



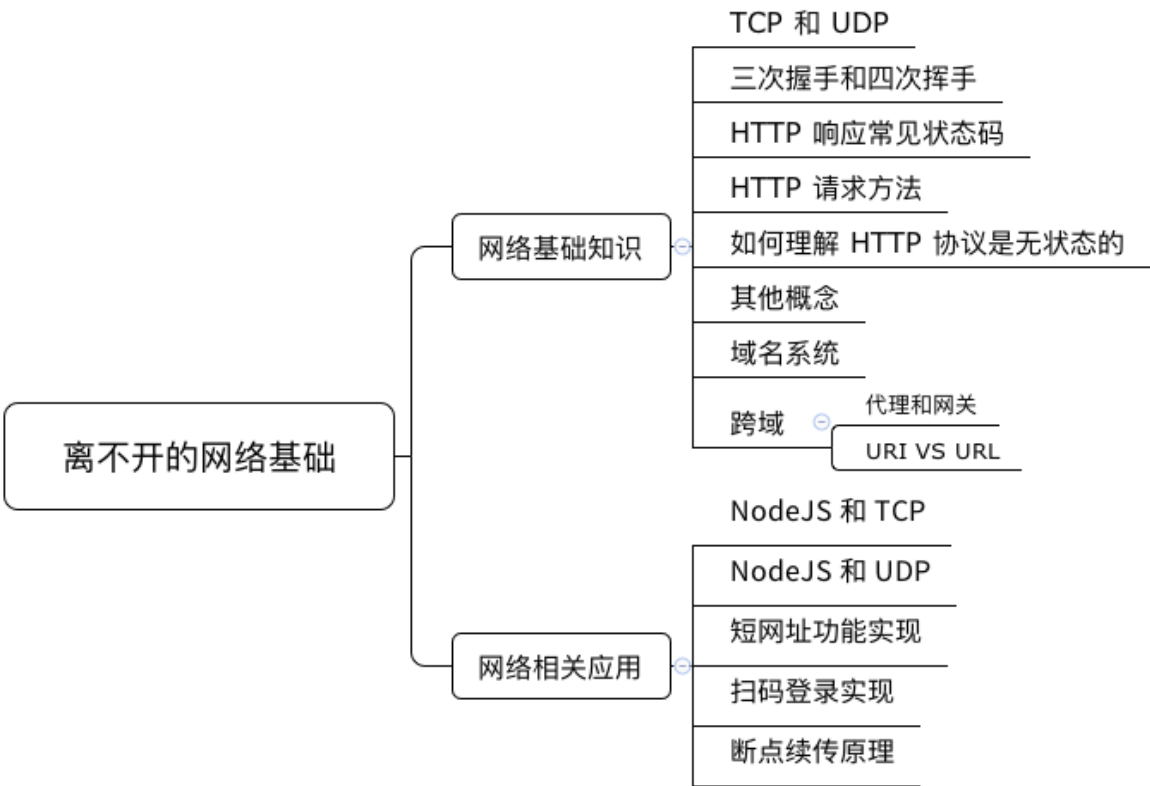
前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

离不开的网络基础

开启本讲之前，我想先问一句：「网络基础对于前端程序员重不重要」？其实答案是毫无疑问的，如果读者仍然对此有怀疑，那可能你还是刚入行的程序员，相信随着工作经验的增加，你会越来越能意识到网络基础的重要性。事实上，具备必要的网络基础，是对于一个程序员的要求，绝不仅仅是对前端工程师的要求，更何况在 Node.js 发展当下，学好网络知识更是对于前途发展意义重大。

网络基础其实并不难，但绝不仅仅是一讲的内容就能「融会贯通」的，系统性地学习还需要回炉到大学课本。本讲，我从网络基础概念和场景应用两大方面来讲解，主要内容如下：



网络基础知识

有一个说法——「大厂前端面试对 HTTP 的要求比 CSS 还要高」，确实从面试的频率上，以 HTTP 为核心的网络基础考察绝对是重点。这些考察点其实并不难，都是基础概念，我们先从几个高频考点入手。

更为基础的内容，比如计算机网络体系结构，OSI 七层协议等我们不再提及，必要知识需要读者先行了解。

TCP 和 UDP

TCP 和 UDP 是运输层的两种协议，什么是运输层呢？

运输层（Transport Layer）就是负责向两台主机进程之间的通信提供通用的数据传输服务，应用进程利用该服务传送应用层报文。「通用的」是指并不针对某一个特定的网络应用，而是多种应用可以使用同一个运输层服务。

对于 TCP 和 UDP 这两种协议的理解，我们可以主要从其特点和区别来掌握。

传输控制协议 TCP（Transmission Control Protocol）是一种**面向连接的、可靠的数据传输服务**。如何理解可靠呢？通过 TCP 协议传送数据可以无差错、不丢失、不重复、并且按序到达；如何理解面向连接呢？就像我们打电话一样，通过拿起电话和挂掉电话来表示连接的建立和中断。此外，TCP 的特点有：

TCP 提供全双工通信，也就是说双方在连接建立之后，都可以在任何时候进行数据发送

TCP 两端连接都设有缓存，在发送和接收时都可以利用缓存临时存放数据

TCP 是面向字节流的

用户数据协议 UDP（User Datagram Protocol）是一种**无连接的、不保证数据传输的可靠性**的运输层协议。其特点：

UDP 无连接

UDP 不保证可靠性，因此不需要维持复杂的链接状态

UDP 是面向报文的

UDP 没有拥塞控制

UDP 支持一对一、一对多、多对一和多对多的交互通信

因此根据 TCP 和 UDP 的特点，可以选择不同的协议进行场景应用，比如对于直播、实时视频会议，你认为哪种协议更加适合呢？

因为 UDP 传输速度更快、效率更高，UDP 没有拥塞控制，所以网络出现拥塞不会使源主机的发送速率降低，且直播、实时视频会议丢失一两帧内容对于应用并没有体验性的影响，因此，UDP 对于直播、实时视频会议的场景会更加适合。

这些内容对应的面试考点：

比较 TCP 和 UDP

TCP 和 UDP 的应用场景

TCP 如何保证传输的可靠性

前两项考点我们已经有所涉及，现在针对 TCP 如何保证传输的可靠性进行展开。TCP 保证传输的可靠性主要手段有以下几个。

数据包校验：如果接收端校验出包有错，则进行丢弃且不进行相应。

对失序数据包重排序：TCP 协议会对失序数据包进行排序，然后再交给应用层。

丢弃重复数据。

应答机制：当接收端接收到数据之后，将发送确认信息。

超时重发：当发送端发出数据后，它启动一个定时器，如果超出计时器的时限，将重发这个报文段。

流量控制：前面提到过，TCP 连接的每一方都有固定大小的缓冲空间，可防止接收端缓冲区溢出，这就是流量控制。TCP 使用可变大小的滑动窗口协议来进行流量控制。

最后再补充一点单工/半双工/双工数据通信的概念区分：

单工数据传输是数据只能在一个方向上传输；

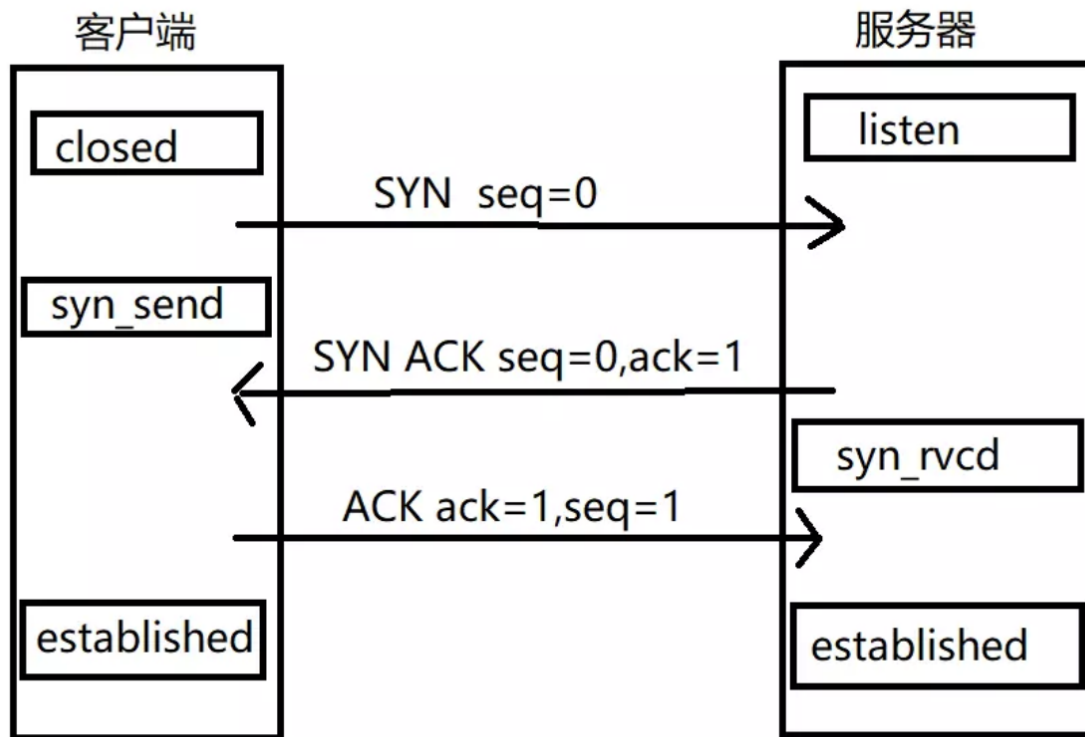
半双工数据传输允许数据在两个方向上传输，但是，在某一时刻，只允许数据在一个方向上传输，它实际上是一种切换方向的单工通信；

全双工数据通信允许数据同时在两个方向上传输，因此，全双工通信是两个单工通信方式的结合，它要求发送设备和接收设备都有独立的接收和发送能力。

三次握手和四次挥手

上面我们说到 TCP 是面向连接的，那么这个连接过程就涉及到著名的三次握手和四次挥手了。也许对于这些名词我们已经「听烂了」，但是你是否真正的掌握呢？

三次握手，简单来说是指当建立一个 TCP 连接时，整个建立过程需要客户端和服务端一共交互三个包，三次握手的目的是连接服务器的指定端口，建立 TCP 连接，并同步连接双方的序列号和确认号并交换 TCP 窗口大小信息。



具体来说 第一次握手

开始建立连接时，客户端向服务器发出连接请求报文，报文首部中的同部位 $\text{SYN} = 1$ ，同时选择一个初始序列号 $\text{seq} = x$ ，这时客户端进程进入了 SYN-SENT （同步已发送状态）状态，等待服务器确认。

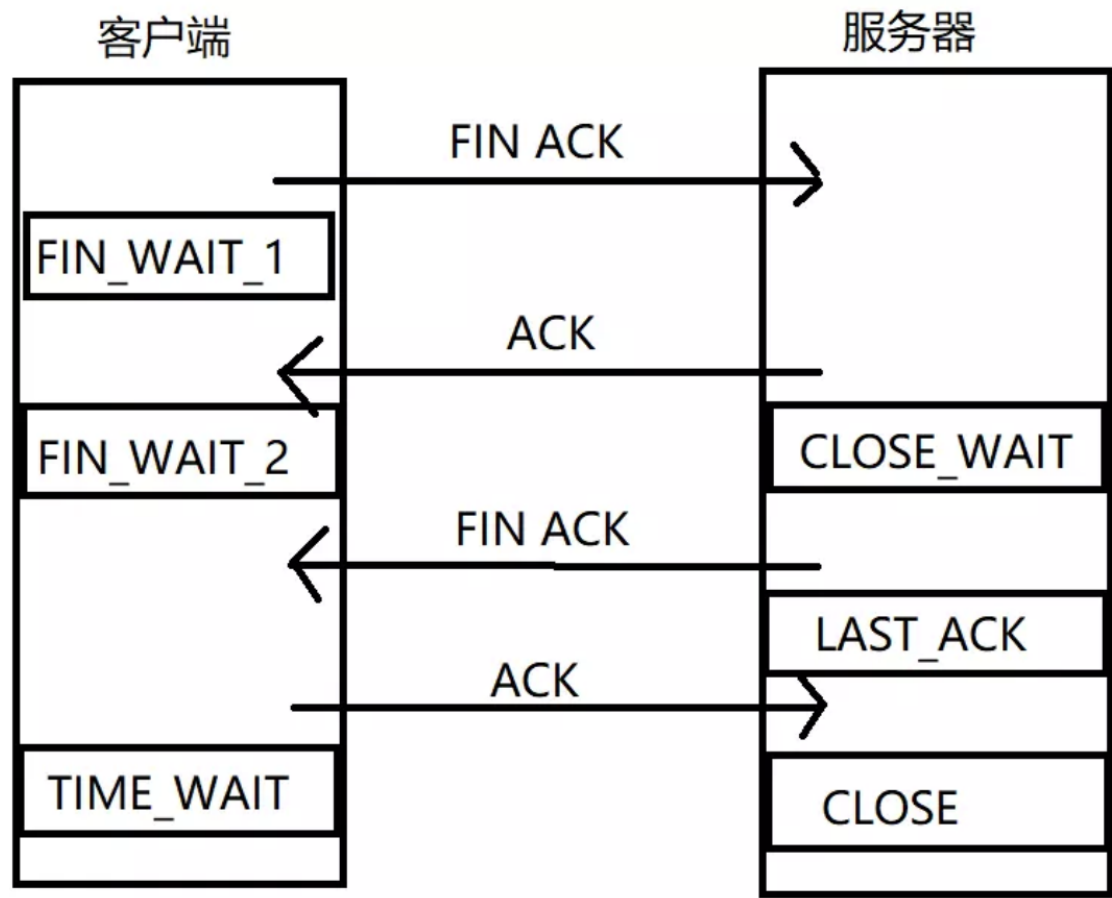
第二次握手

服务器收到 syn 包后，如果同意连接，则发出确认报文；确认报文 $\text{ACK} = 1$ ， $\text{SYN} = 1$ ，确认号是 $\text{ack} = x + 1$ ，同时也要为自己初始化一个序列号 $\text{seq} = y$ ，此时服务器进程进入了 SYN-RCVD （同步收到）状态。

第三次握手

客户端收到服务器的 $\text{SYN} + \text{ACK}$ 包，要向服务器给出确认。确认报文的 $\text{ACK} = 1$ ， $\text{ack} = y + 1$ ，自己的序列号 $\text{seq} = x + 1$ 。此时，TCP 连接建立，客户端进入 ESTABLISHED （已建立连接）状态。

四次挥手是指，TCP 连接的终端需要客户端和服务端总共发送四个包，客户端或者服务器端均可主动发起挥手动作。



具体来说，**第一次挥手**

客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部 $FIN = 1$ ，其序列号为 $seq = u$ （等于前面已经传送过来的数据最后一个字节的序号加 1）。此时，客户端进入 $FIN-WAIT-1$ （终止等待 1）状态。

第二次挥手

服务器收到连接释放报文，发出确认报文， $ACK = 1$ ， $ack = u + 1$ ，并且带上自己的序列号 $seq = v$ （客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受），此时，服务端就进入了 $CLOSE-WAIT$ （关闭等待）状态。

第三次挥手

服务器将最后的数据发送完毕后，就向客户端发送连接中断报文， $FIN = 1$ ， $ack = u + 1$ ，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序

列号为 $seq = w$ ，此时，服务器就进入了 LAST-ACK（最后确认）状态，等待客户端的确认。

第四次挥手

客户端收到服务器的连接释放报文后，必须发出确认， $ACK = 1$ ， $ack = w + 1$ ，而自己的序列号是 $seq = u + 1$ 。此时，客户端就进入了 TIME-WAIT（时间等待）状态。

服务器只要收到了客户端发出的确认，立即进入 CLOSED 状态。同样，撤销 TCB 后，就结束了这次的 TCP 连接（注意此时 TCP 连接还没有释放，必须经过 2MSL（最长报文段寿命）的时间后，当客户端撤销相应的 TCB 后，才进入 CLOSED 状态）。服务器只要收到了客户端发出的确认，立即进入 CLOSED 状态。同样，撤销 TCB 后，就结束了这次的 TCP 连接。

这些内容对应的面试考点：

三次握手和四次挥手过程

三次握手和四次挥手的必要性，不做行不行？

HTTP 响应常见状态码

HTTP 响应常见状态码这里只做一个速查总结。

100-199：表示成功接收请求，要求客户端继续提交下一次请求才能完成整个处理过程，常见的有 101（客户要求服务器转换 HTTP 协议版本）、100（客户必须继续发出请求）

200-299：表示成功接收请求并已完成整个处理过程

300-399：需要客户进一步细化需求，以进一步完成请求，常用的有 301（永久重定向）、302（临时重定向）、304（缓存相关）

400-499：请求出错，包含语法错误或者无法正确执行逻辑，常用的有 404（无对应资源）、401（权限问题）、403（服务器拒绝请求）

500-599：服务器端程序处理出现错误，常见的有 502（错误网关）、504（网关超时）、505（HTTP 版本不受支持）

列举更加具体的状态码说明，出自：[HTTP Status Code](#)

100 //继续 请求者应当继续提出请求。服务器返回此代码表示已收到请求的第一部分，正在等待其余部分

101 //切换协议 请求者已要求服务器切换协议，服务器已确认并准备切换

200 //成功 服务器已经成功处理了请求，通常，这表示服务器提供了请求的网页

201 //已创建 请求成功并且服务器创建了新的资源

202 //已接受 服务器已接受请求，但尚未处理

203 //非授权信息 服务器已经成功处理了请求，但返回的信息可能来自另一来源

204 //无内容 服务器成功处理了请求，但没有返回任何内容

205 //重置内容 服务器成功处理了请求，但没有返回任何内容

206 //部分内容 服务器成功处理了部分 GET 请求

300 //多种选择 针对请求，服务器可执行多种操作。服务器可根据请求者（user agent）选择一项操作，或提供操作列表供请求者选择

301 //永久移动 请求的网页已永久移动到新位置。服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置

302 //临时移动 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求

303 //查看其他位置 请求者应当对不同的位置使用单独的 GET 请求来检索响应时，服务器返回此代码

304 //未修改 自动上次请求后，请求的网页未修改过。服务器返回此响应，不会返回网页的内容

305 //使用代理 请求者只能使用代理访问请求的网页。如果服务器返回此响应，还表示请求者应使用代理

307 //临时性重定向 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有的位置来进行以后的请求

400 //错误请求 服务器不理解请求的语法

- 401 //未授权 请求要求身份验证。对于需要登录的网页，服务器可能返回此响应
- 403 //禁止 服务器拒绝请求
- 404 //未找到 服务器找不到请求的网页
- 405 //方法禁用 禁用请求中指定的方法
- 406 //不接受 无法使用请求的内容特性响应请求的网页
- 407 //需要代理授权 此状态码与 401（未授权）类似，但指定请求者应当授权使用代理
- 408 //请求超时 服务器等候请求时发生超时
- 409 //冲突 服务器在完成请求时发生冲突。服务器必须在响应中包含有关冲突的信息
- 410 //已删除 如果请求的资源已永久删除，服务器就会返回此响应
- 411 //需要有效长度 服务器不接受不含有效内容长度标头字段的请求
- 412 //未满足前提条件 服务器未满足请求者在请求者设置的其中一个前提条件
- 413 //请求实体过大 服务器无法处理请求，因为请求实体过大，超出了服务器的处理能力
- 414 //请求的 URI 过长 请求的 URI（通常为网址）过长，服务器无法处理
- 415 //不支持媒体类型 请求的格式不受请求页面的支持
- 416 //请求范围不符合要求 如果页面无法提供请求的范围，则服务器会返回此状态码
- 417 //未满足期望值 服务器未满足「期望」请求标头字段的要求
-
- 500 //服务器内部错误 服务器遇到错误，无法完成请求
- 501 //尚未实施 服务器不具备完成请求的功能。例如，服务器无法识别请求方法时可能会返回此代码
- 502 //错误网关 服务器作为网关或代理，从上游服务器无法收到无效响应
- 503 //服务器不可用 服务器目前无法使用（由于超载或者停机维护）。通常，这只是暂时状态
- 504 //网关超时 服务器作为网关代理，但是没有及时从上游服务器收到请求
- 505 //HTTP 版本不受支持 服务器不支持请求中所用的 HTTP 协议版本

这些内容对应的面试考点：

状态码的熟悉程度

缓存相关状态码

Restful 相关内容

HTTP 请求方法

常见的 HTTP 请求方法有以下几个。

GET 方法：该方法发送请求来取得服务器上的资源，一般来说 GET 方法应该只用于数据的读取，而不应当用于会产生副作用的非幂等的操作中。

POST 方法：该方法向将指定资源的最新数据传送给服务器取代指定的资源的内容，POST 方法是非幂等的方法，因为这个请求可能会创建新的资源或 / 和修改现有资源。

PUT 方法：类似 POST 方法，该方法指定了资源在服务器上的位置，进行数据提交或数据更新

PATCH 方法：该方法出现的较晚，它在 2010 年的 RFC 5789 标准中被定义。一般用于资源的部分更新，而 PUT 一般用于资源的整体更新。另外，当资源不存在时，PATCH 会创建一个新的资源，而 PUT 只会对已在资源进行更新。

HEAD 方法：该方法只请求页面的首部，也就是说服务端的返回不含内容部分。这个方法，允许我们可以不传输全部内容的情况下，就可以获取服务器的响应头信息。HEAD 方法常被用于客户端查看服务器的性能。

DELETE 方法：该方法删除服务器上的某资源或者数据。

OPTIONS 方法：该方法用于获取指定服务能够支持的方法。当请求成功时，客户端会得到相关头部信息，指定了服务能够支持的方法，比如「GET、POST」等。JavaScript 的 XMLHttpRequest 对象进行 CORS 跨域资源共享时，就是使用 OPTIONS 方法发送嗅探请求，以判断是否有对指定资源的访问权限。

TRACE 方法：被用于激发一个远程的，应用层的请求消息回路，该方法主要用于 HTTP 请求的测试或诊断。

CONNECT 方法：HTTP/1.1 协议预留的，能够将连接改为管道方式的代理服务器。通常用于 SSL 加密服务器的链接与非加密的 HTTP 代理服务器的通信。

我们再从 HTTP 方法的安全性和幂等性角度来进行总结：

安全性是说**多次调用不会产生副作用**，换句话说，**安全的方法不会修改资源状态**；

幂等性，是指该方法**多次调用返回的效果（形式）一致**，客户端可以重复调用并且期望同样的结果。

方法名	安全性	幂等性
GET	是	是
HEAD	是	是
OPTIONS	是	是
DELETE	否	是
PUT	否	是
POST	否	否

这些内容对应的面试考点：

GET/POST 区别

OPTIONS 理解，以及跨域相关内容

Restful 相关内容

如何理解 HTTP 协议是无状态的

综合以上内容，我们来分析一个面试常考题目：如何理解 HTTP 协议是无状态的？

我们之所以说 HTTP 协议是无状态的，其实指的是客户端和服务器的通信，每个请求之间是独立的，指的是 HTTP 协议对于独立的请求是没有记忆能力的。

通俗点说，我们的应用发送一个请求，和下一次再打开该应用，发送同一个请求，这些请求之间没有任何联系。**HTTP 是一个无状态的面向连接的协议**，但是无状态不代表 HTTP 不能保持 TCP 连接，更不能代表 TCP 不是面向连接的。

其他概念

这一讲不再做更多 HTTP 内容的介绍，更多 HTTP 的知识会在《HTTP 的深思：我从何而来，去向何处》中继续说明。我们来看一些更多的概念。

域名系统

域名系统（Domain Name System，DNS）是因特网的一项核心服务，它作为可以将域名和 IP 地址相互映射的一个分布式数据库，能够使人更方便地访问互联网，而不用去记住能够被机器直接读取的 IP 数串。

比如我们经常访问的网站，其域名就相当于其门牌地址，比如 IBM 公司的域名是 `www.ibm.com`；Cisco 公司的域名是 `www.cisco.com`。因此对每一个网站进行访问时，都需要对域名和对应的 IP 地址进行映射，具体过程：

先过浏览器搜索自己的 DNS 缓存（可以使用 `chrome://net-internals/#dns` 来进行查看）

上一步未找到对应缓存的 IP 地址时，搜索操作系统中的 DNS 缓存

上一步未找到对应缓存的 IP 地址时，操作系统将域名发送至 LDNS（本地区域名服务器），LDNS 查询自己的 DNS 缓存（一般查找成功率在 80% 左右），查找成功则返回结果，失败则发起一个迭代 DNS 解析请求，为什么说这是迭代的 DNS 解析请求呢？这个过程

LDNS 向 Root Name Server（根域名服务器，如 com、net、org 等解析顶级域名服务器的地址）发起请求，此处，Root Name Server 返回 com 域的顶级域名服务器地址

LDNS 向 com 域的顶级域名服务器发起请求，返回 baidu.com 域名服务器地址

LDNS 向 baidu.com 域名服务器发起请求，得到 www.baidu.com 的 IP 地址 LDNS 将得到的 IP 地址返回给操作系统，同时自己也将 IP 地址缓存起来

操作系统将 IP 地址返回给浏览器，同时自己也将 IP 地址缓存起来

DNS 解析整个过程，分为：递归查询过程和迭代查询过程。

读者可以通过：

```
cat /etc/resolv.conf
```

获取本机 DNS 配置，我的内容为：



```
cedeMacBook-Pro:~ cehou$ cat /etc/resolv.conf
# This file is not consulted for DNS hostname resolution, address
# resolution, or the DNS query routing mechanism used by most
# processes on this system.
#
# To view the DNS configuration used by this system, use:
#   scutil --dns
#
# SEE ALSO
#   dns-sd(1), scutil(8)
#
# This file is automatically generated.
nameserver 10.0.0.1
```

域名系统

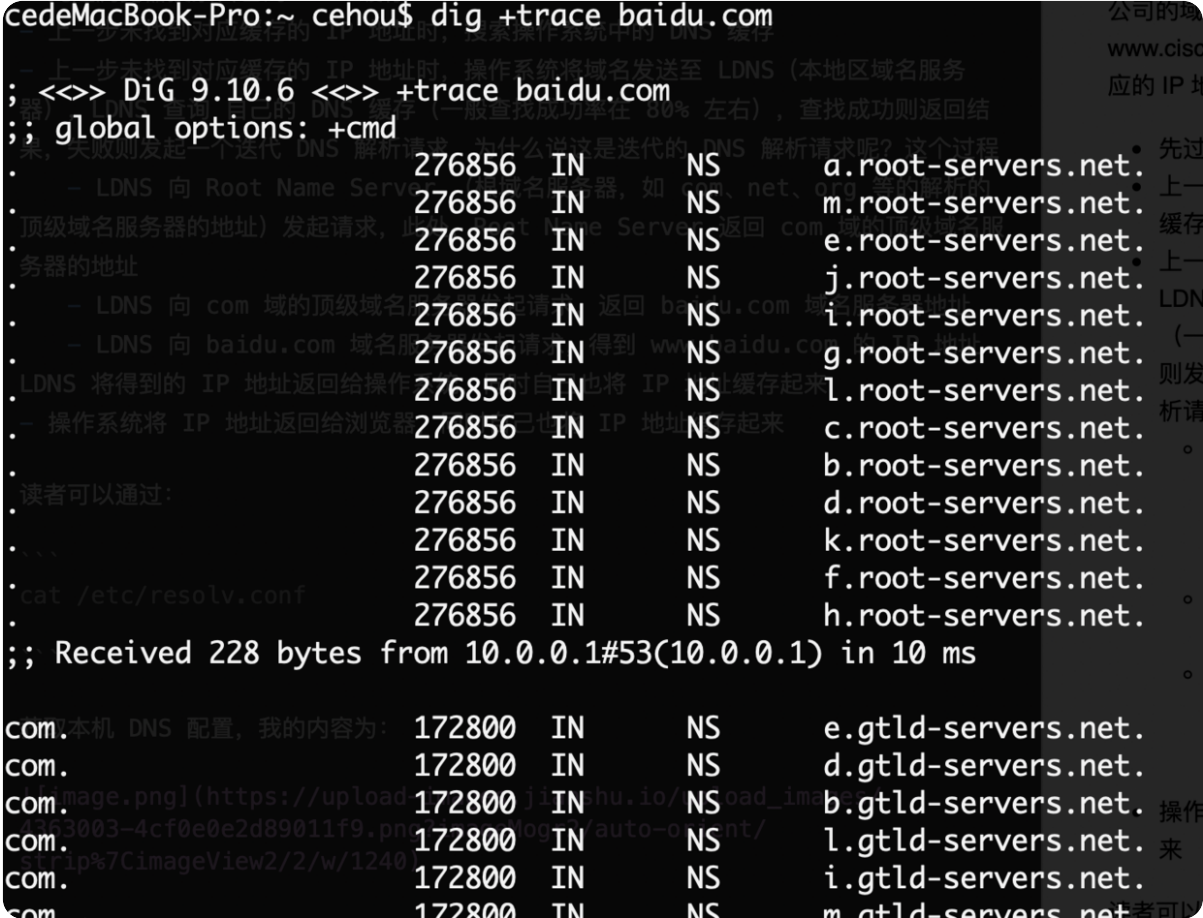
域名系统 (Domain Name System) 是一种用于将域名与 IP 地址进行映射的系统。它使得用户可以通过易于记忆的域名来访问网络资源，而不需要直接使用 IP 地址。

- 先过浏览器缓存
- 上一步未找到缓存
- 上一步未找到缓存

再通过：

```
dig +trace XXX.com
```

来查看完整的分级查询过程：



```

cedMacBook-Pro:~ cehou$ dig +trace baidu.com
; <<>> DiG 9.10.6 <<>> +trace baidu.com
;; global options: +cmd
276856 IN      NS      a.root-servers.net.
276856 IN      NS      m.root-servers.net.
276856 IN      NS      e.root-servers.net.
276856 IN      NS      j.root-servers.net.
276856 IN      NS      i.root-servers.net.
276856 IN      NS      g.root-servers.net.
276856 IN      NS      l.root-servers.net.
276856 IN      NS      c.root-servers.net.
276856 IN      NS      b.root-servers.net.
276856 IN      NS      d.root-servers.net.
276856 IN      NS      k.root-servers.net.
276856 IN      NS      f.root-servers.net.
276856 IN      NS      h.root-servers.net.
;; Received 228 bytes from 10.0.0.1#53(10.0.0.1) in 10 ms

com. 172800 IN      NS      e.gtld-servers.net.
com. 172800 IN      NS      d.gtld-servers.net.
com. 172800 IN      NS      b.gtld-servers.net.
com. 172800 IN      NS      l.gtld-servers.net.
com. 172800 IN      NS      i.gtld-servers.net.
com. 172800 IN      NS      m.gtld-servers.net.

```

这些内容对应的面试考点：

域名系统相关工作方式

顶级域名、一级域名、二级域名概念

最后，需要说明的是 DNS 使用无连接的 UDP 协议来进行查询，这样的方式降低了开销，也使得速度更快，保证了高效的通信，但是没有太考虑安全问题。它使用目的端口为 53 的 UDP 明文进行通信，这也带来了诸如 DNS 欺骗、DNS Cache 污染、DNS 放大攻击等问题，一些「黑心」运营商就可以利用这一点达到一些别用用心的目的。

针对于此，DNSSec（Domain Name System Security Extensions，也叫「DNS 安全扩展」）机制便诞生了，这个机制会让客户端对域名来源身份进行验证，并且检查来自 DNS 域名服务器应答记录的完整性，以及验证是否在传输过程中被篡改过。总之域名系统的安全性话题已经不可忽视。

跨域

跨域其实是浏览器的行为，狭义上说，它指的是浏览器无法执行其他网站的脚本。为什么无法执行呢？这要由浏览器的同源策略说起，简单来说，跨域是浏览器对 JavaScript 施加的安全限制。

同源策略（Same Origin Policy）是一种约定，由 Netscape 公司 1995 年引入浏览器，它是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到 XSS、CSFR 等攻击。

我们都知道，所谓同源，是指「协议、域名、端口」这三要素都相同，在非同源的情况下，以下行为会收到限制：

对 Cookie、LocalStorage 和 IndexedDB 的读取

对 DOM 和 JS 对象的读取

Ajax 请求的正常执行

关于解决跨越方案的内容市面上的资料讲解较多，我们这里不再赘述。

代理和网关

在 NodeJS「攻城略地」的背景下，前端开发者越来越多地接触到了代理和网关的概念。代理简单来说，是一种有转发功能的应用程序，它扮演了位于两端之间「中间人」的角色。比如在我的一个项目中，公司内部中台有基于 StatsD（StatsD 就是一个简单的网络守护进程，基于 Node.js 平台，通过 UDP 或者 TCP 方式侦听各种统计信息）的数据收集和统计系统，因为 StatsD 是基于

TCP/UDP 的，因此为了在 C 端统计用户行为，我设计了在 NodeJS 层的一个代理机制，对 C 端上报数据进行简单的接受和分发。

网关是转发其他服务器通信数据的服务器，当接收从客户端发送来的请求时，它就像自己拥有资源的源服务器一样对请求进行处理，其工作机制和代理类似，但网关能力可能更加强大，它不是提供一个单一服务，而是能给通信链路提供平台化的服务，比如鉴权等内容。

更多应用我们暂不展开，本讲会把更多精力放在网络基础知识的学习上。这里只需要读者明白，前端 BFF 层，或代理层，越来越需要开发者对于网络基础知识的理解和运用。

URI VS URL

很多开发者并不理解 URI 和 URL 的区别，从名称上来看：

URL，统一资源定位符

URI，统一资源标识符

通俗来说，URL 像是一个邮政编码，URI 就是收件地址。因此可知，URL 的范围大于 URI。我们以淘宝的例子来说，<https://www.taobao.com/> 这个域名就是 URL，而每个商品的地址就是一个 URI。

很多 Ajax 请求库的参数都设计成叫做 url，表示请求地址。但实际上，更准确的表达是 URI。

网络相关应用

这部分我们来看一些网络知识的应用，同时剖析一些案例。

NodeJS 和 TCP

在 NodeJS 中，我们可以很轻松地实现一个基于 TCP 的数据通信过程，主要依赖 net 模块：


```
let net = require('net')

let server = net.createServer(socket => {
  console.log('客户端已经链接')
  console.log(socket.address())
})

server.listen('8080', () => {
  /* 获取地址信息，得到的是一个 json { address: '::', family:
  'IPv6', port: 8000 } */
  const address = server.address()

  /* TCP 服务器监听的地址 */
  console.log(`the port of server is ${address.port}`)

  /* IPv6 还是 IPv4 */
  console.log(`the family of server is
  ${address.family}`)
})

server.getConnections((err, count) => {
  console.log(`已经链接 ${count} 个用户`)
})

server.maxConnections = 2
```

这样就使用 NodeJS 创建了一个简单的 TCP 服务器，并设置了最大连接数，监听客户端链接数量，以及对事件的处理等。

也可以利用 socket.write 进行 TCP 服务器的数据发送：

```
var net = require("net")

/* 创建 TCP 服务器 */
let server = net.createServer(socket => {
  var address = server.address()
  var message = `the server address is
```

```
${JSON.stringify(address)}`

    socket.write(message, () => {
        let writeSize = socket.bytesWritten
        console.log(`${message} has send, the size is
${writeSize}`)
    })

    socket.on('data', (data) => {
        console.log(data.toString())
        const readSize = socket.bytesRead
        console.log(`the size of data is ${readSize}`)
    })
})

server.listen(8000, () => {
    console.log("Creat server on http://127.0.0.1:8000/")
})
```

我们也可以用 NodeJS 来构建一个 TCP 客户端，实现 TCP 客户端和 TCP 服务器的通信，这里就不再展开了。

NodeJS 和 UDP

关于 NodeJS 实现 UDP 通信，我们需要依靠 dgram 模块，dgram 模块提供了 UDP 数据包 socket 的实现。先看 UDP server 的创建，创建 server.js 文件：

```
const dgram = require('dgram')

// 创建 UDP server
let udpServer = dgram.createSocket('udp4')
// 绑定端口
udpServer.bind(5678)

// 监听端口
udpServer.on('listening', () => {
    console.log('udp server linstening 5678.')
```

```
})
```

```
//接收消息
```

```
udpServer.on('message', (message, rinfo) => {  
  const messageStr = message.toString()  
  udpServer.send(messageStr.toString(), 0,  
messageStr.length, rinfo.port, rinfo.address)  
  console.log(`udp server received data: ${messageStr}  
from ${rinfo.address}:${rinfo.port}`)  
})
```

```
//错误处理
```

```
udpServer.on('error', err => {  
  console.log('some error on udp server.')  
  udpServer.close()  
})
```

创建 UDP Client, 创建 client.js:

```
const dgram = require('dgram')  
let udpClient = dgram.createSocket('udp4')
```

```
udpClient.on('close', () => {  
  console.log('udp client closed.')  
})
```

```
// 错误处理
```

```
udpClient.on('error', () => {  
  console.log('some error on udp client.')  
})
```

```
// 接收消息
```

```
udpClient.on('message', (message, rinfo) => {  
  console.log(`receive message from ${rinfo.address}:  
${rinfo.port}: ${message}`)  
})
```

```
// 定时向服务器发送消息
setInterval(() => {
  const sendStr = 'hello.'
  const sendStrLen = sendStr.length
  udpClient.send(sendStr, 0, sendStrLen, 5678,
    '172.30.20.10')
}, 3000)
```

在上述代码中，读者可以在 UDP Client 结尾处配置好自己的 IP 地址，并启动：

```
node server.js
node client.js
```

观察 NodeJS 应用状况。

短网址功能实现

常用微博或者关注运营需求的读者应该知道短网址功能：

短网址生成工具

请输入需要缩短的网址：

生成

使用说明：

- 1、输入正确的长连接后，点击“生成”按钮，即可生成短链接
- 2、本站生成的短链接长期有效

简单来说，短网址工具可以将一长串 URL 地址转换成简短的、可访问的短链接形式。自微博盛行以来，在微博字数有限的特色下，短链接盛行于微博网站，以节省微博字数，给博主发布更多文字的空间。

那么一个短网址生成平台该怎么设计呢？其实原理很简单：

用户输入完整网址，服务端接收到完整网址之后，根据算法生成一个短码，维护完整网址和短码的映射关系，并将短码完善成短网址，返回给客户端；

任意客户端访问短网址，服务端根据完整网址和短码的映射关系，重定向到对应的页面。

我们看到了熟悉的三个字：重定向，请读者思考这里的重定向应该对应哪个 HTTP 状态码呢？

我们应该在 302 和 301 中进行选择：从语义上看，短网址对应完整网址的映射关系和跳转关系不会发生变化，应该是 301 永久重定向才对。但是更多的短网址生成平台却采用了 302，这是为什么呢？

引自知乎网友的回答，[原文链接](#)。

如果用了 301，Google、百度等搜索引擎，搜索的时候会直接展示真实地址，那我们就无法统计到短地址被点击的次数了，也无法收集用户的 Cookie、User Agent 等信息，这些信息可以用来做很多有意思的大数据分析，也是短网址服务商的主要盈利来源。

完整的设计我们可以借助 MySQL 和 Redis 实现完整网址和短码之间的映射关系。生成短码的算法主要可以考虑以下几种：

自增 id，然后将 id 值转换为 62 进制的字符串，为了解决短码长度不固定的问题，可以指定数字开始递增。同时为了解决短码有序的安全隐患，可以结合 md5 进行混淆。一种实现为：

```
const string10to62 = number => {
  const chars =
    '0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    'VWXYZ'

  const charsArr = chars.split('')
  const radix = chars.length
```

```
let quotient = +number
let arr = []

do {
  let mod = quotient % radix
  quotient = (quotient - mod) / radix
  arr.unshift(charsArr[mod])
}
while(quotient)

return arr.join('')
}
```

哈希算法，直接输入任意长度的数据，输出固定长度的数据，这种算法比较典型，不再过多介绍

随机数：从 62 个字符串中随机取出固定长度的短码组合，然后去数据库中查询该短码是否已存在。如果已存在，就继续循环该方法重新获取短码，否则就直接返回。这种方法最为简单，但是得到碰撞的概率相对较大，一种实现为：

```
const generateShortLink = () => {
  let str = ''
  const arr = [
    '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
    'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j',
    'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u',
    'v', 'w', 'x', 'y', 'z',
    'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
    'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U',
    'V', 'W', 'X', 'Y', 'Z',
  ]

  for (let i = 0; i < 6; i++) {
    const pos = Math.round(Math.random() * (arr.length
- 1))
    str += arr[pos]
  }
}
```

```
    }  
    return str  
}  
  
async getShortLink() {  
    const shortLink = this.generateShortLink()  
  
    const searchResult = await  
this.searchByLinkInMySQL(shortLink)  
  
    if (searchResult && searchResult.length > 0) {  
        return this.getShortLink();  
    }  
    return shortLink  
}
```

这一部分应用的分析，参考了社区 [shortLink](#) 的实现和分析内容。读者可进行进一步研究。

扫码登录实现

另外一个我们要介绍的实现是扫码登录，现在授权第三方的扫码登录越来越流行。比如我们可以通过微信实现在第三方应用的登录、注册账号等，这种共用账号体系的能力究竟是如何实现的呢？

其实原理同样并不复杂：

首先打开 PC 端页面，PC 端页面生成一个二维码，这个二维码带有服务端下发的一个唯一 id：uuid

接着，手机在微信账号中进行扫码，发送请求，请求包含了上面提到的 uuid 和当前用户的账号信息

与此同时，PC 端不断轮询服务器（或者通过 websocket，服务端主动 push 信息），获取扫码登录的状态

服务端返回扫码状态

针对手机扫码后服务端的返回情况，我们进行梳理：

PC 端在轮询时，如果扫码超时（手机没有授权登录或者就没扫码），服务端会阻塞一个时限（比如 30s），30s 内无响应，服务端返回状态码 408，得到返回后，前端继续轮询

大约 5 分钟内仍然没有扫码响应，则二维码失效，这时候服务端返回 400

如果手机端扫码成功，服务端返回 201 同时并返回用户信息，并等待用户点击「确认授权登录」

用户「确认授权登录」之后，服务端返回 200 同时返回一个 token；PC 端在拿到下响应后，重定向到目标页面，同时停止轮询

关键理解点：

扫码登录一共涉及到三端：服务端、PC 端和手机扫码端

PC 端通过轮询，不断向服务端获取用户扫码登录的状态

服务端阻塞 PC 端请求，这样可以减少 PC 端轮询的次数，优化轮询过程

服务端根据手机扫码状态，返回 408、400、201、200 等状态码

断点续传原理

对于大文件的传输/下载，我们常常采用断点续传原理。想象一下在使用迅雷、网盘上传文件内容时，如果网络条件出现问题，我们仍然可以稍后继续进行上传/下载而不至于丢失之前的上传/下载进度。这种断点续传的能力究竟是如何实现的呢？

实现断点续传，需要依靠 HTTP1.1 协议（RFC2616），该协议版本开始支持获取文件的部分内容，这为并行下载以及断点续传提供了技术支持。

在前端发送请求时，需要在 Header 里加入 Range 参数，同时服务器端响应时返回带有 Content-Range 的 Header，也就是说 Range 和 Content-Range 是一

对对应的 Header 头。

比如：

```
Range: bytes=500-999
```

就表示上传第 500-999 字节范围的内容，而浏览器在发出带 Range 的请求后，服务器会在 Content-Range 头部返回当前接受的范围和文件总大小，比如：

```
Content-Range: bytes 0-499/22400
```

就指当前发送数据的范围是 0-499，22400 则是文件的总 size。

我们来看一个例子：

浏览器下载一个 1024K 的文件，当前已经下载了 512K

这时候不幸网络故障，稍后浏览器请求续传，这时候带有 Range:bytes=512000 的 Header 头，表明本次需要续传的片段

服务端接收到断点续传的请求，从文件的 512K 位置开始传输，并返回 Header 头：Content-Range:bytes 512000-/1024000，注意这时候的 HTTP status code 是 206，而非 200，206 表示：206 Partial Content（使用断点续传方式）

请读者思考一个问题，如果在网络故障期间，服务器端文件发生了变化，导致 512K 部分并不能对上之前的内容，这个怎么办呢？

这时候就需要一个标识文件唯一性的标识符。RFC2616 中规定可以使用 Last-Modified 顾名思义，这样就可以标识文件的最后修改时间，浏览器就可判断出续传文件时是否已经发生过改动。这种方式并不惟一，也可以通过名为 Etag 的 Header，直接表文件的唯一标记（类似文件的 MD5 值）。浏览器端请求时申明 If-None-Match 或者 If-Modified-Since 字段，帮助服务端判别文件变化，同时浏览器也可以采用 If-Range Header，该头部包含 ETag 头或者是 Last-Modified 信息，同样可以帮助服务端进行内容校验。这时候，服务端在校验一致时返回

206 的续传回应校验不通过时，服务端则返回 200 回应，回应的内容为新的文件的全部数据。

总结

本讲我们以面试为切入点，总结了网络基础知识的多个方面。其中一些知识比较基础，但根据我观察，很多面试者都会在这些基础内容上「折腰」。网络知识是一个系统性的知识体系，还需要每一个开发者认真学习。也许你有体会：「在大学里学习网络知识时，很多概念很难真正理解」，而工作后有一定实战经验了，但是「很多网络内容又缺少了理论的支持」。因此关于网络的学习，我认为一定要理论结合实践。这些计算机基础内容，是一个开发者真正进阶的基本功。

点击查看下一节 

缓存谁都懂，一问都哑口