09-11

赞，这是我订阅的课程里最“网络协议”式的课程了。

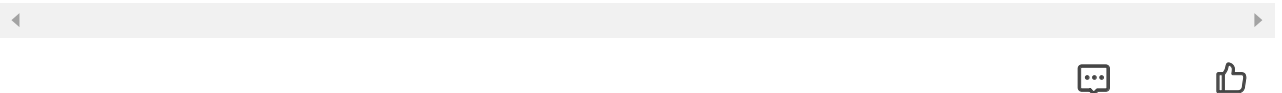
为什么呢？

定义网络协议就要像老师今天的课一样：

1、层次分明。适应于不同水平的人阅读，就像网络协议一样适应多种应用场景。...

展开 ∨

作者回复: 感谢。后面的文章也会尽量保持层次，每篇都有扩展阅读，争取让不同的人都有收获。当然，这篇是第一篇相对来说比较简单。



TossKing

2019-09-11

最近正好要做一个简单的私有协议实现从server下载包列表和具体的包文件，打算用http get，对http协议知之甚少，也没抓过包，这课来的太及时了。

展开 ∨

作者回复: 加油。特别是抓包这项技能，还是很实用的。当然，我只是给了个最简单例子。专栏后面我们还会使用抓包来分析问题。

