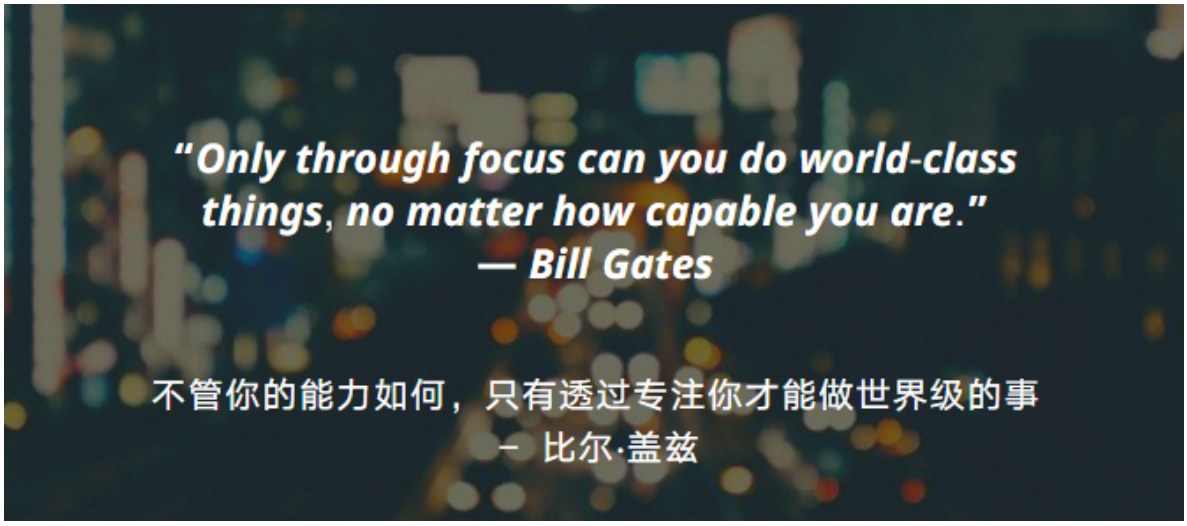
硬核！30张图解 HTTP 常见面试题

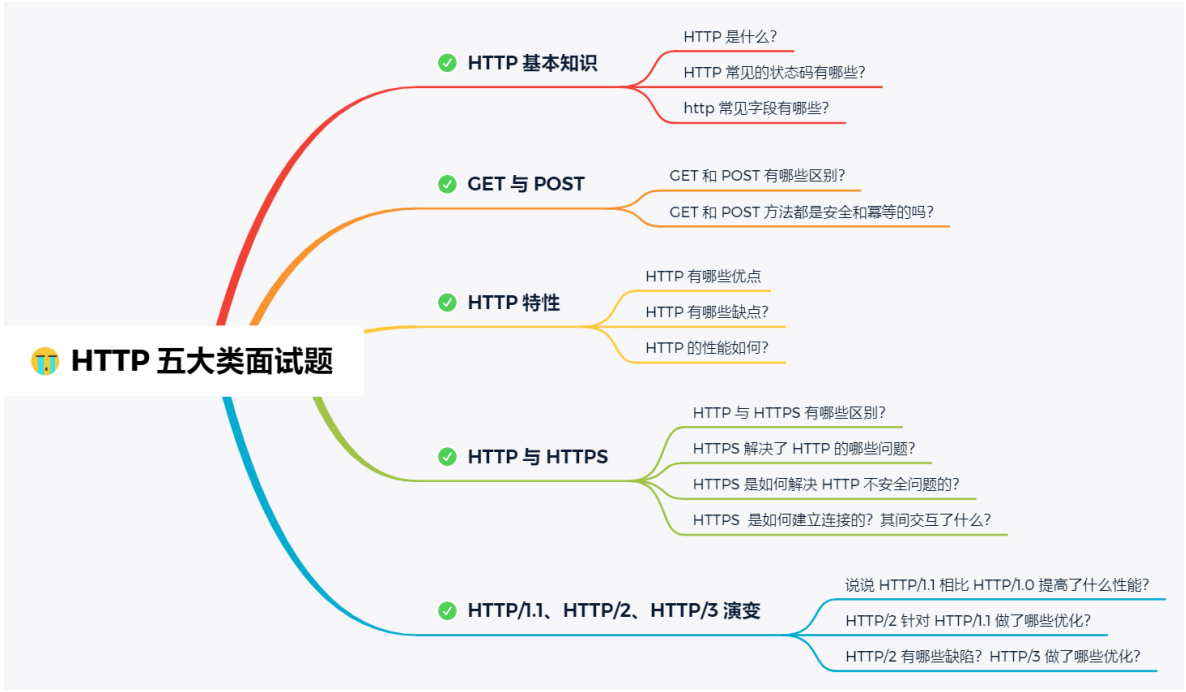


前言

在面试过程中，HTTP 被提问的概率还是比较高的。

小林我搜集了 5 大类 HTTP 面试常问的题目，同时这 5 大类题跟 **HTTP 的发展和演变**关联性是比较大的，通过**问答 + 图解**的形式**由浅入深**的方式帮助大家进一步的学习和理解 HTTP 。

1. HTTP 基本概念
2. Get 与 Post
3. HTTP 特性
4. HTTPS 与 HTTP
5. HTTP/1.1、HTTP/2、HTTP/3 演变



正文

01 HTTP 基本概念

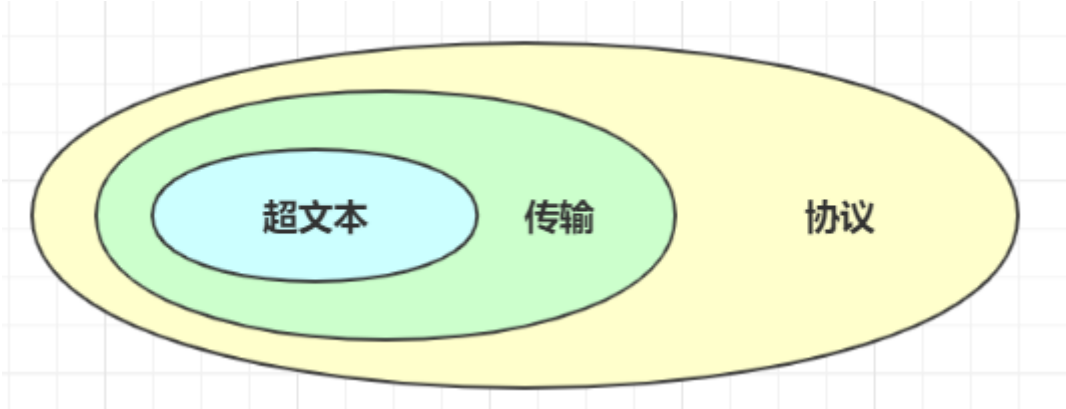
> HTTP 是什么？描述一下

HTTP 是超文本传输协议，也就是**H**yperText **T**ransfer **P**rotocol。

> 能否详细解释「超文本传输协议」？

HTTP的名字「超文本协议传输」，它可以拆成三个部分：

- 超文本
- 传输
- 协议



1. 「协议」

在生活中，我们也能随处可见「协议」，例如：

- 刚毕业时会签一个「三方协议」；
- 找房子时会签一个「租房协议」；

生活中的协议，本质上与计算机中的协议是相同的，协议的特点:

- 「**协**」字，代表的意思是必须有**两个以上的参与者**。例如三方协议里的参与者有三个：你、公司、学校三个；租房协议里的参与者有两个：你和房东。
- 「**议**」字，代表的意思是对参与者的一种**行为约定和规范**。例如三方协议里规定试用期期限、毁约金等；租房协议里规定租期期限、每月租金金额、违约如何处理等。

针对 HTTP **协议**，我们可以这么理解。

HTTP 是一个用在计算机世界里的**协议**。它使用计算机能够理解的语言确立了一种计算机之间交流通信的规范（**两个以上的参与者**），以及相关的各种控制和错误处理方式（**行为约定和规范**）。

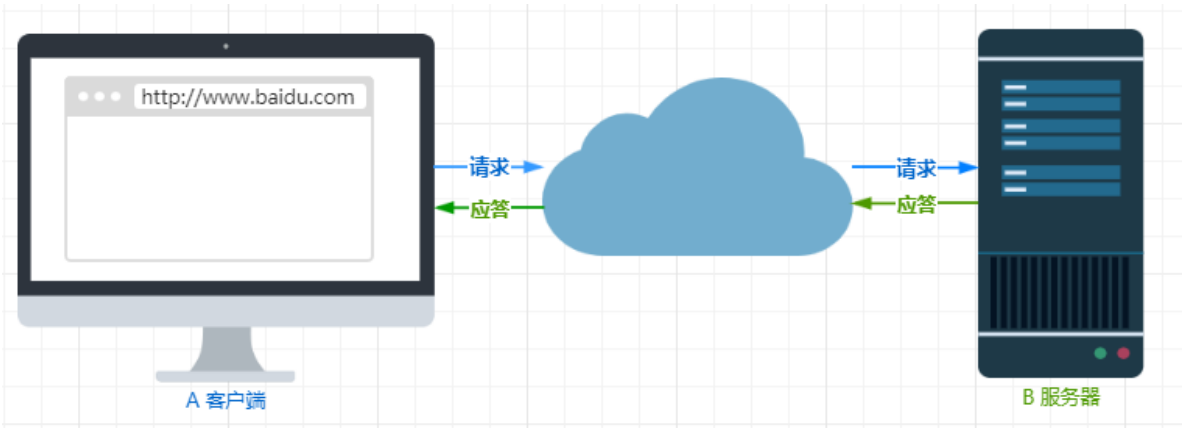
2. 「传输」

所谓的「传输」，很好理解，就是把一堆东西从 A 点搬到 B 点，或者从 B 点 搬到 A 点。

别轻视了这个简单的动作，它至少包含两项重要的信息。

HTTP 协议是一个**双向协议**。

我们在上网冲浪时，浏览器是请求方 A ，百度网站就是应答方 B。双方约定用 HTTP 协议来通信，于是浏览器把请求数据发送给网站，网站再把一些数据返回给浏览器，最后由浏览器渲染在屏幕，就可以看到图片、视频了。



数据虽然是在 A 和 B 之间传输，但允许中间有**中转或接力**。

就好像第一排的同学想传递纸条给最后一排的同学，那么传递的过程中就需要经过好多个同学（中间人），这样的传输方式就从「A < --- > B」，变成了「A <-> N <-> M <-> B」。

而在 HTTP 里，需要中间人遵从 HTTP 协议，只要不打扰基本的数据传输，就可以添加任意额外的东西。

针对**传输**，我们可以进一步理解了 HTTP。

HTTP 是一个在计算机世界里专门用来在**两点之间传输数据**的约定和规范。

3. 「超文本」

HTTP 传输的内容是「超文本」。

我们先来理解「文本」，在互联网早期的时候只是简单的字符文字，但现在「文本」的涵义已经可以扩展为图片、视频、压缩包等，在 HTTP 眼里这些都算作「文本」。

再来理解「超文本」，它就是**超越了普通文本的文本**，它是文字、图片、视频等的混合体，最关键有超链接，能从一个超文本跳转到另外一个超文本。

HTML 就是最常见的超文本了，它本身只是纯文字文件，但内部用很多标签定义了图片、视频等的链接，再经过浏览器的解释，呈现给我们的就是一个文字、有画面的网页了。

OK，经过了对 HTTP 里这三个名词的详细解释，就可以给出比「超文本传输协议」这七个字更准确更有技术含量的答案：

HTTP 是一个在计算机世界里专门在「两点」之间「传输」文字、图片、音频、视频等「超文本」数据的「约定和规范」。

> 那「HTTP 是用于从互联网服务器传输超文本到本地浏览器的协议，这种说法正确吗?

这种说法是**不正确**的。因为也可以是「服务器< -- >服务器」，所以采用**两点之间**的描述会更准确。

> HTTP 常见的状态码，有哪些?

五大类 HTTP 状态码

	具体含义	常见的状态码
1××	提示信息，表示目前是协议处理的中间状态，还需要后续的操作；	
2××	**成功，报文已经收到并被正确处理；**	200、204、206
3××	**重定向，资源位置发生变动，需要客户端重新发送请求；**	301、302、304
4××	**客户端错误，请求报文有误，服务器无法处理；**	400、403、404
5××	**服务器错误，服务器在处理请求时内部发生了错误。**	500、501、502、503

1xx

`1xx` 类状态码属于**提示信息**，是协议处理中的一种中间状态，实际用到的比较少。

2xx

`2xx` 类状态码表示服务器**成功**处理了客户端的请求，也是我们最愿意看到的状态。

「**200 OK**」是最常见的成功状态码，表示一切正常。如果是非 `HEAD` 请求，服务器返回的响应头都会有 body 数据。

「**204 No Content**」也是常见的成功状态码，与 200 OK 基本相同，但响应头没有 body 数据。

「**206 Partial Content**」是应用于 HTTP 分块下载或断点续传，表示响应返回的 body 数据并不是资源的全部，而是其中的一部分，也是服务器处理成功的状态。

3xx

`3xx` 类状态码表示客户端请求的资源发送了变动，需要客户端用新的 URL 重新发送请求获取资源，也就是**重定向**。

「**301 Moved Permanently**」表示永久重定向，说明请求的资源已经不存在了，需改用新的 URL 再次访问。

「**302 Found**」表示临时重定向，说明请求的资源还在，但暂时需要用另一个 URL 来访问。

301 和 302 都会在响应头里使用字段 `Location`，指明后续要跳转的 URL，浏览器会自动重定向新的 URL。

「**304 Not Modified**」不具有跳转的含义，表示资源未修改，重定向已存在的缓冲文件，也称缓存重定向，用于缓存控制。

4xx

`4xx` 类状态码表示客户端发送的**报文有误**，服务器无法处理，也就是错误码的含义。

「**400 Bad Request**」表示客户端请求的报文有错误，但只是个笼统的错误。

「**403 Forbidden**」表示服务器禁止访问资源，并不是客户端的请求出错。

「**404 Not Found**」表示请求的资源在服务器上不存在或未找到，所以无法提供给客户端。

5xx

`5xx` 类状态码表示客户端请求报文正确，但是**服务器处理时内部发生了错误**，属于服务器端的错误码。

「**500 Internal Server Error**」与 400 类型，是个笼统通用的错误码，服务器发生了什么错误，我们并不知道。

「**501 Not Implemented**」表示客户端请求的功能还不支持，类似“即将开业，敬请期待”的意思。

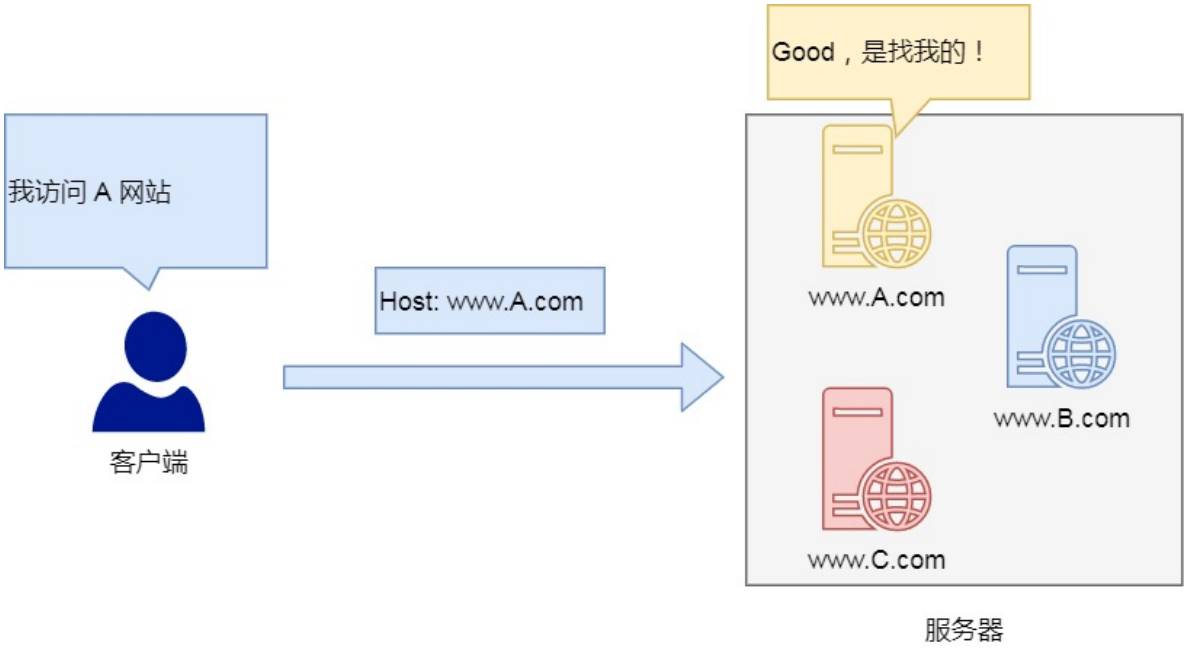
「**502 Bad Gateway**」通常是服务器作为网关或代理时返回的错误码，表示服务器自身工作正常，访问后端服务器发生了错误。

「**503 Service Unavailable**」表示服务器当前很忙，暂时无法响应服务器，类似“网络服务正忙，请稍后重试”的意思。

> http 常见字段有哪些?

Host 字段

客户端发送请求时，用来指定服务器的域名。

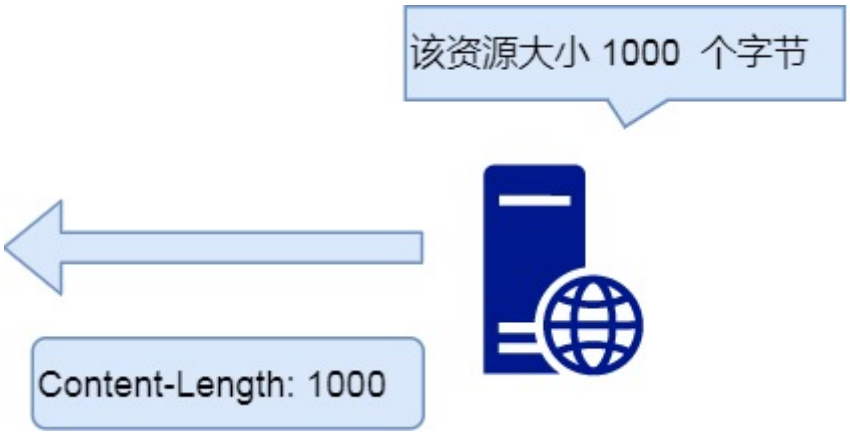



```
Host: www.A.com
```

有了 `Host` 字段，就可以将请求发往「同一台」服务器上的不同网站。

Content-Length 字段

服务器在返回数据时，会有 `Content-Length` 字段，表明本次回应的数据长度。

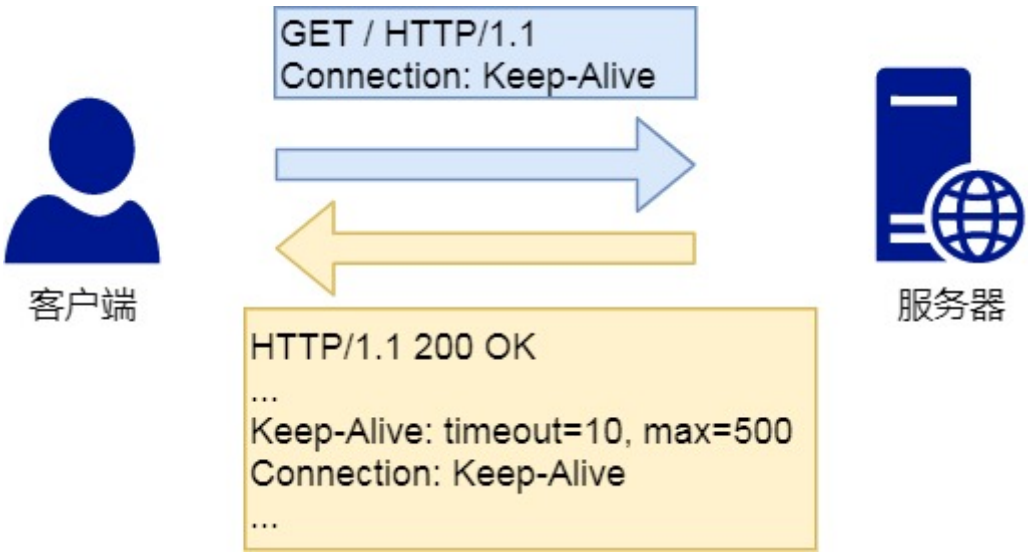


```
Content-Length: 1000
```

如上面则是告诉浏览器，本次服务器回应的数据长度是 1000 个字节，后面的字节就属于下一个回应了。

Connection 字段

`Connection` 字段最常用于客户端要求服务器使用 TCP 持久连接，以便其他请求复用。



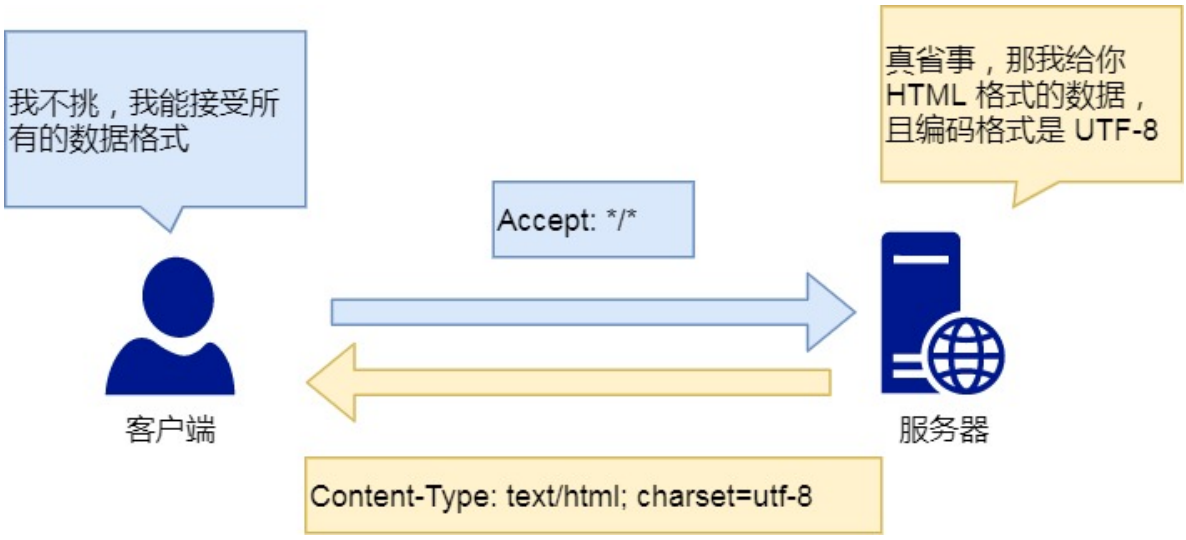
HTTP/1.1 版本的默认连接都是持久连接，但为了兼容老版本的 HTTP，需要指定 `Connection` 首部字段的值为 `Keep-Alive` 。

```
Connection: keep-alive
```

一个可以复用的 TCP 连接就建立了，直到客户端或服务器主动关闭连接。但是，这不是标准字段。

Content-Type 字段

`Content-Type` 字段用于服务器回应时，告诉客户端，本次数据是什么格式。




```
Content-Type: text/html; charset=utf-8
```

上面的类型表明，发送的是网页，而且编码是UTF-8。

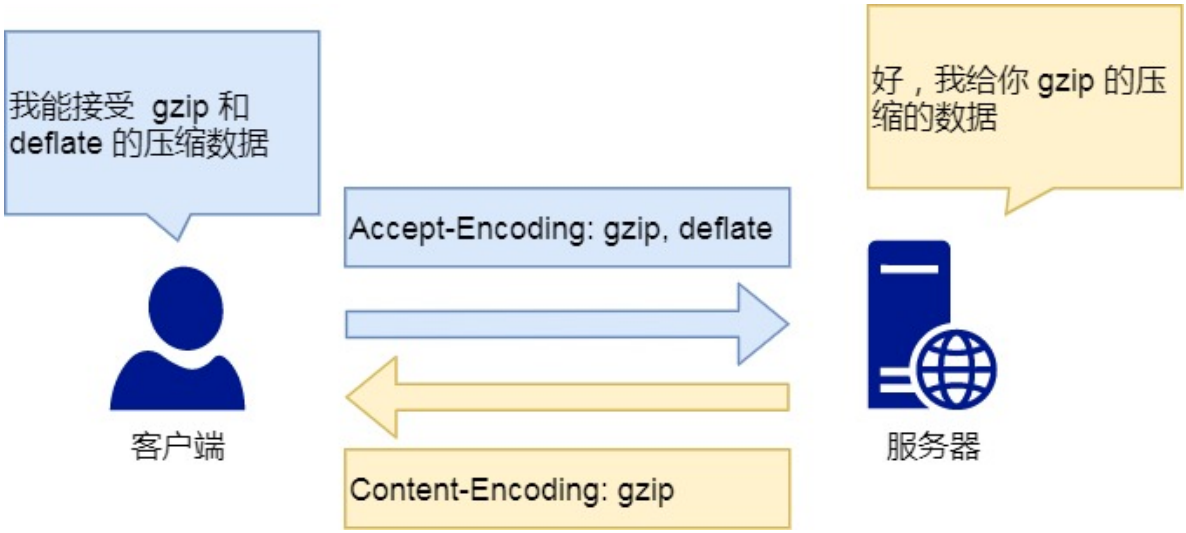
客户端请求的时候，可以使用 `Accept` 字段声明自己可以接受哪些数据格式。

```
Accept: */*
```

上面代码中，客户端声明自己可以接受任何格式的数据。

Content-Encoding 字段

`Content-Encoding` 字段说明数据的压缩方法。表示服务器返回的数据使用了什么压缩格式




```
Content-Encoding: gzip
```

上面表示服务器返回的数据采用了 gzip 方式压缩，告知客户端需要用此方式解压。

客户端在请求时，用 `Accept-Encoding` 字段说明自己可以接受哪些压缩方法。

```
Accept-Encoding: gzip, deflate
```

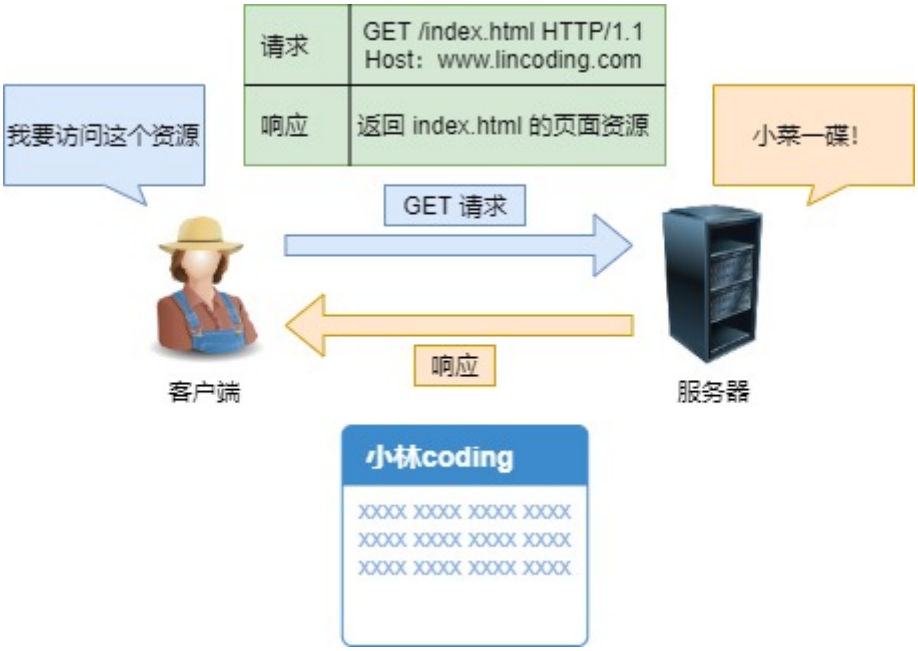
---

02 GET 与 POST

> 说一下 GET 和 POST 的区别?

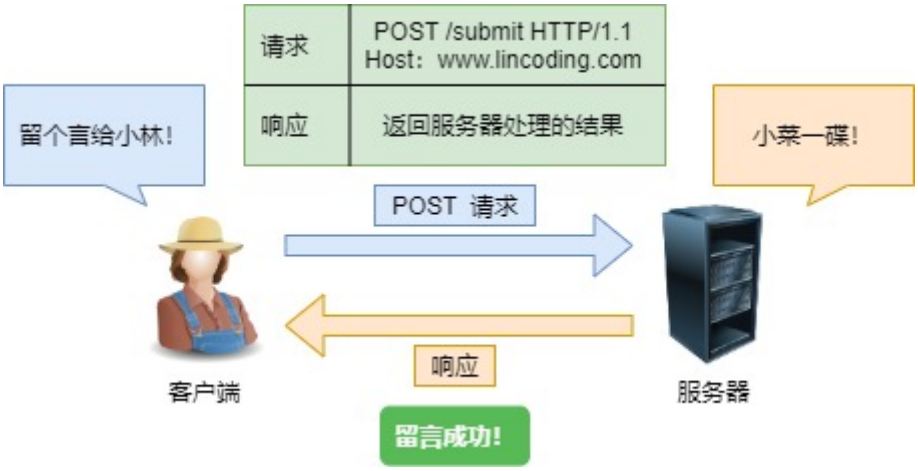
`Get` 方法的含义是请求**从服务器获取资源**，这个资源可以是静态的文本、页面、图片视频等。

比如，你打开我的文章，浏览器就会发送 GET 请求给服务器，服务器就会返回文章的所有文字及资源。



而 `POST` 方法则是相反操作，它向 `URI` 指定的资源提交数据，数据就放在报文的 body 里。

比如，你在我文章底部，敲入了留言后点击「提交」（**暗示你们留言**），浏览器就会执行一次 POST 请求，把你的留言文字放进了报文 body 里，然后拼接好 POST 请求头，通过 TCP 协议发送给服务器。



> GET 和 POST 方法都是安全和幂等的吗?

先说明下安全和幂等的概念：

- 在 HTTP 协议里，所谓的「安全」是指请求方法不会「破坏」服务器上的资源。
- 所谓的「幂等」，意思是多次执行相同的操作，结果都是「相同」的。

那么很明显 **GET 方法就是安全且幂等的**，因为它是「只读」操作，无论操作多少次，服务器上的数据都是安全的，且每次的结果都是相同的。

POST 因为是「新增或提交数据」的操作，会修改服务器上的资源，所以是**不安全**的，且多次提交数据就会创建多个资源，所以**不是幂等**的。

---

03 HTTP 特性

> 你知道的 HTTP（1.1） 的优点有哪些，怎么体现的?

HTTP 最凸出的优点是「简单、灵活和易于扩展、应用广泛和跨平台」。

1. 简单

HTTP 基本的报文格式就是 `header + body`，头部信息也是 `key-value` 简单文本的形式，**易于理解**，降低了学习和使用的门槛。

2. 灵活和易于扩展

HTTP协议里的各类请求方法、URI/URL、状态码、头字段等每个组成要求都没有被固定死，都允许开发人员**自定义和扩充**。

同时 HTTP 由于是工作在应用层（ `OSI` 第七层），则它**下层可以随意变化**。

HTTPS 也就是在 HTTP 与 TCP 层之间增加了 SSL/TLS 安全传输层，HTTP/3 甚至把 TCP 层换成了基于 UDP 的 QUIC。

3. 应用广泛和跨平台

互联网发展至今，HTTP 的应用范围非常的广泛，从台式机的浏览器到手机上的各种 APP，从看新闻、刷贴吧到购物、理财、吃鸡，HTTP 的应用**片地开花**，同时天然具有**跨平台**的优越性。

> 那它的缺点呢?

HTTP 协议里有优缺点一体的**双刃剑**，分别是「无状态、明文传输」，同时还有一大缺点「不安全」。

1. 无状态双刃剑

无状态的**好处**，因为服务器不会去记忆 HTTP 的状态，所以不需要额外的资源来记录状态信息，这能减轻服务器的负担，能够把更多的 CPU 和内存用来对外提供服务。

无状态的**坏处**，既然服务器没有记忆能力，它在完成有关联性的操作时会非常麻烦。

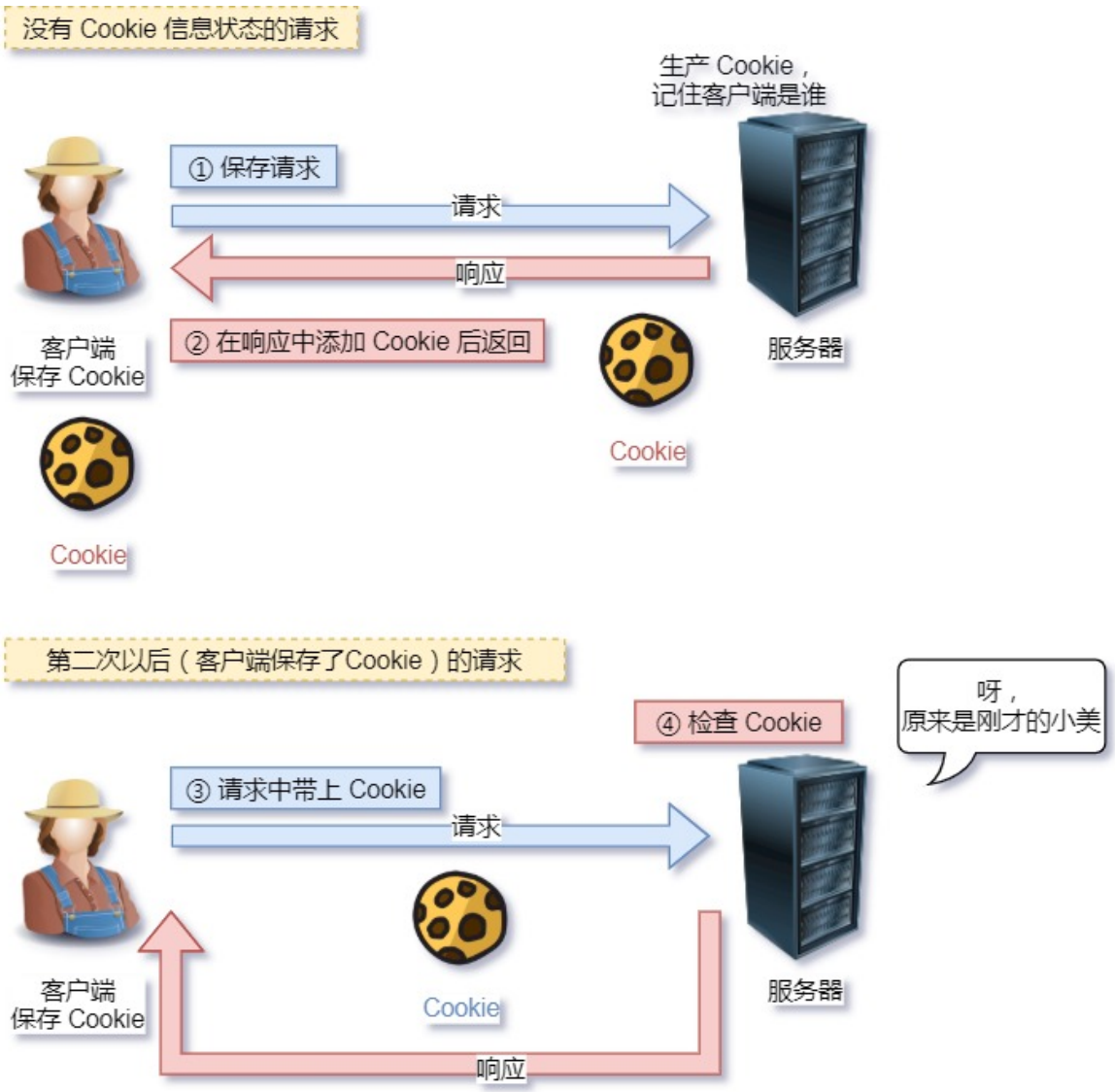
例如登录->添加购物车->下单->结算->支付，这系列操作都要知道用户的身份才行。但服务器不知道这些请求是有关联的，每次都要问一遍身份信息。

这样每操作一次，都要验证信息，这样的购物体验还能愉快吗？别问，问就是**酸爽**！

对于无状态的问题，解法方案有很多种，其中比较简单的方式用 **Cookie** 技术。

`Cookie` 通过在请求和响应报文中写入 Cookie 信息来控制客户端的状态。

相当于，**在客户端第一次请求后，服务器会下发一个装有客户信息的「小贴纸」，后续客户端请求服务器的时候，带上「小贴纸」，服务器就能认得了了**，



2. 明文传输双刃剑

明文意味着在传输过程中的信息，是可方便阅读的，通过浏览器的 F12 控制台或 Wireshark 抓包都可以直接肉眼查看，为我们调试工作带了极大的便利性。

但是这正是这样，HTTP 的所有信息都暴露在了光天化日下，相当于**信息裸奔**。在传输的漫长的过程中，信息的内容都毫无隐私可言，很容易就能被窃取，如果里面有你的账号密码信息，那**你号没了**。

3. 不安全

HTTP 比较严重的缺点就是不安全：

- 通信使用明文（不加密），内容可能会被窃听。比如，**账号信息容易泄漏，那你号没了。**
- 不验证通信方的身份，因此有可能遭遇伪装。比如，**访问假的淘宝、拼多多，那你钱没了。**
- 无法证明报文的完整性，所以有可能已遭篡改。比如，**网页上植入垃圾广告，视觉污染，眼没了。**

HTTP 的安全问题，可以用 HTTPS 的方式解决，也就是通过引入 SSL/TLS 层，使得在安全上达到了极致。

> 那你再说下 HTTP/1.1 的性能如何?

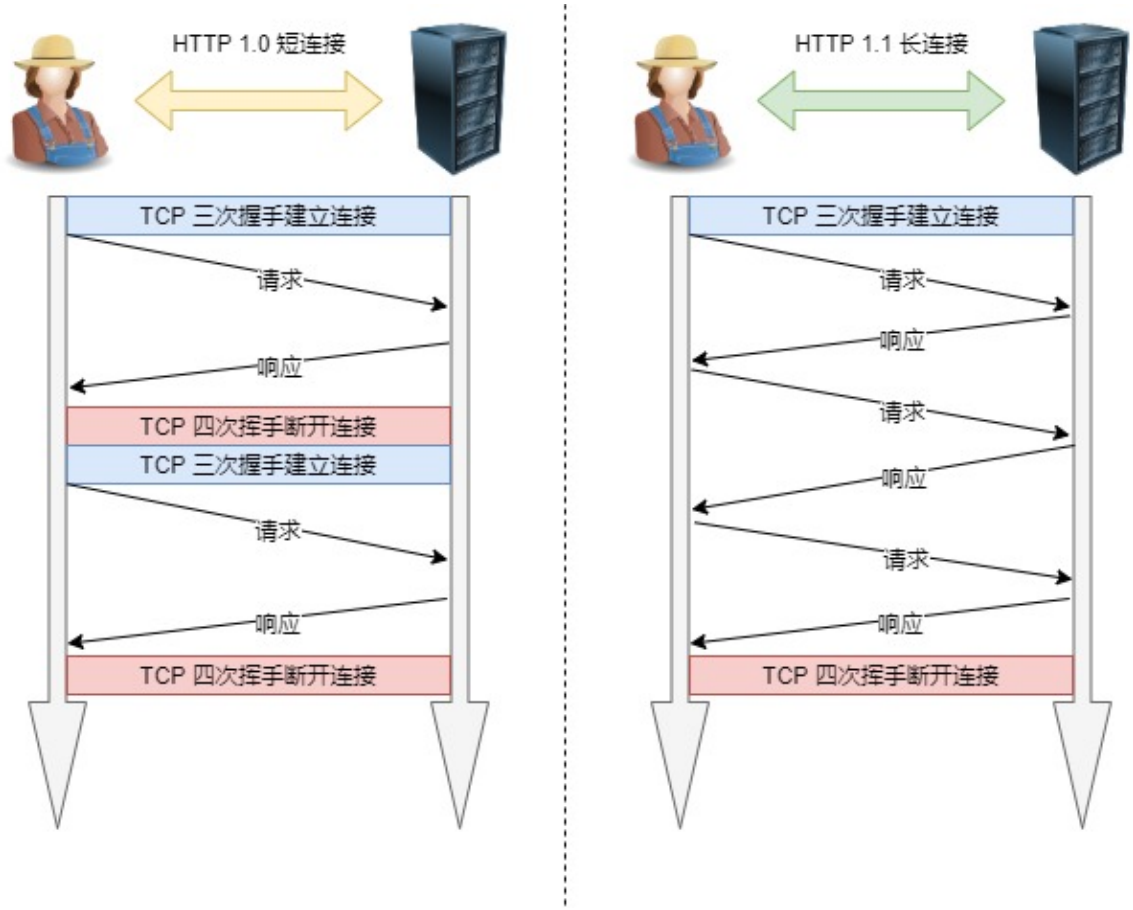
HTTP 协议是基于 **TCP/IP**，并且使用了「**请求 - 应答**」的通信模式，所以性能的关键就在这**两点**里。

1. 长连接

早期 HTTP/1.0 性能上的一个很大的问题，那就是每发起一个请求，都要新建一次 TCP 连接（三次握手），而且是串行请求，做了无谓的 TCP 连接建立和断开，增加了通信开销。

为了解决上述 TCP 连接问题，HTTP/1.1 提出了**长连接**的通信方式，也叫持久连接。这种方式的好处在于减少了 TCP 连接的重复建立和断开所造成的额外开销，减轻了服务器端的负载。

持久连接的特点是，只要任意一端没有明确提出断开连接，则保持 TCP 连接状态。

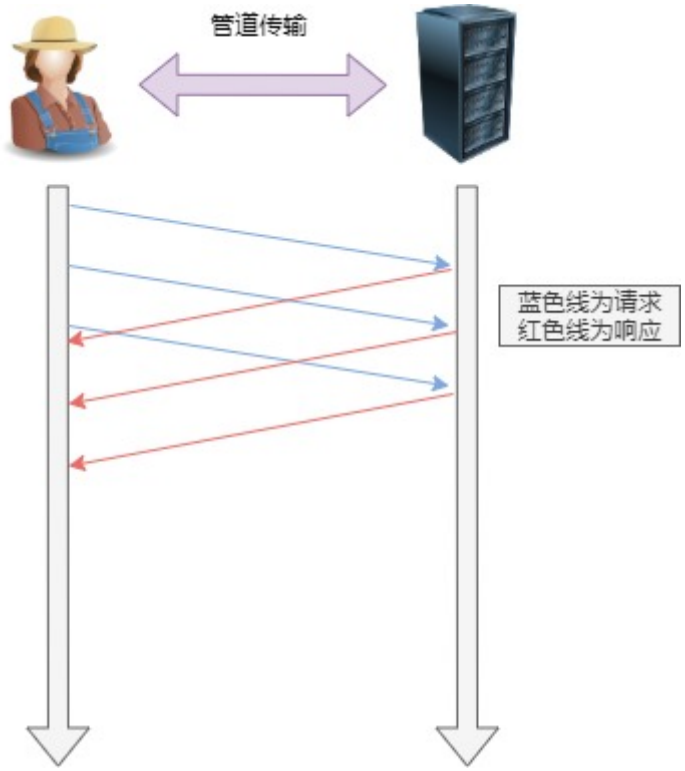


2. 管道网络传输

HTTP/1.1 采用了长连接的方式，这使得管道（pipeline）网络传输成为了可能。

即可在同一个 TCP 连接里面，客户端可以发起多个请求，只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以**减少整体的响应时间。**

举例来说，客户端需要请求两个资源。以前的做法是，在同一个TCP连接里面，先发送 A 请求，然后等待服务器做出回应，收到后再发出 B 请求。管道机制则是允许浏览器同时发出 A 请求和 B 请求。

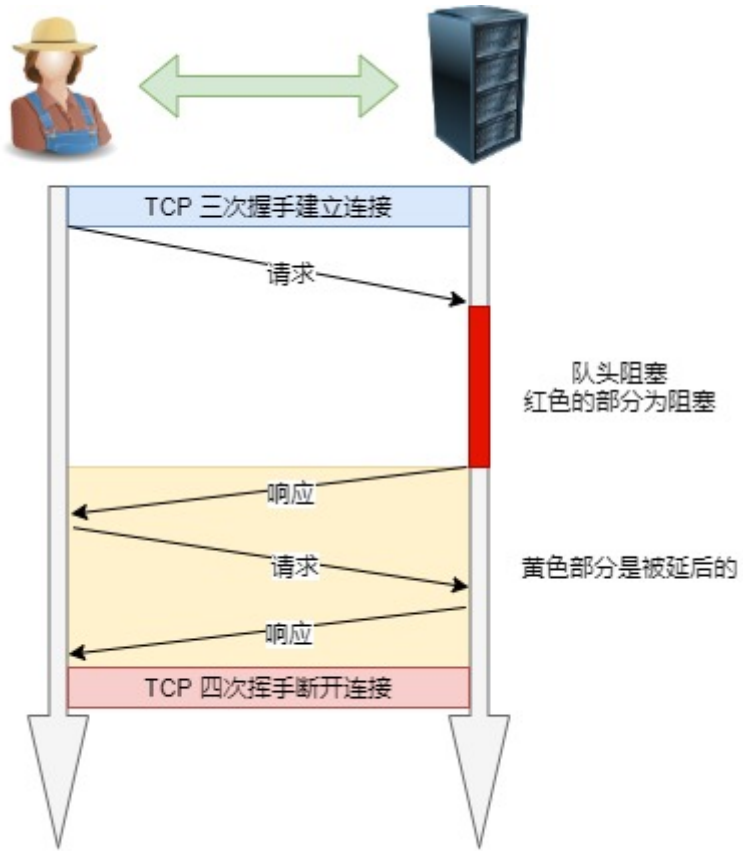


但是服务器还是按照**顺序**，先回应 A 请求，完成后再回应 B 请求。要是前面的回应特别慢，后面就会有许多请求排队等着。这称为「队头堵塞」。

3. 队头阻塞

「请求 - 应答」的模式加剧了 HTTP 的性能问题。

因为当顺序发送的请求序列中的一个请求因为某种原因被阻塞时，在后面排队的所有请求也一同被阻塞了，会招致客户端一直请求不到数据，这也就是「**队头阻塞**」。**好比上班的路上塞车**。



总之 HTTP/1.1 的性能一般般，后续的 HTTP/2 和 HTTP/3 就是在优化 HTTP 的性能。

---

04 HTTP 与 HTTPS

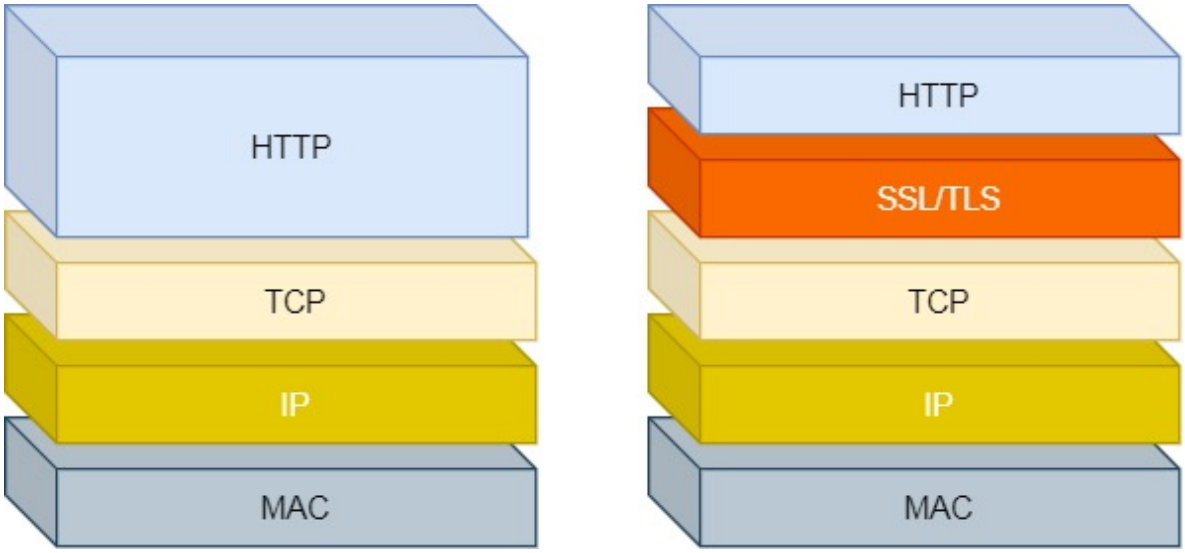
> HTTP 与 HTTPS 有哪些区别?

1. HTTP 是超文本传输协议，信息是明文传输，存在安全风险的问题。HTTPS 则解决 HTTP 不安全的缺陷，在 TCP 和 HTTP 网络层之间加入了 SSL/TLS 安全协议，使得报文能够加密传输。
2. HTTP 连接建立相对简单， TCP 三次握手之后便可进行 HTTP 的报文传输。而 HTTPS 在 TCP 三次握手之后，还需进行 SSL/TLS 的握手过程，才可进入加密报文传输。
3. HTTP 的端口号是 80，HTTPS 的端口号是 443。
4. HTTPS 协议需要向 CA（证书权威机构）申请数字证书，来保证服务器的身份是可信的。

> HTTPS 解决了 HTTP 的哪些问题?

HTTP 由于是明文传输，所以安全上存在以下三个风险：

- **窃听风险**，比如通信链路上可以获取通信内容，用户号容易没。
- **篡改风险**，比如强制植入垃圾广告，视觉污染，用户眼容易瞎。
- **冒充风险**，比如冒充淘宝网站，用户钱容易没。



HTTP**S** 在 HTTP 与 TCP 层之间加入了 `SSL/TLS` 协议，可以很好的解决了上述的风险：

- **信息加密**：交互信息无法被窃取，但你的号会因为「自身忘记」账号而没。
- **校验机制**：无法篡改通信内容，篡改了就不能正常显示，但百度「竞价排名」依然可以搜索垃圾广告。
- **身份证书**：证明淘宝是真的淘宝网，但你的钱还是会因为「剁手」而没。

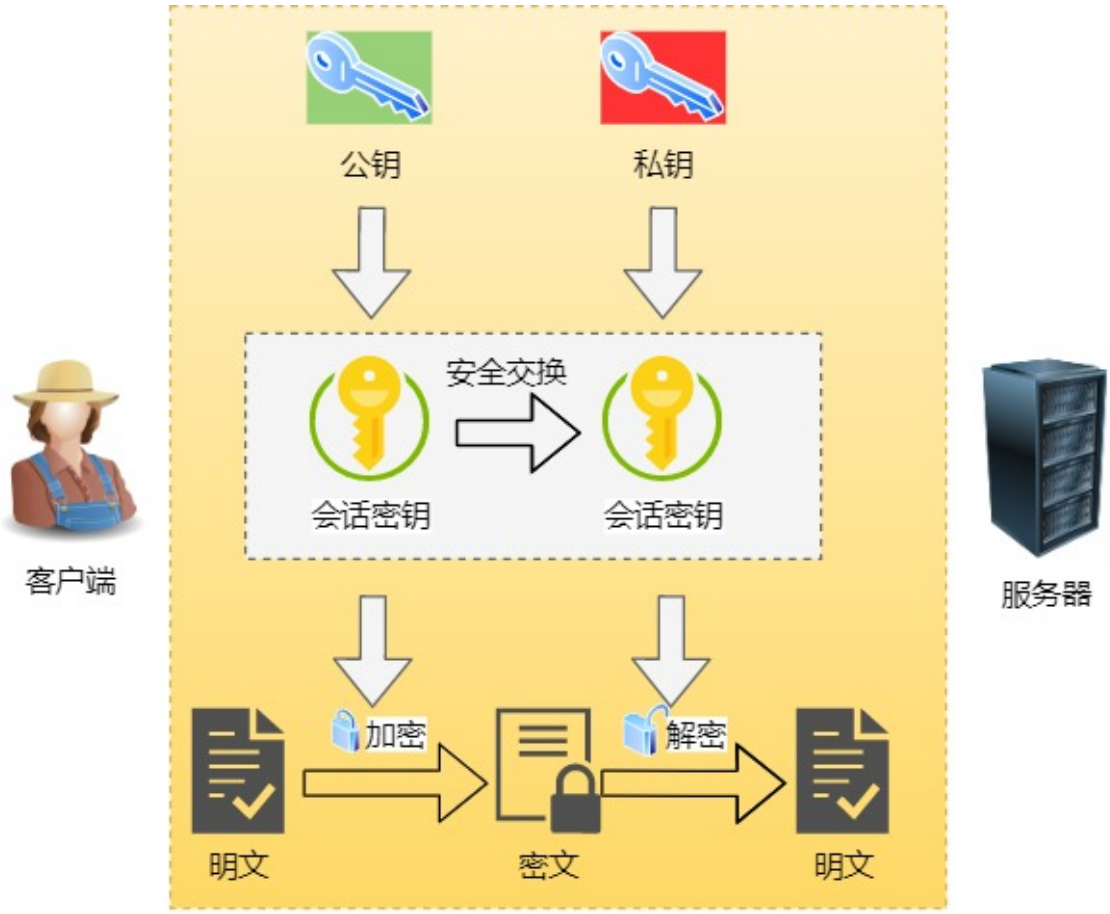
可见，只要自身不做「恶」，SSL/TLS 协议是能保证通信是安全的。

> HTTPS 是如何解决上面的三个风险的?

- **混合加密**的方式实现信息的**机密性**，解决了窃听的风险。
- **摘要算法**的方式来实现**完整性**，它能够为数据生成独一无二的「指纹」，指纹用于校验数据的完整性，解决了篡改的风险。
- 将服务器公钥放入到**数字证书**中，解决了冒充的风险。

1. 混合加密

通过**混合加密**的方式可以保证信息的**机密性**，解决了窃听的风险。



HTTPS 采用的是**对称加密**和**非对称加密**结合的「混合加密」方式：

- 在通信建立前采用**非对称加密**的方式交换「会话秘钥」，后续就不再使用非对称加密。
- 在通信过程中全部使用**对称加密**的「会话秘钥」的方式加密明文数据。

采用「混合加密」的方式的原因：

- **对称加密**只使用一个密钥，运算速度快，密钥必须保密，无法做到安全的密钥交换。
- **非对称加密**使用两个密钥：公钥和私钥，公钥可以任意分发而私钥保密，解决了密钥交换问题但速度慢。

2. 摘要算法

摘要算法用来实现**完整性**，能够为数据生成独一无二的「指纹」，用于校验数据的完整性，解决了篡改的风险。

客户端在发送明文之前会通过摘要算法算出明文的「指纹」，发送的时候把「指纹 + 明文」一同加密成密文后，发送给服务器，服务器解密后，用相同的摘要算法算出发送过来的明文，通过比较客户端携带的「指纹」和当前算出的「指纹」做比较，若「指纹」相同，说明数据是完整的。

3. 数字证书

客户端先向服务器端索要公钥，然后用公钥加密信息，服务器收到密文后，用自己的私钥解密。

这就存在些问题，如何保证公钥不被篡改和信任度?

所以这里就需要借助第三方权威机构 CA （数字证书认证机构），将**服务器公钥放在数字证书**（由数字证书认证机构颁发）中，只要证书是可信的，公钥就是可信的。

通过数字证书的方式保证服务器公钥的身份，解决冒充的风险。

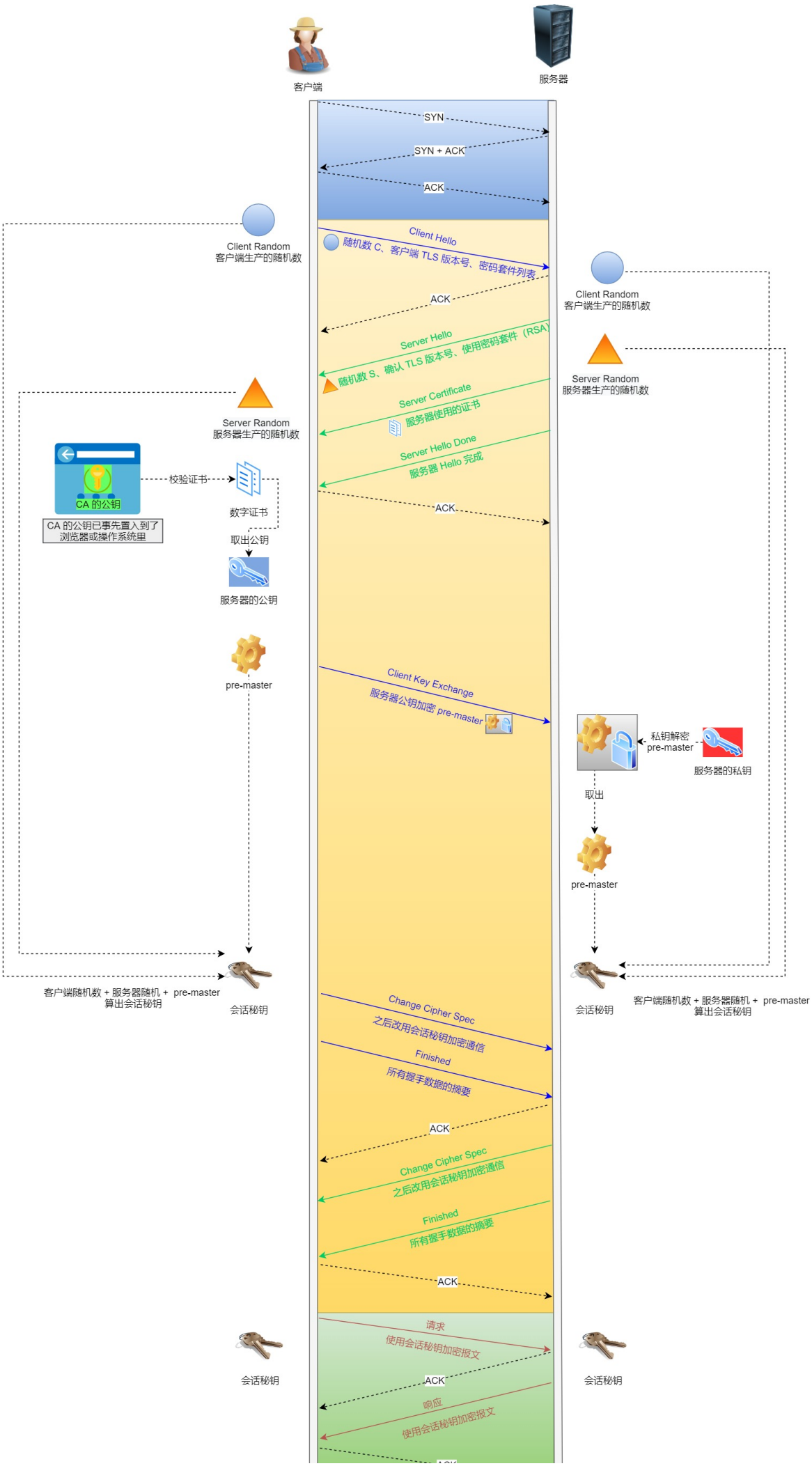
> HTTPS 是如何建立连接的？其间交互了什么?

SSL/TLS 协议基本流程：

- 客户端向服务器索要并验证服务器的公钥。
- 双方协商生产「会话秘钥」。
- 双方采用「会话秘钥」进行加密通信。

前两步也就是 SSL/TLS 的建立过程，也就是握手阶段。

SSL/TLS 的「握手阶段」涉及**四次**通信，可见下图：


客户端
服务器
SYN
SYN + ACK
ACK
Client Hello
随机数 C、客户端 TLS 版本号、密码套件列表
ACK
Server Hello
随机数 S、确认 TLS 版本号、使用密码套件（RSA）
Server Certificate
服务器使用的证书
Server Hello Done
服务器 Hello 完成
ACK
Client Key Exchange
服务器公钥加密 pre-master
Change Cipher Spec
之后改用会话秘钥加密通信
Finished
所有握手数据的摘要
ACK
Change Cipher Spec
之后改用会话秘钥加密通信
Finished
所有握手数据的摘要
ACK
请求
使用会话秘钥加密报文
ACK
响应
使用会话秘钥加密报文
Client Random
客户端生产的随机数
Server Random
服务器生产的随机数
CA 的公钥
CA 的公钥已事先置入到了浏览器或操作系统里
校验证书
数字证书
取出公钥
服务器的公钥
pre-master
客户端随机数 + 服务器随机 + pre-master
算出会话秘钥
会话秘钥
私钥解密 pre-master
服务器的私钥
取出
pre-master
会话秘钥
客户端随机数 + 服务器随机 + pre-master
算出会话秘钥
会话秘钥
会话秘钥

SSL/TLS 协议建立的详细流程：

1. ClientHello

首先，由客户端向服务器发起加密通信请求，也就是 `ClientHello` 请求。

在这一步，客户端主要向服务器发送以下信息：

（1）客户端支持的 SSL/TLS 协议版本，如 TLS 1.2 版本。

（2）客户端生产的随机数（`Client Random`），后面用于生产「会话秘钥」。

（3）客户端支持的密码套件列表，如 RSA 加密算法。

2. SeverHello

服务器收到客户端请求后，向客户端发出响应，也就是 `SeverHello` 。服务器回应的内容有如下内容：

（1）确认 SSL/ TLS 协议版本，如果浏览器不支持，则关闭加密通信。

（2）服务器生产的随机数（`Server Random`），后面用于生产「会话秘钥」。

（3）确认的密码套件列表，如 RSA 加密算法。

（4）服务器的数字证书。

3.客户端回应

客户端收到服务器的回应之后，首先通过浏览器或者操作系统中的 CA 公钥，确认服务器的数字证书的真实性。

如果证书没有问题，客户端会从数字证书中取出服务器的公钥，然后使用它加密报文，向服务器发送如下信息：

（1）一个随机数（`pre-master key`）。该随机数会被服务器公钥加密。

（2）加密通信算法改变通知，表示随后的信息都将用「会话秘钥」加密通信。

（3）客户端握手结束通知，表示客户端的握手阶段已经结束。这一项同时把之前所有内容的发生的数据做个摘要，用来供服务端校验。

上面第一项的随机数是整个握手阶段的第三个随机数，这样服务器和客户端就同时有三个随机数，接着就用双方协商的加密算法，**各自生成**本次通信的「会话秘钥」。

4. 服务器的最后回应

服务器收到客户端的第三个随机数（`pre-master key`）之后，通过协商的加密算法，计算出本次通信的「会话秘钥」。然后，向客户端发生最后的信息：

（1）加密通信算法改变通知，表示随后的信息都将用「会话秘钥」加密通信。

（2）服务器握手结束通知，表示服务器的握手阶段已经结束。这一项同时把之前所有内容的发生的数据做个摘要，用来供客户端校验。

至此，整个 SSL/TLS 的握手阶段全部结束。接下来，客户端与服务器进入加密通信，就完全是使用普通的 HTTP 协议，只不过用「会话秘钥」加密内容。

---

05 HTTP/1.1、HTTP/2、HTTP/3 演变

> 说说 HTTP/1.1 相比 HTTP/1.0 提高了什么性能?

HTTP/1.1 相比 HTTP/1.0 性能上的改进：

- 使用 TCP 长连接的方式改善了 HTTP/1.0 短连接造成的性能开销。
- 支持管道（pipeline）网络传输，只要第一个请求发出去了，不必等其回来，就可以发第二个请求出去，可以减少整体的响应时间。

但 HTTP/1.1 还是有性能瓶颈：

- 请求 / 响应头部（Header）未经压缩就发送，首部信息越多延迟越大。只能压缩 `Body` 的部分；
- 发送冗长的首部。每次互相发送相同的首部造成的浪费较多；
- 服务器是按请求的顺序响应的，如果服务器响应慢，会招致客户端一直请求不到数据，也就是队头阻塞；
- 没有请求优先级控制；
- 请求只能从客户端开始，服务器只能被动响应。

> 那上面的 HTTP/1.1 的性能瓶颈，HTTP/2 做了什么优化?

HTTP/2 协议是基于 HTTPS 的，所以 HTTP/2 的安全性也是有保障的。

那 HTTP/2 相比 HTTP/1.1 性能上的改进：

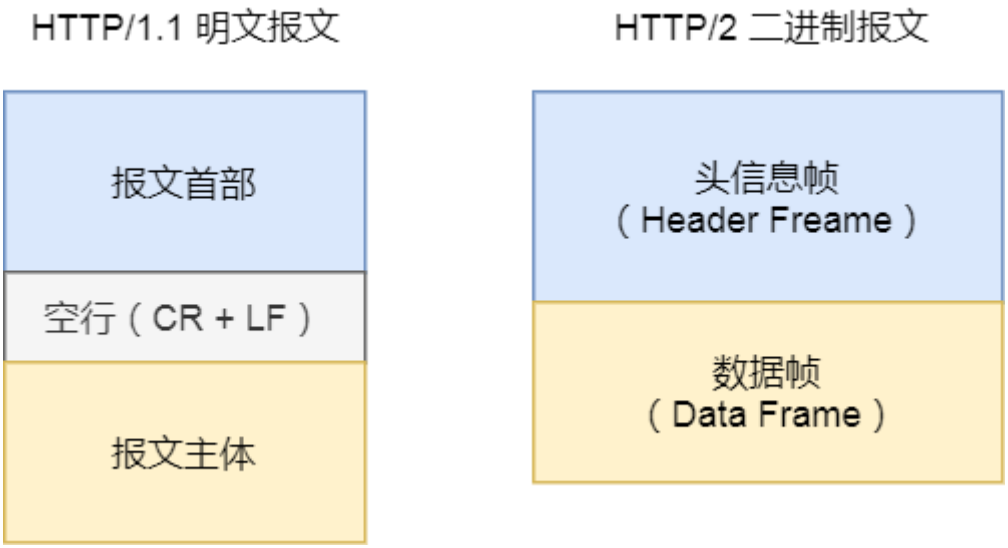
1. 头部压缩

HTTP/2 会**压缩头**（Header）如果你同时发出多个请求，他们的头是一样的或是相似的，那么，协议会帮你**消除重复的部分**。

这就是所谓的 `HPACK` 算法：在客户端和服务器同时维护一张头信息表，所有字段都会存入这个表，生成一个索引号，以后就不发送同样字段了，只发送索引号，这样就**提高速度**了。

2. 二进制格式

HTTP/2 不再像 HTTP/1.1 里的纯文本形式的报文，而是全面采用了**二进制格式**，头信息和数据体都是二进制，并且统称为帧（frame）：**头信息帧和数据帧**。



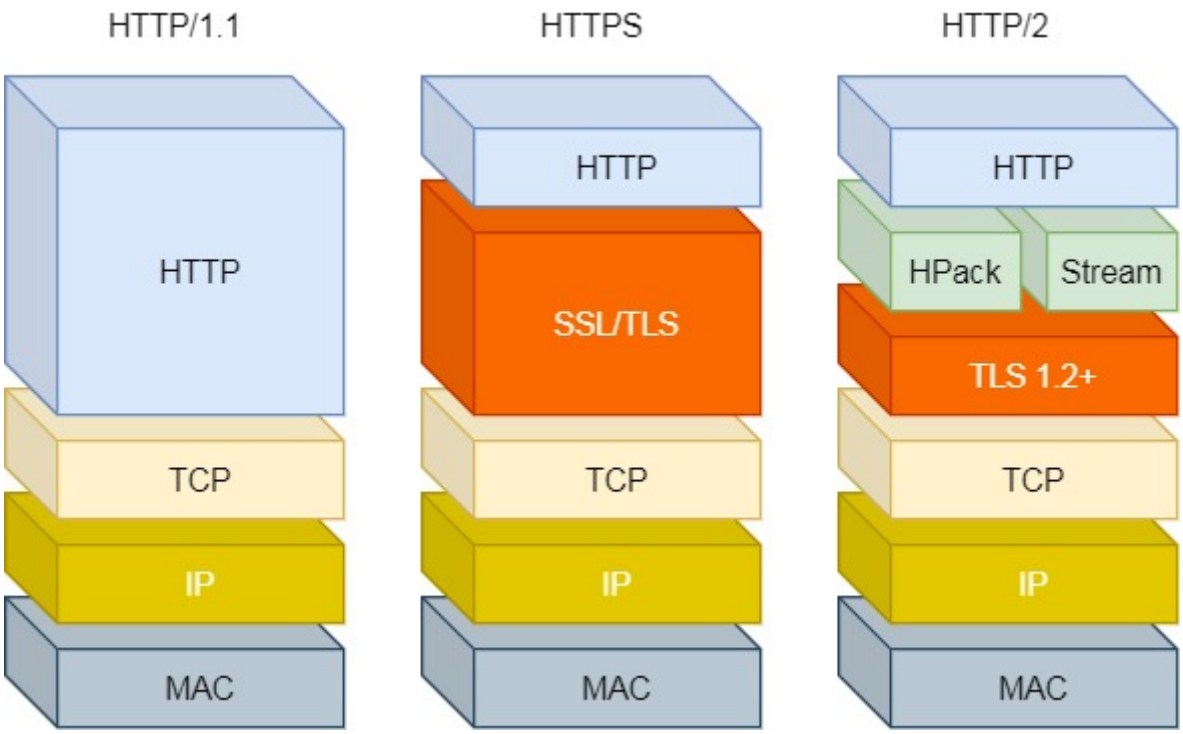
这样虽然对人不友好，但是对计算机非常友好，因为计算机只懂二进制，那么收到报文后，无需再将明文的报文转成二进制，而是直接解析二进制报文，这**增加了数据传输的效率**。

3. 数据流

HTTP/2 的数据包不是按顺序发送的，同一个连接里面连续的数据包，可能属于不同的回应。因此，必须要对数据包做标记，指出它属于哪个回应。

每个请求或回应的所有数据包，称为一个数据流（Stream）。每个数据流都标记着一个独一无二的编号，其中规定客户端发出的数据流编号为奇数， 服务器发出的数据流编号为偶数

客户端还可以**指定数据流的优先级**。优先级高的请求，服务器就先响应该请求。

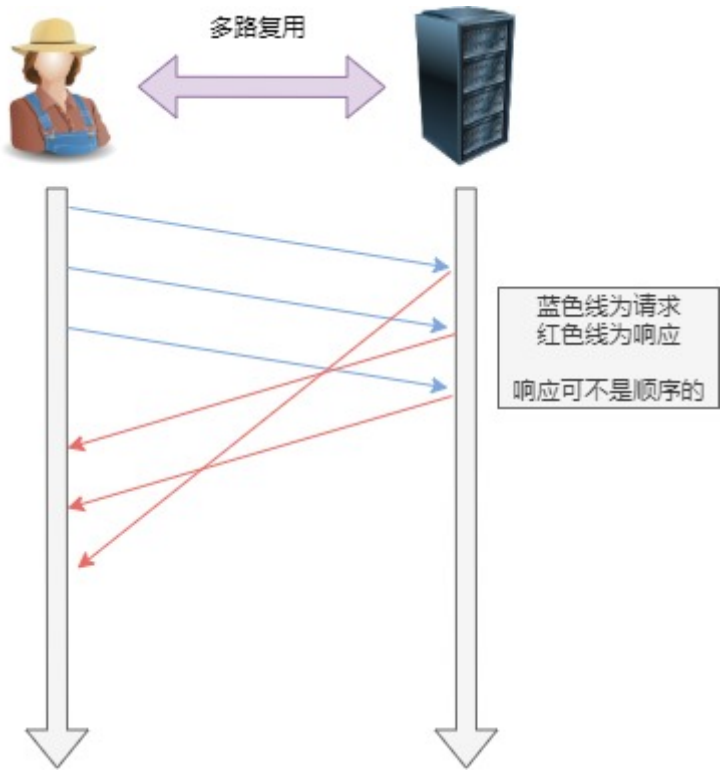


4. 多路复用

HTTP/2 是可以在**一个连接中并发多个请求或回应，而不用按照顺序一一对应**。

移除了 HTTP/1.1 中的串行请求，不需要排队等待，也就不会再出现「队头阻塞」问题，**降低了延迟，大幅度提高了连接的利用率**。

举例来说，在一个 TCP 连接里，服务器收到了客户端 A 和 B 的两个请求，如果发现 A 处理过程非常耗时，于是就回应 A 请求已经处理好的部分，接着回应 B 请求，完成后，再回应 A 请求剩下的部分。



5. 服务器推送

HTTP/2 还在一定程度上改善了传统的「请求 - 应答」工作模式，服务不再是被动地响应，也可以**主动**向客户端发送消息。

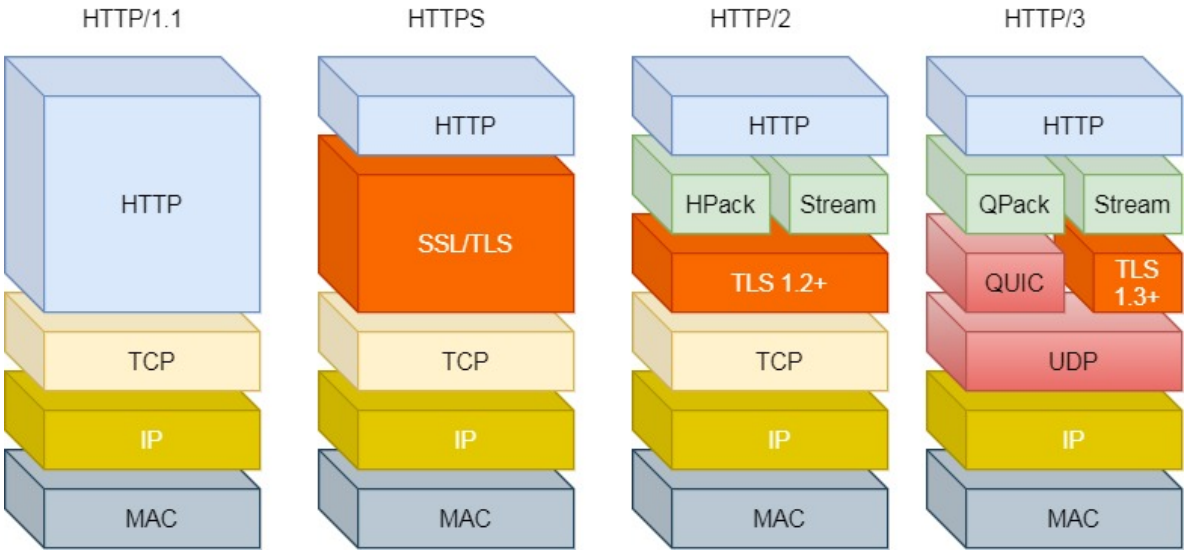
举例来说，在浏览器刚请求 HTML 的时候，就提前把可能会用到的 JS、CSS 文件等静态资源主动发给客户端，**减少延时的等待**，也就是服务器推送（Server Push，也叫 Cache Push）。

> HTTP/2 有哪些缺陷？HTTP/3 做了哪些优化？

HTTP/2 主要的问题在于，多个 HTTP 请求在复用一个 TCP 连接，下层的 TCP 协议是不知道有多少个 HTTP 请求的。所以一旦发生了丢包现象，就会触发 TCP 的重传机制，这样在一个 TCP 连接中的**所有的 HTTP 请求都必须等待这个丢了的包被重传回来**。

- HTTP/1.1 中的管道（ pipeline）传输中如果有一个请求阻塞了，那么队列后请求也统统被阻塞住了
- HTTP/2 多个请求复用一个TCP连接，一旦发生丢包，就会阻塞住所有的 HTTP 请求。

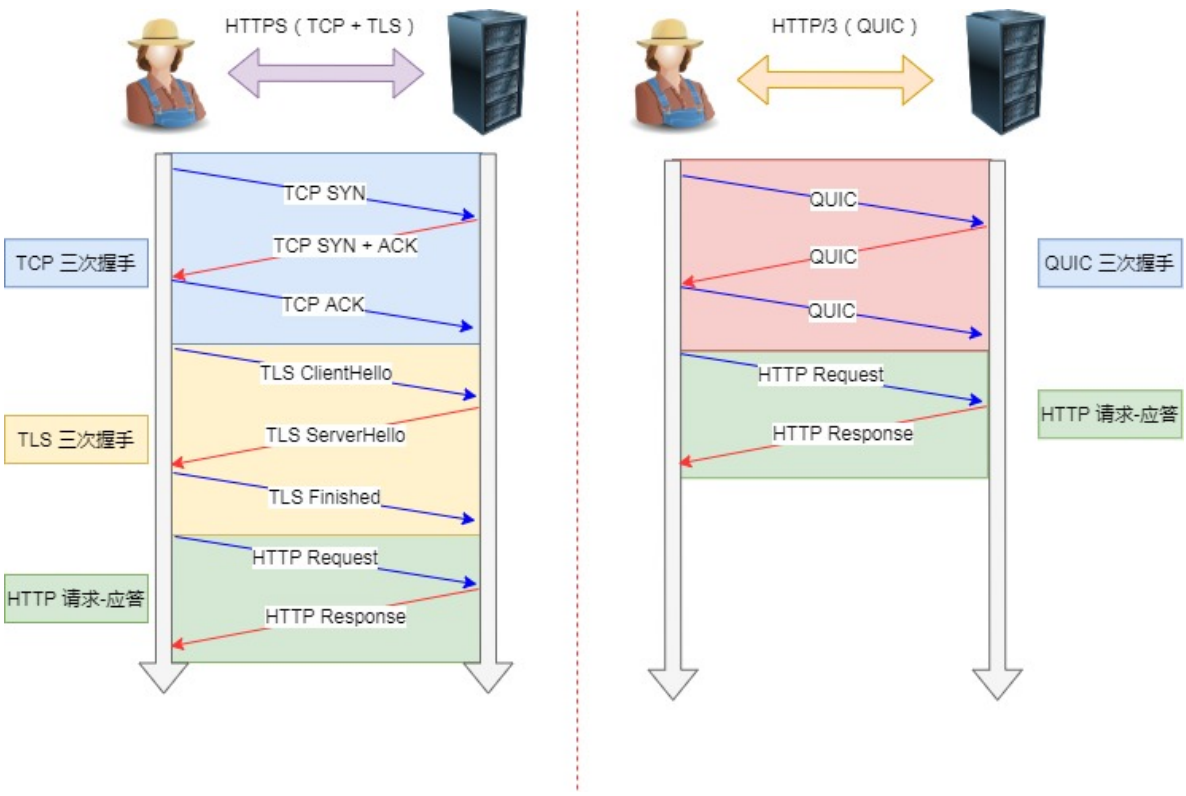
这都是基于 TCP 传输层的问题，所以 **HTTP/3 把 HTTP 下层的 TCP 协议改成了 UDP！**



UDP 发生是不管顺序，也不管丢包的，所以不会出现 HTTP/1.1 的队头阻塞 和 HTTP/2 的一个丢包全部重传问题。

大家都知道 UDP 是不可靠传输的，但基于 UDP 的 **QUIC 协议** 可以实现类似 TCP 的可靠性传输。

- QUIC 有自己的一套机制可以保证传输的可靠性的。当某个流发生丢包时，只会阻塞这个流，**其他流不会受到影响**。
- TLS3 升级成了最新的 `1.3` 版本，头部压缩算法也升级成了 `QPack`。
- HTTPS 要建立一个连接，要花费 6 次交互，先是建立三次握手，然后是 `TLS/1.3` 的三次握手。QUIC 直接把以往的 TCP 和 `TLS/1.3` 的 6 次交互**合并成了 3 次，减少了交互次数**。



所以， QUIC 是一个在 UDP 之上的**伪** TCP + TLS + HTTP/2 的多路复用的协议。

QUIC 是新协议，对于很多网络设备，根本不知道什么是 QUIC，只会当做 UDP，这样会出现新的问题。所以 HTTP/3 现在普及的进度非常的缓慢，不知道未来 UDP 是否能够逆袭 TCP。

巨人的肩膀

[1] 上野 宣.图解HTTP.人民邮电出版社.

[2] 罗剑锋.透视HTTP协议.极客时间.

[3] 陈皓.HTTP的前世今.酷壳CoolShell.https://coolshell.cn/articles/19840.html

[4] 阮一峰.HTTP 协议入门.阮一峰的网络日志.http://www.ruanyifeng.com/blog/2016/08/http.html

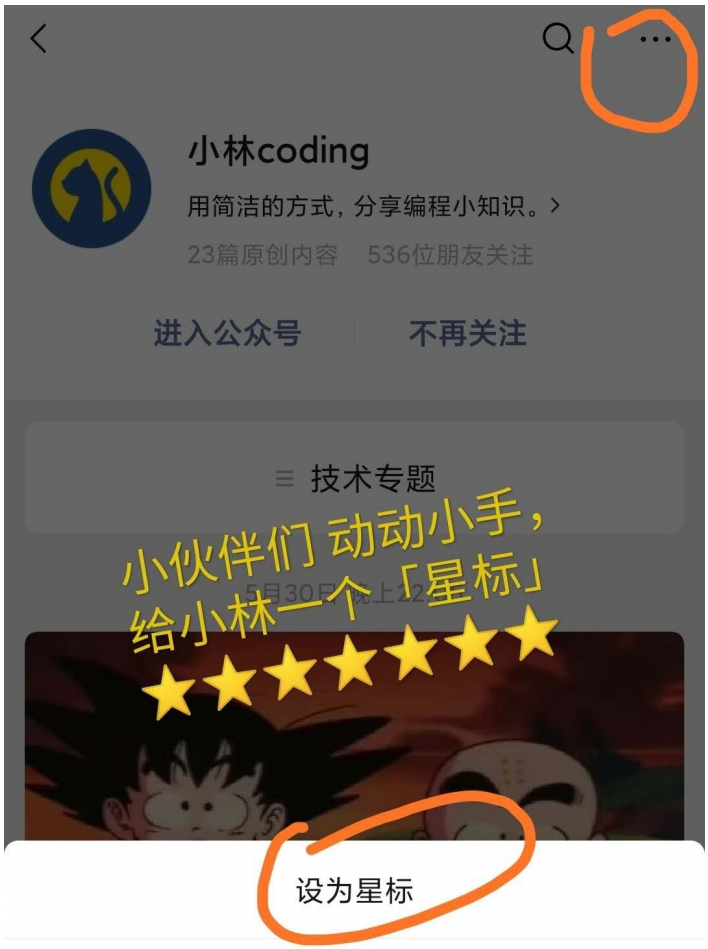
---

唠叨唠叨

本文的 30 张图片，都是从一条线两条线画出来，灰常的费劲，深切感受到画图也是个**体力活**啊！

爱偷懒的我其实不爱画图，但为了让大家能更好的理解，在跟自己无数次斗争后，踏上了耗时耗体力的画图的不归路，希望对你们有帮助！

小林是专为大家图解的工具人，Goodbye，我们下次见！



扫一扫
关注爱图解的
「小林coding」

推荐给朋友

读者问答

> 读者问：“https和http相比，就是传输的内容多了对称加密，可以这么理解吗？”

1. 建立连接时候：https 比 http多了 TLS 的握手过程；
2. 传输内容的时候：https 会把数据进行加密，通常是对称加密数据；

> 读者问：“ 我看文中 TLS 和 SSL 没有做区分，这两个需要区分吗？”

这两实际上是一个东西。

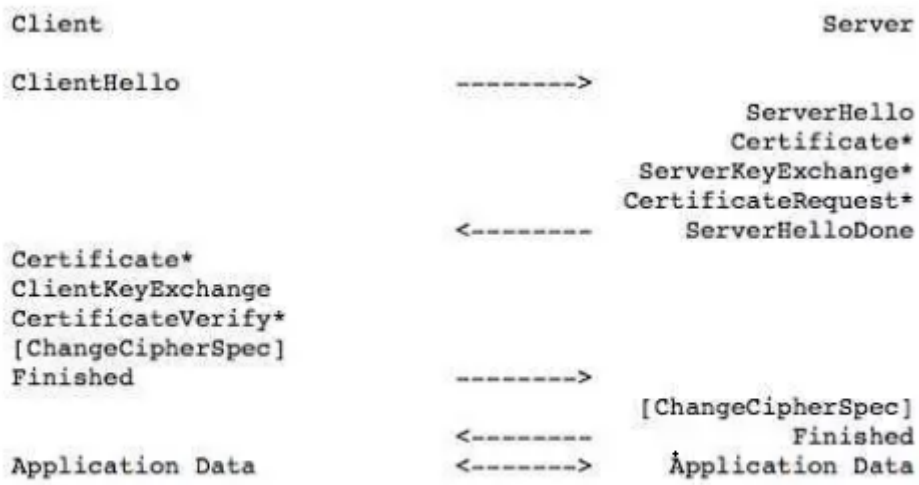
SSL 是洋文 “*Secure Sockets Layer* 的缩写，中文叫做「安全套接层」。它是在上世纪 90 年代中期，由网景公司设计的。

到了1999年，SSL 因为应用广泛，已经成为互联网上的事实标准。IETF 就在那年把 SSL 标准化。标准化之后的名称改为 TLS（是 “*Transport Layer Security*” 的缩写），中文叫做 「传输层安全协议」。

很多相关的文章都把这两者并列称呼（SSL/TLS），因为这两者可以视作同一个东西的不同阶段。

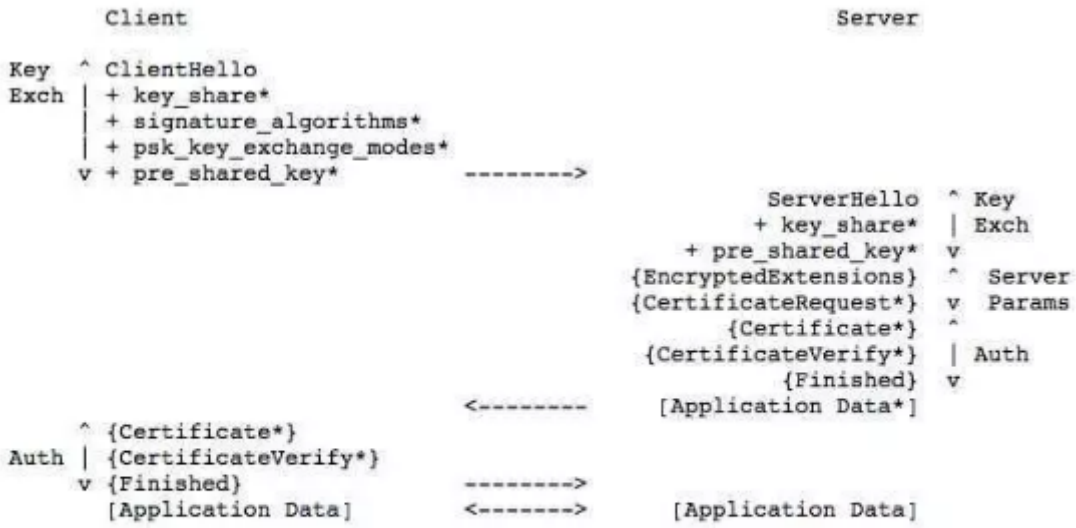
> 读者问：“为啥 ssl 的握手是 4 次？”

SSL/TLS 1.2 需要 4 握手，需要 2 个 RTT 的时延，我文中的图是把每个交互分开画了，实际上把他们合在一起发送，就是 4 次握手：



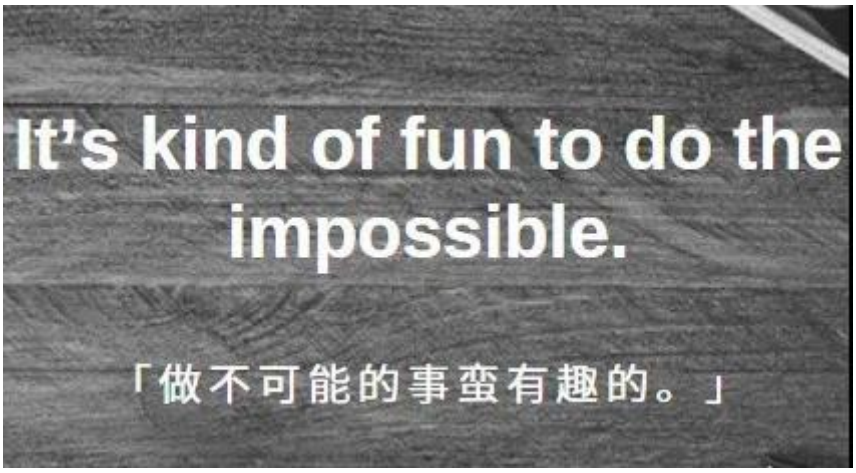
▵ TLS 1.2 完整握手框架（ 来自 RFC 5246 ）

另外， SSL/TLS 1.3 优化了过程，只需要 1 个 RTT 往返时延，也就是只需要 3 次握手：



▲ TLS 1.3 完整握手框架（来自 TLS 1.3 最新草案 ）

IP 基础知识全家桶，45 张图一套带走



前言

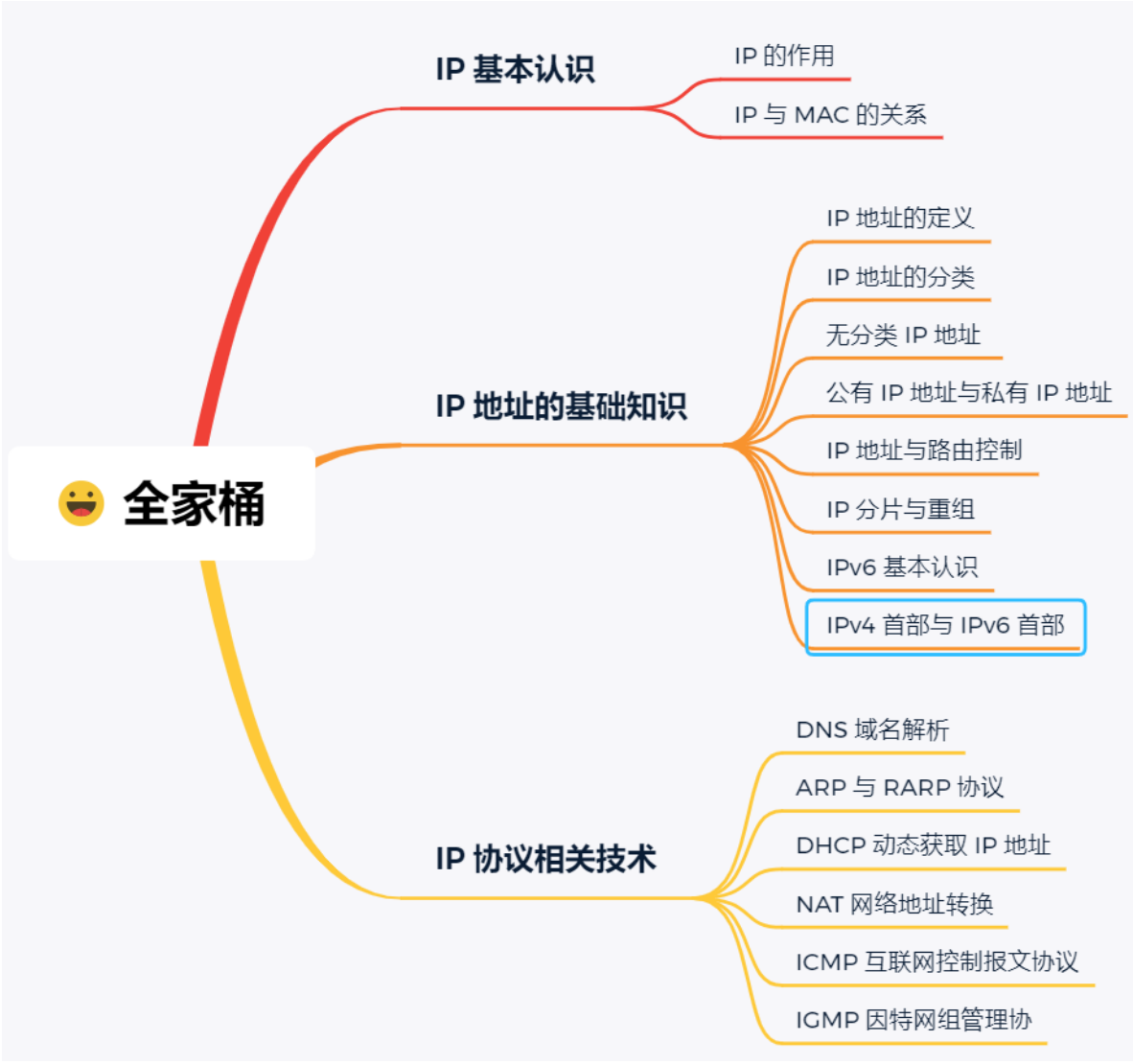
前段时间，有读者希望我写一篇关于 IP 分类地址、子网划分等的文章，他反馈常常混淆，摸不着头脑。

那么，说来就来！而且要盘就盘全一点，顺便挑战下小林的图解功力，所以就来个 **IP 基础知识全家桶**。

吃完这个 IP 基础知识全家桶，包你撑着肚子喊出：“**真香！**”

不多说，直接上菜，共分为**三道菜**：

- 首先是前菜 「 IP 基本认识 」
- 其次是主菜 「IP 地址的基础知识」
- 最后是点心 「IP 协议相关技术」



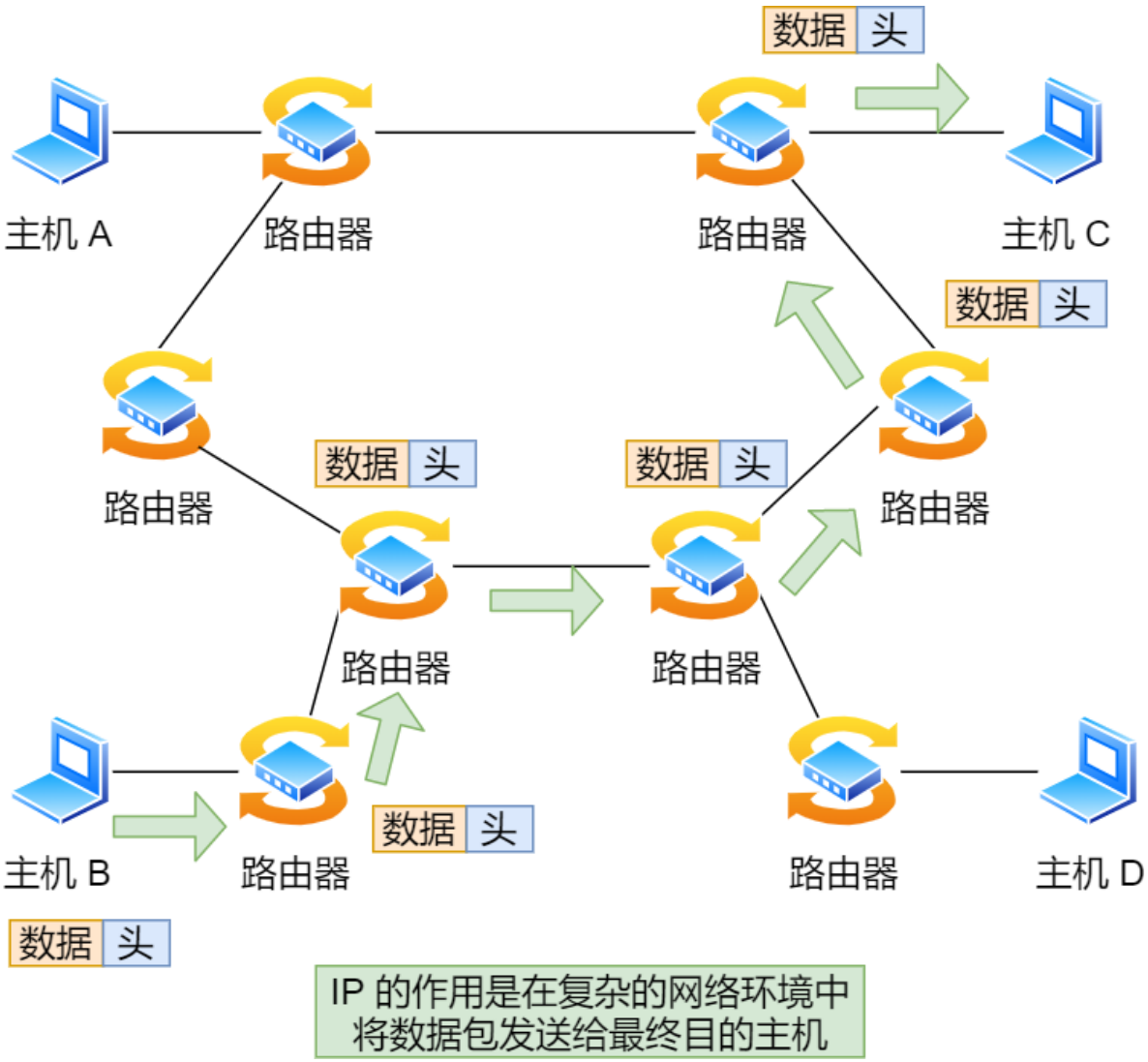
> 为啥要比喻成菜？因为小林是**菜狗**（押韵不？）

正文

前菜 —— IP 基本认识

IP 在 TCP/IP 参考模型中处于第三层，也就是**网络层**。

网络层的主要作用是：**实现主机与主机之间的通信，也叫点对点（end to end）通信。**



> 网络层与数据链路层有什么关系呢?

有的小伙伴分不清 IP（网络层） 和 MAC （数据链路层）之间的区别和关系。

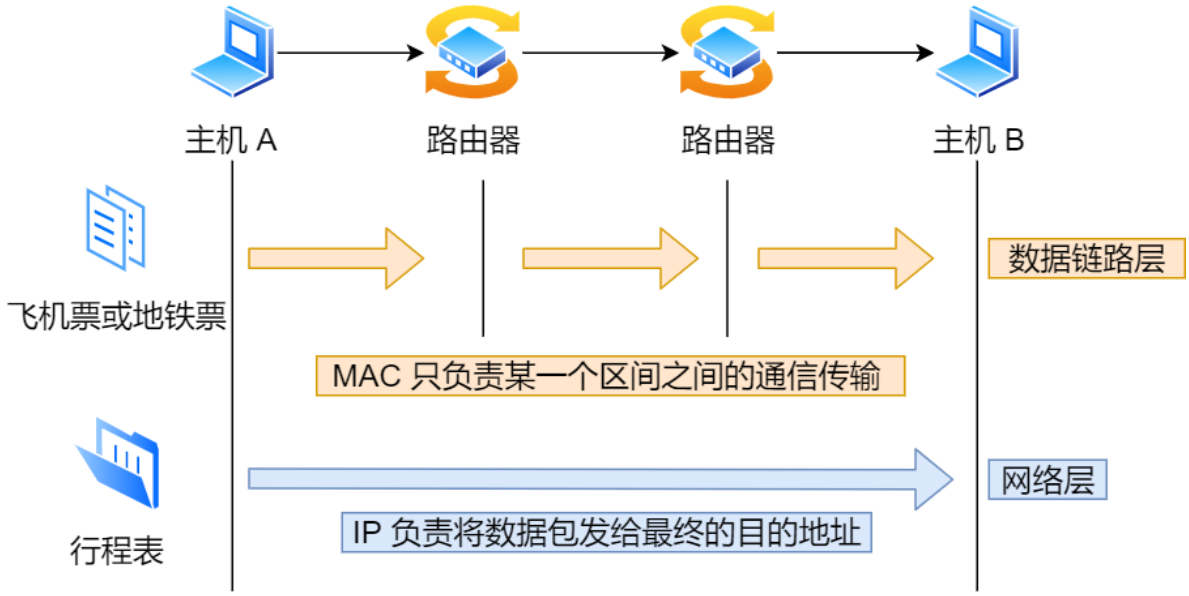
其实很容易区分，在上面我们知道 IP 的作用是主机之间通信用的，而 **MAC 的作用则是实现「直连」的两个设备之间通信，而 IP 则负责在「没有直连」的两个网络之间进行通信传输。**

举个生活的栗子，小林要去一个很远的地方旅行，制定了一个行程表，其间需先后乘坐飞机、地铁、公交车才能抵达目的地，为此小林需要买飞机票，地铁票等。

飞机票和地铁票都是去往特定的地点的，每张票只能够在某一限定区间内移动，此处的「区间内」就如同通信网络中数据链路。

在区间内移动相当于数据链路层，充当区间内两个节点传输的功能，区间内的出发点好比源 MAC 地址，目标地点好比目的 MAC 地址。

整个旅游行程表就相当于网络层，充当远程定位的功能，行程的开始好比源 IP，行程的终点好比目的 IP 地址。



如果小林只有行程表而没有车票，就无法搭乘交通工具到达目的地。相反，如果除了车票而没有行程表，恐怕也很难到达目的地。因为小林不知道该坐什么车，也不知道该在哪里换乘。

因此，只有两者兼备，既有某个区间的车票又有整个旅行的行程表，才能保证到达目的地。与此类似，**计算机网络中也需要「数据链路层」和「网络层」这个分层才能实现向最终目标地址的通信。**

还有重要一点，旅行途中我们虽然不断变化了交通工具，但是旅行行程的起始地址和目的地址始终都没变。其实，在网络中数据包传输中也是如此，**源IP地址和目标IP地址在传输过程中是不会变化的，只有源 MAC 地址和目标 MAC 一直在变化。**

主菜 —— IP 地址的基础知识

在 TCP/IP 网络通信时，为了保证能正常通信，每个设备都需要配置正确的 IP 地址，否则无法实现正常的通信。

IP 地址（IPv4 地址）由 `32` 位正整数来表示，IP 地址在计算机是以二进制的方式处理的。

而人类为了方便记忆采用了**点分十进制**的标记方式，也就是将 32 位 IP 地址以每 8 位为组，共分为 `4` 组，每组以「`.`」隔开，再将每组转换成十进制。

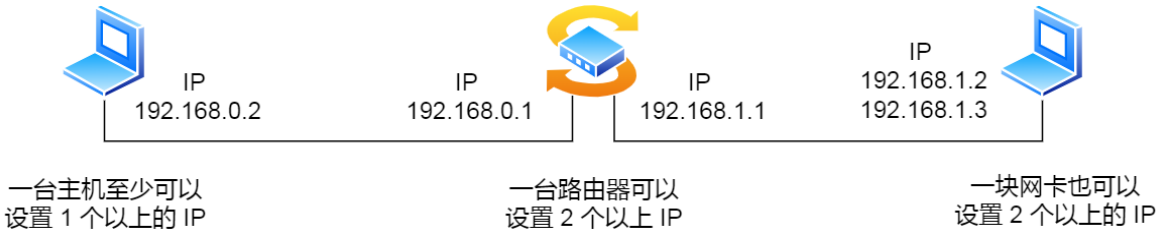
IPv4 二进制	11000000	10101000	00000001	00000001
IPv4 十进制	192	168	1	1
点分十进制	192 .	168 .	1 .	1

那么，IP 地址最大值也就是

$$2^{32} = 4294967296$$

也就说，最大允许 43 亿台计算机连接到网络。

实际上，IP 地址并不是根据主机台数来配置的，而是以网卡。像服务器、路由器等设备都是有 2 个以上的网卡，也就是它们会有 2 个以上的 IP 地址。



因此，让 43 亿台计算机全部连网其实是不可能的，更何况 IP 地址是由「网络标识」和「主机标识」这两个部分组成的，所以实际能够连接到网络的计算机个数更是少了很多。

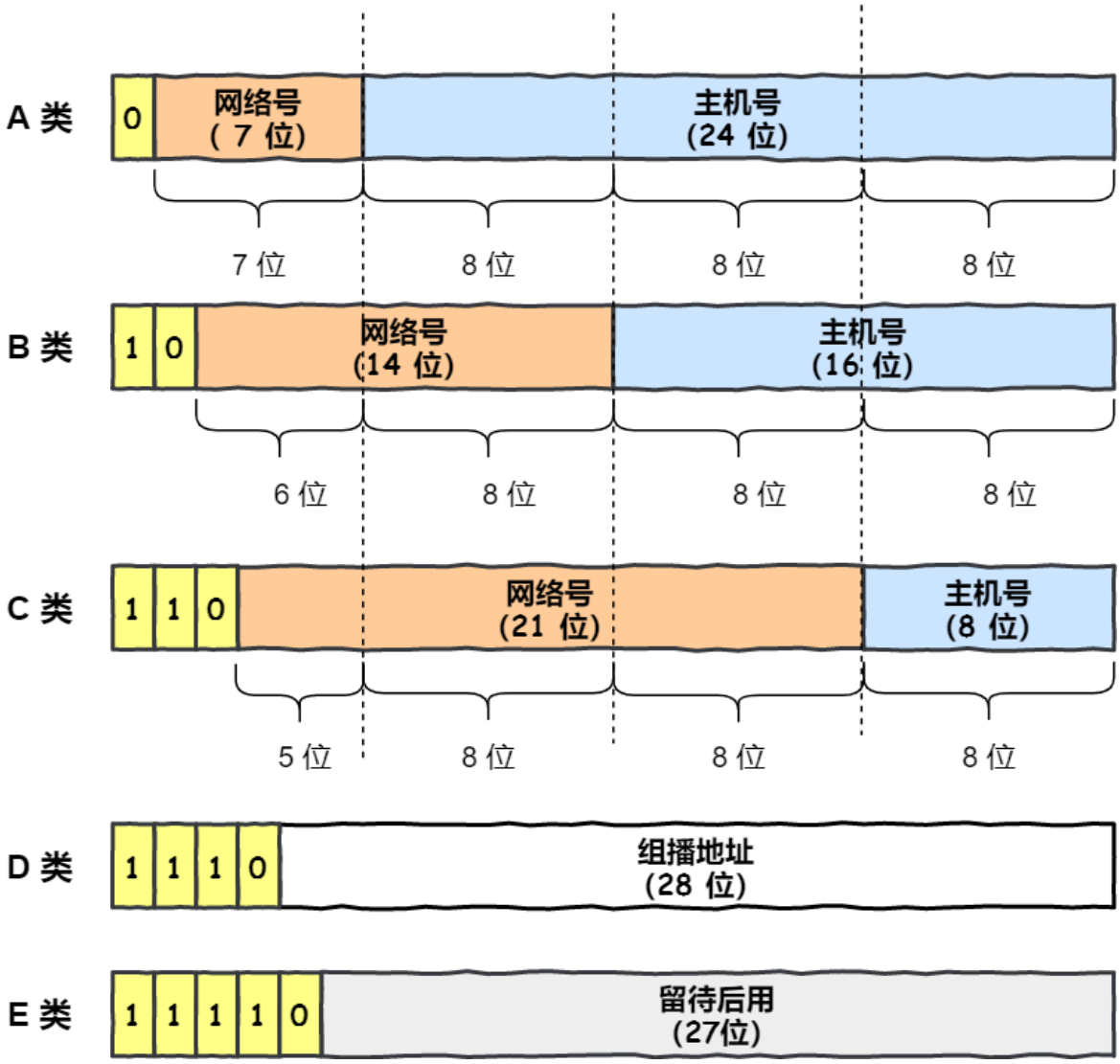
> 可能有的小伙伴提出了疑问，现在不仅电脑配了 IP， 手机、IPad 等电子设备都配了 IP 呀，照理来说肯定会超过 43 亿啦，那是怎么能够支持这么多 IP 的呢?

因为会根据一种可以更换 IP 地址的技术 NAT，使得可连接计算机数超过 43 亿台。 NAT 技术后续会进一步讨论和说明。

IP 地址的分类

互联网诞生之初，IP 地址显得很充裕，于是计算机科学家们设计了**分类地址**。

IP 地址分类成了 5 种类型，分别是 A 类、B 类、C 类、D 类、E 类。



上图中黄色部分为分类号，用以区分 IP 地址类别。

> 什么是 A、B、C 类地址?

其中对于 A、B、C 类主要分为两个部分，分别是**网络号和主机号**。这很好理解，好比小林是 A 小区 1 栋 101 号，你是 B 小区 1 栋 101 号。

我们可以用下面这个表格， 就能很清楚的知道 A、B、C 分类对应的地址范围、最大主机个数。

类别	IP 地址范围	最大主机数
A	0.0.0.0 ~ 127.255.255.255	16777214
B	128.0.0.0 ~ 191.255.255.255	65534
C	192.0.0.0 ~ 223.255.255.255	254

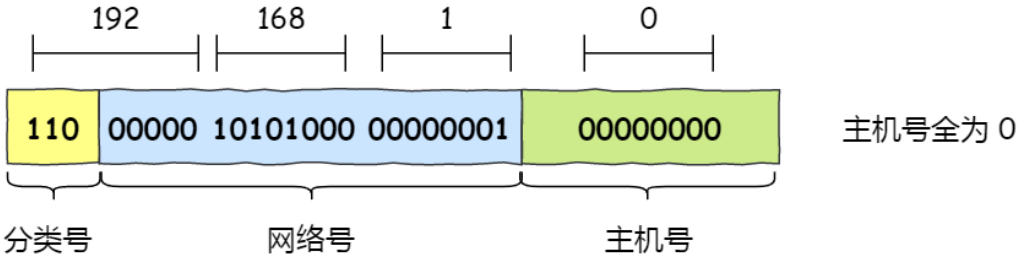
> A、B、C 分类地址最大主机个数是如何计算的呢?

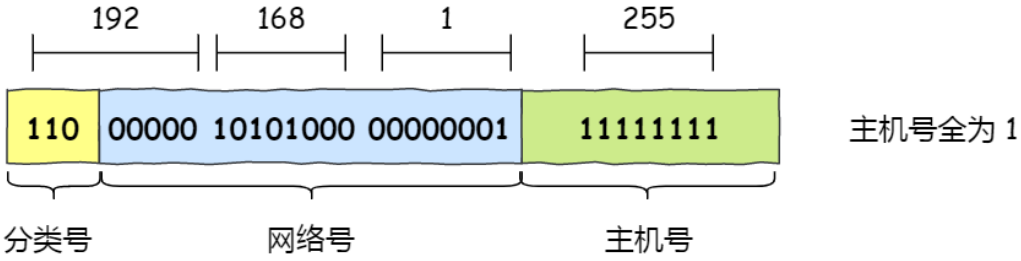
最大主机个数，就是要看主机号的位数，如 C 类地址的主机号占 8 位，那么 C 类地址的最大主机个数：

$$2^8 - 2 = 254$$

为什么要减 2 呢?

因为在 IP 地址中，有两个 IP 是特殊的，分别是主机号全为 1 和 全为 0 地址。





- 主机号全为 1 指定某个网络下的所有主机，用于广播
- 主机号全为 0 指定某个网络

因此，在分配过程中，应该去掉这两种情况。

> 广播地址用于什么?

广播地址用于在**同一个链路中相互连接的主机之间发送数据包**。

学校班级中就有广播的例子，在准备上课的时候，通常班长会喊：“上课， 全体起立！”，班里的同学听到这句话是不是全部都站起来了？这个句话就有广播的含义。

当主机号全为 1 时，就表示该网络的广播地址。例如把 `172.20.0.0/16` 用二进制表示如下：

10101100.00010100.00000000.00000000

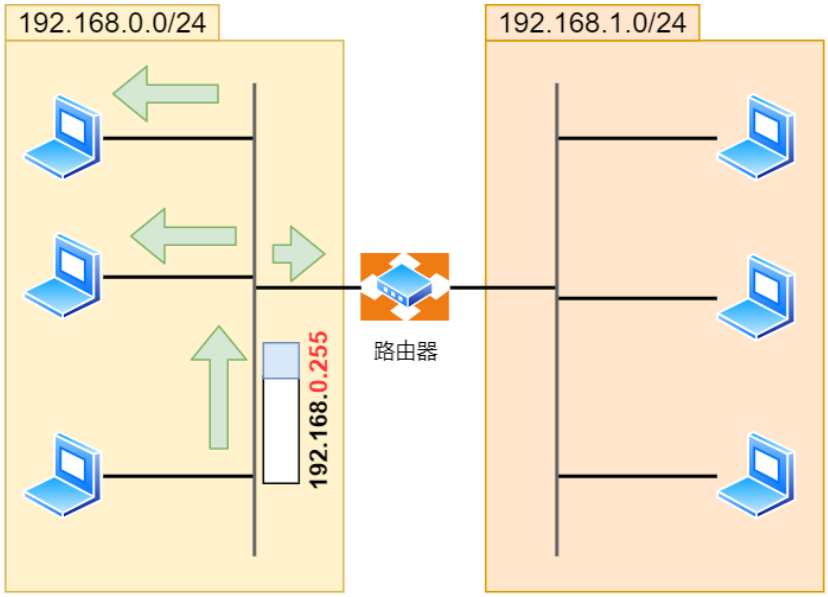
将这个地址的**主机部分全部改为 1**，则形成广播地址：

10101100.00010100. `11111111.11111111`

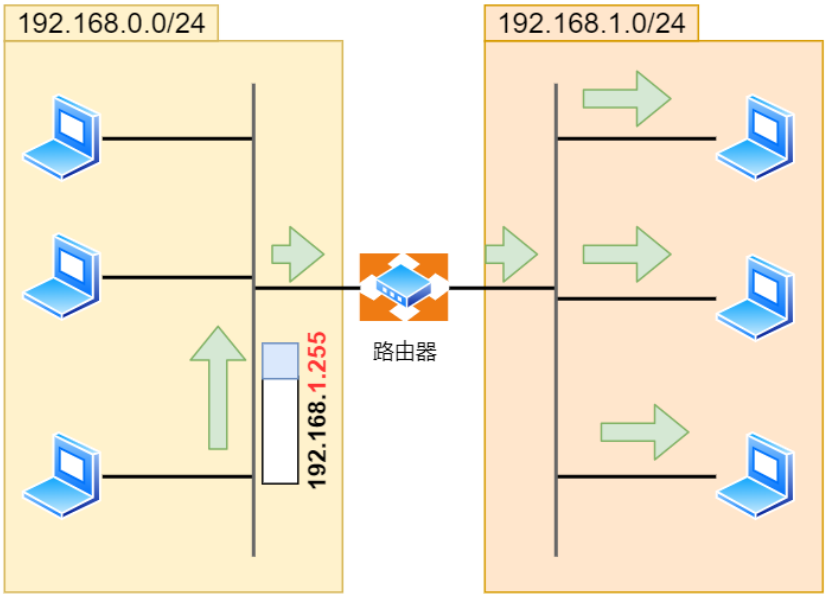
再将这个地址用十进制表示，则为 `172.20.255.255` 。

广播地址可以分为本地广播和直接广播两种。

- **在本网络内广播的叫做本地广播**。例如网络地址为 192.168.0.0/24 的情况下，广播地址是 192.168.0.255 。因为这个广播地址的 IP 包会被路由器屏蔽，所以不会到达 192.168.0.0/24 以外的其他链路上。
- **在不同网络之间的广播叫做直接广播**。例如网络地址为 192.168.0.0/24 的主机向 192.168.1.255/24 的目标地址发送 IP 包。收到这个包的路由器，将数据转发给 192.168.1.0/24，从而使得所有 192.168.1.1~192.168.1.254 的主机都能收到这个包（由于直接广播有一定的安全问题，多数情况下会在路由器上设置为不转发。） 。



本地广播
192.168.0.255 的数据包
不会到达 192.168.1.0/24 的网络



直接广播
192.168.1.255 是广播数据包

> 什么是 D、E 类地址?

而 D 类和 E 类地址是没有主机号的，所以不可用于主机 IP，D 类常被用于**多播**，E 类是预留的分类，暂时未使用。

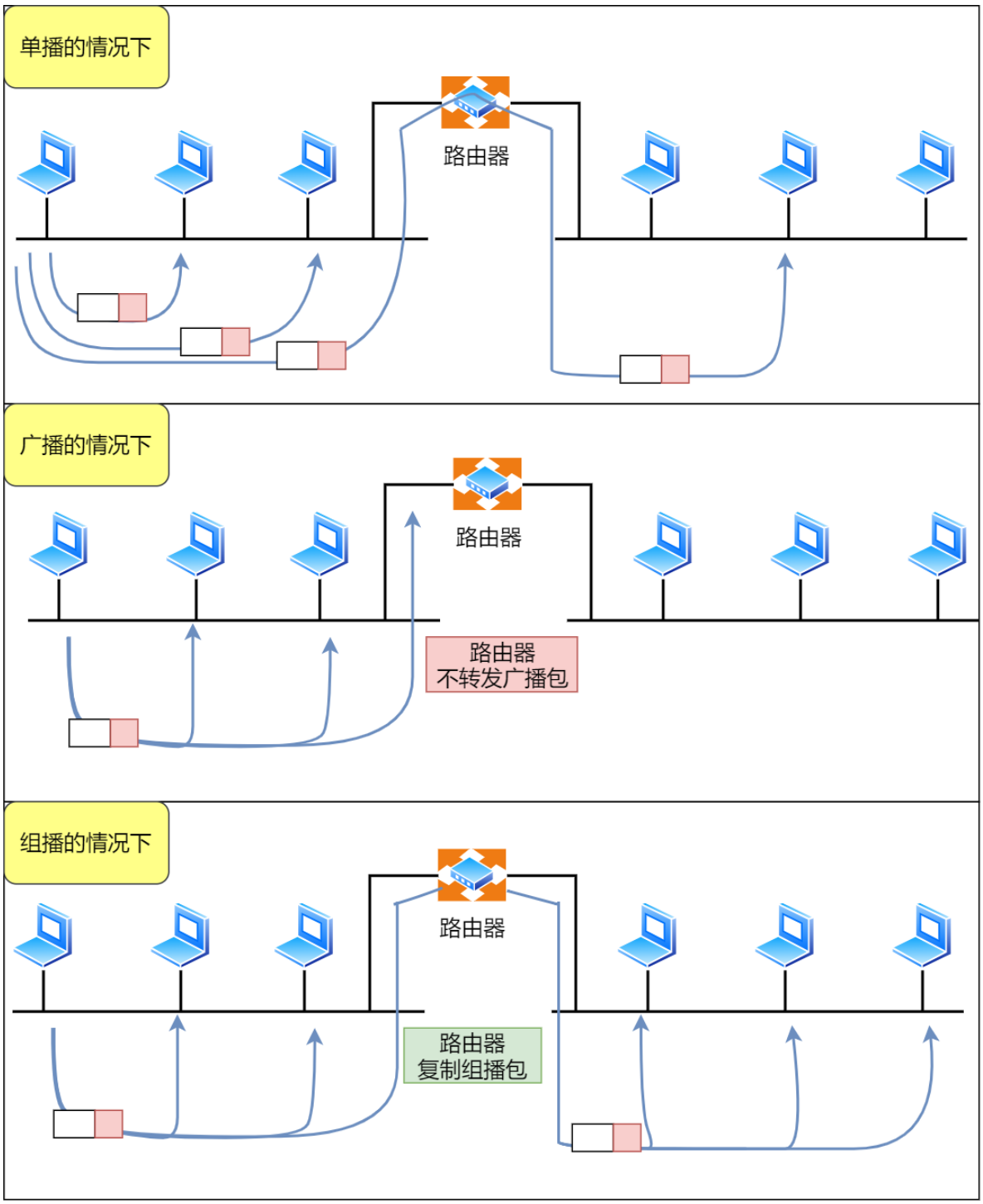
类别	IP 地址范围	用途
D	224.0.0.0 ~ 239.255.255.255	IP 多播
E	240.0.0.0 ~ 255.255.255.255	预留使用

> 多播地址用于什么?

多播用于**将包发送给特定组内的所有主机。**

还是举班级的栗子，老师说：“最后一排的同学，上来做这道数学题。”，老师指定的是最后一排的同学，也就是多播的含义了。

由于广播无法穿透路由，若想给其他网段发送同样的包，就可以使用可以穿透路由的多播。



多播使用的 D 类地址，其前四位是 `1110` 就表示是多播地址，而剩下的 28 位是多播的组编号。

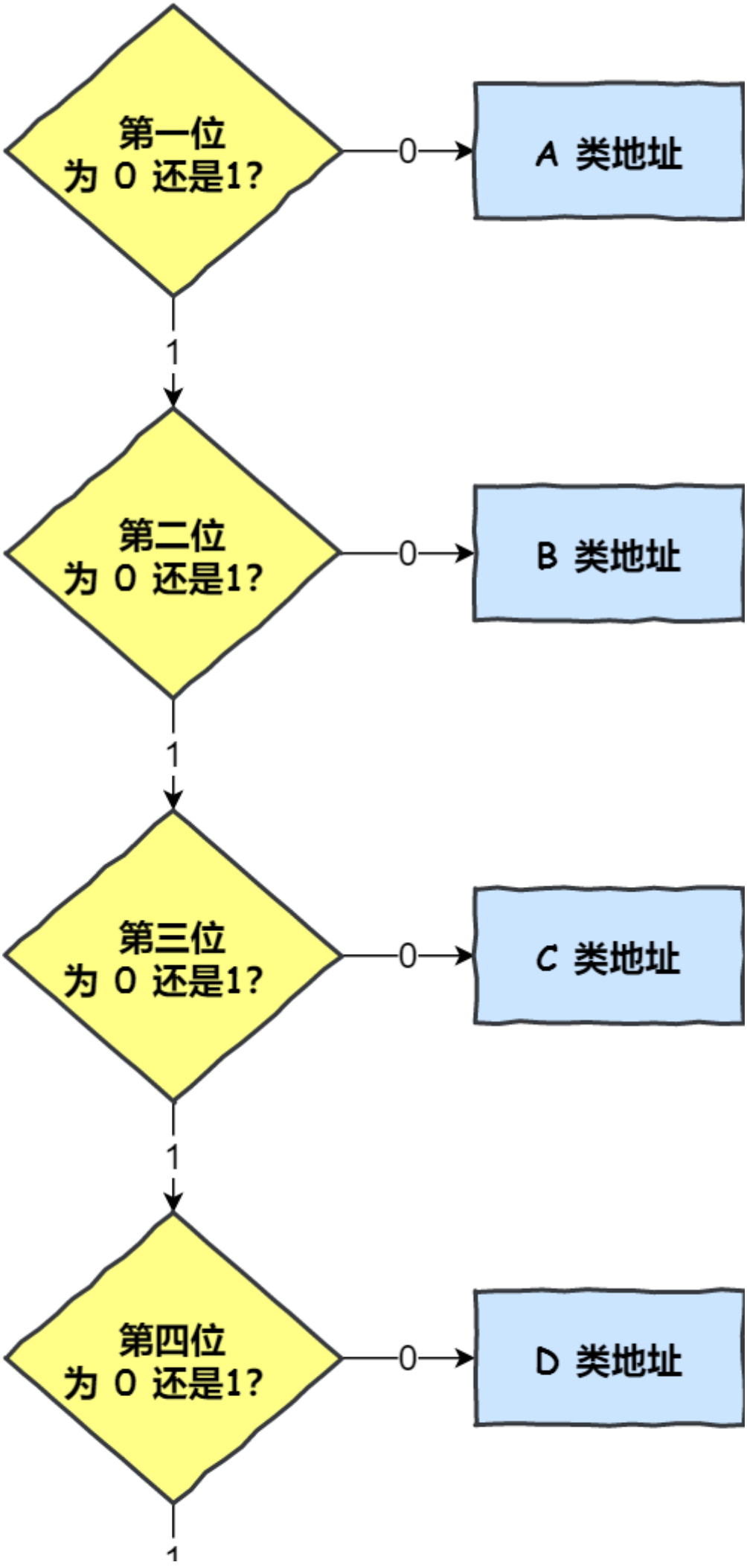
从 224.0.0.0 ~ 239.255.255.255 都是多播的可用范围，其划分为以下三类：

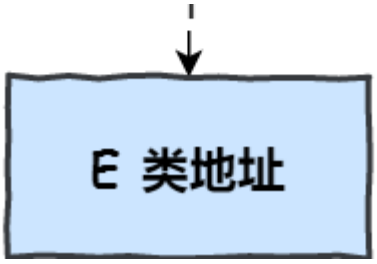
- 224.0.0.0 ~ 224.0.0.255 为预留的组播地址，只能在局域网中，路由器是不会进行转发的。
- 224.0.1.0 ~ 238.255.255.255 为用户可用的组播地址，可以用于 Internet 上。
- 239.0.0.0 ~ 239.255.255.255 为本地管理组播地址，可供内部网在内部使用，仅在特定的本地范围内有效。

> IP 分类的优点

不管是路由器还是主机解析到一个 IP 地址时候，我们判断其 IP 地址的首位是否为 0，为 0 则为 A 类地址，那么就能很快的找出网络地址和主机地址。

其余分类判断方式参考如下图：


第一位
为 0 还是1?
0
A 类地址
1
第二位
为 0 还是1?
0
B 类地址
1
第三位
为 0 还是1?
0
C 类地址
1
第四位
为 0 还是1?
0
D 类地址
1



所以，这种分类地址的优点就是**简单明了、选路（基于网络地址）简单**。

> IP 分类的缺点

缺点一

同一网络下没有地址层次，比如一个公司里用了 B 类地址，但是可能需要根据生产环境、测试环境、开发环境来划分地址层次，而这种 IP 分类是没有地址层次划分的功能，所以这就**缺少地址的灵活性**。

缺点二

A、B、C类有个尴尬处境，就是**不能很好的与现实网络匹配**。

- C 类地址能包含的最大主机数量实在太少了，只有 254 个，估计一个网吧都不够用。
- 而 B 类地址能包含的最大主机数量又太多了，6 万多台机器放在一个网络下面，一般的企业基本达不到这个规模，闲着的地址就是浪费。

这两个缺点，都可以在 `CIDR` 无分类地址解决。

无分类地址 CIDR

正因为 IP 分类存在许多缺点，所以后面提出了无分类地址的方案，即 `CIDR` 。

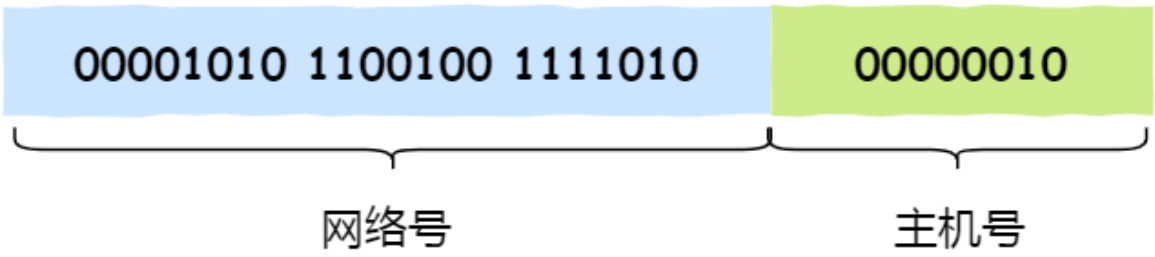
这种方式不再有分类地址的概念，32 比特的 IP 地址被划分为两部分，前面是**网络号**，后面是**主机号**。

> 怎么划分网络号和主机号的呢?

表示形式 `a.b.c.d/x` ，其中 `/x` 表示前 x 位属于**网络号**， x 的范围是 `0 ~ 32` ，这就使得 IP 地址更加具有灵活性。

比如 10.100.122.2/24，这种地址表示形式就是 CIDR，/24 表示前 24 位是网络号，剩余的 8 位是主机号。

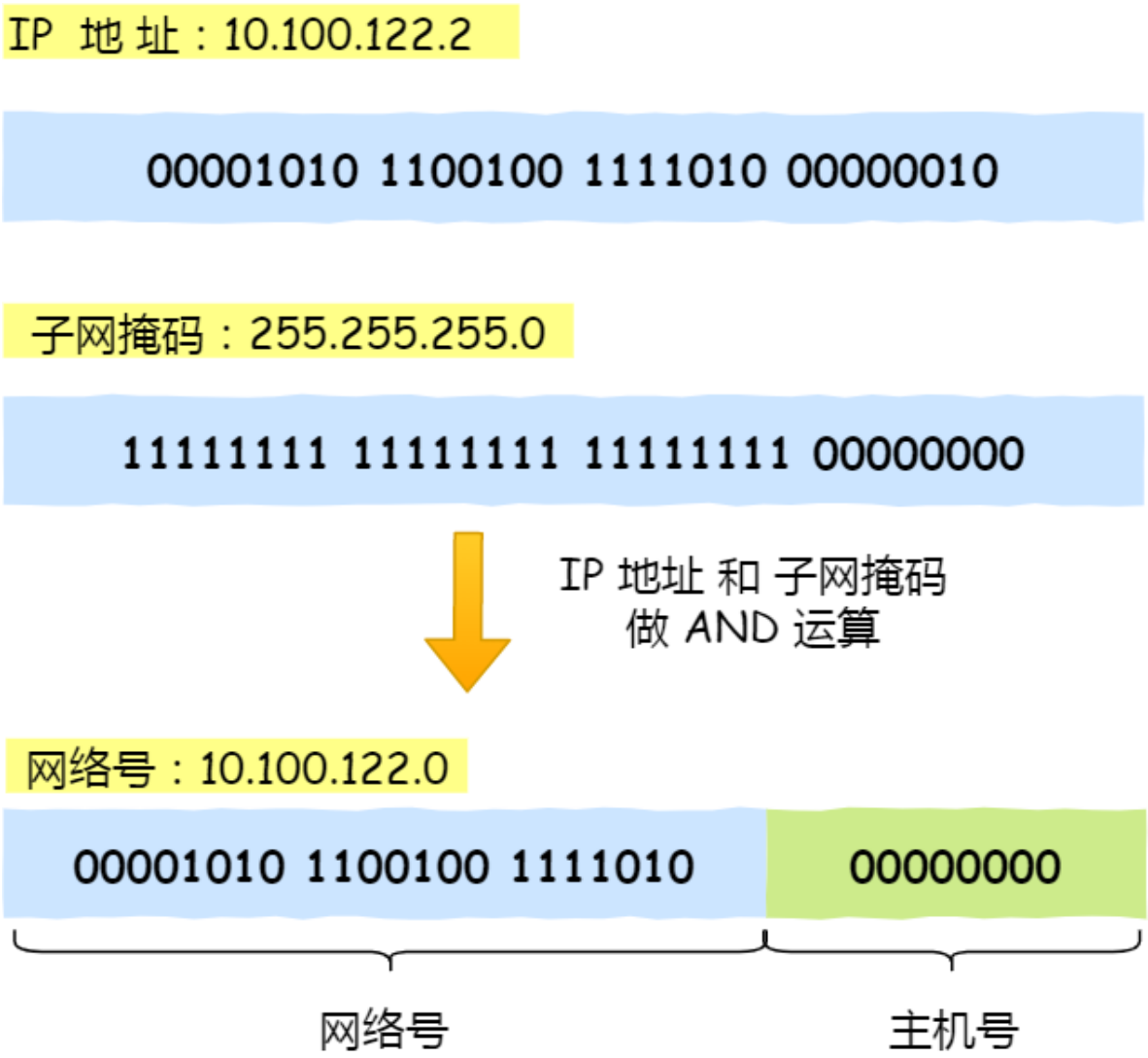
10.100.122.2/24



可用地址个数	254
子网掩码	255.255.255.0
网络号	10.100.122.0
第一个可用地址	10.100.122.1
最后可用地址	10.100.122.254
广播地址	10.100.122.255

还有另一种划分网络号与主机号形式，那就是**子网掩码**，掩码的意思就是掩盖掉主机号，剩余的就是网络号。

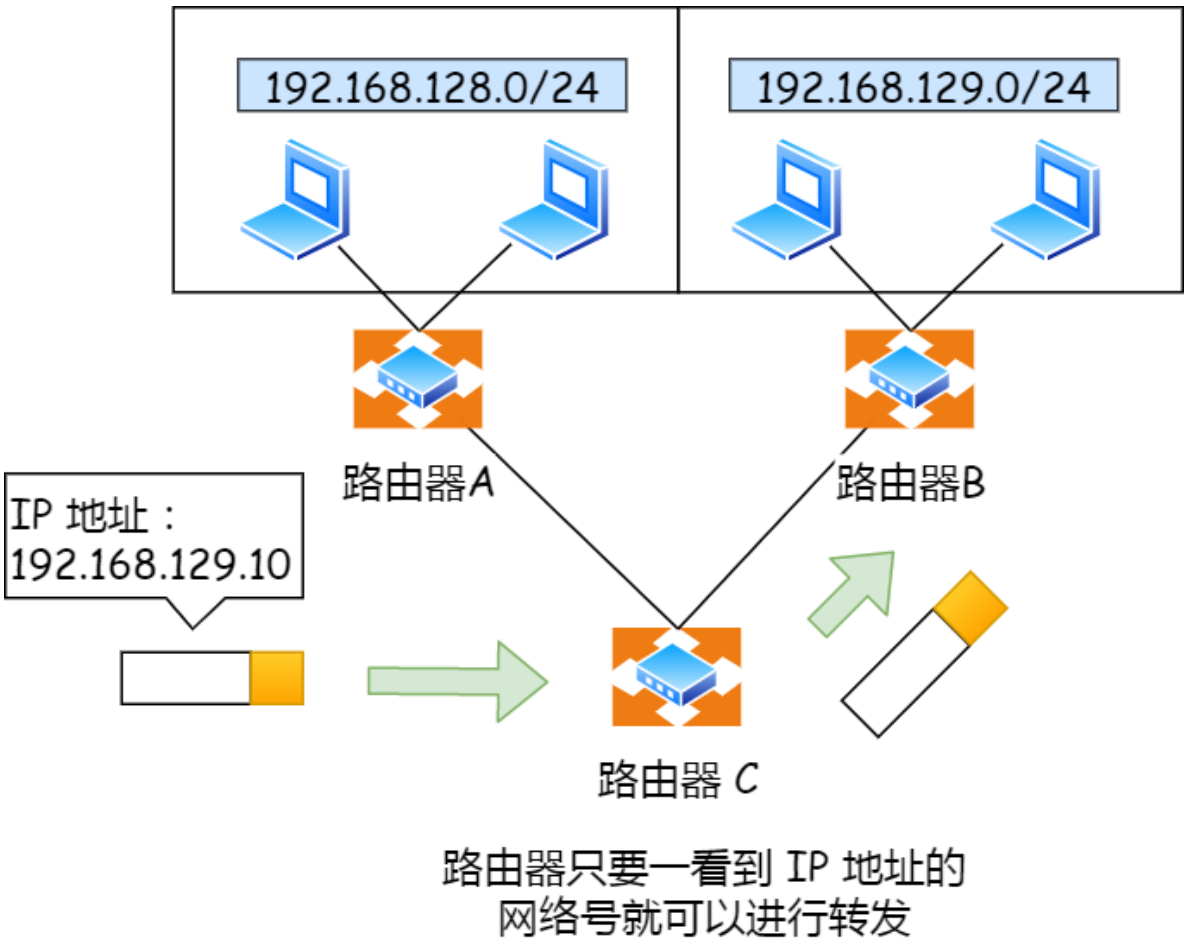
将子网掩码和 IP 地址按位计算 AND，就可得到网络号。



> 为什么要分离网络号和主机号?

因为两台计算机要通讯，首先要判断是否处于同一个广播域内，即网络地址是否相同。如果网络地址相同，表明接受方在本网络上，那么可以把数据包直接发送到目标主机。

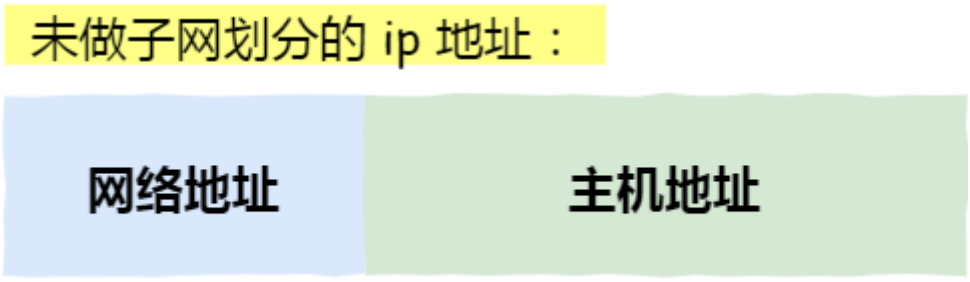
路由器寻址工作中，也就是通过这样的方式来找到对应的网络号的，进而把数据包转发给对应的网络内。

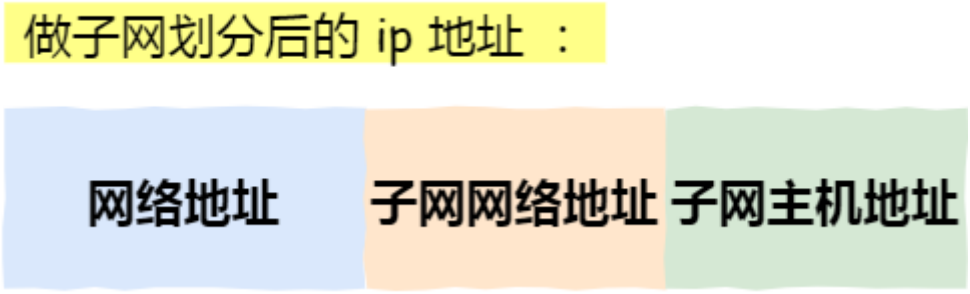


> 怎么进行子网划分?

在上面我们知道可以通过子网掩码划分出网络号和主机号，那实际上子网掩码还有一个作用，那就是**划分子网**。

子网划分实际上是将主机地址分为两个部分：子网网络地址和子网主机地址。形式如下：



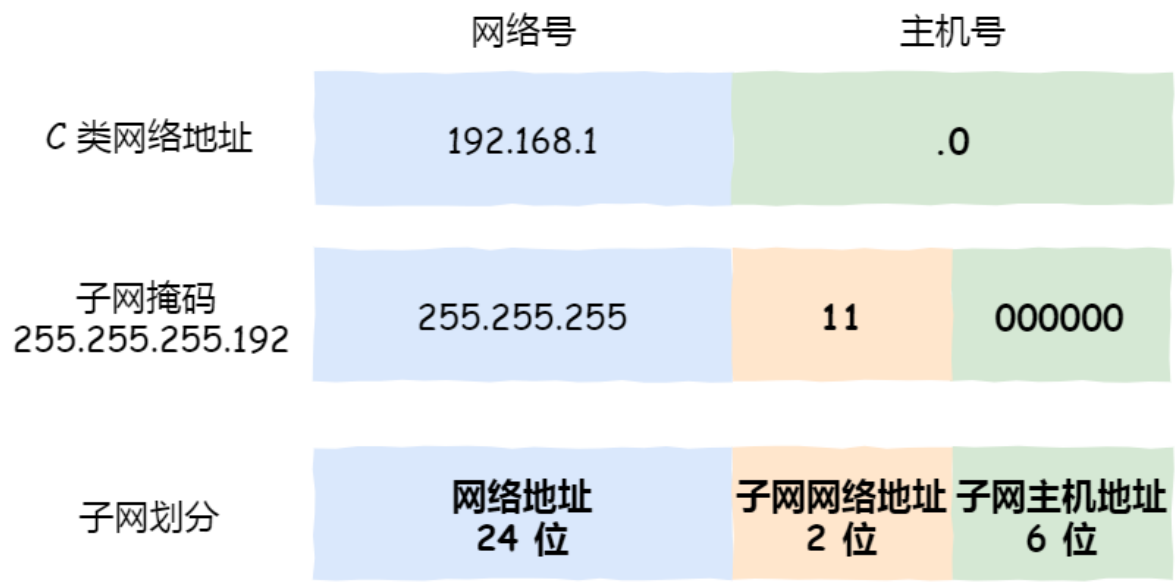


- 未做子网划分的 ip 地址：网络地址 + 主机地址

- 做子网划分后的 ip 地址：网络地址 + （子网网络地址 + 子网主机地址）

假设对 C 类地址进行子网划分，网络地址 192.168.1.0，使用子网掩码 255.255.255.192 对其进行子网划分。

C 类地址中前 24 位是网络号，最后 8 位是主机号，根据子网掩码可知**从 8 位主机号中借用 2 位作为子网号**。



由于子网网络地址被划分成 2 位，那么子网地址就有 4 个，分别是 00、01、10、11，具体划分如下图：

子网 0

192.168.1	00	000000	子网 0 网络地址 192.168.1.0
192.168.1	00	000001	可分配的最小地址 192.168.1.1
	.		
	.		
	.		
192.168.1	00	111110	可分配的最大地址 192.168.1.62
192.168.1	00	111111	子网 0 广播地址 192.168.1.63

子网 1

192.168.1	01	000000	子网 1 网络地址 192.168.1.64
192.168.1	01	000001	可分配的最小地址 192.168.1.65
	.		
	.		
	.		
192.168.1	01	111110	可分配的最大地址 192.168.1.126
192.168.1	01	111111	子网 1 广播地址 192.168.1.127

子网 2

192.168.1	10	000000	子网 2 网络地址 192.168.1.128
192.168.1	10	000001	可分配的最小地址 192.168.1.129
	.		
	.		
	.		
192.168.1	10	111110	可分配的最大地址 192.168.1.190
192.168.1	10	111111	子网 2 广播地址 192.168.1.191

子网 3

192.168.1	11	000000	子网 3 网络地址 192.168.1.192
192.168.1	11	000001	可分配的最小地址 192.168.1.193
.			
.			
.			
192.168.1	11	111110	可分配的最大地址 192.168.1.254
192.168.1	11	111111	子网 3 广播地址 192.168.1.255

划分后的 4 个子网如下表格：

子网号	网络地址	主机地址范围	广播地址
0	192.168.1.0	192.168.1.1 ~ 192.168.1.62	192.168.1.63
1	192.168.1.64	192.168.1.65 ~ 192.168.1.126	192.168.1.127
2	192.168.1.128	192.168.1.129 ~ 192.168.1.190	192.168.1.191
3	192.168.1.192	192.168.1.193 ~ 192.168.1.254	192.168.1.255

公有 IP 地址与私有 IP 地址

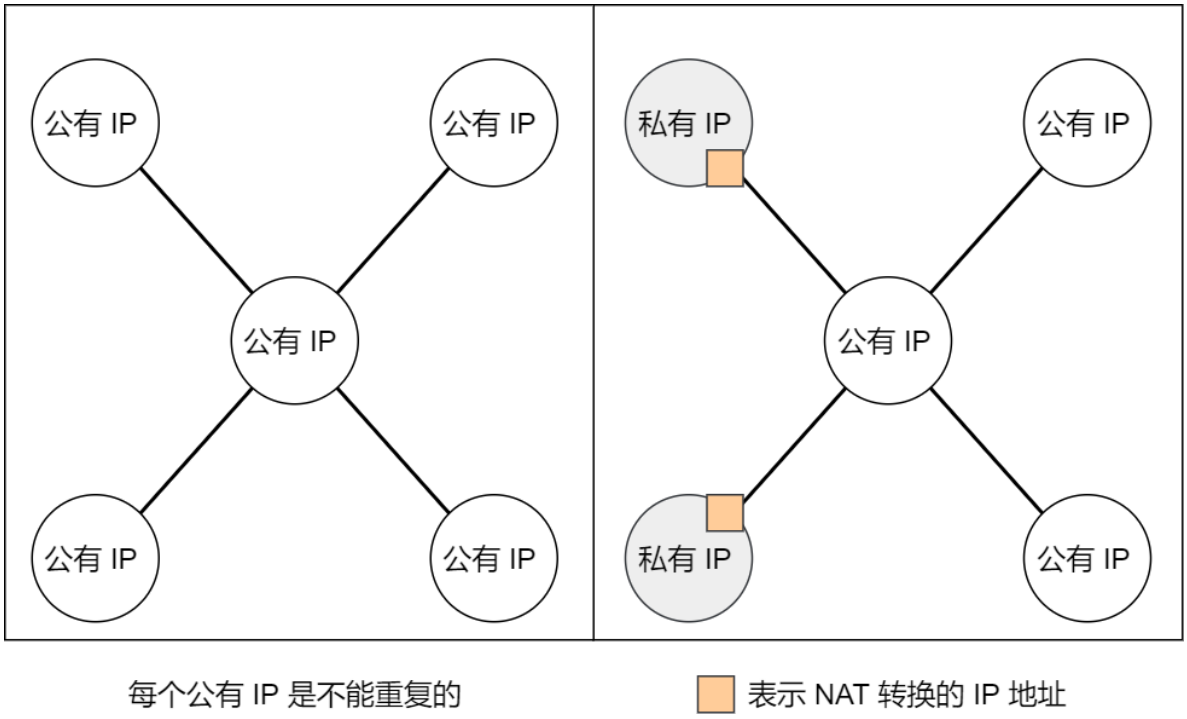
在 A、B、C 分类地址，实际上有分公有 IP 地址和私有 IP 地址。

类别	IP 地址范围	最大主机数	私有 IP 地址范围
A	0.0.0.0 ~ 127.255.255.255	16777214	10.0.0.0 ~ 10.255.255.255
B	128.0.0.0 ~ 191.255.255.255	65534	172.16.0.0 ~ 172.31.255.255
C	192.0.0.0 ~ 223.255.255.255	254	192.168.0.0 ~ 192.168.255.255

平时我们办公室、家里、学校用的 IP 地址，一般都是私有 IP 地址。因为这些地址允许组织内部的 IT 人员自己管理、自己分配，而且可以重复。因此，你学校的某个私有 IP 地址和我学校的可以是一样的。

就像每个小区都有自己的楼编号和门牌号，你小区家可以叫 1 栋 101 号，我小区家也可以叫 1 栋 101，没有任何问题。但一旦出了小区，就需要带上中山路 666 号（公网 IP 地址），是国家统一分配的，不能两个小区都叫中山路 666。

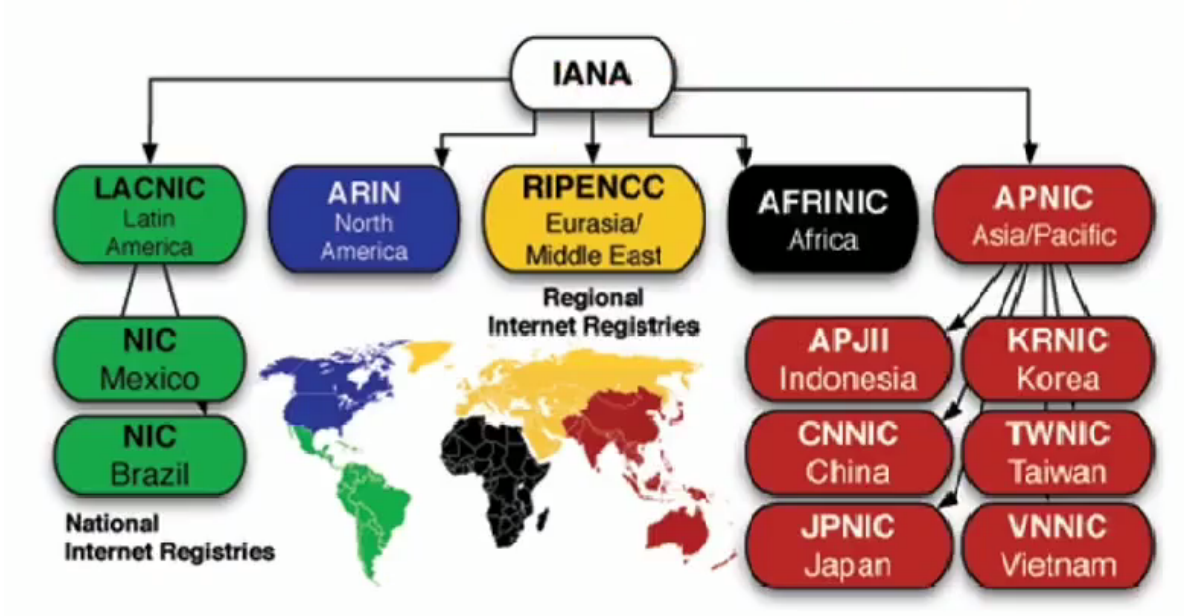
所以，公有 IP 地址是有个组织统一分配的，假设你要开一个博客网站，那么你就需要去申请购买一个公有 IP，这样全世界的人才能访问。并且公有 IP 地址基本上要在整个互联网范围内保持唯一。



> 公有 IP 地址由谁管理呢?

私有 IP 地址通常是内部的 IT 人员管理，公有 IP 地址是由 ICANN 组织管理，中文叫「互联网名称与数字地址分配机构」。

IANA 是 ICANN 的其中一个机构，它负责分配互联网 IP 地址，是按州的方式层层分配。



- ARIN 北美地区

- LACNIC 拉丁美洲和一些加勒比群岛
- RIPE NCC 欧洲、中东和中亚
- AfriNIC 非洲地区
- APNIC 亚太地区

其中，在中国是由 CNNIC 的机构进行管理，它是中国国内唯一指定的全局 IP 地址管理的组织。

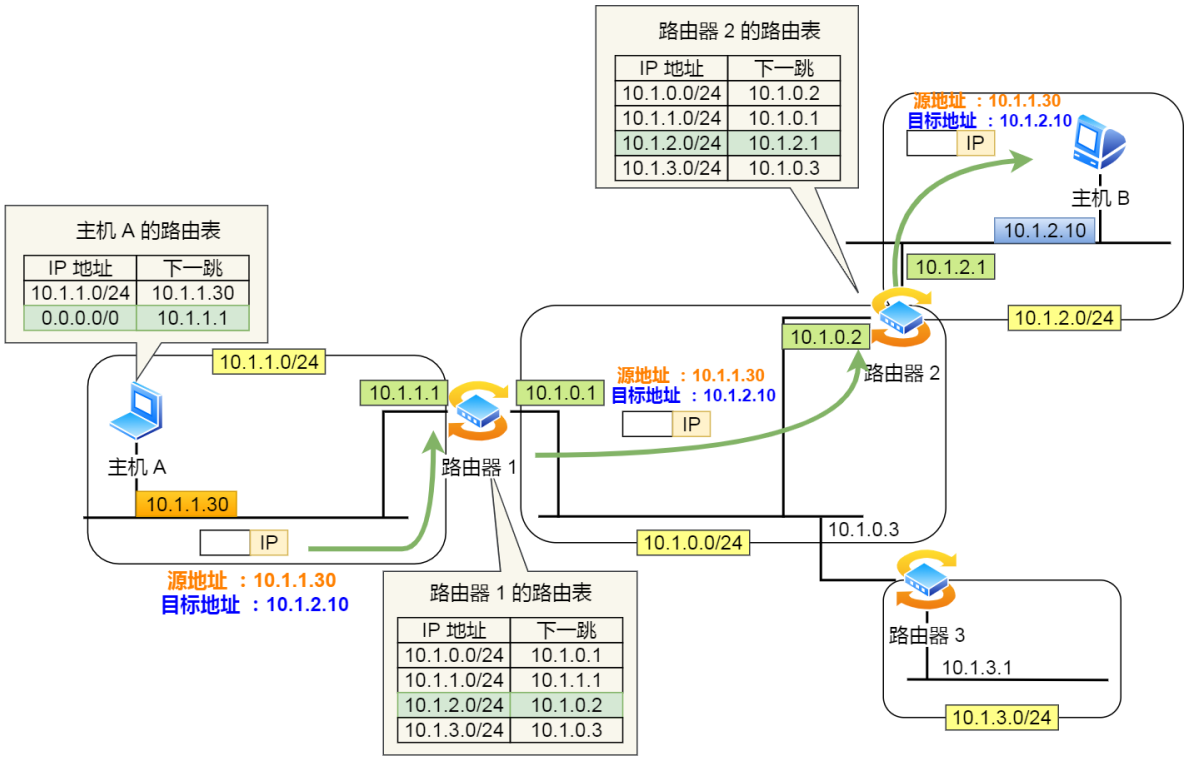
IP 地址与路由控制

IP地址的**网络地址**这一部分是用于进行路由控制。

路由控制表中记录着网络地址与下一步应该发送至路由器的地址。在主机和路由器上都会有各自的路由器控制表。

在发送 IP 包时，首先要确定 IP 包首部中的目标地址，再从路由控制表中找到与该地址具有**相同网络地址**的记录，根据该记录将 IP 包转发给相应的下一个路由器。如果路由控制表中存在多条相同网络地址的记录，就选择相同位数最多的网络地址，也就是最长匹配。

下面以下图的网络链路作为例子说明：



1. 主机 A 要发送一个 IP 包，其源地址是 `10.1.1.30` 和目标地址是 `10.1.2.10`，由于没有在主机 A 的路由表找到与目标地址 `10.1.2.10` 的网络地址，于是包被转发到默认路由（路由器 `1`）
2. 路由器 `1` 收到 IP 包后，也在路由器 `1` 的路由表匹配与目标地址相同的网络地址记录，发现匹配到了，于是就把 IP 数据包转发到了 `10.1.0.2` 这台路由器 `2`
3. 路由器 `2` 收到后，同样对比自身的路由表，发现匹配到了，于是把 IP 包从路由器 `2` 的 `10.1.2.1` 这个接口出去，最终经过交换机把 IP 数据包转发到了目标主机

> 环回地址是不会流向网络

环回地址是在同一台计算机上的程序之间进行网络通信时所使用的一个默认地址。

计算机使用一个特殊的 IP 地址 **127.0.0.1 作为环回地址**。与该地址具有相同意义的是一个叫做 `localhost` 的主机名。使用这个 IP 或主机名时，数据包不会流向网络。

IP 分片与重组

每种数据链路的最大传输单元 `MTU` 都是不相同的，如 FDDI 数据链路 MTU 4352、以太网的 MTU 是 1500 字节等。

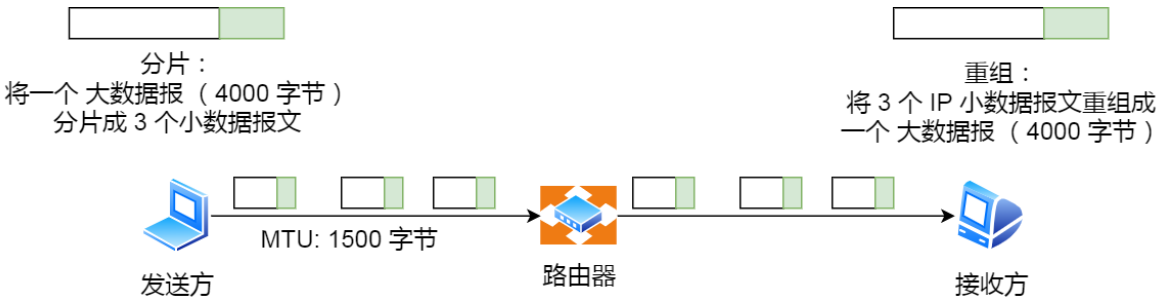
每种数据链路的 MTU 之所以不同，是因为每个不同类型的数据链路的使用目的不同。使用目的不同，可承载的 MTU 也就不同。

其中，我们最常见数据链路是以太网，它的 MTU 是 `1500` 字节。

那么当 IP 数据包大小大于 MTU 时， IP 数据包就会被分片。

经过分片之后的 IP 数据报在被重组的时候，只能由目标主机进行，路由器是不会进行重组的。

假设发送方发送一个 4000 字节的大数据报，若要传输在以太网链路，则需要把数据报分片成 3 个小数据报进行传输，再交由接收方重组成大数据报。



在分片传输中，一旦某个分片丢失，则会造成整个 IP 数据报作废，所以 TCP 引入了 `MSS` 也就是在 TCP 层进行分片不由 IP 层分片，那么对于 UDP 我们尽量不要发送一个大于 `MTU` 的数据报文。

IPv6 基本认识

IPv4 的地址是 32 位的，大约可以提供 42 亿个地址，但是早在 2011 年 IPv4 地址就已经被分配完了。

但是 IPv6 的地址是 `128` 位的，这可分配的地址数量是大的惊人，说个段子 **IPv6 可以保证地球上的每粒沙子都能被分配到一个 IP 地址。**

但 IPv6 除了有更多的地址之外，还有更好的安全性和扩展性，说简单点就是 IPv6 相比于 IPv4 能带来更好的网络体验。

但是因为 IPv4 和 IPv6 不能相互兼容，所以不但要我们电脑、手机之类的设备支持，还需要网络运营商对现有的设备进行升级，所以这可能是 IPv6 普及率比较慢的一个原因。

> IPv6 的亮点

IPv6 不仅仅只是可分配的地址变多了，它还有非常多的亮点。

- IPv6 可自动配置，即使没有 DHCP 服务器也可以实现自动分配IP地址，真是**便捷到即插即用**啊。

- IPv6 包头包首部长度采用固定的值 `40` 字节，去掉了包头校验和，简化了首部结构，减轻了路由器负荷，大大**提高了传输的性能**。
- IPv6 有应对伪造 IP 地址的网络安全功能以及防止线路窃听的功能，大大**提升了安全性**。
- **...** （由你发现更多的亮点）

> IPv6 地址的标识方法

IPv4 地址长度共 32 位，是以每 8 位作为一组，并用点分十进制的表示方式。

IPv6 地址长度是 128 位，是以每 16 位作为一组，每组用冒号 「:」 隔开。

IPv6 用二进制数表示

```
1111111011011100：1011101010011000：0111011001010100：0011001000010000：1111111011011100：1011101010011000：0111011001010100：0011001000010000
```

IPv6 十六进制数表示

```
FEDC：BA98：7654：3210：FEDC：BA98：7654：3210
```

如果出现连续的 0 时还可以将这些 0 省略，并用两个冒号 「::」隔开。但是，一个 IP 地址中只允许出现一次两个连续的冒号。

IPv6 用二进制数表示

```
1111111011011100：1011101010011000：0111011001010100：0000000000000000：0000000000000000：0000000000000000：0000000000000000：0011001000010000
```

IPv6 十六进制数表示

```
FEDC:BA98:7654::3210
```

> IPv6 地址的结构

IPv6 类似 IPv4，也是通过 IP 地址的前几位标识 IP 地址的种类。

IPv6 的地址主要有以下类型地址：

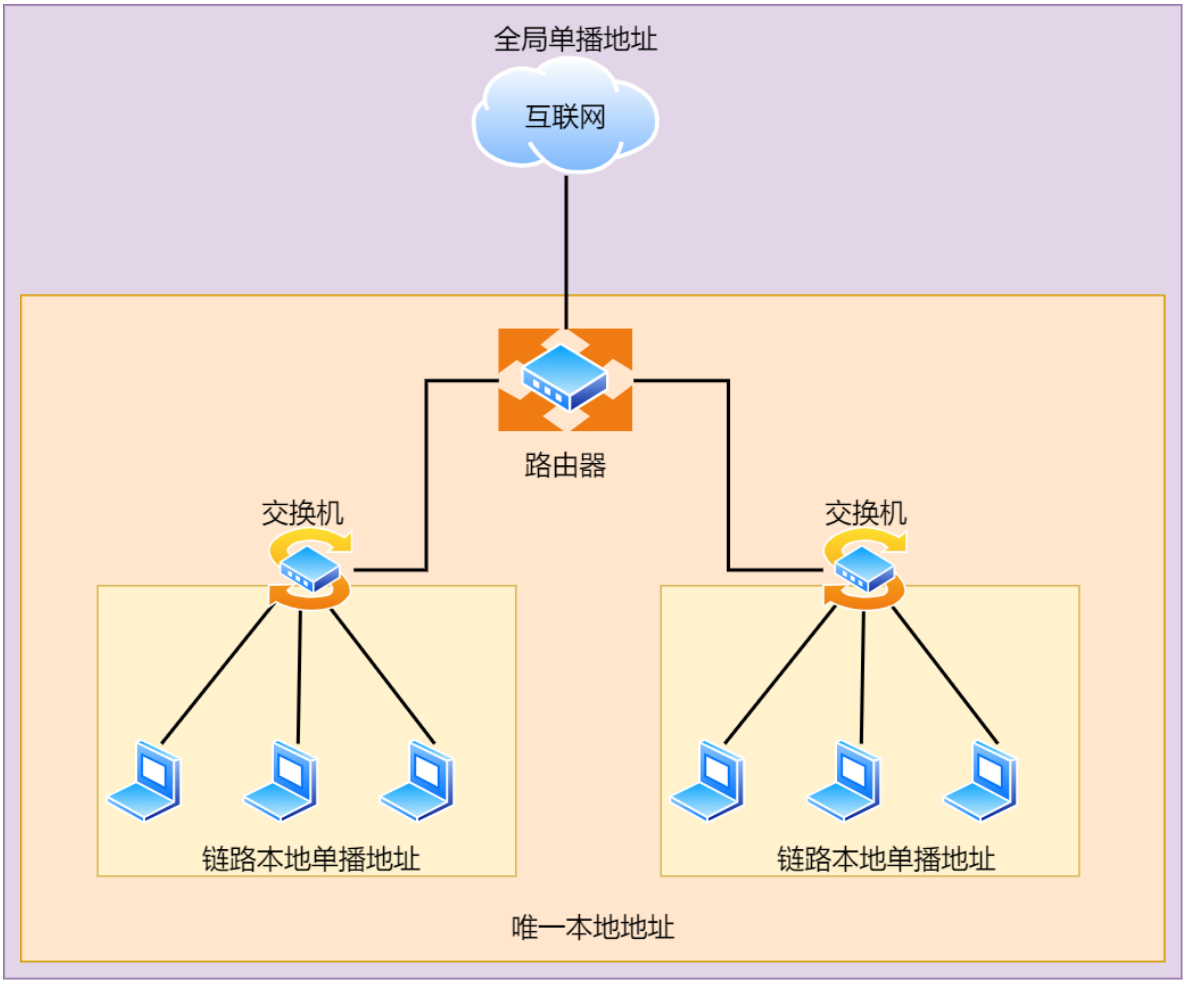
- 单播地址，用于一对一的通信
- 组播地址，用于一对多的通信
- 任播地址，用于通信最近的节点，最近的节点是由路由协议决定
- 没有广播地址

未定义	0000 ... 0000（128 位）	::/128
环回地址	0000 ... 0001（128 位）	::1/128
唯一本地地址	1111 110 ...	FC00::/7
链路本地单播地址	1111 1110 10 ...	FE80::/10
多播地址	1111 1111	FF00::/8
全局单播地址	其他	

> IPv6 单播地址类型

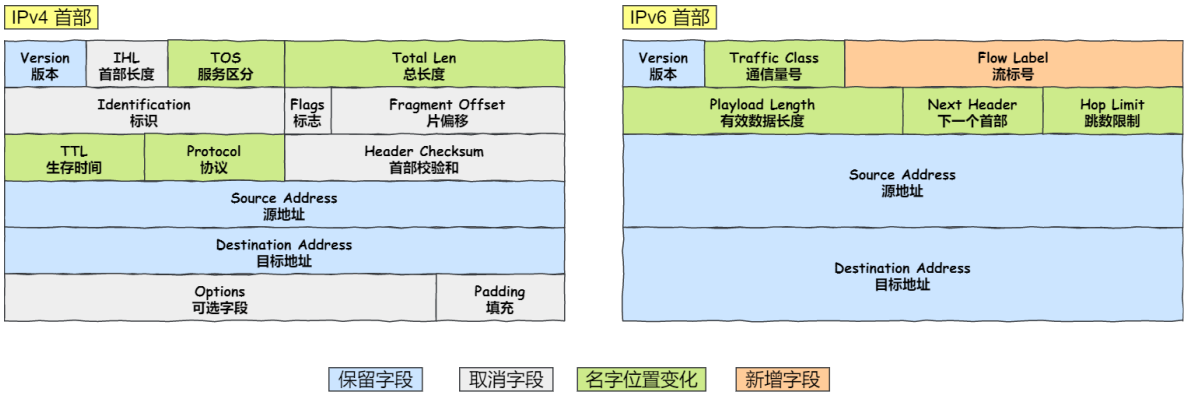
对于一对一通信的 IPv6 地址，主要划分了三类单播地址，每类地址的有效范围都不同。

- 在同一链路单播通信，不经过路由器，可以使用**链路本地单播地址**，IPv4 没有此类型
- 在内网里单播通信，可以使用**唯一本地地址**，相当于 IPv4 的私有 IP
- 在互联网通信，可以使用**全局单播地址**，相当于 IPv4 的公有 IP



IPv4 首部与 IPv6 首部

IPv4 首部与 IPv6 首部的差异如下图：



IPv6 相比 IPv4 的首部改进：

- **取消了首部校验和字段。** 因为在数据链路层和传输层都会校验，因此 IPv6 直接取消了 IP 的校验。
- **取消了分片/重新组装相关字段。** 分片与重组是耗时的过程，IPv6 不允许在中间路由器进行分片与重组，这种操作只能在源与目标主机，这将大大提高了路由器转发的速度。
- **取消选项字段。** 选项字段不再是标准 IP 首部的一部分了，但它并没有消失，而是可能出现在 IPv6 首部中的「下一个首部」指出的位置上。删除该选项字段使的 IPv6 的首部成为固定长度的 `40` 字节。

点心 —— IP 协议相关技术

跟 IP 协议相关的技术也不少，接下来说说与 IP 协议相关的重要且常见的技术。

- DNS 域名解析
- ARP 与 RARP 协议
- DHCP 动态获取 IP 地址
- NAT 网络地址转换
- ICMP 互联网控制报文协议
- IGMP 因特网组管理协

DNS

我们在上网的时候，通常使用的方式是域名，而不是 IP 地址，因为域名方便人类记忆。

那么实现这一技术的就是 **DNS 域名解析**，DNS 可以将域名网址自动转换为具体的 IP 地址。

> 域名的层级关系

DNS 中的域名都是用**句点**来分隔的，比如 `www.server.com`，这里的句点代表了不同层次之间的**界限**。

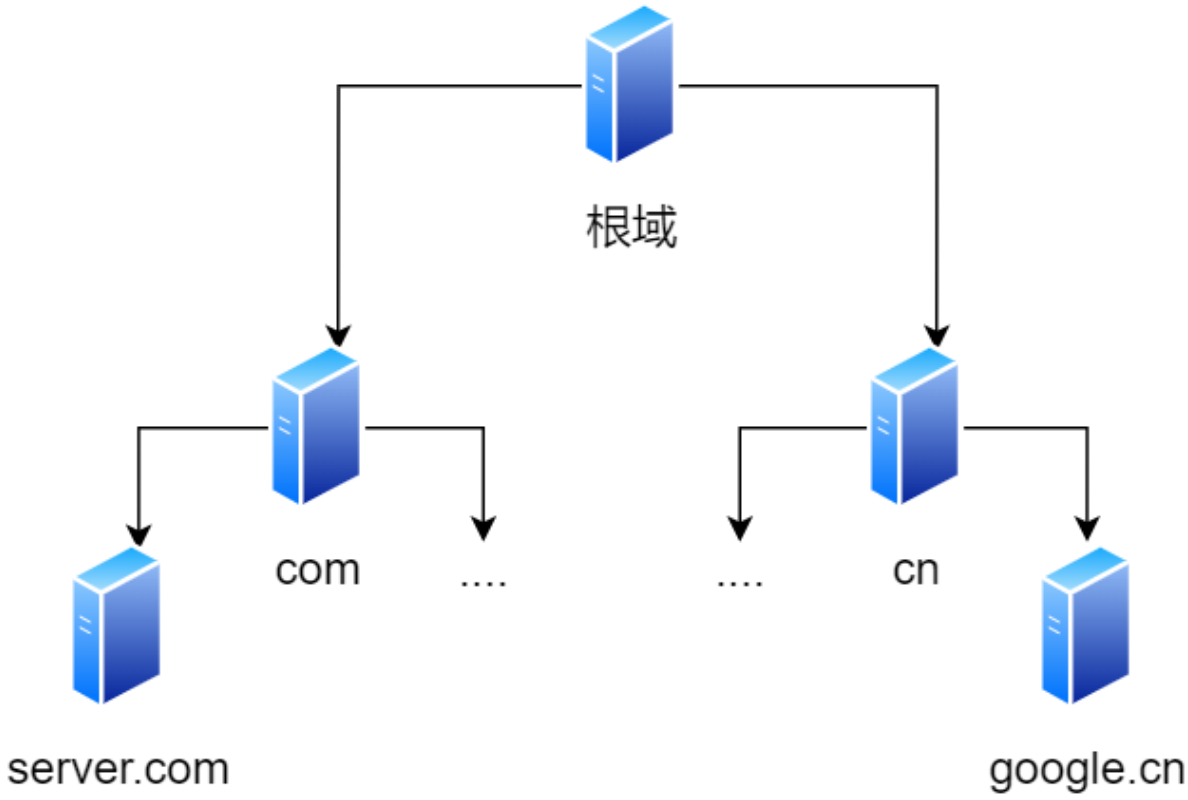
在域名中，**越靠右**的位置表示其层级**越高**。

毕竟域名是外国人发明，所以思维和中国人相反，比如说一个城市地点的时候，外国喜欢从小到大的方式顺序说起（如 XX 街道 XX 区 XX 市 XX 省），而中国则喜欢从大到小的顺序（如 XX 省 XX 市 XX 区 XX 街道）。

根域是在最顶层，它的下一层就是 com 顶级域，再下面是 server.com。

所以域名的层级关系类似一个树状结构：

- 根 DNS 服务器
- 顶级域 DNS 服务器（com）
- 权威 DNS 服务器（server.com）



根域的 DNS 服务器信息保存在互联网中所有的 DNS 服务器中。这样一来，任何 DNS 服务器就都可以找到并访问根域 DNS 服务器了。

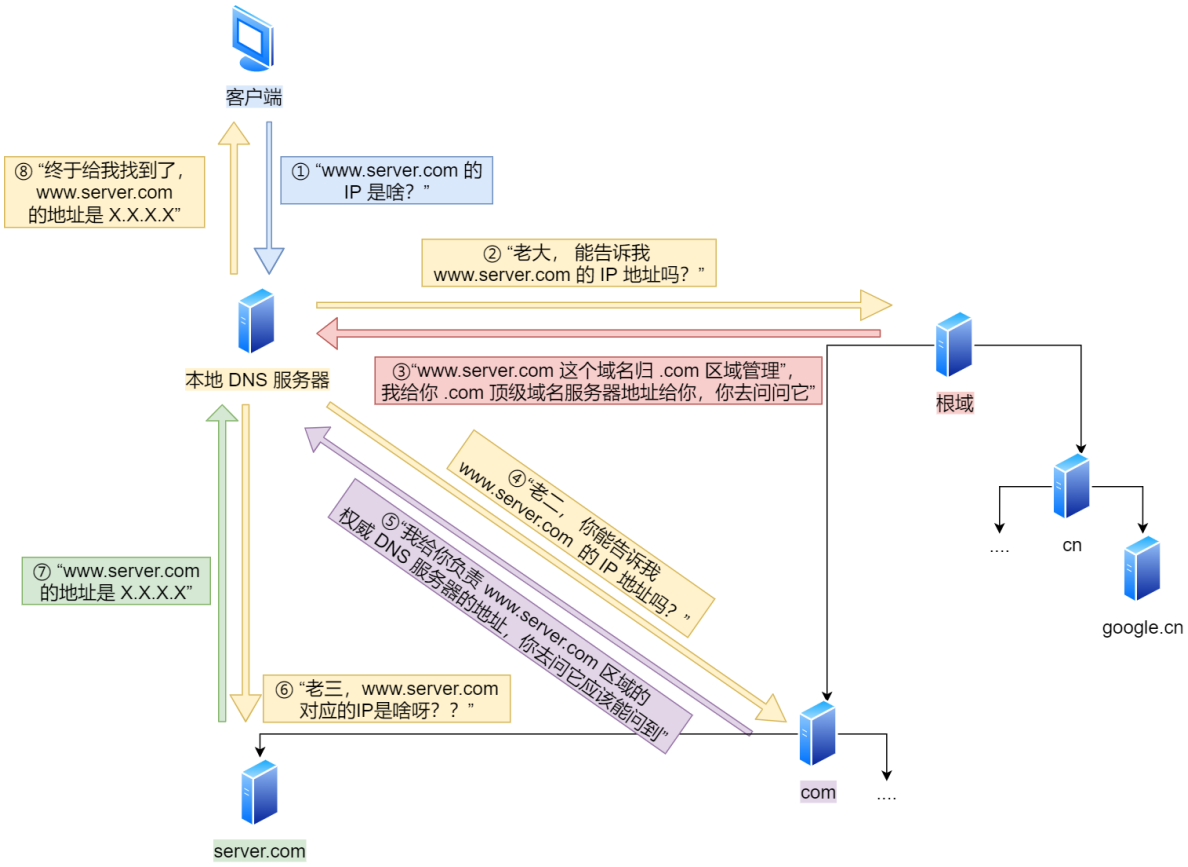
因此，客户端只要能够找到任意一台 DNS 服务器，就可以通过它找到根域 DNS 服务器，然后再一路顺藤摸瓜找到位于下层的某台目标 DNS 服务器。

> 域名解析的工作流程

浏览器首先看一下自己的缓存里有没有，如果没有就向操作系统的缓存要，还没有就检查本机域名解析文件 `hosts`，如果还是没有，就会 DNS 服务器进行查询，查询的过程如下：

1. 客户端首先会发出一个 DNS 请求，问 www.server.com 的 IP 是啥，并发给本地 DNS 服务器（也就是客户端的 TCP/IP 设置中填写的 DNS 服务器地址）。
2. 本地域名服务器收到客户端的请求后，如果缓存里的表格能找到 www.server.com，则它直接返回 IP 地址。如果没有，本地 DNS 会去问它的根域名服务器：“老大， 能告诉我 www.server.com 的 IP 地址吗？”根域名服务器是最高层次的，它不直接用于域名解析，但能指明一条道路。
3. 根 DNS 收到来自本地 DNS 的请求后，发现后置是 .com，说：“www.server.com 这个域名归 .com 区域管理”，我给你 .com 顶级域名服务器地址给你，你去问问它吧。”
4. 本地 DNS 收到顶级域名服务器的地址后，发起请求问“老二， 你能告诉我 www.server.com 的 IP 地址吗？”
5. 顶级域名服务器说：“我给你负责 www.server.com 区域的权威 DNS 服务器的地址，你去问它应该能问到”。
6. 本地 DNS 于是转向问权威 DNS 服务器：“老三，www.server.com对应的IP是啥呀？” server.com 的权威 DNS 服务器，它是域名解析结果的原出处。为啥叫权威呢？就是我的域名我做主。
7. 权威 DNS 服务器查询后将对应的 IP 地址 X.X.X.X 告诉本地 DNS。
8. 本地 DNS 再将 IP 地址返回客户端，客户端和目标建立连接。

至此，我们完成了 DNS 的解析过程。现在总结一下，整个过程我画成了一个图。



DNS 域名解析的过程蛮有意思的，整个过程就和我们日常生活中找人问路的过程类似，**只指路不带路**。

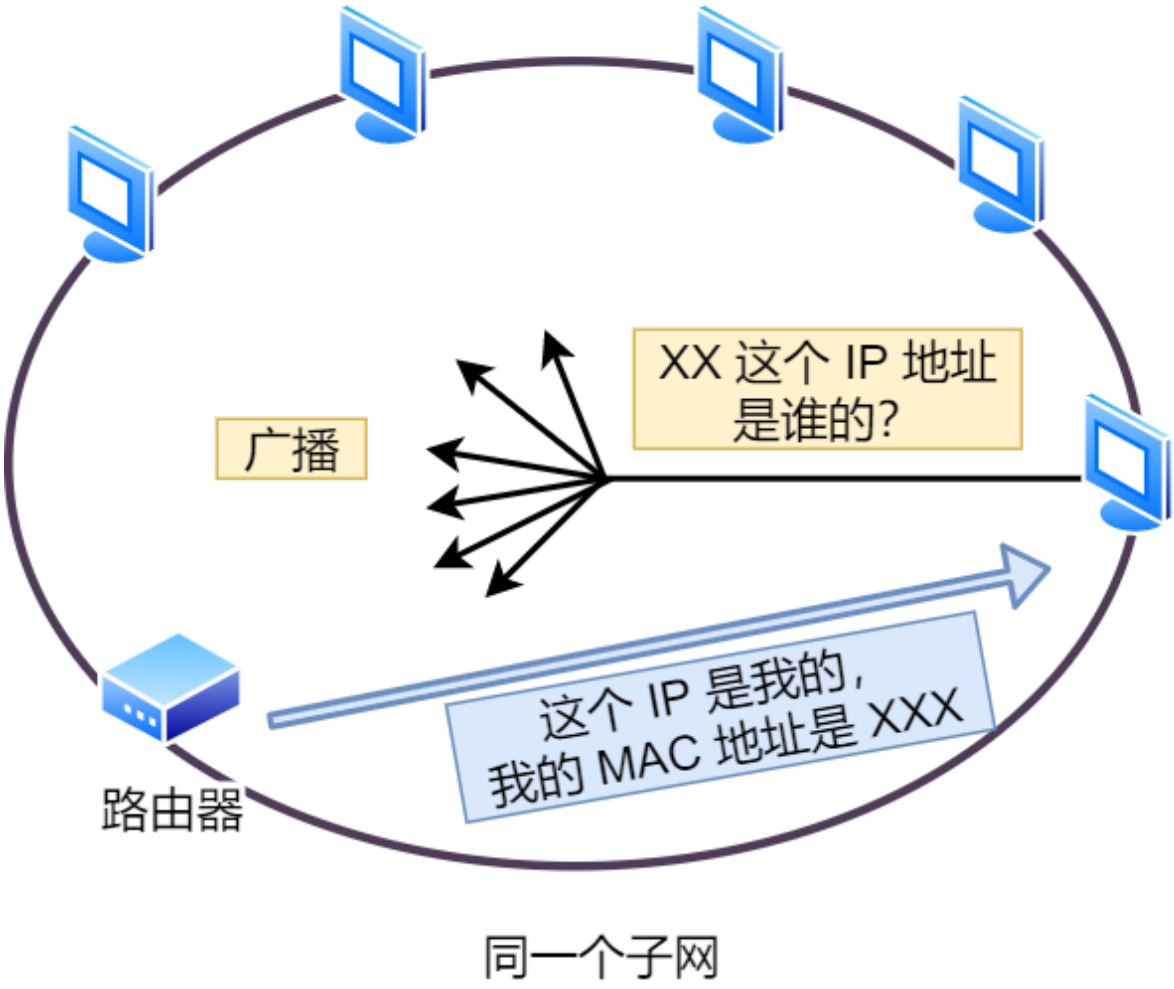
ARP

在传输一个 IP 数据报的时候，确定了源 IP 地址和目标 IP 地址后，就会通过主机「路由表」确定 IP 数据包下一跳。然而，网络层的下一层是数据链路层，所以我们还要知道「下一跳」的 MAC 地址。

由于主机的路由表中可以找到下一跳的 IP 地址，所以可以通过 **ARP 协议**，求得下一跳的 MAC 地址。

> 那么 ARP 又是如何知道对方 MAC 地址的呢?

简单地说，ARP 是借助 **ARP 请求与 ARP 响应**两种类型的包确定 MAC 地址的。



- 主机会通过**广播发送 ARP 请求**，这个包中包含了想要知道的 MAC 地址的主机 IP 地址。
- 当同个链路中的所有设备收到 ARP 请求时，会去拆开 ARP 请求包里的内容，如果 ARP 请求包中的目标 IP 地址与自己的 IP 地址一致，那么这个设备就将自己的 MAC 地址塞入 **ARP 响应包**返回给主机。

操作系统通常会把第一次通过 ARP 获取的 MAC 地址缓存起来，以便下次直接从缓存中找到对应 IP 地址的 MAC 地址。

不过，MAC 地址的缓存是有一定期限的，超过这个期限，缓存的内容将被清除。

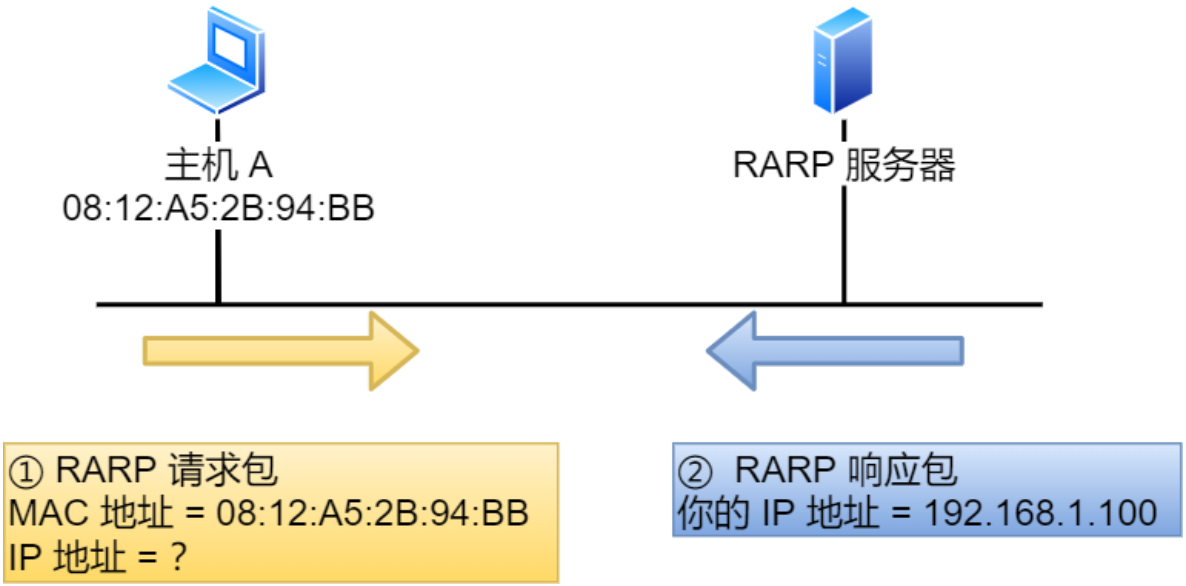
> RARP 协议你知道是什么吗?

ARP 协议是已知 IP 地址求 MAC 地址，那 RARP 协议正好相反，它是**已知 MAC 地址求 IP 地址**。例如将打印机服务器等小型嵌入式设备接入到网络时就经常会用得到。

通常这需要架设一台 `RARP` 服务器，在这个服务器上注册设备的 MAC 地址及其 IP 地址。然后再将这个设备接入到网络，接着：

- 该设备会发送一条「我的 MAC 地址是XXXX，请告诉我，我的IP地址应该是什么」的请求信息。
- RARP 服务器接到这个消息后返回「MAC地址为 XXXX 的设备，IP地址为 XXXX」的信息给这个设备。

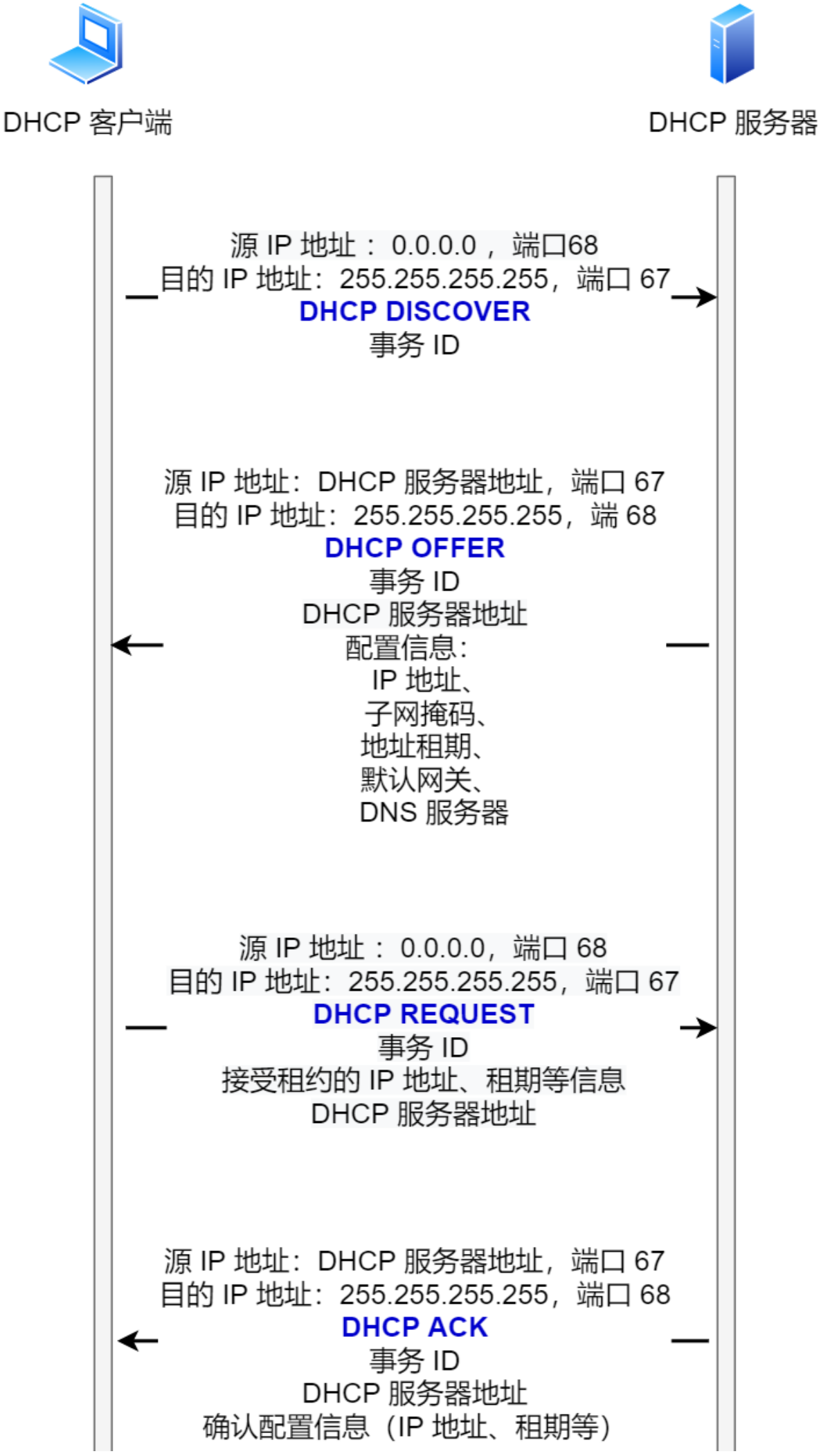
最后，设备就根据从 RARP 服务器所收到的应答信息设置自己的 IP 地址。



DHCP

DHCP 在生活中我们是很常见的了，我们的电脑通常都是通过 DHCP 动态获取 IP 地址，大大省去了配 IP 信息繁琐的过程。

接下来，我们来看看我们的电脑是如何通过 4 个步骤的过程，获取到 IP 的。


DHCP 客户端
DHCP 服务器
源 IP 地址：0.0.0.0，端口68
目的 IP 地址：255.255.255.255，端口 67
DHCP DISCOVER
事务 ID
源 IP 地址：DHCP 服务器地址，端口 67
目的 IP 地址：255.255.255.255，端 68
DHCP OFFER
事务 ID
DHCP 服务器地址
配置信息：
IP 地址、
子网掩码、
地址租期、
默认网关、
DNS 服务器
源 IP 地址：0.0.0.0，端口 68
目的 IP 地址：255.255.255.255，端口 67
DHCP REQUEST
事务 ID
接受租约的 IP 地址、租期等信息
DHCP 服务器地址
源 IP 地址：DHCP 服务器地址，端口 67
目的 IP 地址：255.255.255.255，端口 68
DHCP ACK
事务 ID
DHCP 服务器地址
确认配置信息（IP 地址、租期等）

先说明一点，DHCP 客户端进程监听的是 68 端口号，DHCP 服务端进程监听的是 67 端口号。

这 4 个步骤：

- 客户端首先发起 **DHCP 发现报文（DHCP DISCOVER）** 的 IP 数据报，由于客户端没有 IP 地址，也不知道 DHCP 服务器的地址，所以使用的是 UDP **广播**通信，其使用的广播目的地址是 255.255.255.255（端口 67） 并且使用 0.0.0.0（端口 68） 作为源 IP 地址。DHCP 客户端将该 IP 数据报传递给链路层，链路层然后将帧广播到所有的网络中设备。
- DHCP 服务器收到 DHCP 发现报文时，用 **DHCP 提供报文（DHCP OFFER）** 向客户端做出响应。该报文仍然使用 IP 广播地址 255.255.255.255，该报文信息携带服务器提供可租约的 IP 地址、子网掩码、默认网关、DNS 服务器以及 **IP 地址租用期**。
- 客户端收到一个或多个服务器的 DHCP 提供报文后，从中选择一个服务器，并向选中的服务器发送 **DHCP 请求报文（DHCP REQUEST**进行响应，回显配置的参数。
- 最后，服务端用 **DHCP ACK 报文**对 DHCP 请求报文进行响应，应答所要求的参数。

一旦客户端收到 DHCP ACK 后，交互便完成了，并且客户端能够在租用期内使用 DHCP 服务器分配的 IP 地址。

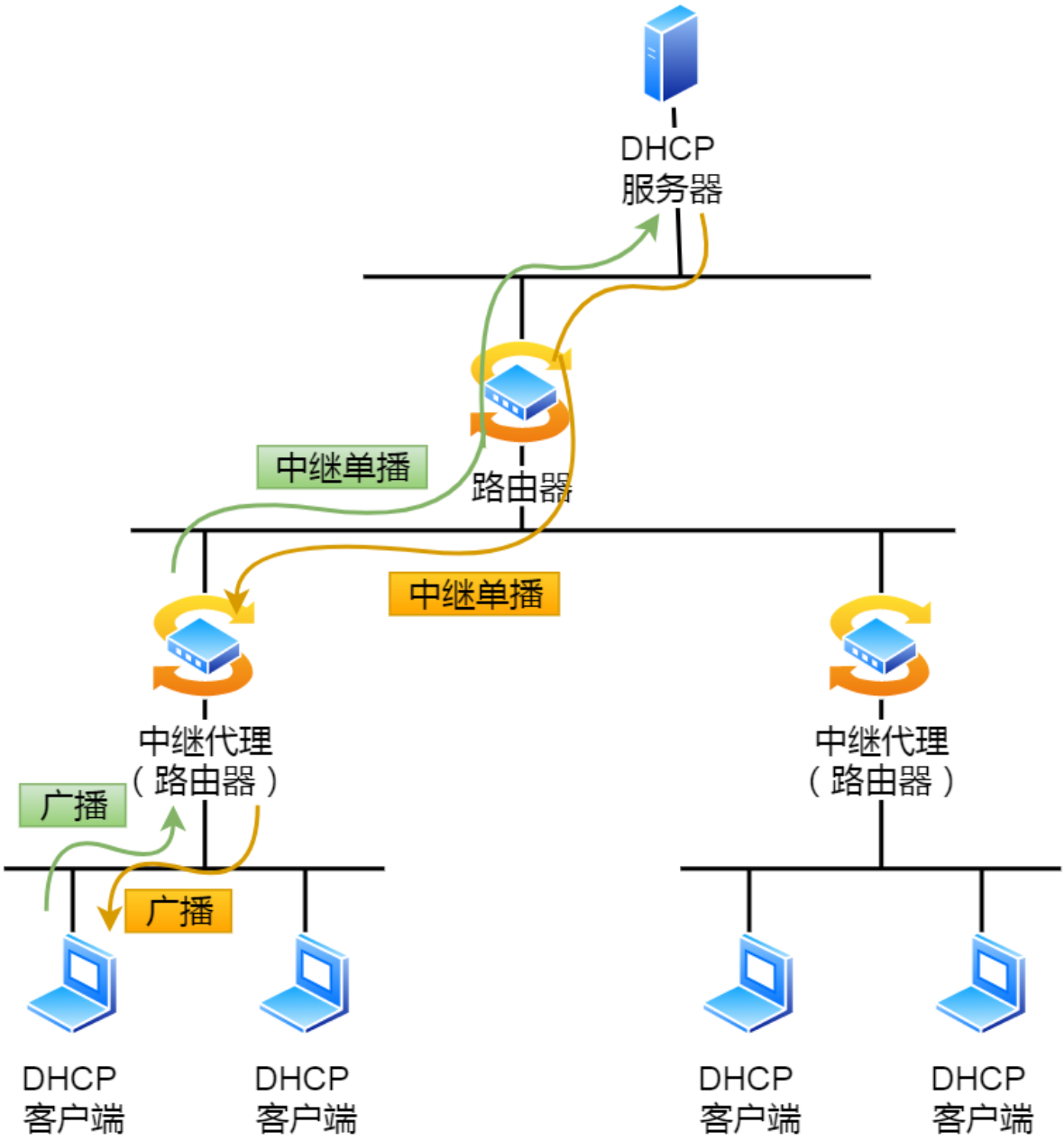
如果租约的 DHCP IP 地址快期后，客户端会向服务器发送 DHCP 请求报文：

- 服务器如果同意继续租用，则用 DHCP ACK 报文进行应答，客户端就会延长租期。
- 服务器如果不同意继续租用，则用 DHCP NACK 报文，客户端就要停止使用租约的 IP 地址。

可以发现，DHCP 交互中，**全程都是使用 UDP 广播通信**。

> 咦，用的是广播，那如果 DHCP 服务器和客户端不是在同一个局域网内，路由器又不会转发广播包，那不是每个网络都要配一个 DHCP 服务器?

所以，为了解决这一问题，就出现了 **DHCP 中继代理**。有了 DHCP 中继代理以后，**对不同网段的 IP 地址分配也可以由一个 DHCP 服务器统一进行管理。**



- DHCP 客户端会向 DHCP 中继代理发送 DHCP 请求包，而 DHCP 中继代理在收到这个广播包以后，再以**单播**的形式发给 DHCP 服务器。
- 服务器端收到该包以后再向 DHCP 中继代理返回应答，并由 DHCP 中继代理将此包广播给 DHCP 客户端 。

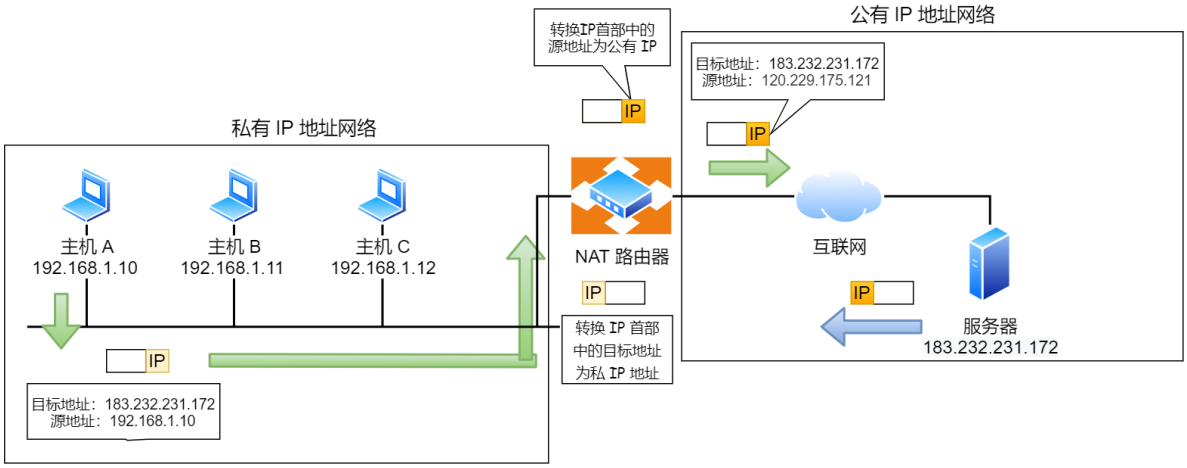
因此，DHCP 服务器即使不在同一个链路上也可以实现统一分配和管理IP地址。

NAT

IPv4 的地址是非常紧缺的，在前面我们也提到可以通过无分类地址来减缓 IPv4 地址耗尽的速度，但是互联网的用户增速是非常惊人的，所以 IPv4 地址依然有被耗尽的危险。

于是，提出了一种**网络地址转换 NAT** 的方法，再次缓解了 IPv4 地址耗尽的问题。

简单的来说 NAT 就是同个公司、家庭、教室内的主机对外部通信时，把私有 IP 地址转换成公有 IP 地址。



> 那不是 N 个私有 IP 地址，你就要 N 个公有 IP 地址？这怎么就缓解了 IPv4 地址耗尽的问题？这不瞎扯吗？

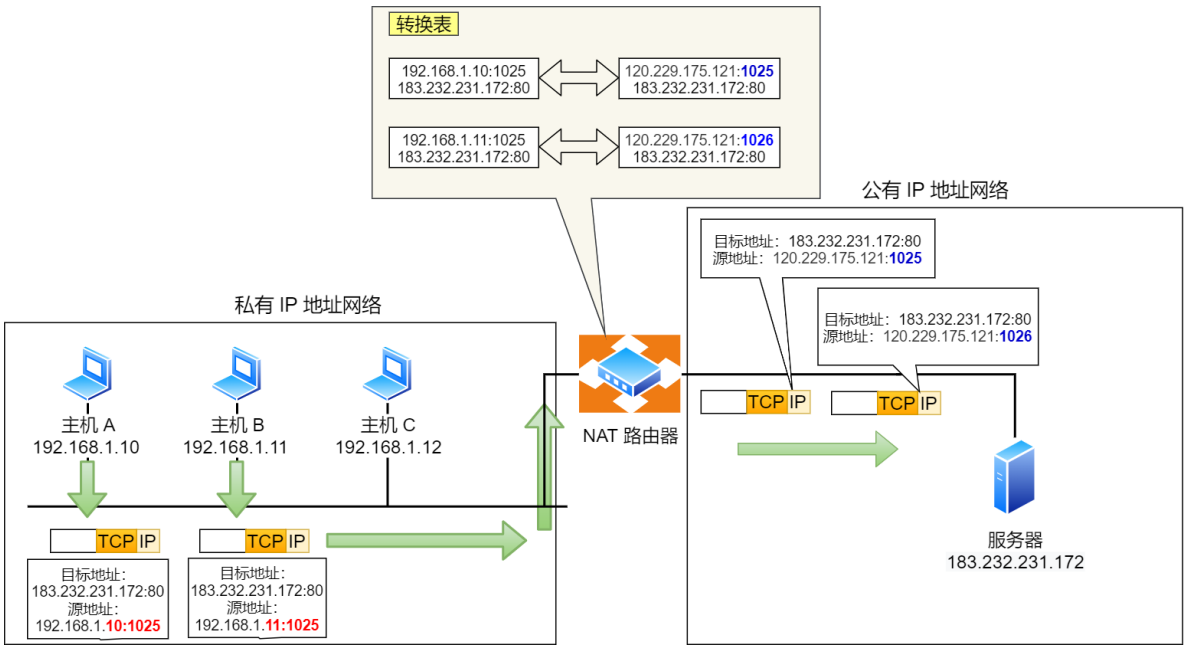
确实是，普通的 NAT 转换没什么意义。

由于绝大多数的网络应用都是使用传输层协议 TCP 或 UDP 来传输数据的。

因此，可以把 IP 地址 + 端口号一起进行转换。

这样，就用一个全球 IP 地址就可以了，这种转换技术就叫**网络地址与端口转换 NAPT。**

很抽象？来，看下面的图解就能瞬间明白了。



图中有两个客户端 192.168.1.10 和 192.168.1.11 同时与服务器 183.232.231.172 进行通信，并且这两个客户端的本地端口都是 1025。

此时，**两个私有 IP 地址都转换 IP 地址为公有地址 120.229.175.121，但是以不同的端口号作为区分。**

于是，生成一个 NAPT 路由器的转换表，就可以正确地转换地址跟端口的组合，令客户端 A、B 能同时与服务器之间进行通信。

这种转换表在 NAT 路由器上自动生成。例如，在 TCP 的情况下，建立 TCP 连接首次握手时的 SYN 包一经发出，就会生成这个表。而后又随着收到关闭连接时发出 FIN 包的确认应答从表中被删除。

> NAT 那么牛逼，难道就没缺点了吗?

当然有缺陷，肯定没有十全十美的方案。

由于 NAT/NAPT 都依赖于自己的转换表，因此会有以下的问题：

- 外部无法主动与 NAT 内部服务器建立连接，因为 NAPT 转换表没有转换记录。
- 转换表的生成与转换操作都会产生性能开销。
- 通信过程中，如果 NAT 路由器重启了，所有的 TCP 连接都将被重置。

> 如何解决 NAT 潜在的问题呢?

解决的方法主要有两种方法。

第一种就是改用 IPv6

IPv6 可用范围非常大，以至于每台设备都可以配置一个公有 IP 地址，就不搞那么多花里胡哨的地址转换了，但是 IPv6 普及速度还需要一些时间。

第二种 NAT 穿透技术

NAT 穿越技术拥有这样的功能，它能够让网络应用程序主动发现自己位于 NAT 设备之后，并且会主动获得 NAT 设备的公有 IP，并为自己建立端口映射条目，注意这些都是 NAT设备后的应用程序自动完成的。

也就是说，在 NAT 穿透技术中，NAT设备后的应用程序处于主动地位，它已经明确地知道 NAT 设备要修改它外发的数据包，于是它主动配合 NAT 设备的操作，主动地建立好映射，这样就不像以前由 NAT 设备来建立映射了。

说人话，就是客户端主动从 NAT 设备获取公有 IP 地址，然后自己建立端口映射条目，然后用这个条目对外通信，就不需要 NAT 设备来进行转换了。

ICMP

ICMP 全称是 **Internet Control Message Protocol**，也就是**互联网控制报文协议**。

里面有个关键词 —— **控制**，如何控制的呢?

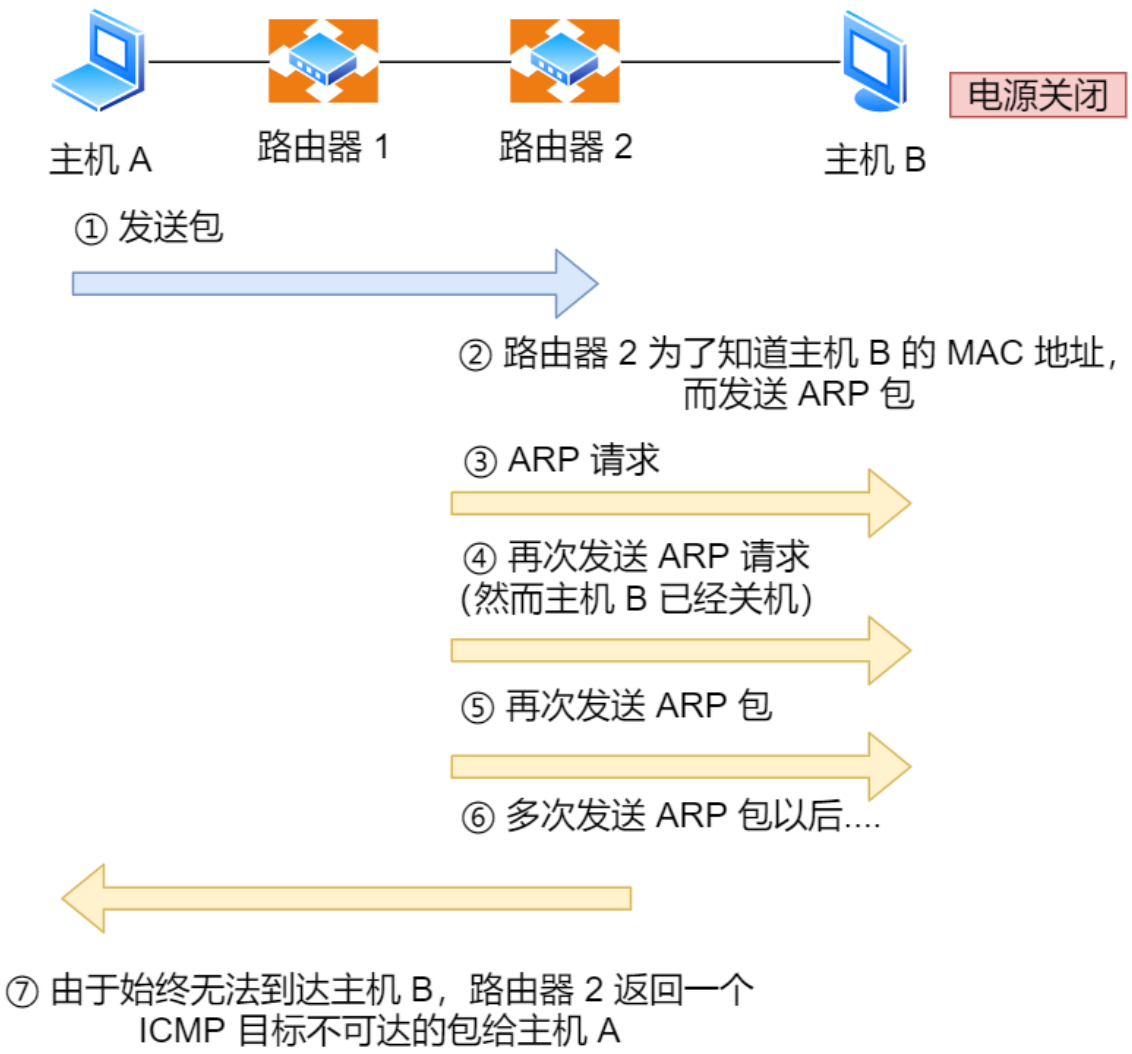
网络包在复杂的网络传输环境里，常常会遇到各种问题。

当遇到问题的时候，总不能死个不明不白，没头没脑的作风不是计算机网络的风格。所以需要传出消息，报告遇到了什么问题，这样才可以调整传输策略，以此来控制整个局面。

> ICMP 功能都有啥?

ICMP 主要的功能包括：**确认 IP 包是否成功送达目标地址、报告发送过程中 IP 包被废弃的原因和改善网络设置等。**

在 IP 通信中如果某个 IP 包因为某种原因未能达到目标地址，那么这个具体的原因将**由 ICMP 负责通知**。



如上图例子，主机 A 向主机 B 发送了数据包，由于某种原因，途中的路由器 2 未能发现主机 B 的存在，这时，路由器 2 就会向主机 A 发送一个 ICMP 目标不可达数据包，说明发往主机 B 的包未能成功。

ICMP 的这种通知消息会使用 IP 进行发送 。

因此，从路由器 2 返回的 ICMP 包会按照往常的路由控制先经过路由器 1 再转发给主机 A 。收到该 ICMP 包的主机 A 则分解 ICMP 的首部和数据域以后得知具体发生问题的原因。

> ICMP 类型

ICMP 大致可以分为两大类：

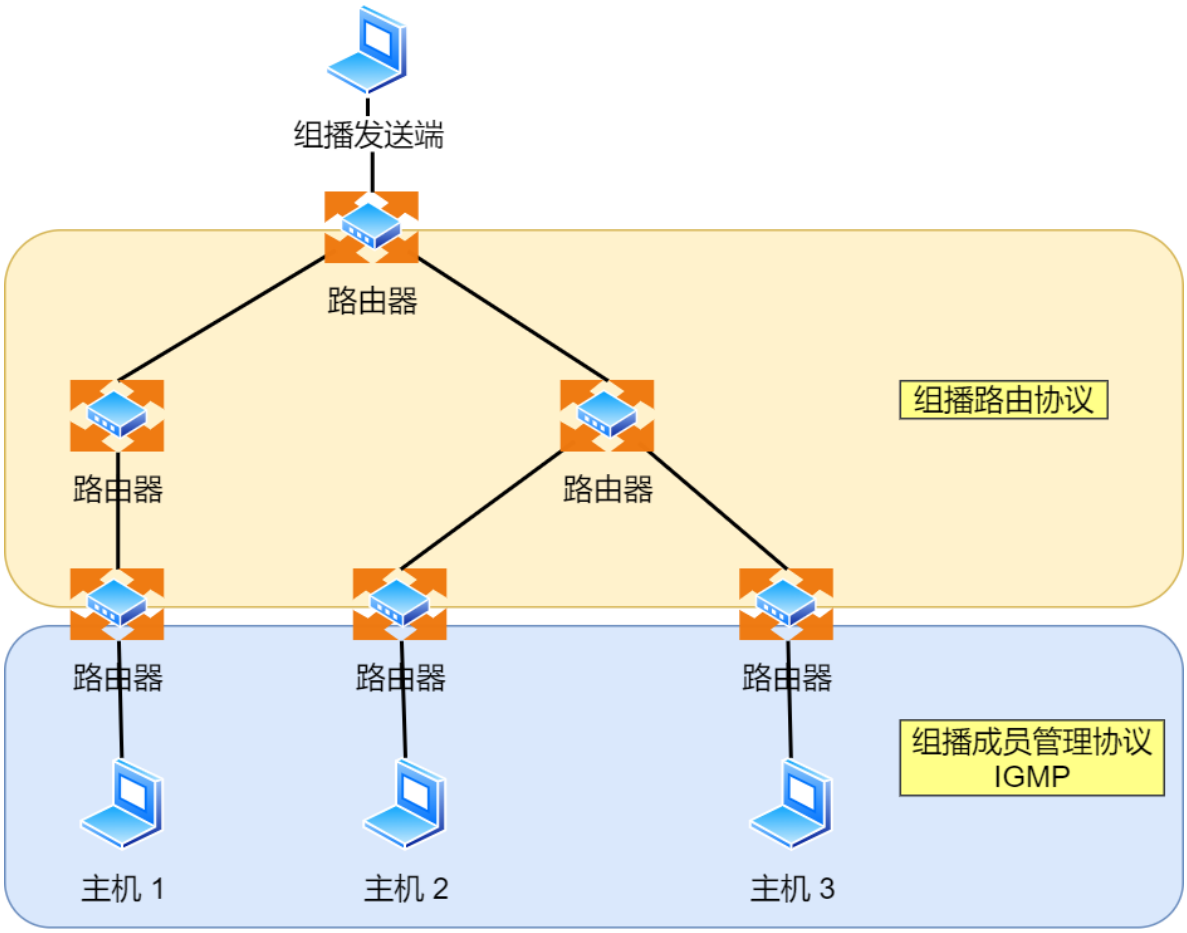
- 一类是用于诊断的查询消息，也就是「**查询报文类型**」
- 另一类是通知出错原因的错误消息，也就是「**差错报文类型**」

ICMP 类型		
	内容	种类
0	回送应答（Echo Reply）	查询报文类型
3	目标不可达（Destination Unreachable）	差错报文类型
4	原点抑制（Source Quench）	差错报文类型
5	重定向或改变路由（Redirect）	差错报文类型
8	回送请求（Echo Request）	查询报文类型
11	超时（Time Exceeded）	差错报文类型

IGMP

ICMP 跟 IGMP 是一点关系都没有的，就好像周杰与周杰伦的区别，大家不要混淆了。

在前面我们知道了组播地址，也就是 D 类地址，既然是组播，那就说明是只有一组的主机能收到数据包，不在一组的主机不能收到数组包，怎么管理是否是在一组呢？那么，就需要 `IGMP` 协议了。



IGMP 是因特网组管理协议，工作在主机（组播成员）和最后一跳路由之间，如上图中的蓝色部分。

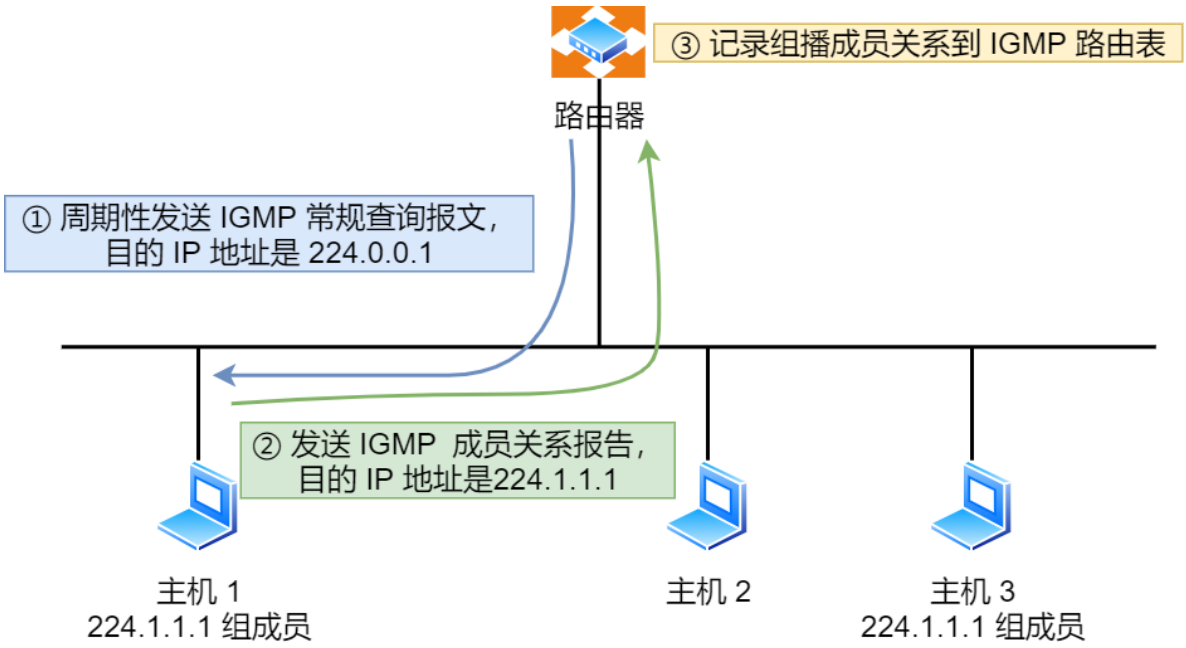
- IGMP 报文向路由器申请加入和退出组播组，默认情况下路由器是不会转发组播包到连接中的主机，除非主机通过 IGMP 加入到组播组，主机申请加入到组播组时，路由器就会记录 IGMP 路由器表，路由器后续就会转发组播包到对应的主机了。
- IGMP 报文采用 IP 封装，IP 头部的协议号为 2，而且 TTL 字段值通常为 1，因为 IGMP 是工作在主机与连接的路由器之间。

> IGMP 工作机制

IGMP 分为了三个版本分别是，IGMPv1、IGMPv2、IGMPv3。

接下来，以 `IGMPv2` 作为例子，说说**常规查询与响应和离开组播组**这两个工作机制。

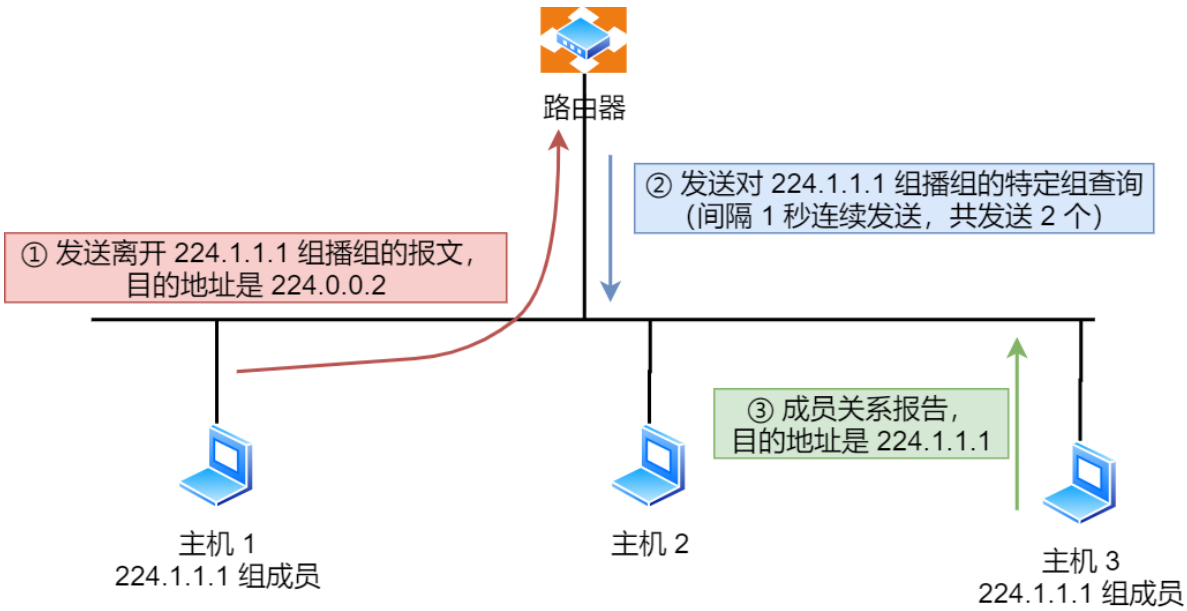
常规查询与响应工作机制



1. 路由器会周期性发送目的地址为 `224.0.0.1`（表示同一网段内所有主机和路由器）**IGMP 常规查询报文**。
2. 主机1 和 主机 3 收到这个查询，随后会启动「报告延迟计时器」，计时器的时间是随机的，通常是 0~10 秒，计时器超时后主机就会发送 **IGMP 成员关系报告报文**（源 IP 地址为自己主机的 IP 地址，目的 IP 地址为组播地址）。如果在定时器超时之前，收到同一个组内的其他主机发送的成员关系报告报文，则自己不再发送，这样可以减少网络中多余的 IGMP 报文数量。
3. 路由器收到主机的成员关系报文后，就会在 IGMP 路由表中加入该组播组，后续网络中一旦该组播地址的数据到达路由器，它会把数据包转发出去。

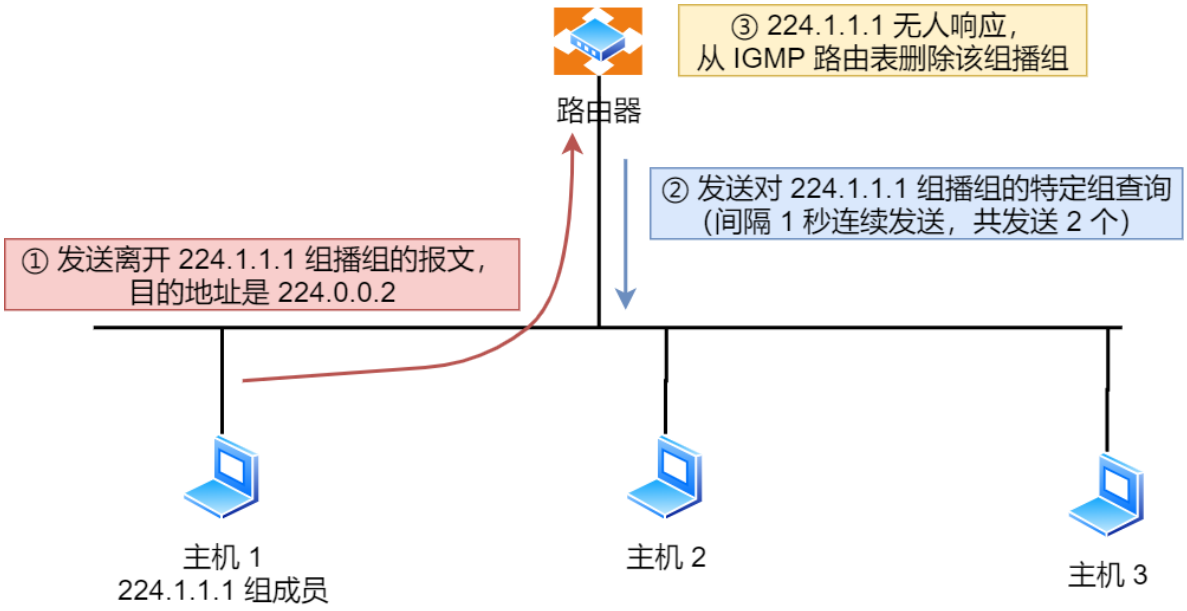
离开组播组工作机制

离开组播组的情况一，网段中仍有该组播组：



1. 主机 1 要离开组 224.1.1.1，发送 IGMPv2 离组报文，报文的目的地址是 224.0.0.2（表示发向网段内的所有路由器）
2. 路由器 收到该报文后，以 1 秒为间隔连续发送 IGMP 特定组查询报文（共计发送 2 个），以便确认该网络是否还有 224.1.1.1 组的其他成员。
3. 主机 3 仍然是组 224.1.1.1 的成员，因此它立即响应这个特定组查询。路由器知道该网络中仍然存在该组播组的成员，于是继续向该网络转发 224.1.1.1 的组播数据包。

离开组播组的情况二，网段中没有该组播组：



1. 主机 1 要离开组播组 224.1.1.1，发送 IGMP 离组报文。
2. 路由器收到该报文后，以 1 秒为间隔连续发送 IGMP 特定组查询报文（共计发送 2 个）。此时在该网段内，组 224.1.1.1 已经没有其他成员了，因此没有主机响应这个查询。
3. 一定时间后，路由器认为该网段中已经没有 224.1.1.1 组播组成员了，将不会再向这个网段转发该组播地址的数据包。

巨人的肩膀

[1] 计算机网络-自顶向下方法.陈鸣 译.机械工业出版社

[2] TCP/IP详解 卷1：协议.范建华 译.机械工业出版社

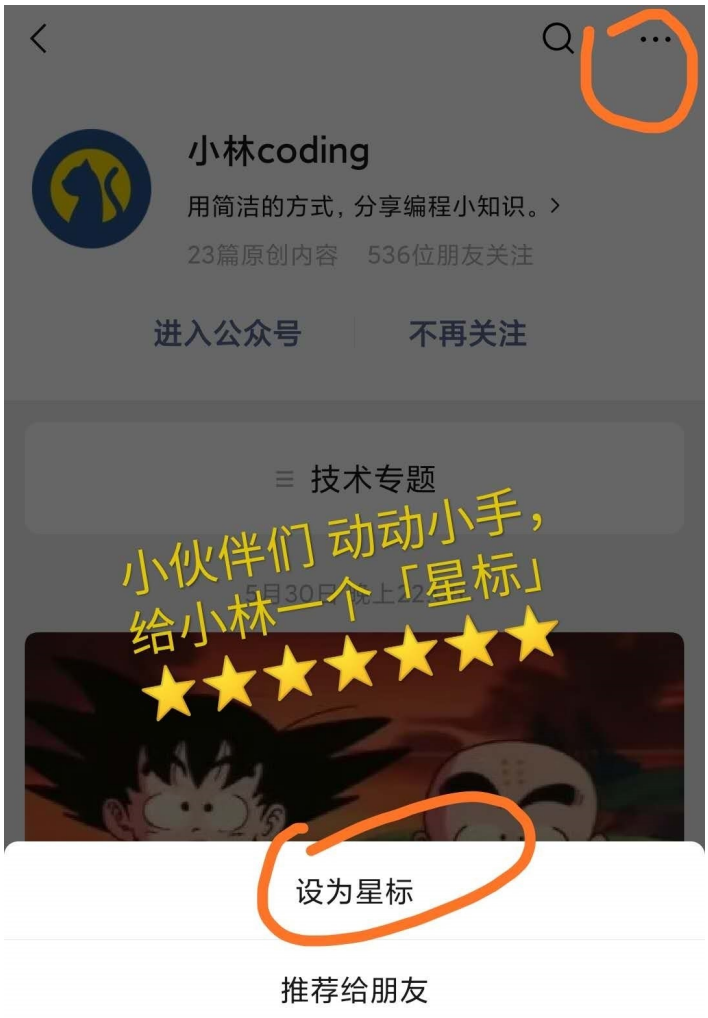
[3] 图解TCP/IP.竹下隆史.人民邮电出版社

唠叨唠叨

这一顿花样图解菜，做起来也不容易。

- 如果这合你口味，给小林点个「**赞**」，那小林我就心满意足啦。
- 如果口味过重，欢迎「**留言**」进行点评！

小林是专为大家图解的工具人，Goodbye，我们下次见！



扫一扫
关注爱图解的
「小林coding」

读者问答

> 读者问题：“组播不太懂。。。假设一台机器加入组播地址，需要把IP改成组播地址吗？如果离开某个组播地址，需要 dhcp 重新请求个IP吗？”

组播地址不是用于机器 ip 地址的，因为组播地址没有网络号和主机号，所以跟 dhcp 没关系。组播地址一般是用于 udp 协议，机器发送 UDP 组播数据时，目标地址填的是组播地址，那么在组播组内的机器都能收到数据包。

是否加入组播组和离开组播组，是由 socket 一个接口实现的，主机 ip 是不用改变的。

听说你 ping 用的很 6？给我图解一下 ping 的工作原理

前言

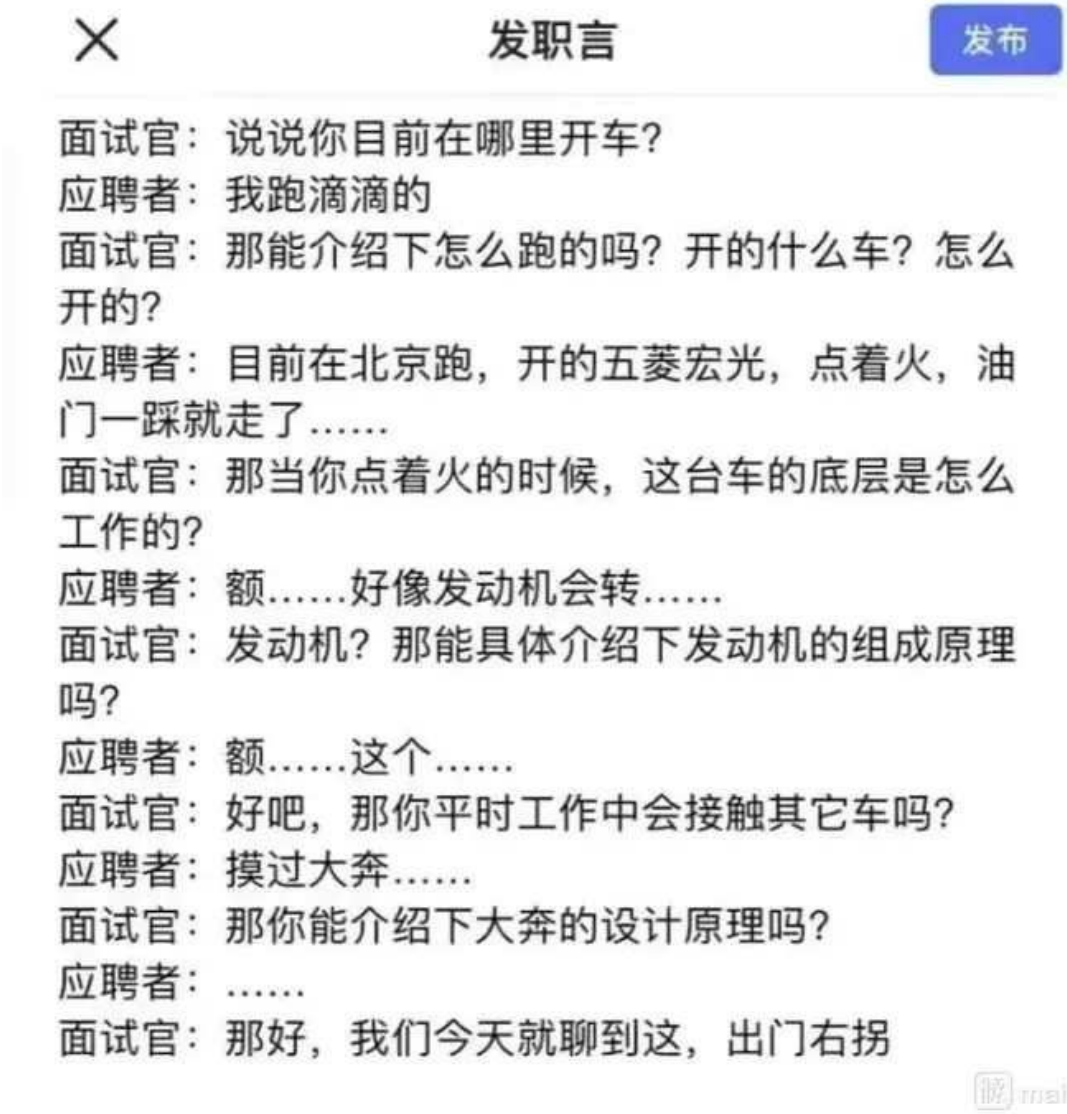
在日常生活或工作中，我们在判断与对方**网络是否畅通**，使用的最多的莫过于 `ping` 命令了。

“**那你知道 `ping` 是如何工作的吗？**” —— 来自小林的灵魂拷问

可能有的小伙伴奇怪的问：“我虽然不明白它的工作，但 ping 我也用的贼 6 啊！”

你用的是 6 ，但你在面试官面前，你就 6 不起来了，毕竟他们也爱问。

所以，我们要抱有「**知其然，知其所以然**」的态度，这样就能避免面试过程中，出门右拐的情况了。



不知道的小伙伴也没关系，今天我们就来搞定它，搞懂它。消除本次的问号，**让问号少一点**。

正文

IP协议的助手 —— ICMP 协议

ping 是基于 ICMP 协议工作的，所以要明白 ping 的工作，首先我们先来熟悉 **ICMP 协议**。

> ICMP 是什么?

ICMP 全称是 **Internet Control Message Protocol**，也就是**互联网控制报文协议**。

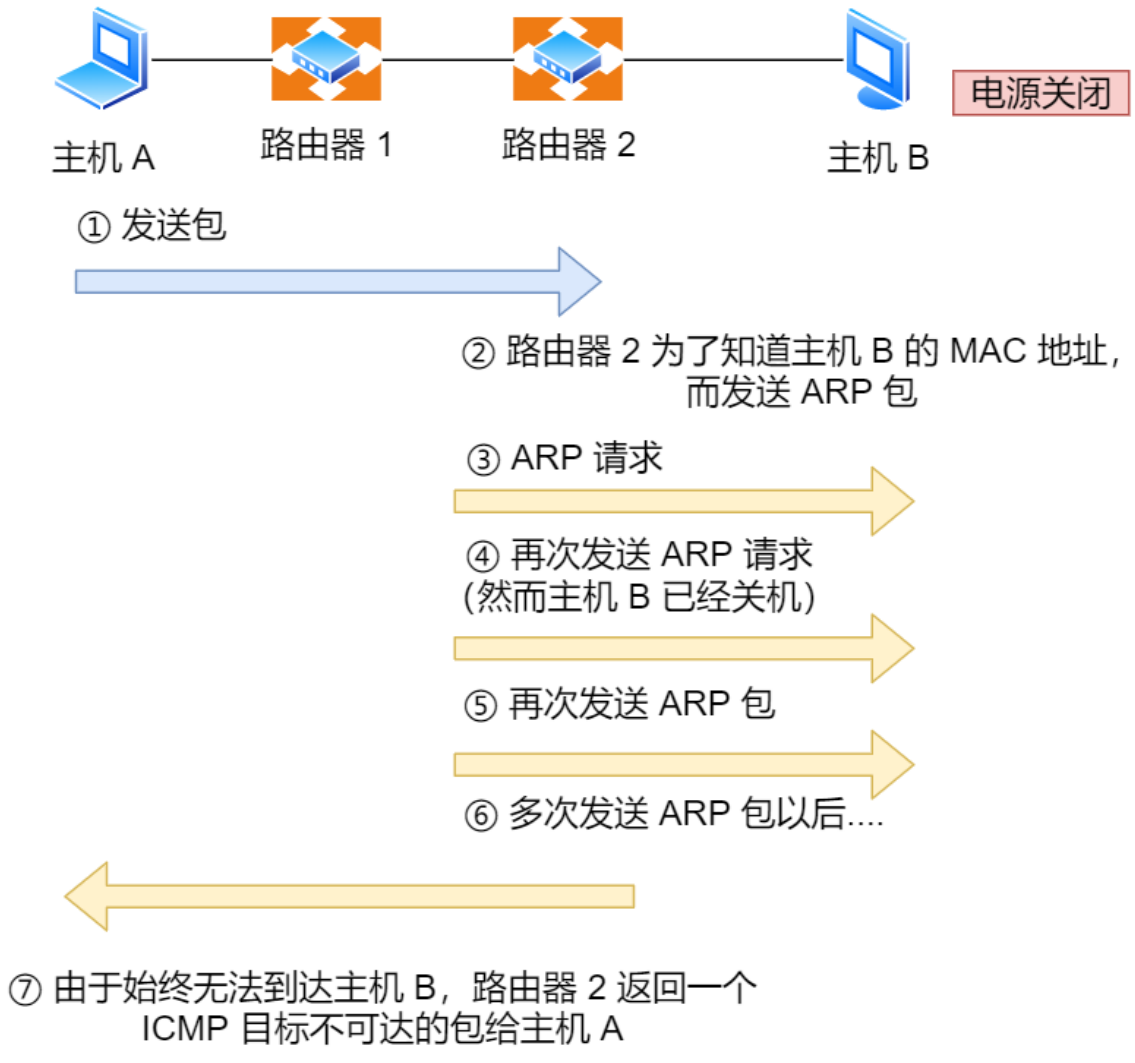
里面有个关键词 —— **控制**，如何控制的呢?

网络包在复杂的网络传输环境里，常常会遇到各种问题。当遇到问题的时候，总不能死的不明不白，没头没脑的作风不是计算机网络的风格。所以需要传出消息，报告遇到了什么问题，这样才可以调整传输策略，以此来控制整个局面。

> ICMP 功能都有啥?

ICMP 主要的功能包括：**确认 IP 包是否成功送达目标地址、报告发送过程中 IP 包被废弃的原因和改善网络设置等。**

在 IP 通信中如果某个 IP 包因为某种原因未能达到目标地址，那么这个具体的原因将**由 ICMP 负责通知**。



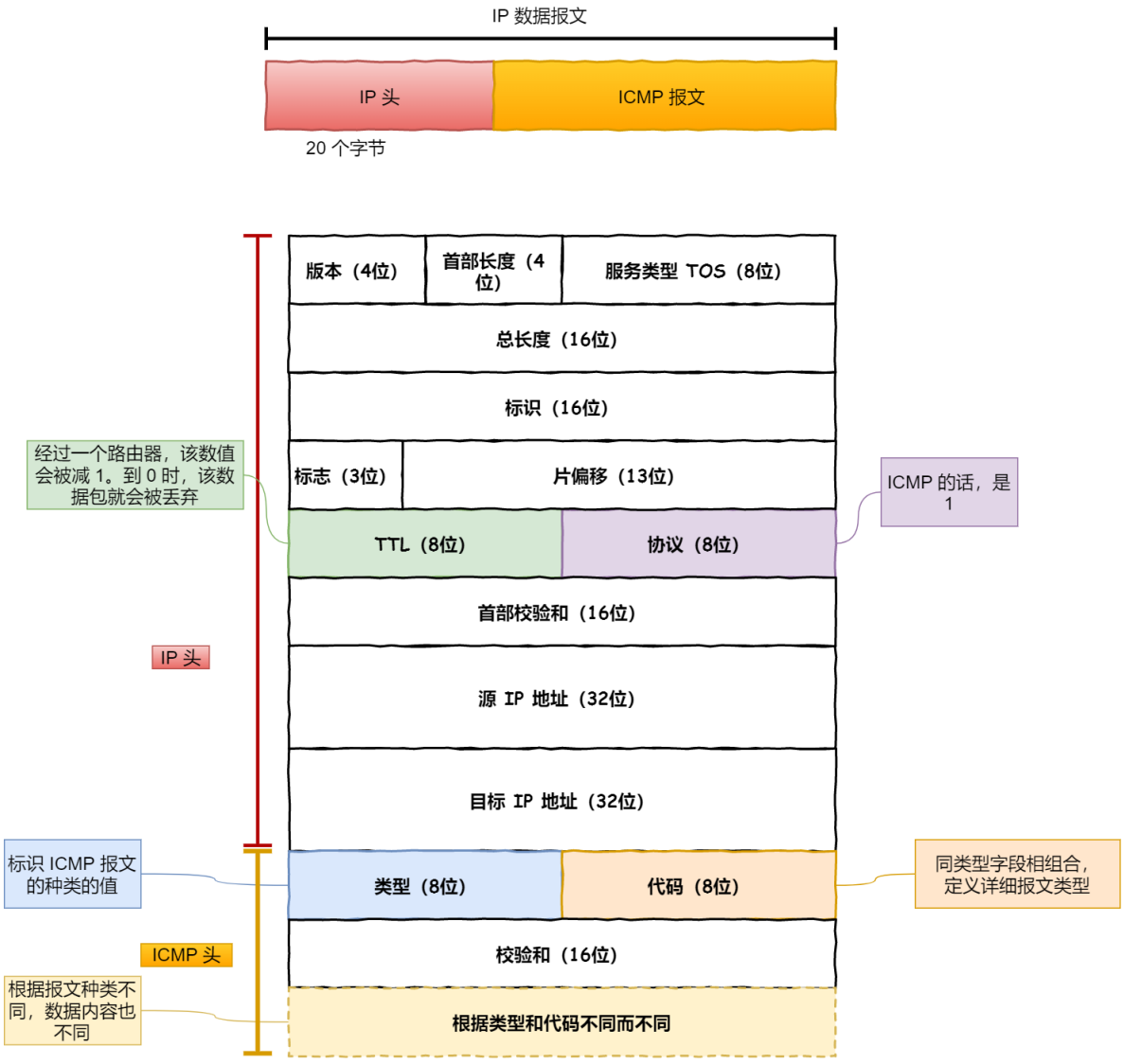
如上图例子，主机 A 向主机 B 发送了数据包，由于某种原因，途中的路由器 2 未能发现主机 B 的存在，这时，路由器 2 就会向主机 A 发送一个 ICMP 目标不可达数据包，说明发往主机 B 的包未能成功。

ICMP 的这种通知消息会使用 IP 进行发送 。

因此，从路由器 2 返回的 ICMP 包会按照往常的路由控制先经过路由器 1 再转发给主机 A 。收到该 ICMP 包的主机 A 则分解 ICMP 的首部和数据域以后得知具体发生问题的原因。

> ICMP 包头格式

ICMP 报文是封装在 IP 包里面，它工作在网络层，是 IP 协议的助手。



ICMP 包头的**类型**字段，大致可以分为两大类：

- 一类是用于诊断的查询消息，也就是「**查询报文类型**」
- 另一类是通知出错原因的错误消息，也就是「**差错报文类型**」

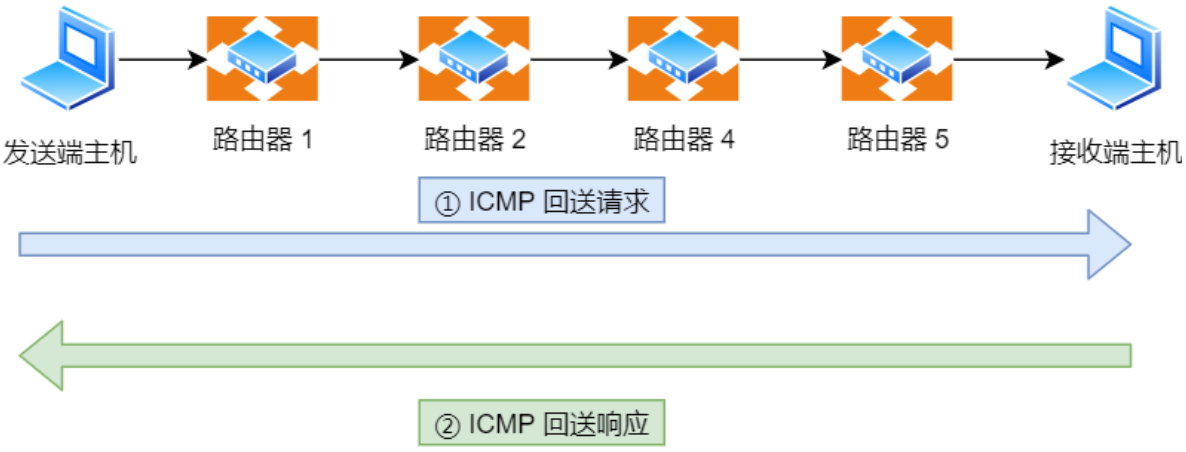
ICMP 类型		
	内容	种类
0	回送应答（Echo Reply）	查询报文类型
3	目标不可达（Destination Unreachable）	差错报文类型
4	原点抑制（Source Quench）	差错报文类型
5	重定向或改变路由（Redirect）	差错报文类型
8	回送请求（Echo Request）	查询报文类型
11	超时（Time Exceeded）	差错报文类型

查询报文类型

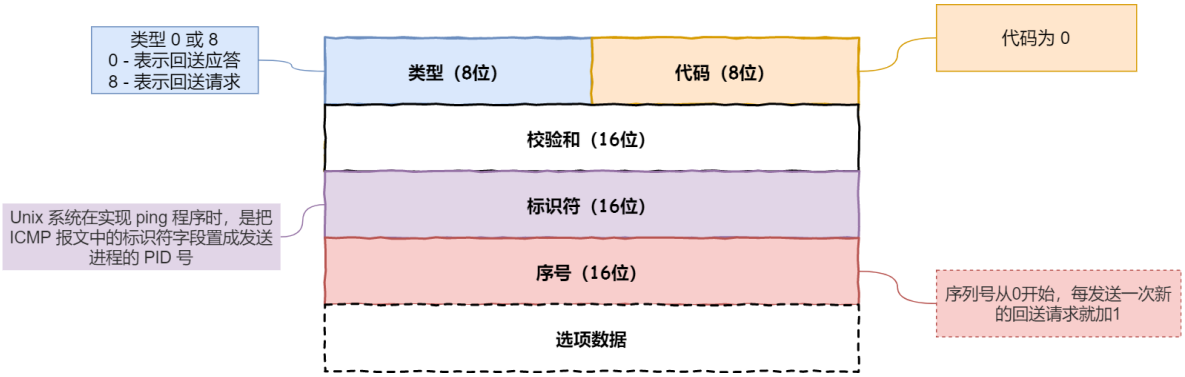
> 回送消息 —— 类型 `0` 和 `8`

回送消息用于进行通信的主机或路由器之间，判断所发送的数据包是否已经成功到达对端的一种消息，`ping` 命令就是利用这个消息实现的。

只要正常返回了 ICMP 回送响应，则代表发送端主机到接收端主机是否可达。



可以向对端主机发送**回送请求**的消息（`ICMP Echo Request Message`，类型 `8`），也可以接收对端主机发回来的**回送应答**消息（`ICMP Echo Reply Message`，类型 `0`）。



相比原生的 ICMP，这里多了两个字段：

- **标识符**：用以区分是哪个应用程序发 ICMP 包，比如用进程 `PID` 作为标识符；
- **序号**：序列号从 `0` 开始，每发送一次新的回送请求就会加 `1`， 可以用来确认网络包是否有丢失。

在**选项数据**中， `ping` 还会存放发送请求的时间值，来计算往返时间，说明路程的长短。

差错报文类型

接下来，说明几个常用的 ICMP 差错报文的例子：

- 目标不可达消息 —— 类型 为 `3`
- 原点抑制消息 —— 类型 `4`
- 重定向消息 —— 类型 `5`
- 超时消息 —— 类型 `11`

> 目标不可达消息（Destination Unreachable Message） —— 类型为 `3`

IP 路由器无法将 IP 数据包发送给目标地址时，会给发送端主机返回一个**目标不可达**的 ICMP 消息，并在这个消息中显示不可达的具体原因，原因记录在 ICMP 包头的**代码**字段。

由此，根据 ICMP 不可达的具体消息，发送端主机也就可以了解此次发送**不可达的具体原因**。

举例 6 种常见的目标不可达类型的**代码**：

ICMP 目标不可达类型的代码号	
	内容
0	网络不可达（ Network Unreachable）
1	主机不可达（Host Unreachable）
2	协议不可达（Protocol Unreachable）
3	端口不可达（Port Unreachable）
4	需要进行分片但设置了不分片（Fragmentation needed but no frag）

- 网络不可达代码为 `0`
- 主机不可达代码为 `1`
- 协议不可达代码为 `2`
- 端口不可达代码为 `3`
- 需要进行分片但设置了不分片位代码为 `4`

为了给大家说清楚上面的目标不可达的原因，**小林牺牲自己给大家送 5 次外卖。**

为什么要送外卖？别问，问就是为 `35` 岁的老林做准备 ...

a. 网络不可达代码为 0

外卖版本：

小林第一次送外卖时，小区里只有 A 和 B 区两栋楼，但送餐地址写的是 C 区楼，小林表示头上很多问号，压根就没这个地方。

正常版本：

IP 地址是分为网络号和主机号的，所以当路由器中的路由器表匹配不到接收方 IP 的网络号，就通过 ICMP 协议以**网络不可达**（ Network Unreachable ）的原因告知主机。

自从不再有网络分类以后，网络不可达也渐渐不再使用了。

b. 主机不可达代码为 1

外卖版本：

小林第二次送外卖时，这次小区有 5 层楼高的 C 区楼了，找到地方了，但送餐地址写的是 C 区楼 601 号房 ，说明找不到这个房间。

正常版本：

当路由表中没有该主机的信息，或者该主机没有连接到网络，那么会通过 ICMP 协议以**主机不可达**（ Host Unreachable ）的原因告知主机。

c. 协议不可达代码为 2

外卖版本：

小林第三次送外卖时，这次小区有 C 区楼，也有 601 号房，找到地方了，也找到房间了，但是一开门人家是外国人说的是英语，我说的是中文！语言不通，外卖送达失败~

正常版本：

当主机使用 TCP 协议访问对端主机时，能找到对端的主机了，可是对端主机的防火墙已经禁止 TCP 协议访问，那么会通过 ICMP 协议以**协议不可达**的原因告知主机。

d. 端口不可达代码为 3

外卖版本：

小林第四次送外卖时，这次小区有 C 区楼，也有 601 号房，找到地方了，也找到房间了，房间里的人也是说中文的人了，但是人家说他要的不是外卖，而是快递。。。

正常版本：

当主机访问对端主机 8080 端口时，这次能找到对端主机了，防火墙也没有限制，可是发现对端主机没有进程监听 8080 端口，那么会通过 ICMP 协议以**端口不可达**的原因告知主机。

e. 需要进行分片但设置了不分片位代码为 4

外卖版本：

小林第五次送外卖时，这次是个吃播博主点了 100 份外卖，但是吃播博主要求一次性要把全部外卖送达，小林的一台电动车装不下呀，这样就没办法送达了。

正常版本：

发送端主机发送 IP 数据报时，将 IP 首部的**分片禁止标志位**设置为 `1` 。根据这个标志位，途中的路由器遇到超过 MTU 大小的数据包时，不会进行分片，而是直接抛弃。

随后，通过一个 ICMP 的不可达消息类型，**代码为 4** 的报文，告知发送端主机。

> 原点抑制消息（ICMP Source Quench Message） —— 类型 `4`

在使用低速广域线路的情况下，连接 WAN 的路由器可能会遇到网络拥堵的问题。

`ICMP` 原点抑制消息的目的就是**为了缓和这种拥堵情况**。

当路由器向低速线路发送数据时，其发送队列的缓存变为零而无法发送出去时，可以向 IP 包的源地址发送一个 ICMP **原点抑制消息**。

收到这个消息的主机借此了解在整个线路的某一处发生了拥堵的情况，从而增大 IP 包的传输间隔，减少网络拥堵的情况。

然而，由于这种 ICMP 可能会引起不公平的网络通信，一般不被使用。

> 重定向消息（ICMP Redirect Message） —— 类型 `5`

如果路由器发现发送端主机使用了「不是最优」的路径发送数据，那么它会返回一个 ICMP **重定向消息**给这个主机。

在这个消息中包含了**最合适的路由信息和源数据**。这主要发生在路由器持有更好的路由信息的情况下。路由器会通过这样的 ICMP 消息告知发送端，让它下次发给另外一个路由器。

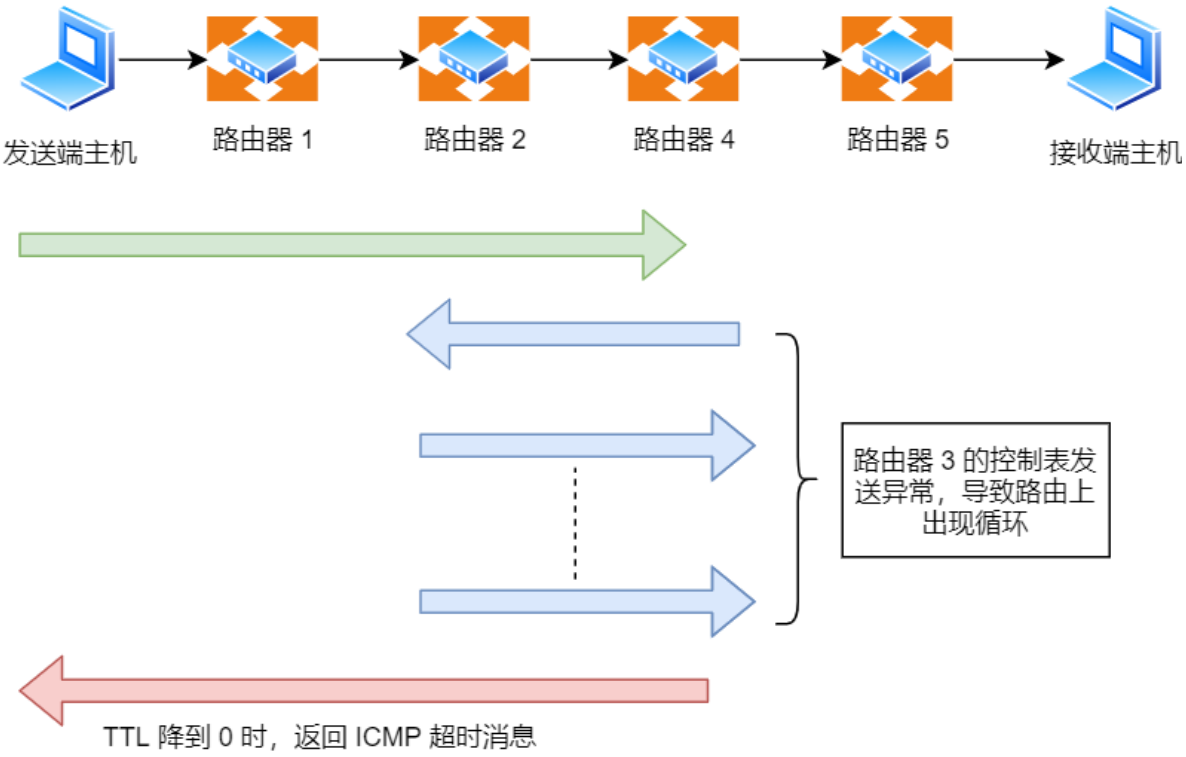
好比，小林本可以过条马路就能到的地方，但小林不知道，所以绕了一圈才到，后面小林知道后，下次小林就不会那么**傻**再绕一圈了。

> 超时消息（ICMP Time Exceeded Message） —— 类型 `11`

IP 包中有一个字段叫做 `TTL` （`Time To Live`，生存周期），它的**值随着每经过一次路由器就会减 1，直到减到 0 时该 IP 包会被丢弃。**

此时，路由器将会发送一个 ICMP **超时消息**给发送端主机，并通知该包已被丢弃。

设置 IP 包生存周期的主要目的，是为了在路由控制遇到问题发生循环状况时，避免 IP 包无休止地在网络上被转发。



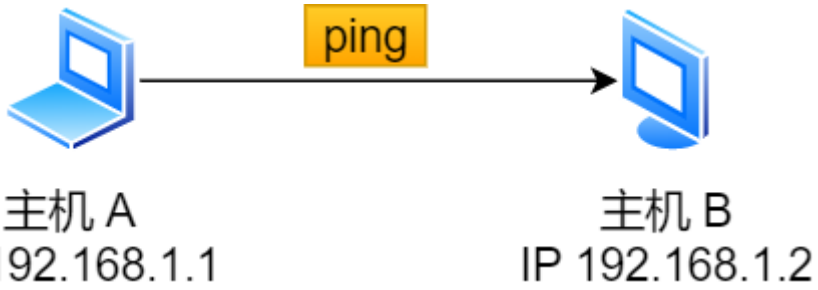
此外，有时可以用 TTL 控制包的到达范围，例如设置一个**较小的 TTL 值**。

---

ping —— 查询报文类型的使用

接下来，我们重点来看 `ping` 的**发送和接收过程**。

同个子网下的主机 A 和 主机 B，主机 A 执行 `ping` 主机 B 后，我们来看看其间发送了什么?

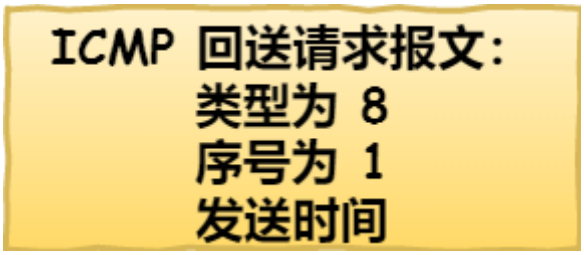


ping 命令执行的时候，源主机首先会构建一个 **ICMP 回送请求消息**数据包。

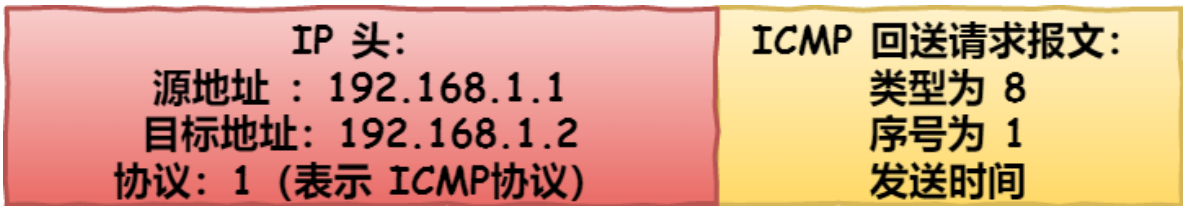
ICMP 数据包内包含多个字段，最重要的是两个:

- 第一个是**类型**，对于回送请求消息而言该字段为 `8` ;
- 另外一个是**序号**，主要用于区分连续 ping 的时候发出的多个数据包。

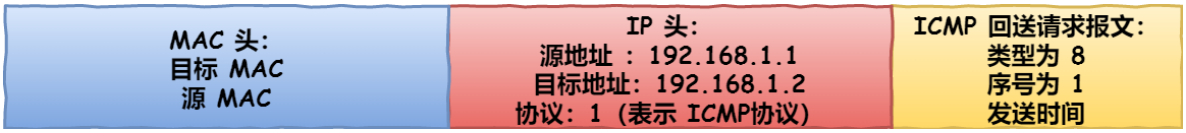
每发出一个请求数据包，序号会自动加 `1` 。为了能够计算往返时间 `RTT` ，它会在报文的数据部分插入发送时间。



然后，由 ICMP 协议将这个数据包连同地址 192.168.1.2 一起交给 IP 层。IP 层将以 192.168.1.2 作为**目的地址**，本机 IP 地址作为**源地址**，**协议**字段设置为 `1` 表示是 `ICMP` 协议，再加上一些其他控制信息，构建一个 `IP` 数据包。



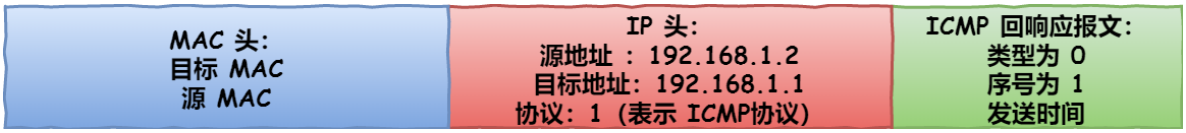
接下来，需要加入 `MAC` 头。如果在本地 ARP 映射表中查找出 IP 地址 192.168.1.2 所对应的 MAC 地址，则可以直接使用；如果没有，则需要发送 `ARP` 协议查询 MAC 地址，获得 MAC 地址后，由数据链路层构建一个数据帧，目的地址是 IP 层传过来的 MAC 地址，源地址则是本机的 MAC 地址；还要附加上一些控制信息，依据以太网的介质访问规则，将它们传送出去。



主机 `B` 收到这个数据帧后，先检查它的目的 MAC 地址，并和本机的 MAC 地址对比，如符合，则接收，否则就丢弃。

接收后检查该数据帧，将 IP 数据包从帧中提取出来，交给本机的 IP 层。同样，IP 层检查后，将有用的信息提取后交给 ICMP 协议。

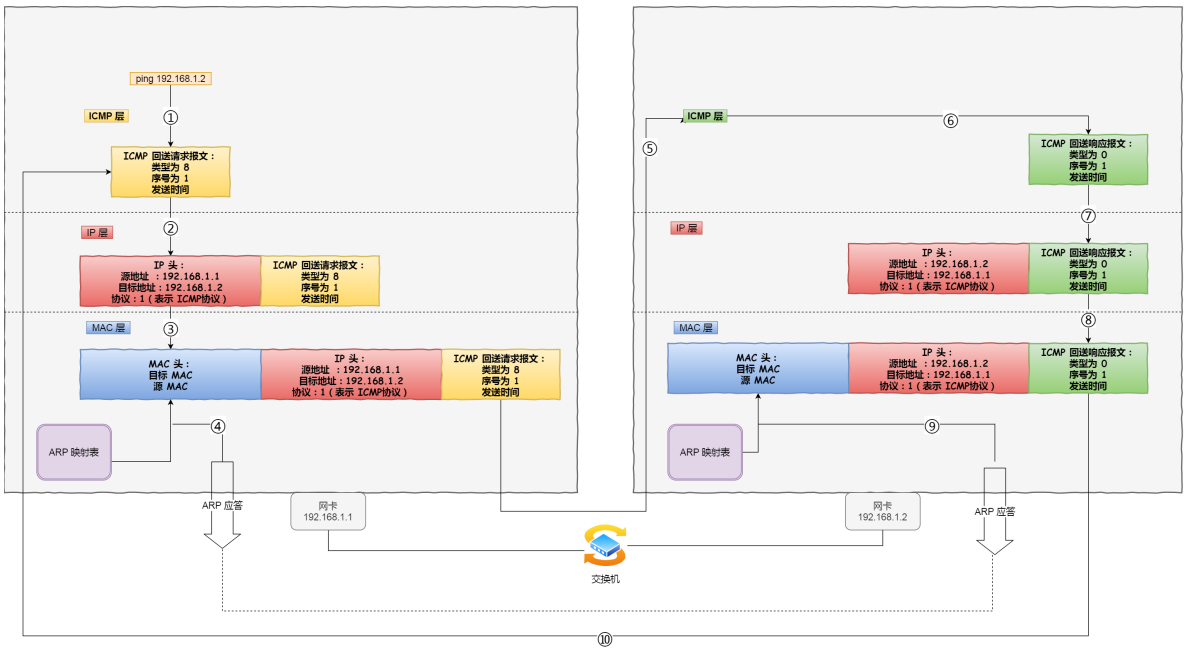
主机 `B` 会构建一个 **ICMP 回送响应消息**数据包，回送响应数据包的**类型**字段为 `0`，**序号**为接收到的请求数据包中的序号，然后再发送出去给主机 A。



在规定的时候间内，源主机如果没有接到 ICMP 的应答包，则说明目标主机不可达；如果接收到了 ICMP 回送响应消息，则说明目标主机可达。

此时，源主机会检查，用当前时刻减去该数据包最初从源主机上发出的时刻，就是 ICMP 数据包的时间延迟。

针对上面发送的事情，总结成了如下图：



当然这只是最简单的，同一个局域网里面的情况。如果跨网段的话，还会涉及网关的转发、路由器的转发等等。

但是对于 ICMP 的头来讲，是没什么影响的。会影响的是根据目标 IP 地址，选择路由的下一跳，还有每经过一个路由器到达一个新的局域网，需要换 MAC 头里面的 MAC 地址。

说了这么多，可以看出 ping 这个程序是**使用了 ICMP 里面的 ECHO REQUEST（类型为 8 ） 和 ECHO REPLY （类型为 0）**。

traceroute —— 差错报文类型的使用

有一款充分利用 ICMP **差错报文类型**的应用叫做 `traceroute` （在UNIX、MacOS中是这个命令，而在Windows中对等的命令叫做 tracert ）。

1. traceroute 作用一

traceroute 的第一个作用就是**故意设置特殊的 TTL，来追踪去往目的地时沿途经过的路由器。**

traceroute 的参数指向某个**目的 IP 地址**：

```
traceroute 192.168.1.100
```

> 这个作用是如何工作的呢?

它的原理就是利用 IP 包的**生存期限** 从 `1` 开始按照顺序递增的同时发送 **UDP 包**，强制接收 **ICMP 超时消息**的一种方法。

比如，将 TTL 设置 为 `1` ，则遇到第一个路由器，就牺牲了，接着返回 ICMP 差错报文网络包，类型是**时间超时**。

接下来将 TTL 设置为 `2` ，第一个路由器过了，遇到第二个路由器也牺牲了，也同时返回了 ICMP 差错报文数据包，如此往复，直到到达目的主机。

这样的过程，traceroute 就可以拿到了所有的路由器 IP。

当然有的路由器根本就不会返回这个 ICMP，所以对于有的公网地址，是看不到中间经过的路由的。

> 发送方如何知道发出的 UDP 包是否到达了目的主机呢?

traceroute 在发送 `UDP` 包时，会填入一个**不可能的端口号**值作为 UDP 目标端口号（大于 `3000`）。当目的主机，收到 UDP 包后，会返回 ICMP 差错报文消息，但这个差错报文消息的类型是「**端口不可达**」。

所以，**当差错报文类型是端口不可达时，说明发送方发出的 UDP 包到达了目的主机。**

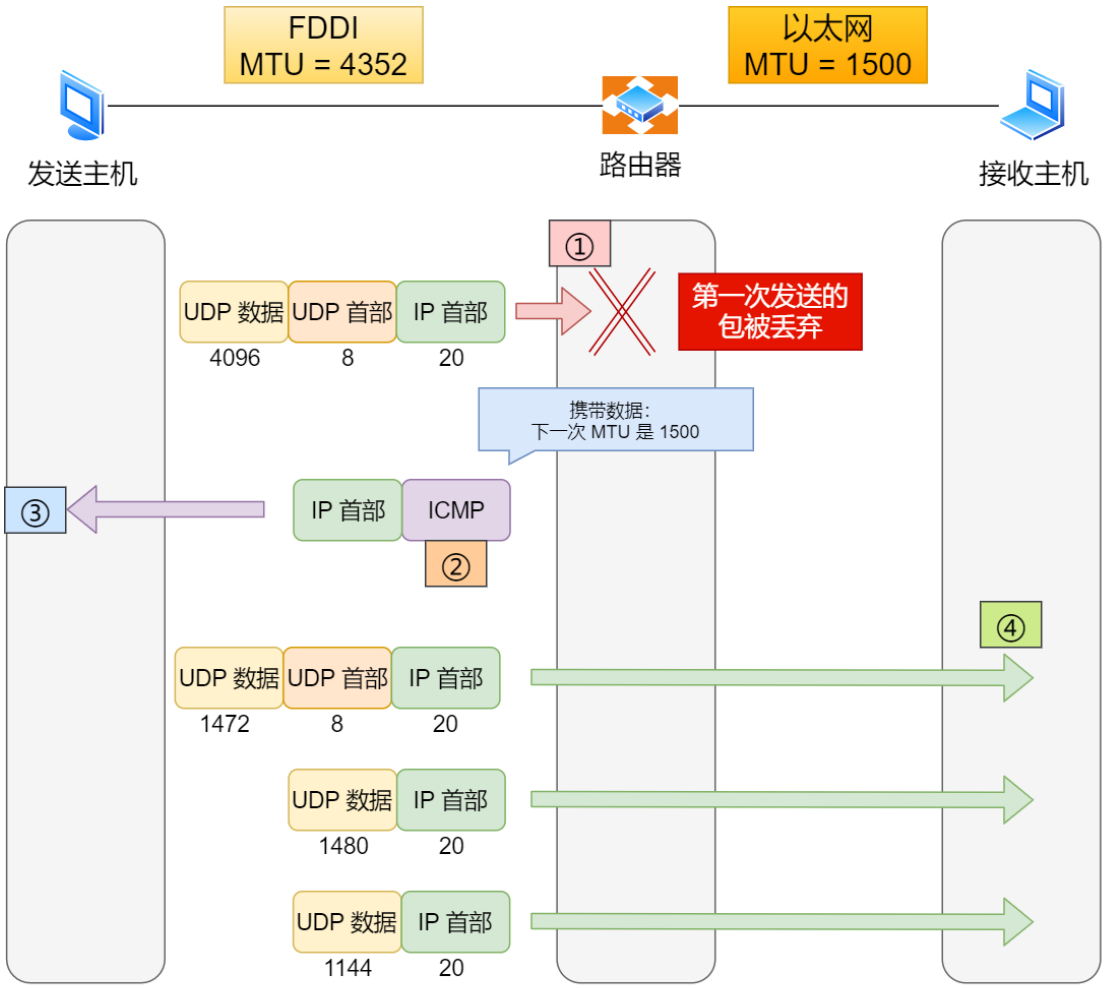
2. traceroute 作用二

traceroute 还有一个作用是**故意设置不分片，从而确定路径的 MTU**。

> 这么做是为了什么?

这样做的目的是为了**路径MTU发现**。

因为有的时候我们并不知道路由器的 `MTU` 大小，以太网的数据链路上的 `MTU` 通常是 `1500` 字节，但是非以外网的 `MTU` 值就不一样了，所以我们要知道 `MTU` 的大小，从而控制发送的包大小。



① 发送时 IP 首部 的分片标志位设置为不分片。路由器将丢弃包

② 由 ICMP 通知下一次 MTU 的大小。

③ 由于 UDP 中没有重发处理，应用在发送下一个消息时才会被分片。
具体来说，就是指 UDP 传过来的 「UDP 首部 + UDP 数据」在 IP 层被分片。
对于 IP ，它并不区分 UDP 首部和应用的数据。

④ 所有的分片到达目标主机后被重组，再传给接收主机的 UDP 层。

它的工作原理如下：

首先在发送端主机发送 IP 数据报时，将 IP 包首部的**分片禁止标志位设置为 1**。根据这个标志位，途中的路由器不会对大数据包进行分片，而是将包丢弃。

随后，通过一个 ICMP 的不可达消息将**数据链路上 MTU 的值**一起给发送主机，不可达消息的类型为「**需要进行分片但设置了不分片位**」。

发送主机端每次收到 ICMP 差错报文时就**减少**包的大小，以此来定位一个合适的 MTU 值，以便能到达目标主机。

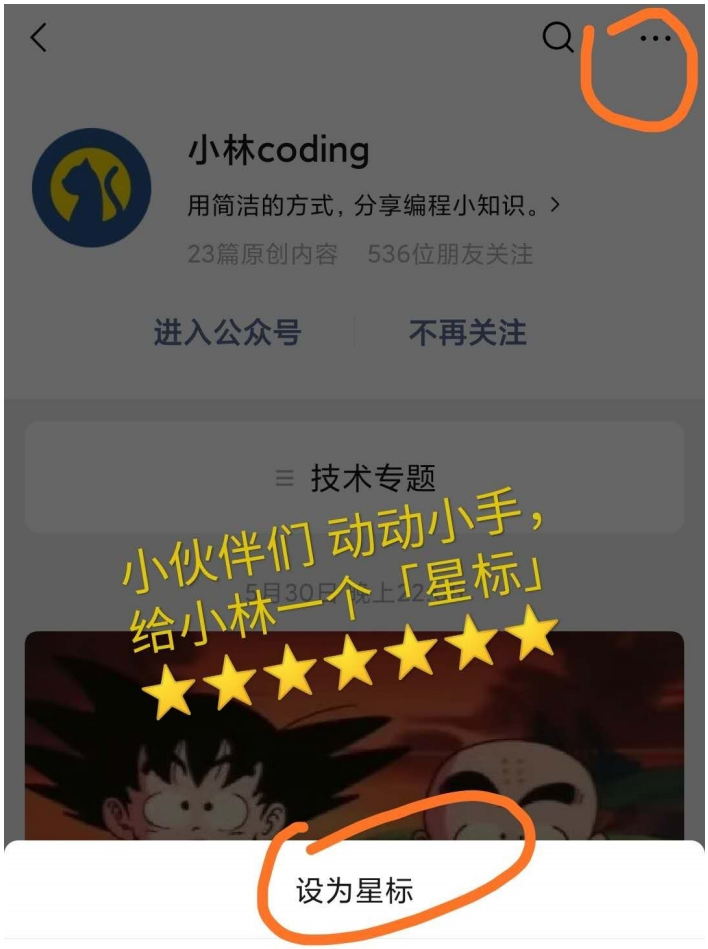
巨人的肩膀

[1] 竹下隆史.图解TCP/IP.人民邮电出版社.

[2] 刘超.趣谈网络协议.极客时间.

唠叨唠叨

小林是专为大家图解的工具人，Goodbye，我们下次见！

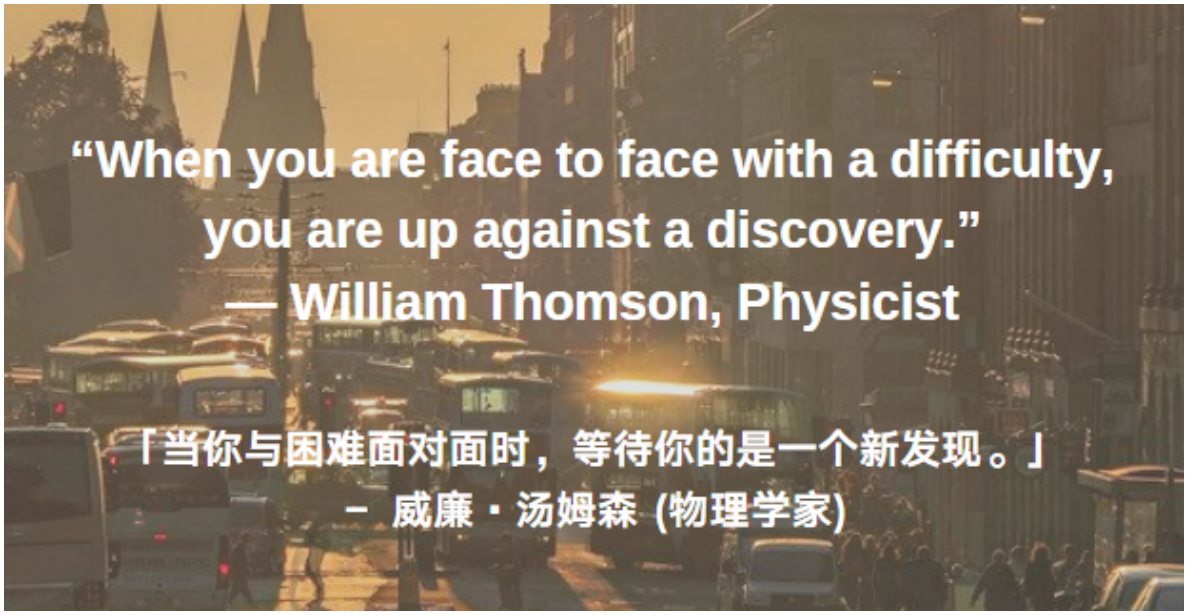


读者问答

> 读者问：“有个问题就是A的icmp到了B后，B为啥会自动给A一个回执0？这是操作系统的底层设计吗？”

你说的“回执0”是指 ICMP 类型为 0 吗？如果是的话，那么 B 收到 A 的回送请求（类型为8） ICMP 报文，B 主机操作系统协议栈发现是个回送请求 ICMP 报文，那么协议栈就会组装一个回送应答（类型为0）的 IMCP 回应给 A。

键入网址后，其间发生了什么？



前言

想必不少小伙伴面试过程中，会遇到「**当键入网址后，到网页显示，其间发生了什么**」的面试题。

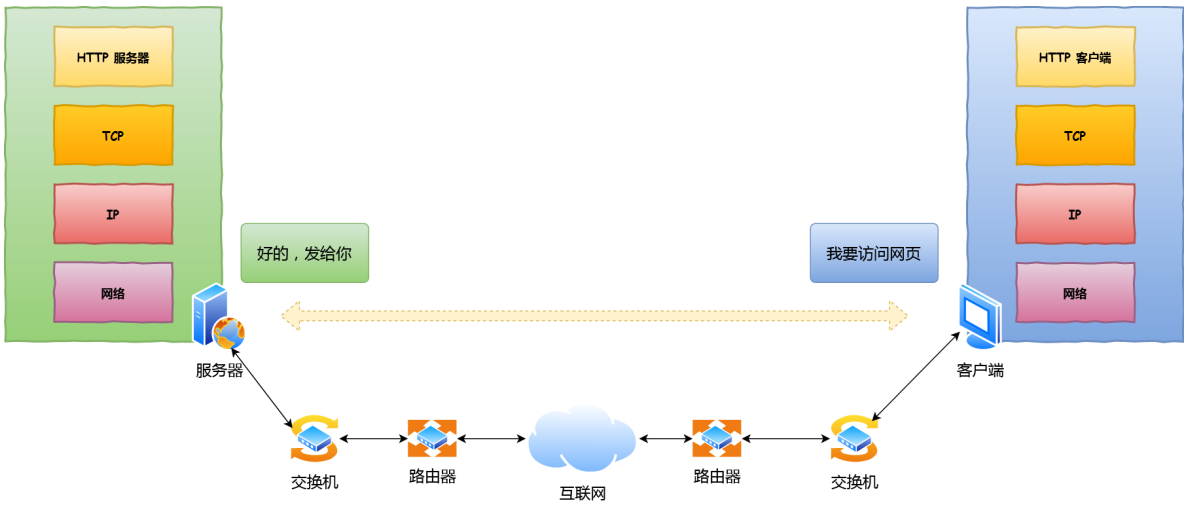
还别说，这真是挺常问的这题，前几天坐在我旁边的主管电话面试应聘者的时候，也问了这个问题。

这次，小林我带大家一起探究下，**一个数据包在网络中的心路历程**。

每个阶段都有数据包的「心路历程」，我们一起看看它说了什么？

正文

接下来以下图较简单的网络拓扑模型作为例子，探究探究其间发生了什么？

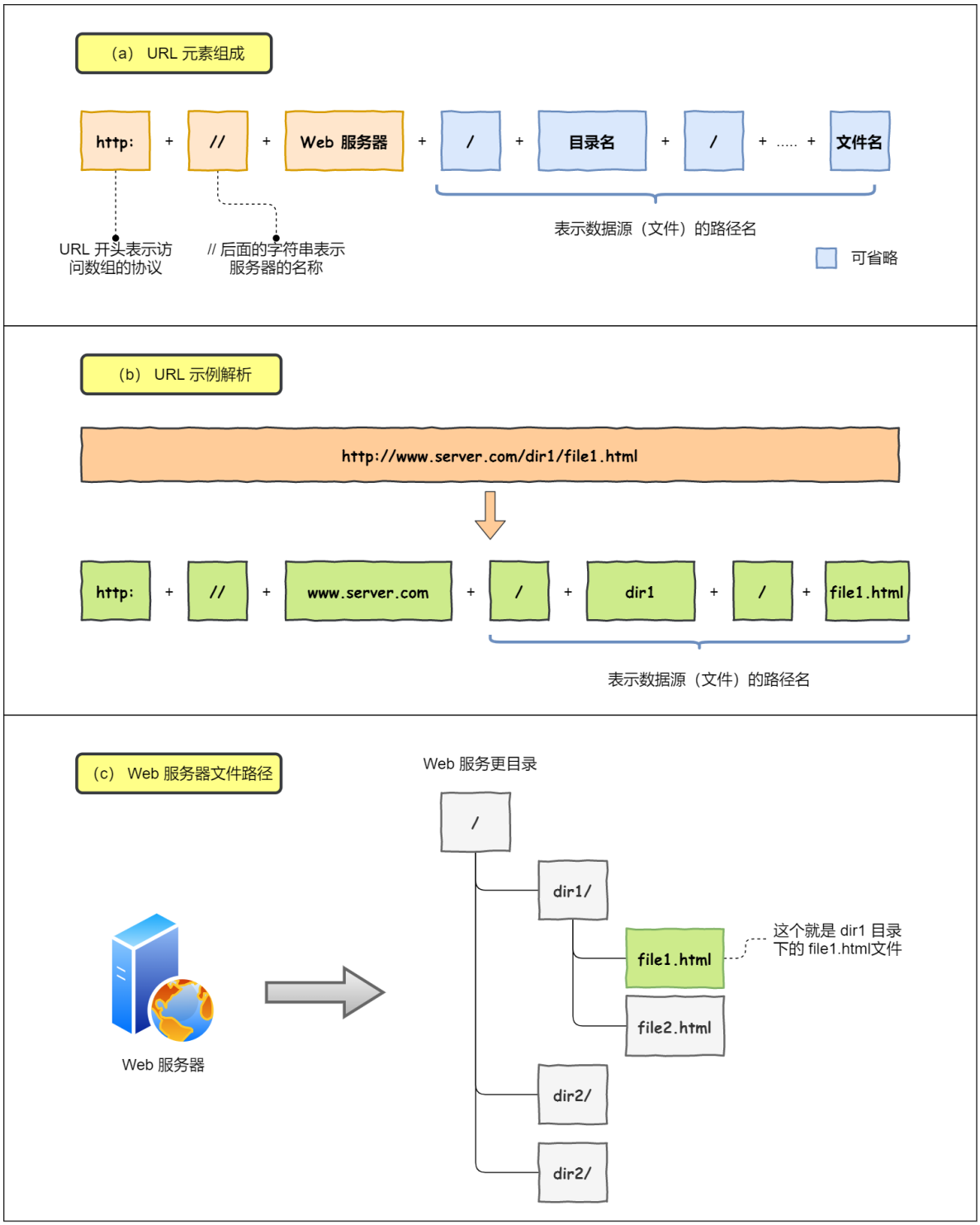


01 孤单小弟 —— HTTP

> 浏览器做的第一步工作是解析 URL

首先浏览器做的第一步工作就是要对 URL 进行解析，从而生成发送给 Web 服务器的请求信息。

让我们看看一条长长的 URL 里的各个元素的代表什么，见下图：



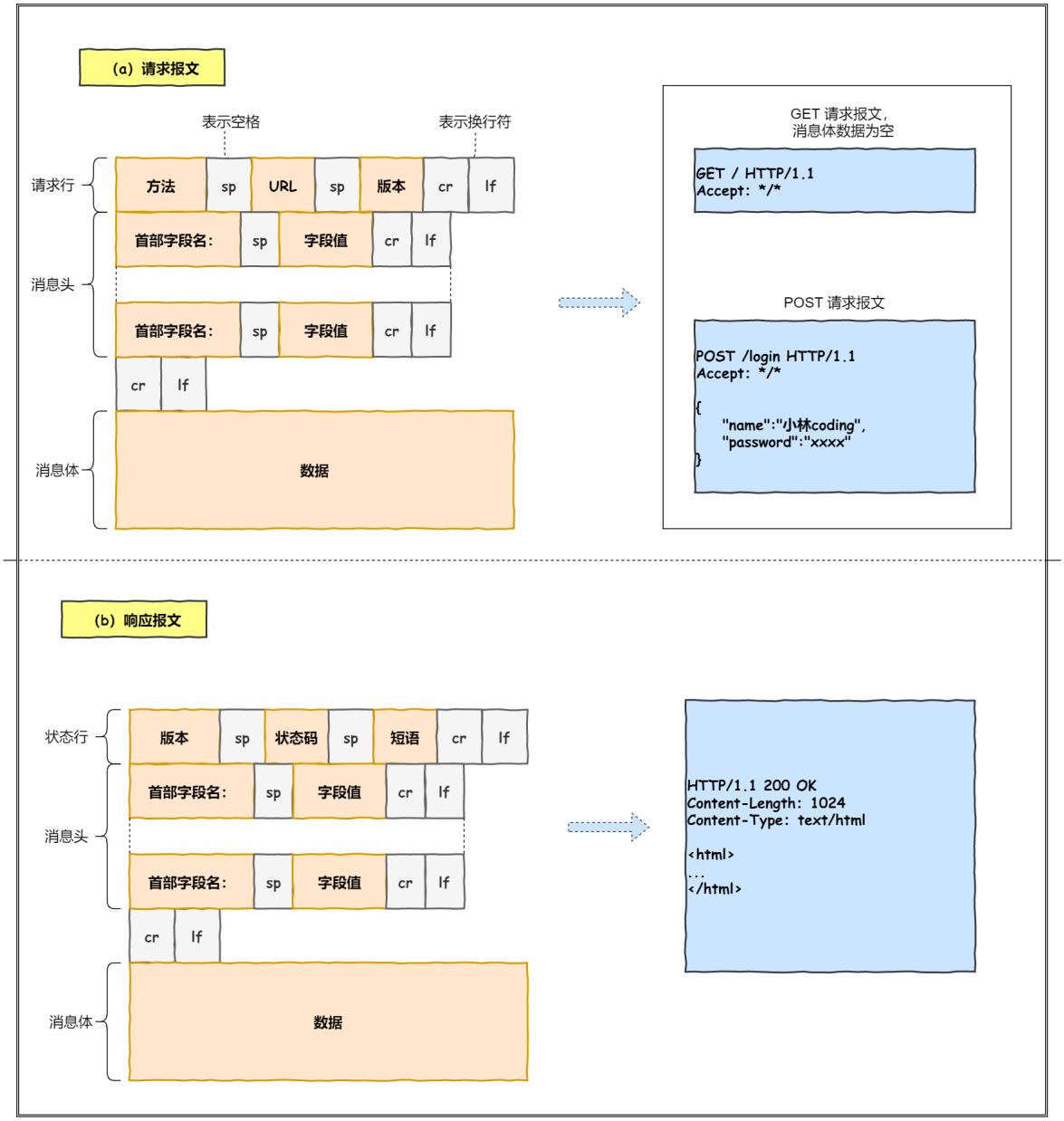
所以图中的长长的 URL 实际上是请求服务器里的文件资源。

> 要是上图中的蓝色部分 URL 元素都省略了，那应该是请求哪个文件呢？

当没有路径名时，就代表访问根目录下事先设置的**默认文件**，也就是 /index.html 或者 /default.html 这些文件，这样就不会发生混乱了。

生产 HTTP 请求信息

对 URL 进行解析之后，浏览器确定了 Web 服务器和文件名，接下来就是根据这些信息来生成 HTTP 请求消息了。



一个孤单 HTTP 数据包表示：“我这么一个小小的数据包，没亲没友，直接发到浩瀚的网络，谁会知道我呢？谁能载我一程呢？谁能保护我呢？我的目的地在哪呢？”。充满各种疑问的它，没有停滞不前，依然踏上了征途！

02 真实地址查询 —— DNS

通过浏览器解析 URL 并生成 HTTP 消息后，需要委托操作系统将消息发送给 Web 服务器。

但在发送之前，还有一项工作需要完成，那就是**查询服务器域名对应的 IP 地址**，因为委托操作系统发送消息时，必须提供通信对象的 IP 地址。

比如我们打电话的时候，必须要知道对方的电话号码，但由于电话号码难以记忆，所以通常我们会将对方电话号 + 姓名保存在通讯录里。

所以，有一种服务器就专门保存了 `Web` 服务器域名与 `IP` 的对应关系，它就是 `DNS` 服务器。

> 域名的层级关系

DNS 中的域名都是用**句点**来分隔的，比如 `www.server.com`，这里的句点代表了不同层次之间的**界限**。

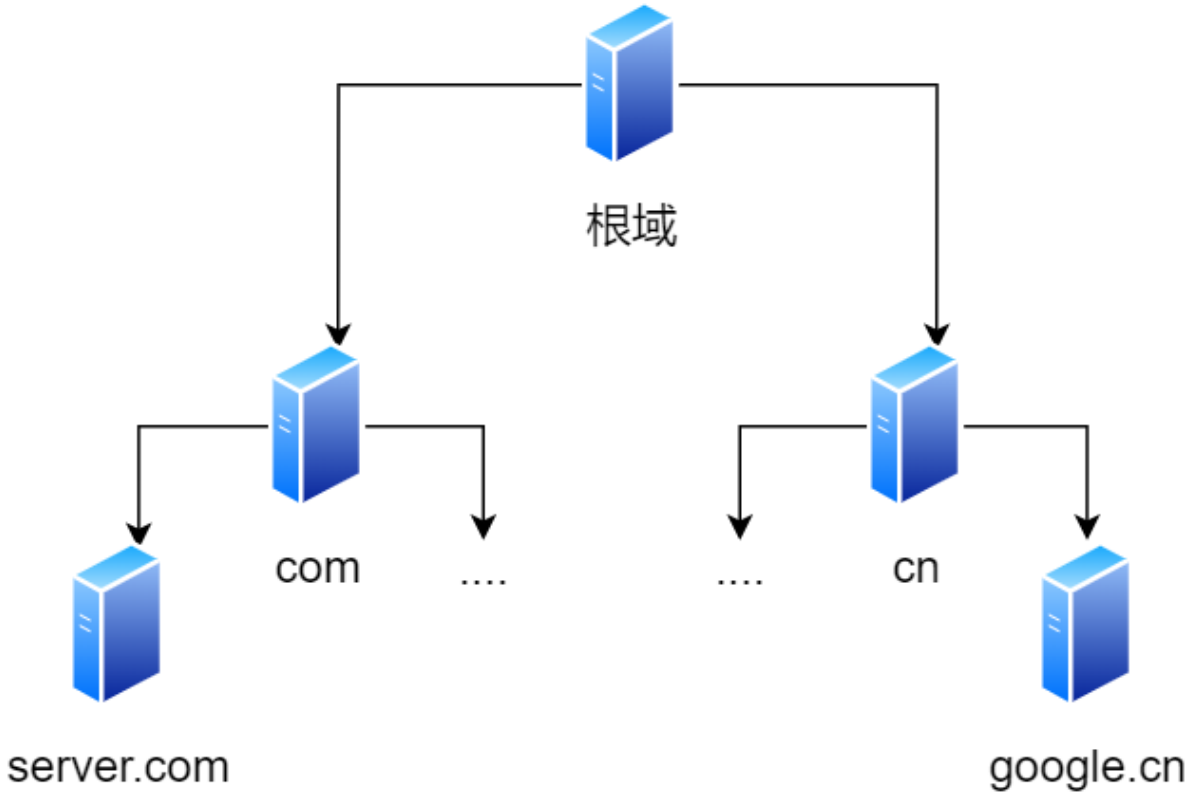
在域名中，**越靠右**的位置表示其层级**越高**。

毕竟域名是外国人发明，所以思维和中国人相反，比如说一个城市地点的时候，外国喜欢从小到大的方式顺序说起（如 XX 街道 XX 区 XX 市 XX 省），而中国则喜欢从大到小的顺序（如 XX 省 XX 市 XX 区 XX 街道）。

根域是在最顶层，它的下一层就是 com 顶级域，再下面是 server.com。

所以域名的层级关系类似一个树状结构：

- 根 DNS 服务器
- 顶级域 DNS 服务器（com）
- 权威 DNS 服务器（server.com）



根域的 DNS 服务器信息保存在互联网中所有的 DNS 服务器中。

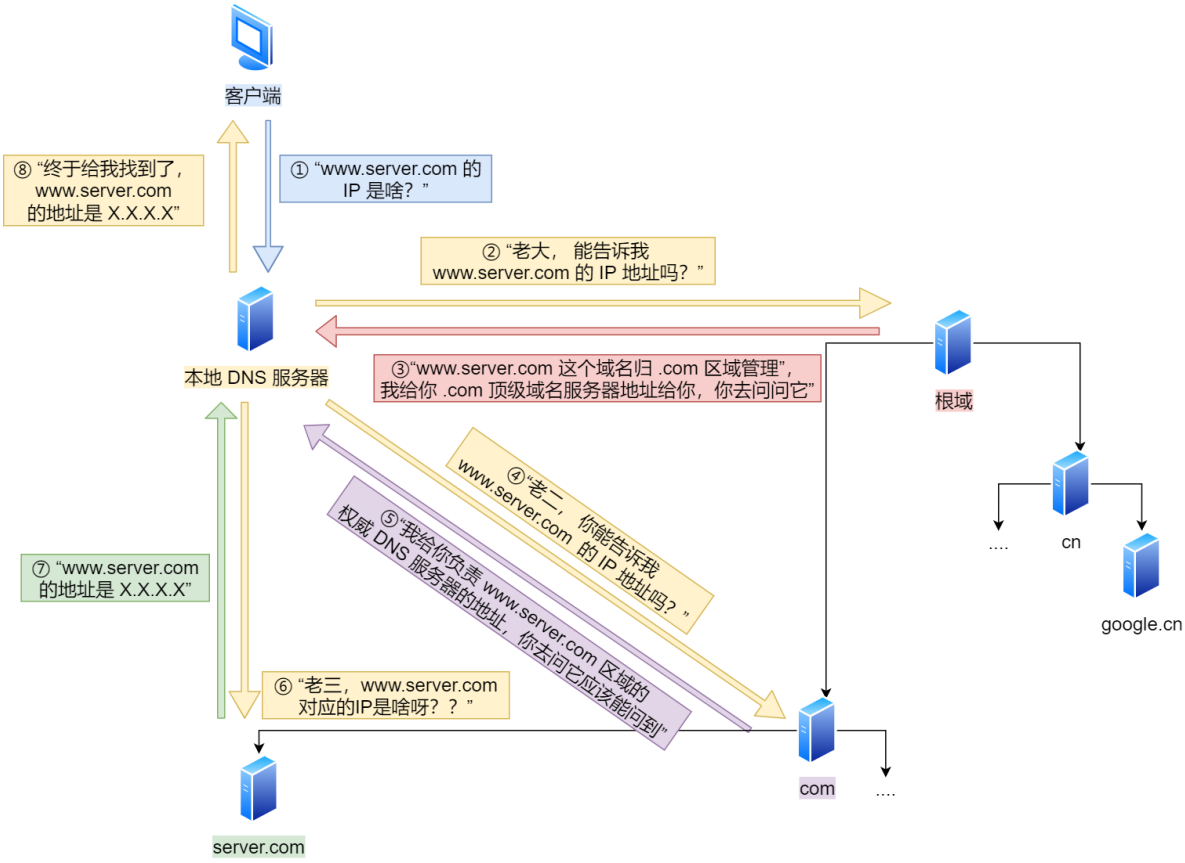
这样一来，任何 DNS 服务器就都可以找到并访问根域 DNS 服务器了。

因此，客户端只要能够找到任意一台 DNS 服务器，就可以通过它找到根域 DNS 服务器，然后再一路顺藤摸瓜找到位于下层的某台目标 DNS 服务器。

域名解析的工作流程

1. 客户端首先会发出一个 DNS 请求，问 www.server.com 的 IP 是啥，并发给本地 DNS 服务器（也就是客户端的 TCP/IP 设置中填写的 DNS 服务器地址）。
2. 本地域名服务器收到客户端的请求后，如果缓存里的表格能找到 www.server.com，则它直接返回 IP 地址。如果没有，本地 DNS 会去问它的根域名服务器：“老大， 能告诉我 www.server.com 的 IP 地址吗？”根域名服务器是最高层次的，它不直接用于域名解析，但能指明一条道路。
3. 根 DNS 收到来自本地 DNS 的请求后，发现后置是 .com，说：“www.server.com 这个域名归 .com 区域管理”，我给你 .com 顶级域名服务器地址给你，你去问问它吧。”
4. 本地 DNS 收到顶级域名服务器的地址后，发起请求问“老二， 你能告诉我 www.server.com 的 IP 地址吗？”
5. 顶级域名服务器说：“我给你负责 www.server.com 区域的权威 DNS 服务器的地址，你去问它应该能问到”。
6. 本地 DNS 于是转向问权威 DNS 服务器：“老三，www.server.com对应的IP是啥呀？” server.com 的权威 DNS 服务器，它是域名解析结果的原出处。为啥叫权威呢？就是我的域名我做主。
7. 权威 DNS 服务器查询后将对应的 IP 地址 X.X.X.X 告诉本地 DNS。
8. 本地 DNS 再将 IP 地址返回客户端，客户端和目标建立连接。

至此，我们完成了 DNS 的解析过程。现在总结一下，整个过程我画成了一个图。



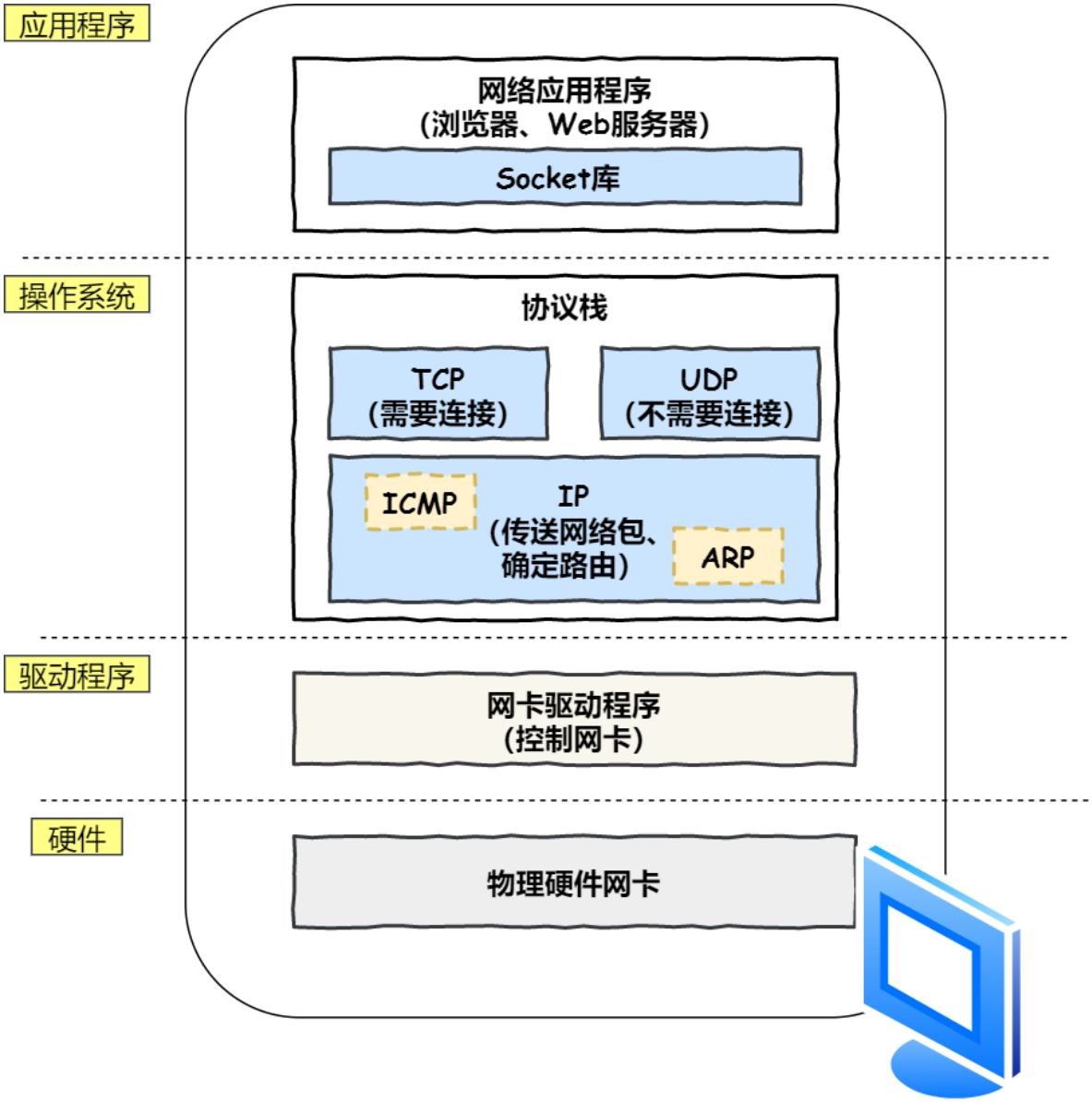
DNS 域名解析的过程蛮有意思的，整个过程就和我们日常生活中找人问路的过程类似，**只指路不带路**。

数据包表示：“DNS 老大哥厉害呀，找到了目的地了！我还是很迷茫呀，我要发出去，接下来我需要谁的帮助呢?”

03 指南好帮手 —— 协议栈

通过 DNS 获取到 IP 后，就可以把 HTTP 的传输工作交给操作系统中的**协议栈**。

协议栈的内部分为几个部分，分别承担不同的工作。上下关系是有一定的规则的，上面的部分会向下面的部分委托工作，下面的部分收到委托的工作并执行。



应用程序（浏览器）通过调用 Socket 库，来委托协议栈工作。协议栈的上半部分有两块，分别是负责收发数据的 TCP 和 UDP 协议，它们两会接受应用层的委托执行收发数据的操作。

协议栈的下面一半是用 IP 协议控制网络包收发操作，在互联网上传数据时，数据会被切分成一块块的网络包，而将网络包发送给对方的操作就是由 IP 负责的。

此外 IP 中还包括 `ICMP` 协议和 `ARP` 协议。

- `ICMP` 用于告知网络包传送过程中产生的错误以及各种控制信息。
- `ARP` 用于根据 IP 地址查询相应的以太网 MAC 地址。

IP 下面的网卡驱动程序负责控制网卡硬件，而最下面的网卡则负责完成实际的收发操作，也就是对网线中的信号执行发送和接收操作。

> 数据包看了这份指南表示：“原来我需要那么多大佬的协助啊，那我先去找找 TCP 大佬！”

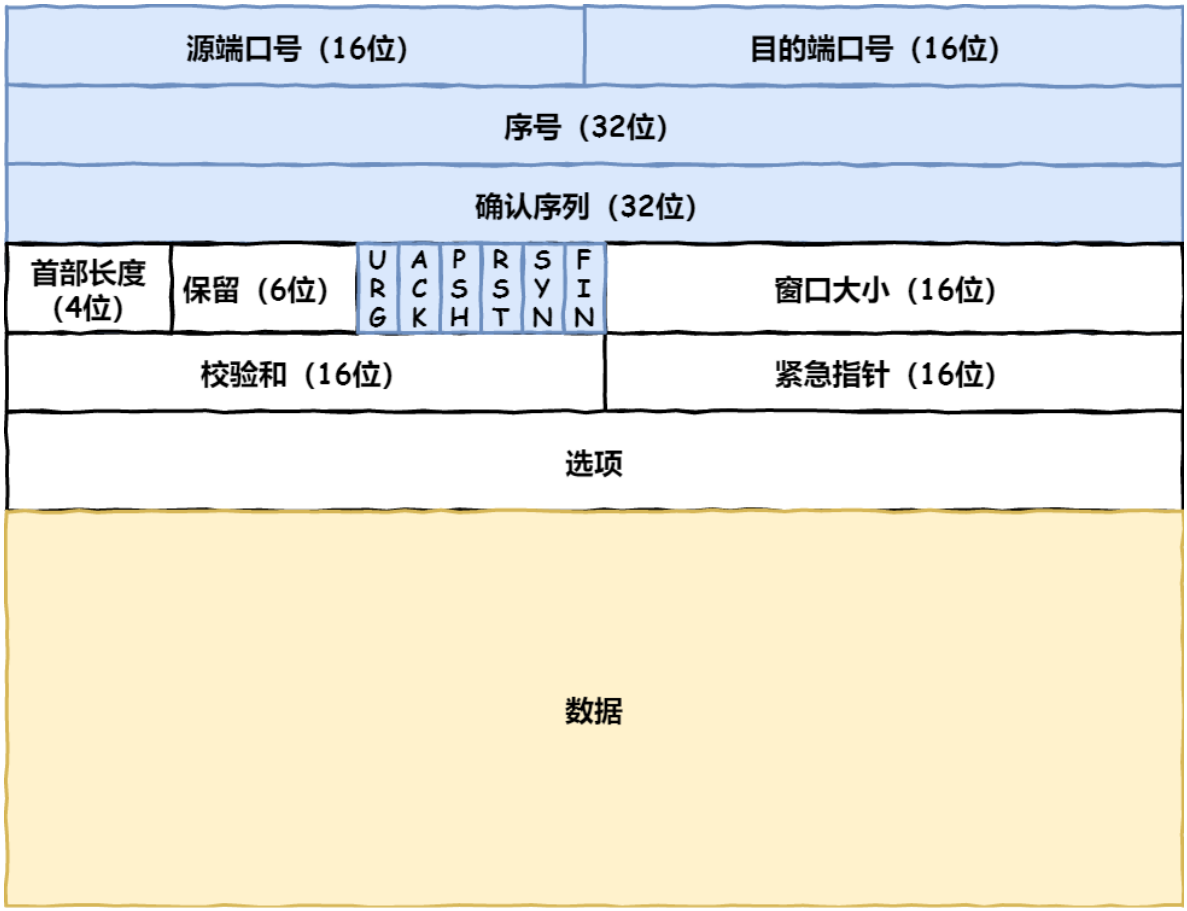
---

04 可靠传输 —— TCP

HTTP 是基于 TCP 协议传输的，所以在这我们先了解下 TCP 协议。

> TCP 包头格式

我们先看看 TCP 报文头部的格式：



首先，**源端口号**和**目标端口**号是不可少的，如果没有这两个端口号，数据就不知道应该发给哪个应用。

接下来有包的**序**号，这个是为了解决包乱序的问题。

还有应该有的是**确认号**，目的是确认发出去对方是否有收到。如果没有收到就应该重新发送，直到送达，这个是为了解决不丢包的问题。

接下来还有一些**状态位**。例如 `SYN` 是发起一个连接，`ACK` 是回复，`RST` 是重新连接，`FIN` 是结束连接等。TCP 是面向连接的，因而双方要维护连接的状态，这些带状态位的包的发送，会引起双方的状态变更。

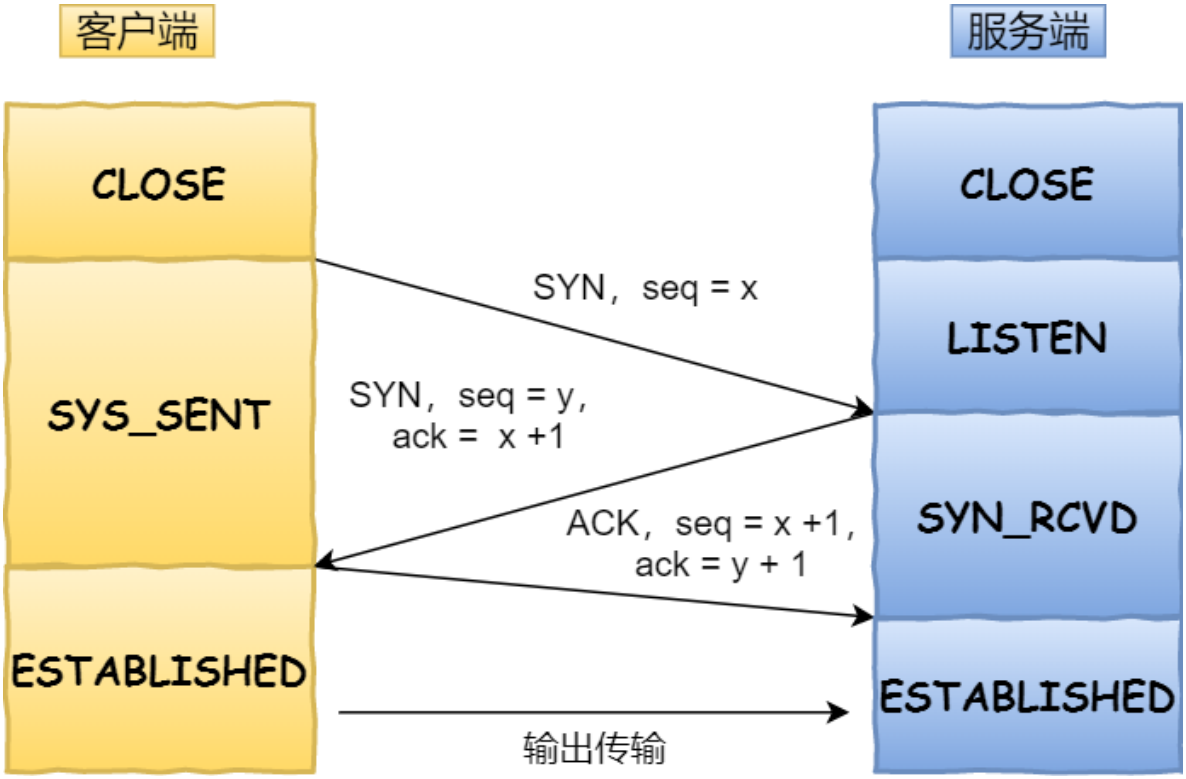
还有一个重要的就是**窗口大小**。TCP 要做**流量控制**，通信双方各声明一个窗口（缓存大小），标识自己当前能够的处理能力，别发送的太快，撑死我，也别发的太慢，饿死我。

除了做流量控制以外，TCP还会做**拥塞控制**，对于真正的通路堵车不堵车，它无能为力，唯一能做的就是控制自己，也即控制发送的速度。不能改变世界，就改变自己嘛。

> TCP 传输数据之前，要先三次握手建立连接

在 HTTP 传输数据之前，首先需要 TCP 建立连接，TCP 连接的建立，通常称为**三次握手**。

这个所谓的「连接」，只是双方计算机里维护一个状态机，在连接建立的过程中，双方的状态变化时序图就像这样。



- 一开始，客户端和服务端都处于 `CLOSED` 状态。先是服务端主动监听某个端口，处于 `LISTEN` 状态。
- 然后客户端主动发起连接 `SYN` ，之后处于 `SYN-SENT` 状态。
- 服务端收到发起的连接，返回 `SYN` ，并且 `ACK` 客户端的 `SYN` ，之后处于 `SYN-RCVD` 状态。
- 客户端收到服务端发送的 `SYN` 和 `ACK` 之后，发送 `ACK` 的 `ACK` ，之后处于 `ESTABLISHED` 状态，因为它一发一收成功了。
- 服务端收到 `ACK` 的 `ACK` 之后，处于 `ESTABLISHED` 状态，因为它也一发一收了。

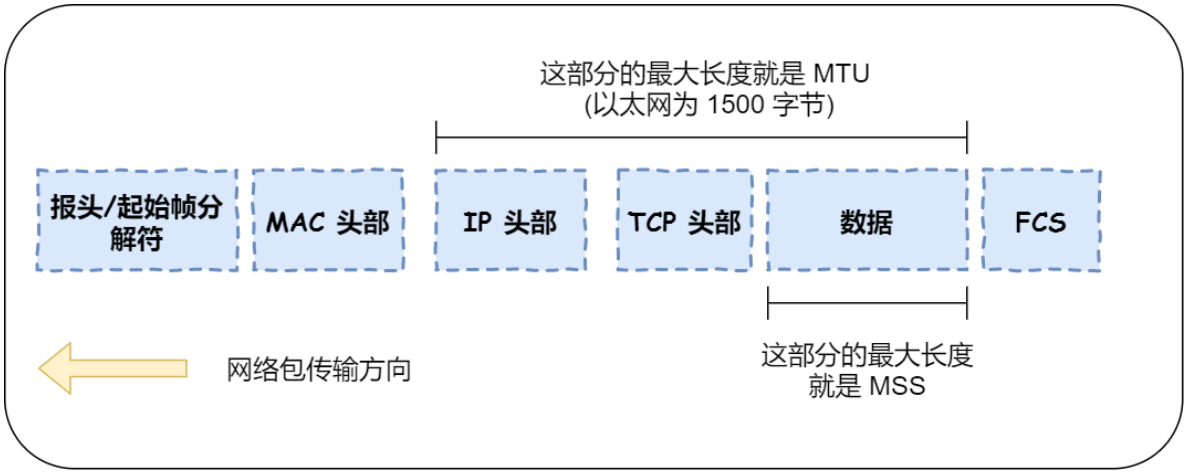
所以三次握手目的是**保证双方都有发送和接收的能力**。

> 如何查看 TCP 的连接状态?

TCP 的连接状态查看，在 Linux 可以通过 `netstat -napt` 命令查看。

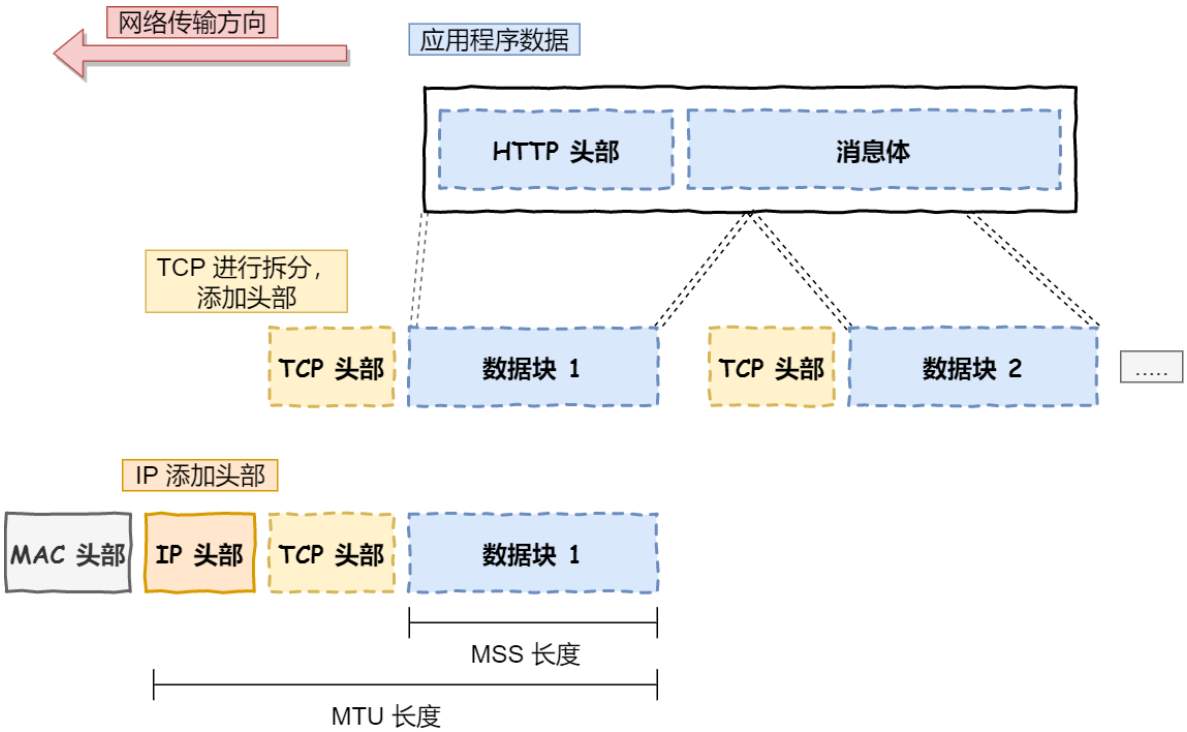
TCP 分割数据

如果 HTTP 请求消息比较长，超过了 MSS 的长度，这时 TCP 就需要把 HTTP 的数据拆解成一块块的数据发送，而不是一次性发送所有数据。



- MTU：一个网络包的最大长度，以太网中一般为 1500 字节。
- MSS：除去 IP 和 TCP 头部之后，一个网络包所能容纳的 TCP 数据的最大长度。

数据会被以 MSS 的长度为单位进行拆分，拆分出来的每一块数据都会被放进单独的网络包中。也就是在每个被拆分的数据加上 TCP 头信息，然后交给 IP 模块来发送数据。

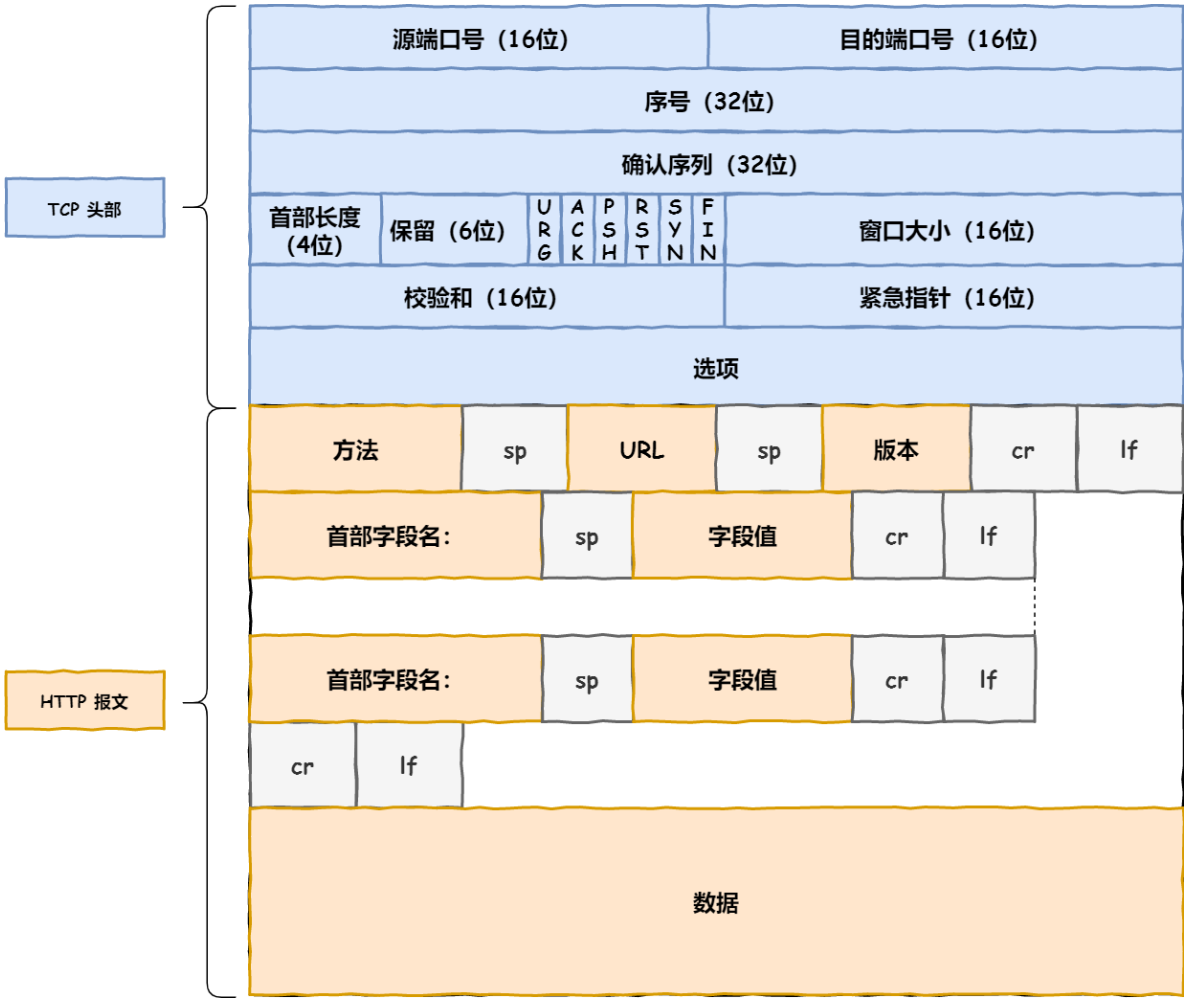


TCP 报文生成

TCP 协议里面会有两个端口，一个是浏览器监听的端口（通常是随机生成的），一个是 Web 服务器监听的端口（HTTP 默认端口号是 `80`，HTTPS 默认端口号是 `443`）。

在双方建立了连接后，TCP 报文中的数据部分就是存放 HTTP 头部 + 数据，组装好 TCP 报文之后，就需交给下面的网络层处理。

至此，网络包的报文如下图。



此时，遇上了 TCP 的 数据包激动表示：“太好了，碰到了可靠传输的 TCP 传输，它给我加上 TCP 头部，我不再孤单了，安全感十足啊！有大佬可以保护我的可靠送达！但我应该往哪走呢？”

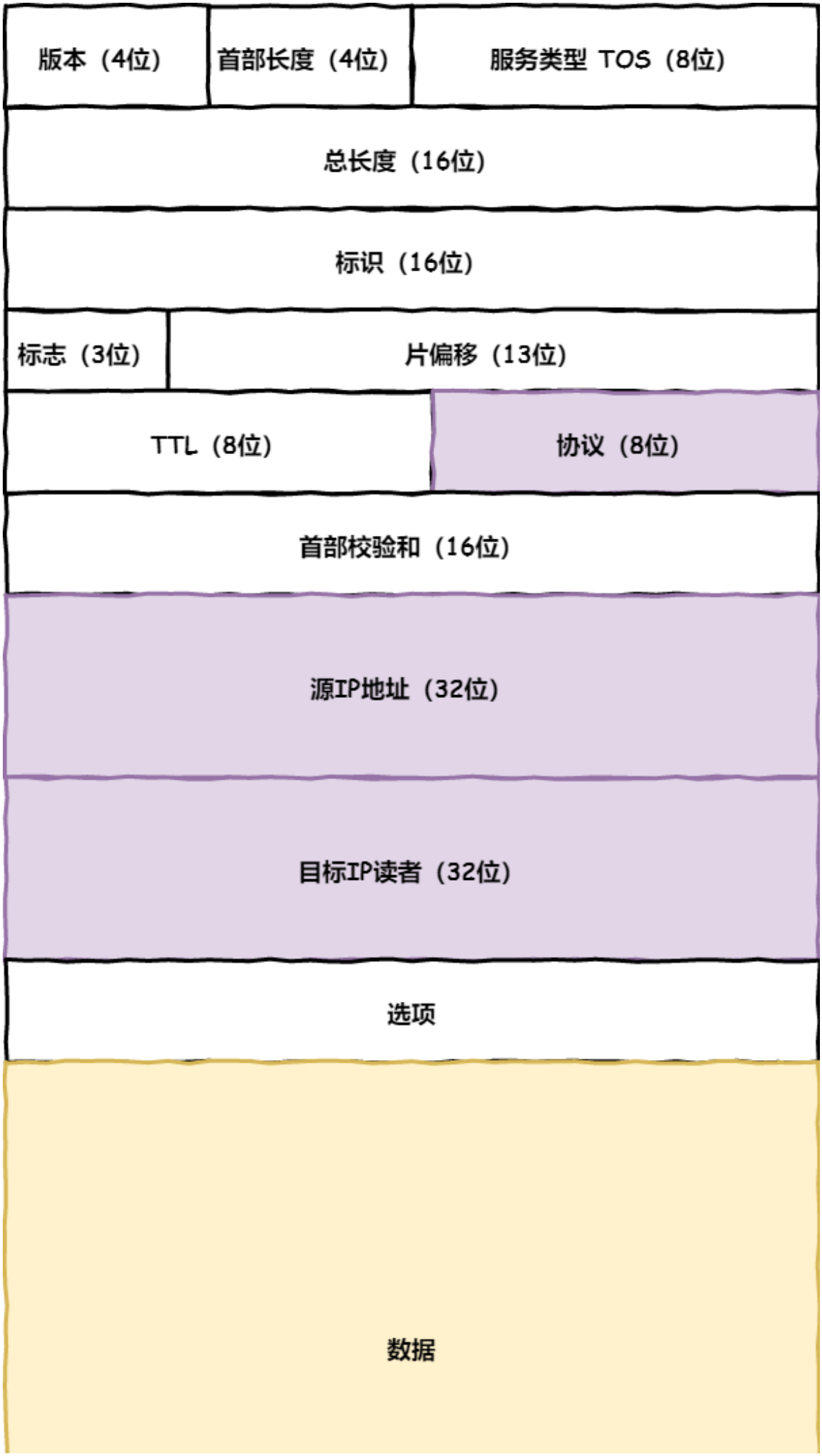
---

05 远程定位 —— IP

TCP 模块在执行连接、收发、断开等各阶段操作时，都需要委托 IP 模块将数据封装成**网络包**发送给通信对象。

IP 包头格式

我们先看看 IP 报文头部的格式：


版本（4位）
首部长度（4位）
服务类型 TOS（8位）
总长度（16位）
标识（16位）
标志（3位）
片偏移（13位）
TTL（8位）
协议（8位）
首部校验和（16位）
源IP地址（32位）
目标IP读者（32位）
选项
数据

在 IP 协议里面需要有**源地址 IP** 和 **目标地址 IP**：

- 源地址IP，即是客户端输出的 IP 地址；
- 目标地址，即通过 DNS 域名解析得到的 Web 服务器 IP。

因为 HTTP 是经过 TCP 传输的，所以在 IP 包头的**协议号**，要填写为 `06` （十六进制），表示协议为 TCP。

> 假设客户端有多个网卡，就会有多个 IP 地址，那 IP 头部的源地址应该选择哪个 IP 呢?

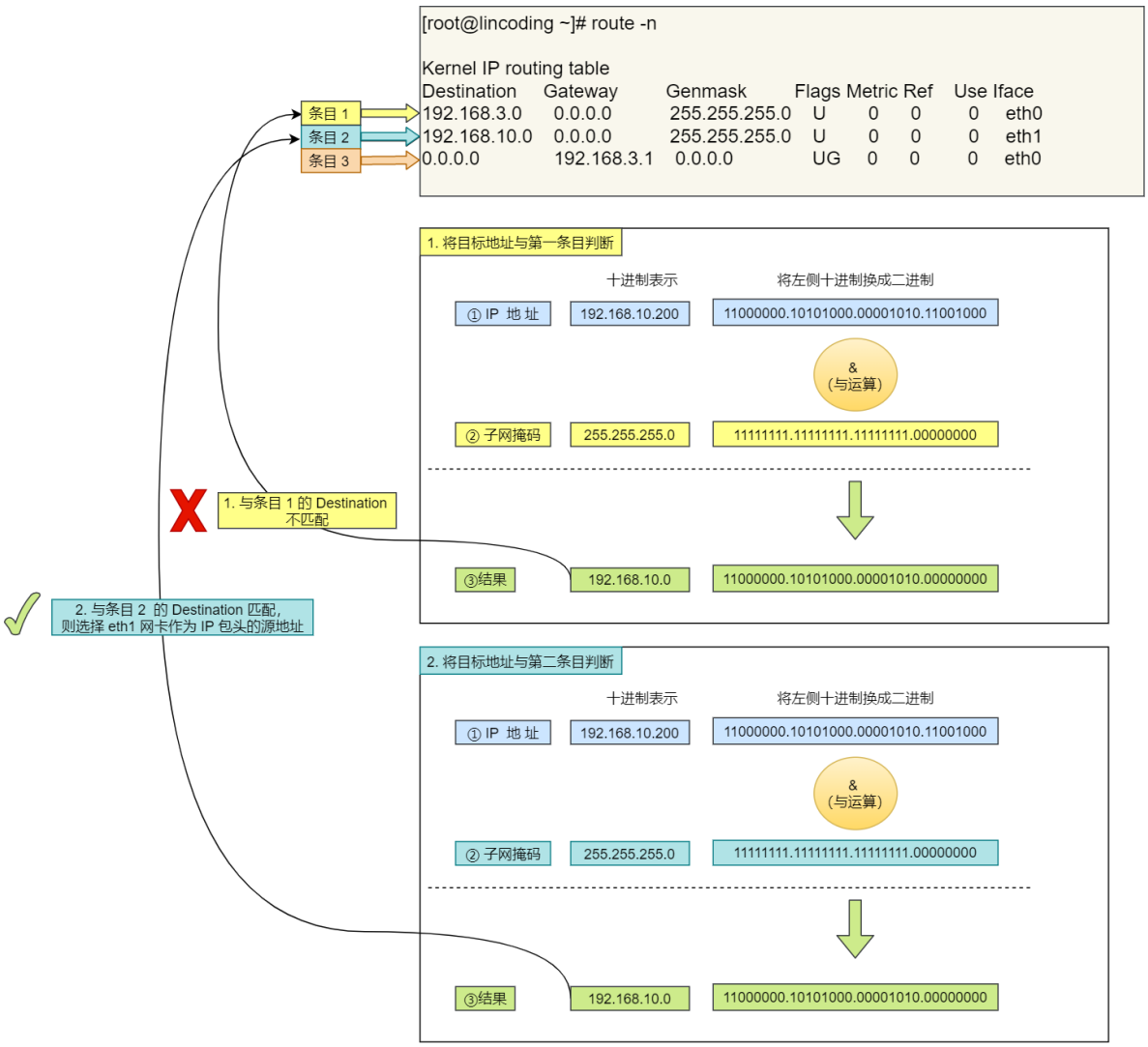
当存在多个网卡时，在填写源地址 IP 时，就需要判断到底应该填写哪个地址。这个判断相当于在多块网卡中判断应该使用哪个一块网卡来发送包。

这个时候就需要根据**路由表**规则，来判断哪一个网卡作为源地址 IP。

在 Linux 操作系统，我们可以使用 `route -n` 命令查看当前系统的路由表。

```
[root@lincoding ~]# route -n

Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.3.0     0.0.0.0         255.255.255.0   U     0      0        0 eth0
192.168.10.0    0.0.0.0         255.255.255.0   U     0      0        0 eth1
0.0.0.0         192.168.3.1     0.0.0.0         UG    0      0        0 eth0
```

举个例子，根据上面的路由表，我们假设 Web 服务器的目标地址是 `192.168.10.200` 。



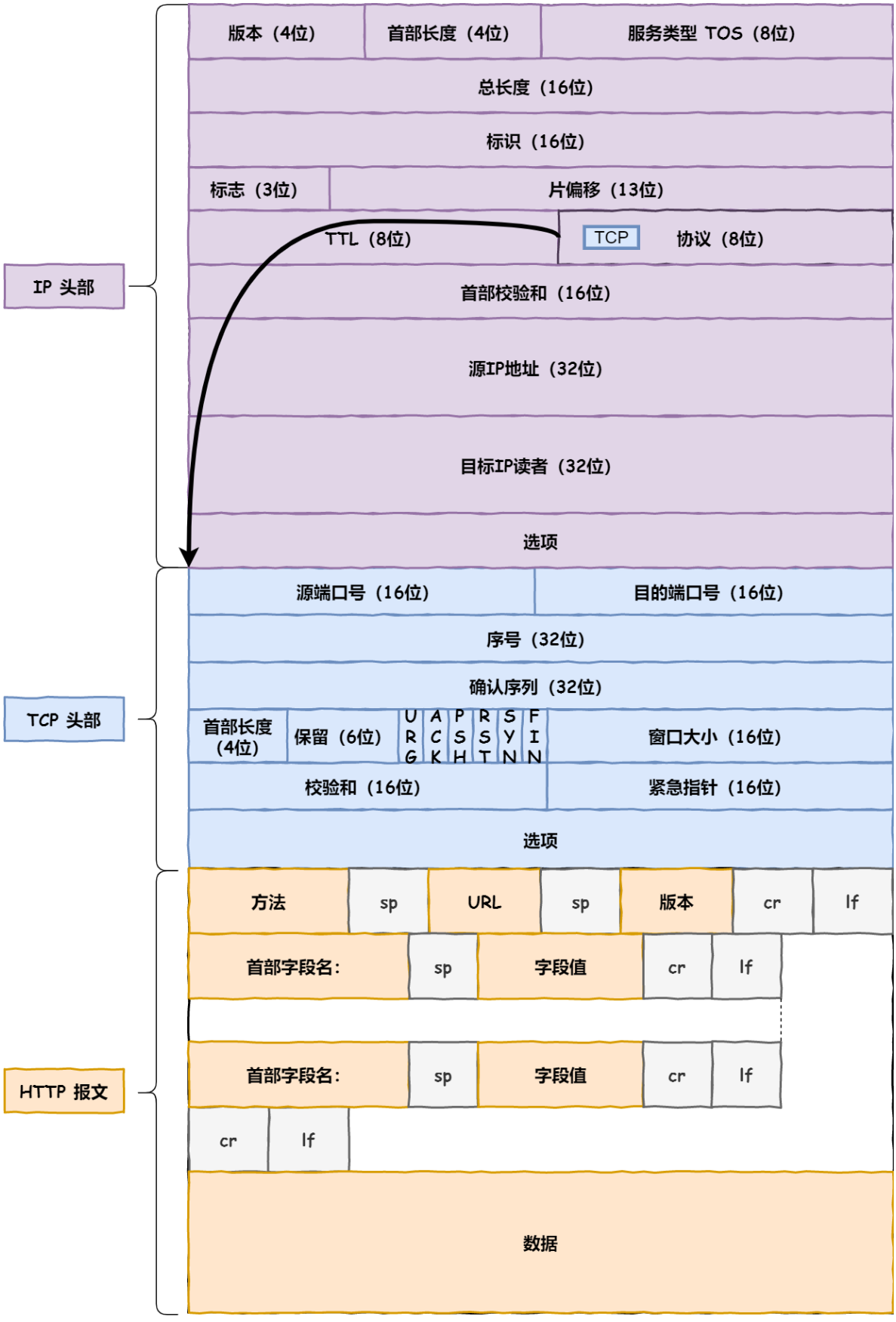
1. 首先先和第一条目的子网掩码（ Genmask ）进行 **与运算**，得到结果为 192.168.10.0 ，但是第一个条目的 Destination 是 192.168.3.0 ，两者不一致所以匹配失败。
2. 再与第二条目的子网掩码进行 **与运算**，得到的结果为 192.168.10.0 ，与第二条目的 Destination 192.168.10.0 匹配成功，所以将使用 eth1 网卡的 IP 地址作为 IP 包头的源地址。

那么假设 Web 服务器的目标地址是 10.100.20.100 ，那么依然依照上面的路由表规则判断，判断后的结果是和第三条目匹配。

第三条目比较特殊，它目标地址和子网掩码都是 0.0.0.0 ，这表示**默认网关**，如果其他所有条目都无法匹配，就会自动匹配这一行。并且后续就把包发给路由器， Gateway 即是路由器的 IP 地址。

> IP 报文生成

至此，网络包的报文如下图。



此时，加上了 IP 头部的数据包表示：“有 IP 大佬给我指路了，感谢 IP 层给我加上了 IP 包头，让我有了远程定位的能力！不会害怕在浩瀚的互联网迷茫了！可是目的地好远啊，我下一站应该去哪呢？”

06 两点传输 —— MAC

生成了 IP 头部之后，接下来网络包还需要在 IP 头部的前面加上 **MAC 头部**。

> MAC 包头格式

MAC 头部是以太网使用的头部，它包含了接收方和发送方的 MAC 地址等信息。

在 MAC 包头里需要**发送方 MAC 地址**和**接收方目标 MAC 地址**，用于**两点之间的传输**。

一般在 TCP/IP 通信里，MAC 包头的**协议类型**只使用：

- `0800` ： IP 协议
- `0806` ： ARP 协议

> MAC 发送方和接收方如何确认？

发送方的 MAC 地址获取就比较简单了，MAC 地址是在网卡生产时写入到 ROM 里的，只要将这个值读取出来写入到 MAC 头部就可以了。

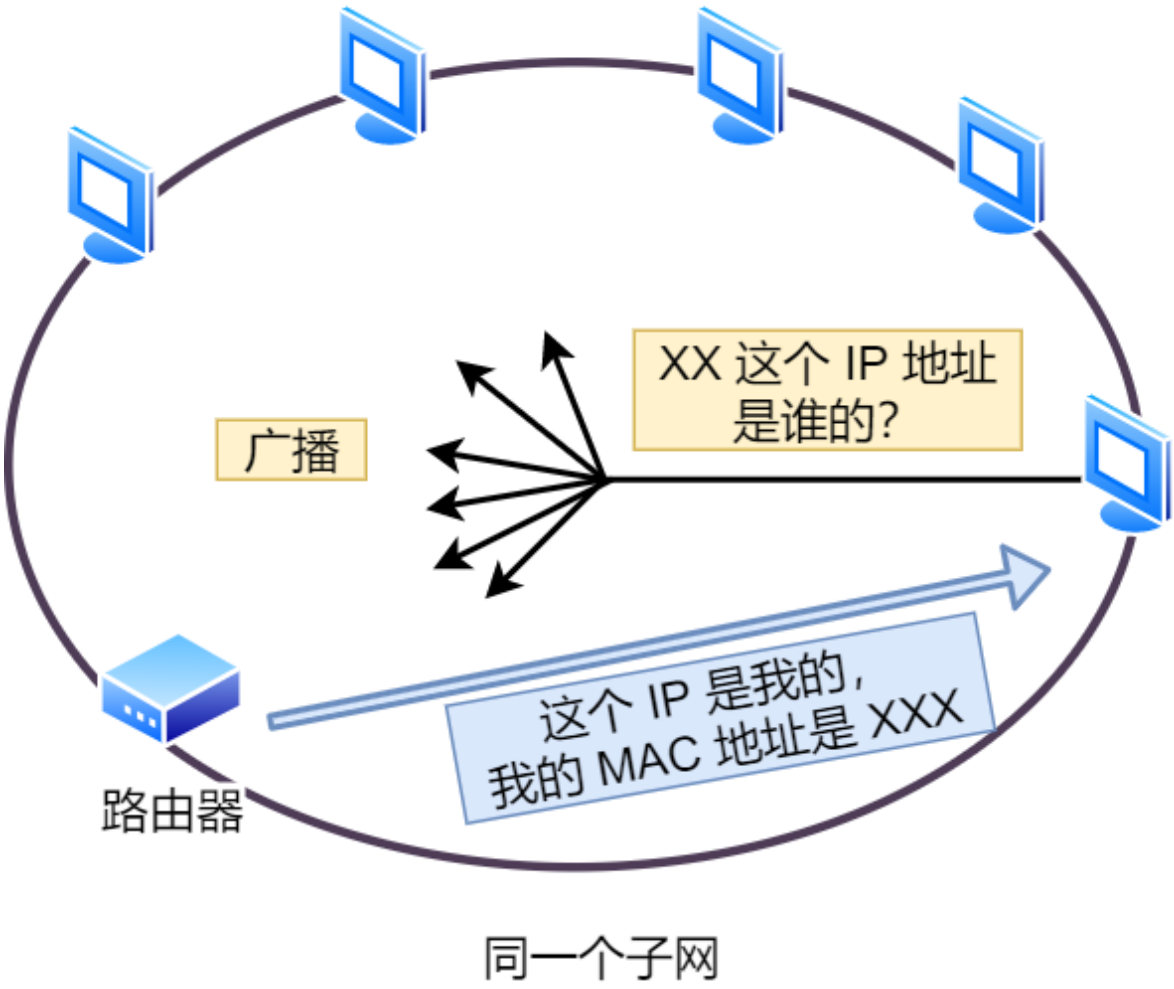
接收方的 MAC 地址就有点复杂了，只要告诉以太网对方的 MAC 的地址，以太网就会帮我们把包发送过去，那么很显然这里应该填写对方的 MAC 地址。

所以先得搞清楚应该把包发给谁，这个只要查一下**路由表**就知道了。在路由表中找到相匹配的条目，然后把包发给 `Gateway` 列中的 IP 地址就可以了。

> 既然知道要发给谁，按如何获取对方的 MAC 地址呢？

不知道对方 MAC 地址？不知道就喊呗。

此时就需要 `ARP` 协议帮我们找到路由器的 MAC 地址。



ARP 协议会在以太网中以**广播**的形式，对以太网所有的设备喊出：“这个 IP 地址是谁的？请把你的 MAC 地址告诉我”。

然后就会有人回答：“这个 IP 地址是我的，我的 MAC 地址是 XXXX”。

如果对方和自己处于同一个子网中，那么通过上面的操作就可以得到对方的 MAC 地址。然后，我们将这个 MAC 地址写入 MAC 头部，MAC 头部就完成了。

> 好像每次都要广播获取，这不是很麻烦吗？

放心，在后续操作系统会把本次查询结果放到一块叫做 **ARP 缓存**的内存空间留着以后用，不过缓存的时间就几分钟。

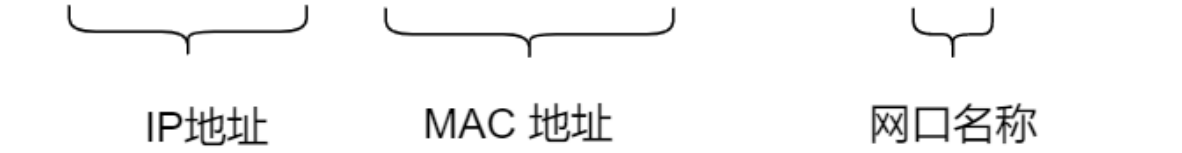
也就是说，在发包时：

- 先查询 ARP 缓存，如果其中已经保存了对方的 MAC 地址，就不需要发送 ARP 查询，直接使用 ARP 缓存中的地址。
- 而当 ARP 缓存中不存在对方 MAC 地址时，则发送 ARP 广播查询。

> 查看 ARP 缓存内容

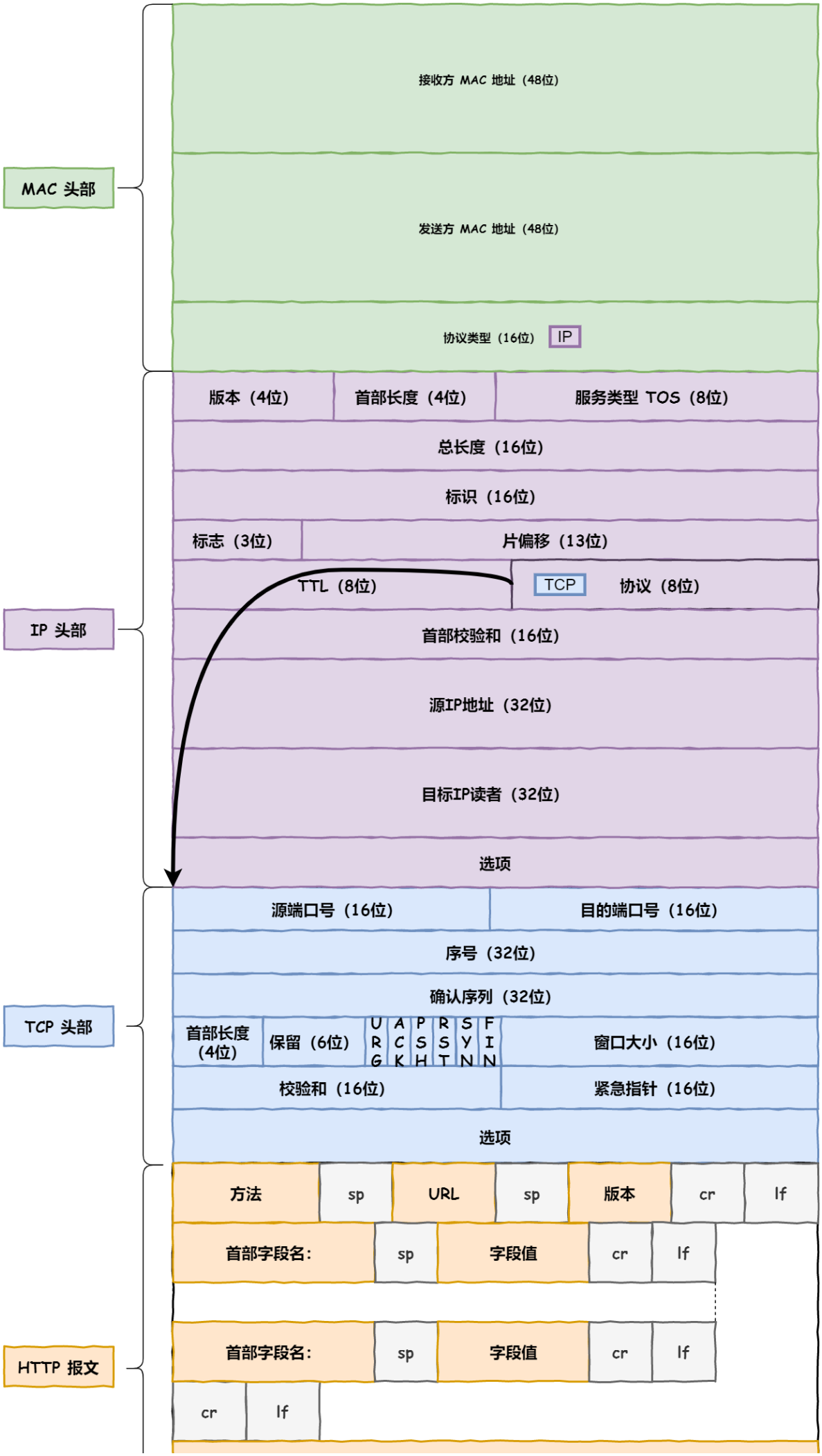
在 Linux 系统中，我们可以使用 `arp -a` 命令来查看 ARP 缓存的内容。

```
[root@lincoding ~]# arp -a
? (192.168.3.20) at f0:76:1c:58:f4:bc [ether] on eth2
```



> MAC 报文生成

至此，网络包的报文如下图。


MAC 头部
接收方 MAC 地址（48位）
发送方 MAC 地址（48位）
协议类型（16位）
IP
IP 头部
版本（4位）
首部长度（4位）
服务类型 TOS（8位）
总长度（16位）
标识（16位）
标志（3位）
片偏移（13位）
TTL（8位）
TCP
协议（8位）
首部校验和（16位）
源IP地址（32位）
目标IP读者（32位）
选项
TCP 头部
源端口号（16位）
目的端口号（16位）
序号（32位）
确认序列（32位）
首部长度（4位）
保留（6位）
URG
ACK
PSH
RST
SYN
FIN
窗口大小（16位）
校验和（16位）
紧急指针（16位）
选项
HTTP 报文
方法
sp
URL
sp
版本
cr
lf
首部字段名:
sp
字段值
cr
lf
首部字段名:
sp
字段值
cr
lf
cr
lf

> 此时，加上了 MAC 头部的数据包万分感谢，说道：“感谢 MAC 大佬，我知道我下一步要去哪了！我现在有很多头部兄弟，相信我可以到达最终的目的地！”。 带着众多头部兄弟的数据包，终于准备要出门了。

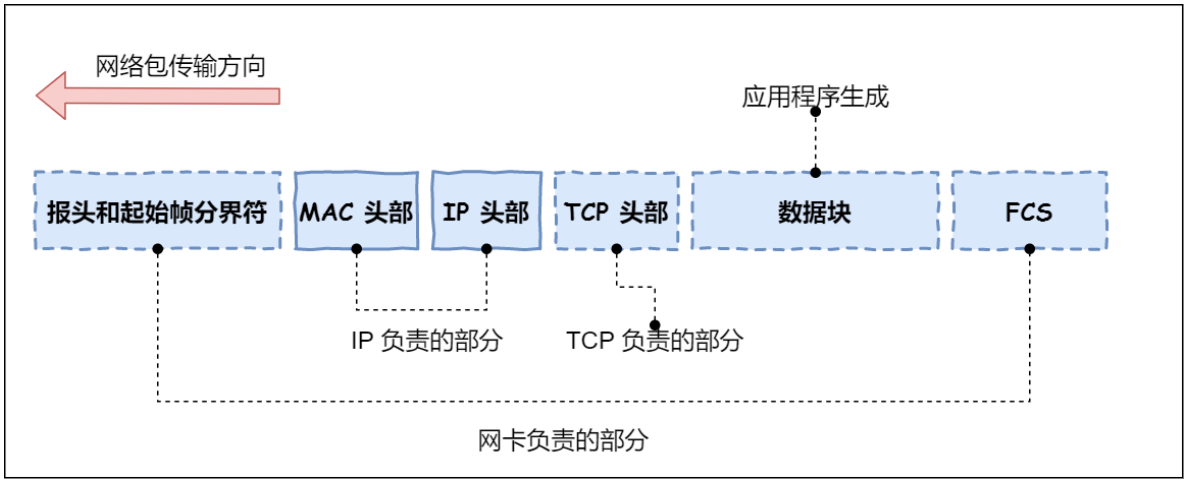
---

07 出口 —— 网卡

IP 生成的网络包只是存放在内存中的一串二进制数字信息，没有办法直接发送给对方。因此，我们需要将**数字信息转换为电信号**，才能在网线上传输，也就是说，这才是真正的数据发送过程。

负责执行这一操作的是**网卡**，要控制网卡还需要靠**网卡驱动程序**。

网卡驱动从 IP 模块获取到包之后，会将其**复制**到网卡内的缓存区中，接着会在其**开头加上报头和起始帧分界符，在末尾加上用于检测错误的帧校验序列**。



- 起始帧分界符是一个用来表示包起始位置的标记
- 末尾的 FCS （帧校验序列）用来检查包传输过程是否有损坏

最后网卡会将包转为电信号，通过网线发送出去。

> 唉，真是不容易，发一个包，真是历经千辛万苦。致此，一个带有许多头部的数据终于踏上寻找目的地的征途了！

---

08 送别者 —— 交换机

下面来看一下包是如何通过交换机的。交换机的设计是将网络包**原样**转发到目的地。交换机工作在 MAC 层，也称为**二层网络设备**。

> 交换机的包接收操作

首先，电信号到达网线接口，交换机里的模块进行接收，接下来交换机里的模块将电信号转换为数字信号。

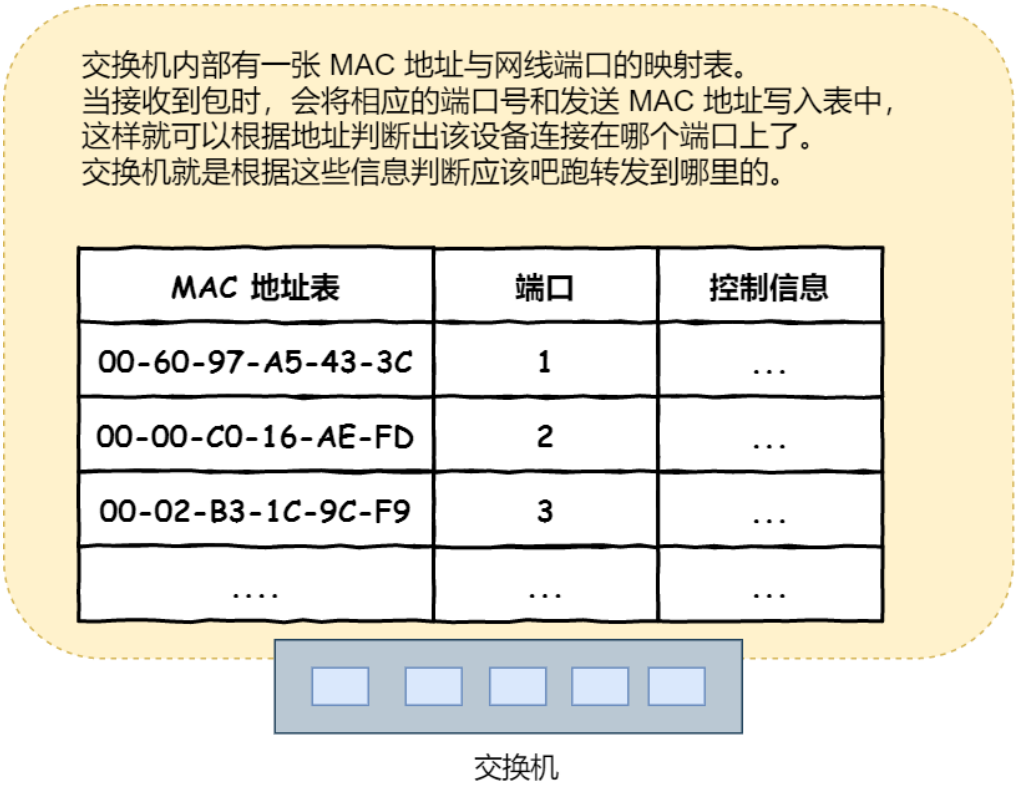
然后通过包末尾的 `FCS` 校验错误，如果没问题则放到缓冲区。这部分操作基本和计算机的网卡相同，但交换机的工作方式和网卡不同。

计算机的网卡本身具有 MAC 地址，并通过核对收到的包的接收方 MAC 地址判断是不是发给自己的，如果不是发给自己的则丢弃；相对地，交换机的端口不核对接收方 MAC 地址，而是直接接收所有的包并存放到缓冲区中。因此，和网卡不同，**交换机的端口不具有 MAC 地址**。

将包存入缓冲区后，接下来需要查询一下这个包的接收方 MAC 地址是否已经在 MAC 地址表中有记录了。

交换机的 MAC 地址表主要包含两个信息：

- 一个是设备的 MAC 地址，
- 另一个是该设备连接在交换机的哪个端口上。



交换机

举个例子，如果收到的包的接收方 MAC 地址为 `00-02-B3-1C-9C-F9`，则与图中表中的第 3 行匹配，根据端口列的信息，可知这个地址位于 `3` 号端口上，然后就可以通过交换电路将包发送到相应的端口了。

所以，**交换机根据 MAC 地址表查找 MAC 地址，然后将信号发送到相应的端口**。

> 当 MAC 地址表找不到指定的 MAC 地址会怎么样?

地址表中找不到指定的 MAC 地址。这可能是因为具有该地址的设备还没有向交换机发送过包，或者这个设备一段时间没有工作导致地址被从地址表中删除了。

这种情况下，交换机无法判断应该把包转发到哪个端口，只能将包转发到除了源端口之外的所有端口上，无论该设备连接在哪个端口上都能收到这个包。

这样做不会产生什么问题，因为以太网的设计本来就是将包发送到整个网络的，然后**只有相应的接收者才接收包，而其他设备则会忽略这个包**。

有人会说：“这样做会发送多余的包，会不会造成网络拥塞呢？”

其实完全不用过于担心，因为发送了包之后目标设备会作出响应，只要返回了响应包，交换机就可以将它的地址写入 MAC 地址表，下次也就不需要把包发到所有端口了。

局域网中每秒可以传输上千个包，多出一两个包并无大碍。

此外，如果接收方 MAC 地址是一个**广播地址**，那么交换机会将包发送到除源端口之外的所有端口。

以下两个属于广播地址：

- MAC 地址中的 `FF:FF:FF:FF:FF:FF`
- IP 地址中的 `255.255.255.255`

> 数据包通过交换机转发抵达了路由器，准备要离开土生土长的子网了。此时，数据包和交换机离别时说道：“感谢交换机兄弟，帮我转发到出境的大门，我要出远门啦！”

---

09 出境大门 —— 路由器

> 路由器与交换机的区别

网络包经过交换机之后，现在到达了**路由器**，并在此被转发到下一个路由器或目标设备。

这一步转发的工作原理和交换机类似，也是通过查表判断包转发的目标。

不过在具体的操作过程上，路由器和交换机是有区别的。

- 因为**路由器**是基于 IP 设计的，俗称**三层**网络设备，路由器的各个端口都具有 MAC 地址和 IP 地址；
- 而**交换机**是基于以太网设计的，俗称**二层**网络设备，交换机的端口不具有 MAC 地址。

> 路由器基本原理

路由器的端口具有 MAC 地址，因此它就能够成为以太网的发送方和接收方；同时还具有 IP 地址，从这个意义上来说，它和计算机的网卡是一样的。

当转发包时，首先路由器端口会接收发给自己的以太网包，然后**路由表**查询转发目标，再由相应的端口作为发送方将以太网包发送出去。

> 路由器的包接收操作

首先，电信号到达网线接口部分，路由器中的模块会将电信号转成数字信号，然后通过包末尾的 FCS 进行错误校验。

如果没问题则检查 MAC 头部中的**接收方 MAC 地址**，看看是不是发给自己的包，如果是就放到接收缓冲区中，否则就丢弃这个包。

总的来说，路由器的端口都具有 MAC 地址，只接收与自身地址匹配的包，遇到不匹配的包则直接丢弃。

> 查询路由表确定输出端口

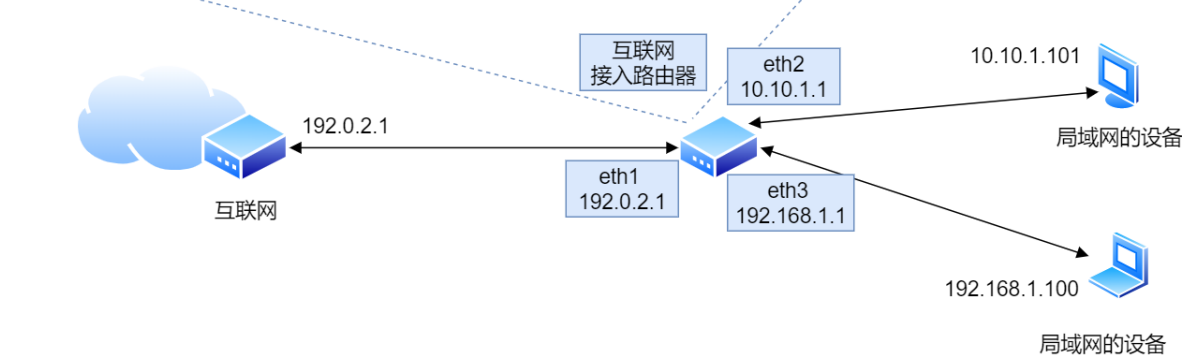
完成包接收操作之后，路由器就会**去掉**包开头的 MAC 头部。

MAC 头部的作用就是将包送达路由器，其中的接收方 MAC 地址就是路由器端口的 MAC 地址。因此，当包到达路由器之后，MAC 头部的任务就完成了，于是 MAC 头部就会**被丢弃**。

接下来，路由器会根据 MAC 头部后方的 IP 头部中的内容进行包的转发操作。

转发操作分为几个阶段，首先是查询**路由表**判断转发目标。

目标地址 (Destination)	子网掩码 (Netmask)	网关 (Gateway)	接口 (Interface)	跃点数 (Metric)
10.10.1.0	255.255.255.0	——	eth2	1
192.168.1.0	255.255.255.0	——	eth3	1
0.0.0.0	0.0.0.0	192.0.2.1	eth1	1



具体的工作流程根据上图，举个例子。

假设地址为 10.10.1.101 的计算机要向地址为 192.168.1.100 的服务器发送一个包，这个包先到达图中的路由器。

判断转发目标的第一步，就是根据包的接收方 IP 地址查询路由表中的目标地址栏，以找到相匹配的记录。

路由匹配和前面讲的一样，每个条目的子网掩码和 `192.168.1.100` IP 做 **& 与运算**后，得到的结果与对应条目的目标地址进行匹配，如果匹配就会作为候选转发目标，如果不匹配就继续与下个条目进行路由匹配。

如第二条目的子网掩码 `255.255.255.0` 与 `192.168.1.100` IP 做 **& 与运算**后，得到结果是 `192.168.1.0` ，这与第二条目的目标地址 `192.168.1.0` 匹配，该第二条目记录就会被作为转发目标。

实在找不到匹配路由时，就会选择**默认路由**，路由表中子网掩码为 `0.0.0.0` 的记录表示「默认路由」。

> 路由器的发送操作

接下来就会进入包的**发送操作**。

首先，我们需要根据**路由表的网关列**判断对方的地址。

- 如果网关是一个 IP 地址，则这个IP 地址就是我们要转发到的目标地址，**还未抵达终点**，还需继续需要路由器转发。
- 如果网关为空，则 IP 头部中的接收方 IP 地址就是要转发到的目标地址，也是就终于找到 IP 包头里的目标地址了，说明**已抵达终点**。

知道对方的 IP 地址之后，接下来需要通过 `ARP` 协议根据 IP 地址查询 MAC 地址，并将查询的结果作为接收方 MAC 地址。

路由器也有 ARP 缓存，因此首先会在 ARP 缓存中查询，如果找不到则发送 ARP 查询请求。

接下来是发送方 MAC 地址字段，这里填写输出端口的 MAC 地址。还有一个以太类型字段，填写 `0080` （十六进制）表示 IP 协议。

网络包完成后，接下来会将其转换成电信号并通过端口发送出去。这一步的工作过程和计算机也是相同的。

发送出去的网络包会通过**交换机**到达下一个路由器。由于接收方 MAC 地址就是下一个路由器的地址，所以交换机会根据这一地址将包传输到下一个路由器。

接下来，下一个路由器会将包转发给再下一个路由器，经过层层转发之后，网络包就到达了最终的目的地。

不知你发现了没有，在网络包传输的过程中，**源 IP 和目标 IP 始终是不会变的，一直变化的是 MAC 地址**，因为需要 MAC 地址在以太网内进行**两个设备**之间的包传输。

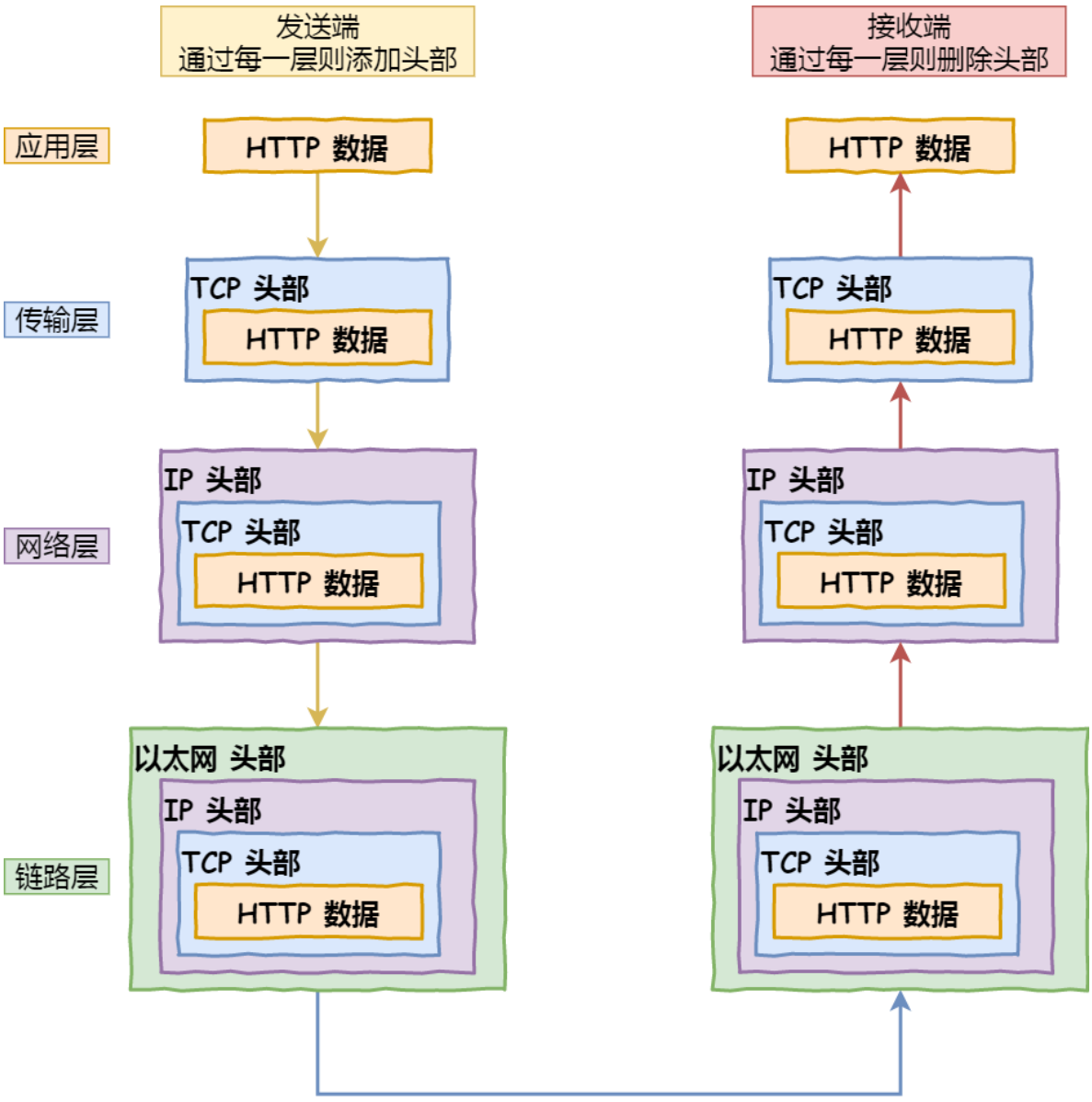
> 数据包通过多个路由器道友的帮助，在网络世界途经了很多路程，最终抵达了目的地的城门！城门值守的路由器，发现了这个小兄弟数据包原来是找城内的人，于是它就将数据包送进了城内，再经由城内的交换机帮助下，最终转发到了目的地了。数据包感慨万千的说道：“多谢这一路上，各路大侠的相助！”

---

10 互相扒皮 —— 服务器 与 客户端

数据包抵达了服务器，服务器肯定高兴呀，正所谓有朋自远方来，不亦乐乎？

服务器高兴的不得了，于是开始扒数据包的皮！就好像你收到快递，能不兴奋吗？



数据包抵达服务器后，服务器会先扒开数据包的 MAC 头部，查看是否和服务器自己的 MAC 地址符合，符合就将包收起来。

接着继续扒开数据包的 IP 头，发现 IP 地址符合，根据 IP 头中协议项，知道自己上层是 TCP 协议。

于是，扒开 TCP 的头，里面有序列号，需要看一看这个序列包是不是我想要的，如果是就放入缓存中然后返回一个 ACK，如果不是就丢弃。TCP头部里面还有端口号， HTTP 的服务器正在监听这个端口号。

于是，服务器自然就知道是 HTTP 进程想要这个包，于是就将包发给 HTTP 进程。

服务器的 HTTP 进程看到，原来这个请求是要访问一个页面，于是就把这个网页封装在 HTTP 响应报文里。

HTTP 响应报文也需要穿上 TCP、IP、MAC 头部，不过这次是源地址是服务器 IP 地址，目的地址是客户端 IP 地址。

穿好头部衣服后，从网卡出去，交由交换机转发到出城的路由器，路由器就把响应数据包发到了下一个路由器，就这样跳啊跳。

最后跳到了客户端的城门把手的路由器，路由器扒开 IP 头部发现是要找城内的人，于是又把包发给了城内的交换机，再由交换机转发到客户端。

客户端收到了服务器的响应数据包后，同样也非常的高兴，客户能拆快递了！

于是，客户端开始扒皮，把收到的数据包的皮扒剩 HTTP 响应报文后，交给浏览器去渲染页面，一份特别的数据包快递，就这样显示出来了！

最后，客户端要离开了，向服务器发起了 TCP 四次挥手，至此双方的连接就断开了。

一个数据包臭不要脸的感受

> 下面内容的 「我」，代表「臭美的数据包角色」。（括号的内容）代表我的吐槽，三连呸！

我一开始我虽然孤单、不知所措，但没有停滞不前。我依然满怀信心和勇气开始了征途。（**你当然有勇气，你是应用层数据，后面有底层兄弟当靠山，我呸！**）

我很庆幸遇到了各路神通广大的大佬，有可靠传输的 TCP、有远程定位功能的 IP、有指明下一站位置的 MAC 等（**你当然会遇到，因为都被计算机安排好的，我呸！**）。

这些大佬都给我前面加上了头部，使得我能在交换机和路由器的转发下，抵达到了目的地！（**哎，你也不容易，不吐槽了，放过你！**）

这一路上的经历，让我认识到了网络世界中各路大侠协作的重要性，是他们维护了网络世界的秩序，感谢他们！（**我呸，你应该感谢众多计算机科学家！**）

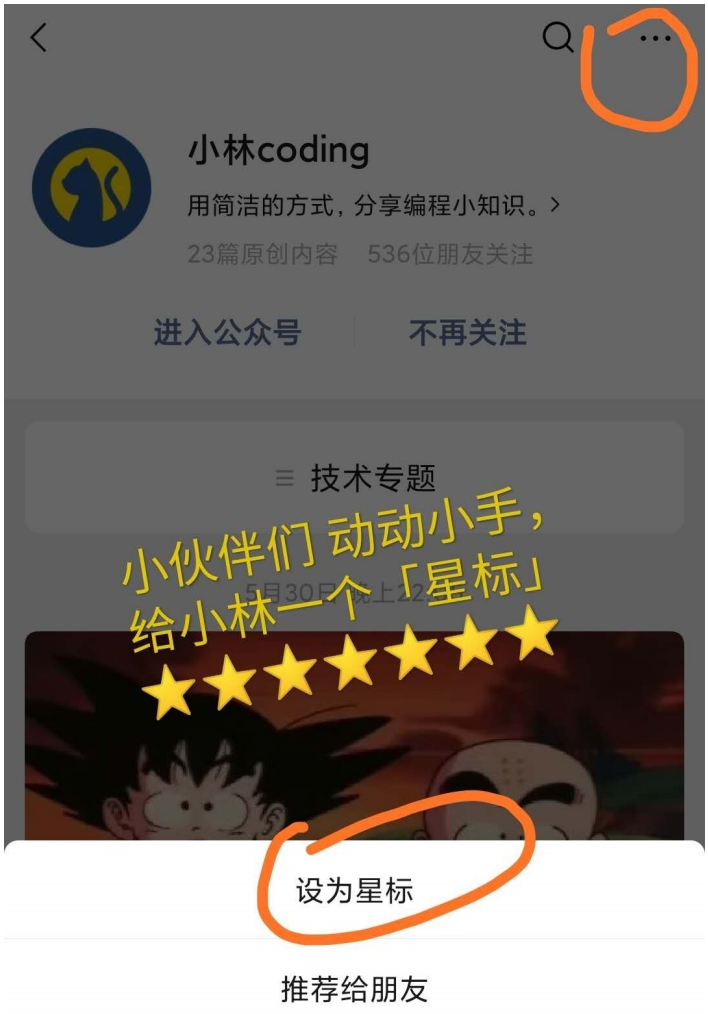
巨人的肩膀

[1] 户根勤.网络是怎么连接的.人民邮电出版社.

[2] 刘超.趣谈网络协议.极客时间.

唠叨唠叨

小林是专为大家图解的工具人，Goodbye，我们下次见！



扫一扫
关注爱图解的
「小林coding」

读者问答

> 读者问：“笔记本的是自带交换机的吗？交换机现在我还不知道是什么”

笔记本不是交换机，交换机通常是2个网口以上。

现在家里的路由器其实有了交换机的功能了。交换机可以简单理解成一个设备，三台电脑网线接到这个设备，这三台电脑就可以互相通信了，交换机嘛，交换数据这么理解就可以。

> 读者问：“如果知道你电脑的mac地址，我可以直接给你发消息吗？”

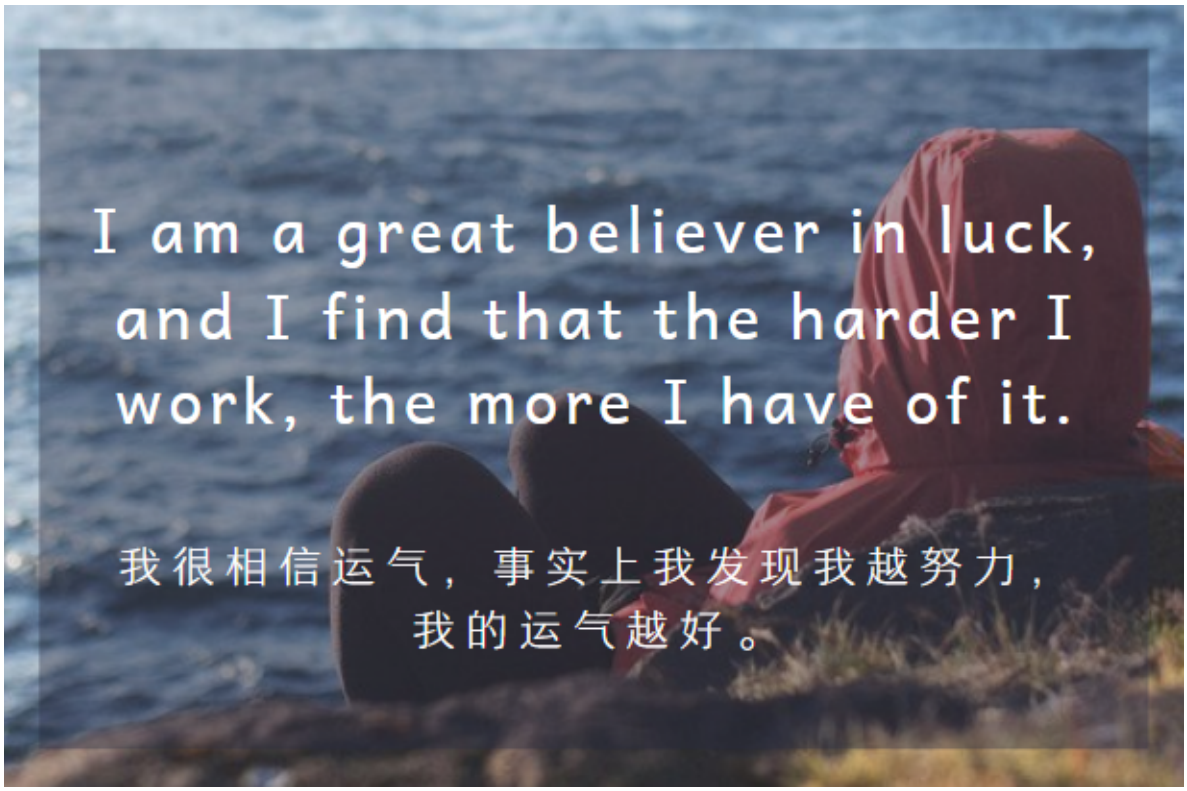
Mac地址只能是两个设备之间传递时使用的，如果你要从大老远给我发消息，是离不开 IP 的。

> 读者问：“请问公网服务器的 Mac 地址是在什么时机通过什么方式获取到的？我看 arp 获取Mac地址只能获取到内网机器的 Mac 地址吧？”

在发送数据包时，如果目标主机不是本地局域网，填入的MAC地址是路由器，也就是把数据包转发给路由器，路由器一直转发下一个路由器，直到转发到目标主机的路由器，发现 IP 地址是自己局域网内的主机，就会 arp 请求获取目标主机的 MAC 地址，从而转发到这个服务器主机。

转发的过程中，源IP地址和目标IP地址是不会变的，源MAC地址和目标MAC地址是会变化的。

硬不硬你说了算！近 40 张图解被问千百遍的 TCP 三次握手和四次挥手面试题



前言

不管面试 Java 、C/C++、Python 等开发岗位， TCP 的知识点可以说是必问的了。

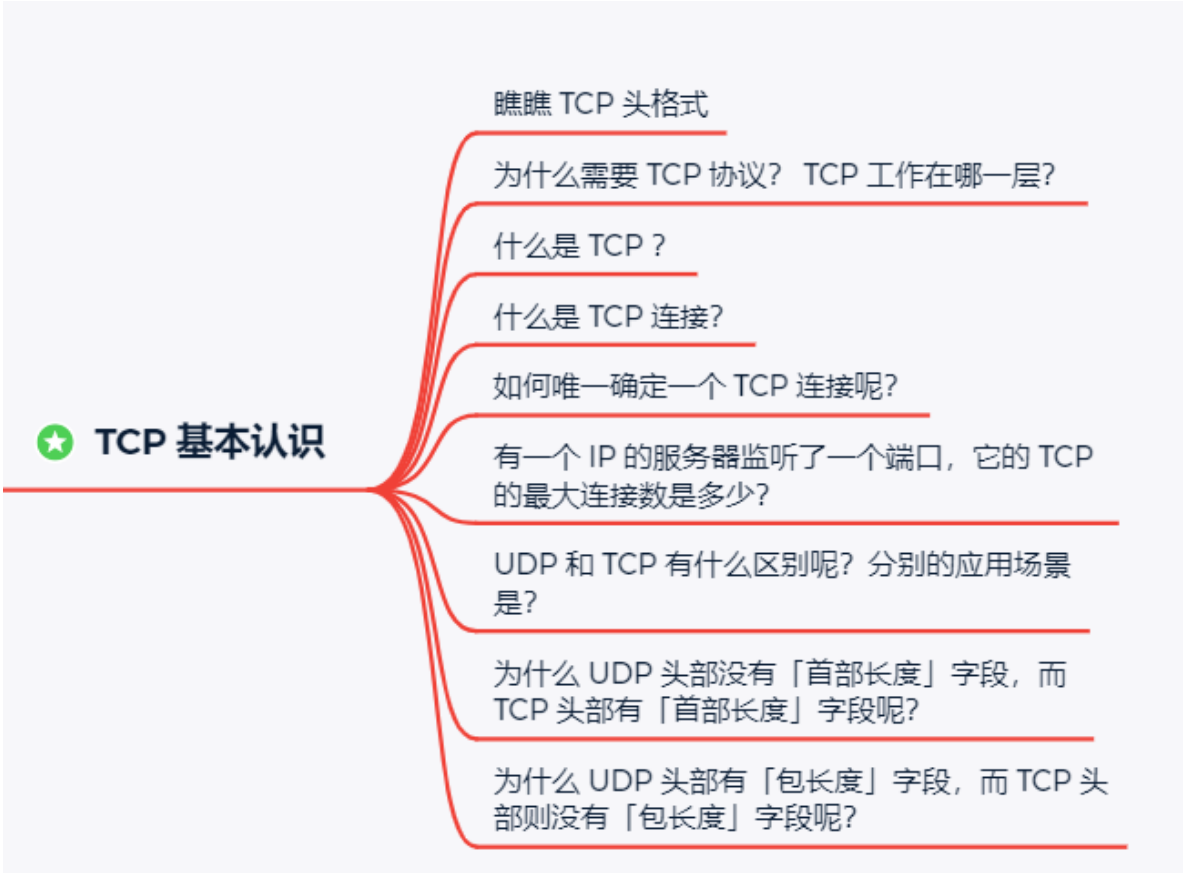
任 TCP 虐我千百遍，我仍待 TCP 如初恋。

遥想小林当年校招时常因 TCP 面试题被刷，真是又爱又恨....

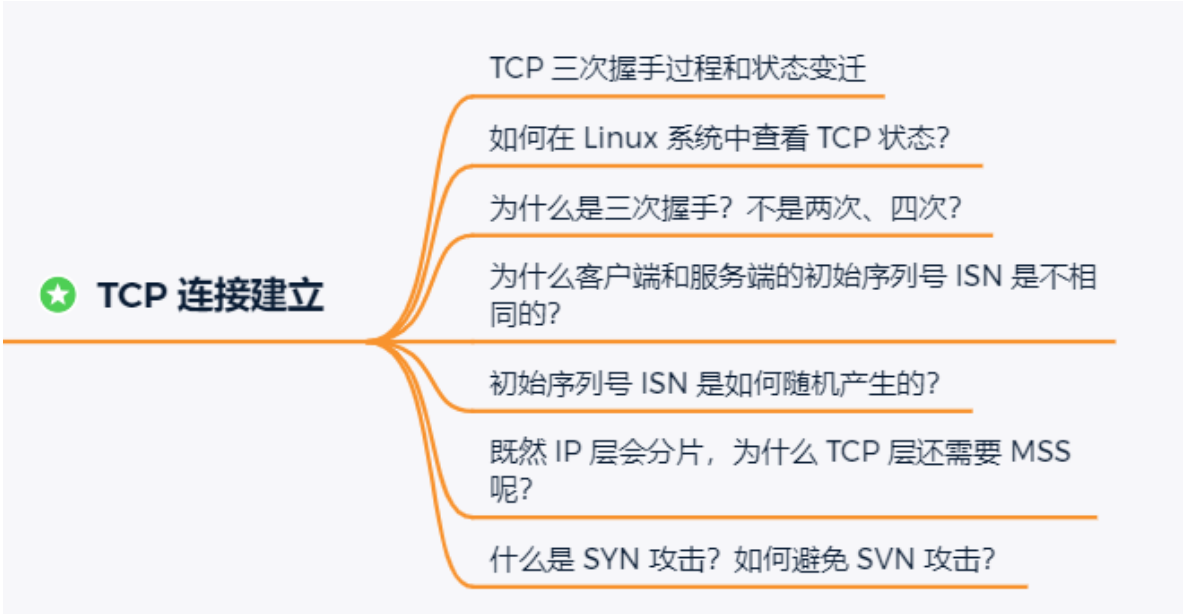
过去不会没关系，今天就让我们来消除这份恐惧，微笑着勇敢的面对它吧！

所以小林整理了关于 **TCP 三次握手和四次挥手的面试题型**，跟大家一起探讨探讨。

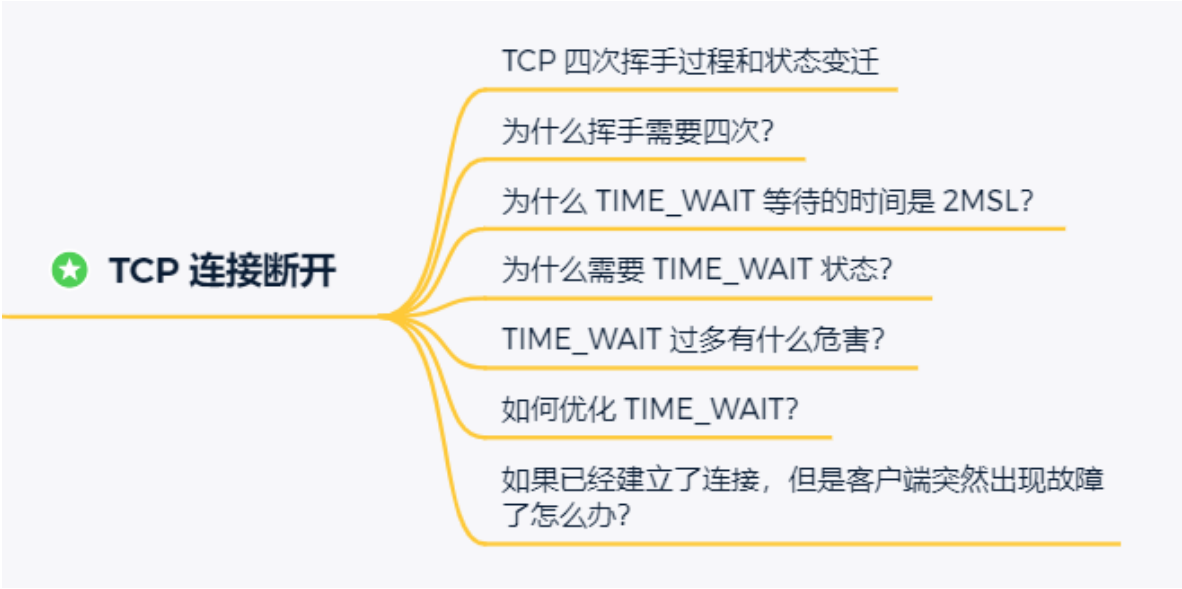
1. TCP 基本认识



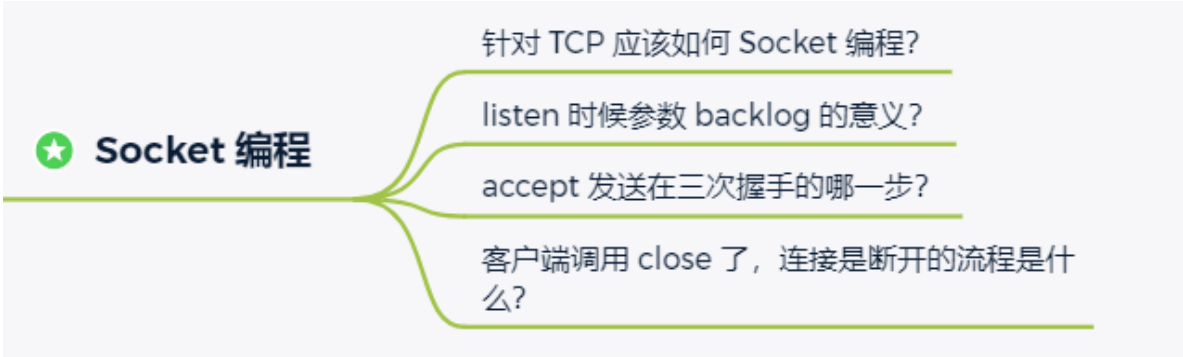
2. TCP 连接建立



3. TCP 连接断开



4. Socket 编程



> PS：本次文章不涉及 TCP 流量控制、拥塞控制、可靠性传输等方面知识，这些留在下篇哈！

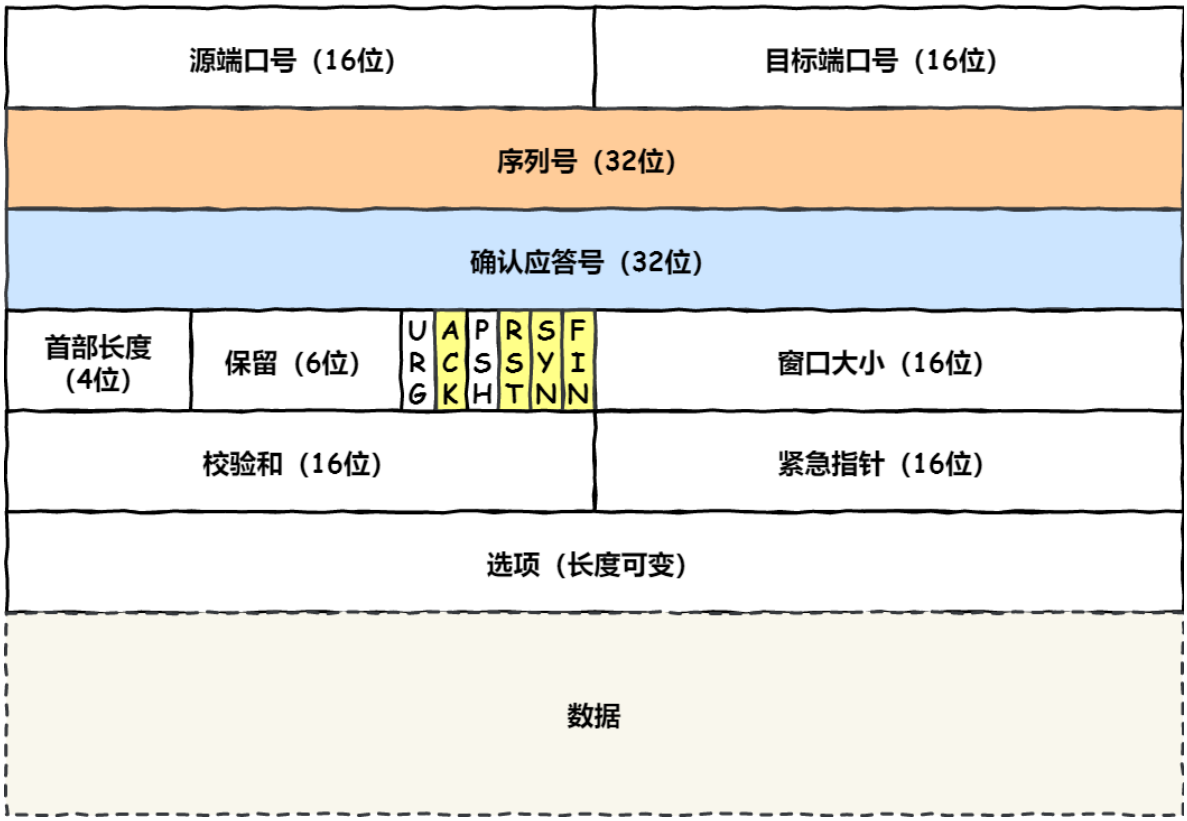
正文

01 TCP 基本认识

> 瞧瞧 TCP 头格式

我们先来看看 TCP 头的格式，标注颜色的表示与本文关联比较大的字段，其他字段不做详细阐述。

TCP 头部格式



序列号：在建立连接时由计算机生成的随机数作为其初始值，通过 SYN 包传给接收端主机，每发送一次数据，就「累加」一次该「数据字节数」的大小。**用来解决网络包乱序问题。**

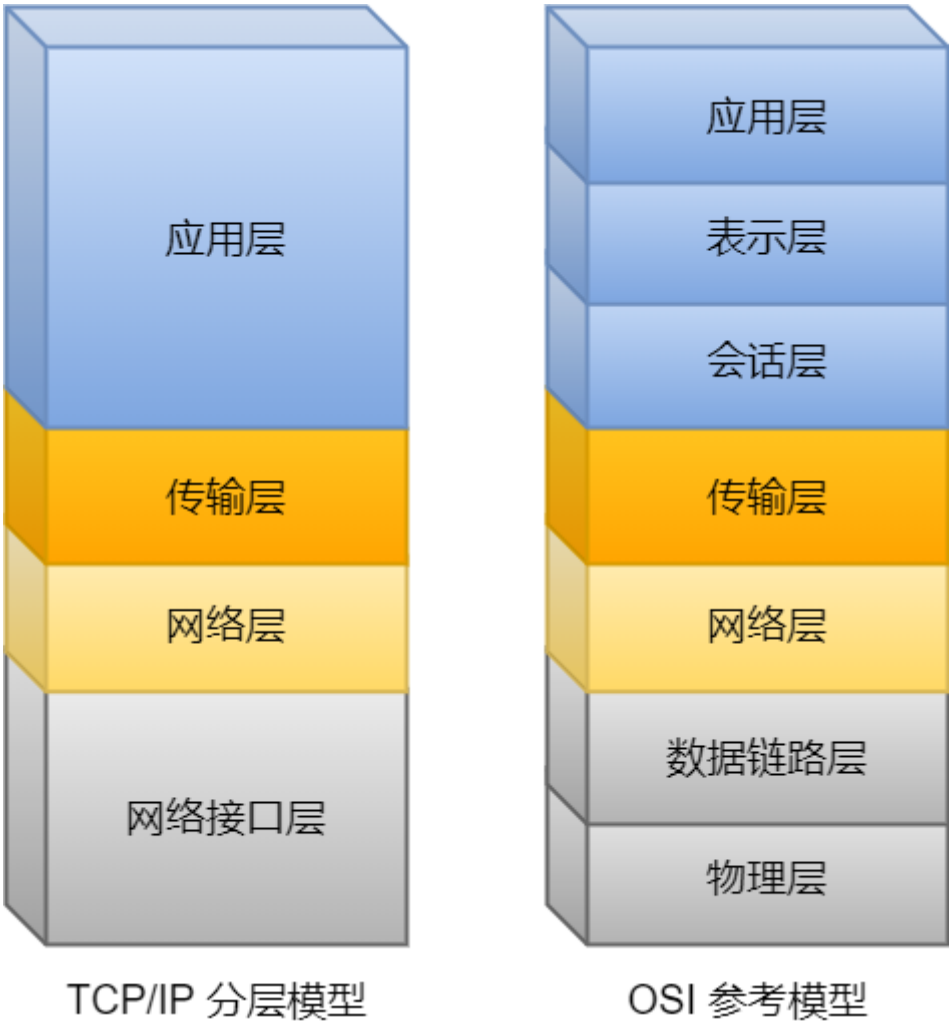
确认应答号：指下一次「期望」收到的数据的序列号，发送端收到这个确认应答以后可以认为在这个序号以前的数据都已经被正常接收。**用来解决不丢包的问题。**

控制位：

- *ACK*：该位为 `1` 时，「确认应答」的字段变为有效，TCP 规定除了最初建立连接时的 `SYN` 包之外该位必须设置为 `1` 。
- *RST*：该位为 `1` 时，表示 TCP 连接中出现异常必须强制断开连接。
- *SYN*：该位为 `1` 时，表示希望建立连接，并在其「序列号」的字段进行序列号初始值的设定。
- *FIN*：该位为 `1` 时，表示今后不会再有数据发送，希望断开连接。当通信结束希望断开连接时，通信双方的主机之间就可以相互交换 `FIN` 位为 1 的 TCP 段。

> 为什么需要 TCP 协议？ TCP 工作在哪一层？

`IP` 层是「不可靠」的，它不保证网络包的交付、不保证网络包的按序交付、也不保证网络包中的数据的完整性。



如果需要保障网络数据包的可靠性，那么就需要由上层（传输层）的 `TCP` 协议来负责。

因为 TCP 是一个工作在**传输层**的**可靠**数据传输的服务，它能确保接收端接收的网络包是**无损坏、无间隔、非冗余和按序的。**

> 什么是 TCP？

TCP 是**面向连接的、可靠的、基于字节流**的传输层通信协议。

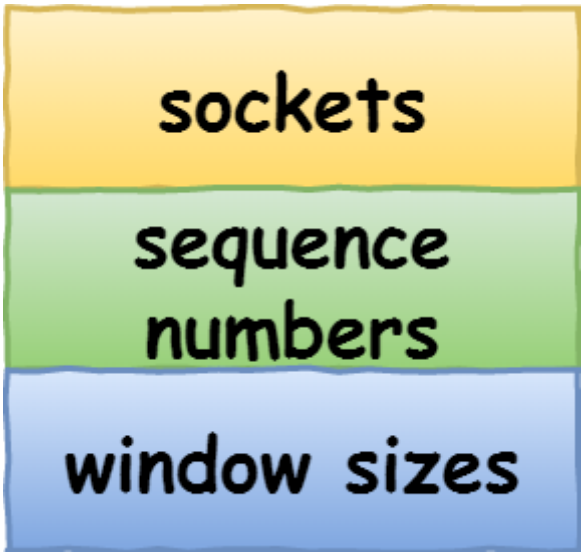
- **面向连接**：一定是「一对一」才能连接，不能像 UDP 协议可以一个主机同时向多个主机发送消息，也就是一对多是无法做到的；
- **可靠的**：无论的网络链路中出现了怎样的链路变化，TCP 都可以保证一个报文一定能够到达接收端；
- **字节流**：消息是「没有边界」的，所以无论我们消息有多大都可以进行传输。并且消息是「有序的」，当「前一个」消息没有收到的时候，即使它先收到了后面的字节，那么也不能扔给应用层去处理，同时对「重复」的报文会自动丢弃。

> 什么是 TCP 连接?

我们来看看 RFC 793 是如何定义「连接」的：

Connections: The reliability and flow control mechanisms described above require that TCPs initialize and maintain certain status information for each data stream. The combination of this information, including sockets, sequence numbers, and window sizes, is called a connection.

简单来说就是，**用于保证可靠性和流量控制维护的某些状态信息，这些信息的组合，包括Socket、序列号和窗口大小称为连接。**



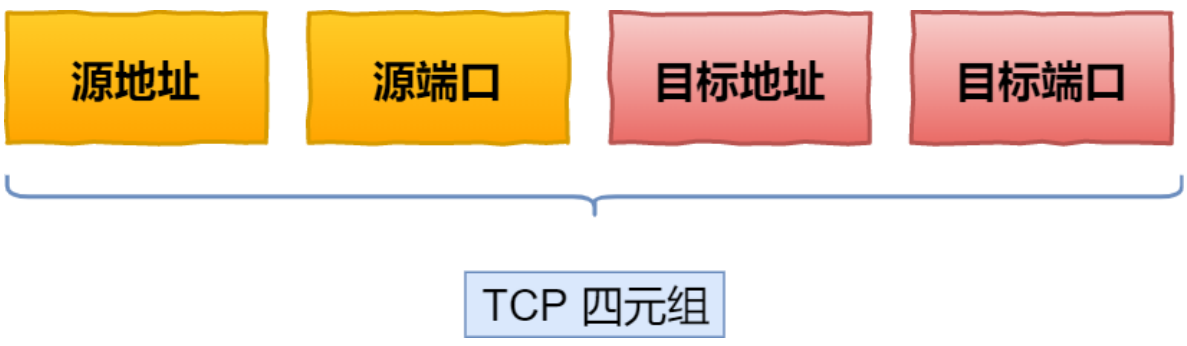
所以我们可以知道，建立一个 TCP 连接是需要客户端与服务器端达成上述三个信息的共识。

- **Socket**：由 IP 地址和端口号组成
- **序列号**：用来解决乱序问题等
- **窗口大小**：用来做流量控制

> 如何唯一确定一个 TCP 连接呢?

TCP 四元组可以唯一的确定一个连接，四元组包括如下：

- 源地址
- 源端口
- 目的地址
- 目的端口



源地址和目的地址的字段（32位）是在 IP 头部中，作用是通过 IP 协议发送报文给对方主机。

源端口和目的端口的字段（16位）是在 TCP 头部中，作用是告诉 TCP 协议应该把报文发给哪个进程。

> 有一个 IP 的服务器监听了一个端口，它的 TCP 的最大连接数是多少?

服务器通常固定在某个本地端口上监听，等待客户端的连接请求。

因此，客户端 IP 和 端口是可变的，其理论值计算公式如下:

最大 TCP 连接数 = 客户端的IP 数 X 客户端的端口数

对 IPv4，客户端的 IP 数最多为 `2` 的 `32` 次方，客户端的端口数最多为 `2` 的 `16` 次方，也就是服务端单机最大 TCP 连接数，约为 `2` 的 `48` 次方。

当然，服务端最大并发 TCP 连接数远不能达到理论上限。

- 首先主要是**文件描述符限制**，Socket 都是文件，所以首先要通过 `ulimit` 配置文件描述符的数目；
- 另一个是**内存限制**，每个 TCP 连接都要占用一定内存，操作系统的内存是有限的。

> UDP 和 TCP 有什么区别呢？分别的应用场景是?

UDP 不提供复杂的控制机制，利用 IP 提供面向「无连接」的通信服务。

UDP 协议真的非常简，头部只有 `8` 个字节（ 64 位），UDP 的头部格式如下：

UDP 头部格式

- 目标和源端口：主要是告诉 UDP 协议应该把报文发给哪个进程。
- 包长度：该字段保存了 UDP 首部的长度跟数据的长度之和。

- 校验和：校验和是为了提供可靠的 UDP 首部和数据而设计。

TCP 和 UDP 区别：

1. 连接

- TCP 是面向连接的传输层协议，传输数据前先要建立连接。
- UDP 是不需要连接，即刻传输数据。

2. 服务对象

- TCP 是一对一的两点服务，即一条连接只有两个端点。
- UDP 支持一对一、一对多、多对多的交互通信

3. 可靠性

- TCP 是可靠交付数据的，数据可以无差错、不丢失、不重复、按需到达。
- UDP 是尽最大努力交付，不保证可靠交付数据。

4. 拥塞控制、流量控制

- TCP 有拥塞控制和流量控制机制，保证数据传输的安全性。
- UDP 则没有，即使网络非常拥堵了，也不会影响 UDP 的发送速率。

5. 首部开销

- TCP 首部长度较长，会有一定的开销，首部在没有使用「选项」字段时是 `20` 个字节，如果使用了「选项」字段则会变长的。
- UDP 首部只有 8 个字节，并且是固定不变的，开销较小。

6. 传输方式

- TCP 是流式传输，没有边界，但保证顺序和可靠。
- UDP 是一个包一个包的发送，是有边界的，但可能会丢包和乱序。

7. 分片不同

- TCP 的数据大小如果大于 MSS 大小，则会在传输层进行分片，目标主机收到后，也同样在传输层组装 TCP 数据包，如果中途丢失了一个分片，只需要传输丢失的这个分片。
- UDP 的数据大小如果大于 MTU 大小，则会在 IP 层进行分片，目标主机收到后，在 IP 层组装完数据，接着再传给传输层，但是如果中途丢了一个分片，则就需要重传所有的数据包，这样传输效率非常差，所以通常 UDP 的报文应该小于 MTU。

TCP 和 UDP 应用场景：

由于 TCP 是面向连接，能保证数据的可靠性交付，因此经常用于：

- `FTP` 文件传输
- `HTTP` / `HTTPS`

由于 UDP 面向无连接，它可以随时发送数据，再加上UDP本身的处理既简单又高效，因此经常用于：

- 包总量较少的通信，如 `DNS` 、 `SNMP` 等
- 视频、音频等多媒体通信
- 广播通信

> 为什么 UDP 头部没有「首部长度」字段，而 TCP 头部有「首部长度」字段呢?

原因是 TCP 有**可变长**的「选项」字段，而 UDP 头部长度则是**不会变化**的，无需多一个字段去记录 UDP 的首部长度。

> 为什么 UDP 头部有「包长度」字段，而 TCP 头部则没有「包长度」字段呢?

先说说 TCP 是如何计算负载数据长度：

TCP 数据的长度 = IP 总长度 - IP 首部长度 - TCP 首部长度

其中 IP 总长度 和 IP 首部长度，在 IP 首部格式是已知的。TCP 首部长度，则是在 TCP 首部格式已知的，所以就可以求得 TCP 数据的长度。

大家这时就奇怪了问：“ UDP 也是基于 IP 层的呀，那 UDP 的数据长度也可以通过这个公式计算呀？为何还要有「包长度」呢？”

这么一问，确实感觉 UDP 「包长度」是冗余的。

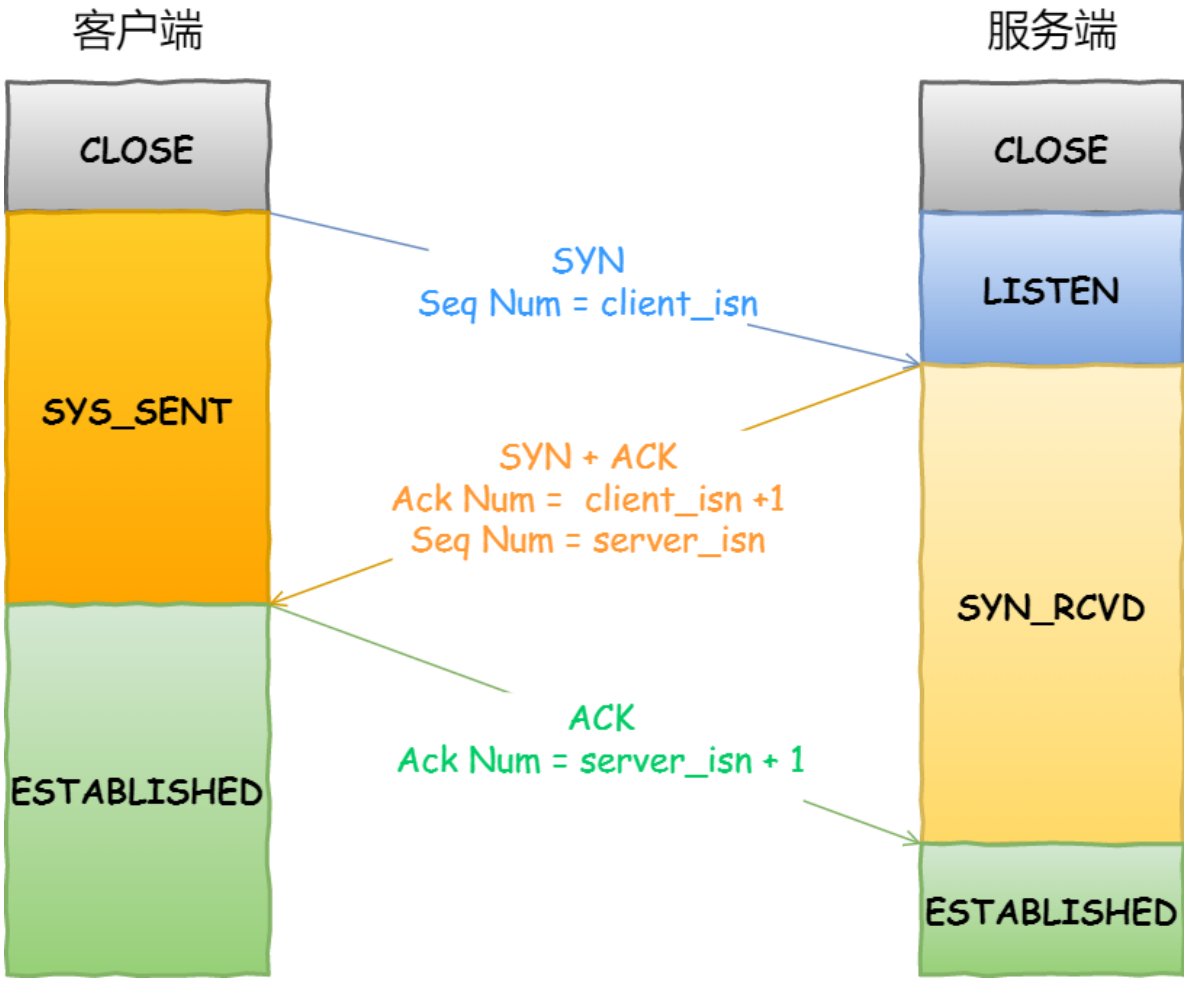
因为为了网络设备硬件设计和处理方便，首部长度需要是 `4` 字节的整数倍。

如果去掉 UDP 「包长度」字段，那 UDP 首部长度就不是 `4` 字节的整数倍了，所以小林觉得这可能是为了补全 UDP 首部长度是 `4` 字节的整数倍，才补充了「包长度」字段。

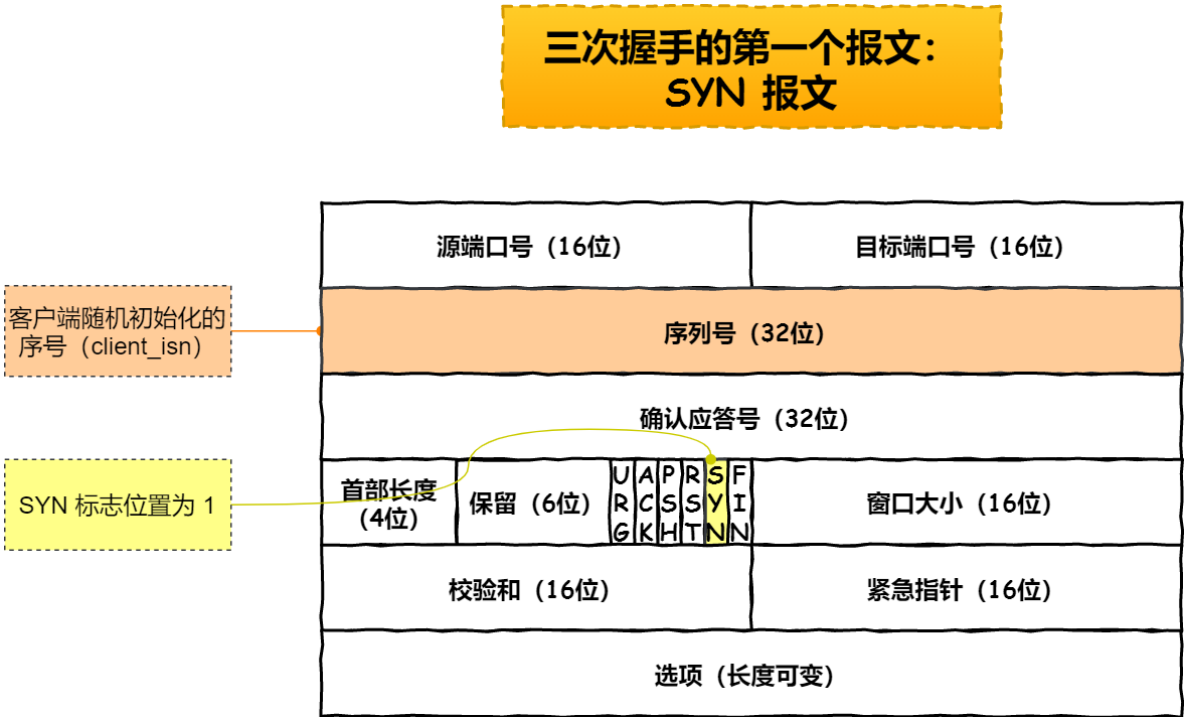
02 TCP 连接建立

> TCP 三次握手过程和状态变迁

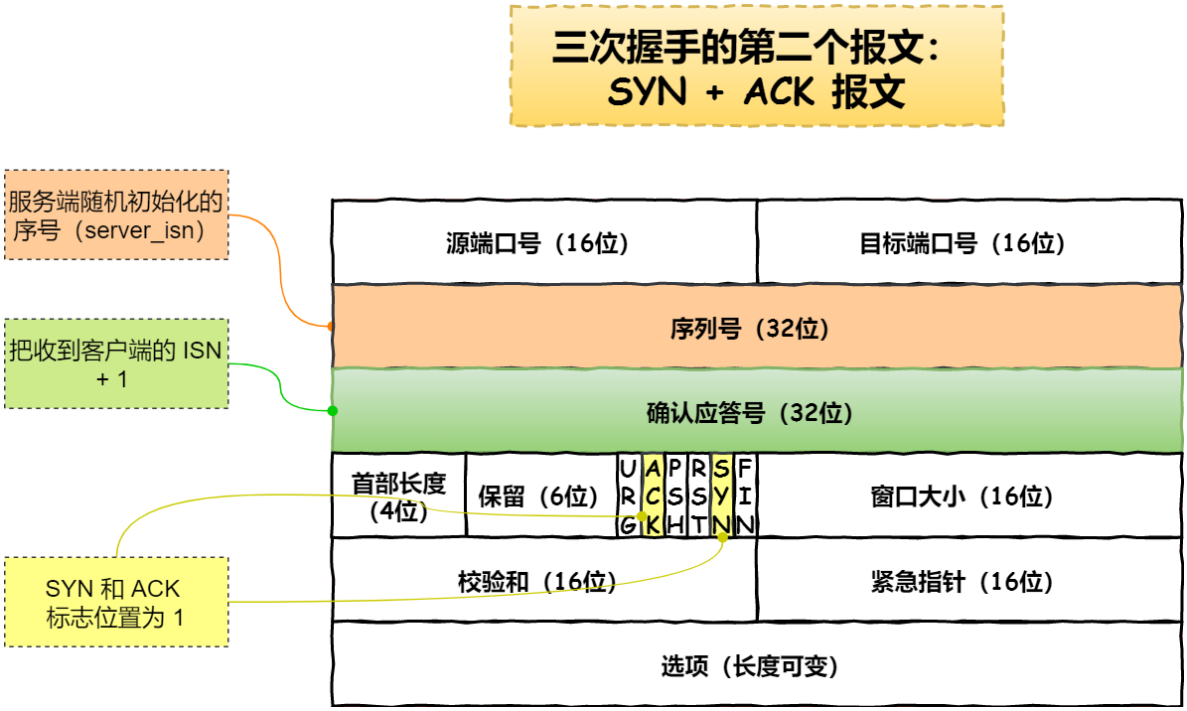
TCP 是面向连接的协议，所以使用 TCP 前必须先建立连接，而**建立连接是通过三次握手来进行的。**



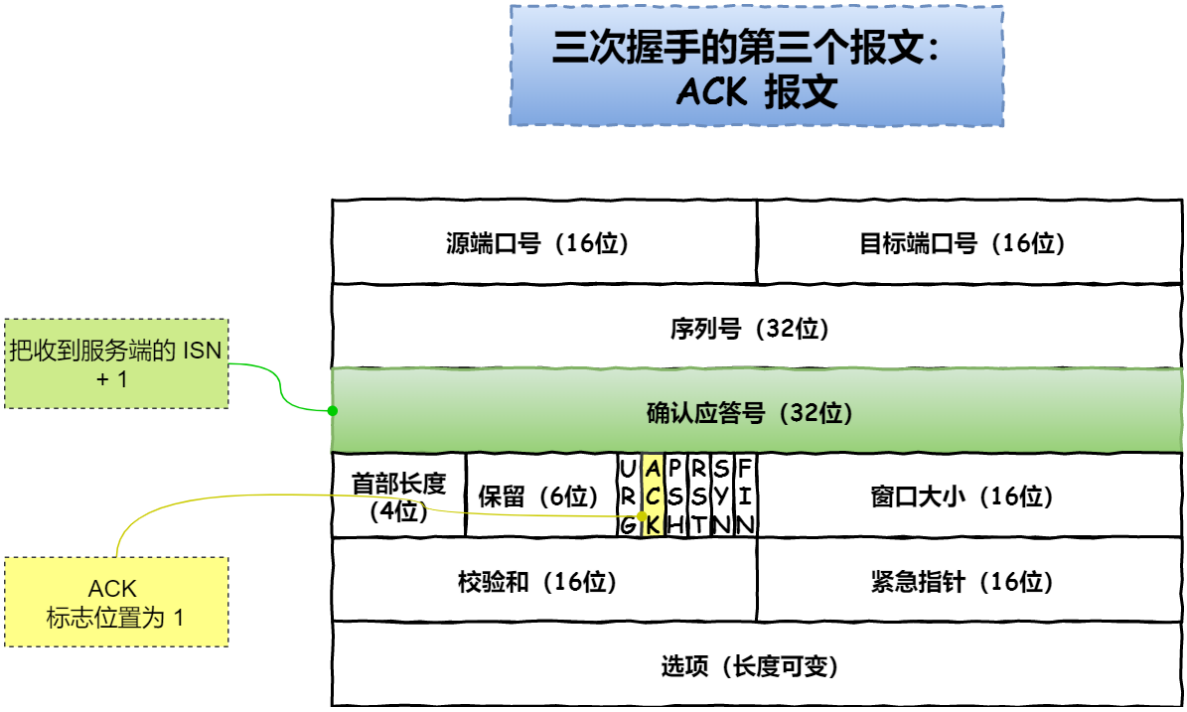
- 一开始，客户端和服务端都处于 `CLOSED` 状态。先是服务端主动监听某个端口，处于 `LISTEN` 状态



- 客户端会随机初始化序号（`client_isn`），将此序号置于 TCP 首部的「序号」字段中，同时把 `SYN` 标志位置为 `1` ，表示 `SYN` 报文。接着把第一个 SYN 报文发送给服务端，表示向服务端发起连接，该报文不包含应用层数据，之后客户端处于 `SYN-SENT` 状态。



- 服务端收到客户端的 `SYN` 报文后，首先服务端也随机初始化自己的序号（ `server_isn` ），将此序号填入 TCP 首部的「序号」字段中，其次把 TCP 首部的「确认应答号」字段填入 `client_isn + 1`，接着把 `SYN` 和 `ACK` 标志位置为 `1` 。最后把该报文发给客户端，该报文也不包含应用层数据，之后服务端处于 `SYN-RCVD` 状态。



- 客户端收到服务端报文后，还要向服务端回应最后一个应答报文，首先该应答报文 TCP 首部 `ACK` 标志位置为 `1` ，其次「确认应答号」字段填入 `server_isn + 1` ，最后把报文发送给服务端，这次报文可以携带客户到服务器的数据，之后客户端处于 `ESTABLISHED` 状态。
- 服务器收到客户端的应答报文后，也进入 `ESTABLISHED` 状态。

从上面的过程可以发现**第三次握手是可以携带数据的，前两次握手是不可以携带数据的**，这也是面试常问的题。

一旦完成三次握手，双方都处于 `ESTABLISHED` 状态，此时连接就已建立完成，客户端和服务端就可以相互发送数据了。

> 如何在 Linux 系统中查看 TCP 状态?

TCP 的连接状态查看，在 Linux 可以通过 `netstat -napt` 命令查看。

> 为什么是三次握手？不是两次、四次?

相信大家比较常回答的是：“因为三次握手才能保证双方具有接收和发送的能力。”

这回答是没问题，但这回答是片面的，并没有说出主要的原因。

在前面我们知道了什么是 **TCP 连接**：

- 用于保证可靠性和流量控制维护的某些状态信息，这些信息的组合，包括**Socket、序列号和窗口大小**称为连接。

所以，重要的是**为什么三次握手才可以初始化Socket、序列号和窗口大小并建立 TCP 连接。**

接下来以三个方面分析三次握手的原因：

- 三次握手才可以阻止重复历史连接的初始化（主要原因）
- 三次握手才可以同步双方的初始序列号
- 三次握手才可以避免资源浪费

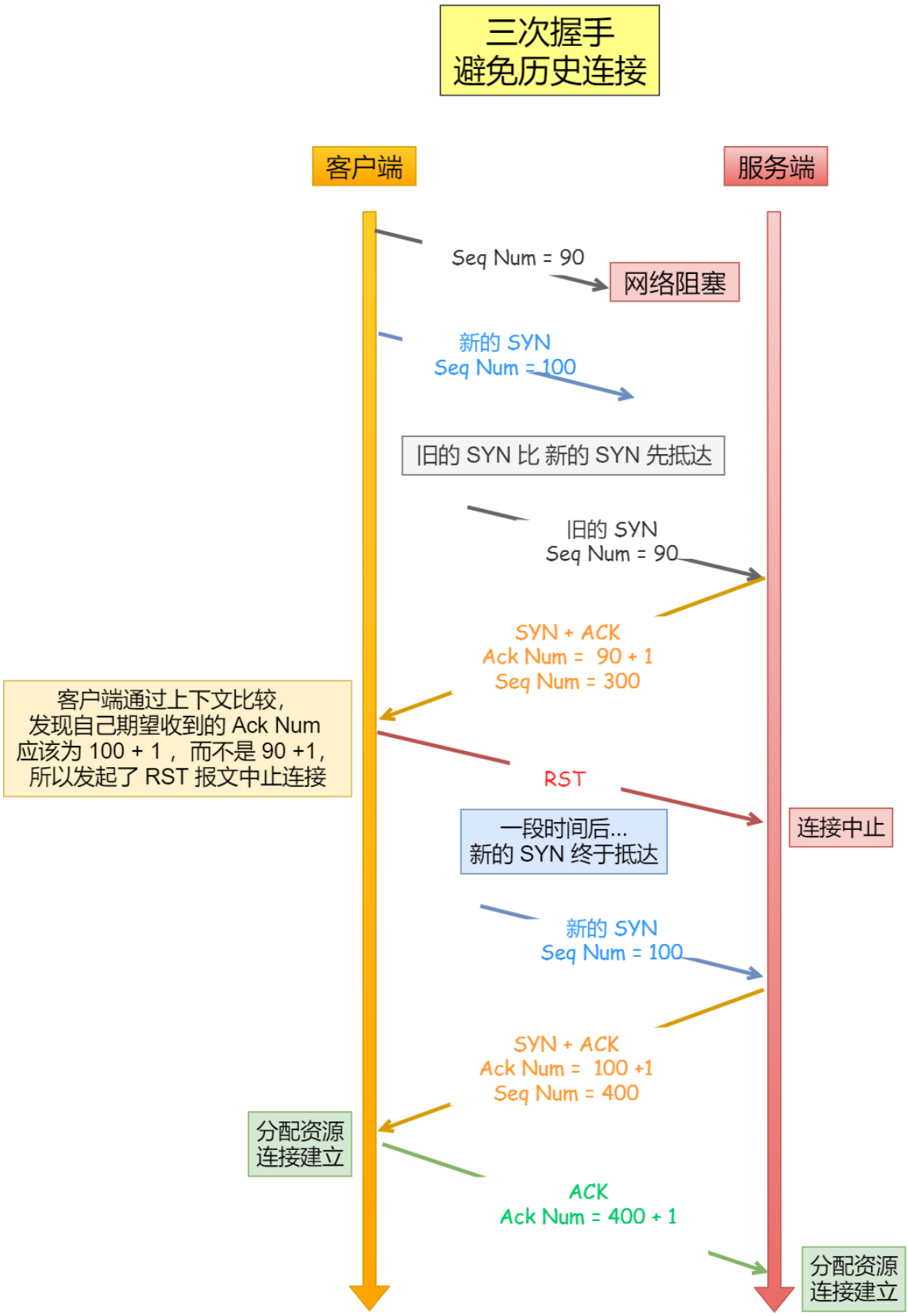
原因一：避免历史连接

我们来看看 RFC 793 指出的 TCP 连接使用三次握手的**首要原因**：

The principle reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion.

简单来说，三次握手的**首要原因是为了防止旧的重复连接初始化造成混乱。**

网络环境是错综复杂的，往往并不是如我们期望的一样，先发送的数据包，就先到达目标主机，反而它很骚，可能会由于网络拥堵等乱七八糟的原因，会使得旧的数据包，先到达目标主机，那么这种情况下 TCP 三次握手是如何避免的呢?



客户端连续发送多次 SYN 建立连接的报文，在**网络拥堵**情况下：

- 一个「旧 SYN 报文」比「最新的 SYN 」 报文早到达了服务端；
- 那么此时服务端就会回一个 `SYN + ACK` 报文给客户端；
- 客户端收到后可以根据自身的上下文，判断这是一个历史连接（序列号过期或超时），那么客户端就会发送 `RST` 报文给服务端，表示中止这一次连接。

如果是两次握手连接，就不能判断当前连接是否是历史连接，三次握手则可以在客户端（发送方）准备发送第三次报文时，客户端因有足够的上下文来判断当前连接是否是历史连接：

- 如果是历史连接（序列号过期或超时），则第三次握手发送的报文是 `RST` 报文，以此中止历史连接；
- 如果不是历史连接，则第三次发送的报文是 `ACK` 报文，通信双方就会成功建立连接；

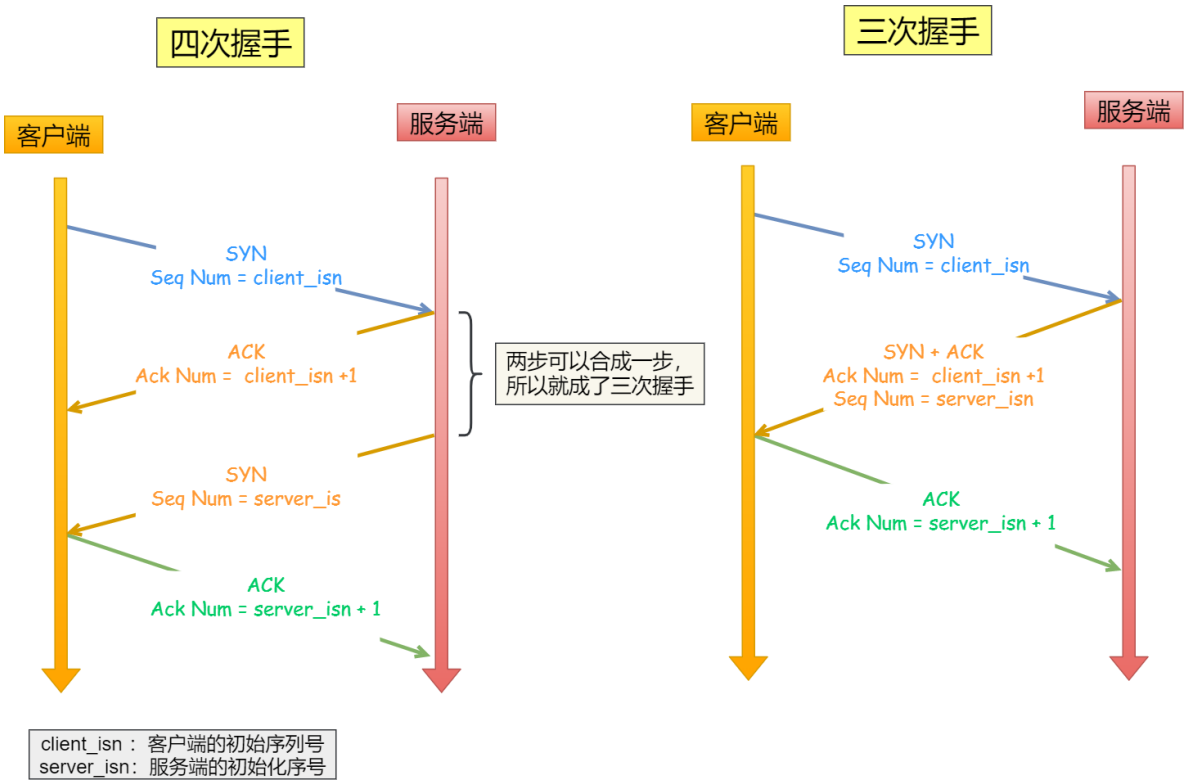
所以，TCP 使用三次握手建立连接的最主要原因是**防止历史连接初始化了连接。**

原因二：同步双方初始序列号

TCP 协议的通信双方， 都必须维护一个「序列号」， 序列号是可靠传输的一个关键因素，它的作用：

- 接收方可以去除重复的数据；
- 接收方可以根据数据包的序列号按序接收；
- 可以标识发送出去的数据包中， 哪些是已经被对方收到的；

可见，序列号在 TCP 连接中占据着非常重要的作用，所以当客户端发送携带「初始序列号」的 `SYN` 报文的时候，需要服务端回一个 `ACK` 应答报文，表示客户端的 SYN 报文已被服务端成功接收，那当服务端发送「初始序列号」给客户端的时候，依然也要得到客户端的应答回应，**这样一来一回，才能确保双方的初始序列号能被可靠的同步。**



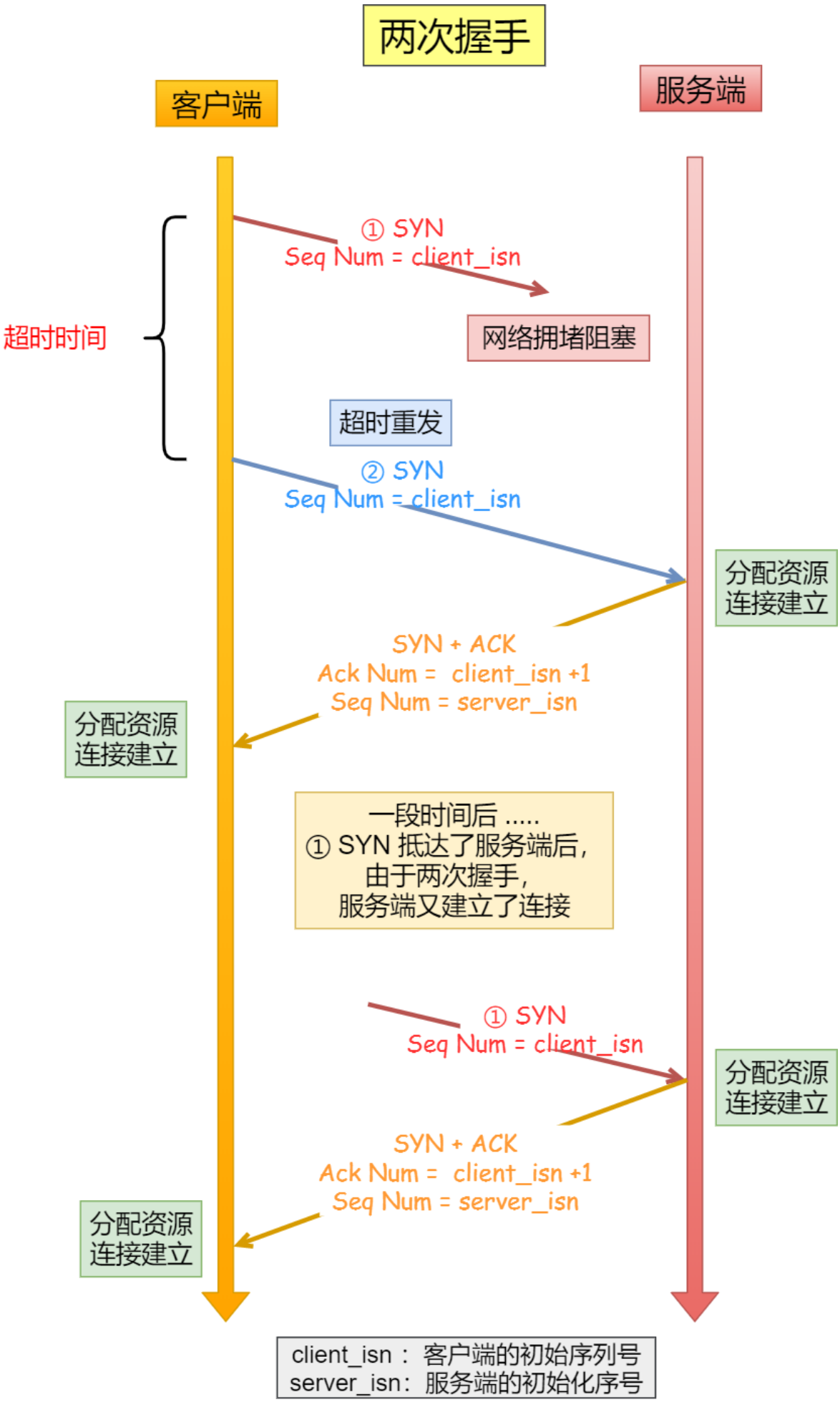
四次握手其实也能够可靠的同步双方的初始化序号，但由于**第二步和第三步可以优化成一步**，所以就成了「三次握手」。

而两次握手只保证了一方的初始序列号能被对方成功接收，没办法保证双方的初始序列号都能被确认接收。

原因三：避免资源浪费

如果只有「两次握手」，当客户端的 `SYN` 请求连接在网络中阻塞，客户端没有接收到 `ACK` 报文，就会重新发送 `SYN` ，由于没有第三次握手，服务器不清楚客户端是否收到了自己发送的建立连接的 `ACK` 确认信号，所以每收到一个 `SYN` 就只能先主动建立一个连接，这会造成什么情况呢？

如果客户端的 `SYN` 阻塞了，重复发送多次 `SYN` 报文，那么服务器在收到请求后就会**建立多个冗余的无效链接，造成不必要的资源浪费。**



即两次握手会造成消息滞留情况下，服务器重复接受无用的连接请求 `SYN` 报文，而造成重复分配资源。

小结

TCP 建立连接时，通过三次握手**能防止历史连接的建立，能减少双方不必要的资源开销，能帮助双方同步初始化序列号**。序列号能够保证数据包不重复、不丢弃和按序传输。

不使用「两次握手」和「四次握手」的原因：

- 「两次握手」：无法防止历史连接的建立，会造成双方资源的浪费，也无法可靠的同步双方序列号；
- 「四次握手」：三次握手就已经理论上最少可靠连接建立，所以不需要使用更多的通信次数。

> 为什么客户端和服务端的初始序列号 ISN 是不相同的?

如果一个已经失效的连接被重用了，但是该旧连接的历史报文还残留在网络中，如果序列号相同，那么就无法分辨出该报文是不是历史报文，如果历史报文被新的连接接收了，则会产生数据错乱。

所以，每次建立连接前重新初始化一个序列号主要是为了通信双方能够根据序号将不属于本连接的报文段丢弃。

另一方面是为了安全性，防止黑客伪造的相同序列号的 TCP 报文被对方接收。

> 初始序列号 ISN 是如何随机产生的?

起始 `ISN` 是基于时钟的，每 4 毫秒 + 1，转一圈要 4.55 个小时。

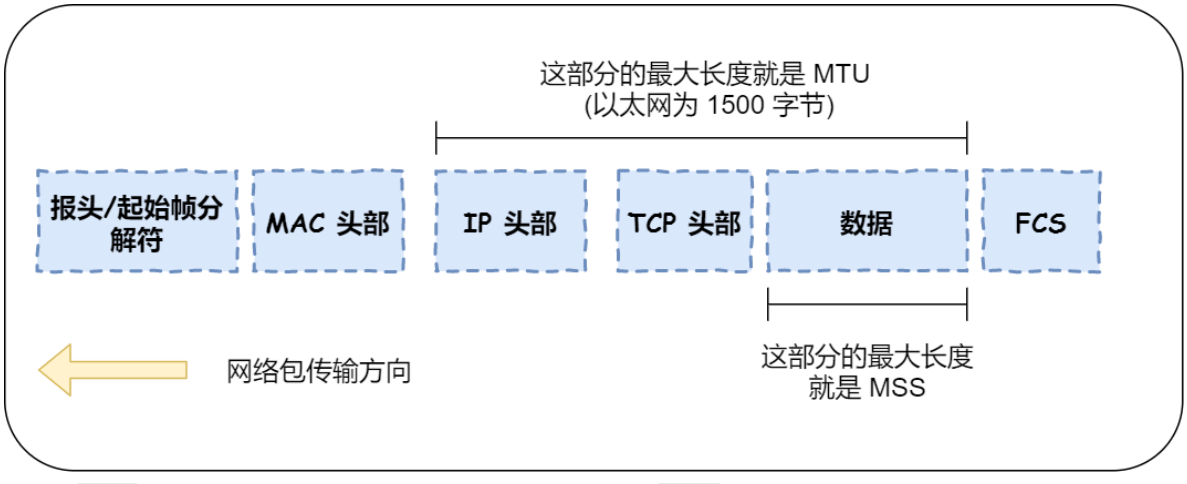
RFC1948 中提出了一个较好的初始化序列号 ISN 随机生成算法。

ISN = M + F (localhost, localport, remotehost, remoteport)

- `M` 是一个计时器，这个计时器每隔 4 毫秒加 1。
- `F` 是一个 Hash 算法，根据源 IP、目的 IP、源端口、目的端口生成一个随机数值。要保证 Hash 算法不能被外部轻易推算得出，用 MD5 算法是一个比较好的选择。

> 既然 IP 层会分片，为什么 TCP 层还需要 MSS 呢?

我们先来认识下 MTU 和 MSS



- `MTU`：一个网络包的最大长度，以太网中一般为 `1500` 字节；
- `MSS`：除去 IP 和 TCP 头部之后，一个网络包所能容纳的 TCP 数据的最大长度；

如果在 TCP 的整个报文（头部 + 数据）交给 IP 层进行分片，会有什么异常呢？

当 IP 层有一个超过 `MTU` 大小的数据（TCP 头部 + TCP 数据）要发送，那么 IP 层就要进行分片，把数据分片成若干片，保证每一个分片都小于 MTU。把一份 IP 数据报进行分片以后，由目标主机的 IP 层来进行重新组装后，再交给上一层 TCP 传输层。

这看起来井然有序，但这存在隐患的，**那么当如果一个 IP 分片丢失，整个 IP 报文的所有分片都得重传**。

因为 IP 层本身没有超时重传机制，它由传输层的 TCP 来负责超时和重传。

当接收方发现 TCP 报文（头部 + 数据）的某一片丢失后，则不会响应 ACK 给对方，那么发送方的 TCP 在超时后，就会重发「整个 TCP 报文（头部 + 数据）」。

因此，可以得知由 IP 层进行分片传输，是非常没有效率的。

所以，为了达到最佳的传输效能 TCP 协议在**建立连接的时候通常要协商双方的 MSS 值**，当 TCP 层发现数据超过 MSS 时，则就先会进行分片，当然由它形成的 IP 包的长度也就不会大于 MTU ，自然也就不用 IP 分片了。

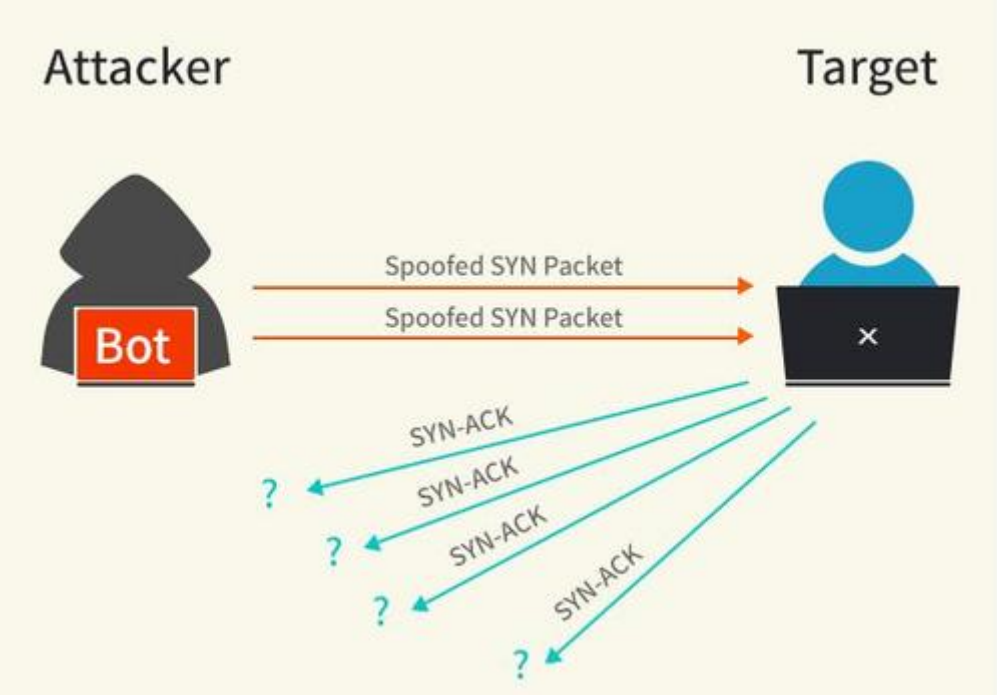
```
[SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
[SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1412 SACK_PERM=1 WS=16384
[ACK] Seq=1 Ack=1 Win=66304 Len=0
```

经过 TCP 层分片后，如果一个 TCP 分片丢失后，**进行重发时也是以 MSS 为单位**，而不用重传所有的分片，大大增加了重传的效率。

> 什么是 SYN 攻击？如何避免 SYN 攻击？

SYN 攻击

我们都知道 TCP 连接建立是需要三次握手，假设攻击者短时间伪造不同 IP 地址的 `SYN` 报文，服务端每接收到一个 `SYN` 报文，就进入 `SYN_RCVD` 状态，但服务端发送出去的 `ACK + SYN` 报文，无法得到未知 IP 主机的 `ACK` 应答，久而久之就会**占满服务端的 SYN 接收队列（未连接队列）**，使得服务器不能为正常用户服务。



避免 SYN 攻击方式一

其中一种解决方式是通过修改 Linux 内核参数，控制队列大小和当队列满时应做什么处理。

- 当网卡接收数据包的速度大于内核处理的速度时，会有一个队列保存这些数据包。控制该队列的最大值如下参数：

```
net.core.netdev_max_backlog
```

- SYN_RCVD 状态连接的最大个数：

```
net.ipv4.tcp_max_syn_backlog
```

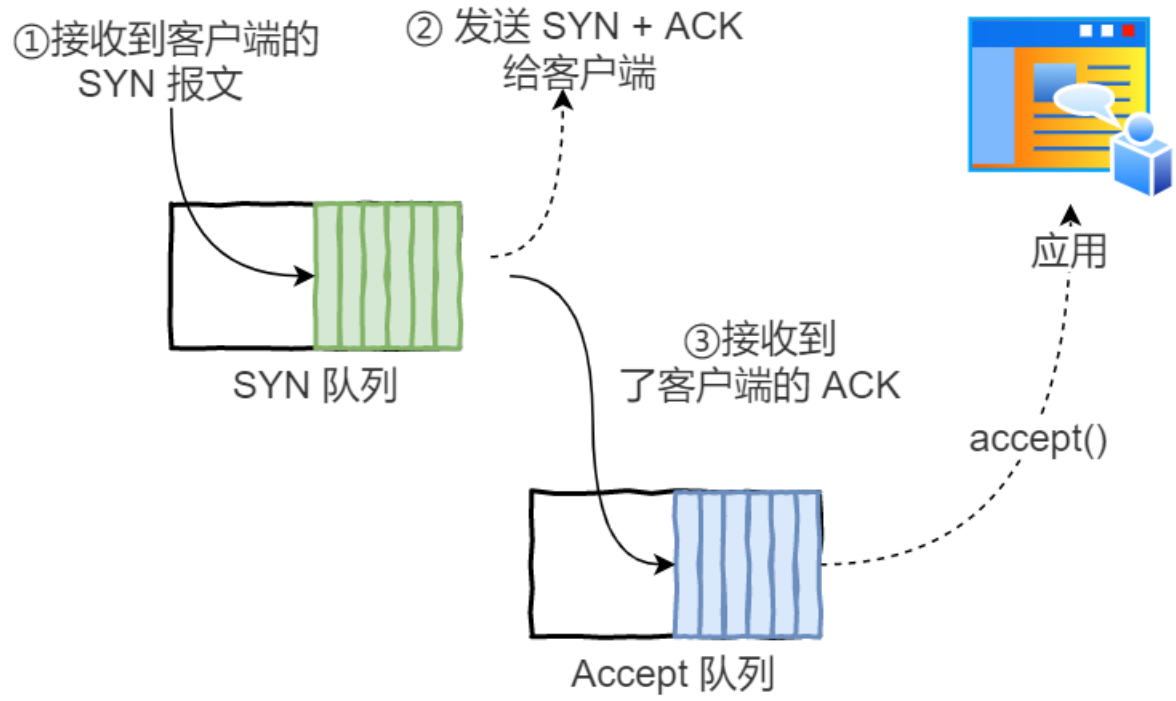
- 超出处理能时，对新的 SYN 直接回报 RST，丢弃连接：

```
net.ipv4.tcp_abort_on_overflow
```

避免 SYN 攻击方式二

我们先来看下 Linux 内核的 `SYN` （未完成连接建立）队列与 `Accpet` （已完成连接建立）队列是如何工作的？

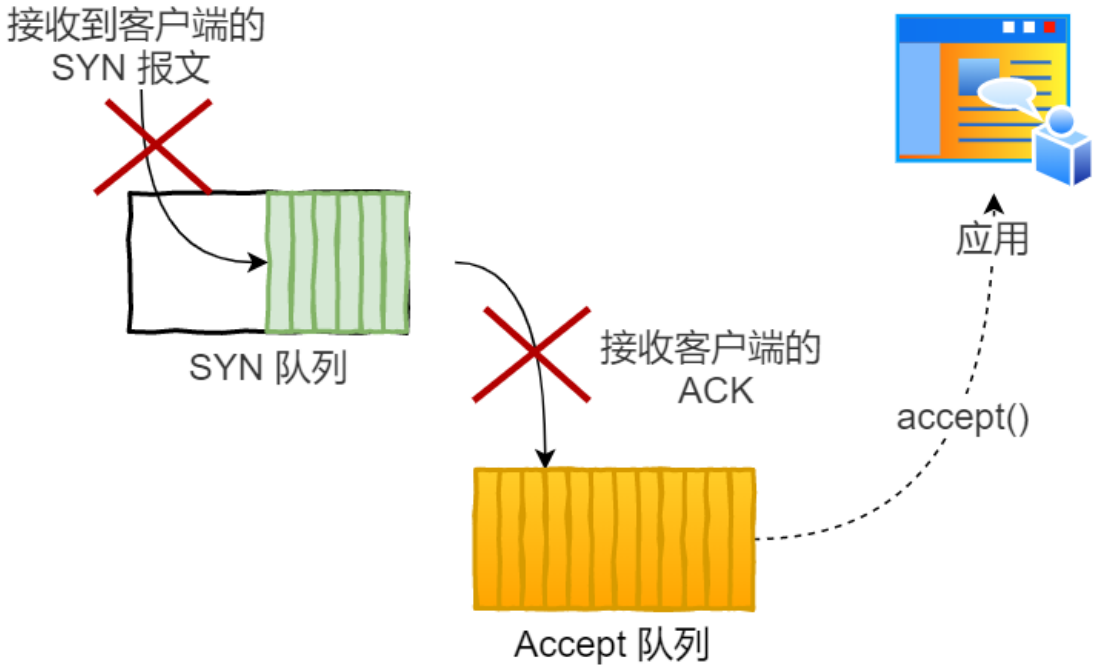
正常流程



正常流程：

- 当服务端接收到客户端的 SYN 报文时，会将其加入到内核的「 SYN 队列」；
- 接着发送 SYN + ACK 给客户端，等待客户端回应 ACK 报文；
- 服务端接收到 ACK 报文后，从「 SYN 队列」移除放入到「 Accept 队列」；
- 应用通过调用 `accpet()` socket 接口，从「 Accept 队列」取出连接。

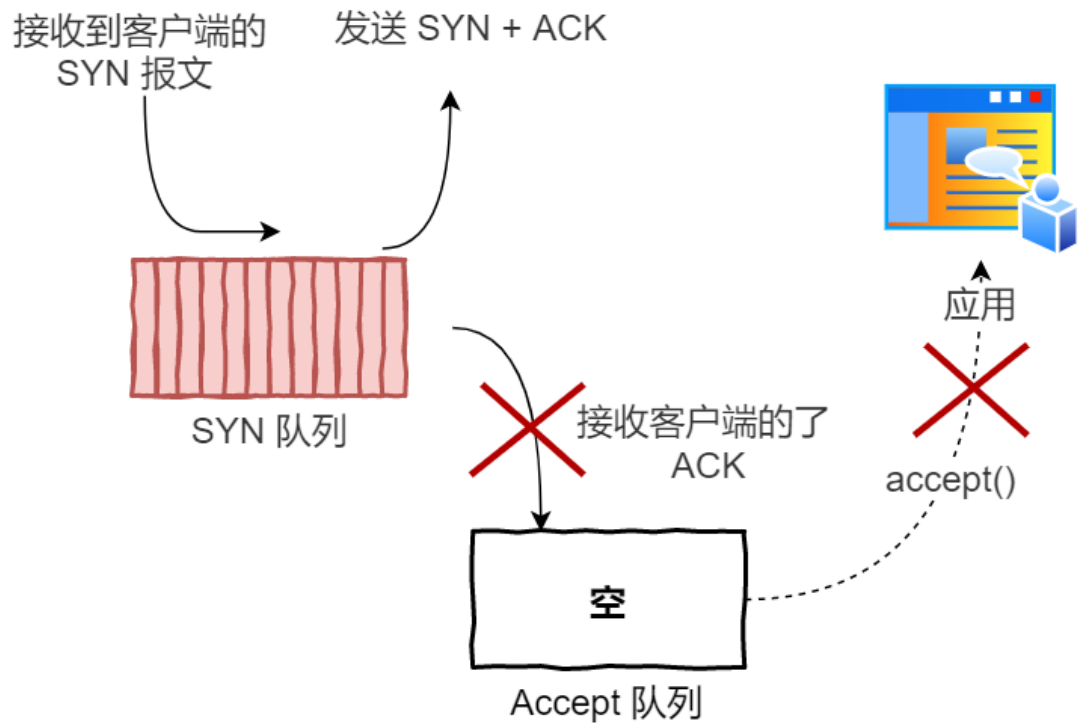
应用程序过慢



应用程序过慢：

- 如果应用程序过慢时，就会导致「 Accept 队列」被占满。

受到 SYN 攻击



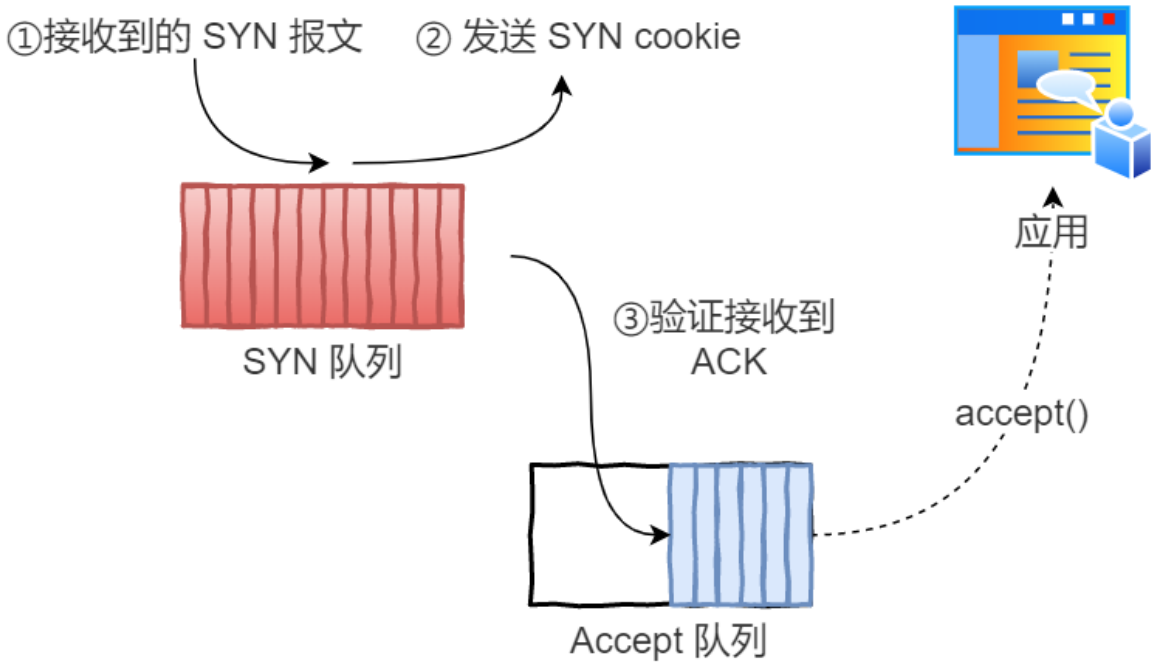
受到 SYN 攻击：

- 如果不断受到 SYN 攻击，就会导致「 SYN 队列」被占满。

`tcp_syncookies` 的方式可以应对 SYN 攻击的方法：

```
net.ipv4.tcp_syncookies = 1
```

SYN 队列占满，启动 cookie



- 当 「 SYN 队列」满之后，后续服务器收到 SYN 包，不进入「 SYN 队列」；
- 计算出一个 `cookie` 值，再以 SYN + ACK 中的「序列号」返回客户端，
- 服务端接收到客户端的应答报文时，服务器会检查这个 ACK 包的合法性。如果合法，直接放入到「 Accept 队列」。
- 最后应用通过调用 `accpet()` socket 接口，从「 Accept 队列」取出的连接。

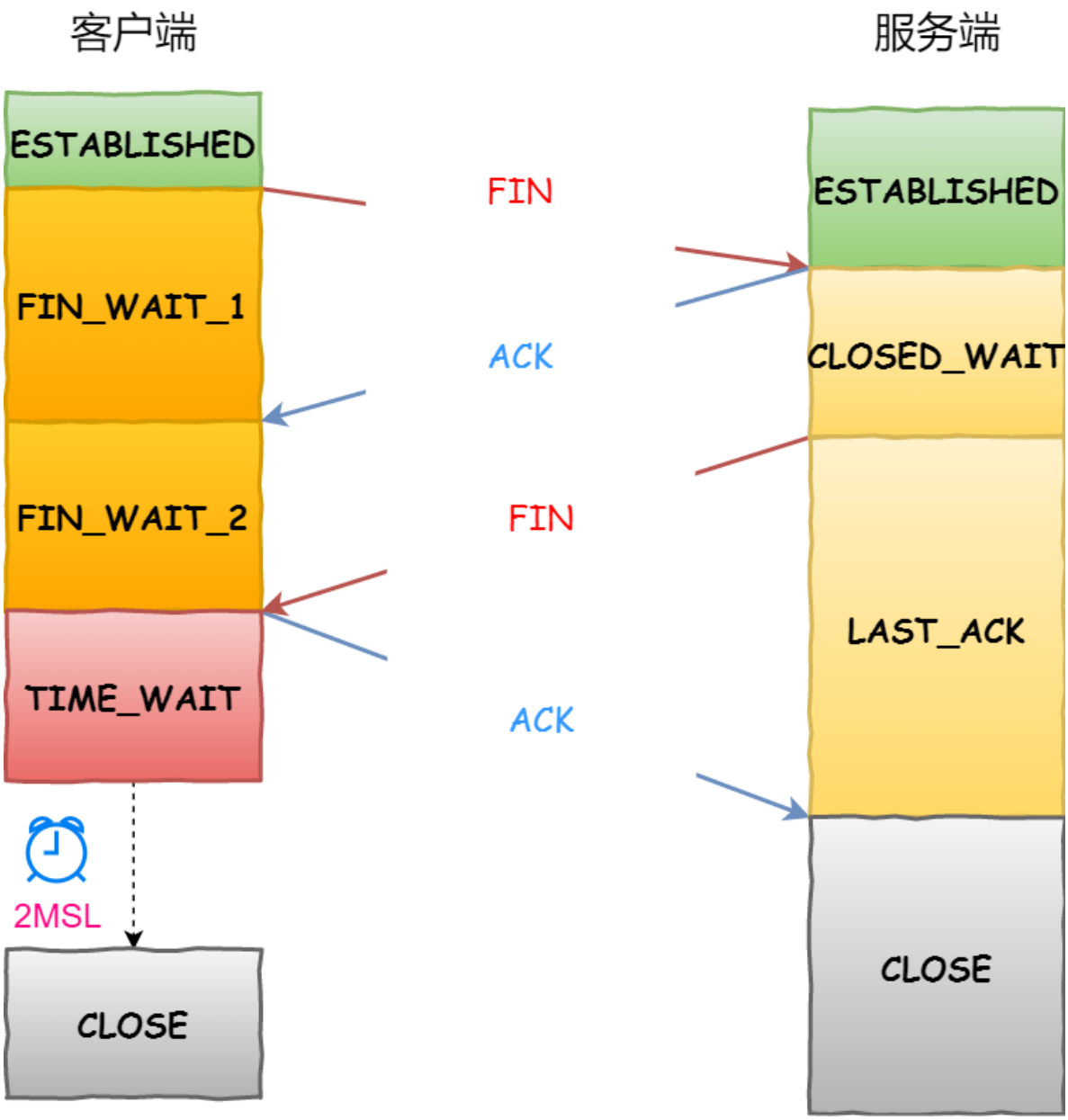
---

03 TCP 连接断开

> TCP 四次挥手过程和状态变迁

天下没有不散的宴席，对于 TCP 连接也是这样， TCP 断开连接是通过**四次挥手**方式。

双方都可以主动断开连接，断开连接后主机中的「资源」将被释放。



- 客户端打算关闭连接，此时会发送一个 TCP 首部 `FIN` 标志位被置为 `1` 的报文，也即 `FIN` 报文，之后客户端进入 `FIN_WAIT_1` 状态。
- 服务端收到该报文后，就向客户端发送 `ACK` 应答报文，接着服务端进入 `CLOSED_WAIT` 状态。
- 客户端收到服务端的 `ACK` 应答报文后，之后进入 `FIN_WAIT_2` 状态。
- 等待服务端处理完数据后，也向客户端发送 `FIN` 报文，之后服务端进入 `LAST_ACK` 状态。
- 客户端收到服务端的 `FIN` 报文后，回一个 `ACK` 应答报文，之后进入 `TIME_WAIT` 状态
- 服务器收到了 `ACK` 应答报文后，就进入了 `CLOSED` 状态，至此服务端已经完成连接的关闭。
- 客户端在经过 `2MSL` 一段时间后，自动进入 `CLOSED` 状态，至此客户端也完成连接的关闭。

你可以看到，每个方向都需要**一个 FIN 和一个 ACK**，因此通常被称为**四次挥手**。

这里一点需要注意是：**主动关闭连接的，才有 TIME_WAIT 状态。**

> 为什么挥手需要四次?

再来回顾下四次挥手双方发 `FIN` 包的过程，就能理解为什么需要四次了。

- 关闭连接时，客户端向服务端发送 `FIN` 时，仅仅表示客户端不再发送数据了但是还能接收数据。
- 服务器收到客户端的 `FIN` 报文时，先回一个 `ACK` 应答报文，而服务端可能还有数据需要处理和发送，等服务端不再发送数据时，才发送 `FIN` 报文给客户端来表示同意现在关闭连接。

从上面过程可知，服务端通常需要等待完成数据的发送和处理，所以服务端的 `ACK` 和 `FIN` 一般都会分开发送，从而比三次握手导致多了一次。

> 为什么 TIME_WAIT 等待的时间是 2MSL?

`MSL` 是 Maximum Segment Lifetime，**报文最大生存时间**，它是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。因为 TCP 报文基于是 IP 协议的，而 IP 头中有一个 `TTL` 字段，是 IP 数据报可以经过的最大路由数，每经过一个处理他的路由器此值就减 1，当此值为 0 则数据报将被丢弃，同时发送 ICMP 报文通知源主机。

MSL 与 TTL 的区别： MSL 的单位是时间，而 TTL 是经过路由跳数。所以 **MSL 应该要大于等于 TTL 消耗为 0 的时间**，以确保报文已被自然消亡。

TIME_WAIT 等待 2 倍的 MSL，比较合理的解释是： 网络中可能存在来自发送方的数据包，当这些发送方的数据包被接收方处理后又会向对方发送响应，所以**一来一回需要等待 2 倍的时间**。

比如如果被动关闭方没有收到断开连接的最后的 ACK 报文，就会触发超时重发 Fin 报文，另一方接收到 FIN 后，会重发 ACK 给被动关闭方， 一来一去正好 2 个 MSL。

`2MSL` 的时间是从**客户端接收到 FIN 后发送 ACK 开始计时的**。如果在 TIME-WAIT 时间内，因为客户端的 ACK 没有传输到服务端，客户端又接收到了服务端重发的 FIN 报文，那么 **2MSL 时间将重新计时**。

在 Linux 系统里 `2MSL` 默认是 `60` 秒，那么一个 `MSL` 也就是 `30` 秒。**Linux 系统停留在 TIME_WAIT 的时间为固定的 60 秒**。

其定义在 Linux 内核代码里的名称为 TCP_TIMEWAIT_LEN：

```
#define TCP_TIMEWAIT_LEN (60*HZ) /* how long to wait to destroy TIME-WAIT
                                    state, about 60 seconds  */
```

如果要修改 TIME_WAIT 的时间长度，只能修改 Linux 内核代码里 TCP_TIMEWAIT_LEN 的值，并重新编译 Linux 内核。

> 为什么需要 TIME_WAIT 状态?

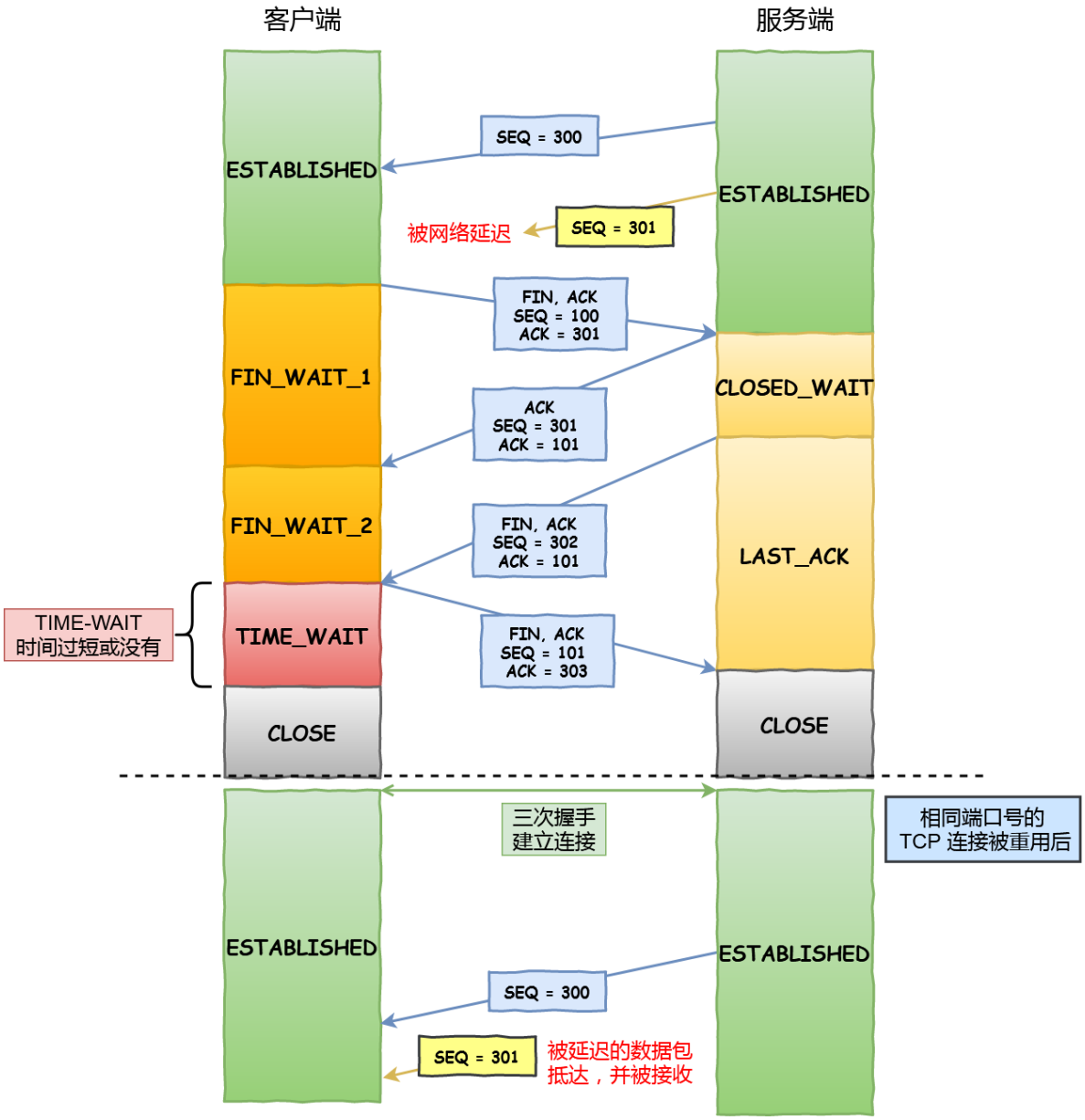
主动发起关闭连接的一方，才会有 `TIME-WAIT` 状态。

需要 TIME-WAIT 状态，主要是两个原因：

- 防止具有相同「四元组」的「旧」数据包被收到；
- 保证「被动关闭连接」的一方能被正确的关闭，即保证最后的 ACK 能让被动关闭方接收，从而帮助其正常关闭；

原因一：防止旧连接的数据包

假设 TIME-WAIT 没有等待时间或时间过短，被延迟的数据包抵达后会发生什么呢？



- 如上图黄色框框服务端在关闭连接之前发送的 `SEQ = 301` 报文，被网络延迟了。
- 这时有相同端口的 TCP 连接被复用后，被延迟的 `SEQ = 301` 抵达了客户端，那么客户端是有可能正常接收这个过期的报文，这就会产生数据错乱等严重的问题。

所以，TCP 就设计出了这么一个机制，经过 `2MSL` 这个时间，**足以让两个方向上的数据包都被丢弃，使得原来连接的数据包在网络中都自然消失，再出现的数据包一定都是新建立连接所产生的。**

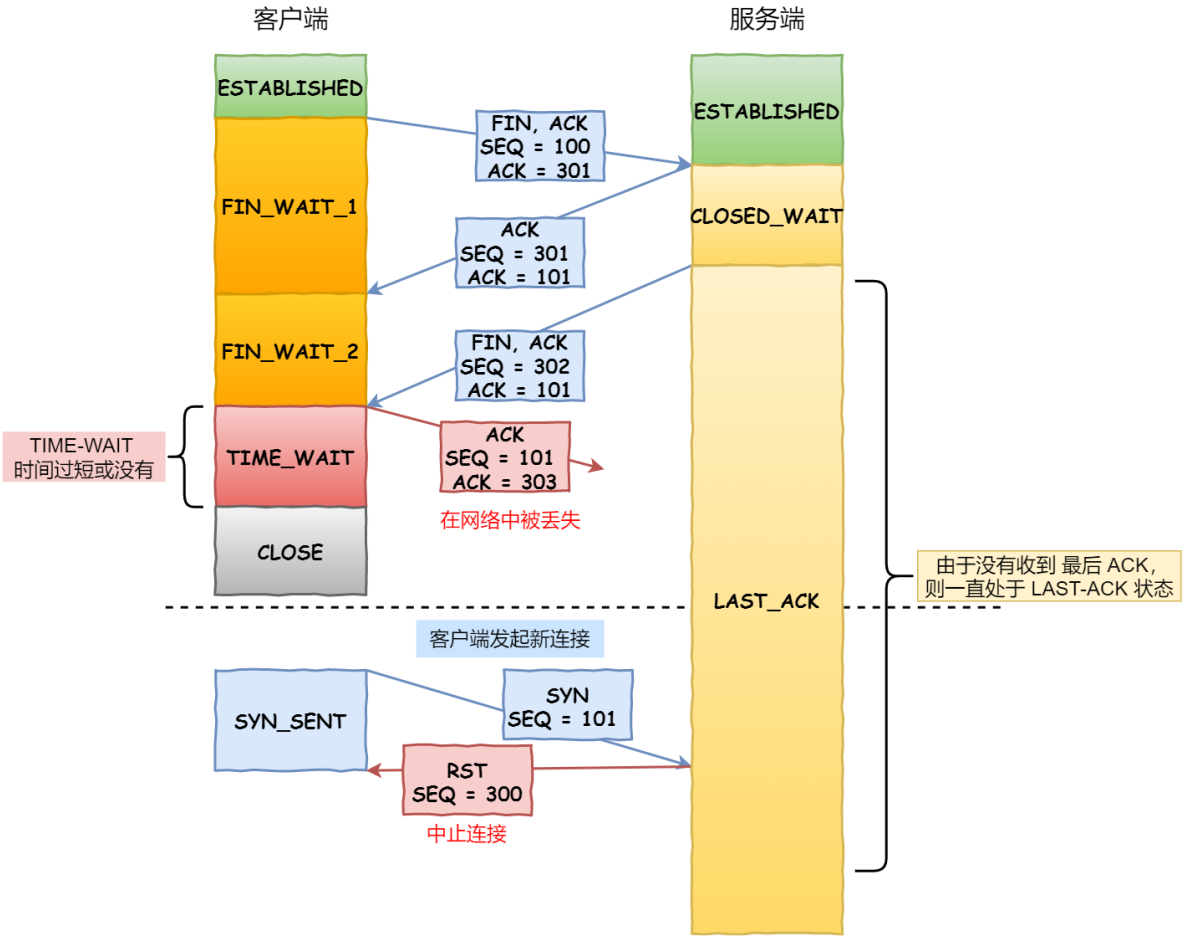
原因二：保证连接正确关闭

在 RFC 793 指出 TIME-WAIT 另一个重要的作用是：

TIME-WAIT - represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

也就是说，TIME-WAIT 作用是**等待足够的时间以确保最后的 ACK 能让被动关闭方接收，从而帮助其正常关闭。**

假设 TIME-WAIT 没有等待时间或时间过短，断开连接会造成什么问题呢？



- 如上图红色框框客户端四次挥手的最后一个 `ACK` 报文如果在网络中被丢失了，此时如果客户端 `TIME-WAIT` 过短或没有，则就直接进入了 `CLOSED` 状态了，那么服务端则会一直处在 `LASE_ACK` 状态。
- 当客户端发起建立连接的 `SYN` 请求报文后，服务端会发送 `RST` 报文给客户端，连接建立的过程就会被终止。

如果 TIME-WAIT 等待足够长的情况就会遇到两种情况：

- 服务端正常收到四次挥手的最后一个 `ACK` 报文，则服务端正常关闭连接。
- 服务端没有收到四次挥手的最后一个 `ACK` 报文时，则会重发 `FIN` 关闭连接报文并等待新的 `ACK` 报文。

所以客户端在 `TIME-WAIT` 状态等待 `2MSL` 时间后，就可以**保证双方的连接都可以正常的关闭。**

> TIME_WAIT 过多有什么危害？

如果服务器有处于 TIME-WAIT 状态的 TCP，则说明是由服务器方主动发起的断开请求。

过多的 TIME-WAIT 状态主要的危害有两种：

- 第一是内存资源占用；
- 第二是对端口资源的占用，一个 TCP 连接至少消耗一个本地端口；

第二个危害是会造成严重的后果的，要知道，端口资源也是有限的，一般可以开启的端口为 `32768～61000`，也可以通过如下参数设置指定

```
net.ipv4.ip_local_port_range
```

如果发起连接一方的 TIME_WAIT 状态过多，占满了所有端口资源，则会导致无法创建新连接。

客户端受端口资源限制：

- 客户端TIME_WAIT过多，就会导致端口资源被占用，因为端口就65536个，被占满就会导致无法创建新的连接。

服务端受系统资源限制：

- 由于一个四元组表示 TCP 连接，理论上服务端可以建立很多连接，服务端确实只监听一个端口 但是会把连接扔给处理线程，所以理论上监听的端口可以继续监听。但是线程池处理不了那么多一直不断的连接了。所以当服务端出现大量 TIME_WAIT 时，系统资源被占满时，会导致处理不过来新的连接。

> 如何优化 TIME_WAIT?

这里给出优化 TIME-WAIT 的几个方式，都是有利有弊：

- 打开 net.ipv4.tcp_tw_reuse 和 net.ipv4.tcp_timestamps 选项；
- net.ipv4.tcp_max_tw_buckets
- 程序中使用 SO_LINGER ，应用强制使用 RST 关闭。

方式一：net.ipv4.tcp_tw_reuse 和 tcp_timestamps

如下的 Linux 内核参数开启后，则可以**复用处于 TIME_WAIT 的 socket 为新的连接所用**。

有一点需要注意的是，**tcp_tw_reuse 功能只能用客户端（连接发起方），因为开启了该功能，在调用 connect() 函数时，内核会随机找一个 time_wait 状态超过 1 秒的连接给新的连接复用。**

```
net.ipv4.tcp_tw_reuse = 1
```

使用这个选项，还有一个前提，需要打开对 TCP 时间戳的支持，即

```
net.ipv4.tcp_timestamps=1（默认即为 1）
```

这个时间戳的字段是在 TCP 头部的「选项」里，用于记录 TCP 发送方的当前时间戳和从对端接收到的最新时间戳。

由于引入了时间戳，我们在前面提到的 `2MSL` 问题就不复存在了，因为重复的数据包会因为时间戳过期被自然丢弃。

方式二：net.ipv4.tcp_max_tw_buckets

这个值默认为 18000，当系统中处于 TIME_WAIT 的连接**一旦超过这个值时，系统就会将所有的 TIME_WAIT 连接状态重置。**

这个方法过于暴力，而且治标不治本，带来的问题远比解决的问题多，不推荐使用。

方式三：程序中使用 SO_LINGER

我们可以通过设置 socket 选项，来设置调用 close 关闭连接行为。

```
struct linger so_linger;
so_linger.l_onoff = 1;
so_linger.l_linger = 0;
setsockopt(s, SOL_SOCKET, SO_LINGER, &so_linger,sizeof(so_linger));
```

如果 `l_onoff` 为非 0， 且 `l_linger` 值为 0，那么调用 `close` 后，会立该发送一个 `RST` 标志给对端，该 TCP 连接将跳过四次挥手，也就跳过了 `TIME_WAIT` 状态，直接关闭。

但这为跨越 `TIME_WAIT` 状态提供了一个可能，不过是一个非常危险的行为，不值得提倡。

> 如果已经建立了连接，但是客户端突然出现故障了怎么办?

TCP 有一个机制是**保活机制**。这个机制的原理是这样的：

定义一个时间段，在这个时间段内，如果没有任何连接相关的活动，TCP 保活机制会开始作用，每隔一个时间间隔，发送一个探测报文，该探测报文包含的数据非常少，如果连续几个探测报文都没有得到响应，则认为当前的 TCP 连接已经死亡，系统内核将错误信息通知给上层应用程序。

在 Linux 内核可以有对应的参数可以设置保活时间、保活探测的次数、保活探测的时间间隔，以下都为默认值：

```
net.ipv4.tcp_keepalive_time=7200
net.ipv4.tcp_keepalive_intvl=75
net.ipv4.tcp_keepalive_probes=9
```

- tcp_keepalive_time=7200：表示保活时间是 7200 秒（2小时），也就 2 小时内如果没有任何连接相关的活动，则会启动保活机制
- tcp_keepalive_intvl=75：表示每次检测间隔 75 秒；
- tcp_keepalive_probes=9：表示检测 9 次无响应，认为对方是不可达的，从而中断本次的连接。

也就是说在 Linux 系统中，最少需要经过 2 小时 11 分 15 秒才可以发现一个「死亡」连接。

```
tcp_keepalive_time +  (tcp_keepalive_intvl * tcp_keepalive_probes)
```



```
7200 +  (75 * 9) = 7875 秒（ 2 小时 11 分 15 秒）
```

这个时间是有点长的，我们也可以根据实际的需求，对以上的保活相关的参数进行设置。

如果开启了 TCP 保活，需要考虑以下几种情况：

第一种，对端程序是正常工作的。当 TCP 保活的探测报文发送给对端, 对端会正常响应，这样 **TCP 保活时间会被重置**，等待下一个 TCP 保活时间的到来。

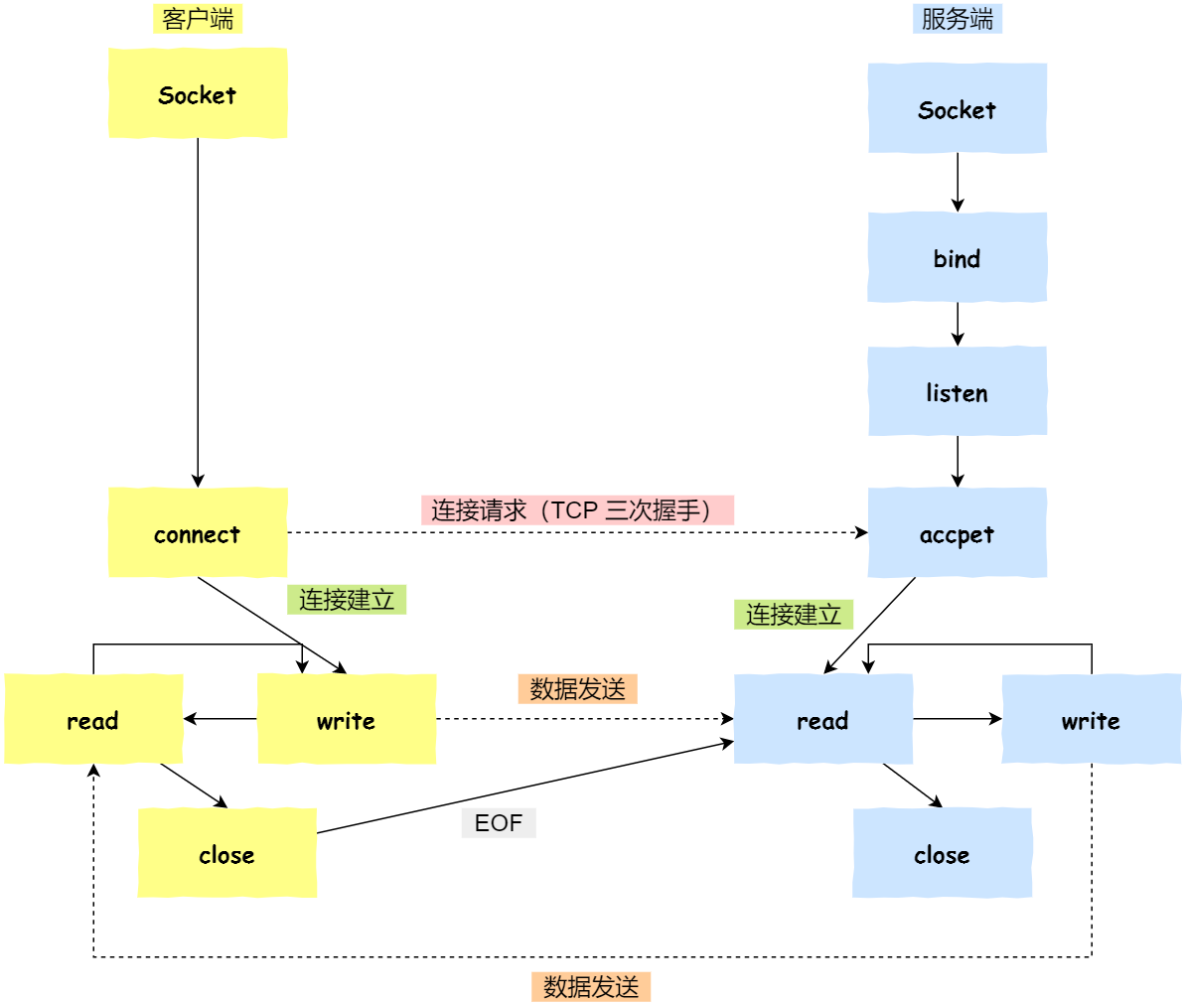
第二种，对端程序崩溃并重启。当 TCP 保活的探测报文发送给对端后，对端是可以响应的，但由于没有该连接的有效信息，**会产生一个 RST 报文**，这样很快就会发现 TCP 连接已经被重置。

第三种，是对端程序崩溃，或对端由于其他原因导致报文不可达。当 TCP 保活的探测报文发送给对端后，石沉大海，没有响应，连续几次，达到保活探测次数后，**TCP 会报告该 TCP 连接已经死亡**。

---

04 Socket 编程

> 针对 TCP 应该如何 Socket 编程?



- 服务端和客户端初始化 `socket`，得到文件描述符；
- 服务端调用 `bind`，将绑定在 IP 地址和端口;
- 服务端调用 `listen`，进行监听；
- 服务端调用 `accept`，等待客户端连接；
- 客户端调用 `connect`，向服务器端的地址和端口发起连接请求；
- 服务端 `accept` 返回用于传输的 `socket` 的文件描述符；
- 客户端调用 `write` 写入数据；服务端调用 `read` 读取数据；
- 客户端断开连接时，会调用 `close`，那么服务端 `read` 读取数据的时候，就会读取到了 `EOF`，待处理完数据后，服务端调用 `close`，表示连接关闭。

这里需要注意的是，服务端调用 `accept` 时，连接成功了会返回一个已完成连接的 socket，后续用来传输数据。

所以，监听的 socket 和真正用来传送数据的 socket，是「两个」 socket，一个叫作**监听 socket**，一个叫作**已完成连接 socket**。

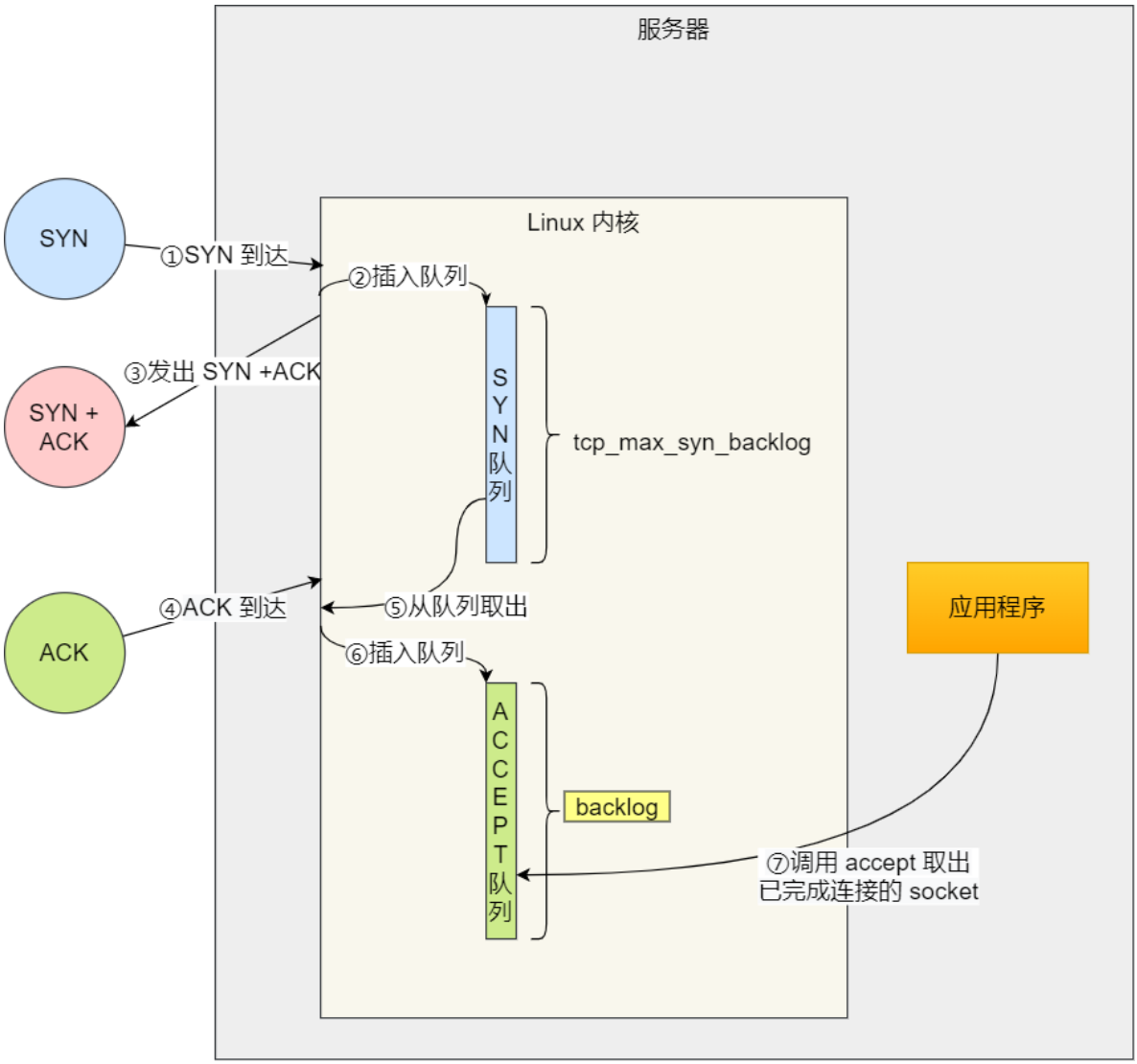
成功连接建立之后，双方开始通过 read 和 write 函数来读写数据，就像往一个文件流里面写东西一样。

> listen 时候参数 backlog 的意义?

Linux内核中会维护两个队列：

- 未完成连接队列（SYN 队列）：接收到一个 SYN 建立连接请求，处于 SYN_RCVD 状态；

- 已完成连接队列（Accpet 队列）：已完成 TCP 三次握手过程，处于 ESTABLISHED 状态；




```
int listen (int socketfd, int backlog)
```

- 参数一 socketfd 为 socketfd 文件描述符
- 参数二 backlog，这参数在历史版本有一定的变化

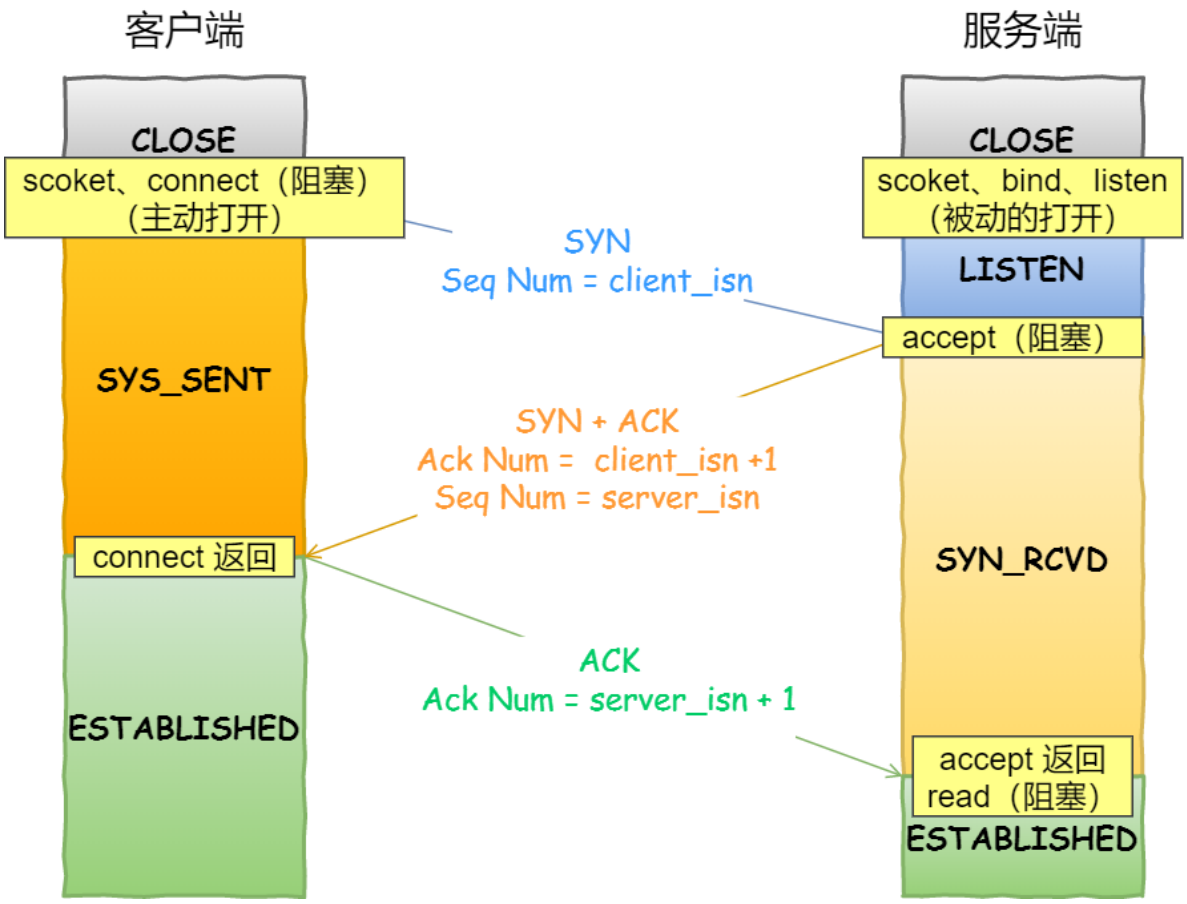
在早期 Linux 内核 backlog 是 SYN 队列大小，也就是未完成的队列大小。

在 Linux 内核 2.2 之后，backlog 变成 accept 队列，也就是已完成连接建立的队列长度，**所以现在通常认为 backlog 是 accept 队列。**

但是上限值是内核参数 somaxconn 的大小，也就说 accpet 队列长度 = min(backlog, somaxconn)。

> accept 发生在三次握手的哪一步？

我们先看看客户端连接服务端时，发送了什么？

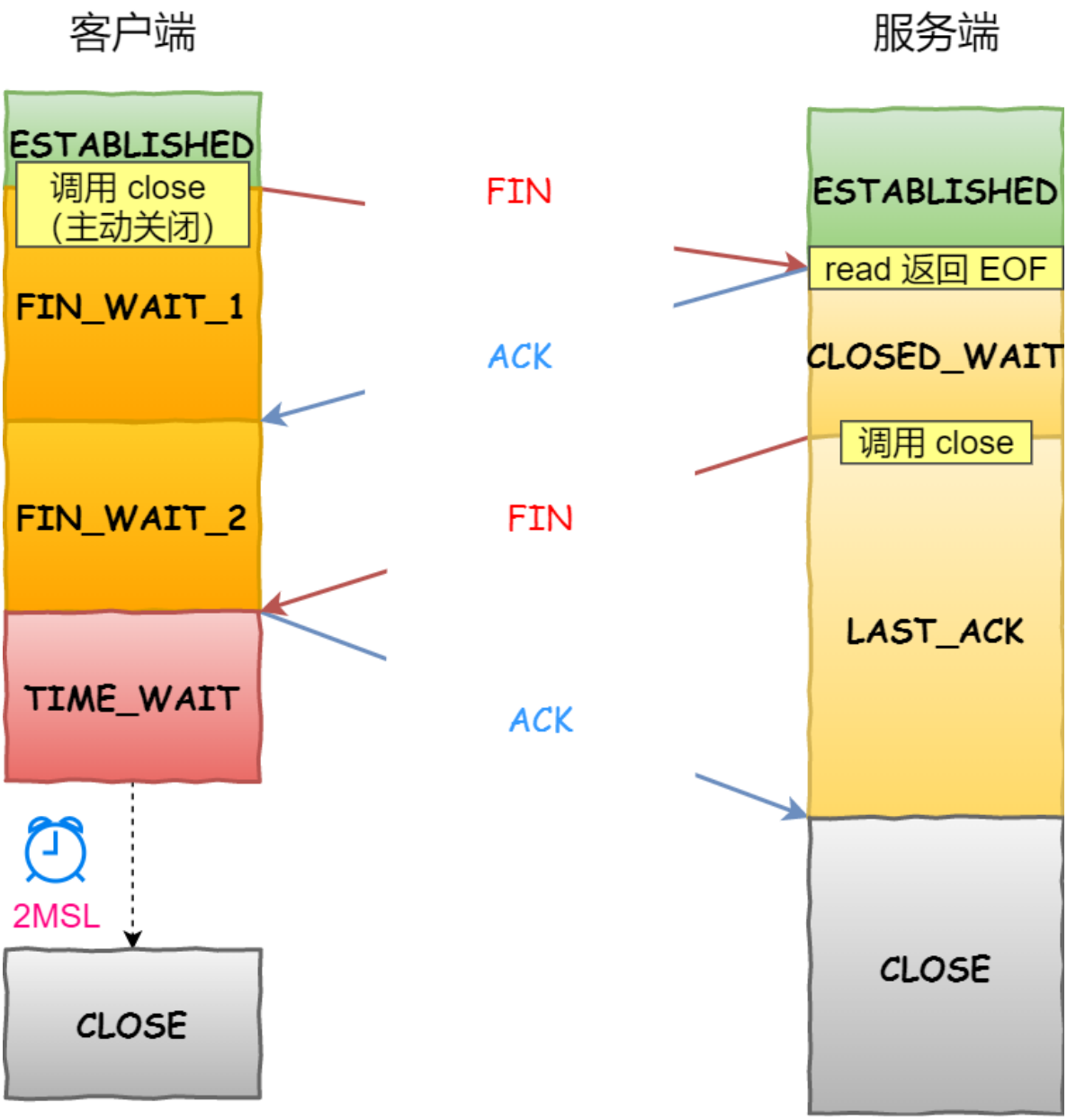


- 客户端的协议栈向服务器端发送了 SYN 包，并告诉服务器端当前发送序列号 client_isn，客户端进入 SYN_SENT 状态；
- 服务器端的协议栈收到这个包之后，和客户端进行 ACK 应答，应答的值为 client_isn+1，表示对 SYN 包 client_isn 的确认，同时服务器也发送一个 SYN 包，告诉客户端当前我的发送序列号为 server_isn，服务器端进入 SYN_RCVD 状态；
- 客户端协议栈收到 ACK 之后，使得应用程序从 `connect` 调用返回，表示客户端到服务器端的单向连接建立成功，客户端的状态为 ESTABLISHED，同时客户端协议栈也会对服务器端的 SYN 包进行应答，应答数据为 server_isn+1；
- 应答包到达服务器端后，服务器端协议栈使得 `accept` 阻塞调用返回，这个时候服务器端到客户端的单向连接也建立成功，服务器端也进入 ESTABLISHED 状态。

从上面的描述过程，我们可以得知**客户端 connect 成功返回是在第二次握手，服务端 accept 成功返回是在三次握手成功之后。**

> 客户端调用 close 了，连接是断开的流程是什么？

我们看看客户端主动调用了 `close`，会发生什么？



- 客户端调用 `close`，表明客户端没有数据需要发送了，则此时会向服务端发送 FIN 报文，进入 FIN_WAIT_1 状态；
- 服务端接收到了 FIN 报文，TCP 协议栈会为 FIN 包插入一个文件结束符 `EOF` 到接收缓冲区中，应用程序可以通过 `read` 调用来感知这个 FIN 包。这个 `EOF` 会被**放在已排队等候的其他已接收的数据之后**，这就意味着服务端需要处理这种异常情况，因为 EOF 表示在该连接上再无额外数据到达。此时，服务端进入 CLOSE_WAIT 状态；
- 接着，当处理完数据后，自然就会读到 `EOF`，于是也调用 `close` 关闭它的套接字，这会使得客户端会发出一个 FIN 包，之后处于 LAST_ACK 状态；
- 客户端接收到服务端的 FIN 包，并发送 ACK 确认包给服务端，此时客户端将进入 TIME_WAIT 状态；
- 服务端收到 ACK 确认包后，就进入了最后的 CLOSE 状态；
- 客户端经过 `2MSL` 时间之后，也进入 CLOSE 状态；

巨人的肩膀

[1] 趣谈网络协议专栏.刘超.极客时间.

[2] 网络编程实战专栏.盛延敏.极客时间.

[3] 计算机网络-自顶向下方法.陈鸣 译.机械工业出版社

[4] TCP/IP详解 卷1：协议.范建华 译.机械工业出版社

[5] 图解TCP/IP.竹下隆史.人民邮电出版社

[6] https://www.rfc-editor.org/rfc/rfc793.html

[7] https://draveness.me/whys-the-design-tcp-three-way-handshake

[8] https://draveness.me/whys-the-design-tcp-time-wait/

唠叨唠叨

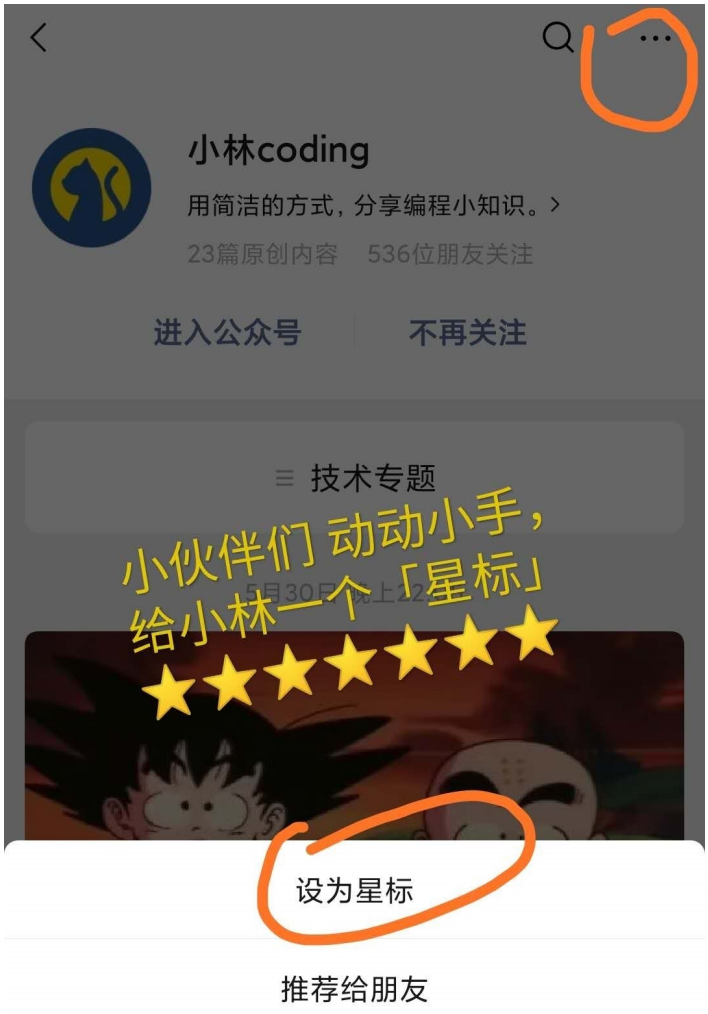
小林为写此文重学了一篇 TCP，深感 TCP 真的是一个非常复杂的协议，要想轻易拿下，也不是一天两天的事，所以小林花费了一个星期多才写完此文章。

正所谓知道的越多，不知道的也越多。

下篇给大家带来 TCP 滑动窗口、流量控制、拥塞控制的图解文章！

本文只是抛砖引玉，若你有更好的想法或文章有误的地方，欢迎留言讨论！

小林是专为大家图解的工具人，Goodbye，我们下次见！



扫一扫
关注爱图解的
「小林coding」

读者问答

> 读者问：“关于文中三次握手最主要的原因，有一个疑问：为什么新包和旧包的 seq 会不一样？查阅了tcp/ip 详解 卷1，以及一些其他网络书籍和 RFC793 部分，都没有明确说明关于第一个 SYN 如果发生重传会改变 seq。”

文章的例子不是超时重发的 SYN 报文，而是新产生的一个 SYN 报文，所以 seq 是不一样的。

我文章的例子是 RFC 793： https://www.rfc-editor.org/rfc/rfc793.html , 33 页的 Figure 9。

> 读者问：“请教个问题，为了方便调试服务器程序，一般会在服务端设置 SO_REUSEADDR 选项，这样服务器程序在重启后，可以立刻使用。这里设置SO_REUSEADDR 是不是就等价于对这个 socket 设置了内核中的 net.ipv4.tcp_tw_reuse=1 这个选项？”

这两个东西没有关系的哦。

1. tcp_tw_reuse 是内核选项，主要用在连接的发起方（客户端）。TIME_WAIT 状态的连接创建时间超过 1 秒后，新的连接才可以被复用，注意，这里是「连接的发起方」；
2. SO_REUSEADDR 是用户态的选项，用于「连接的服务方」，用来告诉操作系统内核，如果端口已被占用，但是 TCP 连接状态位于 TIME_WAIT ，可以重用端口。如果端口忙，而 TCP 处于其他状态，重用会有 “Address already in use” 的错误信息。

tcp_tw_reuse 是为了缩短 time_wait 的时间，避免出现大量的 time_wait 连接而占用系统资源，解决的是 accept 后的问题。

SO_REUSEADDR 是为了解决 time_wait 状态带来的端口占用问题，以及支持同一个 port 对应多个 ip，解决的是 bind 时的问题。

> 读者问：“请教一下，如果客户端第四次挥手ack丢失，服务端超时重发的fin报文也丢失，客户端timewait时间超过了2msl，这个时候会发生什么？认为连接已经关闭吗？”

当客户端 timewait 时间超过了 2MSL，则客户端就直接进入关闭状态。

服务端超时重发 fin 报文的次数如果超过 tcp_orphan_retries 大小后，服务端也会关闭 TCP 连接。

> 读者问：“求教两个小问题：文章在解释IP分片和TCP MSS分片时说，如果用IP分片会有两个问题：（1）IP按MTU分片，如果某一片丢失则需要所有分片都重传；（2）IP没有重传机制，所以需要等TCP发送方超时才能重传；问题一：MSS跟IP的MTU分片相比，只是多了一步协商MSS值的过程，而IP的MTU可以看作是默认协商好就是1500字节，所以为什么协商后的MSS可以做到丢失后只发丢失的这一片来提高效率，而默认协商好1500字节的IP分片就需要所有片都重传呢？问题二：TCP MSS分片如果丢失了一片，是不是也需要发送方等待超时再重传？如果不是，MSS的协商如何能在超时前就直到丢了分片从而提高效率的呢？谢谢老师。”

问题一：

- 如果一个大的 TCP 报文是被 MTU 分片，那么只有「第一个分片」才具有 TCP 头部，后面的分片则没有 TCP 头部，接收方 IP 层只有重组了这些分片，才会认为是一个 TCP 报文，那么丢失了其中一个分片，接收方 IP 层就不会把 TCP 报文丢给 TCP 层，那么就会等待对方超时重传这一整个 TCP 报文。
- 如果一个大的 TCP 报文被 MSS 分片，那么所有「分片都具有 TCP 头部」，因为每个 MSS 分片的是具有 TCP 头部的TCP报文，那么其中一个 MSS 分片丢失，就只需要重传这一个分片就可以。

问题二：

- TCP MSS分片如果丢失了一片，发送方没收到对方ACK应答，也是会触发超时重传的，因为TCP层是会保证数据的可靠交付。

> 读者问：“大佬，请教个问题，如果是服务提供方发起的 close ，然后引起过多的 time_wait 状态的 tcp 链接，time_wait 会影响服务端的端口吗？谢谢。”

不会。

如果发起连接一方（客户端）的 TIME_WAIT 状态过多，占满了所有端口资源，则会导致无法创建新连接。

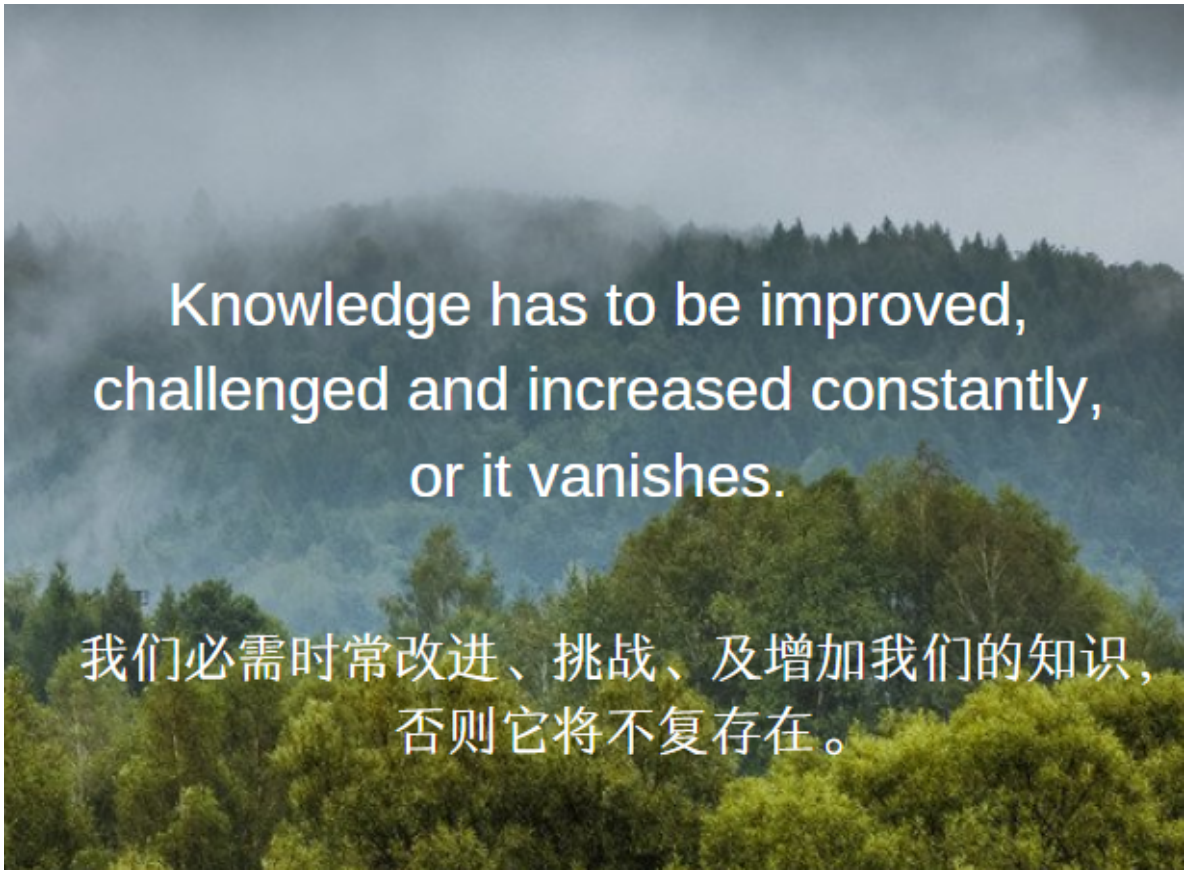
客户端受端口资源限制：

- 客户端TIME_WAIT过多，就会导致端口资源被占用，因为端口就65536个，被占满就会导致无法创建新连接。

服务端受系统资源限制：

- 由于一个 TCP 四元组表示 TCP 连接，理论上服务端可以建立很多连接，服务端只监听一个端口，但是会把连接扔给处理线程，所以理论上监听的端口可以继续监听。但是线程池处理不了那么多一直不断的连接了。所以当服务端出现大量 TIMEWAIT 时，系统资源容易被耗尽。

你还在为 TCP 重传、滑动窗口、流量控制、拥塞控制发愁吗？看完图解就不愁了



前言

前一篇「硬不硬你说了算！近 40 张图解被问千百遍的 TCP 三次握手和四次挥手面试题」得到了很多读者的认可，在此特别感谢你们的认可，大家都暖暖的。

来了，今天又来图解 TCP 了，**小林可能会迟到，但不会缺席。**

迟到的原因，主要是 TCP **巨复杂**，它为了保证可靠性，用了巨多的机制来保证，真是个「伟大」的协议，写着写着发现这水太深了。。。

本文的全部图片都是小林绘画的，非常的辛苦且累，不废话了，直接进入正文，Go！

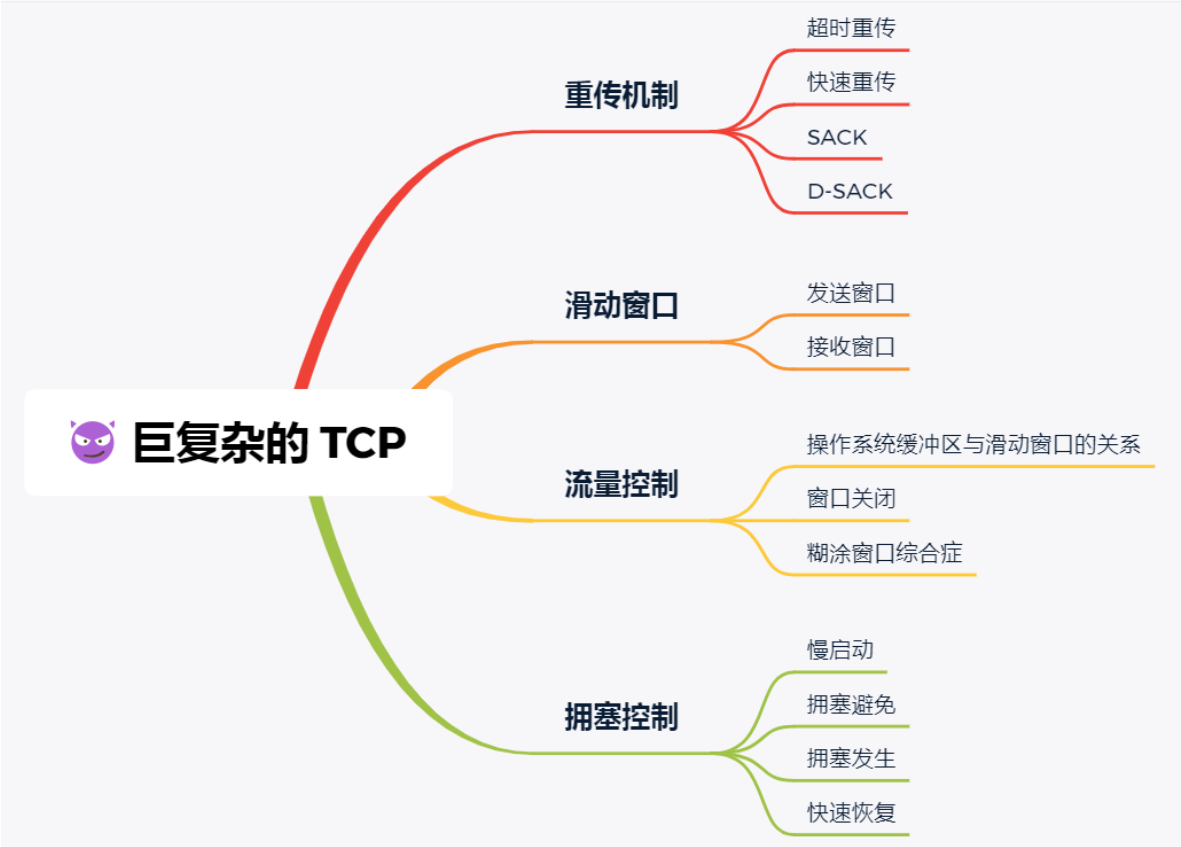
正文

相信大家都知道 TCP 是一个可靠传输的协议，那它是如何保证可靠的呢？

为了实现可靠性传输，需要考虑很多事情，例如数据的破坏、丢包、重复以及分片顺序混乱等问题。如不能解决这些问题，也就无从谈起可靠传输。

那么，TCP 是通过序列号、确认应答、重发控制、连接管理以及窗口控制等机制实现可靠性传输的。

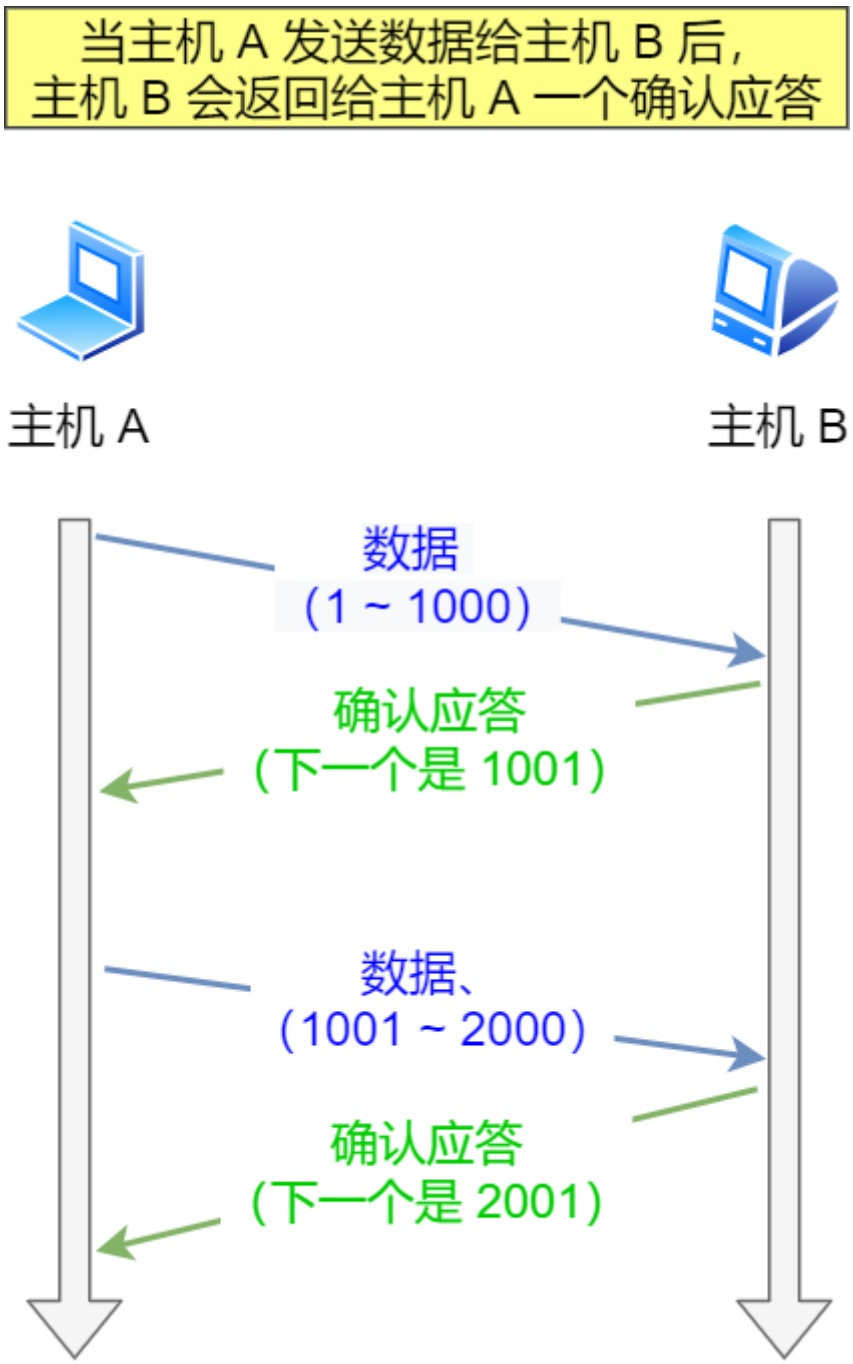
今天，将重点介绍 TCP 的**重传机制、滑动窗口、流量控制、拥塞控制。**



重传机制

TCP 实现可靠传输的方式之一，是通过序列号与确认应答。

在 TCP 中，当发送端的数据到达接收主机时，接收端主机会返回一个确认应答消息，表示已收到消息。



但在错综复杂的网络，并不一定能如上图那么顺利能正常的数据传输，万一数据在传输过程中丢失了呢？

所以 TCP 针对数据包丢失的情况，会用**重传机制**解决。

接下来说说常见的重传机制：

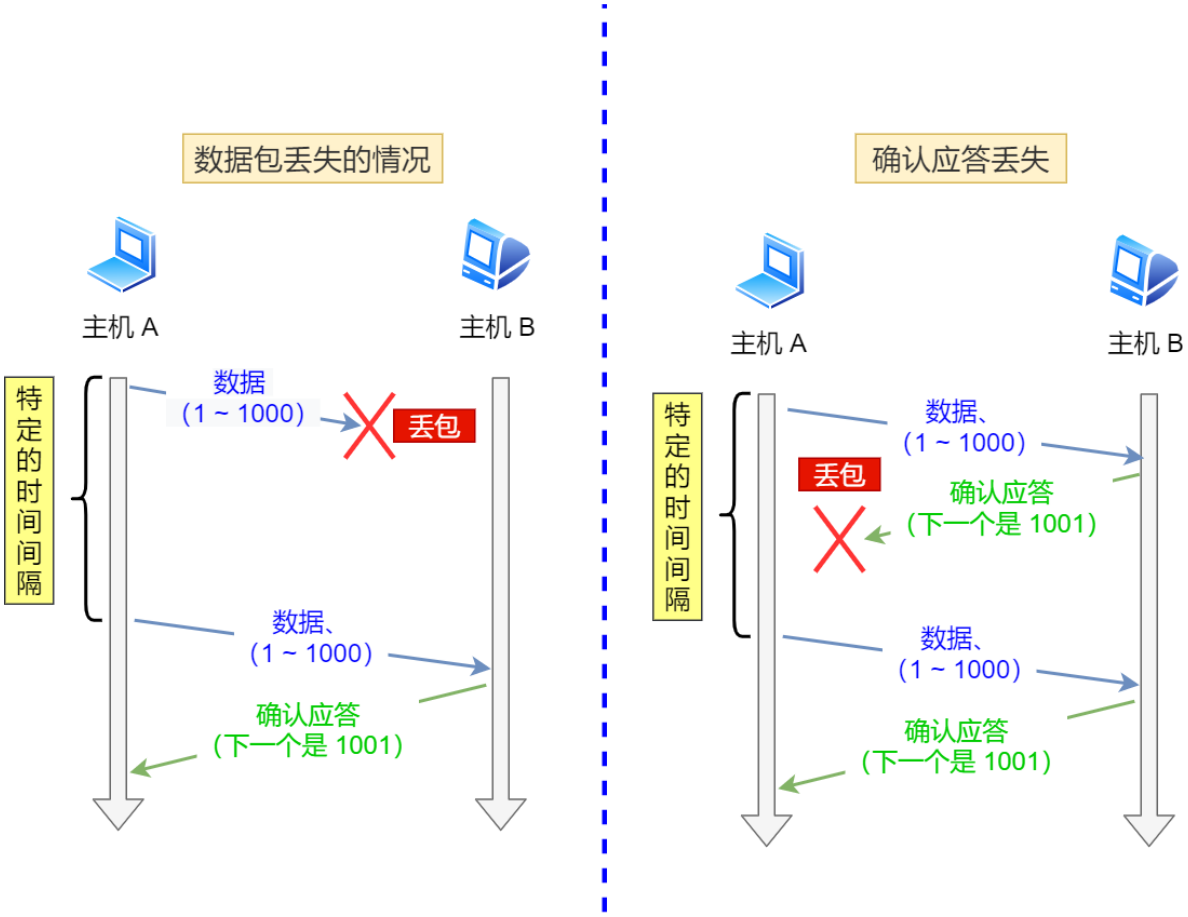
- 超时重传
- 快速重传
- SACK
- D-SACK

超时重传

重传机制的其中一个方式，就是在发送数据时，设定一个定时器，当超过指定的时间后，没有收到对方的 `ACK` 确认应答报文，就会重发该数据，也就是我们常说的**超时重传**。

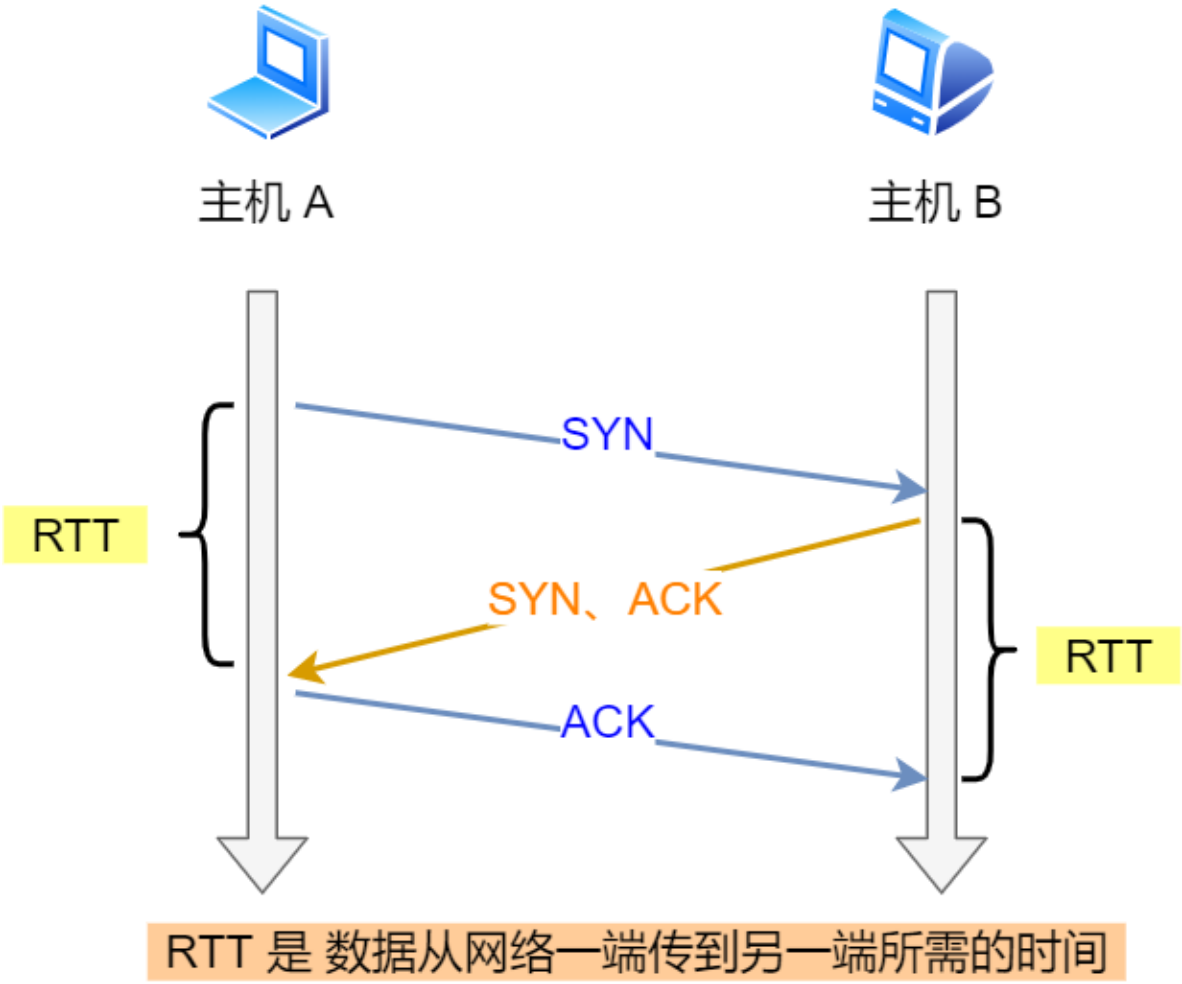
TCP 会在以下两种情况发生超时重传：

- 数据包丢失
- 确认应答丢失



> 超时时间应该设置为多少呢？

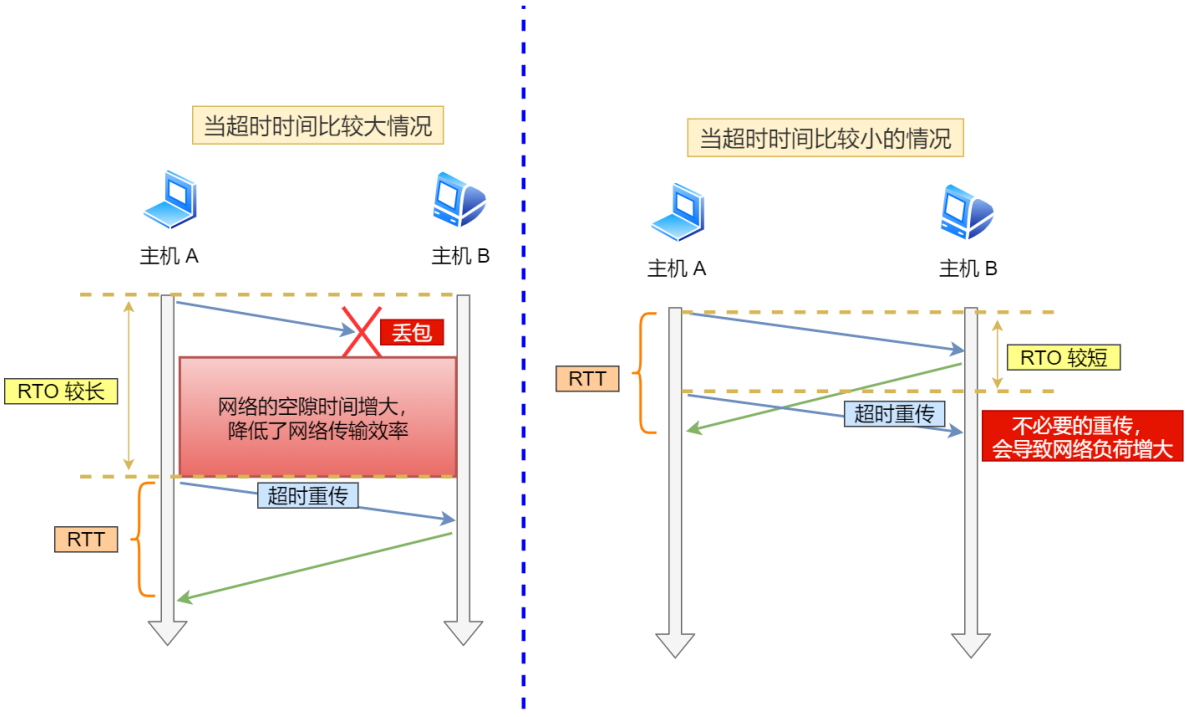
我们先来了解一下什么是 `RTT` （Round-Trip Time 往返时延），从下图我们就可以知道：



RTT 就是**数据从网络一端传送到另一端所需的时间**，也就是包的往返时间。

超时重传时间是以 RTO （Retransmission Timeout 超时重传时间）表示。

假设在重传的情况下，超时时间 RTO 「较长或较短」时，会发生什么事情呢？

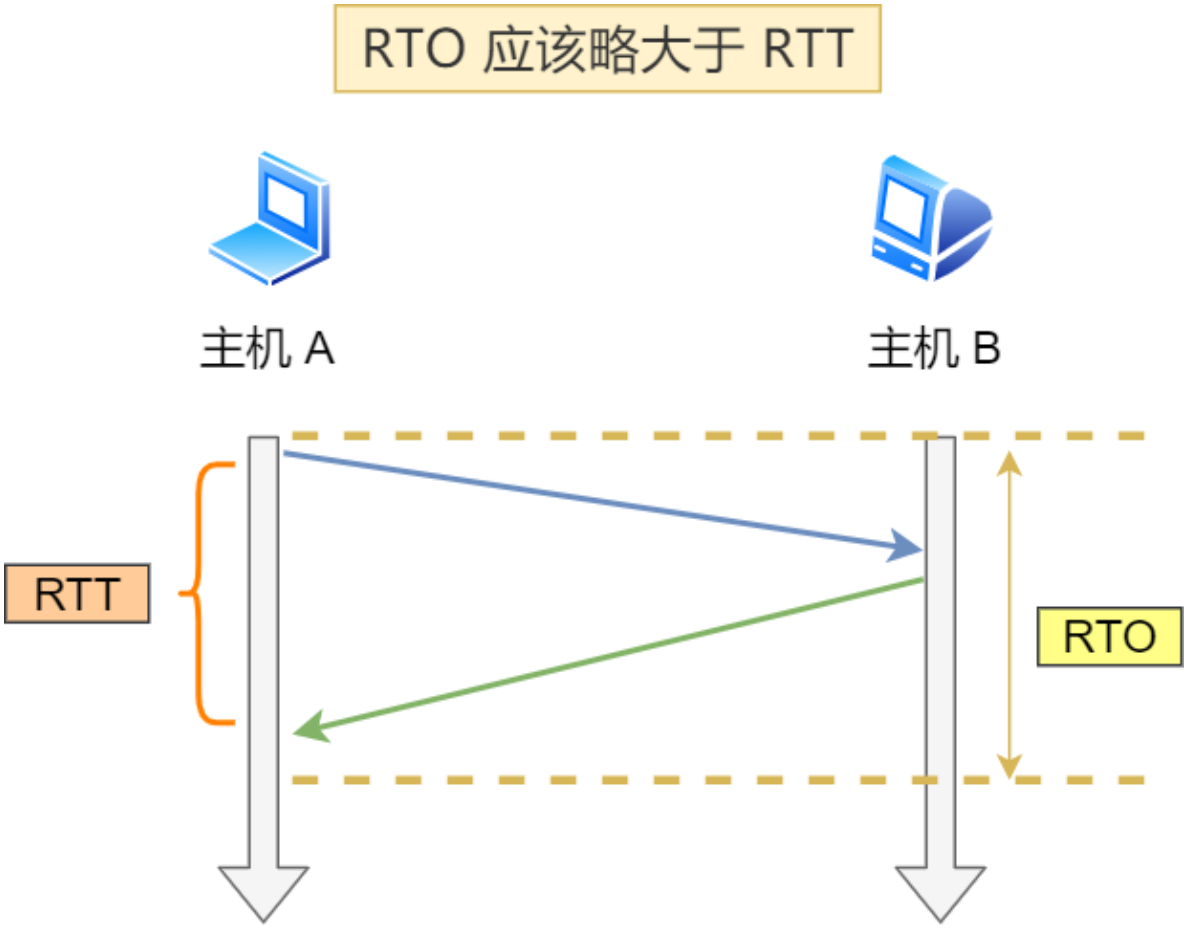


上图中有两种超时时间不同的情况：

- 当超时时间 **RTO 较大**时，重发就慢，丢了老半天才重发，没有效率，性能差；
- 当超时时间 **RTO 较小**时，会导致可能并没有丢就重发，于是重发的就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

精确的测量超时时间 `RTO` 的值是非常重要的，这可让我们的重传机制更高效。

根据上述的两种情况，我们可以得知，**超时重传时间 RTO 的值应该略大于报文往返 RTT 的值**。



至此，可能大家觉得超时重传时间 `RTO` 的值计算，也不是很复杂嘛。

好像就是在发送端发包时记下 `t0` ，然后接收端再把这个 `ack` 回来时再记一个 `t1` ，于是 `RTT = t1 – t0` 。没那么简单，**这只是一个采样，不能代表普遍情况**。

实际上「报文往返 RTT 的值」是经常变化的，因为我们的网络也是时常变化的。也就因为「报文往返 RTT 的值」 是经常波动变化的，所以「超时重传时间 RTO 的值」应该是一个**动态变化的值**。

我们来看看 Linux 是如何计算 `RTO` 的呢?

估计往返时间，通常需要采样以下两个：

- 需要 TCP 通过采样 RTT 的时间，然后进行加权平均，算出一个平滑 RTT 的值，而且这个值还是要不断变化的，因为网络状况不断地变化。
- 除了采样 RTT，还要采样 RTT 的波动范围，这样就避免如果 RTT 有一个大的波动的话，很难被发现的情况。

RFC6289 建议使用以下的公式计算 RTO：

① 首次计算 RTO ， 其中 R1 为第一次测量的 RTT

$$SRTT = R1$$

$$DevRTT = R1/2$$

$$RTO = \mu * SRTT + \partial * DevRT = \mu * R1 + \partial * (R1/2)$$

② 后续计算 RTO ， 其中 R2 为最新测量的 RTT

$$SRTT = SRTT + \alpha (RTT - SRTT) = R1 + \alpha * (R2 - R1)$$

$$DevRTT = (1-\beta)*DevRTT + \beta *(|RTT-SRTT|) =(1-\beta) * (R1/2) + \beta * (|R2 - R1|)$$

$$RTO = \mu * SRTT + \partial * DevRTT$$

其中 SRTT 是计算平滑的RTT， DevRTR 是计算平滑的RTT 与 最新 RTT 的差距。

在 Linux 下，$\alpha = 0.125$，$\beta = 0.25$， $\mu = 1$，$\partial = 4$。别问怎么来的，问就是大量实验中调出来的。

如果超时重发的数据，再次超时的时候，又需要重传的时候，TCP 的策略是**超时间隔加倍。**

也就是**每当遇到一次超时重传的时候，都会将下一次超时时间间隔设为先前值的两倍。两次超时，就说明网络环境差，不宜频繁反复发送。**

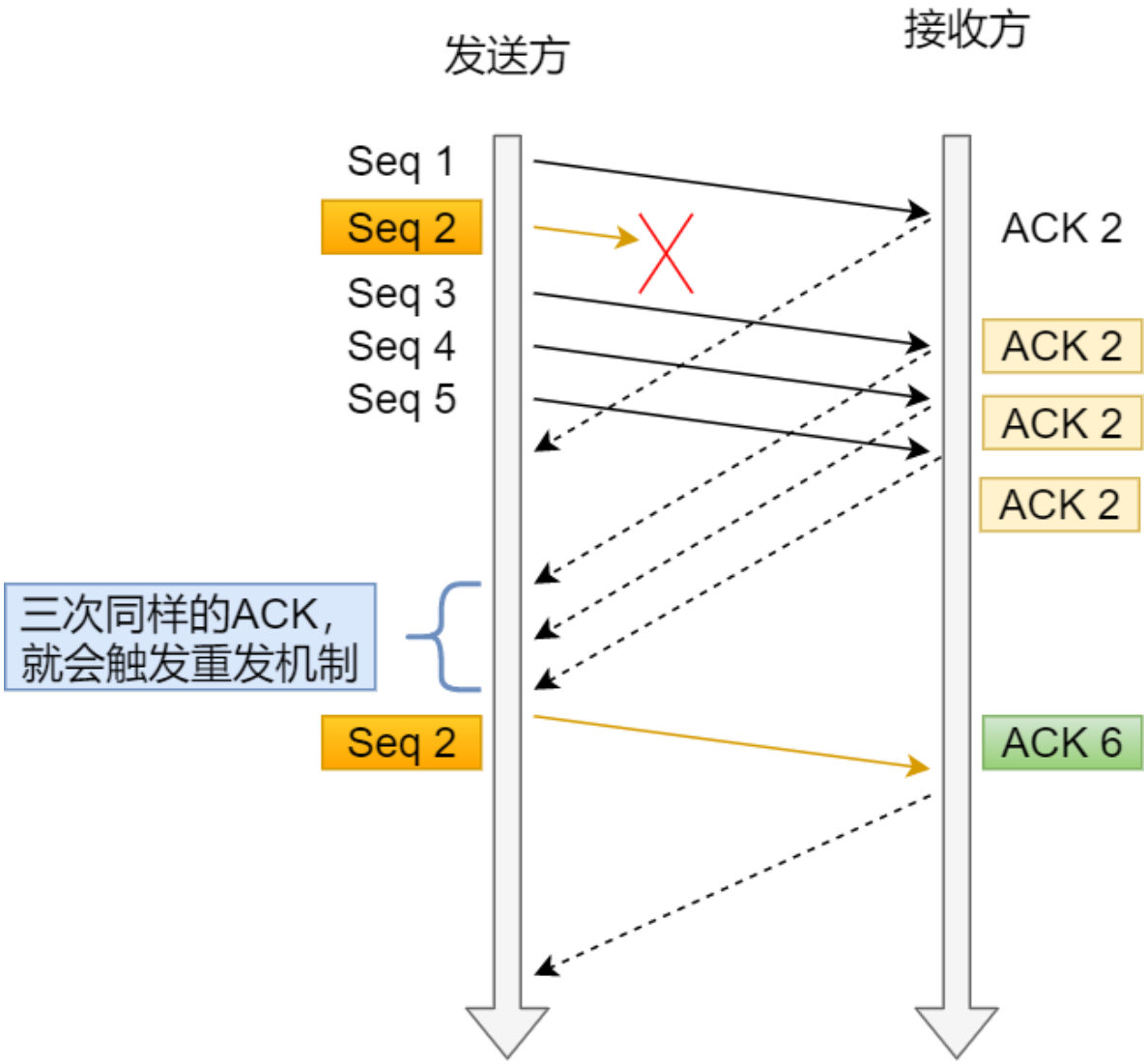
超时触发重传存在的问题是，超时周期可能相对较长。那是不是可以有更快的方式呢?

于是就可以用「快速重传」机制来解决超时重发的时间等待。

快速重传

TCP 还有另外一种**快速重传（Fast Retransmit）机制**，它**不以时间为驱动，而是以数据驱动重传**。

快速重传机制，是如何工作的呢？其实很简单，一图胜千言。



在上图，发送方发出了 1，2，3，4，5 份数据：

- 第一份 Seq1 先送到了，于是就 Ack 回 2；
- 结果 Seq2 因为某些原因没收到，Seq3 到达了，于是还是 Ack 回 2；
- 后面的 Seq4 和 Seq5 都到了，但还是 Ack 回 2，因为 Seq2 还是没有收到；
- **发送端收到了三个 Ack = 2 的确认，知道了 Seq2 还没有收到，就会在定时器过期之前，重传丢失的 Seq2。**
- 最后，收到了 Seq2，此时因为 Seq3，Seq4，Seq5 都收到了，于是 Ack 回 6 。

所以，快速重传的工作方式是当收到三个相同的 ACK 报文时，会在定时器过期之前，重传丢失的报文段。

快速重传机制只解决了一个问题，就是超时时间的问题，但是它依然面临着另外一个问题。就是**重传的时候，是重传之前的一个，还是重传所有的问题。**

比如对于上面的例子，是重传 Seq2 呢？还是重传 Seq2、Seq3、Seq4、Seq5 呢？因为发送端并不清楚这连续的三个 Ack 2 是谁传回来的。

根据 TCP 不同的实现，以上两种情况都是有可能的。可见，这是一把双刃剑。

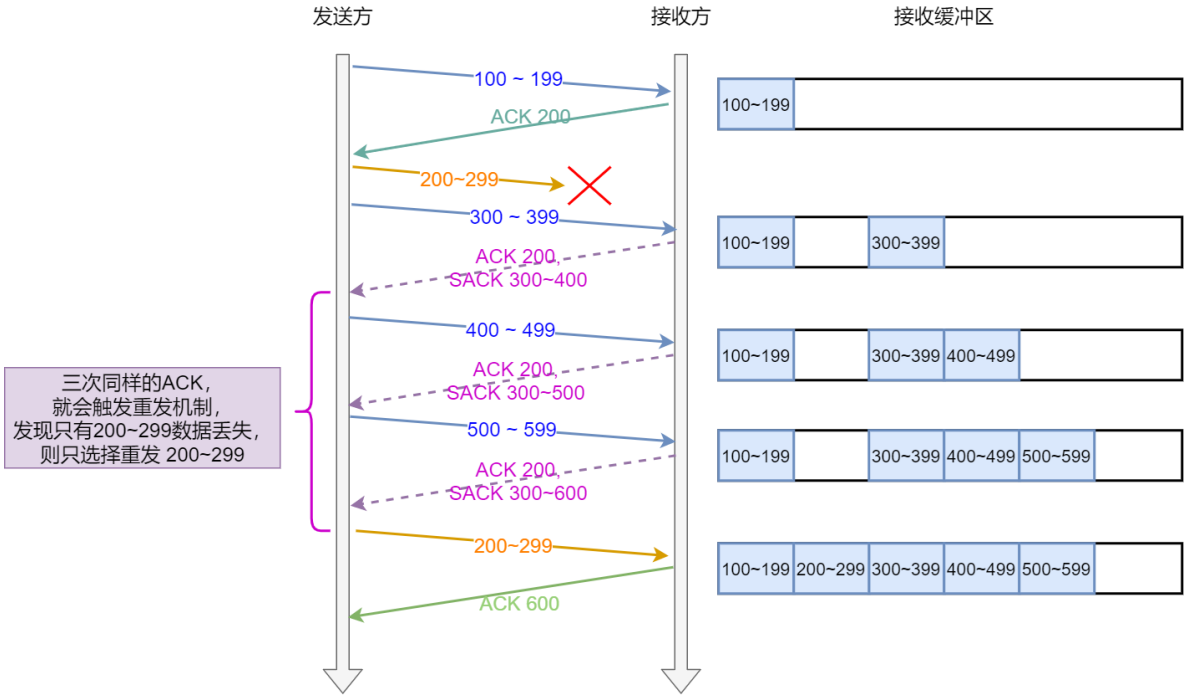
为了解决不知道该重传哪些 TCP 报文，于是就有 `SACK` 方法。

SACK 方法

还有一种实现重传机制的方式叫：`SACK`（Selective Acknowledgment 选择性确认）。

这种方式需要在 TCP 头部「选项」字段里加一个 `SACK` 的东西，它**可以将缓存的地图发送给发送方**，这样发送方就可以知道哪些数据收到了，哪些数据没收到，知道了这些信息，就可以**只重传丢失的数据**。

如下图，发送方收到了三次同样的 ACK 确认报文，于是就会触发快速重发机制，通过 `SACK` 信息发现只有 `200~299` 这段数据丢失，则重发时，就只选择了这个 TCP 段进行重复。



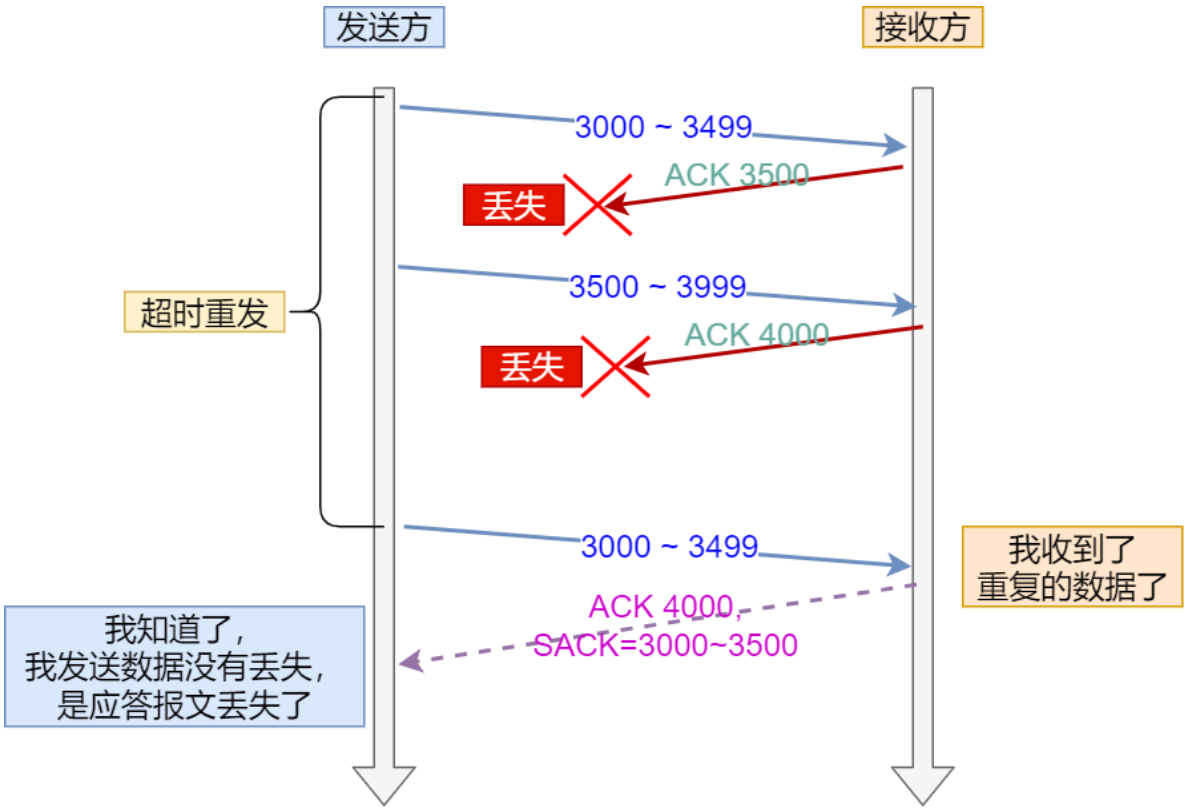
如果要支持 `SACK`，必须双方都要支持。在 Linux 下，可以通过 `net.ipv4.tcp_sack` 参数打开这个功能（Linux 2.4 后默认打开）。

Duplicate SACK

Duplicate SACK 又称 `D-SACK`，其主要**使用了 SACK 来告诉「发送方」有哪些数据被重复接收了。**

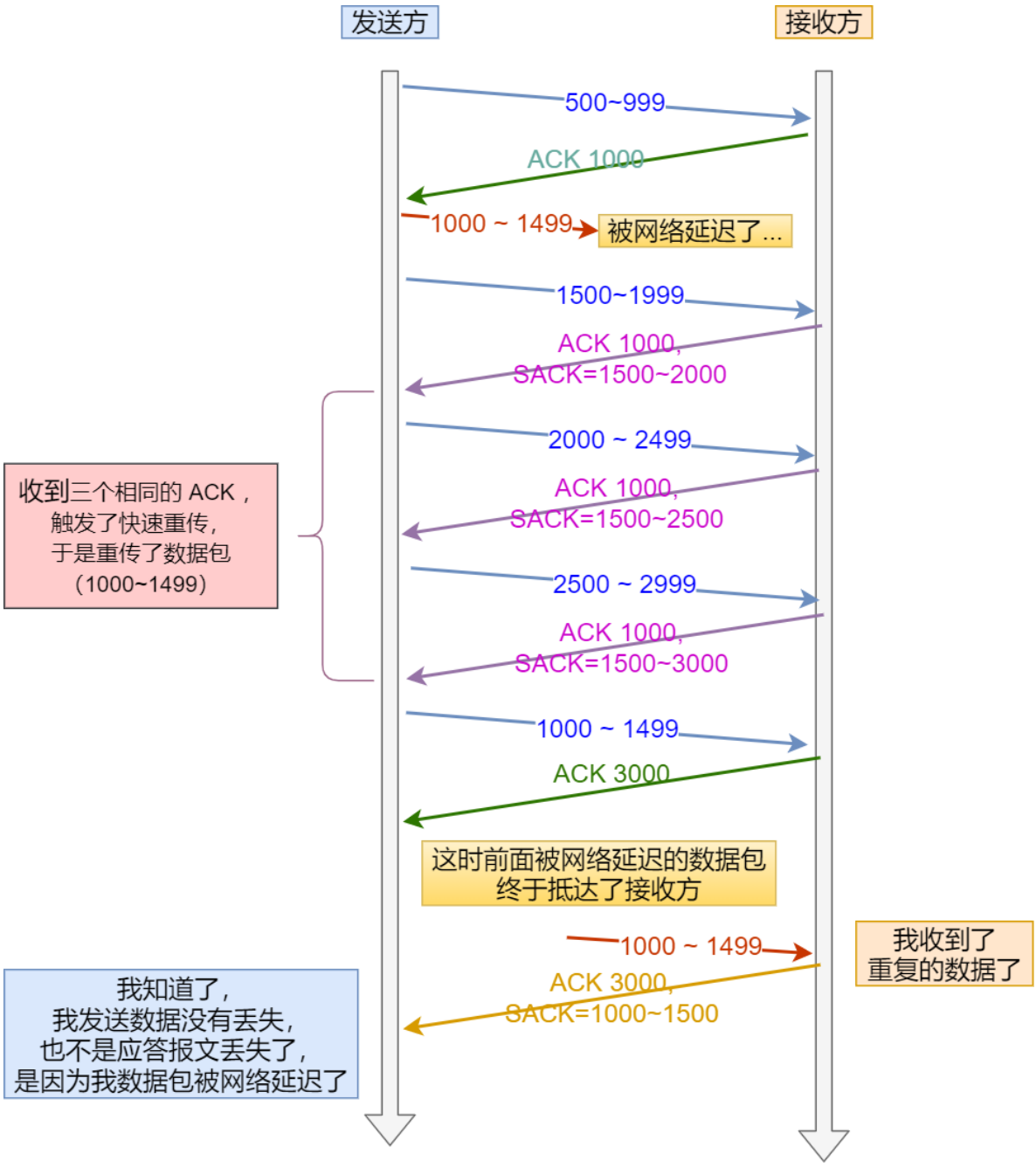
下面举例两个栗子，来说明 `D-SACK` 的作用。

栗子一号：ACK 丢包



- 「接收方」发给「发送方」的两个 ACK 确认应答都丢失了，所以发送方超时后，重传第一个数据包（3000 ~ 3499）
- **于是「接收方」发现数据是重复收到的，于是回了一个 SACK = 3000~3500**，告诉「发送方」3000~3500 的数据早已被接收了，因为 ACK 都到了 4000 了，已经意味着 4000 之前的所有数据都已收到，所以这个 SACK 就代表着 D-SACK 。
- 这样「发送方」就知道了，数据没有丢，是「接收方」的 ACK 确认报文丢了。

栗子二号：网络延时



- 数据包（1000~1499） 被网络延迟了，导致「发送方」没有收到 Ack 1500 的确认报文。
- 而后面报文到达的三个相同的 ACK 确认报文，就触发了快速重传机制，但是在重传后，被延迟的数据包（1000~1499）又到了「接收方」；
- **所以「接收方」回了一个 SACK=1000~1500，因为 ACK 已经到了 3000，所以这个 SACK 是 D-SACK，表示收到了重复的包。**
- 这样发送方就知道快速重传触发的原因不是发出去的包丢了，也不是因为回应的 ACK 包丢了，而是因为网络延迟了。

可见， D-SACK 有这么几个好处：

1. 可以让「发送方」知道，是发出去的包丢了，还是接收方回应的 ACK 包丢了;
2. 可以知道是不是「发送方」的数据包被网络延迟了;
3. 可以知道网络中是不是把「发送方」的数据包给复制了;

在 Linux 下可以通过 net.ipv4.tcp_dsack 参数开启/关闭这个功能（Linux 2.4 后默认打开）。

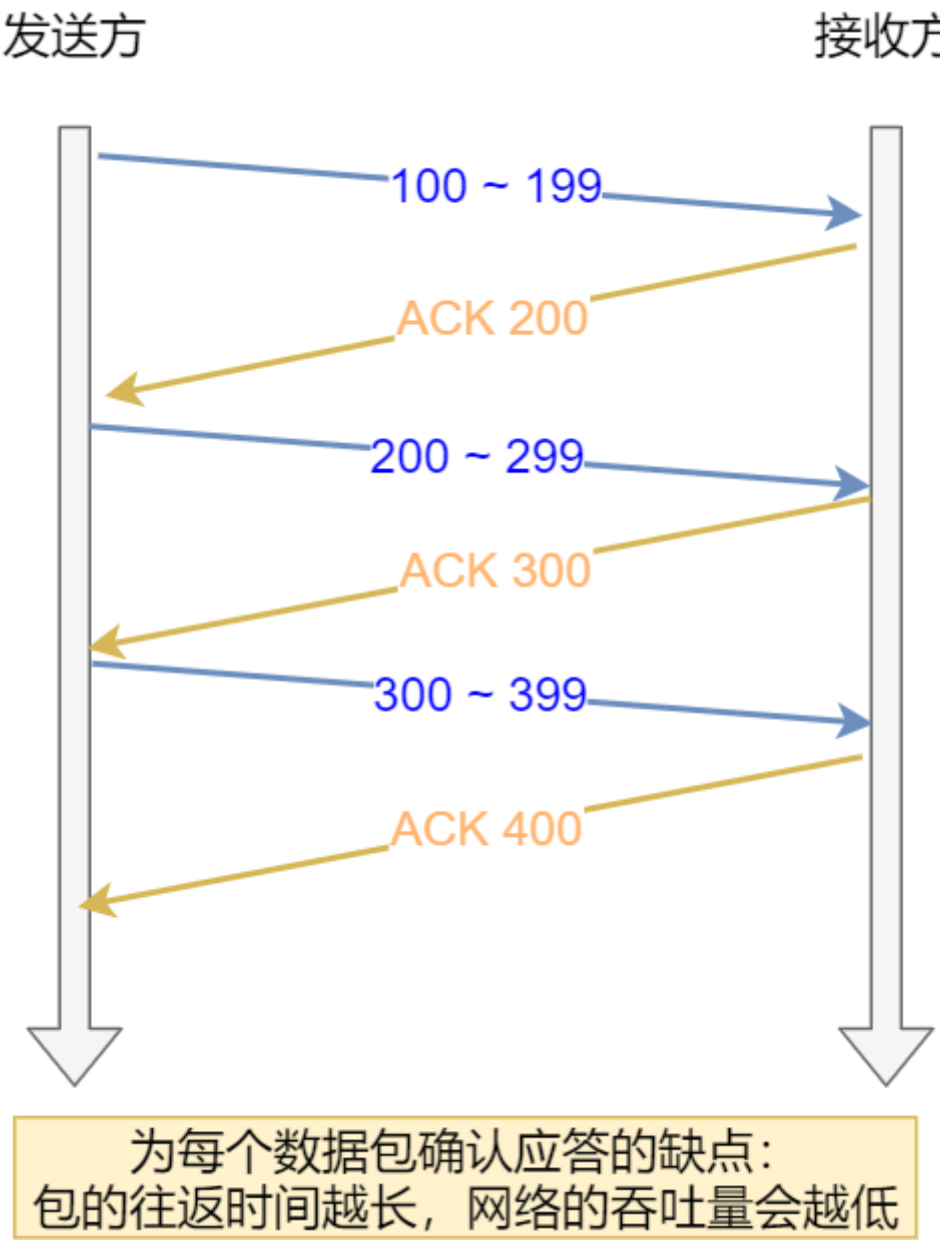
滑动窗口

> 引入窗口概念的原因

我们都知道 TCP 是每发送一个数据，都要进行一次确认应答。当上一个数据包收到了应答了， 再发送下一个。

这个模式就有点像我和你面对面聊天，你一句我一句。但这种方式的缺点是效率比较低的。

如果你说完一句话，我在处理其他事情，没有及时回复你，那你不是要干等着我做完其他事情后，我回复你，你才能说下一句话，很显然这不现实。



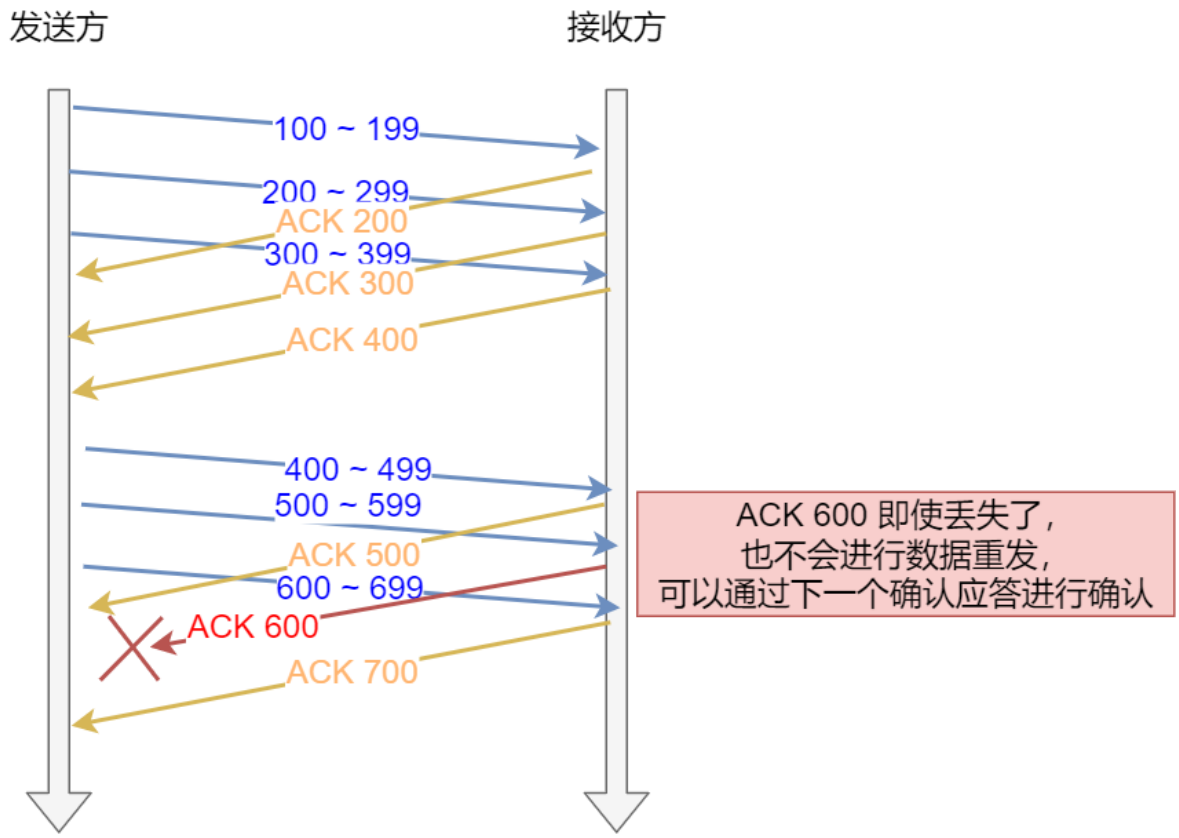
所以，这样的传输方式有一个缺点：数据包的**往返时间越长，通信的效率就越低**。

为解决这个问题，TCP 引入了**窗口**这个概念。即使在往返时间较长的情况下，它也不会降低网络通信的效率。

那么有了窗口，就可以指定窗口大小，窗口大小就是指**无需等待确认应答，而可以继续发送数据的最大值**。

窗口的实现实际上是操作系统开辟的一个缓存空间，发送方主机在等到确认应答返回之前，必须在缓冲区中保留已发送的数据。如果按期收到确认应答，此时数据就可以从缓存区清除。

假设窗口大小为 `3` 个 TCP 段，那么发送方就可以「连续发送」 `3` 个 TCP 段，并且中途若有 ACK 丢失，可以通过「下一个确认应答进行确认」。如下图：



图中的 ACK 600 确认应答报文丢失，也没关系，因为可以通过下一个确认应答进行确认，只要发送方收到了 ACK 700 确认应答，就意味着 700 之前的所有数据「接收方」都收到了。这个模式就叫**累计确认**或者**累计应答**。

> 窗口大小由哪一方决定?

TCP 头里有一个字段叫 `Window` ，也就是窗口大小。

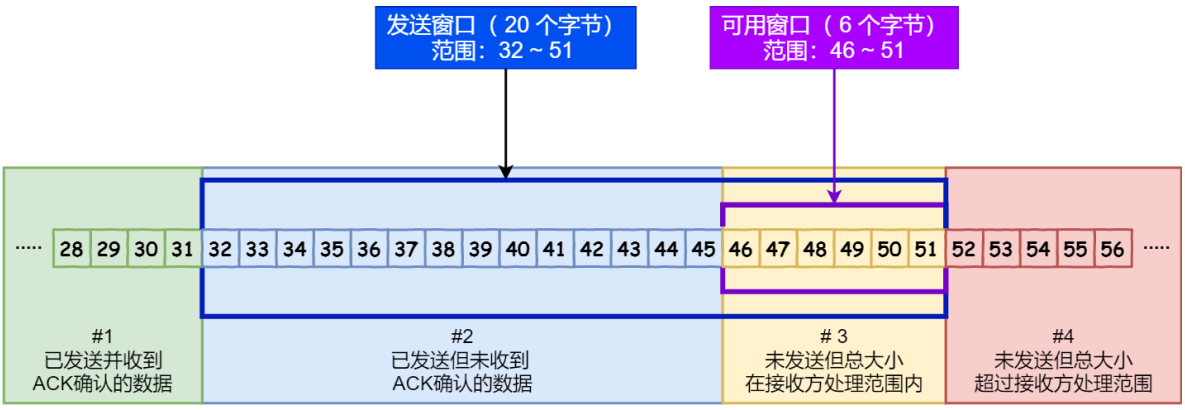
这个字段是接收端告诉发送端自己还有多少缓冲区可以接收数据。于是发送端就可以根据这个接收端的处理能力来发送数据，而不会导致接收端处理不过来。

所以，通常窗口的大小是由接收方的窗口大小来决定的。

发送方发送的数据大小不能超过接收方的窗口大小，否则接收方就无法正常接收到数据。

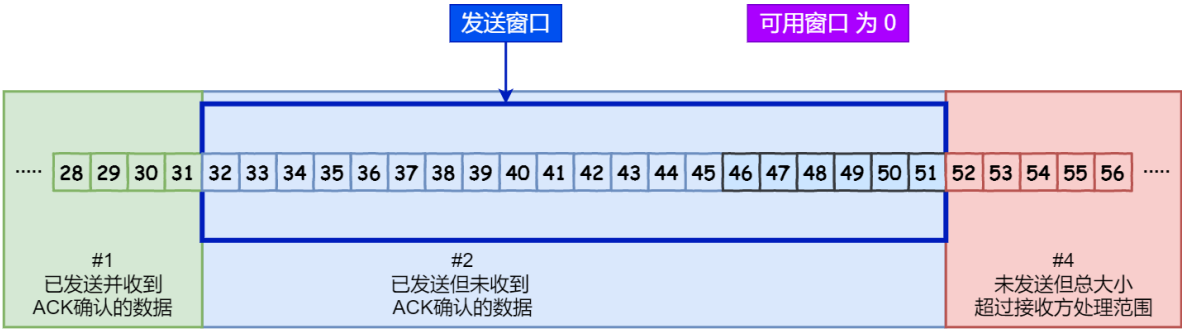
> 发送方的滑动窗口

我们先来看看发送方的窗口，下图就是发送方缓存的数据，根据处理的情况分成四个部分，其中深蓝色方框是发送窗口，紫色方框是可用窗口：

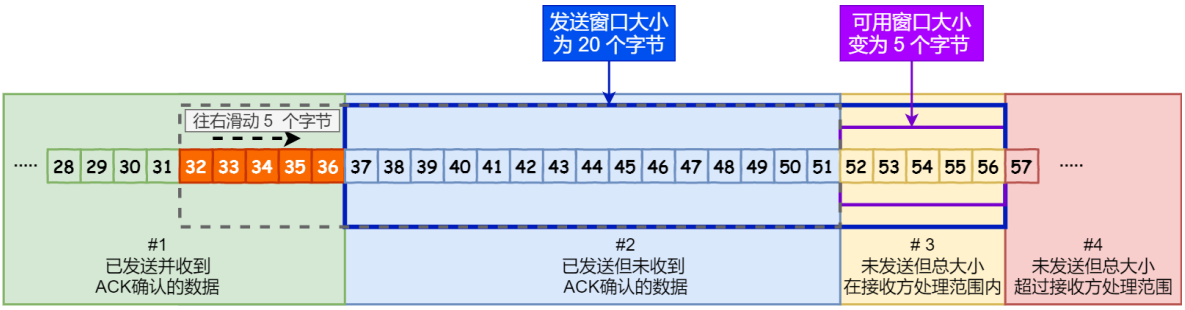


- #1 是已发送并收到 ACK确认的数据：1~31 字节
- #2 是已发送但未收到 ACK确认的数据：32~45 字节
- #3 是未发送但总大小在接收方处理范围内（接收方还有空间）：46~51字节
- #4 是未发送但总大小超过接收方处理范围（接收方没有空间）：52字节以后

在下图，当发送方把数据「全部」都一下发送出去后，可用窗口的大小就为 0 了，表明可用窗口耗尽，在没收到 ACK 确认之前是无法继续发送数据了。

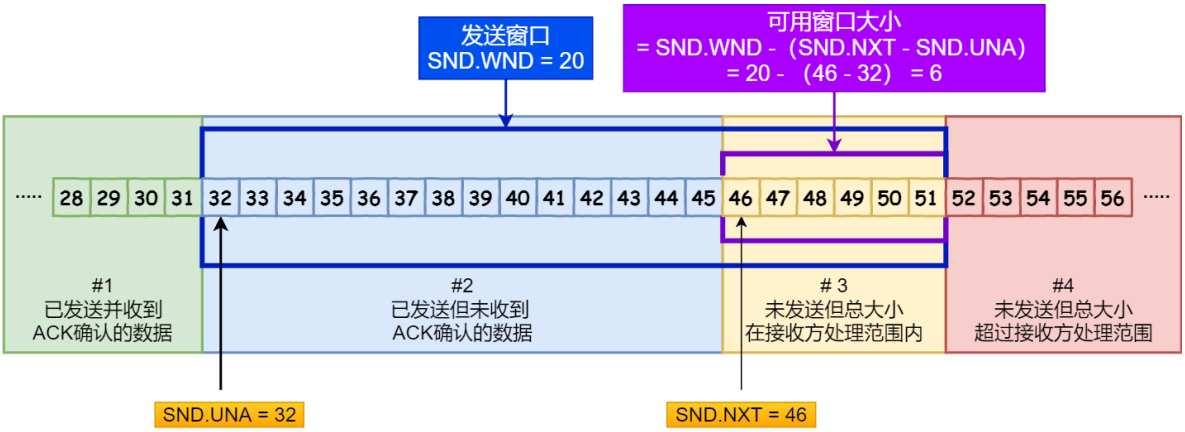


在下图，当收到之前发送的数据 32~36 字节的 ACK 确认应答后，如果发送窗口的大小没有变化，则**滑动窗口往右边移动 5 个字节，因为有 5 个字节的数据被应答确认**，接下来 52~56 字节又变成了可用窗口，那么后续也就可以发送 52~56 这 5 个字节的数据了。



> 程序是如何表示发送方的四个部分的呢?

TCP 滑动窗口方案使用三个指针来跟踪在四个传输类别中的每一个类别中的字节。其中两个指针是绝对指针（指特定的序列号），一个是相对指针（需要做偏移）。



- SND.WND：表示发送窗口的大小（大小是由接收方指定的）；
- SND.UNA：是一个绝对指针，它指向的是已发送但未收到确认的第一个字节的序列号，也就是 #2 的第一个字节。
- SND.NXT：也是一个绝对指针，它指向未发送但可发送范围的第一个字节的序列号，也就是 #3 的第一个字节。
- 指向 #4 的第一个字节是个相对指针，它需要 SND.UNA 指针加上 SND.WND 大小的偏移量，就可以指向 #4 的第一个字节了。

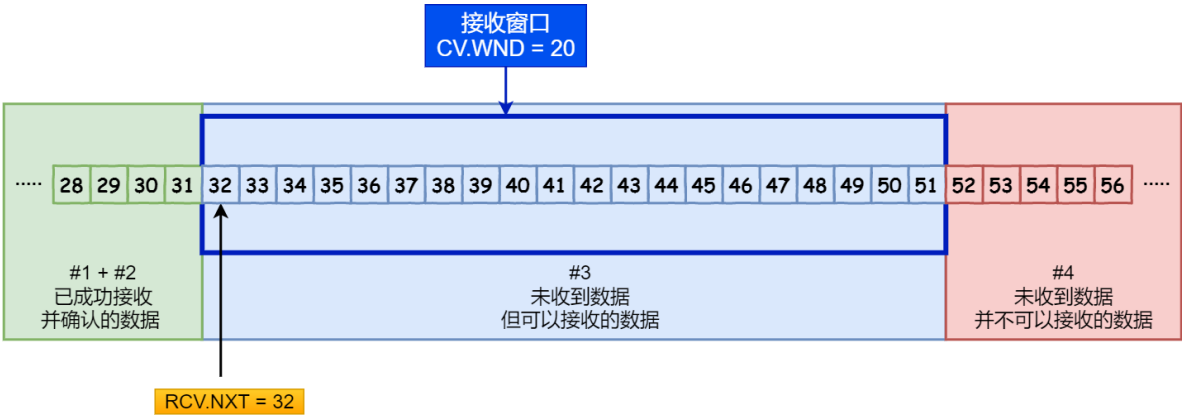
那么可用窗口大小的计算就可以是：

可用窗口大 = SND.WND - (SND.NXT - SND.UNA)

> 接收方的滑动窗口

接下来我们看看接收方的窗口，接收窗口相对简单一些，根据处理的情况划分成三个部分：

- #1 + #2 是已成功接收并确认的数据（等待应用进程读取）；
- #3 是未收到数据但可以接收的数据；
- #4 未收到数据并不可以接收的数据；



其中三个接收部分，使用两个指针进行划分：

- RCV.WND：表示接收窗口的大小，它会通告给发送方。
- RCV.NXT：是一个指针，它指向期望从发送方发送来的下一个数据字节的序列号，也就是 #3 的第一个字节。

- 指向 #4 的第一个字节是个相对指针，它需要 `RCV.NXT` 指针加上 `RCV.WND` 大小的偏移量，就可以指向 #4 的第一个字节了。

> 接收窗口和发送窗口的大小是相等的吗?

并不是完全相等，接收窗口的大小是**约等于**发送窗口的大小的。

因为滑动窗口并不是一成不变的。比如，当接收方的应用进程读取数据的速度非常快的话，这样的话接收窗口可以很快的就空缺出来。那么新的接收窗口大小，是通过 TCP 报文中的 Windows 字段来告诉发送方。那么这个传输过程是存在时延的，所以接收窗口和发送窗口是约等于的关系。

流量控制

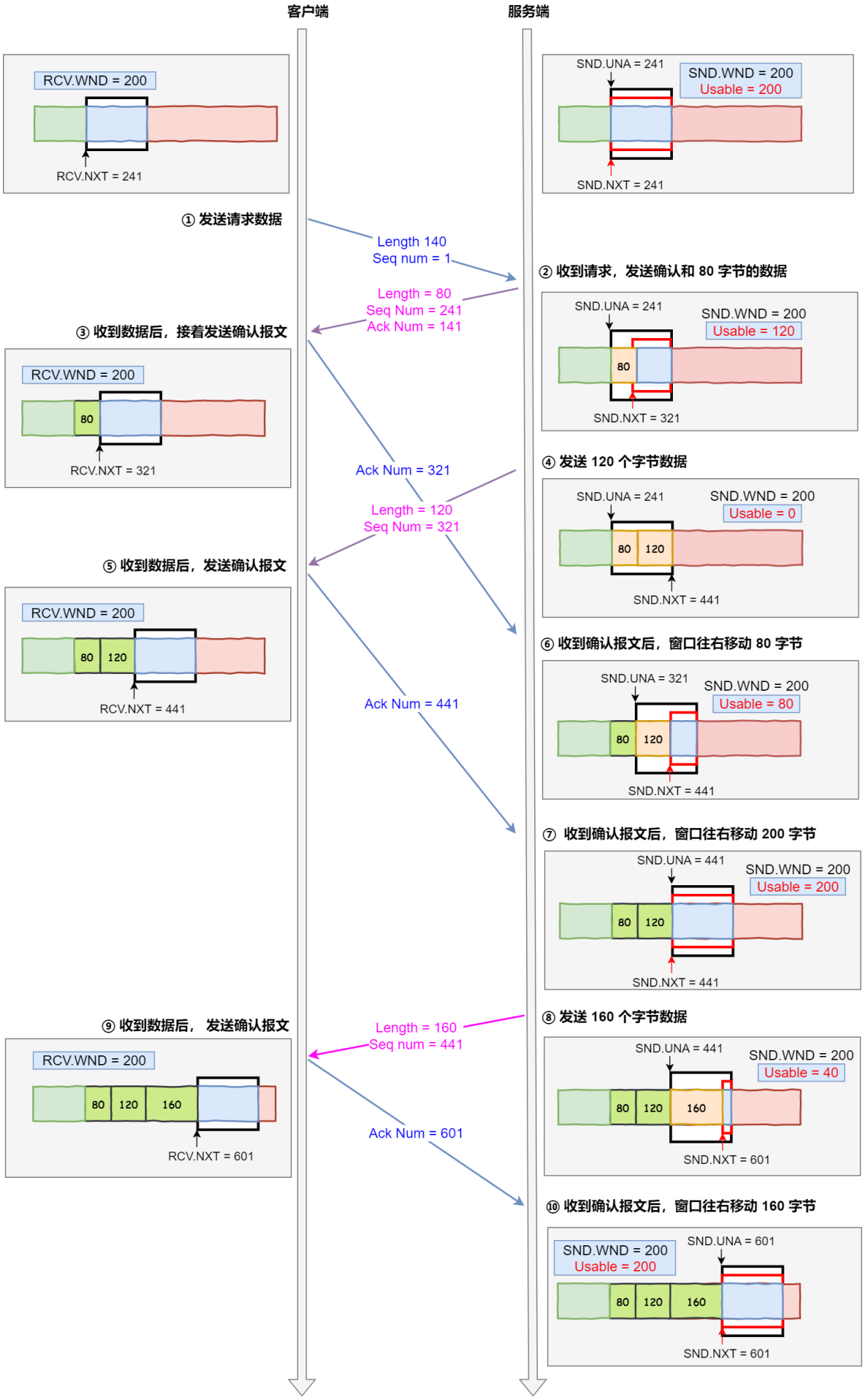
发送方不能无脑的发数据给接收方，要考虑接收方处理能力。

如果一直无脑的发数据给对方，但对方处理不过来，那么就会导致触发重发机制，从而导致网络流量的无端的浪费。

为了解决这种现象发生，**TCP 提供一种机制可以让「发送方」根据「接收方」的实际接收能力控制发送的数据量，这就是所谓的流量控制。**

下面举个栗子，为了简单起见，假设以下场景：

- 客户端是接收方，服务端是发送方
- 假设接收窗口和发送窗口相同，都为 `200`
- 假设两个设备在整个传输过程中都保持相同的窗口大小，不受外界影响



根据上图的流量控制，说明下每个过程：

1. 客户端向服务端发送请求数据报文。这里要说明下，本次例子是把服务端作为发送方，所以没有画出服务端的接收窗口。
2. 服务端收到请求报文后，发送确认报文和 80 字节的数据，于是可用窗口 `Usable` 减少为 120 字节，同时 `SND.NXT` 指针也向右偏移 80 字节后，指向 321，**这意味着下次发送数据的时候，序列号是 321。**
3. 客户端收到 80 字节数据后，于是接收窗口往右移动 80 字节，`RCV.NXT` 也就指向 321，**这意味着客户端期望的下一个报文的序列号是 321**，接着发送确认报文给服务端。
4. 服务端再次发送了 120 字节数据，于是可用窗口耗尽为 0，服务端无法再继续发送数据。
5. 客户端收到 120 字节的数据后，于是接收窗口往右移动 120 字节，`RCV.NXT` 也就指向 441，接着发送确认报文给服务端。
6. 服务端收到对 80 字节数据的确认报文后，`SND.UNA` 指针往右偏移后指向 321，于是可用窗口 `Usable` 增大到 80。
7. 服务端收到对 120 字节数据的确认报文后，`SND.UNA` 指针往右偏移后指向 441，于是可用窗口 `Usable` 增大到 200。
8. 服务端可以继续发送了，于是发送了 160 字节的数据后，`SND.NXT` 指向 601，于是可用窗口 `Usable` 减少到 40。
9. 客户端收到 160 字节后，接收窗口往右移动了 160 字节，`RCV.NXT` 也就是指向了 601，接着发送确认报文给服务端。
10. 服务端收到对 160 字节数据的确认报文后，发送窗口往右移动了 160 字节，于是 `SND.UNA` 指针偏移了 160 后指向 601，可用窗口 `Usable` 也就增大至了 200。

操作系统缓冲区与滑动窗口的关系

前面的流量控制例子，我们假定了发送窗口和接收窗口是不变的，但是实际上，发送窗口和接收窗口中所存放的字节数，都是放在操作系统内存缓冲区中的，而操作系统的缓冲区，会**被操作系统调整**。

当应用进程没办法及时读取缓冲区的内容时，也会对我们的缓冲区造成影响。

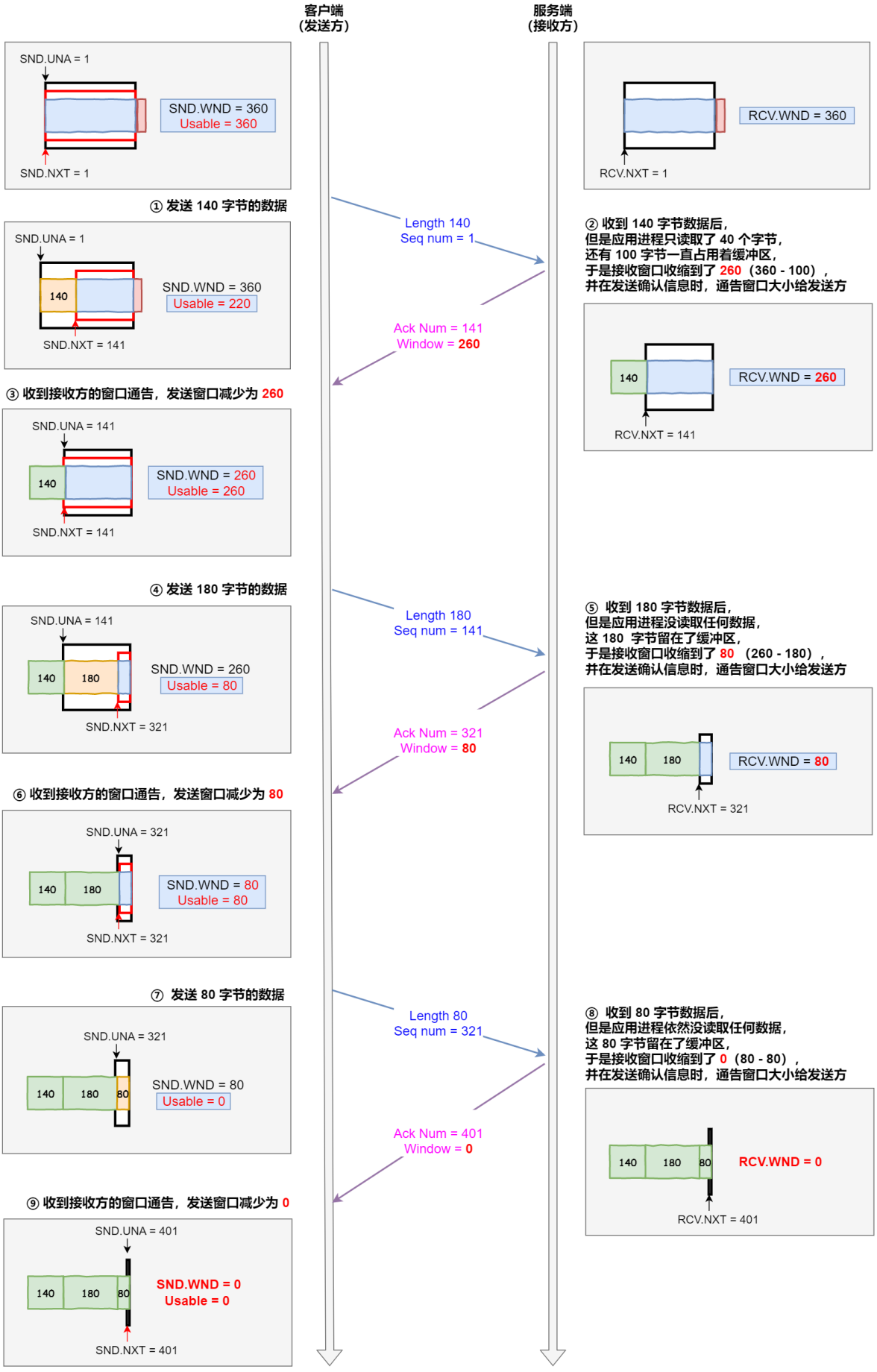
> 那操心系统的缓冲区，是如何影响发送窗口和接收窗口的呢？

我们先来看看第一个例子。

当应用程序没有及时读取缓存时，发送窗口和接收窗口的变化。

考虑以下场景：

- 客户端作为发送方，服务端作为接收方，发送窗口和接收窗口初始大小为 `360`；
- 服务端非常的繁忙，当收到客户端的数据时，应用层不能及时读取数据。



根据上图的流量控制，说明下每个过程：

1. 客户端发送 140 字节数据后，可用窗口变为 220 （360 - 140）。
2. 服务端收到 140 字节数据，**但是服务端非常繁忙，应用进程只读取了 40 个字节，还有 100 字节占用着缓冲区，于是接收窗口收缩到了 260 （360 - 100）**，最后发送确认信息时，将窗口大小通

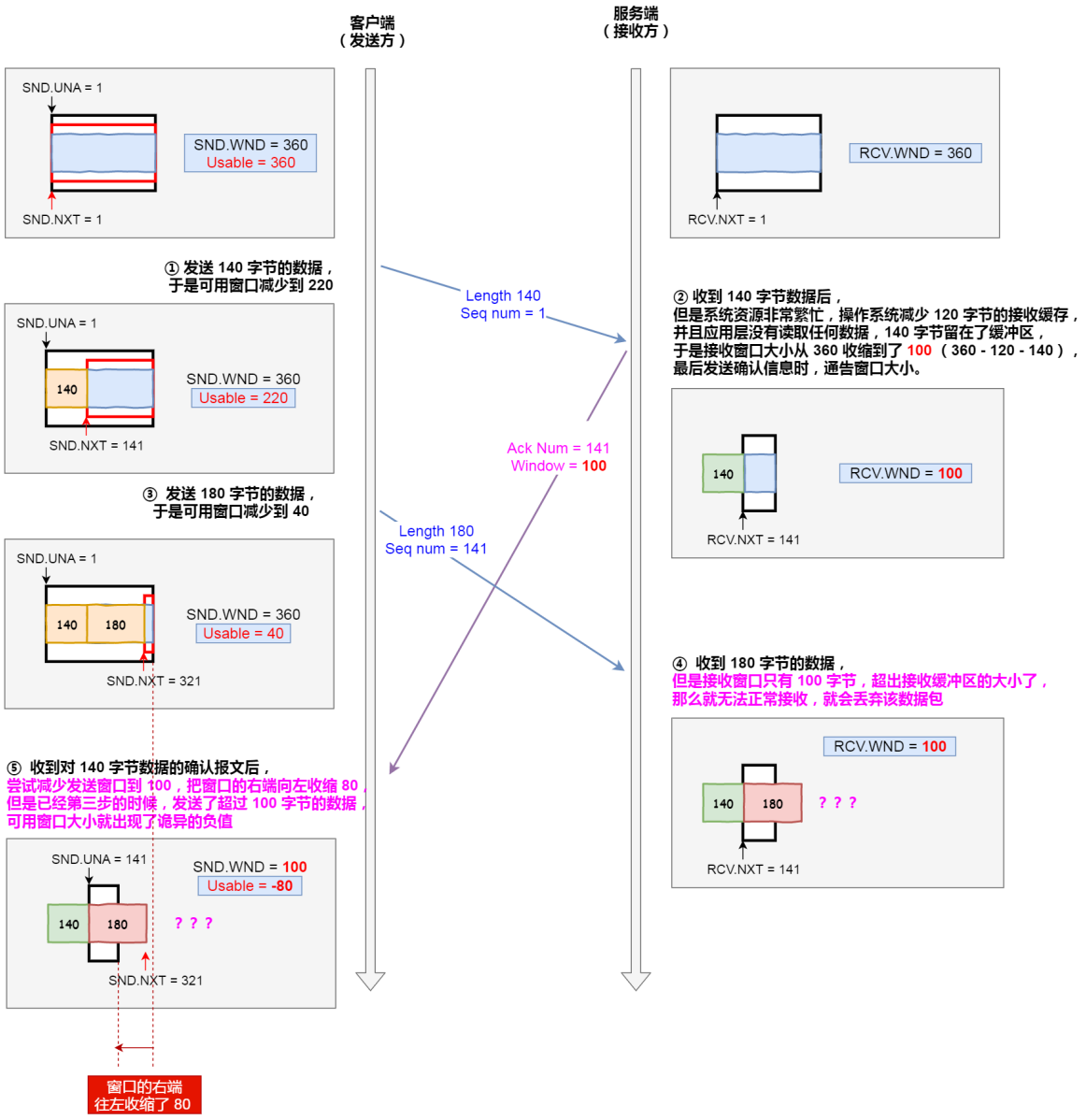
告给客户端。

3. 客户端收到确认和窗口通告报文后，发送窗口减少为 260。
4. 客户端发送 180 字节数据，此时可用窗口减少到 80。
5. 服务端收到 180 字节数据，**但是应用程序没有读取任何数据，这 180 字节直接就留在了缓冲区，于是接收窗口收缩到了 80 （260 - 180）**，并在发送确认信息时，通过窗口大小给客户端。
6. 客户端收到确认和窗口通告报文后，发送窗口减少为 80。
7. 客户端发送 80 字节数据后，可用窗口耗尽。
8. 服务端收到 80 字节数据，**但是应用程序依然没有读取任何数据，这 80 字节留在了缓冲区，于是接收窗口收缩到了 0**，并在发送确认信息时，通过窗口大小给客户端。
9. 客户端收到确认和窗口通告报文后，发送窗口减少为 0。

可见最后窗口都收缩为 0 了，也就是发生了窗口关闭。当发送方可用窗口变为 0 时，发送方实际上会定时发送窗口探测报文，以便知道接收方的窗口是否发生了改变，这个内容后面会说，这里先简单提一下。

我们先来看看第二个例子。

当服务端系统资源非常紧张的时候，操心系统可能会直接减少了接收缓冲区大小，这时应用程序又无法及时读取缓存数据，那么这时候就有严重的事情发生了，会出现数据包丢失的现象。



说明下每个过程：

1. 客户端发送 140 字节的数据，于是可用窗口减少到了 220。
2. **服务端因为现在非常的繁忙，操作系统于是就把接收缓存减少了 120 字节，当收到 140 字节数据后，又因为应用程序没有读取任何数据，所以 140 字节留在了缓冲区中，于是接收窗口大小从 360 收缩成了 100**，最后发送确认信息时，通告窗口大小给对方。
3. 此时客户端因为还没有收到服务端的通告窗口报文，所以不知道此时接收窗口收缩成了 100，客户端只会看自己的可用窗口还有 220，所以客户端就发送了 180 字节数据，于是可用窗口减少到 40。
4. 服务端收到了 180 字节数据时，**发现数据大小超过了接收窗口的大小，于是就把数据包丢失了。**
5. 客户端收到第 2 步时，服务端发送的确认报文和通告窗口报文，尝试减少发送窗口到 100，把窗口的右端向左收缩了 80，此时可用窗口的大小就会出现诡异的负值。

所以，如果发生了先减少缓存，再收缩窗口，就会出现丢包的现象。

为了防止这种情况发生，TCP 规定是不允许同时减少缓存又收缩窗口的，而是采用先收缩窗口，过段时间再减少缓存，这样就可以避免了丢包情况。

窗口关闭

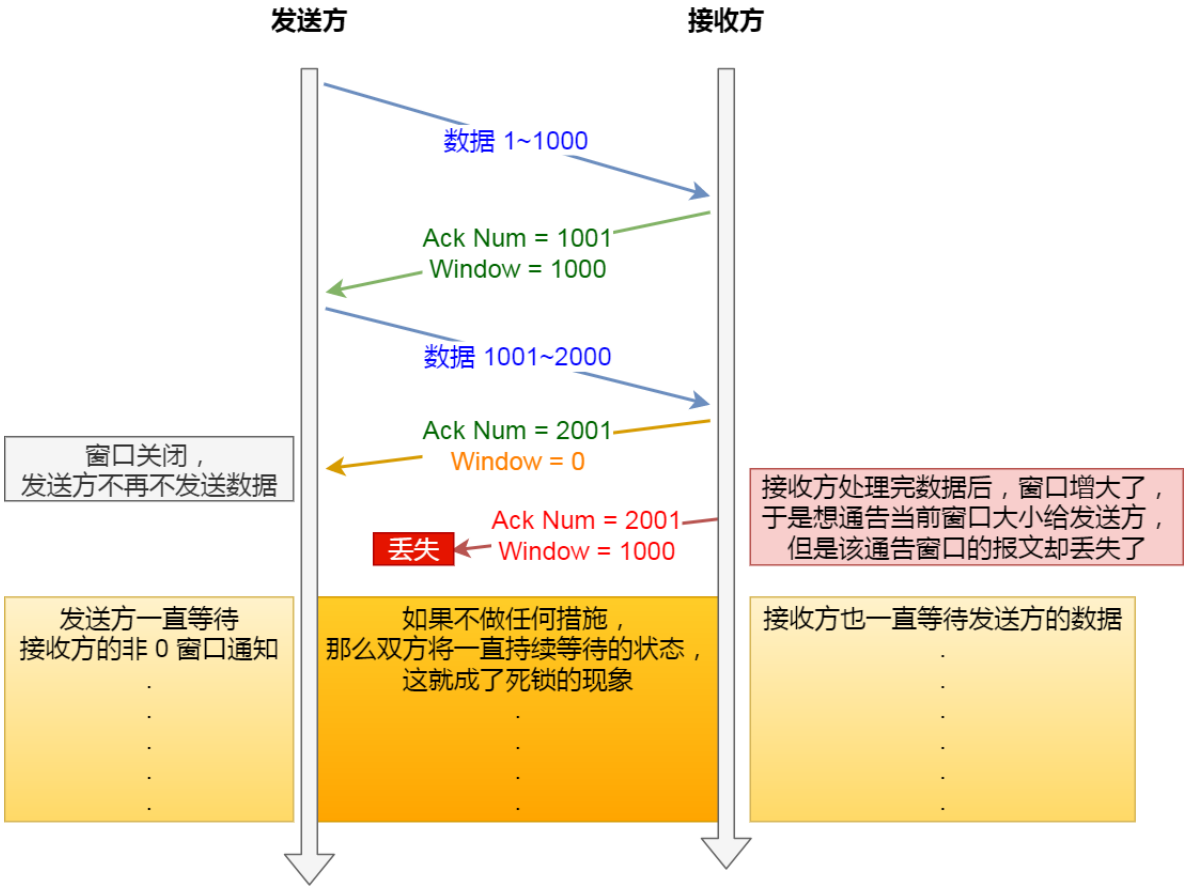
在前面我们都看到了，TCP 通过让接收方指明希望从发送方接收的数据大小（窗口大小）来进行流量控制。

如果窗口大小为 0 时，就会阻止发送方给接收方传递数据，直到窗口变为非 0 为止，这就是窗口关闭。

> 窗口关闭潜在的危险

接收方向发送方通告窗口大小时，是通过 `ACK` 报文来通告的。

那么，当发生窗口关闭时，接收方处理完数据后，会向发送方通告一个窗口非 0 的 ACK 报文，如果这个通告窗口的 ACK 报文在网络中丢失了，那麻烦就大了。

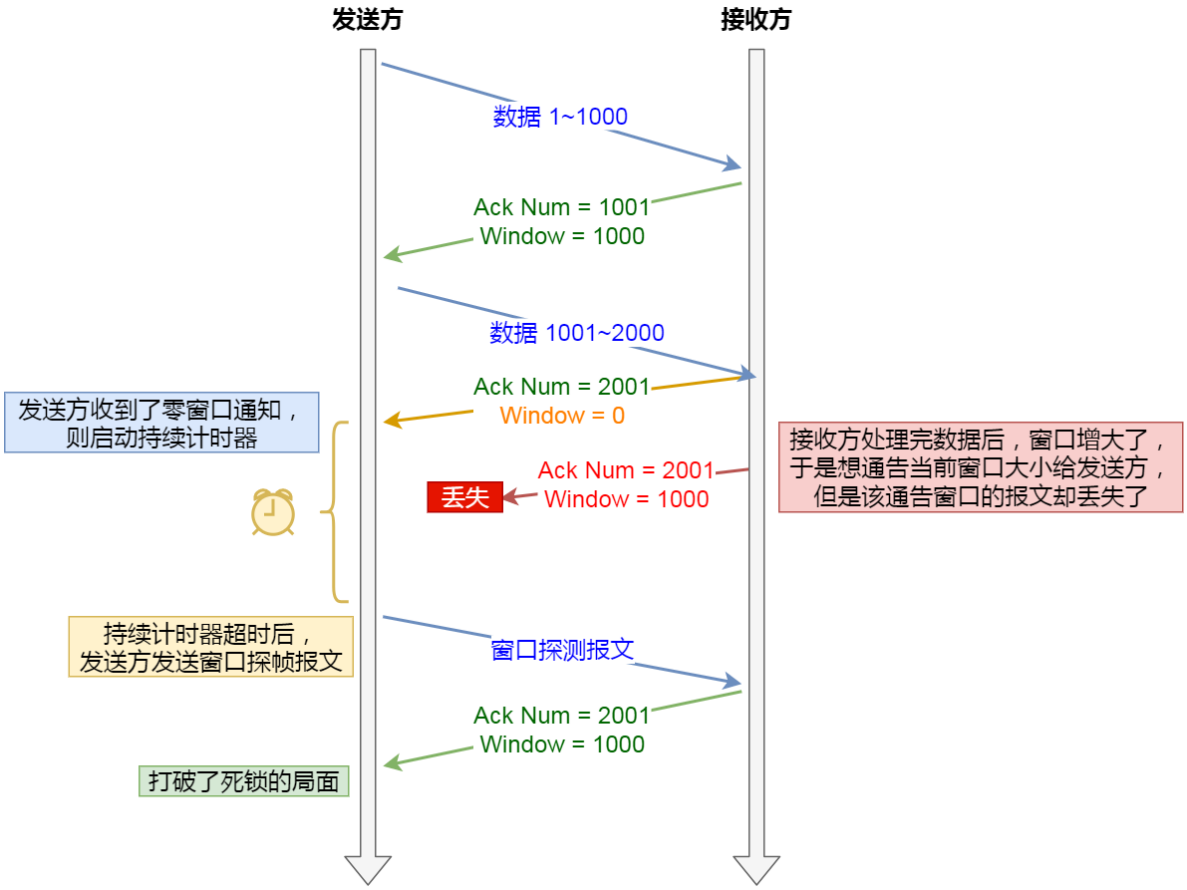


这会导致发送方一直等待接收方的非 0 窗口通知，接收方也一直等待发送方的数据，如不采取措施，这种相互等待的过程，会造成了死锁的现象。

> TCP 是如何解决窗口关闭时，潜在的死锁现象呢?

为了解决这个问题，TCP 为每个连接设有一个持续定时器，**只要 TCP 连接一方收到对方的零窗口通知，就启动持续计时器。**

如果持续计时器超时，就会发送**窗口探测 (Window probe) 报文**，而对方在确认这个探测报文时，给出自己现在的接收窗口大小。



- 如果接收窗口仍然为 0，那么收到这个报文的一方就会重新启动持续计时器；
- 如果接收窗口不是 0，那么死锁的局面就可以被打破了。

窗口探测的次数一般为 3 次，每次大约 30-60 秒（不同的实现可能会不一样）。如果 3 次过后接收窗口还是 0 的话，有的 TCP 实现就会发 `RST` 报文来中断连接。

糊涂窗口综合症

如果接收方太忙了，来不及取走接收窗口里的数据，那么就会导致发送方的发送窗口越来越小。

到最后，**如果接收方腾出几个字节并告诉发送方现在有几个字节的窗口，而发送方会义无反顾地发送这几个字节，这就是糊涂窗口综合症。**

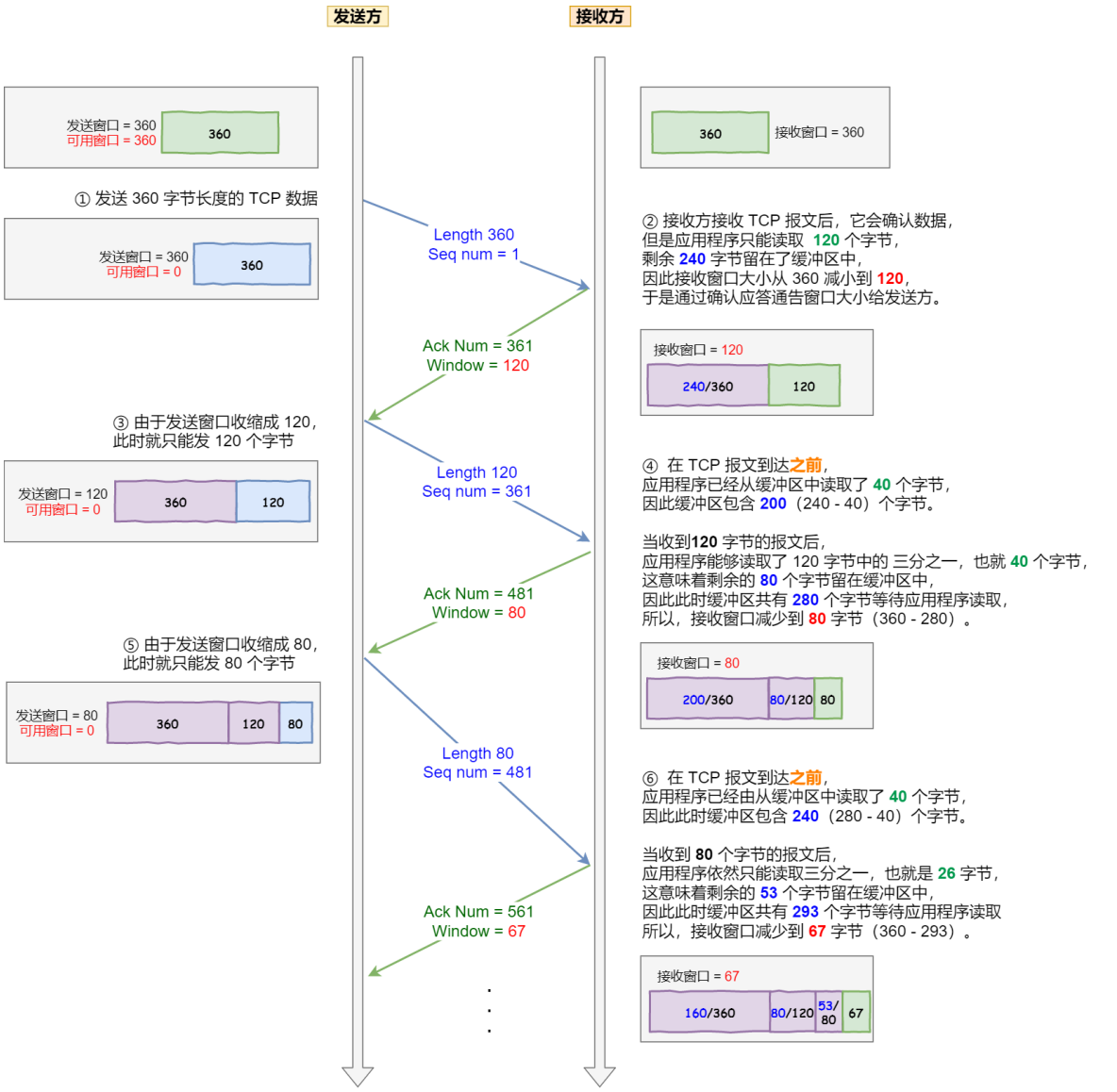
要知道，我们的 `TCP + IP` 头有 `40` 个字节，为了传输那几个字节的数据，要达上这么大的开销，这太不经济了。

就好像一个可以承载 50 人的大巴车，每次来了一两个人，就直接发车。除非家里有矿的大巴司机，才敢这样玩，不然迟早破产。要解决这个问题也不难，大巴司机等乘客数量超过了 25 个，才认定可以发车。

现举个糊涂窗口综合症的栗子，考虑以下场景：

接收方的窗口大小是 360 字节，但接收方由于某些原因陷入困境，假设接收方的应用层读取的能力如下：

- 接收方每接收 3 个字节，应用程序就只能从缓冲区中读取 1 个字节的数据；
- 在下一个发送方的 TCP 段到达之前，应用程序还从缓冲区中读取了 40 个额外的字节；



每个过程的窗口大小的变化，在图中都描述的很清楚了，可以发现窗口不断减少了，并且发送的数据都是比较小的了。

所以，糊涂窗口综合症的现象是可以发生在发送方和接收方：

- 接收方可以通告一个小的窗口
- 而发送方可以发送小数据

于是，要解决糊涂窗口综合症，就解决上面两个问题就可以了

- 让接收方不通告小窗口给发送方
- 让发送方避免发送小数据

> 怎么让接收方不通告小窗口呢？

接收方通常的策略如下：

当「窗口大小」小于 min(MSS，缓存空间/2)，也就是小于 MSS 与 1/2 缓存大小中的最小值时，就会向发送方通告窗口为 `0`，也就阻止了发送方再发数据过来。

等到接收方处理了一些数据后，窗口大小 >= MSS，或者接收方缓存空间有一半可以使用，就可以把窗口打开让发送方发送数据过来。

> 怎么让发送方避免发送小数据呢?

发送方通常的策略:

使用 Nagle 算法，该算法的思路是延时处理，它满足以下两个条件中的一条才可以发送数据：

- 要等到窗口大小 >= `MSS` 或是 数据大小 >= `MSS`
- 收到之前发送数据的 `ack` 回包

只要没满足上面条件中的一条，发送方一直在囤积数据，直到满足上面的发送条件。

另外，Nagle 算法默认是打开的，如果对于一些需要小数据包交互的场景的程序，比如，telnet 或 ssh 这样的交互性比较强的程序，则需要关闭 Nagle 算法。

可以在 Socket 设置 `TCP_NODELAY` 选项来关闭这个算法（关闭 Nagle 算法没有全局参数，需要根据每个应用自己的特点来关闭）

```
setsockopt(sock_fd, IPPROTO_TCP, TCP_NODELAY, (char *)&value, sizeof(int));
```

拥塞控制

> 为什么要有拥塞控制呀，不是有流量控制了吗?

前面的流量控制是避免「发送方」的数据填满「接收方」的缓存，但是并不知道网络的中发生了什么。

一般来说，计算机网络都处在一个共享的环境。因此也有可能会因为其他主机之间的通信使得网络拥堵。

在网络出现拥堵时，如果继续发送大量数据包，可能会导致数据包时延、丢失等，这时 TCP 就会重传数据，但是一重传就会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这个情况就会进入恶性循环被不断地放大....

所以，TCP 不能忽略网络上发生的事，它被设计成一个无私的协议，当网络发送拥塞时，TCP 会自我牺牲，降低发送的数据量。

于是，就有了**拥塞控制**，控制的目的就是**避免「发送方」的数据填满整个网络。**

为了在「发送方」调节所要发送数据的量，定义了一个叫做「**拥塞窗口**」的概念。

> 什么是拥塞窗口？和发送窗口有什么关系呢?

拥塞窗口 cwnd是发送方维护的一个的状态变量，它会根据**网络的拥塞程度动态变化的。**

我们在前面提到过发送窗口 `swnd` 和接收窗口 `rwnd` 是约等于的关系，那么由于加入了拥塞窗口的概念后，此时发送窗口的值是swnd = min(cwnd, rwnd)，也就是拥塞窗口和接收窗口中的最小值。

拥塞窗口 `cwnd` 变化的规则：

- 只要网络中没有出现拥塞， `cwnd` 就会增大；
- 但网络中出现了拥塞， `cwnd` 就减少；

> 那么怎么知道当前网络是否出现了拥塞呢？

其实只要「发送方」没有在规定时间内接收到 ACK 应答报文，也就是**发生了超时重传，就会认为网络出现了用拥塞。**

> 拥塞控制有哪些控制算法？

拥塞控制主要是四个算法：

- 慢启动
- 拥塞避免
- 拥塞发生
- 快速恢复

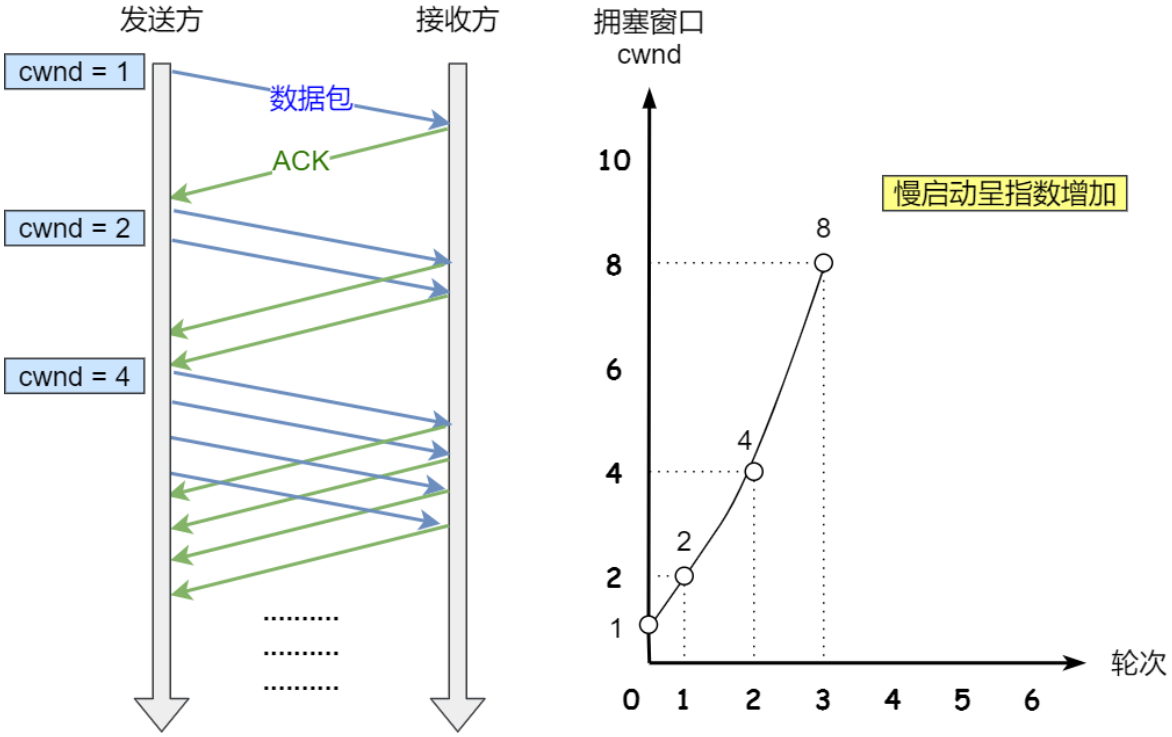
慢启动

TCP 在刚建立连接完成后，首先是有个慢启动的过程，这个慢启动的意思就是一点一点的提高发送数据包的数量，如果一上来就发大量的数据，这不是给网络添堵吗？

慢启动的算法记住一个规则就行：**当发送方每收到一个 ACK，拥塞窗口 cwnd 的大小就会加 1。**

这里假定拥塞窗口 `cwnd` 和发送窗口 `swnd` 相等，下面举个栗子：

- 连接建立完成后，一开始初始化 `cwnd = 1` ，表示可以传一个 `MSS` 大小的数据。
- 当收到一个 ACK 确认应答后，cwnd 增加 1，于是一次能够发送 2 个
- 当收到 2 个的 ACK 确认应答后， cwnd 增加 2，于是就可以比之前多发2 个，所以这一次能够发送 4 个
- 当这 4 个的 ACK 确认到来的时候，每个确认 cwnd 增加 1， 4 个确认 cwnd 增加 4，于是就可以比之前多发 4 个，所以这一次能够发送 8 个。



可以看出慢启动算法，发包的个数是**指数性的增长**。

> 那慢启动涨到什么时候是个头呢?

有一个叫慢启动门限 `ssthresh` （slow start threshold）状态变量。

- 当 `cwnd` < `ssthresh` 时，使用慢启动算法。
- 当 `cwnd` >= `ssthresh` 时，就会使用「拥塞避免算法」。

拥塞避免算法

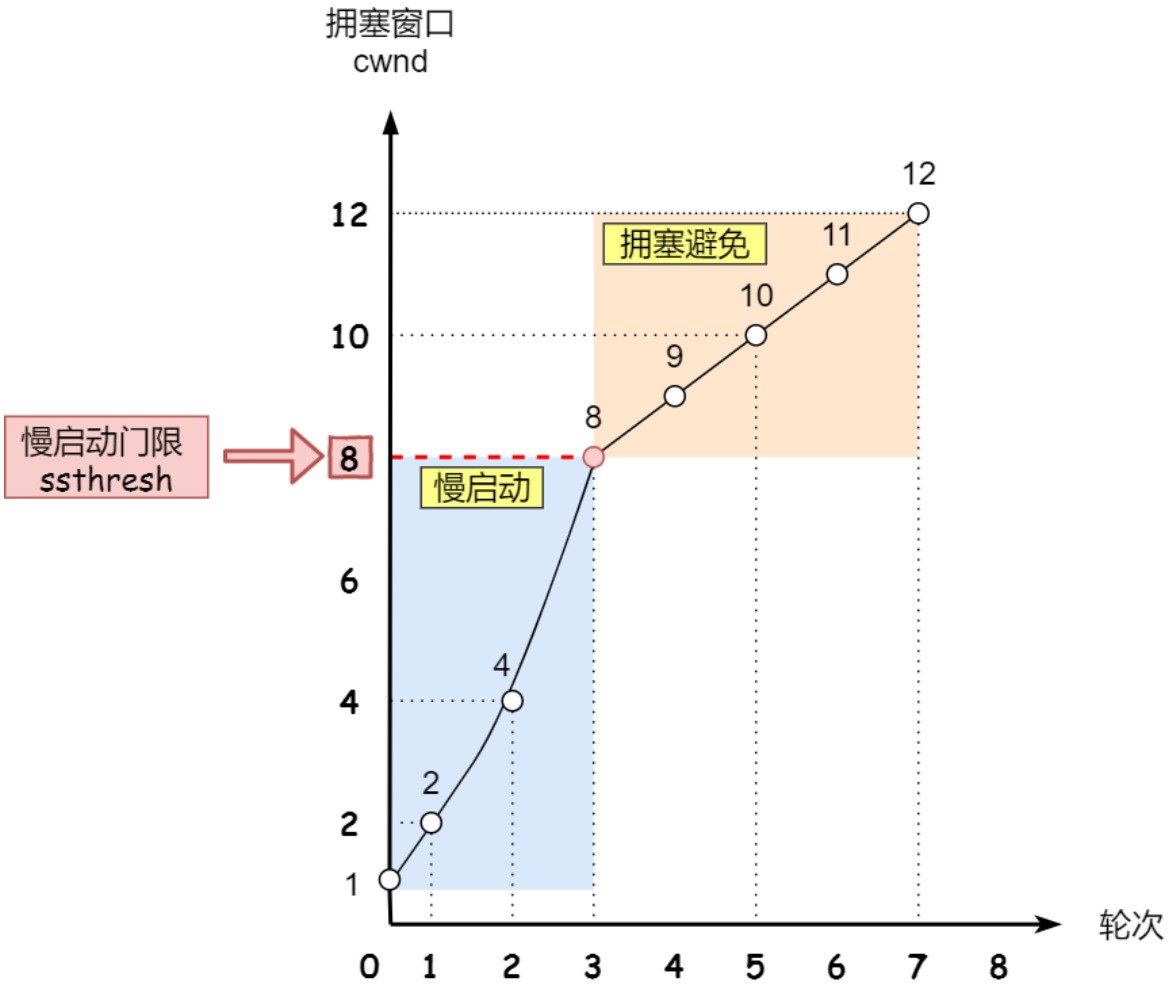
前面说道，当拥塞窗口 `cwnd` 「超过」慢启动门限 `ssthresh` 就会进入拥塞避免算法。

一般来说 `ssthresh` 的大小是 `65535` 字节。

那么进入拥塞避免算法后，它的规则是：**每当收到一个 ACK 时，cwnd 增加 1/cwnd。**

接上前面的慢启动的栗子，现假定 `ssthresh` 为 `8`：

- 当 8 个 ACK 应答确认到来时，每个确认增加 1/8，8 个 ACK 确认 cwnd 一共增加 1，于是这一次能够发送 9 个 `MSS` 大小的数据，变成了**线性增长。**



所以，我们可以发现，拥塞避免算法就是将原本慢启动算法的指数增长变成了线性增长，还是增长阶段，但是增长速度缓慢了一些。

就这么一直增长着后，网络就会慢慢进入了拥塞的状况了，于是就会出现丢包现象，这时就需要对丢失的数据包进行重传。

当触发了重传机制，也就进入了「拥塞发生算法」。

拥塞发生

当网络出现拥塞，也就是会发生数据包重传，重传机制主要有两种：

- 超时重传
- 快速重传

这两种使用的拥塞发送算法是不同的，接下来分别来说说。

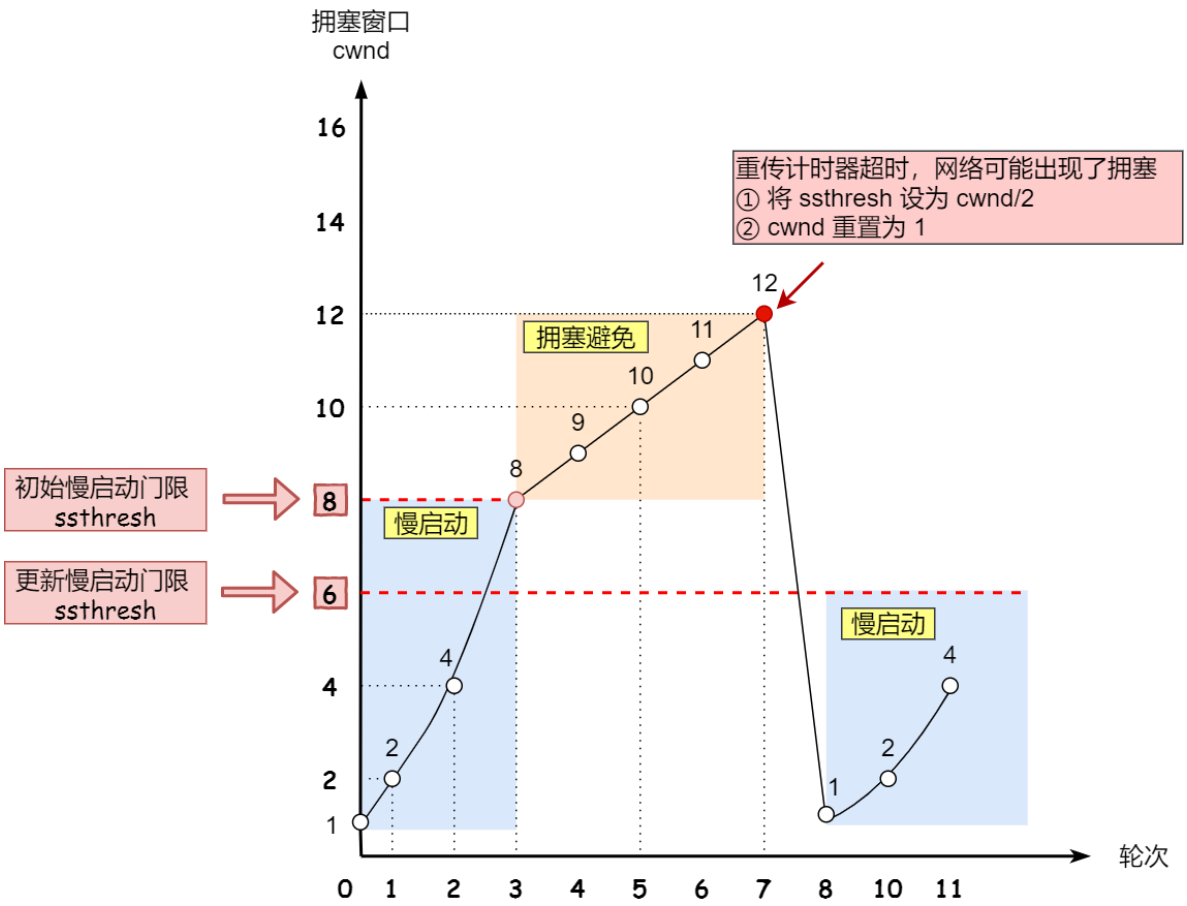
> 发生超时重传的拥塞发生算法

当发生了「超时重传」，则就会使用拥塞发生算法。

这个时候，ssthresh 和 cwnd 的值会发生变化：

- `ssthresh` 设为 `cwnd/2`，

- `cwnd` 重置为 `1`



接着，就重新开始慢启动，慢启动是会突然减少数据流的。这真是一旦「超时重传」，马上回到解放前。但是这种方式太激进了，反应也很强烈，会造成网络卡顿。

就好像本来在秋名山高速漂移着，突然来个紧急刹车，轮胎受得了吗。。。

> 发生快速重传的拥塞发生算法

还有更好的方式，前面我们讲过「快速重传算法」。当接收方发现丢了一个中间包的时候，发送三次前一个包的 ACK，于是发送端就会快速地重传，不必等待超时再重传。

TCP 认为这种情况不严重，因为大部分没丢，只丢了一小部分，则 `ssthresh` 和 `cwnd` 变化如下：

- `cwnd = cwnd/2` ，也就是设置为原来的一半;
- `ssthresh = cwnd` ;
- 进入快速恢复算法

快速恢复

快速重传和快速恢复算法一般同时使用，快速恢复算法是认为，你还能收到 3 个重复 ACK 说明网络也不那么糟糕，所以没有必要像 `RTO` 超时那么强烈。

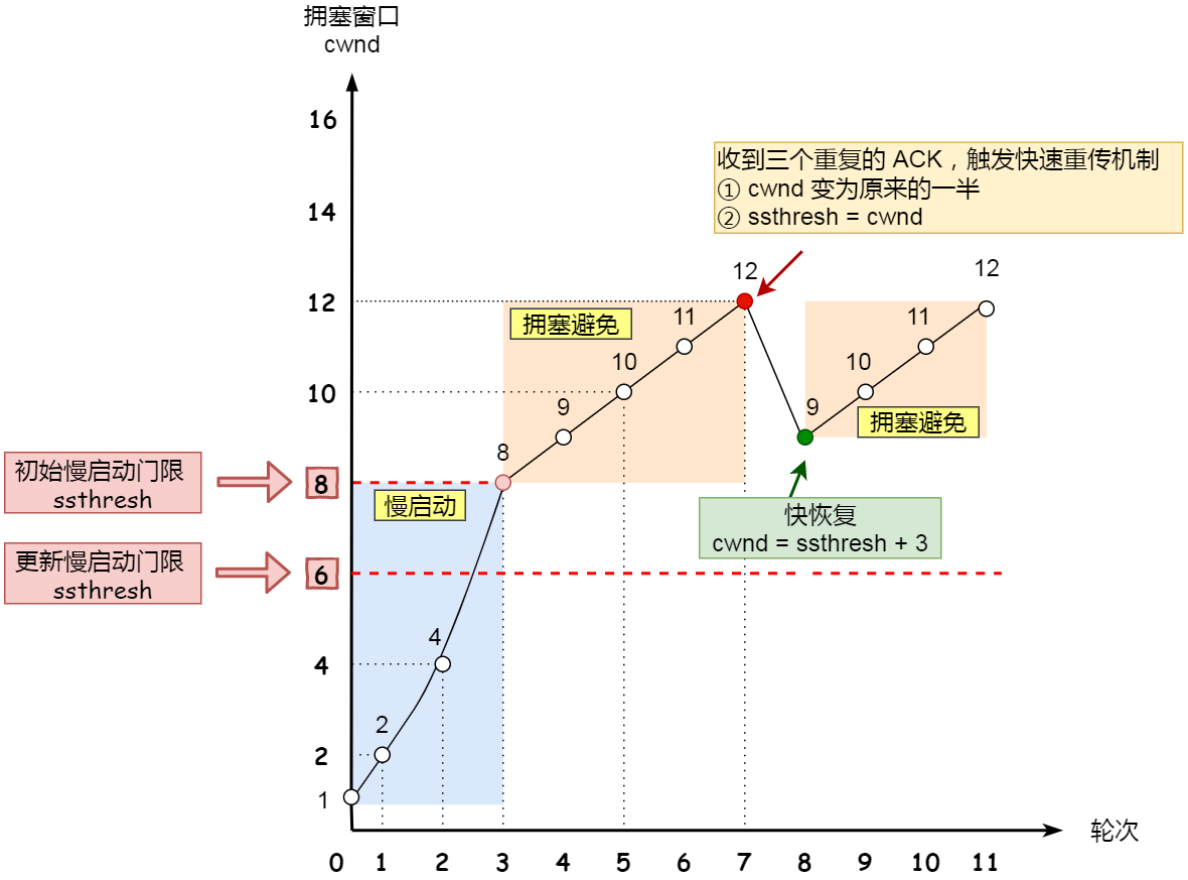
正如前面所说，进入快速恢复之前， `cwnd` 和 `ssthresh` 已被更新了：

- `cwnd = cwnd/2` ，也就是设置为原来的一半;

- `ssthresh = cwnd`；

然后，进入快速恢复算法如下：

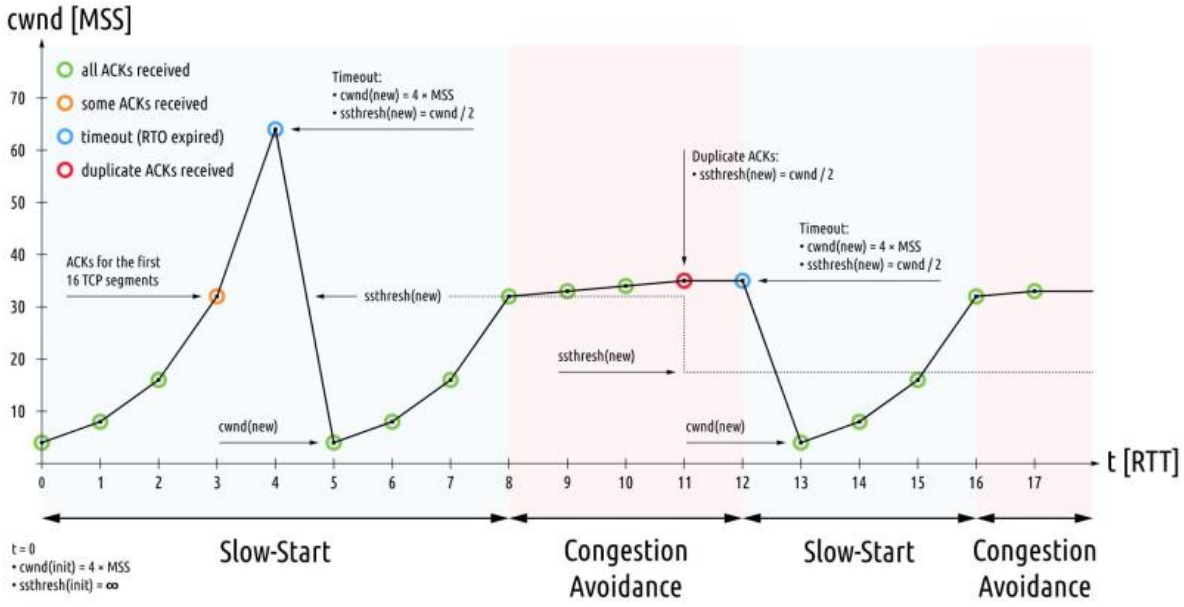
- 拥塞窗口 `cwnd = ssthresh + 3` （3 的意思是确认有 3 个数据包被收到了）；
- 重传丢失的数据包；
- 如果再收到重复的 ACK，那么 cwnd 增加 1；
- 如果收到新数据的 ACK 后，把 cwnd 设置为第一步中的 ssthresh 的值，原因是该 ACK 确认了新的数据，说明从 duplicated ACK 时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态；



也就是没有像「超时重传」一夜回到解放前，而是还在比较高的值，后续呈线性增长。

拥塞算法示意图

好了，以上就是拥塞控制的全部内容了，看完后，你再来看下面这张图片，每个过程我相信你都能明白：



巨人的肩膀

[1] 趣谈网络协议专栏.刘超.极客时间

[2] Web协议详解与抓包实战专栏.陶辉.极客时间

[3] TCP/IP详解 卷1：协议.范建华 译.机械工业出版社

[4] 图解TCP/IP.竹下隆史.人民邮电出版社

[5]The TCP/IP Guide.Charles M. Kozierok.

[6] TCP那些事（上）.陈皓.酷壳博客. https://coolshell.cn/articles/11564.html

[7] TCP那些事（下）.陈皓.酷壳博客.https://coolshell.cn/articles/11609.html

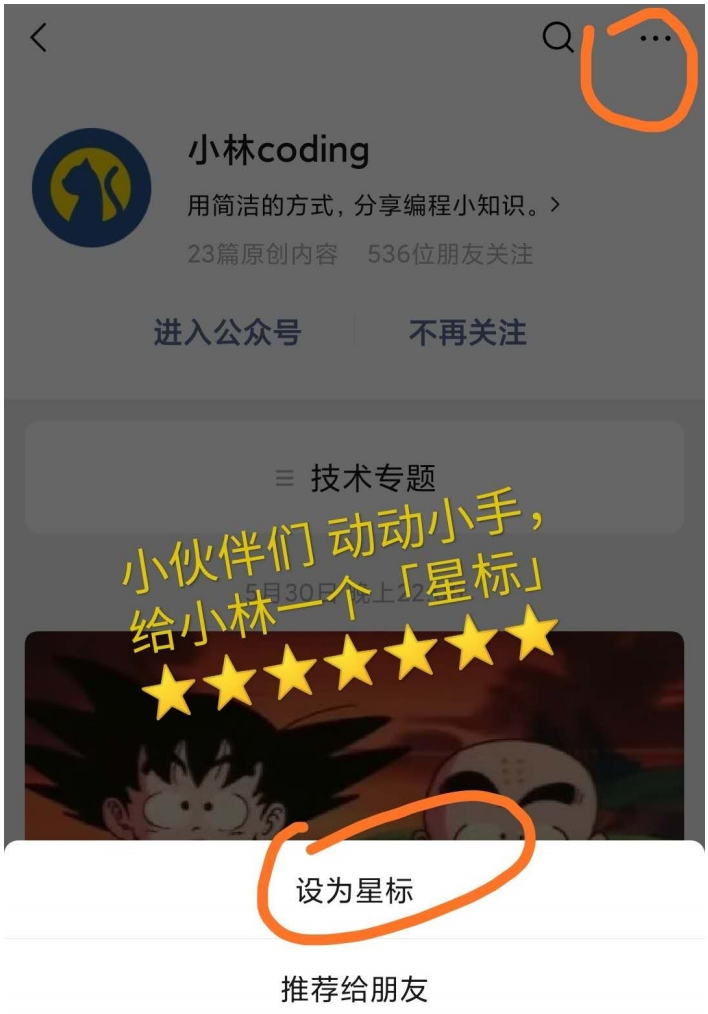
唠叨唠叨

是吧？ TCP 巨复杂吧？看完很累吧？

但这还只是 TCP 冰山一脚，它的更深处就由你们自己去探索啦。

本文只是抛砖引玉，若你有更好的想法或文章有误的地方，欢迎留言讨论！

小林是专为大家图解的工具人，Goodbye，我们下次见！



扫一扫
关注爱图解的
「小林coding」

读者问答

> 读者问：“整个看完收获很大，下面是我的一些疑问（稍后 会去确认）： 1.拥塞避免这一段，蓝色字体：每当收到一个 ACK时，cwnd增加1/cwnd。是否应该是 1/ssthresh?否则不符合线性增长。 2.快速重传的拥塞发生算法，步骤一和步骤2是 否写反了？否则快速恢复算法中最后一步【如果 收到新数据的ACK后，设置cwnd为 ssthresh,接看就进入了拥塞避免算法】没什么 意义。 3.对ssthresh的变化介绍的比较含糊。”

1. 是 1/cwnd，你可以在 RFC2581 第 3 页找到答案
2. 没有写反，同样你可以在 RFC2581 第 5 页找到答案
3. ssthresh 就是慢启动门限，我觉得 ssthresh 我已经说的很清楚了，当然你可以找其他资料补充你的疑惑

实战！我用 Wireshark 让你「看得见」TCP

前言

> “哈？啥是大白鲨？”

咳咳，主要是因为网络分析工具 Wireshark 的图标特别像大白鲨顶部的角。

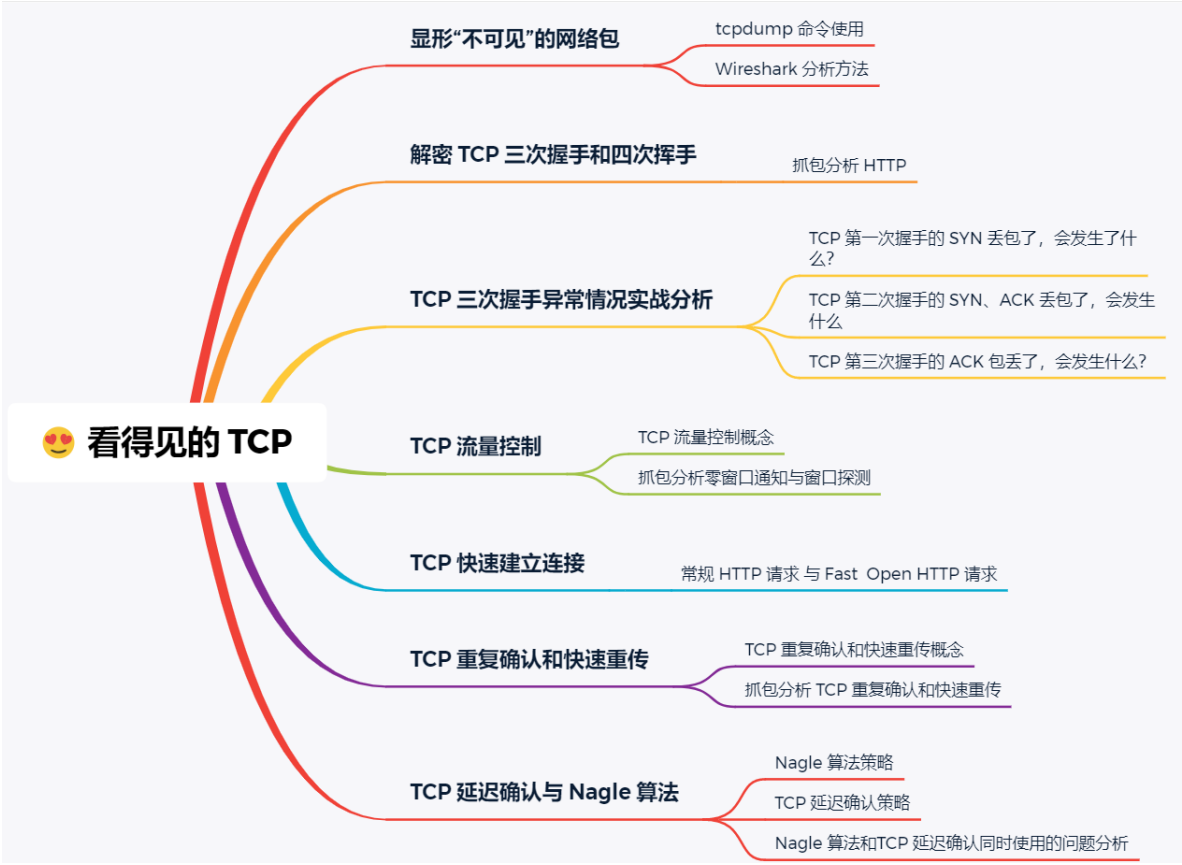
不信你看：

> “为什么拖了怎么久才发文？”

为了让大家更容易「看得见」 TCP，我搭建不少测试环境，并且数据包抓很多次，花费了不少时间，才抓到比较容易分析的数据包。

接下来丢包、乱序、超时重传、快速重传、选择性确认、流量控制等等 TCP 的特性，都能「一览无余」。

没错，我把 TCP 的"衣服扒光"了，就为了给大家看的清楚，嘻嘻。



正文

显形“不可见”的网络包

网络世界中的数据包交互我们肉眼是看不见的，它们就好像隐形了一样，我们对着课本学习计算机网络的时候就会觉得非常的抽象，加大了学习的难度。

还别说，我自己在大学的时候，也是如此。

直到工作后，认识了两大分析网络的利器：**tcpdump 和 Wireshark**，这两大利器把我们“看不见”的数据包，呈现在我们眼前，一目了然。

唉，当初大学学习计网的时候，要是能知道这两个工具，就不会学的一脸懵逼。

> tcpdump 和 Wireshark 有什么区别?

tcpdump 和 Wireshark 就是最常用的网络抓包和分析工具，更是分析网络性能必不可少的利器。

- tcpdump 仅支持命令行格式使用，常用在 Linux 服务器中抓取和分析网络包。
- Wireshark 除了可以抓包外，还提供了可视化分析网络包的图形页面。

所以，这两者实际上是搭配使用的，先用 tcpdump 命令在 Linux 服务器上抓包，接着把抓包的文件拖出到 Windows 电脑后，用 Wireshark 可视化分析。

当然，如果你是在 Windows 上抓包，只需要用 Wireshark 工具就可以。

> tcpdump 在 Linux 下如何抓包?

tcpdump 提供了大量的选项以及各式各样的过滤表达式，来帮助你抓取指定的数据包，不过不要担心，只需要掌握一些常用选项和过滤表达式，就可以满足大部分场景的需要了。

假设我们要抓取下面的 ping 的数据包：

```
# -I eth1 表示指定从 eth1 网口出去
# -c 3    表示发出 3 个 icmp 数据包
$ ping -I eth1 -c 3 183.232.231.174
PING 183.232.231.174 (183.232.231.174) from 192.168.3.33 eth1: 56(84) bytes of data.
64 bytes from 183.232.231.174: icmp_seq=1 ttl=56 time=17.2 ms
64 bytes from 183.232.231.174: icmp_seq=2 ttl=56 time=15.7 ms
64 bytes from 183.232.231.174: icmp_seq=3 ttl=56 time=15.2 ms

--- 183.232.231.174 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2021ms
rtt min/avg/max/mdev = 15.212/16.051/17.220/0.852 ms
```

要抓取上面的 ping 命令数据包，首先我们要知道 ping 的数据包是 `icmp` 协议，接着在使用 tcpdump 抓包的时候，就可以指定只抓 icmp 协议的数据包：

```
#  -i eth1  表示抓取 eth1 网口的数据包
#  icmp     表示抓取 icmp 协议的数据包
#  host     表示主机过滤，抓取对应 IP 的数据包
#  -nn      表示不解析 IP 地址和端口号的名称

tcpdump -i eth1 icmp and host 183.232.231.174 -nn
```

那么当 tcpdump 抓取到 icmp 数据包后， 输出格式如下：

```
时间戳 协议 源地址.源端口 > 目的地址.目的端口 网络包详细信息
```

```
#  -i eth1 表示抓取 eth1 网口的数据包
#  icmp    表示抓取 icmp 协议的数据包
#  host    表示主机过滤，抓取对应 IP 的数据包
#  -nn     表示不解析 IP 地址和端口号的名称
$ tcpdump -i eth1 icmp and host 183.232.231.174 -nn
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 65535 bytes
17:20:31.855490 IP 192.168.3.33 > 183.232.231.174: ICMP echo request, id 3341, seq 1, length 64
17:20:31.872698 IP 183.232.231.174 > 192.168.3.33: ICMP echo reply, id 3341, seq 1, length 64
17:20:32.857495 IP 192.168.3.33 > 183.232.231.174: ICMP echo request, id 3341, seq 2, length 64
17:20:32.873195 IP 183.232.231.174 > 192.168.3.33: ICMP echo reply, id 3341, seq 2, length 64
17:20:33.861987 IP 192.168.3.33 > 183.232.231.174: ICMP echo request, id 3341, seq 3, length 64
17:20:33.877174 IP 183.232.231.174 > 192.168.3.33: ICMP echo reply, id 3341, seq 3, length 64
```

从 tcpdump 抓取的 icmp 数据包，我们很清楚的看到 `icmp echo` 的交互过程了，首先发送方发起了 `ICMP echo request` 请求报文，接收方收到后回了一个 `ICMP echo reply` 响应报文，之后 `seq` 是递增的。

我在这里也帮你整理了一些最常见的用法，并且绘制成了表格，你可以参考使用。

首先，先来看看常用的选项类，在上面的 ping 例子中，我们用过 `-i` 选项指定网口，用过 `-nn` 选项不对 IP 地址和端口名称解析。其他常用的选项，如下表格：

tcpdump 使用 —— 选项类		
选项	**示例**	**说明**
-i	tcpdump -i eth0	指定网络接口，默认是 0 浩接口（如 eth0 ），any 表示所有接口
-nn	tcpdump -nn	不解析 IP 地址和端口号的名称
-c	tcpdump -c 5	限制要抓取的网络包的个数
-w	tcpdump -w file.pcap	保持到文件中，文件名通常以 .pcap 为后缀

接下来，我们再来看看常用的过滤表用法，在上面的 ping 例子中，我们用过的是 `icmp and host 183.232.231.174` ，表示抓取 icmp 协议的数据包，以及源地址或目标地址为 183.232.231.174 的包。其他常用的过滤选项，我也整理成了下面这个表格。

tcpdump 使用 —— 过滤表达式类		
选项	示例	说明
host、src host、dst host	tcpdump -nn host 192.168.1.100	主机过滤
port、src port、dst port	tcpdump -nn port 80	端口过滤
ip、ip6、arp、tcp、udp、icmp	tcpdump -nn tcp	协议过滤
and、or、not	tcpdump -nn host 192.168.1.100 and port 80	逻辑表达式
tcp[tcoflages]	tcpdump -nn "tcp[tcpflags] & tcp-syn != 0"	特定状态的 TCP 包

说了这么多，你应该也发现了，tcpdump 虽然功能强大，但是输出的格式并不直观。

所以，在工作中 tcpdump 只是用来抓取数据包，不用来分析数据包，而是把 tcpdump 抓取的数据包保存成 pcap 后缀的文件，接着用 Wireshark 工具进行数据包分析。

> Wireshark 工具如何分析数据包?

Wireshark 除了可以抓包外，还提供了可视化分析网络包的图形页面，同时，还内置了一系列的汇总分析工具。

比如，拿上面的 ping 例子来说，我们可以使用下面的命令，把抓取的数据包保存到 ping.pcap 文件

```
tcpdump -i eth1 icmp and host 183.232.231.174 -w ping.pcap
```

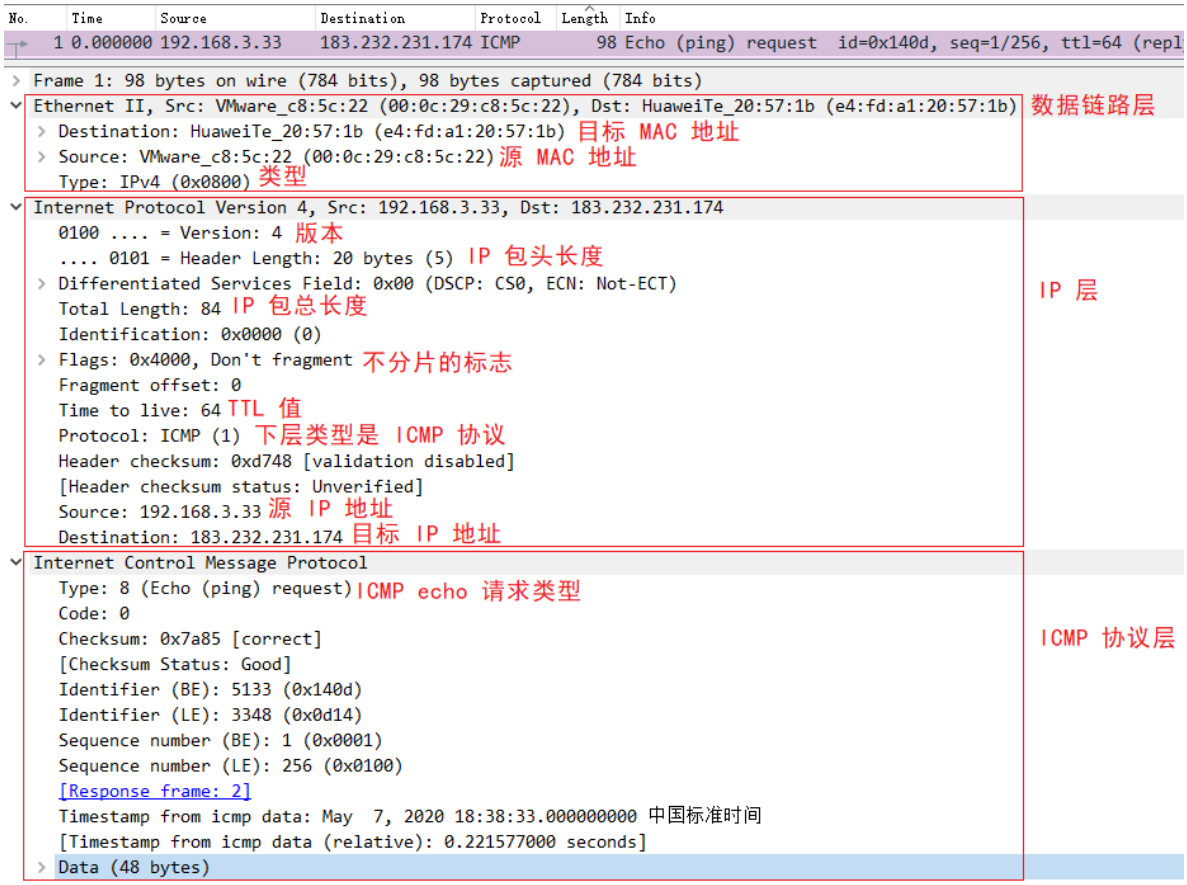
接着把 ping.pcap 文件拖到电脑，再用 Wireshark 打开它。打开后，你就可以看到下面这个界面：

编号 时间 源地址 目标地址 协议 包长度 网络包信息

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.3.33	183.232.231.174	ICMP	98	Echo (ping) request id=0x140d, seq=1/256, ttl=64 (reply in 2)
2	0.015866	183.232.231.174	192.168.3.33	ICMP	98	Echo (ping) reply id=0x140d, seq=1/256, ttl=56 (request in 1)
3	0.999765	192.168.3.33	183.232.231.174	ICMP	98	Echo (ping) request id=0x140d, seq=2/512, ttl=64 (reply in 4)
4	1.014721	183.232.231.174	192.168.3.33	ICMP	98	Echo (ping) reply id=0x140d, seq=2/512, ttl=56 (request in 3)
5	2.000958	192.168.3.33	183.232.231.174	ICMP	98	Echo (ping) request id=0x140d, seq=3/768, ttl=64 (reply in 6)
6	2.016625	183.232.231.174	192.168.3.33	ICMP	98	Echo (ping) reply id=0x140d, seq=3/768, ttl=56 (request in 5)

是吧？在 Wireshark 的页面里，可以更加直观的分析数据包，不仅展示各个网络包的头部信息，还会用不同的颜色来区分不同的协议，由于这次抓包只有 ICMP 协议，所以只有紫色的条目。

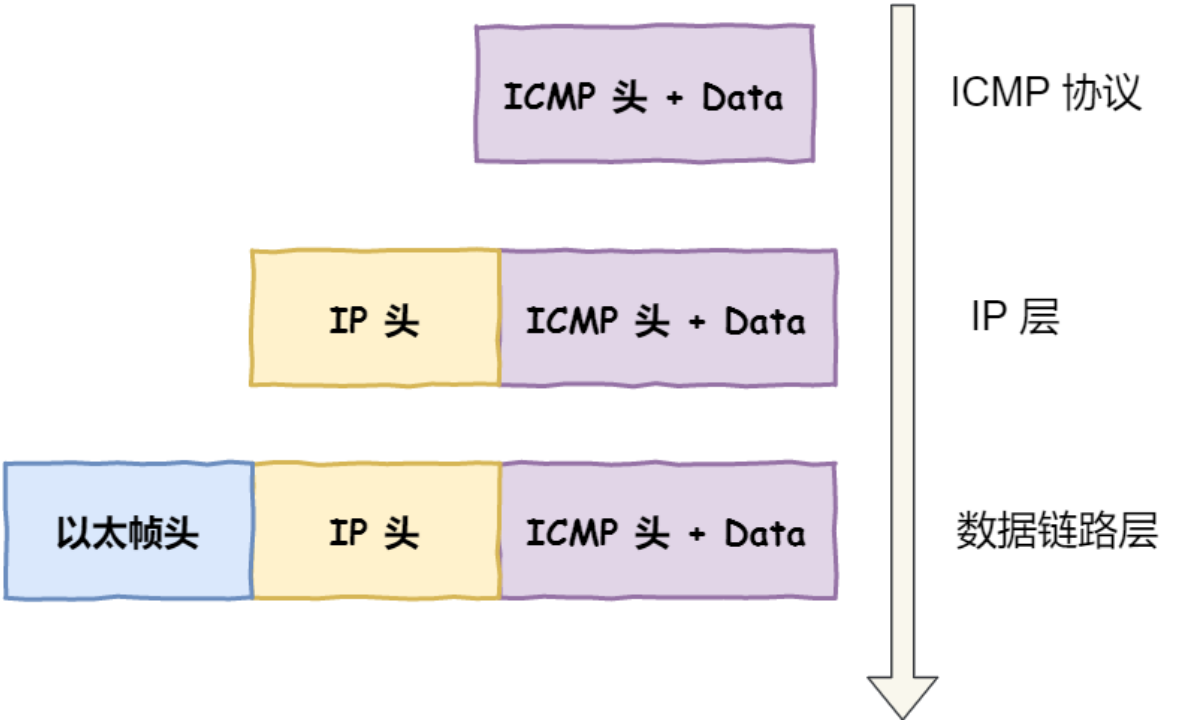
接着，在网络包列表中选择某一个网络包后，在其下面的网络包详情中，**可以更清楚的看到，这个网络包在协议栈各层的详细信息**。比如，以编号 1 的网络包为例子：



- 可以在数据链路层，看到 MAC 包头信息，如源 MAC 地址和目标 MAC 地址等字段；
- 可以在 IP 层，看到 IP 包头信息，如源 IP 地址和目标 IP 地址、TTL、IP 包长度、协议等 IP 协议各个字段的数值和含义；
- 可以在 ICMP 层，看到 ICMP 包头信息，比如 Type、Code 等 ICMP 协议各个字段的数值和含义；

Wireshark 用了分层的方式，展示了各个层的包头信息，把“不可见”的数据包，清清楚楚的展示了给我们，还有理由学不好计算机网络吗？是不是**相见恨晚**？

从 ping 的例子中，我们可以看到网络分层就像有序的分工，每一层都有自己的责任范围和信息，上层协议完成工作后就交给下一层，最终形成一个完整的网络包。



解密 TCP 三次握手和四次挥手

既然学会了 tcpdump 和 Wireshark 两大网络分析利器，那我们快马加鞭，接下用它俩抓取和分析 HTTP 协议网络包，并理解 TCP 三次握手和四次挥手的工作原理。

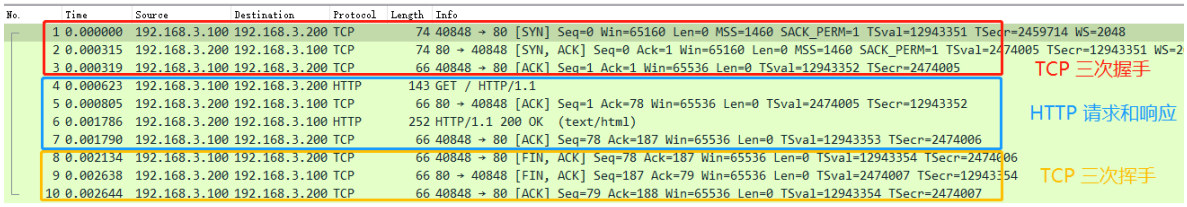
本次例子，我们将要访问的 http://192.168.3.200 服务端。在终端一用 tcpdump 命令抓取数据包：

```
# 客户端执行 tcpdump 抓包
$ tcpdump -i any tcp and host 192.168.3.200 and port 80 -w http.pcap
```

接着，在终端二执行下面的 curl 命令：

最后，回到终端一，按下 Ctrl+C 停止 tcpdump，并把得到的 http.pcap 取出到电脑。

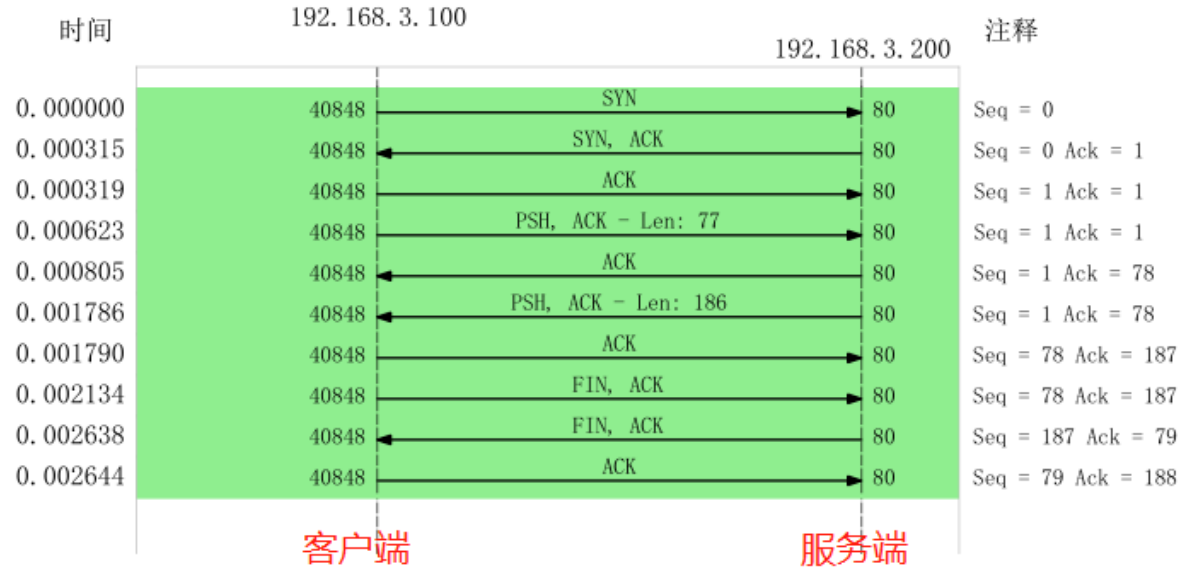
使用 Wireshark 打开 http.pcap 后，你就可以在 Wireshark 中，看到如下的界面：



我们都知道 HTTP 是基于 TCP 协议进行传输的，那么：

- 最开始的 3 个包就是 TCP 三次握手建立连接的包
- 中间是 HTTP 请求和响应的包
- 而最后的 3 个包则是 TCP 断开连接的挥手包

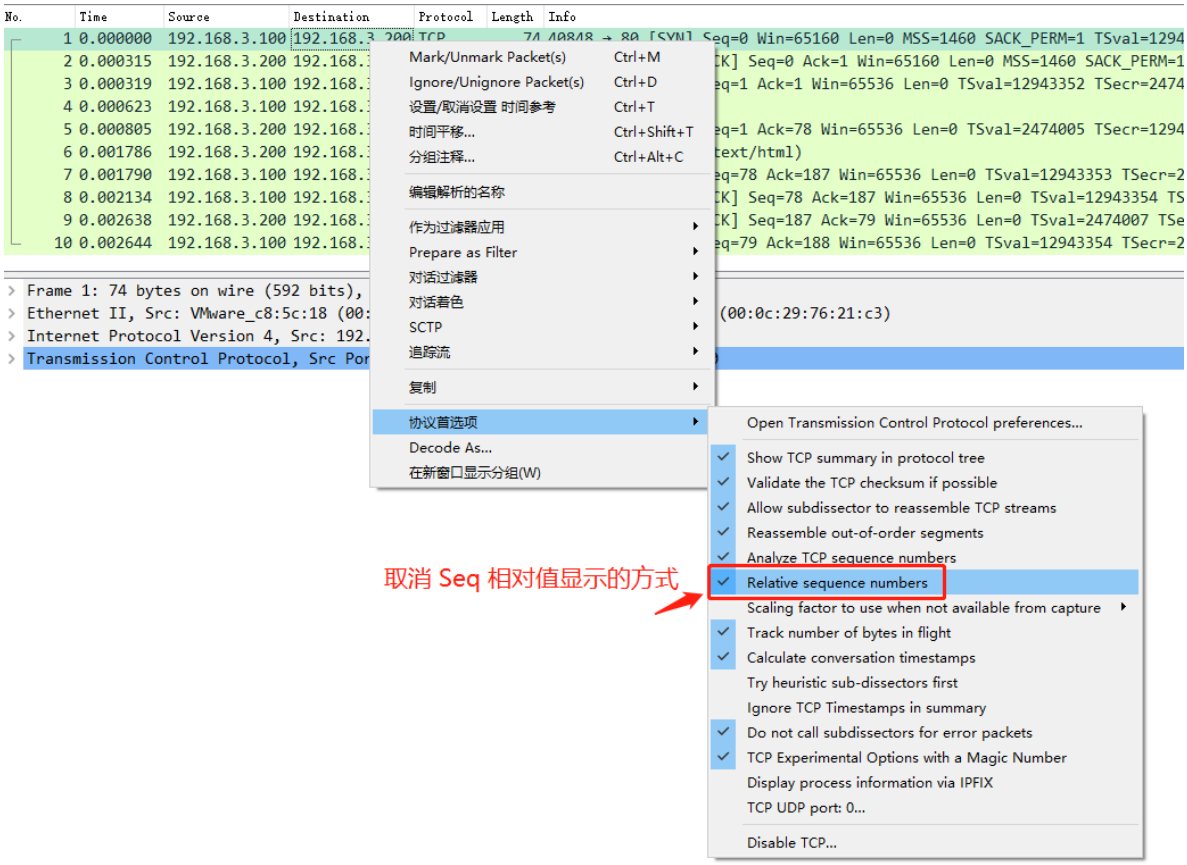
Wireshark 可以用时序图的方式显示数据包交互的过程，从菜单栏中，点击 统计 (Statistics) -> 流量图 (Flow Graph)，然后，在弹出的界面中的「流量类型」选择 「TCP Flows」，你可以更清晰的看到，整个过程中 TCP 流的执行过程：



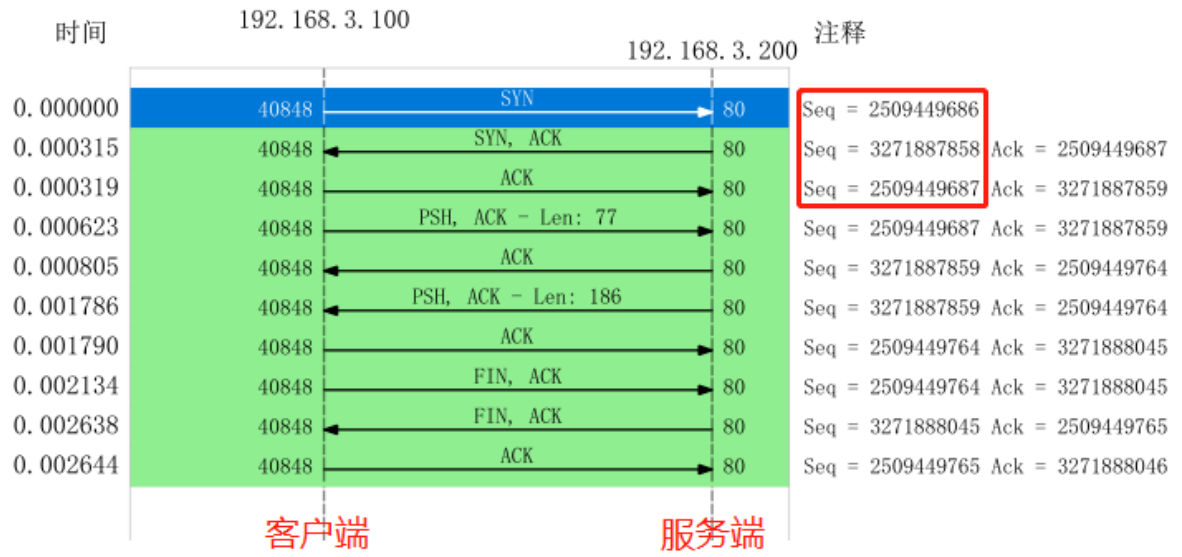
> 你可能会好奇，为什么三次握手连接过程的 Seq 是 0？

实际上是因为 Wireshark 工具帮我们做了优化，它默认显示的是序列号 seq 是相对值，而不是真实值。

如果你想看到实际的序列号的值，可以右键菜单， 然后找到「协议首选项」，接着找到「Relative Seq」后，把它给取消，操作如下：

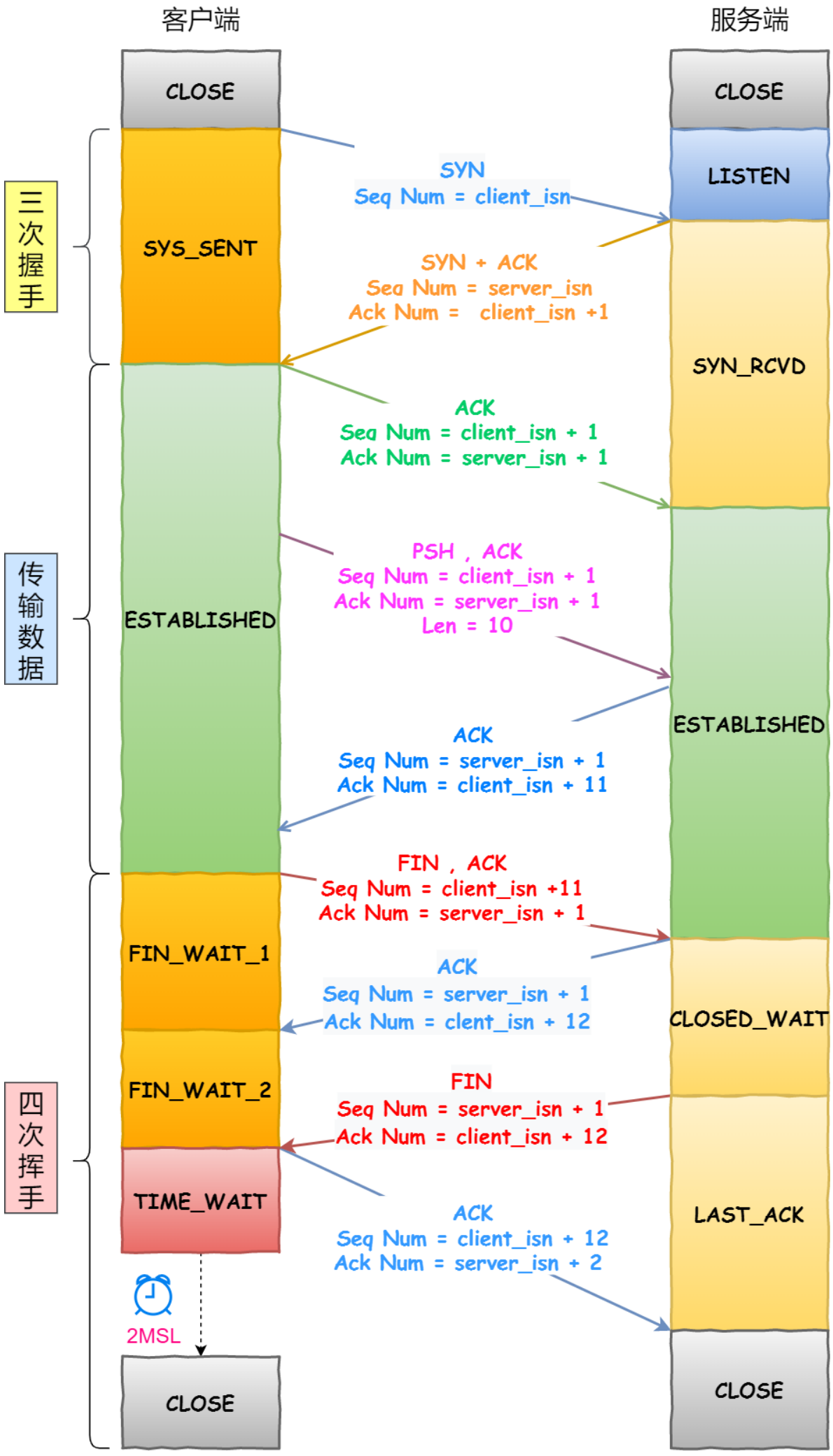


取消后，Seq 显示的就是真实值了：



可见，客户端和服务端的序列号实际上是不同的，序列号是一个随机值。

这其实跟我们书上看到的 TCP 三次握手和四次挥手很类似，作为对比，你通常看到的 TCP 三次握手和四次挥手的流程，基本是这样的：


客户端
服务端
CLOSE
CLOSE
三次握手
SYS_SENT
SYN
Seq Num = client_isn
LISTEN
SYN + ACK
Seq Num = server_isn
Ack Num = client_isn +1
SYN_RCVD
ACK
Seq Num = client_isn + 1
Ack Num = server_isn + 1
传输数据
ESTABLISHED
PSH , ACK
Seq Num = client_isn + 1
Ack Num = server_isn + 1
Len = 10
ESTABLISHED
ACK
Seq Num = server_isn + 1
Ack Num = client_isn + 11
FIN , ACK
Seq Num = client_isn +11
Ack Num = server_isn + 1
FIN_WAIT_1
ACK
Seq Num = server_isn + 1
Ack Num = clent_isn + 12
CLOSED_WAIT
四次挥手
FIN_WAIT_2
FIN
Seq Num = server_isn + 1
Ack Num = client_isn + 12
TIME_WAIT
LAST_ACK
ACK
Seq Num = client_isn + 12
Ack Num = server_isn + 2
2MSL
CLOSE
CLOSE

client_isn 是客户端初始化的序列号
server_isn 是服务端初始化的序列号

> 为什么抓到的 TCP 挥手是三次，而不是书上说的四次？

因为服务器端收到客户端的 `FIN` 后，服务器端同时也要关闭连接，这样就可以把 `ACK` 和 `FIN` 合并到一起发送，节省了一个包，变成了“三次挥手”。

而通常情况下，服务器端收到客户端的 `FIN` 后，很可能还没发送完数据，所以就会先回复客户端一个 `ACK` 包，稍等一会儿，完成所有数据包的发送后，才会发送 `FIN` 包，这也就是四次挥手了。

如下图，就是四次挥手的过程：

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	65.208.228.…	145.254.160…	TCP	54	80 → 3372 [FIN, ACK] Seq=290236744 Ack=951058419 Win=6432 Len=0
2	0.000000	145.254.160…	65.208.228.…	TCP	54	3372 → 80 [ACK] Seq=951058419 Ack=290236745 Win=9236 Len=0
3	12.157481	145.254.160…	65.208.228.…	TCP	54	3372 → 80 [FIN, ACK] Seq=951058419 Ack=290236745 Win=9236 Len=0
4	12.487957	65.208.228.…	145.254.160…	TCP	54	80 → 3372 [ACK] Seq=290236745 Ack=951058420 Win=6432 Len=0

TCP 三次握手异常情况实战分析

TCP 三次握手的过程相信大家都背的滚瓜烂熟，那么你有没有想过这三个异常情况：

- **TCP 第一次握手的 SYN 丢包了，会发生了什么？**
- **TCP 第二次握手的 SYN、ACK 丢包了，会发生什么？**
- **TCP 第三次握手的 ACK 包丢了，会发生什么？**

有的小伙伴可能说：“很简单呀，包丢了就会重传嘛。”

那我在继续问你：

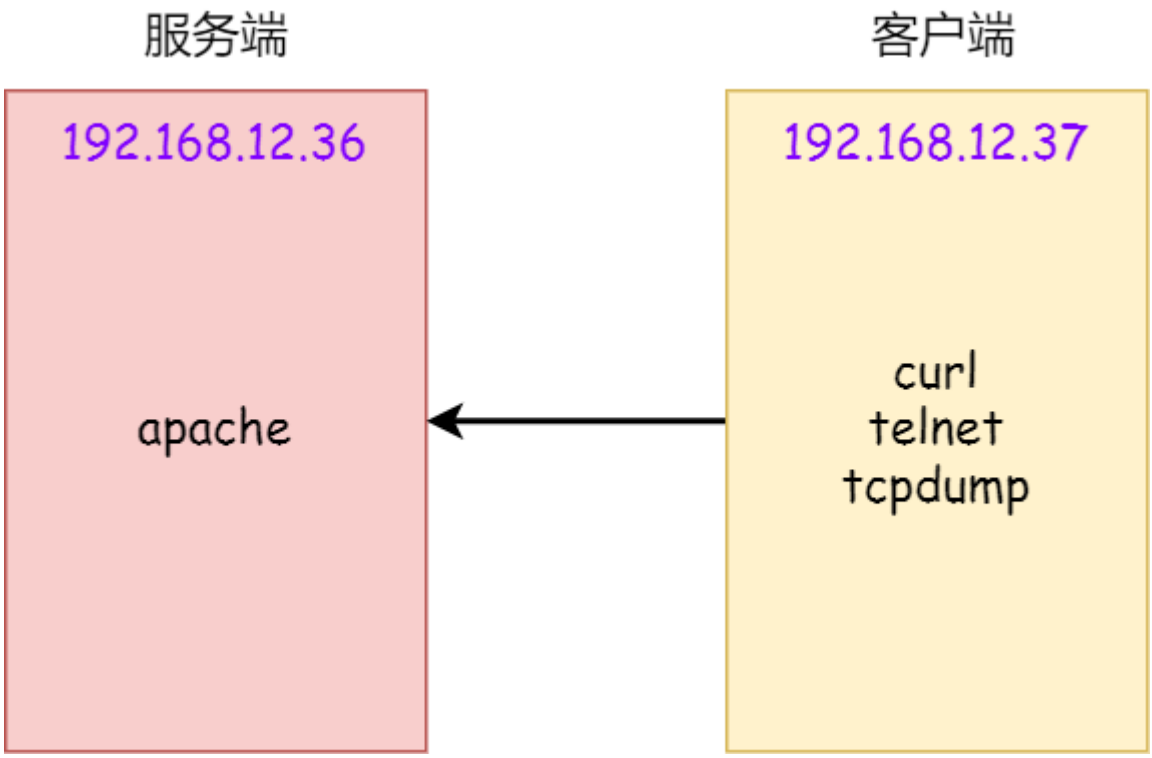
- 那会重传几次？
- 超时重传的时间 RTO 会如何变化？
- 在 Linux 下如何设置重传次数？
- ....

是不是哑口无言，无法回答？

不知道没关系，接下里我用三个实验案例，带大家一起探究探究这三种异常。

实验场景

本次实验用了两台虚拟机，一台作为服务端，一台作为客户端，它们的关系如下：



- 客户端和服务端都是 CentOs 6.5 Linux，Linux 内核版本 2.6.32
- 服务端 192.168.12.36，apache web 服务
- 客户端 192.168.12.37

实验一：TCP 第一次握手 SYN 丢包

为了模拟 TCP 第一次握手 SYN 丢包的情况，我是在拔掉服务器的网线后，立刻在客户端执行 curl 命令：

```
# 客户端发起 curl 请求
$ date;curl http://192.168.12.36;date
Sat May 16 12:47:22 CST 2020
                                    # 阻塞中 ....
```

其间 tcpdump 抓包的命令如下：

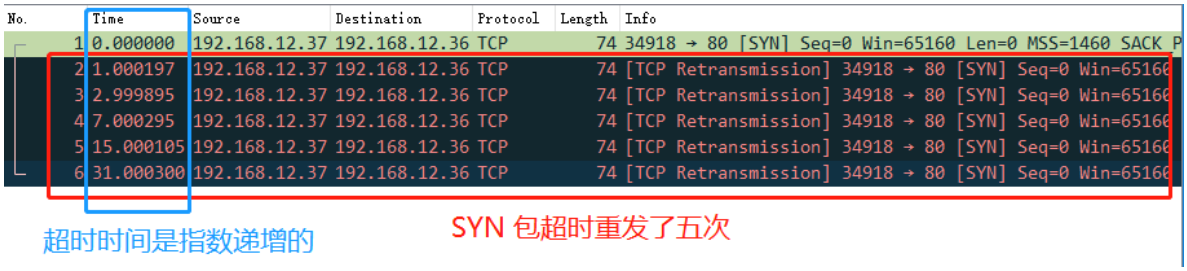
```
# 客户端执行 tcpdump，抓取访问服务端的 HTTP 数据包
$ tcpdump -i eth0 tcp and host 192.168.12.36 and port 80 -w tcp_sys_timeout.pcap
```

过了一会， curl 返回了超时连接的错误：

```
$ date;curl http://192.168.12.36;date
Sat May 16 12:47:22 CST 2020
curl: (7) Failed to connect to 192.168.12.36 port 80: Connection timed out
Sat May 16 12:48:25 CST 2020
```

从 `date` 返回的时间，可以发现在超时接近 1 分钟的时间后，curl 返回了错误。

接着，把 tcp_sys_timeout.pcap 文件用 Wireshark 打开分析，显示如下图：



从上图可以发现， 客户端发起了 SYN 包后，一直没有收到服务端的 ACK ，所以一直超时重传了 5 次，并且每次 RTO 超时时间是不同的：

- 第一次是在 1 秒超时重传
- 第二次是在 3 秒超时重传
- 第三次是在 7 秒超时重传
- 第四次是在 15 秒超时重传
- 第五次是在 31 秒超时重传

可以发现，每次超时时间 RTO 是**指数（翻倍）上涨的**，当超过最大重传次数后，客户端不再发送 SYN 包。

在 Linux 中，第一次握手的 `SYN` 超时重传次数，是如下内核参数指定的：

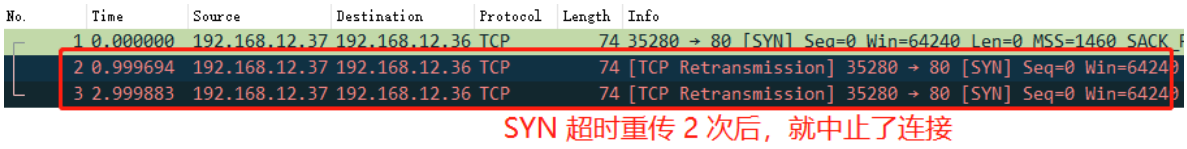
```
$ cat /proc/sys/net/ipv4/tcp_syn_retries
5
```

`tcp_syn_retries` 默认值为 5，也就是 SYN 最大重传次数是 5 次。

接下来，我们继续做实验，把 tcp_syn_retries 设置为 2 次：

```
$ echo 2 > /proc/sys/net/ipv4/tcp_syn_retries
```

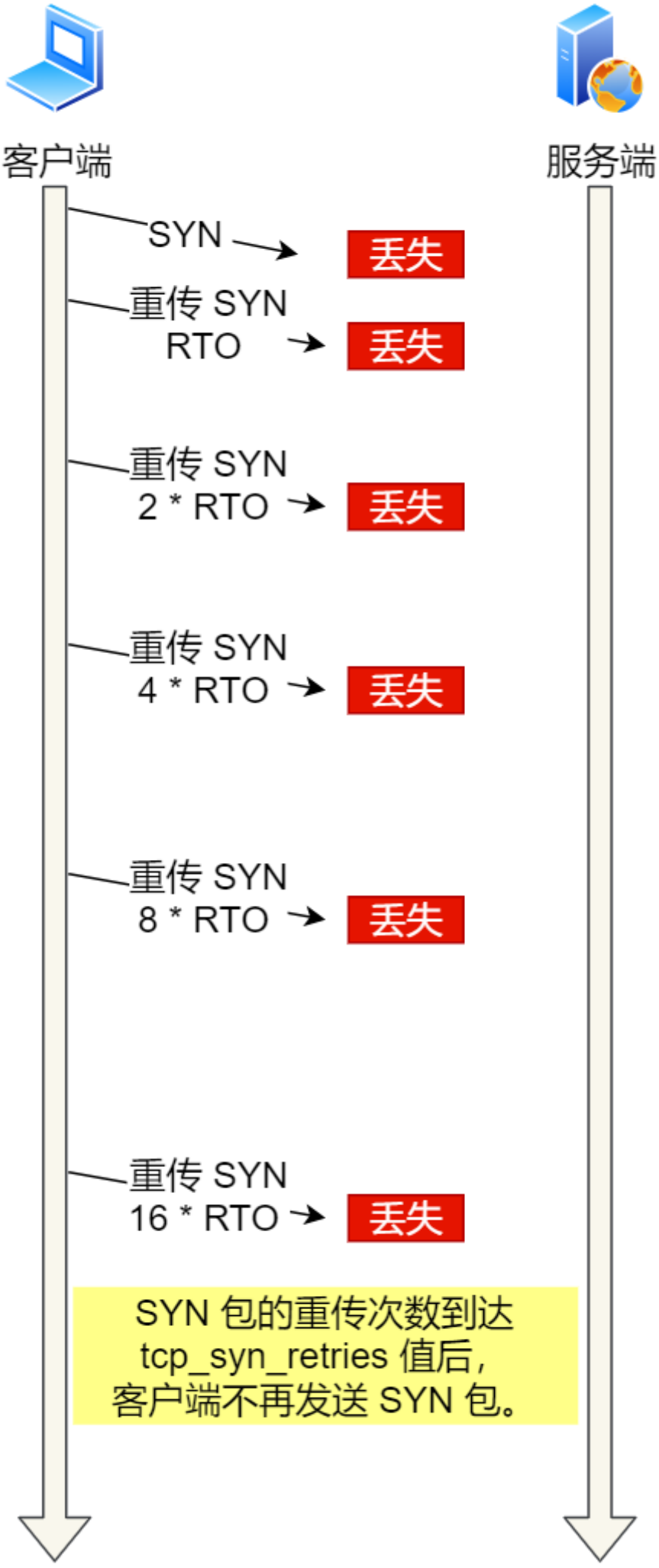
重传抓包后，用 Wireshark 打开分析，显示如下图：



SYN 超时重传 2 次后，就中止了连接

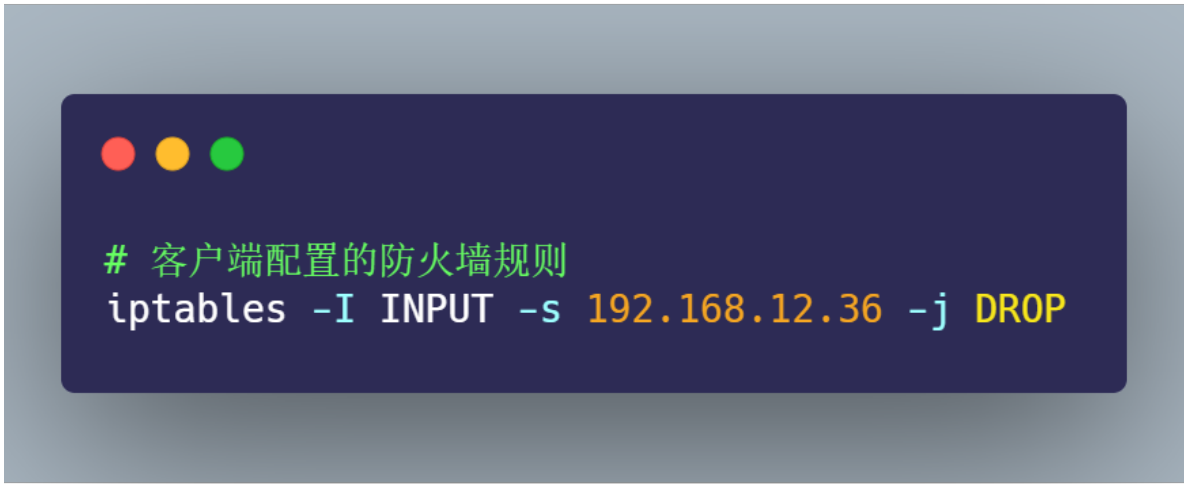
> 实验一的实验小结

通过实验一的实验结果，我们可以得知，当客户端发起的 TCP 第一次握手 SYN 包，在超时时间内没收到服务端的 ACK，就会在超时重传 SYN 数据包，每次超时重传的 RTO 是翻倍上涨的，直到 SYN 包的重传次数到达 tcp_syn_retries 值后，客户端不再发送 SYN 包。

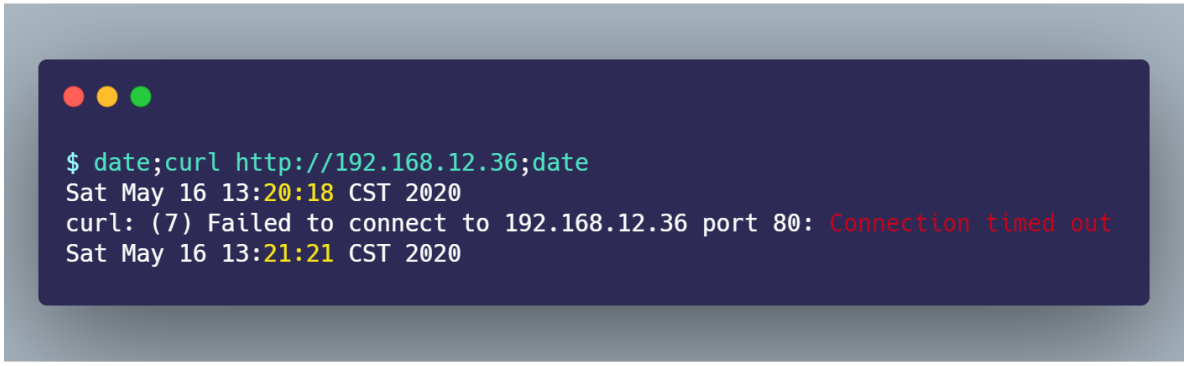

客户端
服务端
SYN
丢失
重传 SYN
RTO
丢失
重传 SYN
2 * RTO
丢失
重传 SYN
4 * RTO
丢失
重传 SYN
8 * RTO
丢失
重传 SYN
16 * RTO
丢失
SYN 包的重传次数到达
tcp_syn_retries 值后，
客户端不再发送 SYN 包。

实验二：TCP 第二次握手 SYN、ACK 丢包

为了模拟客户端收不到服务端第二次握手 SYN、ACK 包，我的做法是在客户端加上防火墙限制，直接粗暴的把来自服务端的数据都丢弃，防火墙的配置如下：



接着，在客户端执行 curl 命令：



从 `date` 返回的时间前后，可以算出大概 1 分钟后，curl 报错退出了。

客户端在这其间抓取的数据包，用 Wireshark 打开分析，显示的时序图如下：

从图中可以发现：

- 客户端发起 SYN 后，由于防火墙屏蔽了服务端的所有数据包，所以 curl 是无法收到服务端的 SYN、ACK 包，当发生超时后，就会重传 SYN 包
- 服务端收到客户的 SYN 包后，就会回 SYN、ACK 包，但是客户端一直没有回 ACK，服务端在超时后，重传了 SYN、ACK 包，**接着一会，客户端超时重传的 SYN 包又抵达了服务端，服务端收到后，超时定时器就重新计时，然后回了 SYN、ACK 包，所以相当于服务端的超时定时器只触发了一次，又被重置了。**
- 最后，客户端 SYN 超时重传次数达到了 5 次（tcp_syn_retries 默认值 5 次），就不再继续发送 SYN 包了。

所以，我们可以发现，**当第二次握手的 SYN、ACK 丢包时，客户端会超时重发 SYN 包，服务端也会超时重传 SYN、ACK 包。**

> 咦？客户端设置了防火墙，屏蔽了服务端的网络包，为什么 tcpdump 还能抓到服务端的网络包？

添加 iptables 限制后， tcpdump 是否能抓到包 ，这要看添加的 iptables 限制条件：

- 如果添加的是 `INPUT` 规则，则可以抓得到包
- 如果添加的是 `OUTPUT` 规则，则抓不到包

网络包进入主机后的顺序如下：

- 进来的顺序 Wire -> NIC -> **tcpdump -> netfilter/iptables**
- 出去的顺序 **iptables -> tcpdump** -> NIC -> Wire

> tcp_syn_retries 是限制 SYN 重传次数，那第二次握手 SYN、ACK 限制最大重传次数是多少？

TCP 第二次握手 SYN、ACK 包的最大重传次数是通过 `tcp_synack_retries` 内核参数限制的，其默认值如下：

```
$ cat /proc/sys/net/ipv4/tcp_synack_retries
5
```

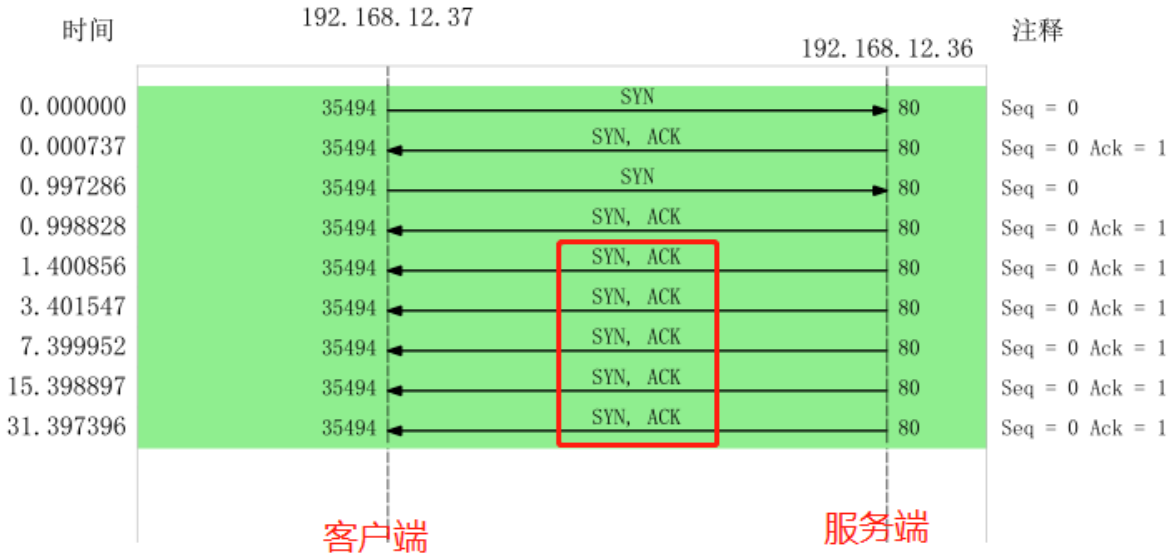
是的，TCP 第二次握手 SYN、ACK 包的最大重传次数默认值是 `5` 次。

为了验证 SYN、ACK 包最大重传次数是 5 次，我们继续做下实验，我们先把客户端的 `tcp_syn_retries` 设置为 1，表示客户端 SYN 最大超时次数是 1 次，目的是为了防止多次重传 SYN，把服务端 SYN、ACK 超时定时器重置。

接着，还是如上面的步骤：

1. 客户端配置防火墙屏蔽服务端的数据包
2. 客户端 tcpdump 抓取 curl 执行时的数据包

把抓取的数据包，用 Wireshark 打开分析，显示的时序图如下：



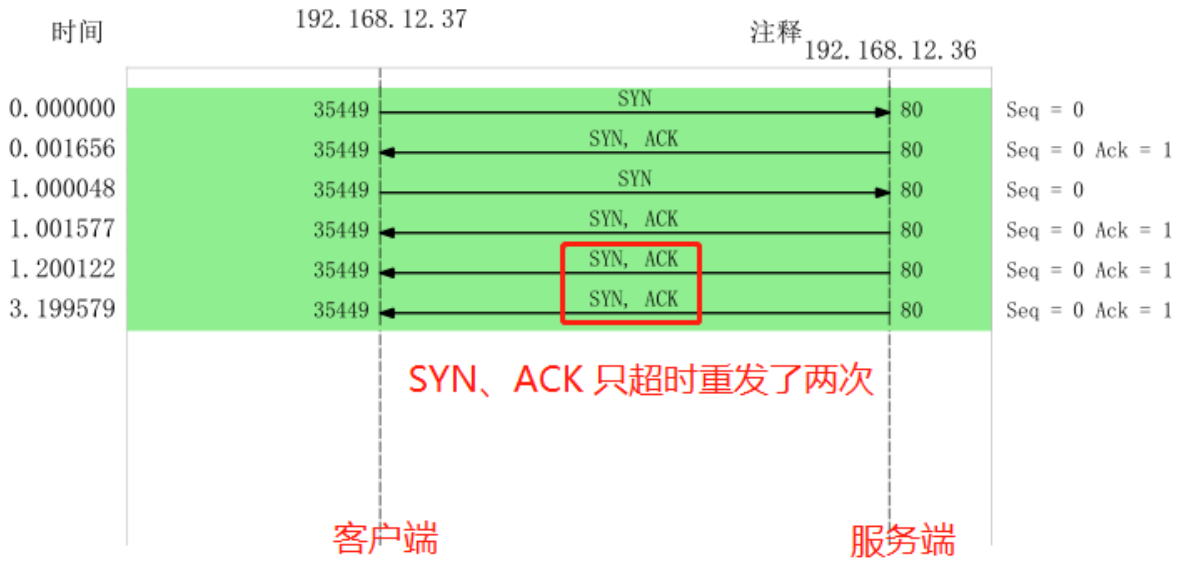
从上图，我们可以分析出：

- 客户端的 SYN 只超时重传了 1 次，因为 `tcp_syn_retries` 值为 1
- 服务端应答了客户端超时重传的 SYN 包后，由于一直收不到客户端的 ACK 包，所以服务端一直在超时重传 SYN、ACK 包，每次的 RTO 也是指数上涨的，一共超时重传了 5 次，因为 `tcp_synack_retries` 值为 5

接着，我把 **tcp_synack_retries 设置为 2**，`tcp_syn_retries` 依然设置为 1：

```
$ echo 2 > /proc/sys/net/ipv4/tcp_synack_retries
$ echo 1 > /proc/sys/net/ipv4/tcp_syn_retries
```

依然保持一样的实验步骤进行操作，接着把抓取的数据包，用 Wireshark 打开分析，显示的时序图如下：



可见：

- 客户端的 SYN 包只超时重传了 1 次，符合 tcp_syn_retries 设置的值；
- 服务端的 SYN、ACK 超时重传了 2 次，符合 tcp_synack_retries 设置的值

> 实验二的实验小结

通过实验二的实验结果，我们可以得知，当 TCP 第二次握手 SYN、ACK 包丢了后，客户端 SYN 包会发生超时重传，服务端 SYN、ACK 也会发生超时重传。

客户端 SYN 包超时重传的最大次数，是由 tcp_syn_retries 决定的，默认值是 5 次；服务端 SYN、ACK 包时重传的最大次数，是由 tcp_synack_retries 决定的，默认值是 5 次。

实验三：TCP 第三次握手 ACK 丢包

为了模拟 TCP 第三次握手 ACK 包丢，我的实验方法是在服务端配置防火墙，屏蔽客户端 TCP 报文中标志位是 ACK 的包，也就是当服务端收到客户端的 TCP ACK 的报文时就会丢弃，iptables 配置命令如下：

```
# 服务端配置防火墙
$ iptables -I INPUT -s 192.168.12.37 -p tcp --tcp-flag ACK ACK -j DROP
```

接着，在客户端执行如下 tcpdump 命令：

```
# 客户端执行 tcpdump 抓取数据包
tcpdump -i eth0 tcp and host 192.168.12.36 and port 80 -w tcp_thir_ack_timeout.pcap
```

然后，客户端向服务端发起 telnet，因为 telnet 命令是会发起 TCP 连接，所以用此命令做测试：

```
# 客户端执行 telnet
$ telnet 192.168.12.36 80
Trying 192.168.12.36...
Connected to 192.168.12.36.
Escape character is '^]'.
                              #  阻塞中....
```

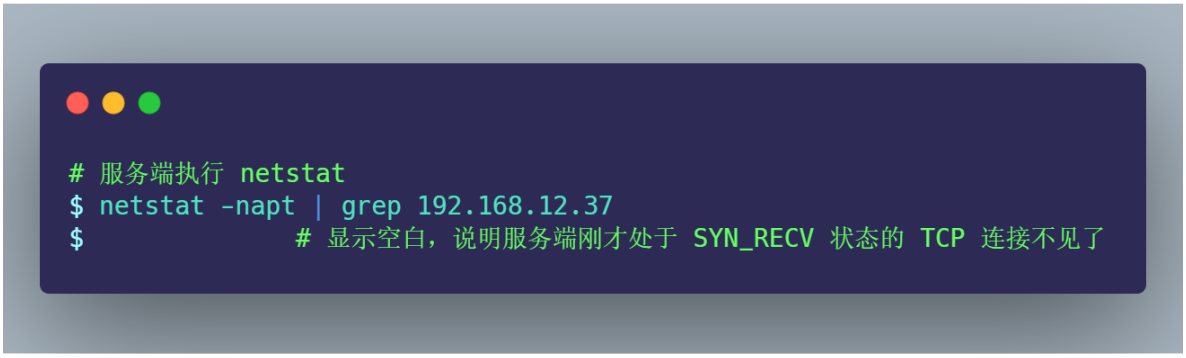
此时，由于服务端收不到第三次握手的 ACK 包，所以一直处于 `SYN_RECV` 状态：

```
# 服务端执行 netstat
$ netstat -napt | grep 192.168.12.37
tcp        0      0 192.168.12.36:80          192.168.12.37:36008         SYN_RECV    -
```

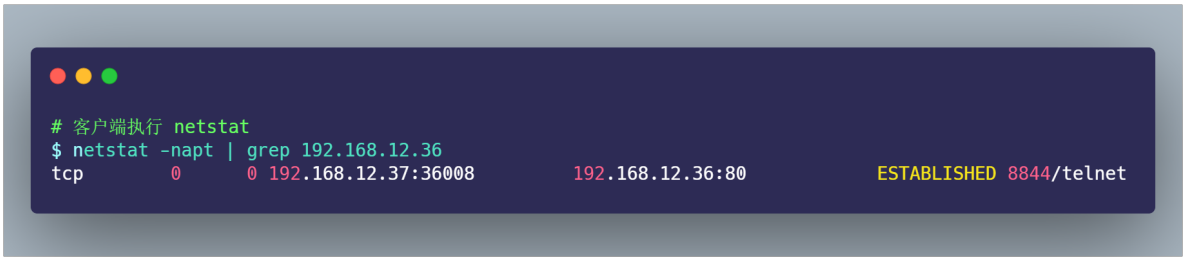
而客户端是已完成 TCP 连接建立，处于 `ESTABLISHED` 状态：

```
# 客户端执行 netstat
$ netstat -napt | grep 192.168.12.36
tcp        0      0 192.168.12.37:36008       192.168.12.36:80            ESTABLISHED 8844/telnet
```

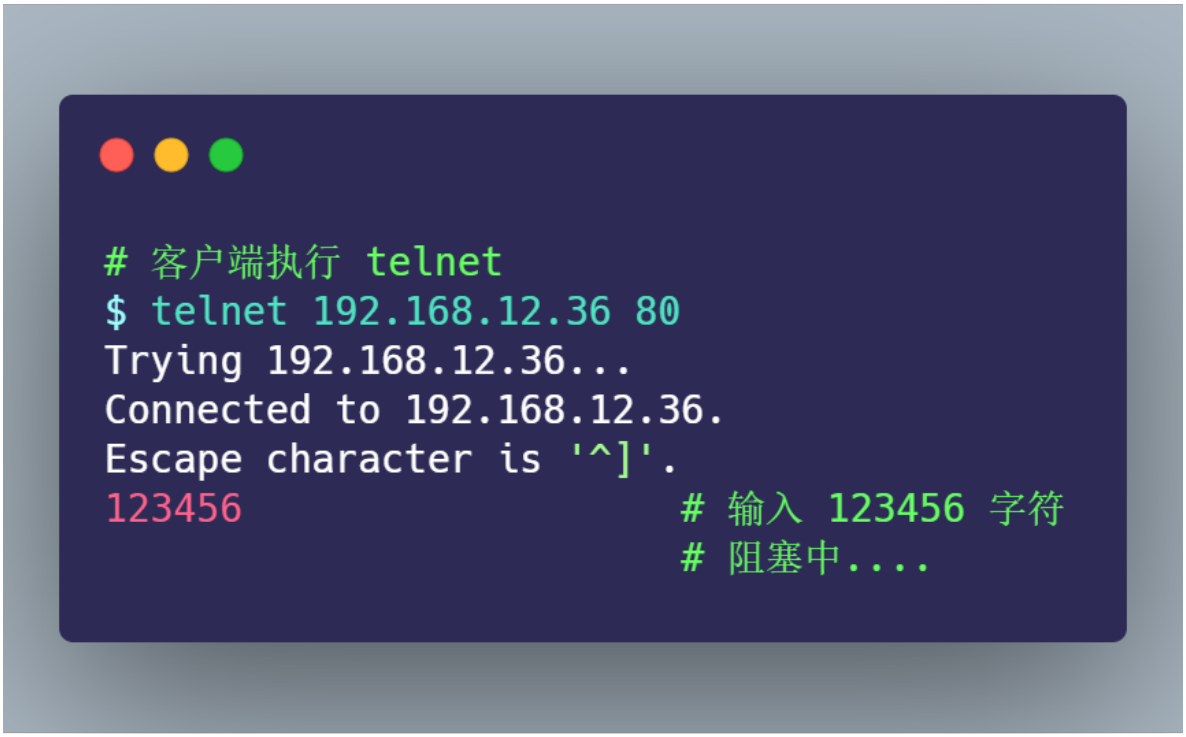
过了 1 分钟后，观察发现服务端的 TCP 连接不见了：



过了 30 分别，客户端依然还是处于 `ESTABLISHED` 状态：



接着，在刚才客户端建立的 telnet 会话，输入 123456 字符，进行发送：

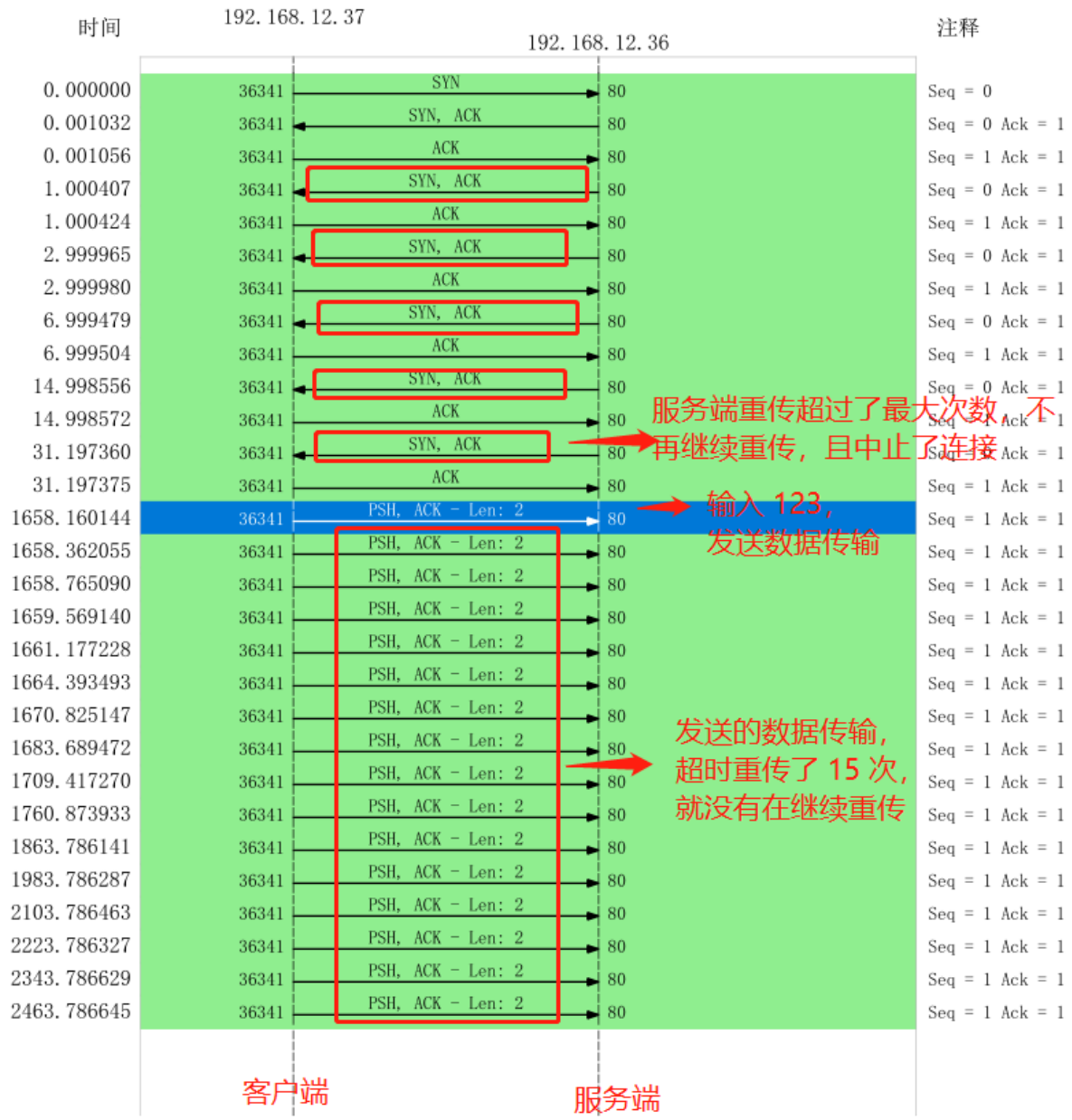


持续「好长」一段时间，客户端的 telnet 才断开连接：

以上就是本次的实现三的现象，这里存在两个疑点：

- 为什么服务端原本处于 `SYN_RECV` 状态的连接，过 1 分钟后就消失了？
- 为什么客户端 telnet 输入 123456 字符后，过了好长一段时间，telnet 才断开连接？

不着急，我们把刚抓的数据包，用 Wireshark 打开分析，显示的时序图如下：



上图的流程：

- 客户端发送 SYN 包给服务端，服务端收到后，回了个 SYN、ACK 包给客户端，此时服务端的 TCP 连接处于 `SYN_RECV` 状态；
- 客户端收到服务端的 SYN、ACK 包后，给服务端回了个 ACK 包，此时客户端的 TCP 连接处于 `ESTABLISHED` 状态；
- 由于服务端配置了防火墙，屏蔽了客户端的 ACK 包，所以服务端一直处于 `SYN_RECV` 状态，没有进入 `ESTABLISHED` 状态，tcpdump 之所以能抓到客户端的 ACK 包，是因为数据包进入系统的顺序是先进入 tcpudmp，后经过 iptables；
- 接着，服务端超时重传了 SYN、ACK 包，重传了 5 次后，也就是**超过 tcp_synack_retries 的值（默认值是 5），然后就没有继续重传了，此时服务端的 TCP 连接主动中止了，所以刚才处于 SYN_RECV 状态的 TCP 连接断开了**，而客户端依然处于 `ESTABLISHED` 状态；
- 虽然服务端 TCP 断开了，但过了一段时间，发现客户端依然处于 `ESTABLISHED` 状态，于是就在客户端的 telnet 会话输入了 123456 字符；
- 此时由于服务端已经断开连接，**客户端发送的数据报文，一直在超时重传，每一次重传，RTO 的值是指数增长的，所以持续了好长一段时间，客户端的 telnet 才报错退出了，此时共重传了 15 次。**

通过这一波分析，刚才的两个疑点已经解除了：

- 服务端在重传 SYN、ACK 包时，超过了最大重传次数 `tcp_synack_retries`，于是服务端的 TCP 连接主动断开了。
- 客户端向服务端发送数据包时，由于服务端的 TCP 连接已经退出了，所以数据包一直在超时重传，共重传了 15 次， telnet 就断开了连接。

> TCP 第一次握手的 SYN 包超时重传最大次数是由 tcp_syn_retries 指定，TCP 第二次握手的 SYN、ACK 包超时重传最大次数是由 tcp_synack_retries 指定，那 TCP 建立连接后的数据包最大超时重传次数是由什么参数指定呢?

TCP 建立连接后的数据包传输，最大超时重传次数是由 `tcp_retries2` 指定，默认值是 15 次，如下：

```
$ cat /proc/sys/net/ipv4/tcp_retries2
15
```

如果 15 次重传都做完了，TCP 就会告诉应用层说：“搞不定了，包怎么都传不过去！”

> 那如果客户端不发送数据，什么时候才会断开处于 ESTABLISHED 状态的连接?

这里就需要提到 TCP 的 **保活机制**。这个机制的原理是这样的：

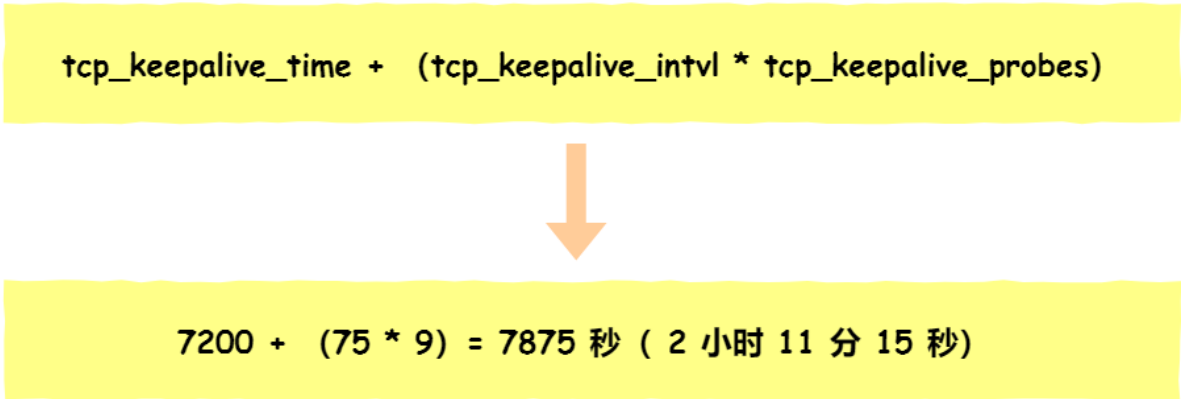
定义一个时间段，在这个时间段内，如果没有任何连接相关的活动，TCP 保活机制会开始作用，每隔一个时间间隔，发送一个「探测报文」，该探测报文包含的数据非常少，如果连续几个探测报文都没有得到响应，则认为当前的 TCP 连接已经死亡，系统内核将错误信息通知给上层应用程序。

在 Linux 内核可以有对应的参数可以设置保活时间、保活探测的次数、保活探测的时间间隔，以下都为默认值：

```
net.ipv4.tcp_keepalive_time=7200
net.ipv4.tcp_keepalive_intvl=75
net.ipv4.tcp_keepalive_probes=9
```

- tcp_keepalive_time=7200：表示保活时间是 7200 秒（2小时），也就 2 小时内如果没有任何连接相关的活动，则会启动保活机制
- tcp_keepalive_intvl=75：表示每次检测间隔 75 秒；
- tcp_keepalive_probes=9：表示检测 9 次无响应，认为对方是不可达的，从而中断本次的连接。

也就是说在 Linux 系统中，最少需要经过 2 小时 11 分 15 秒才可以发现一个「死亡」连接。



这个时间是有点长的，所以如果我抓包足够久，或许能抓到探测报文。

> 实验三的实验小结

在建立 TCP 连接时，如果第三次握手的 ACK，服务端无法收到，则服务端就会短暂处于 `SYN_RECV` 状态，而客户端会处于 `ESTABLISHED` 状态。

由于服务端一直收不到 TCP 第三次握手的 ACK，则会一直重传 SYN、ACK 包，直到重传次数超过 `tcp_synack_retries` 值（默认值 5 次）后，服务端就会断开 TCP 连接。

而客户端则会有两种情况：

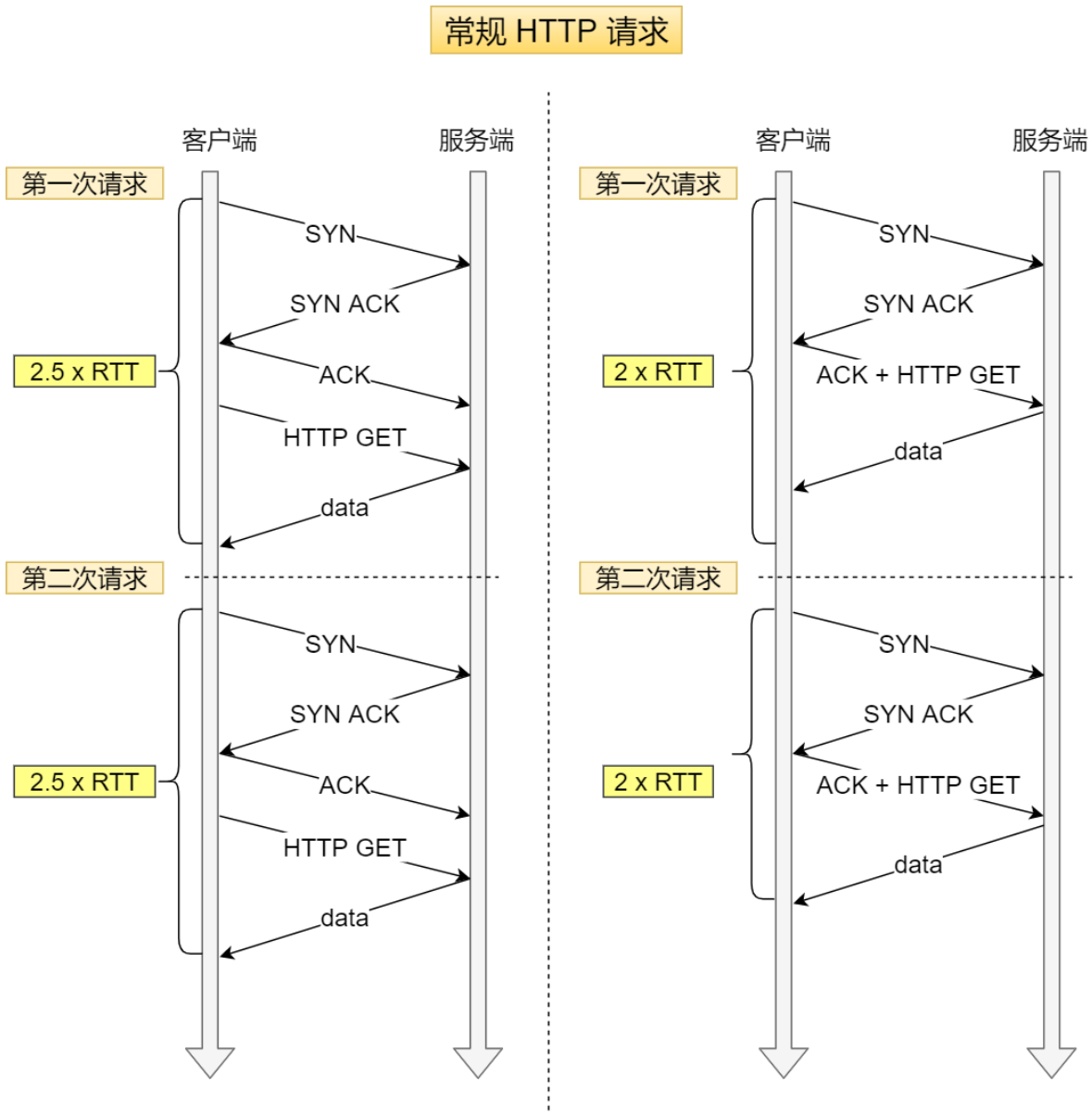
- 如果客户端没发送数据包，一直处于 `ESTABLISHED` 状态，然后经过 2 小时 11 分 15 秒才可以发现一个「死亡」连接，于是客户端连接就会断开连接。
- 如果客户端发送了数据包，一直没有收到服务端对该数据包的确认报文，则会一直重传该数据包，直到重传次数超过 `tcp_retries2` 值（默认值 15 次）后，客户端就会断开 TCP 连接。

TCP 快速建立连接

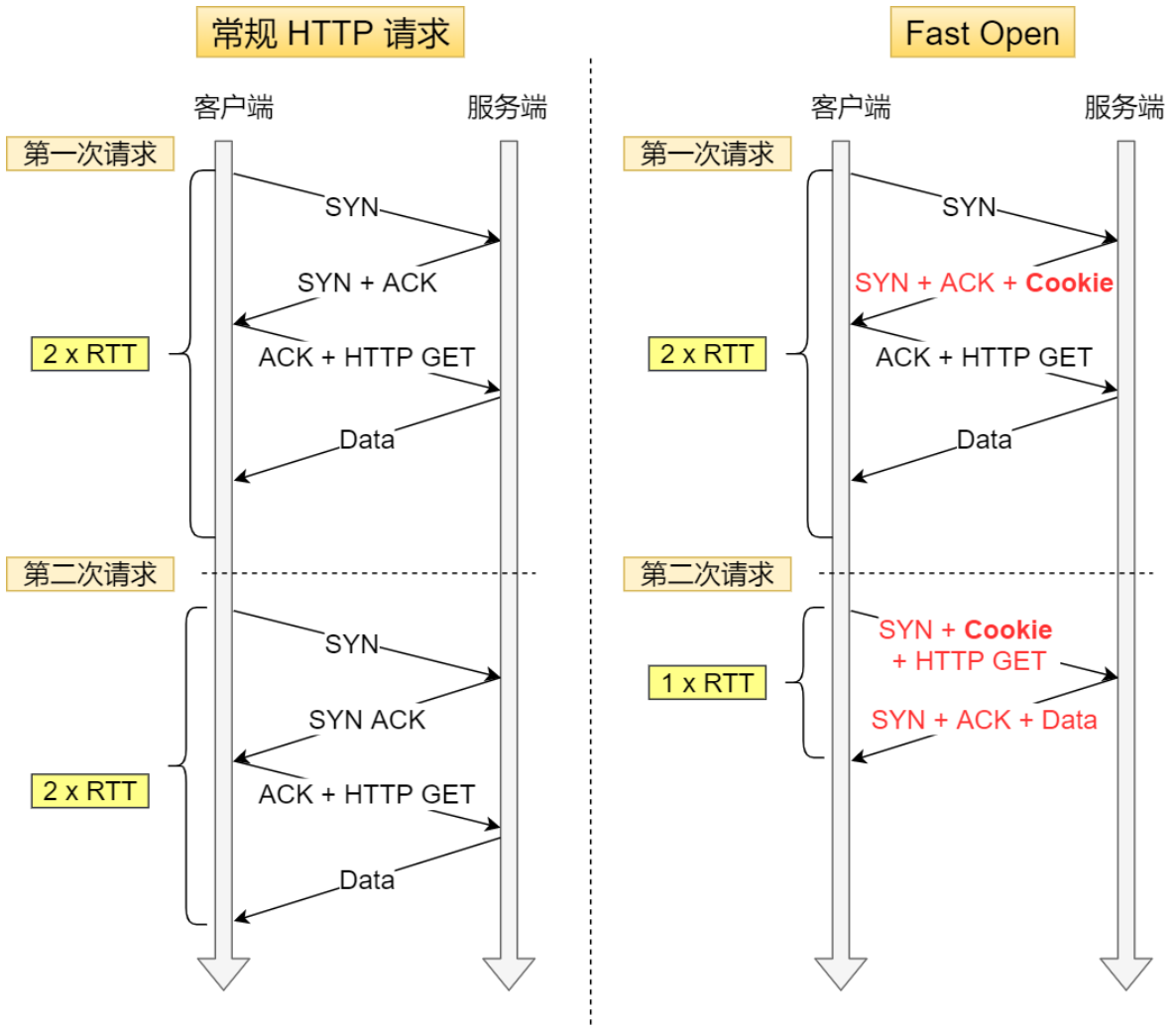
客户端在向服务端发起 HTTP GET 请求时，一个完整的交互过程，需要 2.5 个 RTT 的时延。

由于第三次握手是可以携带数据的，这时如果在第三次握手发起 HTTP GET 请求，需要 2 个 RTT 的时延。

但是在下一次（不是同个 TCP 连接的下一次）发起 HTTP GET 请求时，经历的 RTT 也是一样，如下图：



在 Linux 3.7 内核版本中，提供了 TCP Fast Open 功能，这个功能可以减少 TCP 连接建立的时延。



- 在第一次建立连接的时候，服务端在第二次握手产生一个 `Cookie` （已加密）并通过 SYN、ACK 包一起发给客户端，于是客户端就会缓存这个 `Cookie` ，所以第一次发起 HTTP Get 请求的时候，还是需要 2 个 RTT 的时延；
- 在下次请求的时候，客户端在 SYN 包带上 `Cookie` 发给服务端，就提前可以跳过三次握手的过程，因为 `Cookie` 中维护了一些信息，服务端可以从 `Cookie` 获取 TCP 相关的信息，这时发起的 HTTP GET 请求就只需要 1 个 RTT 的时延；

注：客户端在请求并存储了 Fast Open Cookie 之后，可以不断重复 TCP Fast Open 直至服务器认为 Cookie 无效（通常为过期）

> 在 Linux 上如何打开 Fast Open 功能?

可以通过设置 `net.ipv4.tcp_fastopn` 内核参数，来打开 Fast Open 功能。

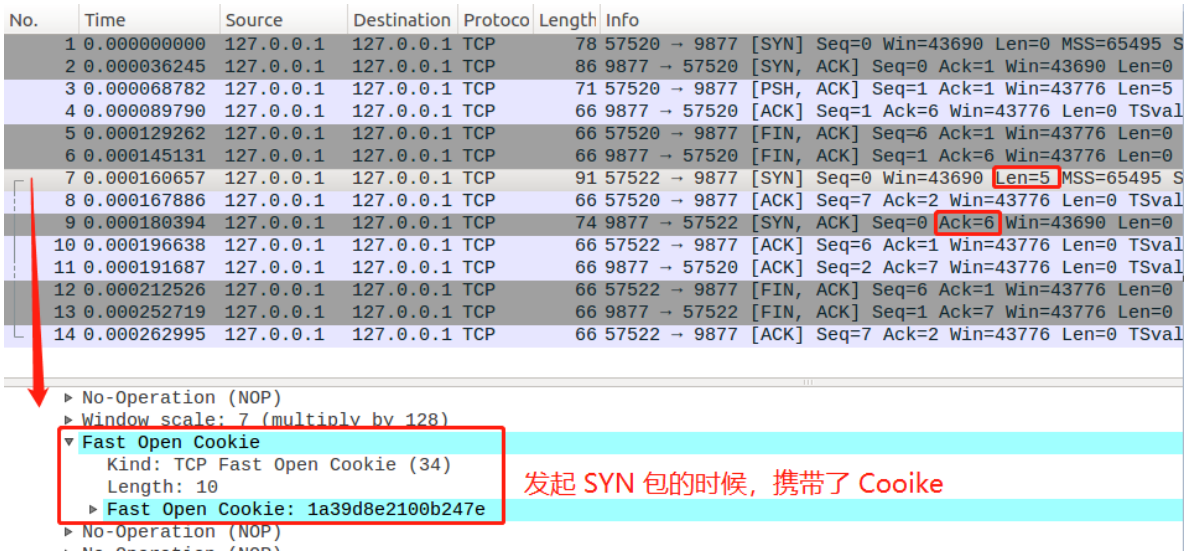
net.ipv4.tcp_fastopn 各个值的意义:

- 0 关闭
- 1 作为客户端使用 Fast Open 功能
- 2 作为服务端使用 Fast Open 功能
- 3 无论作为客户端还是服务器，都可以使用 Fast Open 功能

> TCP Fast Open 抓包分析

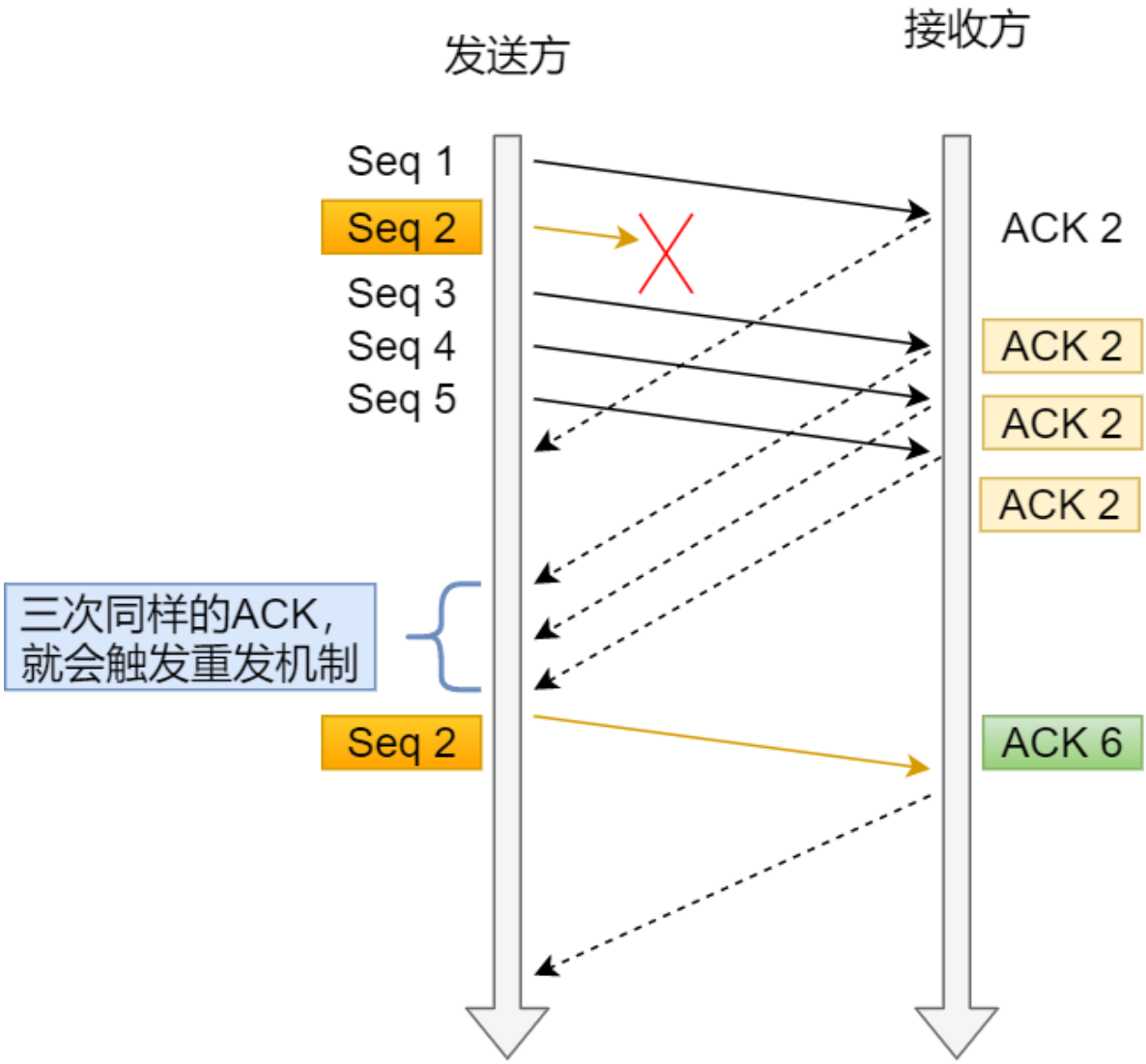
在下图，数据包 7 号，客户端发起了第二次 TCP 连接时，SYN 包会携带 Cooike，并且长度为 5 的数据。

服务端收到后，校验 Cooike 合法，于是就回了 SYN、ACK 包，并且确认应答收到了客户端的数据包，ACK = 5 + 1 = 6



TCP 重复确认和快速重传

当接收方收到乱序数据包时，会发送重复的 ACK，以便告知发送方要重发该数据包，**当发送方收到 3 个重复 ACK 时，就会触发快速重传，立刻重发丢失数据包。**



TCP 重复确认和快速重传的一个案例，用 Wireshark 分析，显示如下：

```
No.  Time     Source          Destination     Protocol  Length  Info
  1  0.000000 172.31.136.85   195.81.202.68   TCP         78    38760 → 80 [ACK] Seq=1 Ack=1 Win=382 Len=0 TSval=22247173 TSecr=1332354
  2  0.000190 195.81.202.68   172.31.136.85   TCP       1434    80 → 38760 [ACK] Seq=10945 Ack=1 Win=108 Len=1368 TSval=1332354 TSecr=22
  3  0.000201 172.31.136.85   195.81.202.68   TCP         78    [TCP Dup ACK 1#1] 38760 → 80 [ACK] Seq=1 Ack=1 Win=382 Len=0 TSval=22247
  4  0.000294 195.81.202.68   172.31.136.85   TCP       1434    80 → 38760 [ACK] Seq=12313 Ack=1 Win=108 Len=1368 TSval=1332354 TSecr=22
  5  0.000304 172.31.136.85   195.81.202.68   TCP         78    [TCP Dup ACK 1#2] 38760 → 80 [ACK] Seq=1 Ack=1 Win=382 Len=0 TSval=22247
  6  0.000425 195.81.202.68   172.31.136.85   TCP       1434    80 → 38760 [ACK] Seq=13681 Ack=1 Win=108 Len=1368 TSval=1332354 TSecr=22
  7  0.000435 172.31.136.85   195.81.202.68   TCP         78    [TCP Dup ACK 1#3] 38760 → 80 [ACK] Seq=1 Ack=1 Win=382 Len=0 TSval=22247
  8  0.000527 195.81.202.68   172.31.136.85   TCP       1434    [TCP Fast Retransmission] 80 → 38760 [ACK] Seq=1 Ack=1 Win=108 Len=1368
  9  0.000537 172.31.136.85   195.81.202.68   TCP         78    38760 → 80 [ACK] Seq=1 Ack=1369 Win=361 Len=0 TSval=22247173 TSecr=13323

> Frame 1: 78 bytes on wire (624 bits), 78 bytes captured (624 bits)
> Ethernet II, Src: HewlettP_a4:c1:c6 (00:16:35:a4:c1:c6), Dst: All-HSRP-routers_01 (00:00:0c:07:ac:01)
> Internet Protocol Version 4, Src: 172.31.136.85, Dst: 195.81.202.68
v Transmission Control Protocol, Src Port: 38760, Dst Port: 80, Seq: 1, Ack: 1, Len: 0
    Source Port: 38760
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 1    (relative sequence number)
    Sequence number (raw): 704338729
    [Next sequence number: 1    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    Acknowledgment number (raw): 1310973186
    1011 .... = Header Length: 44 bytes (11)
```

Seq + Len = 1 + 1368

编号 1 数据包期望的下一个 Seq 是 1

- 数据包 1 期望的下一个数据包 Seq 是 1，但是数据包 2 发送的 Seq 却是 10945，说明收到的是乱序数据包，于是回了数据包 3 ，还是同样的 Seq = 1，Ack = 1，这表明是重复的 ACK；
- 数据包 4 和 6 依然是乱序的数据包，于是依然回了重复的 ACK；
- 当对方收到三次重复的 ACK 后，于是就快速重传了 Seq = 1 、Len = 1368 的数据包 8；

- 当收到重传的数据包后，发现 Seq = 1 是期望的数据包，于是就发送了个确认收到快速重传的 ACK

注意：快速重传和重复 ACK 标记信息是 Wireshark 的功能，非数据包本身的信息。

以上案例在 TCP 三次握手时协商开启了**选择性确认 SACK**，因此一旦数据包丢失并收到重复 ACK，即使在丢失数据包之后还成功接收了其他数据包，也只需要重传丢失的数据包。如果不启用 SACK，就必须重传丢失包之后的每个数据包。

如果要支持 `SACK`，必须双方都要支持。在 Linux 下，可以通过 `net.ipv4.tcp_sack` 参数打开这个功能（Linux 2.4 后默认打开）。

TCP 流量控制

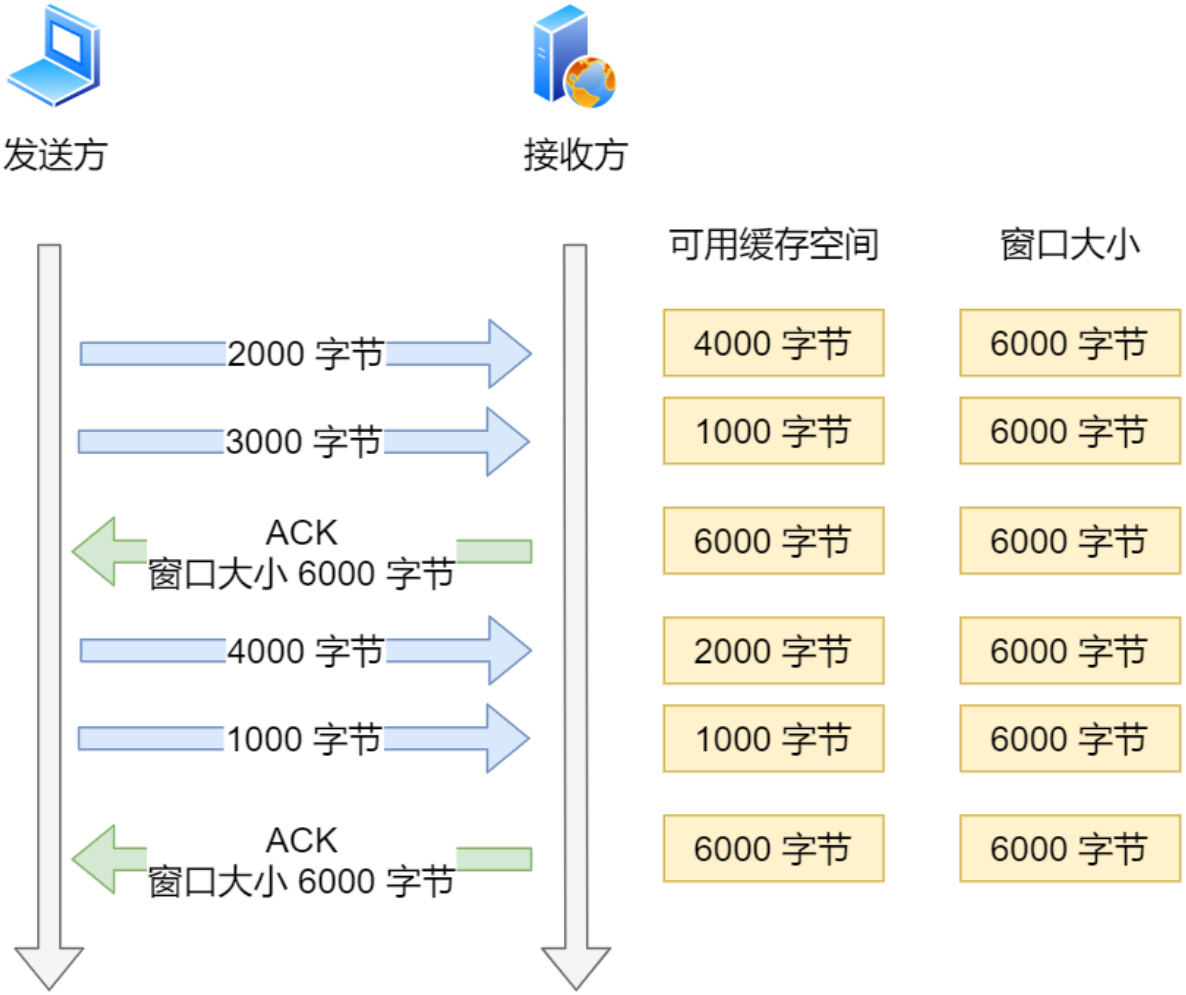
TCP 为了防止发送方无脑的发送数据，导致接收方缓冲区被填满，所以就有了滑动窗口的机制，它可利用接收方的接收窗口来控制发送方要发送的数据量，也就是流量控制。

接收窗口是由接收方指定的值，存储在 TCP 头部中，它可以告诉发送方自己的 TCP 缓冲空间区大小，这个缓冲区是给应用程序读取数据的空间：

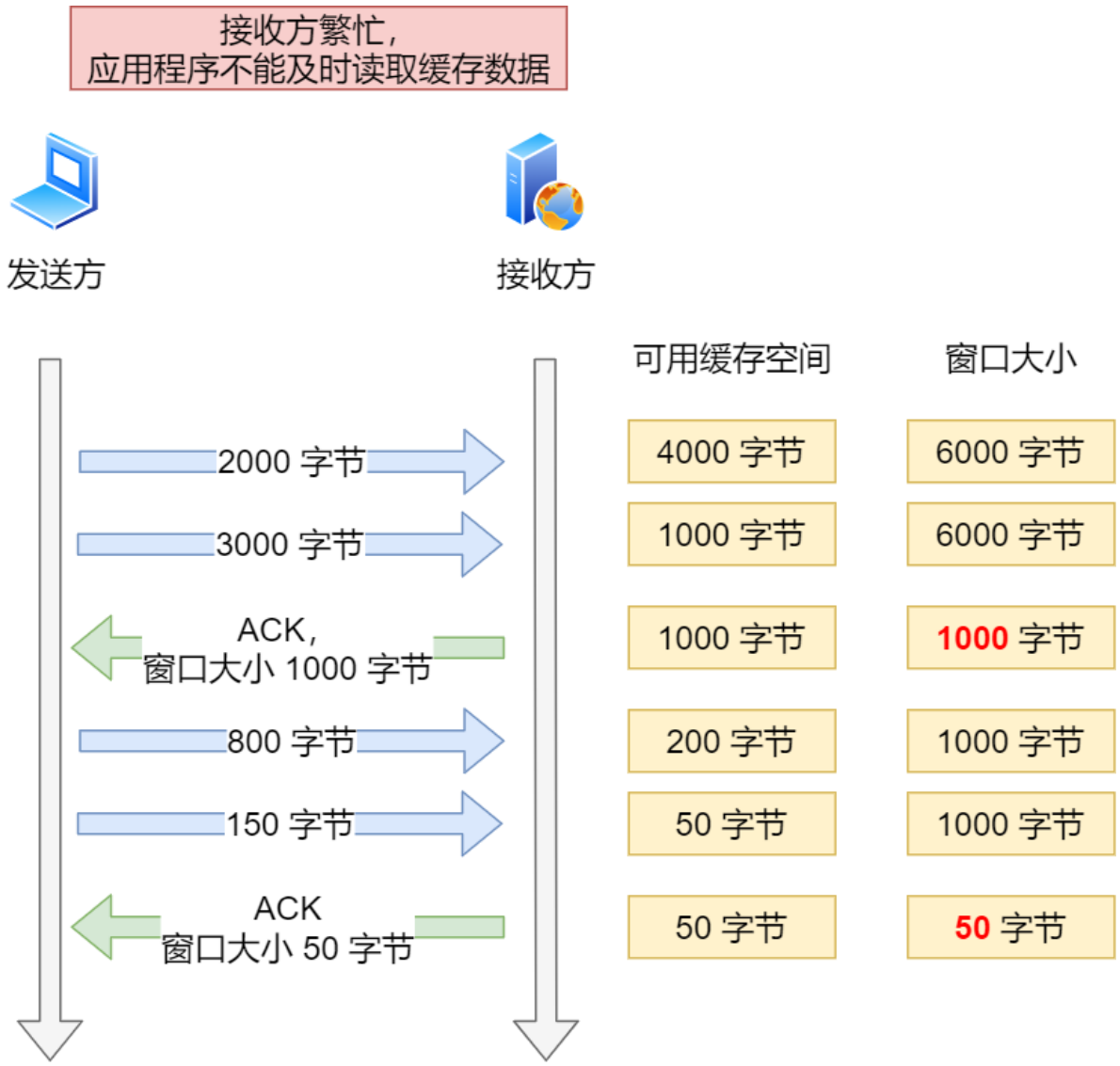
- 如果应用程序读取了缓冲区的数据，那么缓冲空间区就会把被读取的数据移除
- 如果应用程序没有读取数据，则数据会一直滞留在缓冲区。

接收窗口的大小，是在 TCP 三次握手中协商好的，后续数据传输时，接收方发送确认应答 ACK 报文时，会携带当前的接收窗口的大小，以此来告知发送方。

假设接收方接收到数据后，应用层能很快的从缓冲区里读取数据，那么窗口大小会一直保持不变，过程如下：



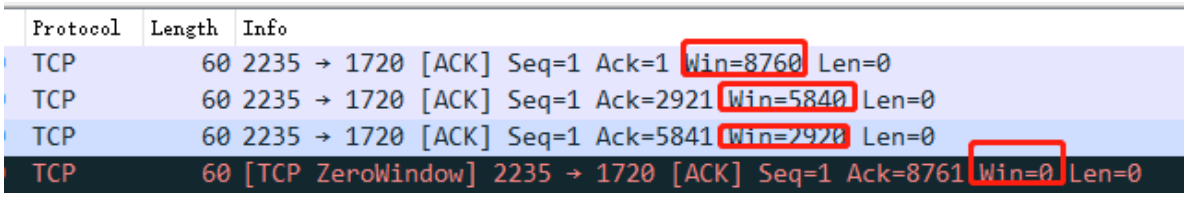
但是现实中服务器会出现繁忙的情况，当应用程序读取速度慢，那么缓存空间会慢慢被占满，于是为了保证发送方发送的数据不会超过缓冲区大小，服务器则会调整窗口大小的值，接着通过 ACK 报文通知给对方，告知现在的接收窗口大小，从而控制发送方发送的数据大小。



零窗口通知与窗口探测

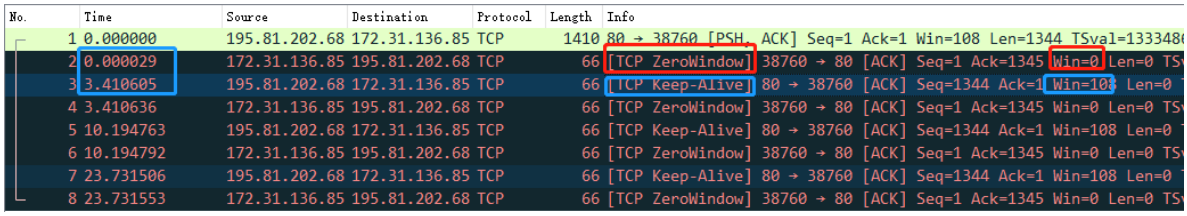
假设接收方处理数据的速度跟不上接收数据的速度，缓存就会被占满，从而导致接收窗口为 0，当发送方接收到零窗口通知时，就会停止发送数据。

如下图，可以看到接收方的窗口大小在不断的收缩至 0：



接着，发送方会**定时发送窗口大小探测报文**，以便及时知道接收方窗口大小的变化。

以下图 Wireshark 分析图作为例子说明：



- 发送方发送了数据包 1 给接收方，接收方收到后，由于缓冲区被占满，回了个零窗口通知；
- 发送方收到零窗口通知后，就不再发送数据了，直到过了 3.4 秒后，发送了一个 TCP Keep-Alive 报文，也就是窗口大小探测报文；
- 当接收方收到窗口探测报文后，就立马回一个窗口通知，但是窗口大小还是 0；
- 发送方发现窗口还是 0，于是继续等待了 6.8 （翻倍） 秒后，又发送了窗口探测报文，接收方依然还是回了窗口为 0 的通知；
- 发送方发现窗口还是 0，于是继续等待了 13.5 （翻倍） 秒后，又发送了窗口探测报文，接收方依然还是回了窗口为 0 的通知；

可以发现，这些窗口探测报文以 3.4s、6.5s、13.5s 的间隔出现，说明超时时间会**翻倍**递增。

这连接暂停了 25s，想象一下你在打王者的时候，25s 的延迟你还能上王者吗？

发送窗口的分析

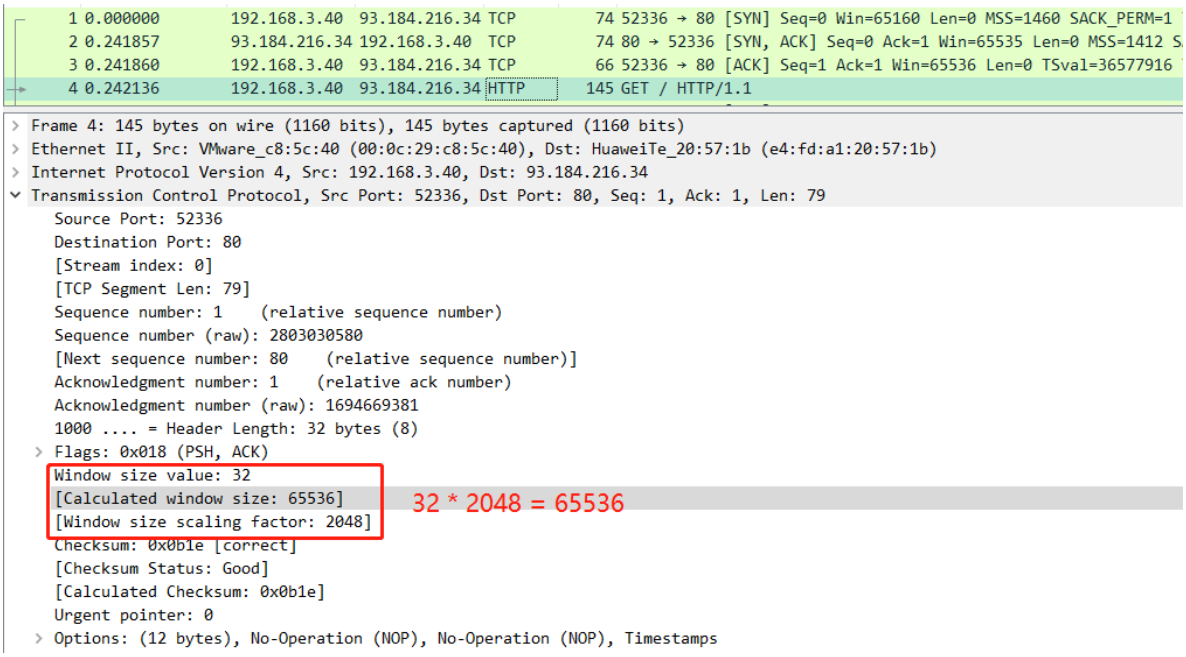
> 在 Wireshark 看到的 Windows size 也就是 " win = "，这个值表示发送窗口吗？

这不是发送窗口，而是在向对方声明自己的接收窗口。

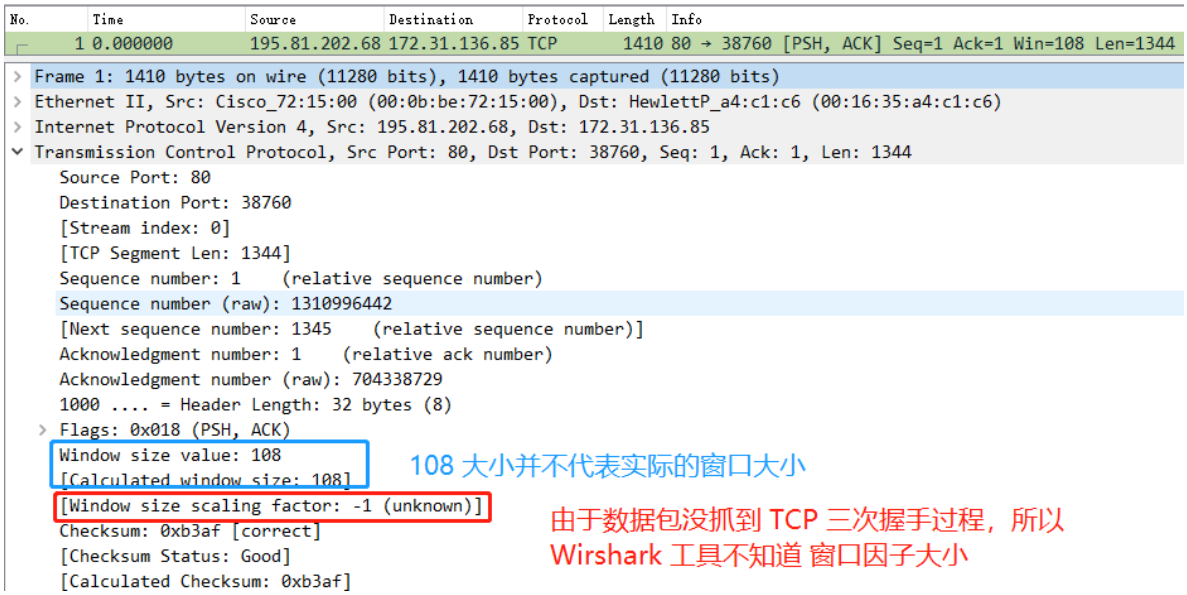
你可能会好奇，抓包文件里有「Window size scaling factor」，它其实是算出实际窗口大小的乘法因子，「Window size value」实际上并不是真实的窗口大小，真实窗口大小的计算公式如下：

「Window size value」 * 「Window size scaling factor」 = 「Caculated window size 」

对应的下图案例，也就是 32 * 2048 = 65536。



实际上是 Caculated window size 的值是 Wireshark 工具帮我们算好的，Window size scaling factor 和 Windos size value 的值是在 TCP 头部中，其中 Window size scaling factor 是在三次握手过程中确定的，如果你抓包的数据没有 TCP 三次握手，那可能就无法算出真实的窗口大小的值，如下图：



> 如何在包里看出发送窗口的大小？

很遗憾，没有简单的办法，发送窗口虽然是由接收窗口决定，但是它又可以被网络因素影响，也就是拥塞窗口，实际上发送窗口是值是 min(拥塞窗口，接收窗口)。

> 发送窗口和 MSS 有什么关系？

发送窗口决定了一口气能发多少字节，而 MSS 决定了这些字节要分多少包才能发完。

举个例子，如果发送窗口为 16000 字节的情况下，如果 MSS 是 1000 字节，那就需要发送 1600/1000 = 16 个包。

> 发送方在一个窗口发出 n 个包，是不是需要 n 个 ACK 确认报文？

不一定，因为 TCP 有累计确认机制，所以当收到多个数据包时，只需要应答最后一个数据包的 ACK 报文就可以了。

TCP 延迟确认与 Nagle 算法

当我们 TCP 报文的承载的数据非常小的时候，例如几个字节，那么整个网络的效率是很低的，因为每个 TCP 报文中都会有 20 个字节的 TCP 头部，也会有 20 个字节的 IP 头部，而数据只有几个字节，所以在整个报文中有效数据占有的比重就会非常低。

这就好像快递员开着大货车送一个小包裹一样浪费。

那么就出现了常见的两种策略，来减少小报文的传输，分别是：

- Nagle 算法
- 延迟确认

>

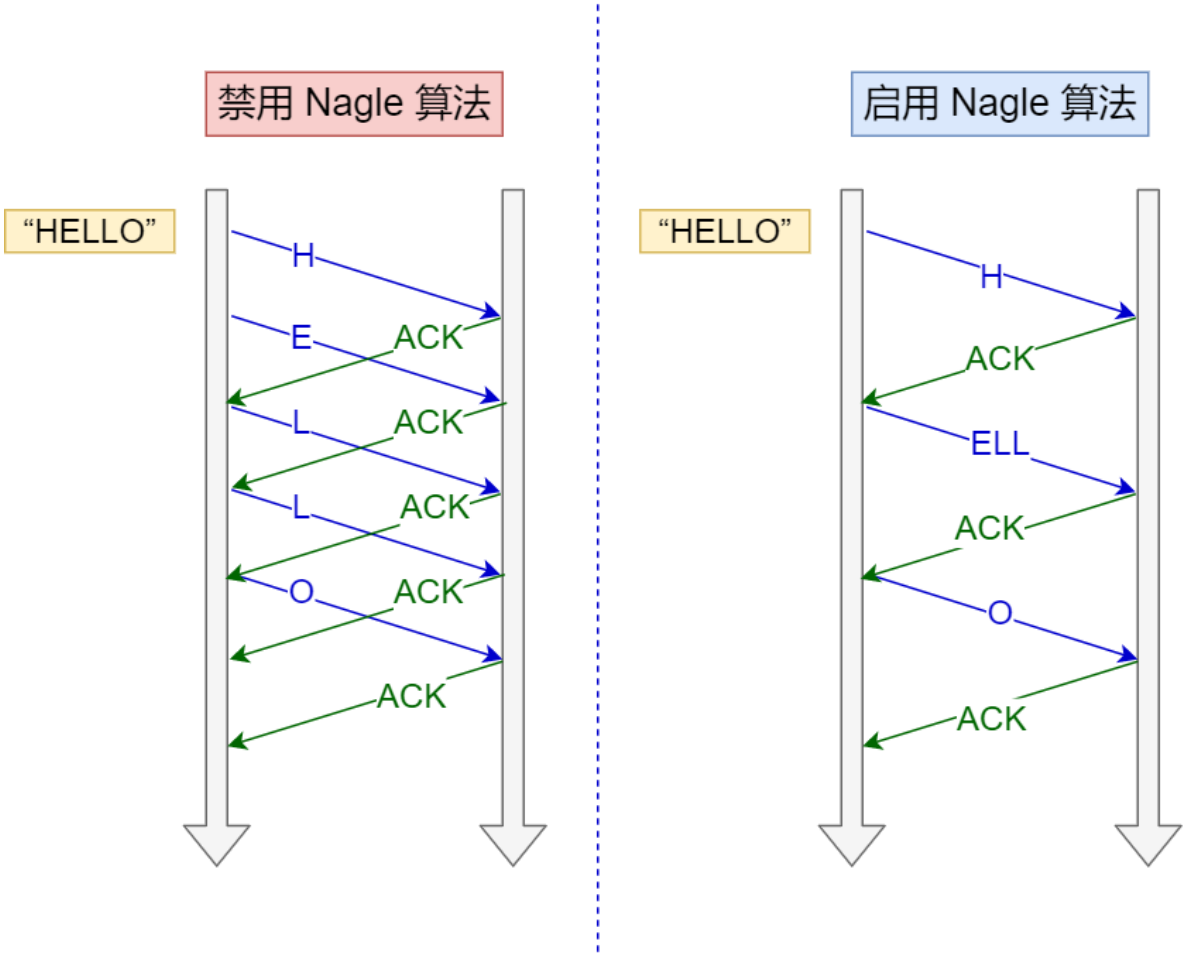
> Nagle 算法是如何避免大量 TCP 小数据报文的传输?

Nagle 算法做了一些策略来避免过多的小数据报文发送，这可提高传输效率。

Nagle 算法的策略：

- 没有已发送未确认报文时，立刻发送数据。
- 存在未确认报文时，直到「没有已发送未确认报文」或「数据长度达到 MSS 大小」时，再发送数据。

只要没满足上面条件中的一条，发送方一直在囤积数据，直到满足上面的发送条件。



上图右侧启用了 Nagle 算法，它的发送数据的过程：

- 一开始由于没有已发送未确认的报文，所以就立刻发了 H 字符；
- 接着，在还没收到对 H 字符的确认报文时，发送方就一直在囤积数据，直到收到了确认报文后，此时没有已发送未确认的报文，于是就把囤积后的 ELL 字符一起发给了接收方；
- 待收到对 ELL 字符的确认报文后，于是把最后一个 O 字符发送了出去

可以看出，**Nagle 算法一定会有一个小报文，也就是在最开始的时候。**

另外，Nagle 算法默认是打开的，如果对于一些需要小数据包交互的场景的程序，比如，telnet 或 ssh 这样的交互性比较强的程序，则需要关闭 Nagle 算法。

可以在 Socket 设置 `TCP_NODELAY` 选项来关闭这个算法（关闭 Nagle 算法没有全局参数，需要根据每个应用自己的特点来关闭）。

```
# 关闭 Nagle 算法
setsockopt(sock_fd, IPPROTO_TCP, TCP_NODELAY, (char *)&value, sizeof(int));
```

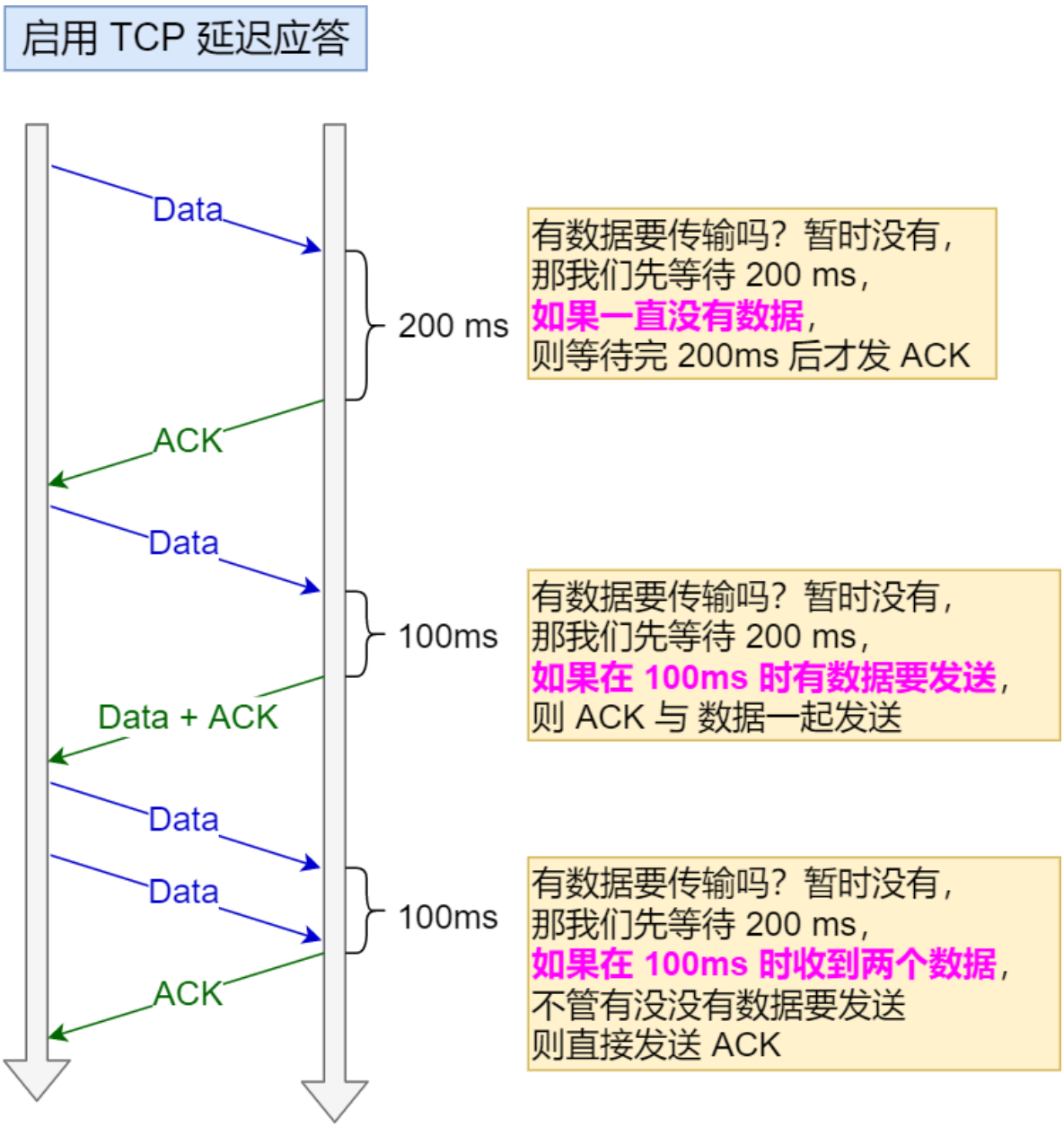
> 那延迟确认又是什么？

事实上当没有携带数据的 ACK，它的网络效率也是很低的，因为它也有 40 个字节的 IP 头 和 TCP 头，但却没有携带数据报文。

为了解决 ACK 传输效率低问题，所以就衍生出了 **TCP 延迟确认**。

TCP 延迟确认的策略：

- 当有响应数据要发送时，ACK 会随着响应数据一起立刻发送给对方
- 当没有响应数据要发送时，ACK 将会延迟一段时间，以等待是否有响应数据可以一起发送
- 如果在延迟等待发送 ACK 期间，对方的第二个数据报文又到达了，这时就会立刻发送 ACK



延迟等待的时间是在 Linux 内核中定义的，如下图：

```
#define TCP_DELACK_MAX ((unsigned)(HZ/5))  # 最大延迟确认时间
#define TCP_DELACK_MIN ((unsigned)(HZ/25)) # 最大延迟确认时间
```

关键就需要 `HZ` 这个数值大小，HZ 是跟系统的时钟频率有关，每个操作系统都不一样，在我的 Linux 系统中 HZ 大小是 `1000`，如下图：

```
$ cat /boot/config-2.6.32-431.el6.x86_64 | grep '^CONFIG_HZ='
CONFIG_HZ=1000
```

知道了 HZ 的大小，那么就可以算出：

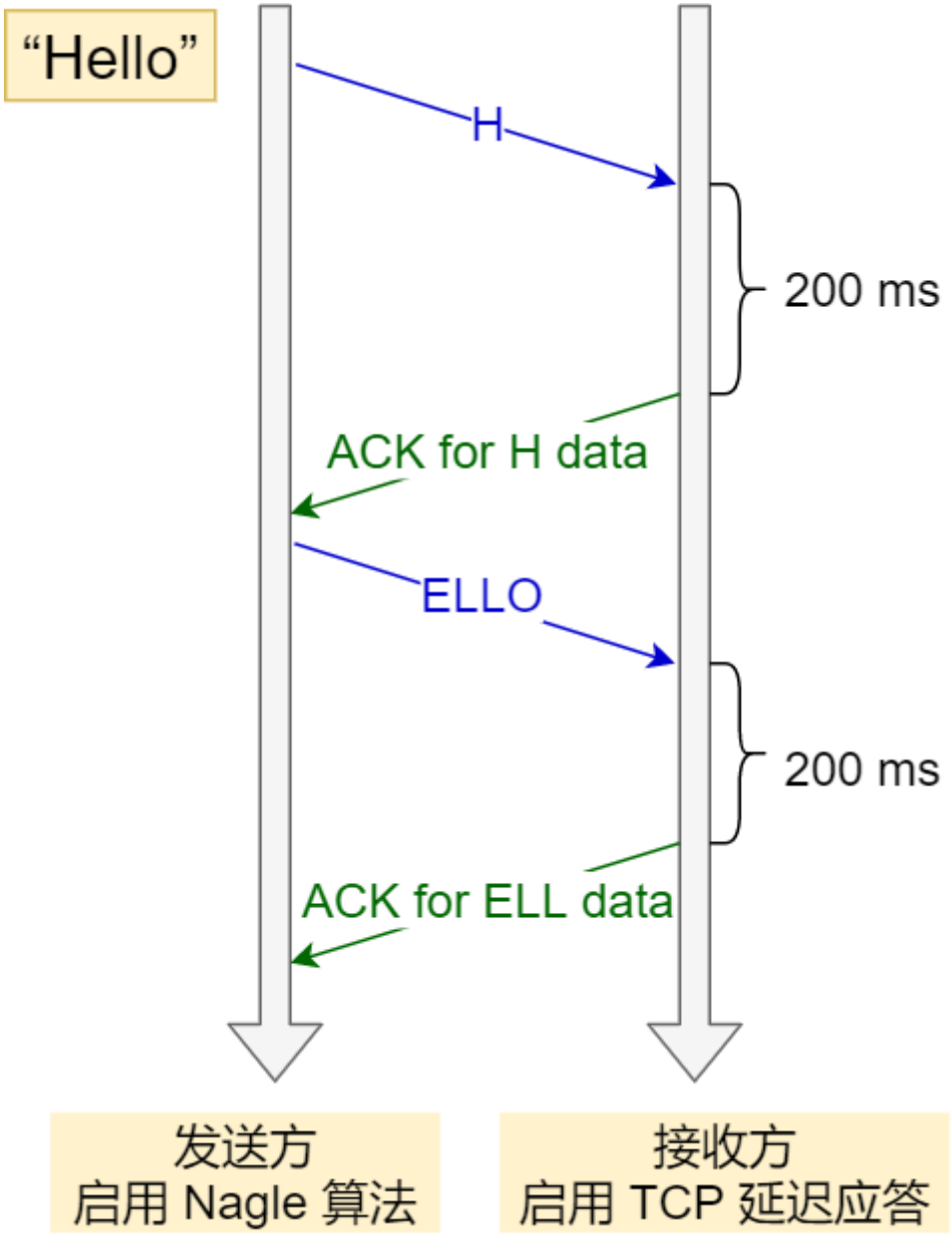
- 最大延迟确认时间是 `200` ms （1000/5）
- 最短延迟确认时间是 `40` ms （1000/25）

TCP 延迟确认可以在 Socket 设置 `TCP_QUICKACK` 选项来关闭这个算法。

```
# 关闭 TCP 延迟确认
setsockopt(sock_fd, IPPROTO_TCP, TCP_QUICKACK, (char *)&value, sizeof(int));
```

> 延迟确认 和 Nagle 算法混合使用时，会产生新的问题

当 TCP 延迟确认 和 Nagle 算法混合使用时，会导致时耗增长，如下图：



发送方使用了 Nagle 算法，接收方使用了 TCP 延迟确认会发生如下的过程：

- 发送方先发出一个小报文，接收方收到后，由于延迟确认机制，自己又没有要发送的数据，只能干等着发送方的下一个报文到达；
- 而发送方由于 Nagle 算法机制，在未收到第一个报文的确认前，是不会发送后续的数据；
- 所以接收方只能等待最大时间 200 ms 后，才回 ACK 报文，发送方收到第一个报文的确认报文后，也才可以发送后续的数据。

很明显，这两个同时使用会造成额外的时延，这就会使得网络"很慢"的感觉。

要解决这个问题，只有两个办法：

- 要不发送方关闭 Nagle 算法
- 要不接收方关闭 TCP 延迟确认

巨人的肩膀

[1] Wireshark网络分析的艺术.林沛满.人民邮电出版社.

[2] Wireshark网络分析就这么简单.林沛满.人民邮电出版社.

[3] Wireshark数据包分析实战.Chris Sanders .人民邮电出版社.

唠叨唠叨

文章中 Wireshark 分析的截图，可能有些会看的不清楚，为了方便大家用 Wireshark 分析，**我已把文中所有抓包的源文件，已分享到公众号了，大家在后台回复「抓包」，就可以获取了。**

名称	修改日期	类型	大
http	2020/5/7 20:43	Wireshark captu...	
ping	2020/5/7 18:38	Wireshark captu...	
tcp_4times_close	2020/5/17 13:01	Wireshark captu...	
tcp_dupack	2010/6/3 15:10	Wireshark captu...	
tcp_sys_timeout	2020/5/16 22:34	Wireshark captu...	
tcp_sys_timeout_2times	2020/5/16 22:34	Wireshark captu...	
tcp_sysack_timeout_2times_syn_timeo...	2020/5/16 22:35	Wireshark captu...	
tcp_sysack_timeout_5times_syn_timeo...	2020/5/16 22:35	Wireshark captu...	
tcp_thir_ack_timeout	2020/5/16 22:34	Wireshark captu...	
tcp_zerowindowdead	2010/6/12 17:15	Wireshark captu...	
tcp_zerowindowrecovery	2010/6/12 16:48	Wireshark captu...	

小林是专为大家图解的工具人，Goodbye，我们下次见！

读者问答

> 读者问：“两个问题，请教一下作者: tcp_retries1 参数，是什么场景下生效？ tcp_retries2是不是只受限于规定的次数，还是受限于次数和时间限制的最小值？”

tcp_retries1和tcp_retries2都是在TCP三次握手之后的场景。

- 当重传次数超过tcp_retries1就会指示 IP 层进行 MTU 探测、刷新路由等过程，并不会断开TCP连接，当重传次数超过 tcp_retries2 才会断开TCP流。
- tcp_retries1 和 tcp_retries2 两个重传次数都是受一个 timeout 值限制的，timeout 的值是根据它俩的值计算出来的，当重传时间超过 timeout，就不会继续重传了，即使次数还没到达。

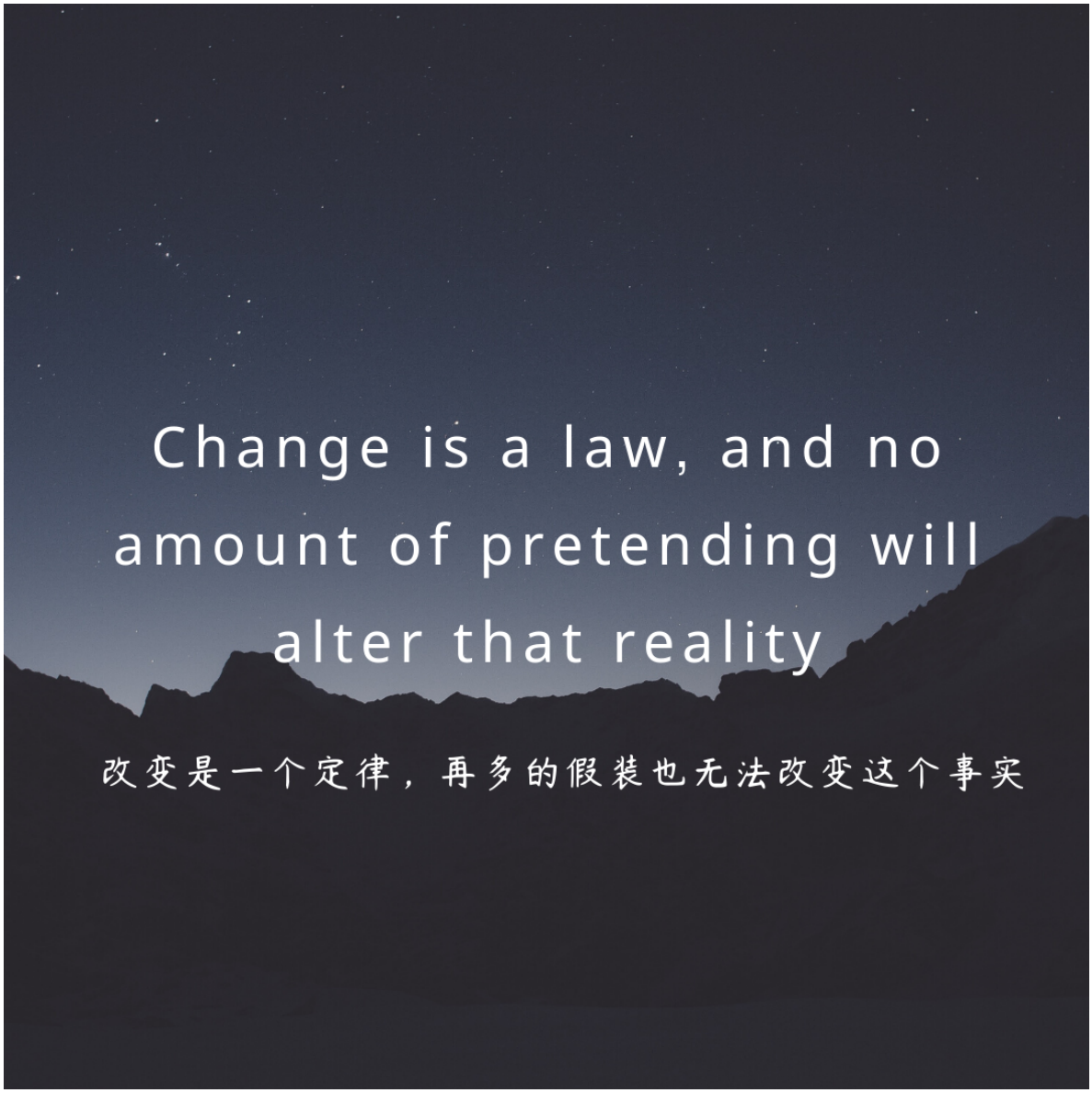
> 读者问：“tcp_orphan_retries也是控制tcp连接的关闭。这个跟tcp_retries1 tcp_retries2有什么区别吗？”

主动方发送 FIN 报文后，连接就处于 FIN_WAIT1 状态下，该状态通常应在数十毫秒内转为 FIN_WAIT2。如果迟迟收不到对方返回的 ACK 时，此时，内核会定时重发 FIN 报文，其中重发次数由 tcp_orphan_retries 参数控制。

> 读者问：“请问，为什么连续两个报文的seq会是一样的呢，比如三次握手之后的那个报文？还是说，序号相同的是同一个报文，只是拆开显示了？”

1. 三次握手中的前两次，是 seq+1；
2. 三次握手中的最后一个 ack，实际上是可以携带数据的，由于我文章的例子是没有发送数据的，你可以看到第三次握手的 len=0 ，在数据传输阶段「下一个 seq=seq+len 」，所以第三次握手的 seq 和下一个数据报的 seq 是一样的，因为 len 为 0；

TCP 半连接队列和全连接队列满了会发生什么？又该如何应对？



前言

网上许多博客针对增大 TCP 半连接队列和全连接队列的方式如下：

- 增大 TCP 半连接队列的方式是增大 /proc/sys/net/ipv4/tcp_max_syn_backlog；
- 增大 TCP 全连接队列的方式是增大 listen() 函数中的 backlog；

这里先跟大家说下，**上面的方式都是不准确的。**

> “你怎么知道不准确？”

很简单呀，因为我做了实验和看了 TCP 协议栈的内核源码，发现要增大这两个队列长度，不是简简单单增大某一个参数就可以的。

接下来，就会以**实战 + 源码分析，带大家解密 TCP 半连接队列和全连接队列。**

> “源码分析，那不是劝退吗？我们搞 Java 的看不懂呀”

放心，本文的源码分析不会涉及很深的知识，因为都被我删减了，你只需要会条件判断语句 if、左移右移操作符、加减法等基本语法，就可以看懂。

另外，不仅有源码分析，还会介绍 Linux 排查半连接队列和全连接队列的命令。

> “哦？似乎很有看头，那我姑且看一下吧！”

行，没有被劝退的小伙伴，值得鼓励，下面这图是本文的提纲：

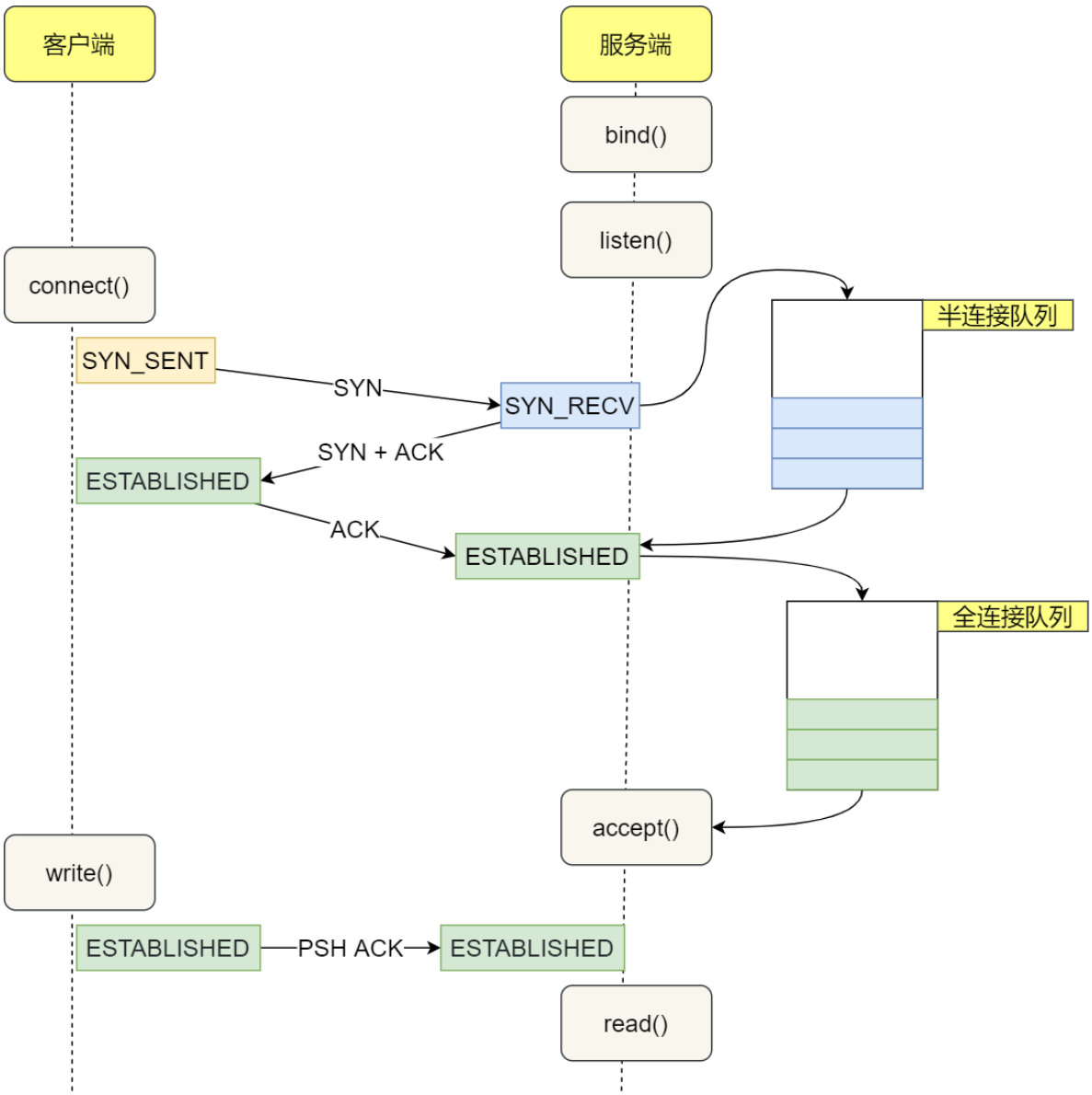
正文

什么是 TCP 半连接队列和全连接队列?

在 TCP 三次握手的时候，Linux 内核会维护两个队列，分别是：

- 半连接队列，也称 SYN 队列；
- 全连接队列，也称 accepet 队列；

服务端收到客户端发起的 SYN 请求后，**内核会把该连接存储到半连接队列**，并向客户端响应 SYN+ACK，接着客户端会返回 ACK，服务端收到第三次握手的 ACK 后，**内核会把连接从半连接队列移除，然后创建新的完全的连接，并将其添加到 accept 队列，等待进程调用 accept 函数时把连接取出来。**



不管是半连接队列还是全连接队列，都有最大长度限制，超过限制时，内核会直接丢弃，或返回 RST 包。

实战 - TCP 全连接队列溢出

> 如何知道应用程序的 TCP 全连接队列大小?

在服务端可以使用 `ss` 命令，来查看 TCP 全连接队列的情况：

但需要注意的是 `ss` 命令获取的 `Recv-Q/Send-Q` 在「LISTEN 状态」和「非 LISTEN 状态」所表达的含义是不同的。从下面的内核代码可以看出区别：

```c
// Linux 2.6.32 内核文件：net/ipv4/tcp_diag.c
static void tcp_diag_get_info(struct sock *sk,
                              struct inet_diag_msg *r,
                              void *_info)
{
    ...

    // 如果 TCP 连接状态是 LISTEN 时
    if (sk->sk_state == TCP_LISTEN) {
        // 当前全连接队列的大小
        r->idiag_rqueue = sk->sk_ack_backlog;
        // 当前全连接最大队列长度
        r->idiag_wqueue = sk->sk_max_ack_backlog;
    }
    // 如果 TCP 连接状态不是 LISTEN 时
    else {
        // 已收到但未被应用进程读取的字节数
        r->idiag_rqueue = tp->rcv_nxt - tp->copied_seq;
        // 已发送但未收到确认的字节数
        r->idiag_wqueue = tp->write_seq - tp->snd_una;
    }
    ...
}
```

在「LISTEN 状态」时，`Recv-Q/Send-Q` 表示的含义如下：

```
# -l 显示正在监听（ listening ）的 socket
# -n 不解析服务名称
# -t 只显示 tcp socket
$ ss -lnt
State       Recv-Q Send-Q      Local Address:Port    Peer Address:Port
LISTEN      0      128         *:8088                *:*
```

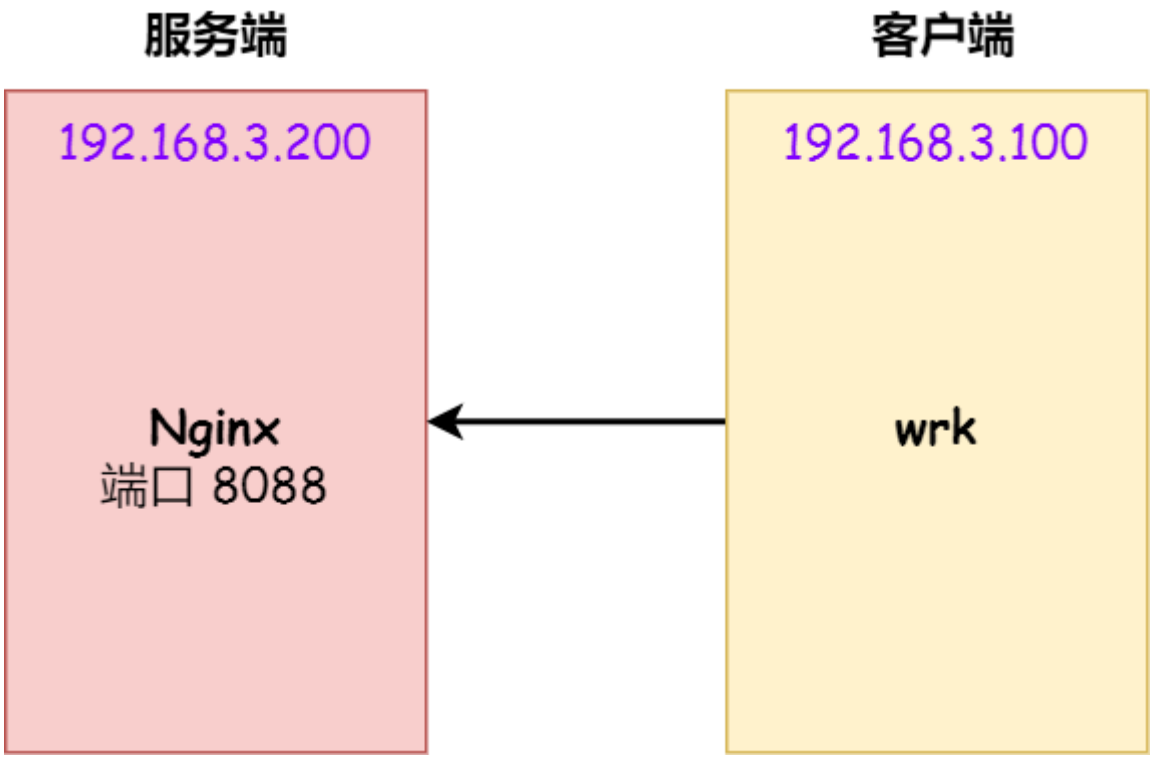
- Recv-Q：当前全连接队列的大小，也就是当前已完成三次握手并等待服务端 `accept()` 的 TCP 连接；
- Send-Q：当前全连接最大队列长度，上面的输出结果说明监听 8088 端口的 TCP 服务，最大全连接长度为 128；

在「非 LISTEN 状态」时，`Recv-Q/Send-Q` 表示的含义如下：

```
# -n 不解析服务名称
# -t 只显示 tcp socket
$ ss -nt
State       Recv-Q Send-Q      Local Address:Port    Peer Address:Port
ESTAB       0      850         *:8088               *:*
```

- Recv-Q：已收到但未被应用进程读取的字节数；
- Send-Q：已发送但未收到确认的字节数；

> 如何模拟 TCP 全连接队列溢出的场景？



实验环境：

- 客户端和服务端都是 CentOs 6.5 ， Linux 内核版本 2.6.32
- 服务端 IP 192.168.3.200，客户端 IP 192.168.3.100
- 服务端是 Nginx 服务，端口为 8088

这里先介绍下 `wrk` 工具，它是一款简单的 HTTP 压测工具，它能够在单机多核 CPU 的条件下，使用系统自带的高性能 I/O 机制，通过多线程和事件模式，对目标机器产生大量的负载。

本次模拟实验就使用 `wrk` 工具来压力测试服务端，发起大量的请求，一起看看服务端 TCP 全连接队列满了会发生什么？有什么观察指标？

客户端执行 `wrk` 命令对服务端发起压力测试，并发 3 万个连接：

```
# 客户端对服务端进行压测
# -t 6      表示 6 个线程
# -c 30000  表示 3 万个连接
# -d 60s    表示持续压测 60 秒
$ wrk -t 6 -c 30000 -d 60s http://192.168.3.200:8088
Running 1m test @ http://192.168.3.200:8088
  6 threads and 30000 connections
                                    # 压测中，阻塞...
```

在服务端可以使用 `ss` 命令，来查看当前 TCP 全连接队列的情况：

```
# 服务端查看 TCP 全连接队列的情况
$ ss -lnt | grep 8088
State       Recv-Q Send-Q      Local Address:Port     Peer Address:Port
LISTEN      125    128         *:8088                 *:*
$ ss -lnt | grep 8088
State       Recv-Q Send-Q      Local Address:Port     Peer Address:Port
LISTEN      129    128         *:8088                 *:*
```

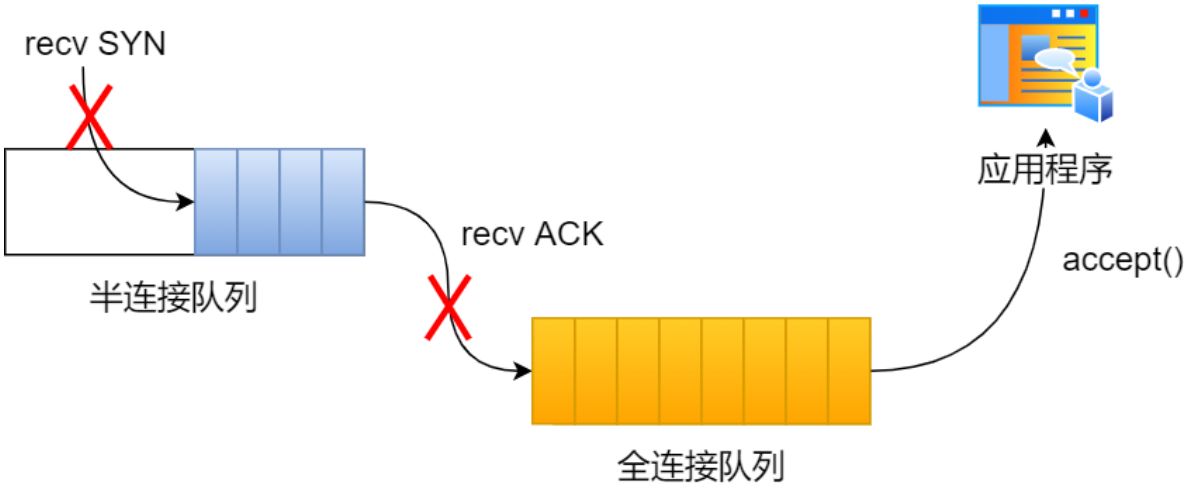
其间共执行了两次 ss 命令，从上面的输出结果，可以发现当前 TCP 全连接队列上升到了 129 大小，超过了最大 TCP 全连接队列。

当超过了 TCP 最大全连接队列，服务端则会丢掉后续进来的 TCP 连接，丢掉的 TCP 连接的个数会被统计起来，我们可以使用 netstat -s 命令来查看：

```
# 查看 TCP 全连接队列溢出情况
$ date;netstat -s | grep overflowed
Sun May 17 07:35:40 CST 2020
    41150 times the listen queue of a socket overflowed

$ date;netstat -s | grep overflowed
Sun May 17 07:35:41 CST 2020
    42512 times the listen queue of a socket overflowed
```

上面看到的 41150 times ，表示全连接队列溢出的次数，注意这个是累计值。可以隔几秒钟执行下，如果这个数字一直在增加的话肯定全连接队列偶尔满了。

从上面的模拟结果，可以得知，**当服务端并发处理大量请求时，如果 TCP 全连接队列过小，就容易溢出。发生 TCP 全连接队溢出的时候，后续的请求就会被丢弃，这样就会出现服务端请求数量上不去的现象。**



> Linux 有个参数可以指定当 TCP 全连接队列满了会使用什么策略来回应客户端。

实际上，丢弃连接只是 Linux 的默认行为，我们还可以选择向客户端发送 RST 复位报文，告诉客户端连接已经建立失败。

```
$ cat /proc/sys/net/ipv4/tcp_abort_on_overflow
0      # 默认值为 0
```

tcp_abort_on_overflow 共有两个值分别是 0 和 1，其分别表示：

- 0：如果全连接队列满了，那么 server 扔掉 client 发过来的 ack；
- 1：如果全连接队列满了，server 发送一个 `reset` 包给 client，表示废掉这个握手过程和这个连接；

如果要想知道客户端连接不上服务端，是不是服务端 TCP 全连接队列满的原因，那么可以把 tcp_abort_on_overflow 设置为 1，这时如果在客户端异常中可以看到很多 `connection reset by peer` 的错误，那么就可以证明是由于服务端 TCP 全连接队列溢出的问题。

通常情况下，应当把 tcp_abort_on_overflow 设置为 0，因为这样更有利于应对突发流量。

举个例子，当 TCP 全连接队列满导致服务器丢掉了 ACK，与此同时，客户端的连接状态却是 ESTABLISHED，进程就在建立好的连接上发送请求。只要服务器没有为请求回复 ACK，请求就会被多次**重发**。如果服务器上的进程只是**短暂的繁忙造成 accept 队列满，那么当 TCP 全连接队列有空位时，再次接收到的请求报文由于含有 ACK，仍然会触发服务器端成功建立连接。**

所以，tcp_abort_on_overflow 设为 0 可以提高连接建立的成功率，只有你非常肯定 TCP 全连接队列会长期溢出时，才能设置为 1 以尽快通知客户端。

> 如何增大 TCP 全连接队列呢?

是的，当发现 TCP 全连接队列发生溢出的时候，我们就需要增大该队列的大小，以便可以应对客户端大量的请求。

TCP 全连接队列的最大值取决于 somaxconn 和 backlog 之间的最小值，也就是 min(somaxconn, backlog)。从下面的 Linux 内核代码可以得知：

```
// Linux 2.6.35 net/socket.c
// listen 函数调用的内核源码
SYSCALL_DEFINE2(listen, int, fd, int, backlog)
{
   ...

   // /proc/sys/net/core/somaxconn
   somaxconn = sock_net(sock->sk)->core.sysctl_somaxconn;

   // TCP 全连接队列最大值 = min(somaxconn, backlog)
   if ((unsigned)backlog > somaxconn)
        backlog = somaxconn;

    ...
}
```

- `somaxconn` 是 Linux 内核的参数，默认值是 128，可以通过 `/proc/sys/net/core/somaxconn` 来设置其值；
- `backlog` 是 `listen(int sockfd, int backlog)` 函数中的 backlog 大小，Nginx 默认值是 511，可以通过修改配置文件设置其长度；

前面模拟测试中，我的测试环境：

- somaxconn 是默认值 128；
- Nginx 的 backlog 是默认值 511

所以测试环境的 TCP 全连接队列最大值为 min(128, 511)，也就是 `128` ，可以执行 `ss` 命令查看：

```
# -l 显示正在监听（ listening ）的 socket
# -n 不解析服务名称
# -t 只显示 tcp socket
$ ss -lnt | grep 8088
State       Recv-Q Send-Q     Local Address:Port    Peer Address:Port
LISTEN      0      128        *:8088                *:*
```

现在我们重新压测，把 TCP 全连接队列**搞大**，把 somaxconn 设置成 5000：

```
# 服务端增大 somaxconn 的值
$ echo 5000 > /proc/sys/net/core/somaxconn
```

接着把 Nginx 的 backlog 也同样设置成 5000：

```
# /usr/local/nginx/conf/nginx.conf
server {
        listen  8088 default backlog=5000;
        server_name  localhost;
        ....
}
```

最后要重启 Nginx 服务，因为只有重新调用 listen() 函数 TCP 全连接队列才会重新初始化。

重启完后 Nginx 服务后，服务端执行 ss 命令，查看 TCP 全连接队列大小：

```
# -l 显示正在监听（ listening ）的 socket
# -n 不解析服务名称
# -t 只显示 tcp socket
$ ss -lnt | grep 8088
State       Recv-Q Send-Q     Local Address:Port      Peer Address:Port
LISTEN      0      5000       *:8088                  *:*
```

从执行结果，可以发现 TCP 全连接最大值为 5000。

> 增大 TCP 全连接队列后，继续压测

客户端同样以 3 万个连接并发发送请求给服务端：

```
# 客户端对服务端进行压测
# -t 6     表示 6 个线程
# -c 30000 表示 3 万个连接
# -d 60s   表示持续压测 60 秒
$ wrk -t 6 -c 30000 -d 60s http://192.168.3.200:8088
Running 1m test @ http://192.168.3.200:8088
  6 threads and 30000 connections
                                    # 压测中，阻塞...
```

服务端执行 `ss` 命令，查看 TCP 全连接队列使用情况：

```
# 服务端查看 TCP 全连接队列使用情况
$ ss -lnt | grep 8088
LISTEN     695    5000       *:8088       *:*
$ ss -lnt | grep 8088
LISTEN     764    5000       *:8088       *:*
$ ss -lnt | grep 8088
LISTEN     1504   5000       *:8088       *:*
$ ss -lnt | grep 8088
LISTEN     101    5000       *:8088       *:*
```

从上面的执行结果，可以发现全连接队列使用增长的很快，但是一直都没有超过最大值，所以就不会溢出，那么 `netstat -s` 就不会有 TCP 全连接队列溢出个数的显示：

```
# 服务端查看 TCP 全连接队列是否有溢出
$ netstat -s | grep overflowed
                # 没返回任何值，说明没有连接溢出
```

说明 TCP 全连接队列最大值从 128 增大到 5000 后，服务端抗住了 3 万连接并发请求，也没有发生全连接队列溢出的现象了。

如果持续不断地有连接因为 TCP 全连接队列溢出被丢弃，就应该调大 backlog 以及 somaxconn 参数。

实战 - TCP 半连接队列溢出

> 如何查看 TCP 半连接队列长度?

很遗憾，TCP 半连接队列长度的长度，没有像全连接队列那样可以用 ss 命令查看。

但是我们可以抓住 TCP 半连接的特点，就是服务端处于 `SYN_RECV` 状态的 TCP 连接，就是 TCP 半连接队列。

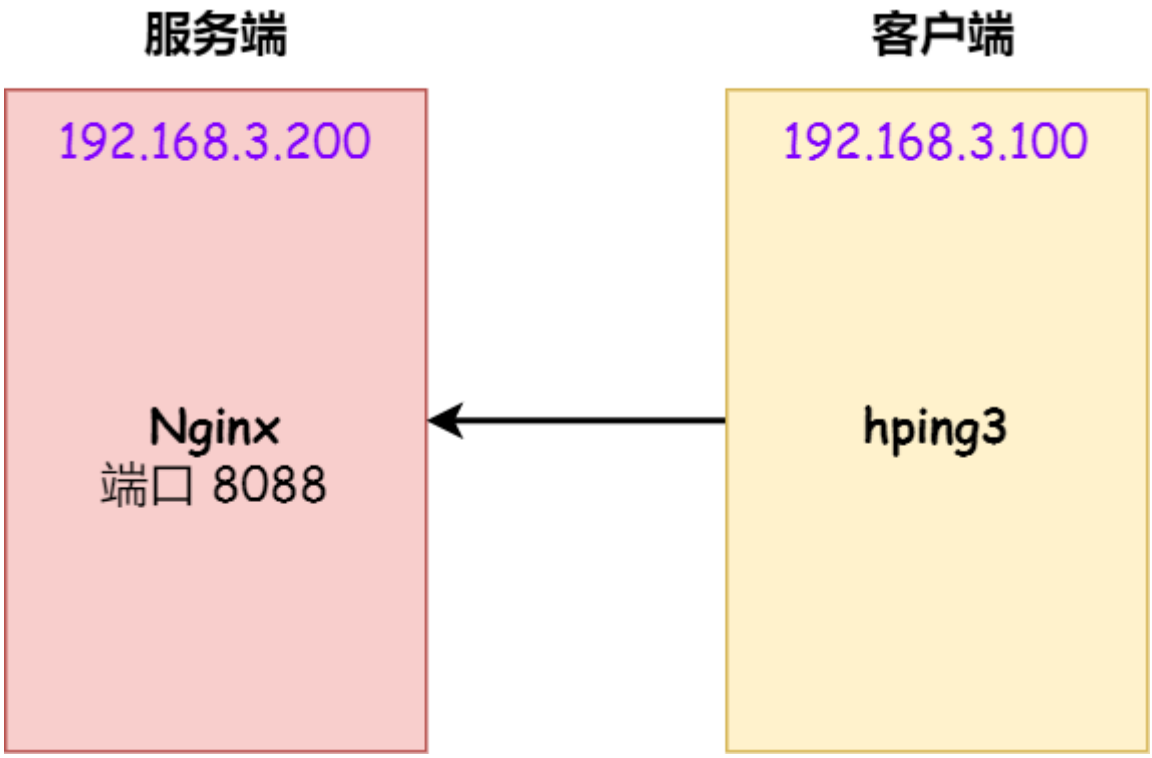
于是，我们可以使用如下命令计算当前 TCP 半连接队列长度：

```
# 查看当前 TCP 半连接队列长度
$ netstat -natp | grep SYN_RECV | wc -l
256  # 表示处于半连接状态的 TCP 连接有 256 个
```

> 如何模拟 TCP 半连接队列溢出场景?

模拟 TCP 半连接溢出场景不难，实际上就是对服务端一直发送 TCP SYN 包，但是不回第三次握手 ACK，这样就会使得服务端有大量的处于 `SYN_RECV` 状态的 TCP 连接。

这其实也就是所谓的 SYN 洪泛、SYN 攻击、DDos 攻击。



实验环境：

- 客户端和服务端都是 CentOs 6.5 ， Linux 内核版本 2.6.32
- 服务端 IP 192.168.3.200，客户端 IP 192.168.3.100
- 服务端是 Nginx 服务，端口为 8088

注意：本次模拟实验是没有开启 tcp_syncookies，关于 tcp_syncookies 的作用，后续会说明。

本次实验使用 `hping3` 工具模拟 SYN 攻击：

```
# 客户端发起 SYN 攻击
# -S        指定 TCP 包的标志位 SYN
# -p 8088   指定探测的目的端口
# ---flood 以泛洪的方式攻击
$ hping3 -S -p 8088 --flood 192.168.3.200
HPING 192.168.3.200 (eth0 192.168.3.200): S set, 40 headers + 0 data bytes
hping in flood mode, no replies will be shown
                          # 不停的 SYN 攻击中，阻塞着...
```

当服务端受到 SYN 攻击后，连接服务端 ssh 就会断开了，无法再连上。只能在服务端主机上执行查看当前 TCP 半连接队列大小：

```
# 服务端查看当前 TCP 半连接队列大小
$ netstat -natp | grep SYN_RECV | wc -l
256       # 如果一直是 256，说明 TCP 半连接队列最大长度为 256
```

同时，还可以通过 netstat -s 观察半连接队列溢出的情况：

```
$ netstat -s | grep "SYNs to LISTEN"
    3479887 SYNs to LISTEN sockets dropped
$ netstat -s | grep "SYNs to LISTEN"
    3526030 SYNs to LISTEN sockets dropped
```

上面输出的数值是**累计值**，表示共有多少个 TCP 连接因为半连接队列溢出而被丢弃。**隔几秒执行几次，如果有上升的趋势，说明当前存在半连接队列溢出的现象**。

> 大部分人都说 tcp_max_syn_backlog 是指定半连接队列的大小，是真的吗？

很遗憾，半连接队列的大小并不单单只跟 `tcp_max_syn_backlog` 有关系。

上面模拟 SYN 攻击场景时，服务端的 tcp_max_syn_backlog 的默认值如下：

```
$ cat /proc/sys/net/ipv4/tcp_max_syn_backlog
512   # CentOs 6.5 默认值是 512
```

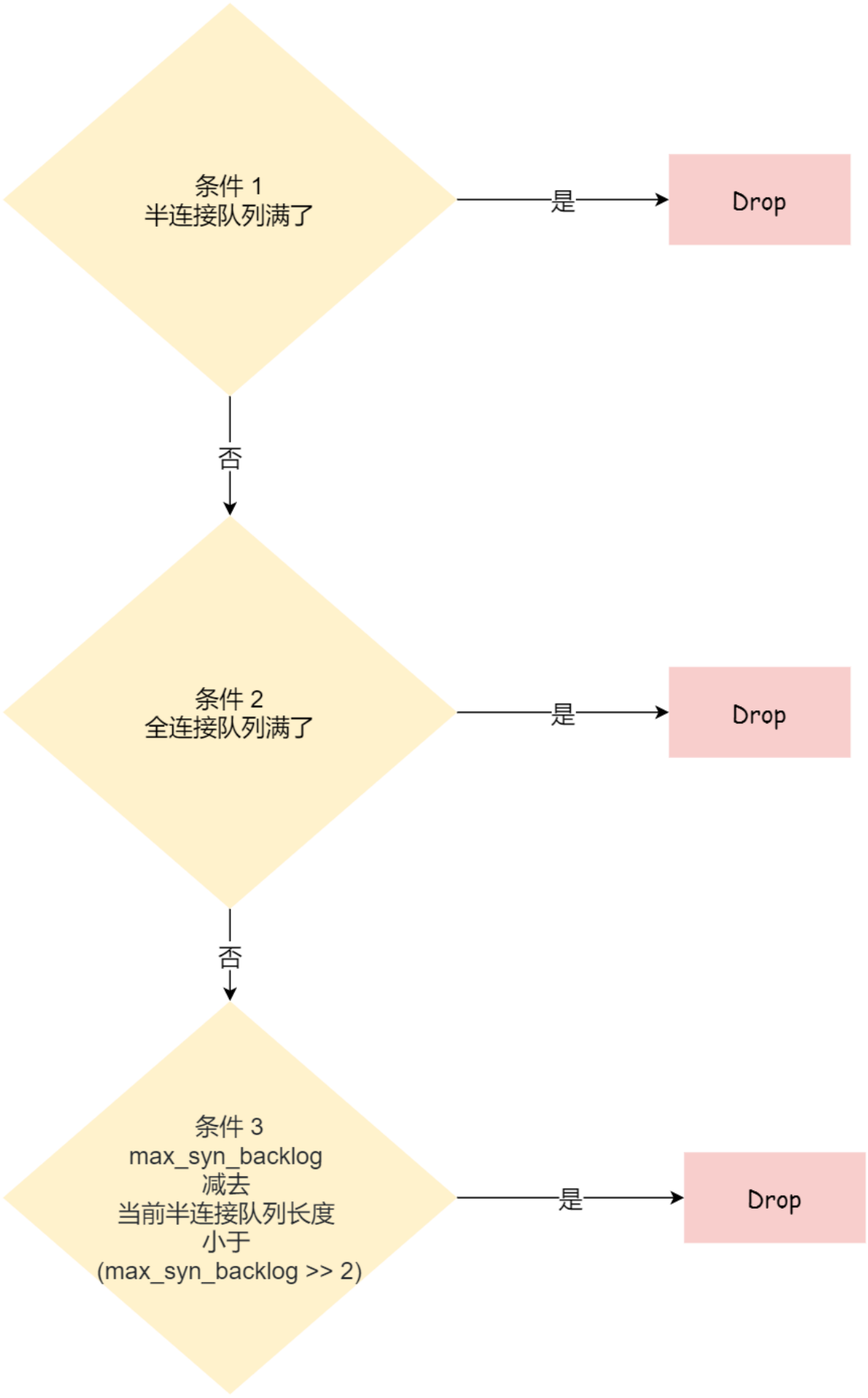
但是在测试的时候发现，服务端最多只有 256 个半连接队列，而不是 512，所以**半连接队列的最大长度不一定由 tcp_max_syn_backlog 值决定的**。

> 接下来，走进 Linux 内核的源码，来分析 TCP 半连接队列的最大值是如何决定的。

TCP 第一次握手（收到 SYN 包）的 Linux 内核代码如下，其中缩减了大量的代码，只需要重点关注 TCP 半连接队列溢出的处理逻辑：

```c
// Linux 2.6.32 内核：net/ipv4/tcp_ipv4.c
int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb)
{
    ...

    /*
       条件 1：
       如果半连接队列满了
    */
    if (inet_csk_reqsk_queue_is_full(sk) && !isn) {
        if (sysctl_tcp_syncookies) {
            want_cookie = 1;
        } else
        // 如果半连接队列满了，并且没有开启 tcp_syncookies，则会丢弃连接。
        goto drop;
    }

    /*
       条件 2：
       若全连接队列满，且没有重传 SYN+ACK 包的连接请求多于 1 个，则会丢弃
    */
    if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) > 1)
        goto drop;

    ...

    if (want_cookie) {
        ...
    } else if (!isn) {
        ...
        /*
          条件 3：
          如果没有开启 tcp_syncookies，
          并且 max_syn_backlog 减去 当前半连接队列长度
          小于 (max_syn_backlog >> 2)，则会丢弃
        */
        if (!sysctl_tcp_syncookies &&
            (sysctl_max_syn_backlog - inet_csk_reqsk_queue_len(sk) <
             (sysctl_max_syn_backlog >> 2)) && ...) {
            goto drop_and_release;
        }
    }

    ...
}
```

从源码中，我可以得出共有三个条件因队列长度的关系而被丢弃的：



1. **如果半连接队列满了，并且没有开启 tcp_syncookies，则会丢弃；**
2. **若全连接队列满了，且没有重传 SYN+ACK 包的连接请求多于 1 个，则会丢弃；**

3. **如果没有开启 tcp_syncookies，并且 max_syn_backlog 减去 当前半连接队列长度小于 (max_syn_backlog >> 2)，则会丢弃；**

关于 tcp_syncookies 的设置，后面在详细说明，可以先给大家说一下，开启 tcp_syncookies 是缓解 SYN 攻击其中一个手段。

接下来，我们继续跟一下检测半连接队列是否满的函数 inet_csk_reqsk_queue_is_full 和 检测全连接队列是否满的函数 sk_acceptq_is_full：

```
// Linux 2.6.32 内核：include\net\inet_connection_sock.h
// 检测半连接队列是否满的函数
static inline int inet_csk_reqsk_queue_is_full(const struct sock *sk)
{
    //检测半连接队列是否满的函数
    return reqsk_queue_is_full(&inet_csk(sk)->icsk_accept_queue);
}

// Linux 2.6.32 内核：include\net\request_sock.h
// 检测半连接队列是否满的函数
static inline int reqsk_queue_is_full(const struct request_sock_queue *queue)
{
    /*
       qlen          当前半连接队列的长度
       max_qlen_log 半连接队列最大值
    */
    // 注意这里是用 >> (右移) 来判断的，不是大于号
    return queue->listen_opt->qlen >> queue->listen_opt->max_qlen_log;
}

-----------------------------------------------------------------------------

// Linux 2.6.32 内核：include\net\sock.h
// 检测全连接队列是否满的函数
static inline int sk_acceptq_is_full(struct sock *sk)
{
    /*
       sk_ack_backlog      当前全连接队列的长度
       sk_max_ack_backlog 全连接队列最大值
    */
    // sk_max_ack_backlog = min(somaxconn, backlog)
    return sk->sk_ack_backlog > sk->sk_max_ack_backlog;
}
```

从上面源码，可以得知：

- **全**连接队列的最大值是 `sk_max_ack_backlog` 变量，sk_max_ack_backlog 实际上是在 listen() 源码里指定的，也就是 **min(somaxconn, backlog)**；
- **半**连接队列的最大值是 `max_qlen_log` 变量，max_qlen_log 是在哪指定的呢？现在暂时还不知道，我们继续跟进；

我们继续跟进代码，看一下是哪里初始化了半连接队列的最大值 max_qlen_log：

```c
// Linux 2.6.32 内核：net\core\request_sock.c

/*
  本次服务端环境相关变量的值：
  somaxconn 为 128 默认值
  backlog 为 511 Nginx 服务的默认值
  tcp_max_syn_backlog 为 512 默认值
*/

/*
  nr_table_entries 是全连接队列最大值，
  也就是 min(somaxconn, backlog) = 128
*/
int reqsk_queue_alloc(struct request_sock_queue *queue,
                      unsigned int nr_table_entries)
{
    ...

    // nr_table_entries = min(128, 512) = 128
    nr_table_entries = min_t(u32, nr_table_entries,
                                  sysctl_max_syn_backlog);

    // nr_table_entries = max(128, 8) = 128
    nr_table_entries = max_t(u32, nr_table_entries, 8);

    /*
      roundup_pow_of_two 的算法：
      找出当前数的二级制中最大位为 1 位的位置，然后用 1 左移位数即可。
      比如数据 5：
      二进制形式为101，最高位为 1 的位置是 3，然后左移3位，等于 1000，即数字 8。
    */
    // nr_table_entries = roundup_pow_of_two(128 +1) = 256
    nr_table_entries = roundup_pow_of_two(nr_table_entries + 1);

    ...

    /*
       nr_table_entries 在上面算出是 256，代入此 for 循环，
       就可以算出 max_qlen_log 的值 为 8
    */
    for (lopt->max_qlen_log = 3;
         (1 << lopt->max_qlen_log) < nr_table_entries;
         lopt->max_qlen_log++);

    ...

    return 0;
}
```

从上面的代码中，我们可以算出 max_qlen_log 是 8，于是代入到 检测半连接队列是否满的函数 reqsk_queue_is_full：

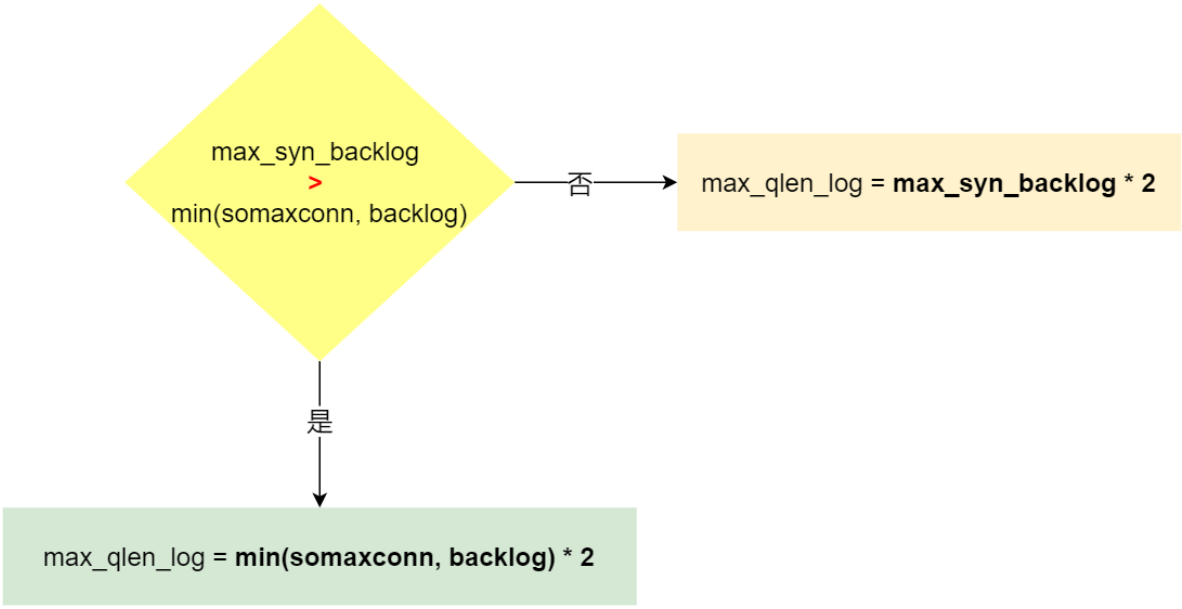
```c
// Linux 2.6.32 内核：include\net\request_sock.h
// 检测半连接队列是否满的函数
static inline int reqsk_queue_is_full(const struct request_sock_queue *queue)
{
    /*
       qlen          当前半连接队列的长度
       max_qlen_log 半连接队列最大值
    */
    // qlen >> 8
    return queue->listen_opt->qlen >> queue->listen_opt->max_qlen_log;
}
```

也就是 `qlen >> 8` 什么时候为 1 就代表半连接队列满了。这计算这不难，很明显是当 qlen 为 256 时， `256 >> 8 = 1` 。

至此，总算知道为什么上面模拟测试 SYN 攻击的时候，服务端处于 `SYN_RECV` 连接最大只有 256 个。

可见，**半连接队列最大值不是单单由 max_syn_backlog 决定，还跟 somaxconn 和 backlog 有关系。**

在 Linux 2.6.32 内核版本，它们之间的关系，总体可以概况为：



- 当 max_syn_backlog > min(somaxconn, backlog) 时， 半连接队列最大值 max_qlen_log = min(somaxconn, backlog) * 2;
- 当 max_syn_backlog < min(somaxconn, backlog) 时， 半连接队列最大值 max_qlen_log = max_syn_backlog * 2;

> 半连接队列最大值 max_qlen_log 就表示服务端处于 SYN_REVC 状态的最大个数吗?

依然很遗憾，并不是。

max_qlen_log 是**理论**半连接队列最大值，并不一定代表服务端处于 SYN_REVC 状态的最大个数。

在前面我们在分析 TCP 第一次握手（收到 SYN 包）时会被丢弃的三种条件：

1. 如果半连接队列满了，并且没有开启 tcp_syncookies，则会丢弃；
2. 若全连接队列满了，且没有重传 SYN+ACK 包的连接请求多于 1 个，则会丢弃；
3. **如果没有开启 tcp_syncookies，并且 max_syn_backlog 减去 当前半连接队列长度小于 (max_syn_backlog >> 2)，则会丢弃；**

假设条件 1 当前半连接队列的长度 「没有超过」理论的半连接队列最大值 max_qlen_log，那么如果条件 3 成立，则依然会丢弃 SYN 包，也就会使得服务端处于 SYN_REVC 状态的最大个数不会是理论值 max_qlen_log。

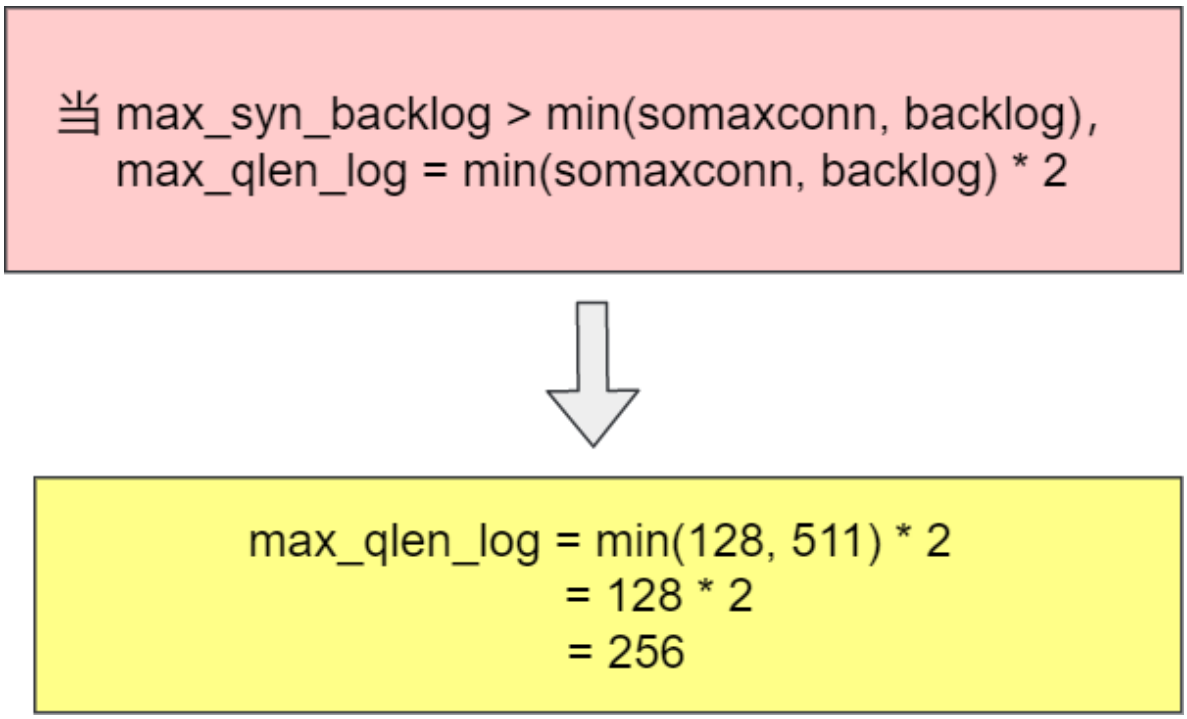
似乎很难理解，我们继续接着做实验，实验见真知。

服务端环境如下：

```
$ echo 256  > /proc/sys/net/ipv4/tcp_max_syn_backlog # 设置为 256
$ echo 128  > /proc/sys/net/core/somaxconn           # 依然保持默认值 128
# Nginx 的 backlog 依然保持默认值 511
```

配置完后，服务端要重启 Nginx，因为全连接队列最大值和半连接队列最大值是在 listen() 函数初始化。

根据前面的源码分析，我们可以计算出半连接队列 max_qlen_log 的最大值为 256：



客户端执行 hping3 发起 SYN 攻击：

```
# 客户端发起 SYN 攻击
$ hping3 -S -p 8088 --flood 192.168.3.200
```

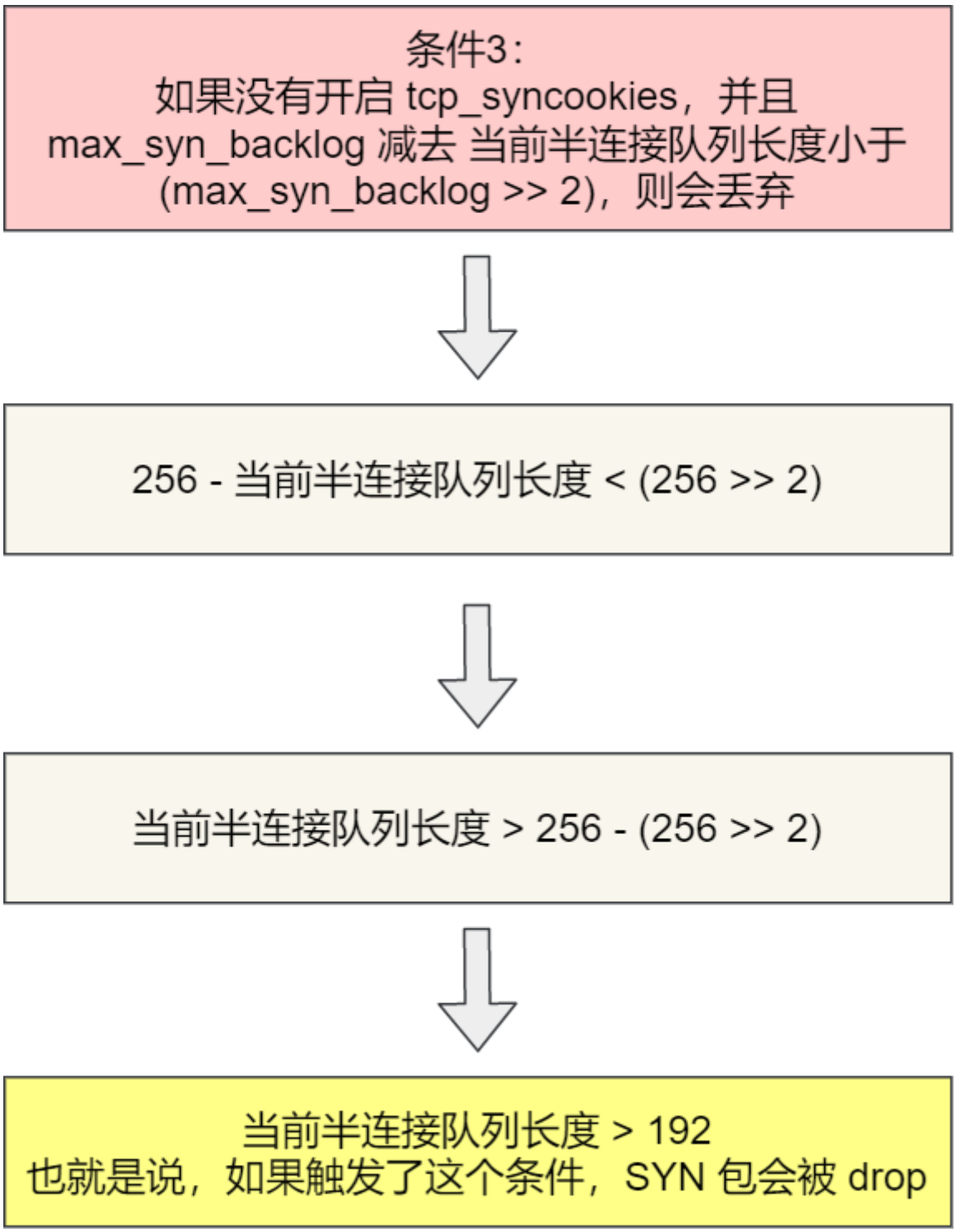
服务端执行如下命令，查看处于 SYN_RECV 状态的最大个数：

```
$ netstat -natp | grep SYN_RECV | wc -l
193  # 处于 SYN_RECV 状态的最大个数是 193
```

可以发现，服务端处于 SYN_RECV 状态的最大个数并不是 max_qlen_log 变量的值。

这就是前面所说的原因：**如果当前半连接队列的长度「没有超过」理论半连接队列最大值 max_qlen_log，那么如果条件 3 成立，则依然会丢弃 SYN 包，也就会使得服务端处于 SYN_REVC 状态的最大个数不会是理论值 max_qlen_log。**

我们来分析一波条件 3 :



从上面的分析，可以得知如果触发「当前半连接队列长度 > 192」条件，TCP 第一次握手的 SYN 包是会被丢弃的。

在前面我们测试的结果，服务端处于 SYN_RECV 状态的最大个数是 193，正好是触发了条件 3，所以处于 SYN_RECV 状态的个数还没到「理论半连接队列最大值 256」，就已经把 SYN 包丢弃了。

所以，服务端处于 SYN_RECV 状态的最大个数分为如下两种情况：

- 如果「当前半连接队列」**没超过**「理论半连接队列最大值」，但是**超过** max_syn_backlog - (max_syn_backlog >> 2)，那么处于 SYN_RECV 状态的最大个数就是 max_syn_backlog - (max_syn_backlog >> 2)；
- 如果「当前半连接队列」**超过**「理论半连接队列最大值」，那么处于 SYN_RECV 状态的最大个数就是「理论半连接队列最大值」；

> 每个 Linux 内核版本「理论」半连接最大值计算方式会不同。

在上面我们是针对 Linux 2.6.32 版本分析的「理论」半连接最大值的算法，可能每个版本有些不同。

比如在 Linux 5.0.0 的时候，「理论」半连接最大值就是全连接队列最大值，但依然还是有队列溢出的三个条件：

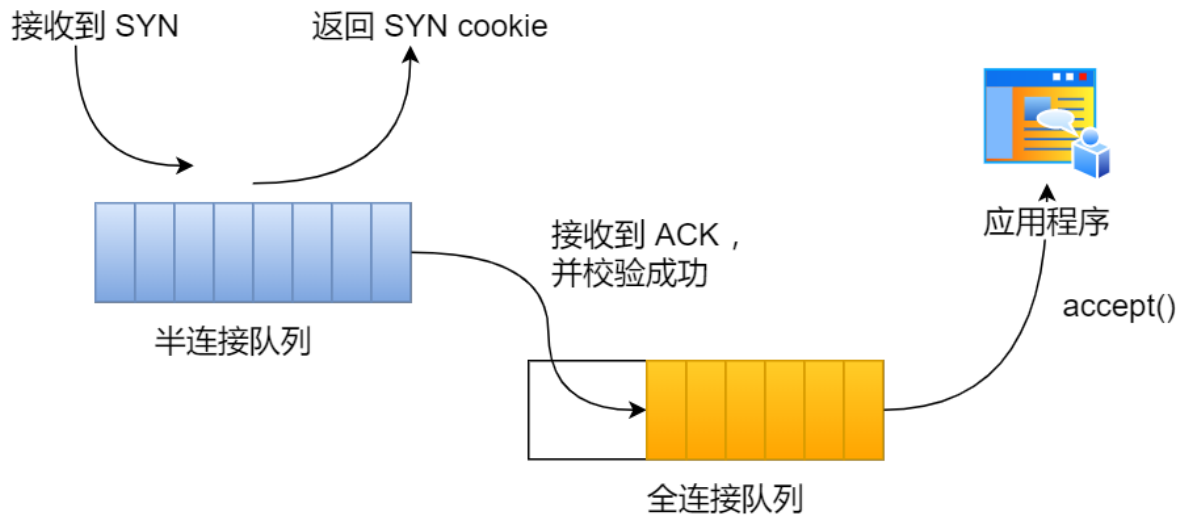
```c
// Linux 5.0.0 Version

// 检测半连接队列是否溢出
static inline int inet_csk_reqsk_queue_is_full(const struct sock *sk)
{
    /* inet_csk_reqsk_queue_len  当前半连接队列大小
       sk_max_ack_backlog        全连接队列最大值 min(somaxconn, backlog)
    */
    // 如果当前半连接队列大小 >= 全连接队列最大值，则表示半连接队列溢出
    return inet_csk_reqsk_queue_len(sk) >= sk->sk_max_ack_backlog;
}

// 检测半连接队列是否溢出
static inline bool sk_acceptq_is_full(const struct sock *sk)
{
    /* sk_ack_backlog     当前全连接队列大小
       sk_max_ack_backlog 全连接队列最大值 min(somaxconn, backlog)
    */
    // 如果当前全连接队列大小 > 全连接队列最大值，则表示全连接队列溢出
    return sk->sk_ack_backlog > sk->sk_max_ack_backlog;
}

// TCP 第一次握手，即收到 SYN 包的处理
int tcp_conn_request(struct request_sock_ops *rsk_ops,
             const struct tcp_request_sock_ops *af_ops,
             struct sock *sk, struct sk_buff *skb)
{
    ...

    if ((net->ipv4.sysctl_tcp_syncookies == 2 ||
         inet_csk_reqsk_queue_is_full(sk)) && !isn) {
        want_cookie = tcp_syn_flood_action(sk, skb, rsk_ops->slab_name);
        if (!want_cookie)
            /* 条件 1:
               如果没有开启 tcp_syncookies，并且半连接队列溢出，则会丢弃
            */
            goto drop;
    }

    /* 条件 2:
       如果全连接队列溢出，则会丢弃
    */
    if (sk_acceptq_is_full(sk)) {
        NET_INC_STATS(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
        goto drop;
    }

    ...

    if (!want_cookie && !isn) {
        if (!net->ipv4.sysctl_tcp_syncookies &&
            (net->ipv4.sysctl_max_syn_backlog -
            inet_csk_reqsk_queue_len(sk)
            <
             (net->ipv4.sysctl_max_syn_backlog >> 2)) && ...) {

            ...
```

```
            /* 条件 3:
               如果没有开启 tcp_syncookies,
               并且 max_syn_backlog - 当前半连接队列大小
               <
               (max_syn_backlog >> 2)，则会丢弃
            */
            goto drop_and_release;
        }

        ...
    }

    ...

    return 0;
}
```

> 如果 SYN 半连接队列已满，只能丢弃连接吗？

并不是这样，**开启 syncookies 功能就可以在不使用 SYN 半连接队列的情况下成功建立连接**，在前面我们源码分析也可以看到这点，当开启了 syncookies 功能就不会丢弃连接。

syncookies 是这么做的：服务器根据当前状态计算出一个值，放在己方发出的 SYN+ACK 报文中发出，当客户端返回 ACK 报文时，取出该值验证，如果合法，就认为连接建立成功，如下图所示。



syncookies 参数主要有以下三个值：

- 0 值，表示关闭该功能；
- 1 值，表示仅当 SYN 半连接队列放不下时，再启用它；
- 2 值，表示无条件开启功能；

那么在应对 SYN 攻击时，只需要设置为 1 即可：

```
# 仅当 SYN 半连接队列放不下时，再启用它
$ echo 1 > /proc/sys/net/ipv4/tcp_syncookies
```

> 如何防御 SYN 攻击?

这里给出几种防御 SYN 攻击的方法：

- 增大半连接队列；
- 开启 tcp_syncookies 功能
- 减少 SYN+ACK 重传次数

方式一：增大半连接队列

在前面源码和实验中，得知**要想增大半连接队列，我们得知不能只单纯增大 tcp_max_syn_backlog 的值，还需一同增大 somaxconn 和 backlog，也就是增大全连接队列**。否则，只单纯增大 tcp_max_syn_backlog 是无效的。

增大 tcp_max_syn_backlog 和 somaxconn 的方法是修改 Linux 内核参数：

```
# 增大 tcp_max_syn_backlog
$ echo 1024 > /proc/sys/net/ipv4/tcp_max_syn_backlog
# 增大 somaxconn
$ echo 1024 > /proc/sys/net/core/somaxconn
```

增大 backlog 的方式，每个 Web 服务都不同，比如 Nginx 增大 backlog 的方法如下：

```
# /usr/local/nginx/conf/nginx.conf
server {
        listen  8088 default backlog=1024;
        server_name  localhost;
        ....
}
```

最后，改变了如上这些参数后，要重启 Nginx 服务，因为半连接队列和全连接队列都是在 listen() 初始化的。

方式二：开启 tcp_syncookies 功能

开启 tcp_syncookies 功能的方式也很简单，修改 Linux 内核参数：

```
# 仅当 SYN 半连接队列放不下时，再启用它
$ echo 1 > /proc/sys/net/ipv4/tcp_syncookies
```

方式三：减少 SYN+ACK 重传次数

当服务端受到 SYN 攻击时，就会有大量处于 SYN_REVC 状态的 TCP 连接，处于这个状态的 TCP 会重传 SYN+ACK ，当重传超过次数达到上限后，就会断开连接。

那么针对 SYN 攻击的场景，我们可以减少 SYN+ACK 的重传次数，以加快处于 SYN_REVC 状态的 TCP 连接断开。

```
# SYN+ACK 重传次数上限设置为 1 次
$ echo 1 > /proc/sys/net/ipv4/tcp_synack_retries
```

巨人的肩膀

[1] 系统性能调优必知必会.陶辉.极客时间.

[2] https://www.cnblogs.com/zengkefu/p/5606696.html

[3] https://blog.cloudflare.com/syn-packet-handling-in-the-wild/

唠叨唠叨

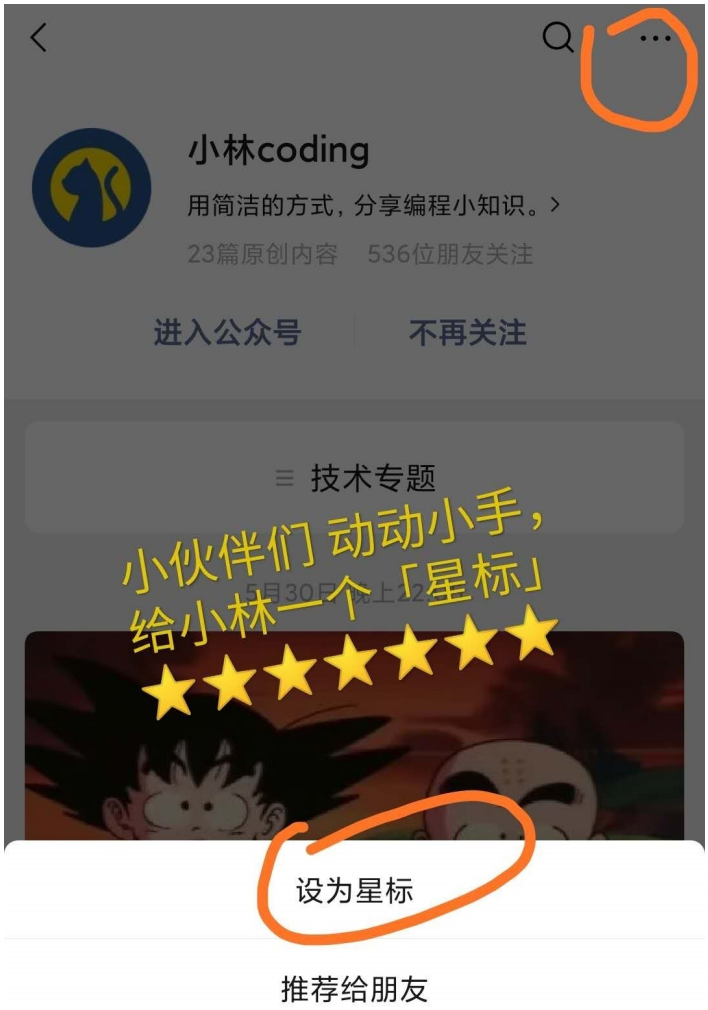
本文是以 Linux 2.6.32 版本的内核用实验 + 源码的方式，给大家说明了 TCP 半连接队列和全连接队列，我们可以看到 TCP 半连接队列「并不是」如网上说的那样 tcp_max_syn_backlog 表示半连接队列。

TCP 半连接队列的大小对于不同的 Linux 内核版本会有不同的计算方式，所以并不要求大家要死记住本文计算 TCP 半连接队列的大小。

重要的是要学会自我源码分析，这样不管碰到什么版本的 Linux 内核，都不再怕了。

网上搜索出来的信息，并不一定针对你的系统，通过自我分析一波，你会更了解你当前使用的 Linux 内核版本！

小林是专为大家图解的工具人，Goodbye，我们下次见！



读者问答

> 读者问：“咦 我比较好奇博主都是从哪里学到这些知识的呀？书籍？视频？还是多种参考资料”

你可以看我的参考文献呀，知识点我主要是在极客专栏学的，实战模拟实验和源码解析是自己瞎折腾出来的。

> 读者问：“syncookies 启用后就不需要半链接了？那请求的数据会存在哪里？”

syncookies = 1 时，半连接队列满后，后续的请求就不会存放到半连接队列了，而是在第二次握手的时候，服务端会计算一个 cookie 值，放入到 SYN +ACK 包中的序列号发给客户端，客户端收到后并回 ack ，服务端就会校验连接是否合法，合法就直接把连接放入到全连接队列。

面试官：换人！他连 TCP 这几个参数都不懂

前言

TCP 性能的提升不仅考察 TCP 的理论知识，还考察了对于操作系统提供的内核参数的理解与应用。

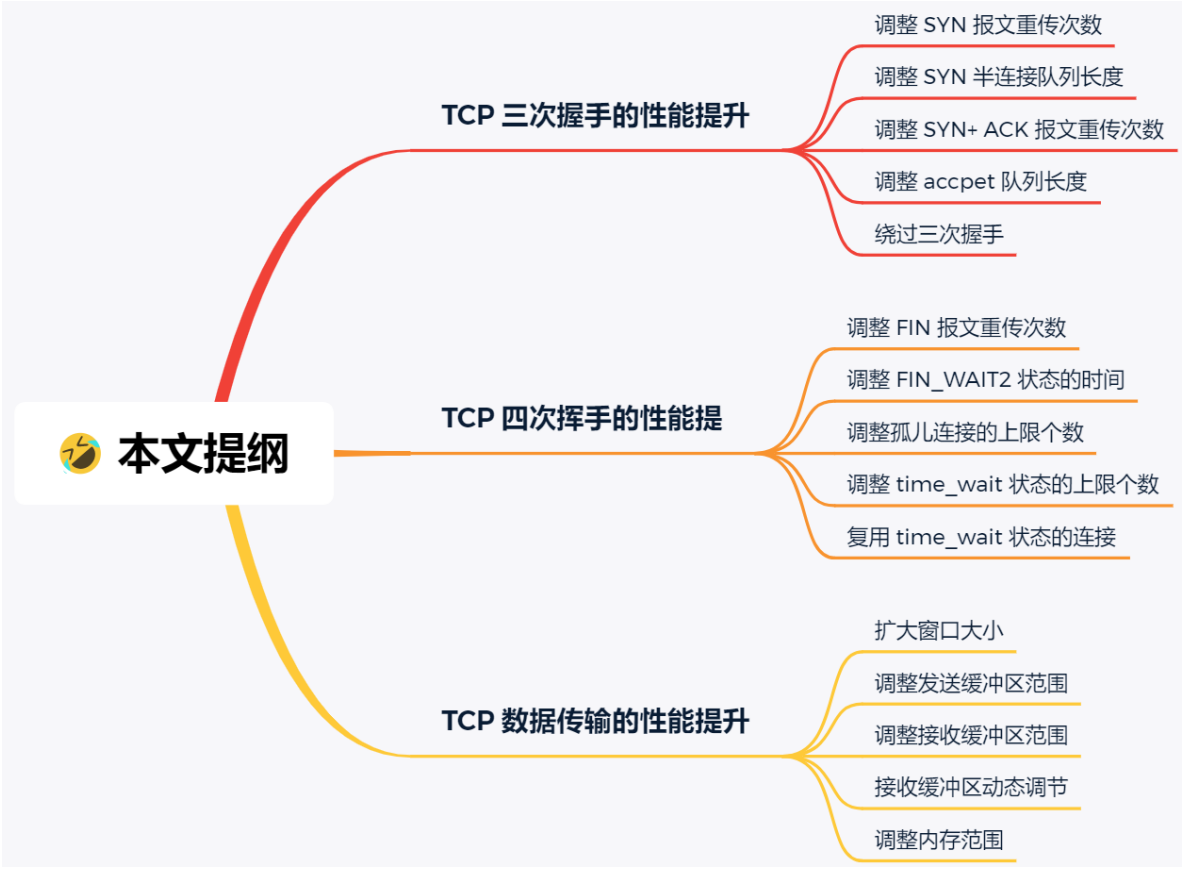
TCP 协议是由操作系统实现，所以操作系统提供了不少调节 TCP 的参数。

```
[root@lincoding ipv4]# ls -l /proc/sys/net/ipv4/tcp*
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_abc
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_abort_on_overflow
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_adv_win_scale
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_allowed_congestion_control
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_app_win
-r--r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_available_congestion_control
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_base_mss
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_challenge_ack_limit
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_congestion_control
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_dma_copybreak
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_dsack
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_ecn
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_fack
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_fin_timeout
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_frto
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_frto_response
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_keepalive_intvl
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_keepalive_probes
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_keepalive_time
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_limit_output_bytes
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_low_latency
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_max_orphans
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_max_ssthresh
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_max_syn_backlog
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_max_tw_buckets
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_mem
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_min_tso_segs
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_moderate_rcvbuf
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_mtu_probing
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_no_metrics_save
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_orphan_retries
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_reordering
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_retrans_collapse
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_retries1
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_retries2
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_rfc1337
-rw-r--r-- 1 root root 0 May 27 23:17 /proc/sys/net/ipv4/tcp_rmem
-rw-r--r-- 1 root root 0 May 27 23:19 /proc/sys/net/ipv4/tcp_sack
```

如何正确有效的使用这些参数，来提高 TCP 性能是一个不那么简单事情。我们需要针对 TCP 每个阶段的问题来对症下药，而不是病急乱投医。

接下来，将以三个角度来阐述提升 TCP 的策略，分别是：

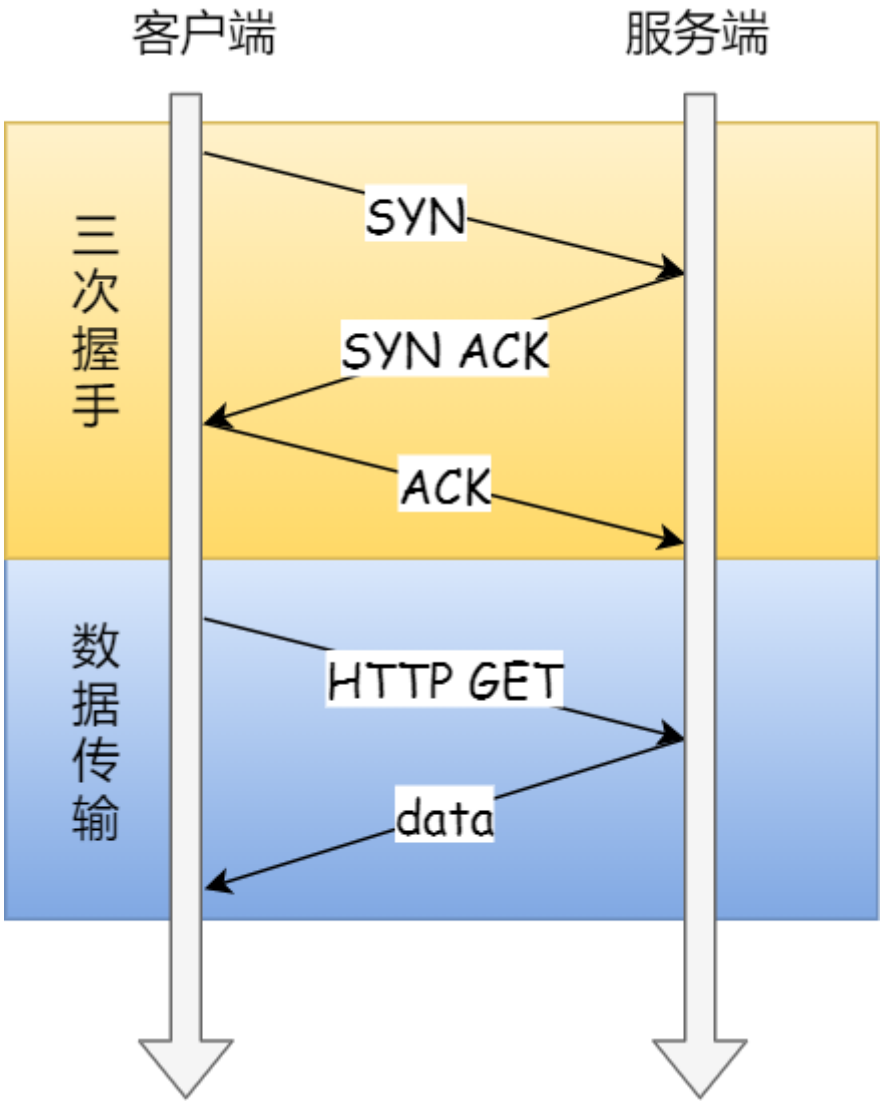
- TCP 三次握手的性能提升；
- TCP 四次挥手的性能提升；
- TCP 数据传输的性能提升；



正文

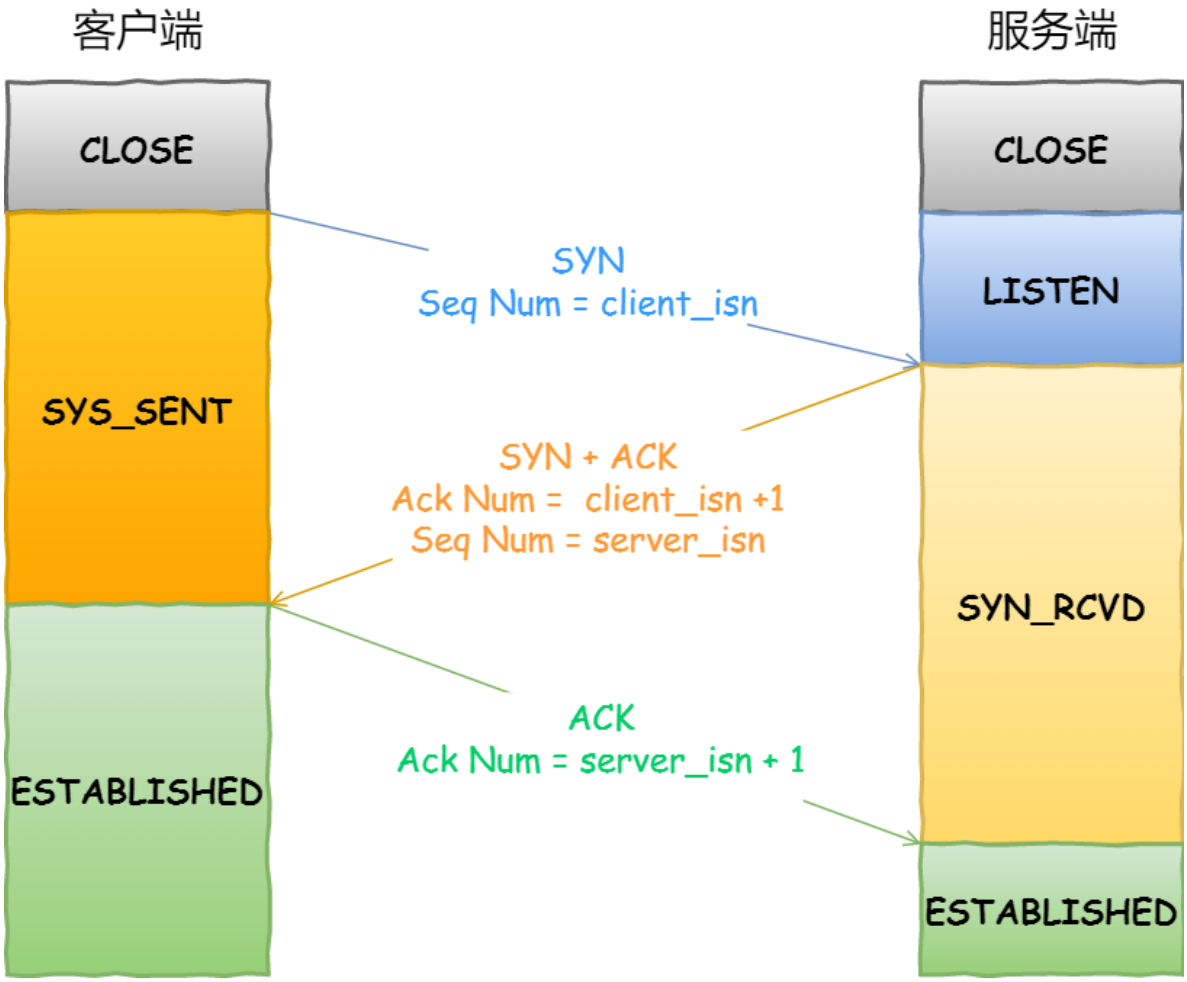
01 TCP 三次握手的性能提升

TCP 是面向连接的、可靠的、双向传输的传输层通信协议，所以在传输数据之前需要经过三次握手才能建立连接。



那么，三次握手的过程在一个 HTTP 请求的平均时间占比 10% 以上，在网络状态不佳、高并发或者遭遇 SYN 攻击等场景中，如果不能有效正确的调节三次握手中的参数，就会对性能产生很多的影响。

如何正确有效的使用这些参数，来提高 TCP 三次握手的性能，这就需要理解「三次握手的状态变迁」，这样当出现问题时，先用 `netstat` 命令查看是哪个握手阶段出现了问题，再来对症下药，而不是病急乱投医。



客户端和服务端都可以针对三次握手优化性能。主动发起连接的客户端优化相对简单些，而服务端需要监听端口，属于被动连接方，其间保持许多的中间状态，优化方法相对复杂一些。

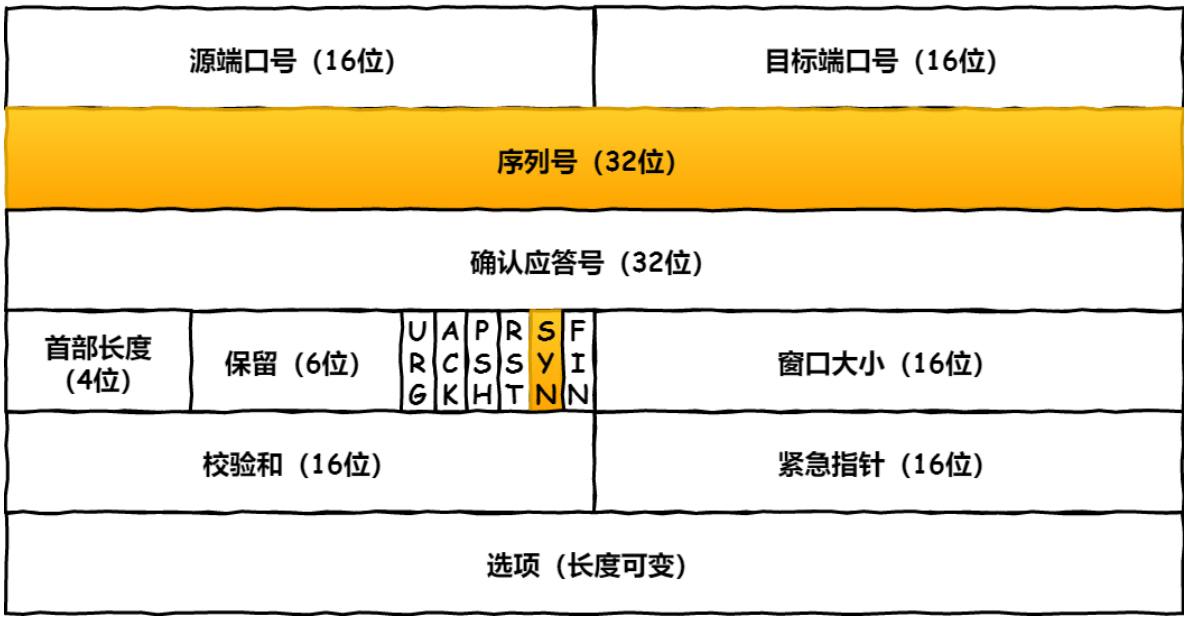
所以，客户端（主动发起连接方）和服务端（被动连接方）优化的方式是不同的，接下来分别针对客户端和服务端优化。

客户端优化

三次握手建立连接的首要目的是「同步序列号」。

只有同步了序列号才有可靠传输，TCP 许多特性都依赖于序列号实现，比如流量控制、丢包重传等，这也是三次握手中的报文称为 SYN 的原因，SYN 的全称就叫 *Synchronize Sequence Numbers*（同步序列号）。

TCP 头部格式



> SYN_SENT 状态的优化

客户端作为主动发起连接方，首先它将发送 SYN 包，于是客户端的连接就会处于 `SYN_SENT` 状态。

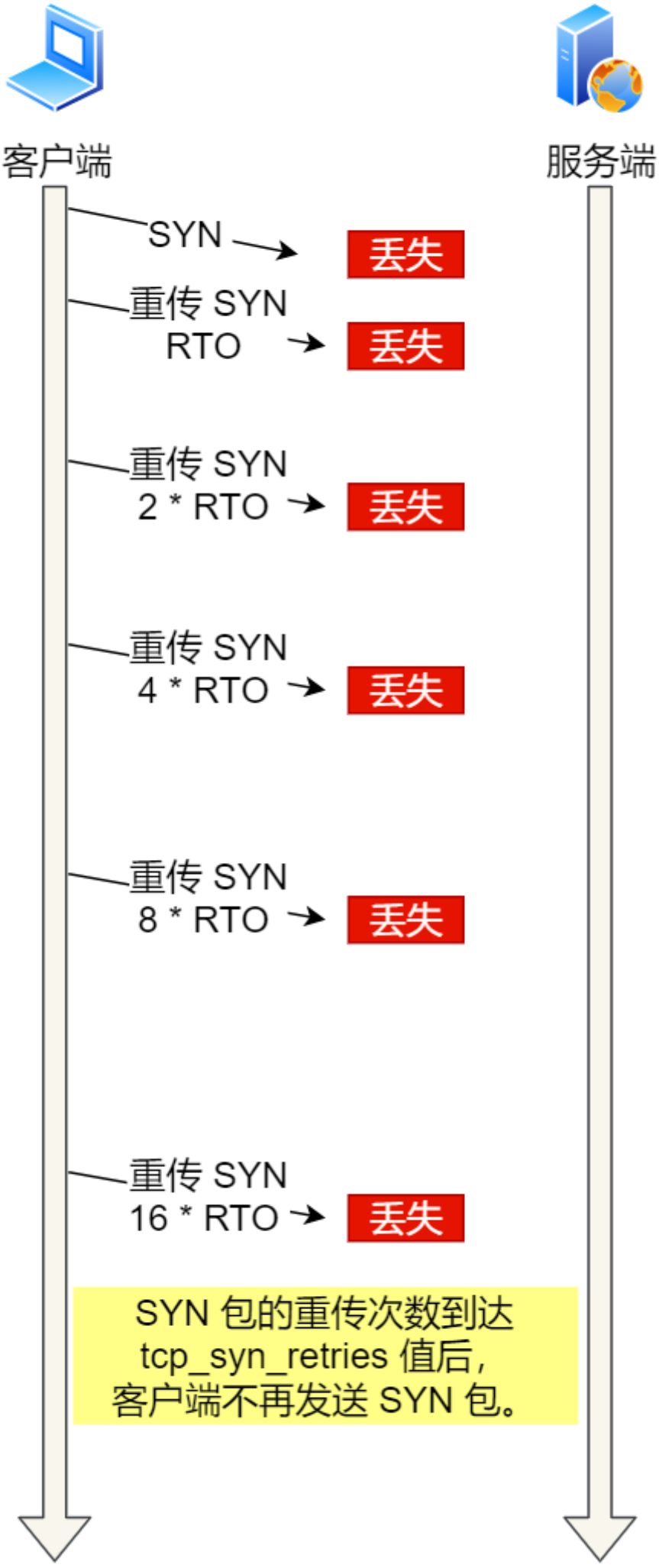
客户端在等待服务端回复的 ACK 报文，正常情况下，服务器会在几毫秒内返回 SYN+ACK ，但如果客户端长时间没有收到 SYN+ACK 报文，则会重发 SYN 包，**重发的次数由 tcp_syn_retries 参数控制**，默认是 5 次：

```
# tcp_syn_retries 控制 SYN 包重传的次数，默认值是 5 次
$ echo 5 > /proc/sys/net/ipv4/tcp_syn_retries
```

通常，第一次超时重传是在 1 秒后，第二次超时重传是在 2 秒，第三次超时重传是在 4 秒后，第四次超时重传是在 8 秒后，第五次是在超时重传 16 秒后。没错，**每次超时的时间是上一次的 2 倍。**

当第五次超时重传后，会继续等待 32 秒，如果服务端仍然没有回应 ACK，客户端就会终止三次握手。

所以，总耗时是 1+2+4+8+16+32=63 秒，大约 1 分钟左右。

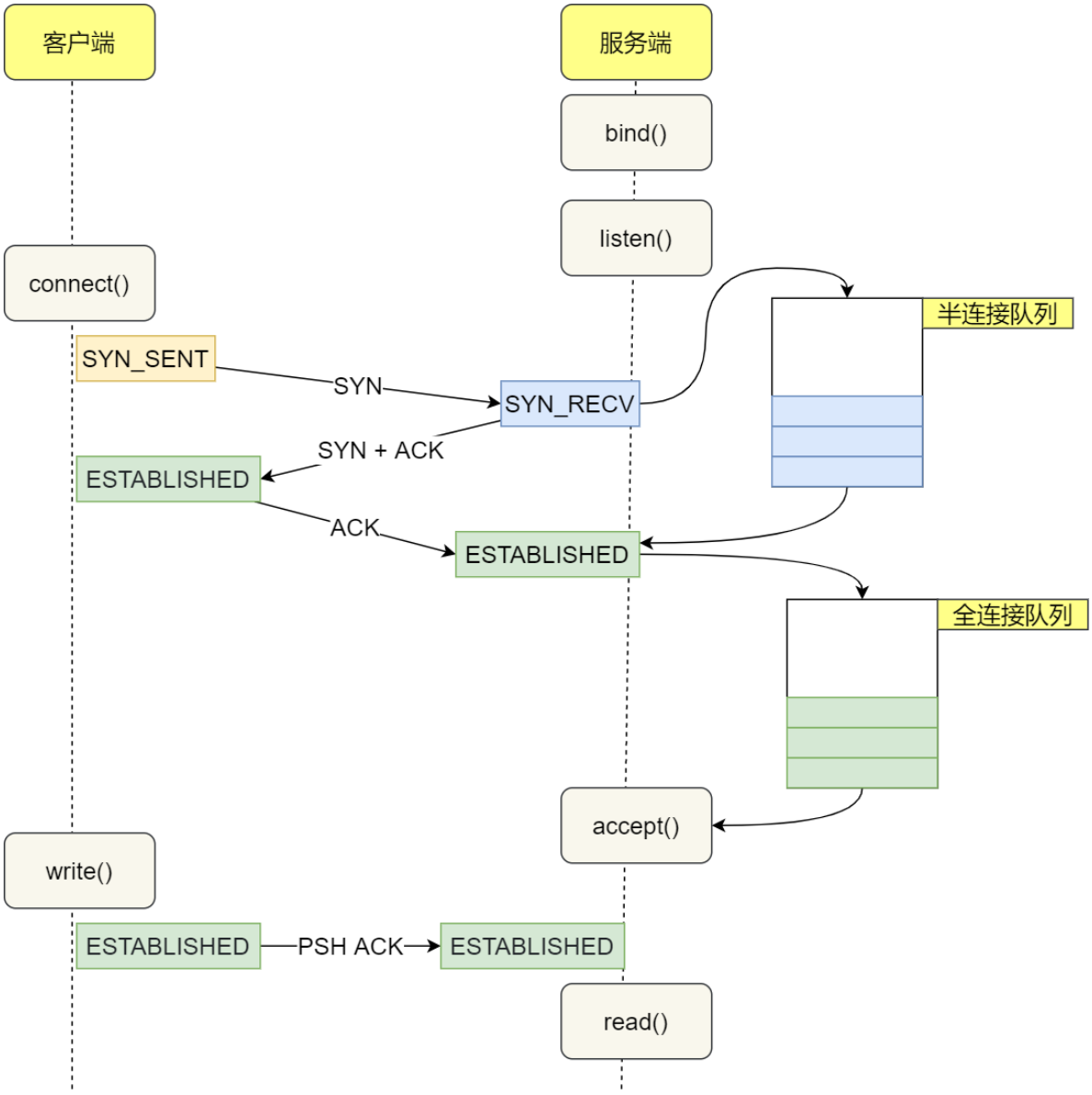

客户端
服务端
SYN
丢失
重传 SYN
RTO
丢失
重传 SYN
2 * RTO
丢失
重传 SYN
4 * RTO
丢失
重传 SYN
8 * RTO
丢失
重传 SYN
16 * RTO
丢失
SYN 包的重传次数到达
tcp_syn_retries 值后，
客户端不再发送 SYN 包。

你可以根据网络的稳定性和目标服务器的繁忙程度修改 SYN 的重传次数，调整客户端的三次握手时间上限。比如内网中通讯时，就可以适当调低重试次数，尽快把错误暴露给应用程序。

服务端优化

当服务端收到 SYN 包后，服务端会立马回复 SYN+ACK 包，表明确认收到了客户端的序列号，同时也把自己的序列号发给对方。

此时，服务端出现了新连接，状态是 `SYN_RCV`。在这个状态下，Linux 内核就会建立一个「半连接队列」来维护「未完成」的握手信息，当半连接队列溢出后，服务端就无法再建立新的连接。



SYN 攻击，攻击的是就是这个半连接队列。

> 如何查看由于 SYN 半连接队列已满，而被丢弃连接的情况?

我们可以通过该 `netstat -s` 命令给出的统计结果中， 可以得到由于半连接队列已满，引发的失败次数：

```
$ netstat -s | grep "SYNs to LISTEN"
    3479887 SYNs to LISTEN sockets dropped
$ netstat -s | grep "SYNs to LISTEN"
    3526030 SYNs to LISTEN sockets dropped
```

上面输出的数值是**累计值**，表示共有多少个 TCP 连接因为半连接队列溢出而被丢弃。**隔几秒执行几次，如果有上升的趋势，说明当前存在半连接队列溢出的现象**。

> 如何调整 SYN 半连接队列大小?

要想增大半连接队列，**不能只单纯增大 tcp_max_syn_backlog 的值，还需一同增大 somaxconn 和 backlog，也就是增大 accept 队列。否则，只单纯增大 tcp_max_syn_backlog 是无效的。**

增大 tcp_max_syn_backlog 和 somaxconn 的方法是修改 Linux 内核参数：

```
# 增大 tcp_max_syn_backlog
$ echo 1024 > /proc/sys/net/ipv4/tcp_max_syn_backlog

# 增大 somaxconn
$ echo 1024 > /proc/sys/net/core/somaxconn
```

增大 backlog 的方式，每个 Web 服务都不同，比如 Nginx 增大 backlog 的方法如下：

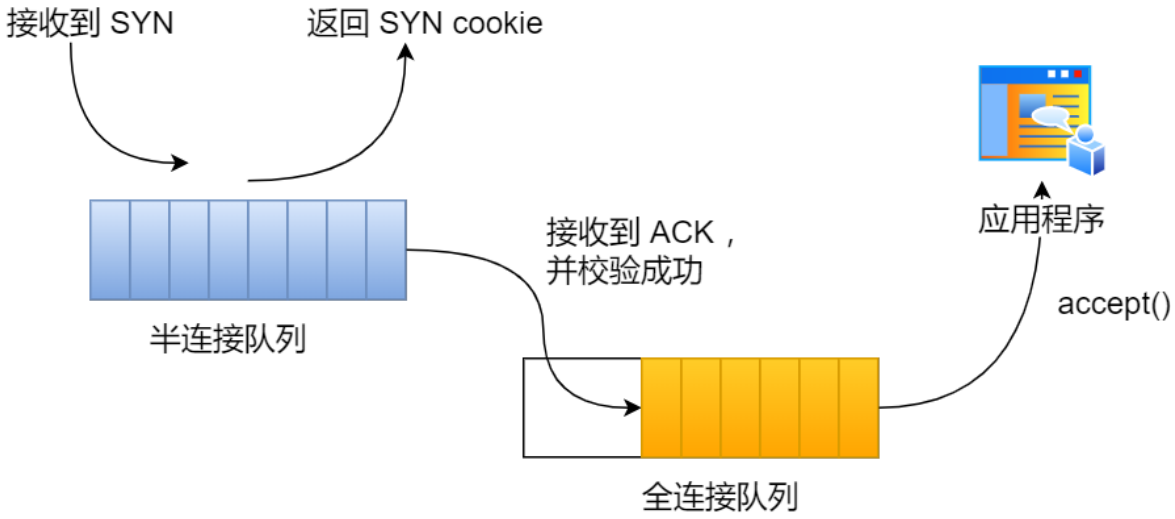
```
# /usr/local/nginx/conf/nginx.conf
server {
        listen  8088 default backlog=1024;
        server_name  localhost;
        ....
}
```

最后，改变了如上这些参数后，要重启 Nginx 服务，因为 SYN 半连接队列和 accept 队列都是在 `listen()` 初始化的。

> 如果 SYN 半连接队列已满，只能丢弃连接吗？

并不是这样，**开启 syncookies 功能就可以在不使用 SYN 半连接队列的情况下成功建立连接**。

syncookies 的工作原理：服务器根据当前状态计算出一个值，放在己方发出的 SYN+ACK 报文中发出，当客户端返回 ACK 报文时，取出该值验证，如果合法，就认为连接建立成功，如下图所示。



syncookies 参数主要有以下三个值：

- 0 值，表示关闭该功能；
- 1 值，表示仅当 SYN 半连接队列放不下时，再启用它；
- 2 值，表示无条件开启功能；

那么在应对 SYN 攻击时，只需要设置为 1 即可：

```
# 开启 tcp_syncookies 功能，默认是开启的
$ echo 1 > /proc/sys/net/ipv4/tcp_syncookies
```

> SYN_RCV 状态的优化

当客户端接收到服务器发来的 SYN+ACK 报文后，就会回复 ACK 给服务器，同时客户端连接状态从 SYN_SENT 转换为 ESTABLISHED，表示连接建立成功。

服务器端连接成功建立的时间还要再往后，等到服务端收到客户端的 ACK 后，服务端的连接状态才变为 ESTABLISHED。

如果服务器没有收到 ACK，就会重发 SYN+ACK 报文，同时一直处于 SYN_RCV 状态。

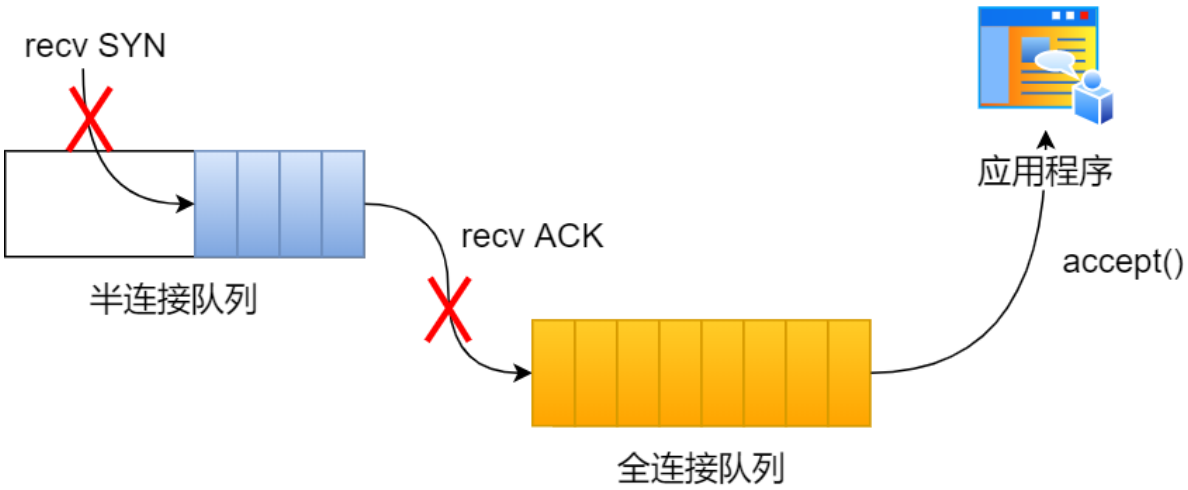
当网络繁忙、不稳定时，报文丢失就会变严重，此时应该调大重发次数。反之则可以调小重发次数。**修改重发次数的方法是，调整 tcp_synack_retries 参数**：

```
# tcp_synack_retries 控制 SYN+ACK 包重传的次数，默认值是 5 次
$ echo 5 > /proc/sys/net/ipv4/tcp_synack_retries
```

tcp_synack_retries 的默认重试次数是 5 次，与客户端重传 SYN 类似，它的重传会经历 1、2、4、8、16 秒，最后一次重传后会继续等待 32 秒，如果服务端仍然没有收到 ACK，才会关闭连接，故共需要等待 63 秒。

服务器收到 ACK 后连接建立成功，此时，内核会把连接从半连接队列移除，然后创建新的完全的连接，并将其添加到 accept 队列，等待进程调用 accept 函数时把连接取出来。

如果进程不能及时地调用 accept 函数，就会造成 accept 队列（也称全连接队列）溢出，最终导致建立好的 TCP 连接被丢弃。



> accept 队列已满，只能丢弃连接吗?

丢弃连接只是 Linux 的默认行为，我们还可以选择向客户端发送 RST 复位报文，告诉客户端连接已经建立失败。打开这一功能需要将 tcp_abort_on_overflow 参数设置为 1。

```
# 打开后，则当 accpet 队列满了会回 RST，默认值是 0 关闭
$ echo 1 > /proc/sys/net/ipv4/tcp_abort_on_overflow
```

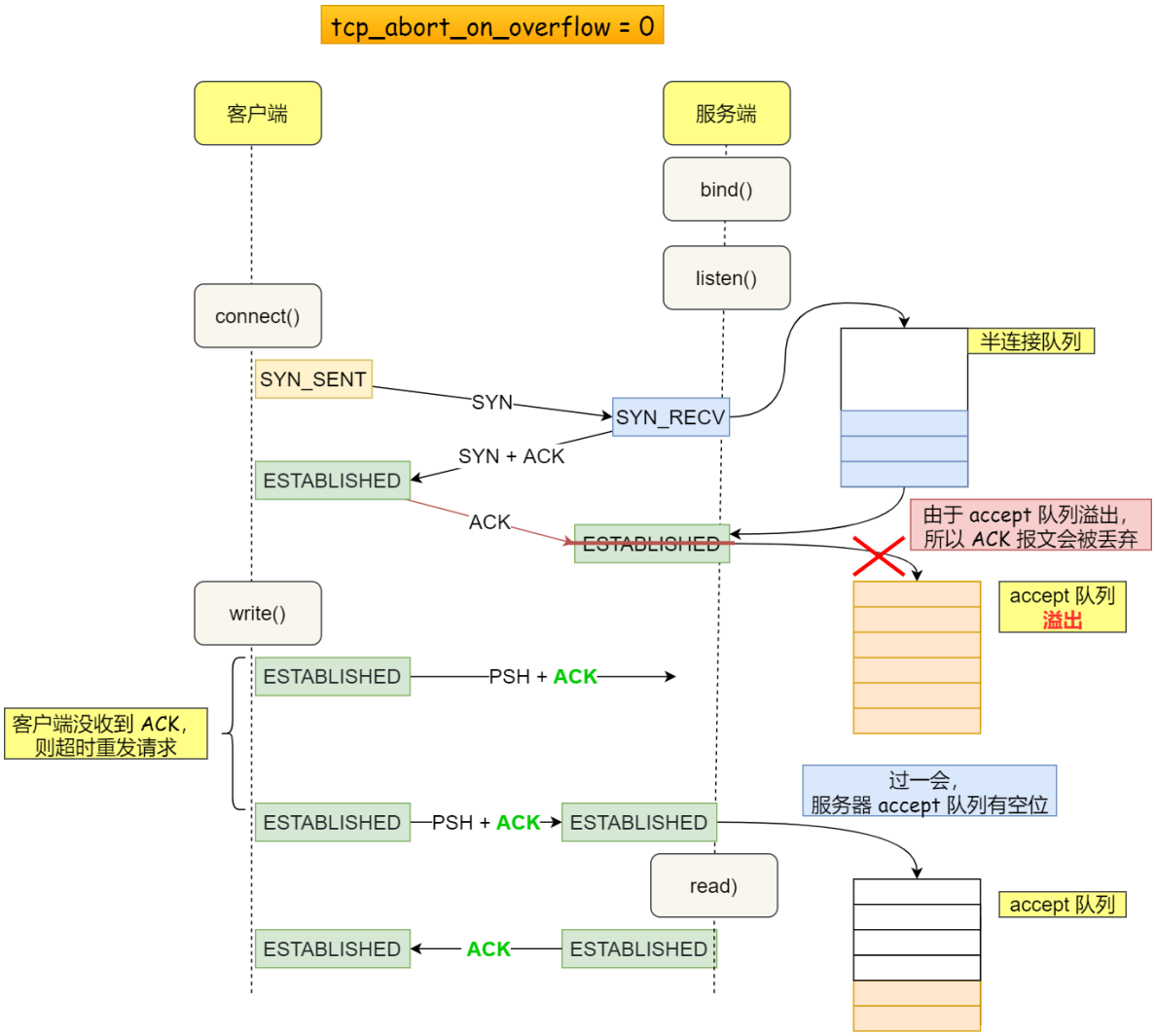
tcp_abort_on_overflow 共有两个值分别是 0 和 1，其分别表示：

- 0：如果 accept 队列满了，那么 server 扔掉 client 发过来的 ack；
- 1：如果 accept 队列满了，server 发送一个 `RST` 包给 client，表示废掉这个握手过程和这个连接；

如果要想知道客户端连接不上服务端，是不是服务端 TCP 全连接队列满的原因，那么可以把 tcp_abort_on_overflow 设置为 1，这时如果在客户端异常中可以看到很多 `connection reset by peer` 的错误，那么就可以证明是由于服务端 TCP 全连接队列溢出的问题。

通常情况下，应当把 tcp_abort_on_overflow 设置为 0，因为这样更有利于应对突发流量。

举个例子，当 accept 队列满导致服务器丢掉了 ACK，与此同时，客户端的连接状态却是 ESTABLISHED，客户端进程就在建立好的连接上发送请求。只要服务器没有为请求回复 ACK，客户端的请求就会被多次「重发」。**如果服务器上的进程只是短暂的繁忙造成 accept 队列满，那么当 accept 队列有空位时，再次接收到的请求报文由于含有 ACK，仍然会触发服务器端成功建立连接。**



所以，tcp_abort_on_overflow 设为 0 可以提高连接建立的成功率，只有你非常肯定 TCP 全连接队列会长期溢出时，才能设置为 1 以尽快通知客户端。

> 如何调整 accept 队列的长度呢?

accept 队列的长度取决于 somaxconn 和 backlog 之间的最小值，也就是 min(somaxconn, backlog)，其中：

- somaxconn 是 Linux 内核的参数，默认值是 128，可以通过 `net.core.somaxconn` 来设置其值；

- backlog 是 `listen(int sockfd, int backlog)` 函数中的 backlog 大小；

Tomcat、Nginx、Apache 常见的 Web 服务的 backlog 默认值都是 511。

> 如何查看服务端进程 accept 队列的长度?

可以通过 `ss -ltn` 命令查看：

```
# -l 显示正在监听（ listening ）的 socket
# -n 不解析服务名称
# -t 只显示 tcp socket
$ ss -lnt
State       Recv-Q Send-Q     Local Address:Port      Peer Address:Port
LISTEN      0      128         *:8088                 *:*
```

- Recv-Q：当前 accept 队列的大小，也就是当前已完成三次握手并等待服务端 `accept()` 的 TCP 连接；
- Send-Q：accept 队列最大长度，上面的输出结果说明监听 8088 端口的 TCP 服务，accept 队列的最大长度为 128；

> 如何查看由于 accept 连接队列已满，而被丢弃的连接?

当超过了 accept 连接队列，服务端则会丢掉后续进来的 TCP 连接，丢掉的 TCP 连接的个数会被统计起来，我们可以使用 netstat -s 命令来查看：

```
# 查看 TCP accpet 队列溢出情况
$ date;netstat -s | grep overflowed
Sun May 17 07:35:40 CST 2020
    41150 times the listen queue of a socket overflowed

$ date;netstat -s | grep overflowed
Sun May 17 07:35:41 CST 2020
    42512 times the listen queue of a socket overflowed
```

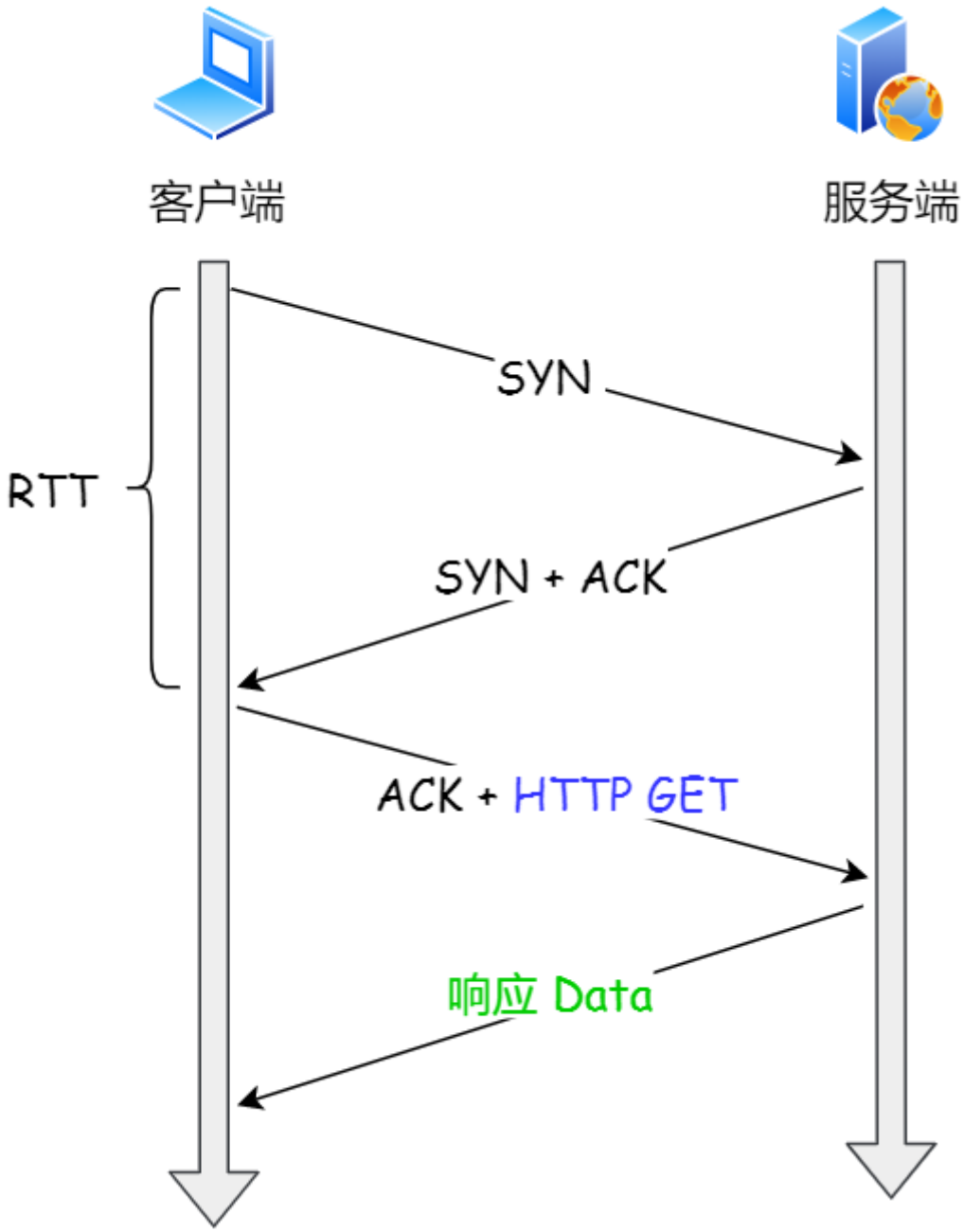
上面看到的 41150 times ，表示 accept 队列溢出的次数，注意这个是累计值。可以隔几秒钟执行下，如果这个数字一直在增加的话，说明 accept 连接队列偶尔满了。

如果持续不断地有连接因为 accept 队列溢出被丢弃，就应该调大 backlog 以及 somaxconn 参数。

如何绕过三次握手?

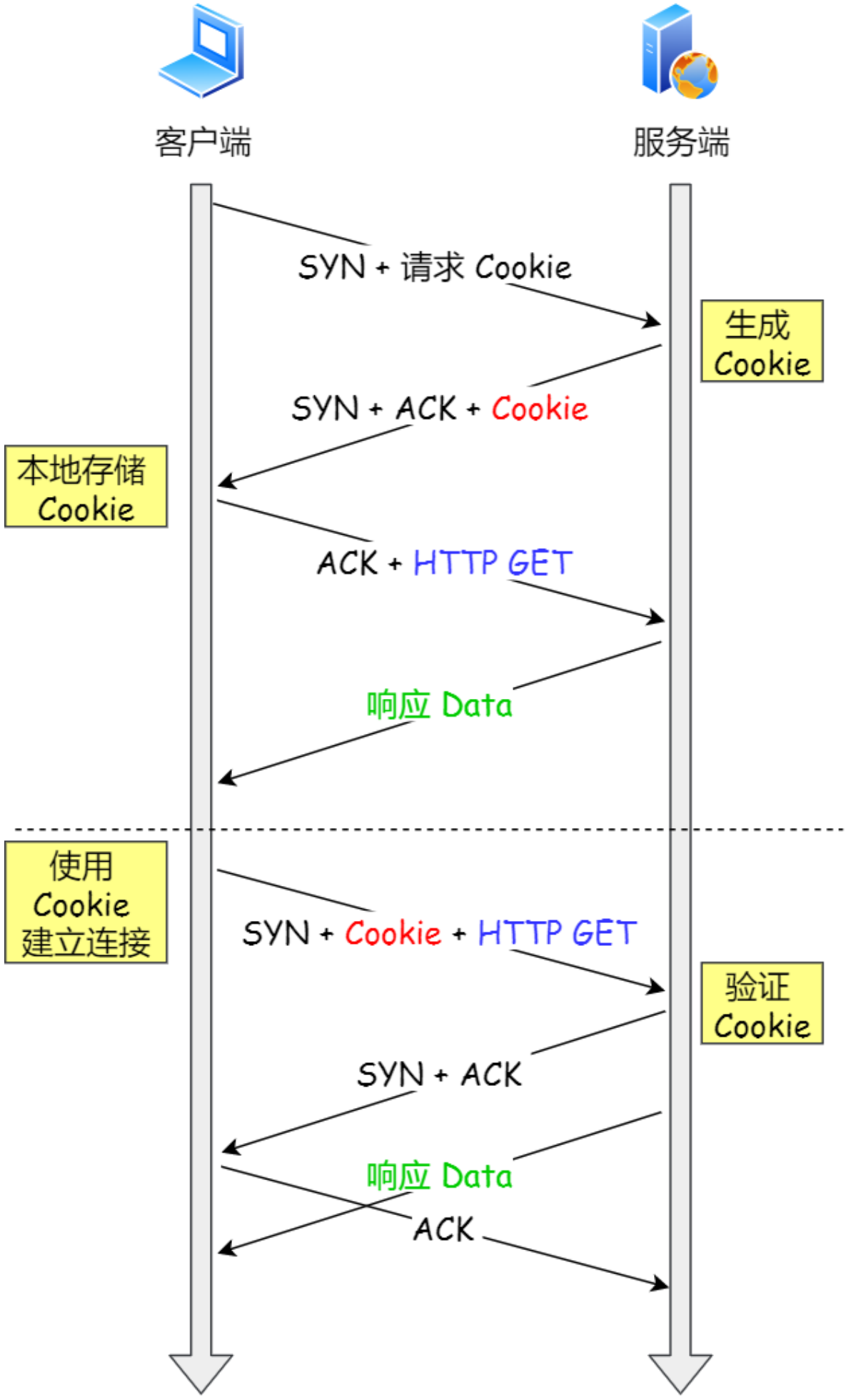
以上我们只是在对三次握手的过程进行优化，接下来我们看看如何绕过三次握手发送数据。

三次握手建立连接造成的后果就是，HTTP 请求必须在一个 RTT（从客户端到服务器一个往返的时间）后才能发送。



在 Linux 3.7 内核版本之后，提供了 TCP Fast Open 功能，这个功能可以减少 TCP 连接建立的时延。

> 接下来说说，TCP Fast Open 功能的工作方式。



在客户端首次建立连接时的过程：

1. 客户端发送 SYN 报文，该报文包含 Fast Open 选项，且该选项的 Cookie 为空，这表明客户端请求 Fast Open Cookie；
2. 支持 TCP Fast Open 的服务器生成 Cookie，并将其置于 SYN-ACK 数据包中的 Fast Open 选项以发回客户端；
3. 客户端收到 SYN-ACK 后，本地缓存 Fast Open 选项中的 Cookie。

所以，第一次发起 HTTP GET 请求的时候，还是需要正常的三次握手流程。

之后，如果客户端再次向服务器建立连接时的过程：

1. 客户端发送 SYN 报文，该报文包含「数据」（对于非 TFO 的普通 TCP 握手过程，SYN 报文中不包含「数据」）以及此前记录的 Cookie；
2. 支持 TCP Fast Open 的服务器会对收到 Cookie 进行校验：如果 Cookie 有效，服务器将在 SYN-ACK 报文中对 SYN 和「数据」进行确认，服务器随后将「数据」递送至相应的应用程序；如果 Cookie 无效，服务器将丢弃 SYN 报文中包含的「数据」，且其随后发出的 SYN-ACK 报文将只确认 SYN 的对应序列号；
3. 如果服务器接受了 SYN 报文中的「数据」，服务器可在握手完成之前发送「数据」，**这就减少了握手带来的 1 个 RTT 的时间消耗**；
4. 客户端将发送 ACK 确认服务器发回的 SYN 以及「数据」，但如果客户端在初始的 SYN 报文中发送的「数据」没有被确认，则客户端将重新发送「数据」；
5. 此后的 TCP 连接的数据传输过程和非 TFO 的正常情况一致。

所以，之后发起 HTTP GET 请求的时候，可以绕过三次握手，这就减少了握手带来的 1 个 RTT 的时间消耗。

开启了 TFO 功能，cookie 的值是存放到 TCP option 字段里的：

TCP option 字段			
类型	**总长度（字节）**	**数据**	**描述**
0	-	-	选项列表末尾标识
1	-	-	没有意义，用于 32 位对齐使用
2	4	MSS 值	三次握手时，发送端告知可以接收的最大报文段大小
3	3	窗口移位	指明最大窗口扩展后的大小
4	2	-	表明支持 SACK 选择性确认功能
5	可变	确认报文段	选择性确认窗口中间的报文段
8	10	Timestamps 时间戳	用于更精准的计算 RTT，以及解决序列号绕回的问题
14	3	校验和算法	双方认可后，可使用新的校验和算法
15	可变	校验和	当 16 位标准校验和放不下时，放置在这里
34	可变	FOC	TFO中Cookie

注：客户端在请求并存储了 Fast Open Cookie 之后，可以不断重复 TCP Fast Open 直至服务器认为 Cookie 无效（通常为过期）。

> Linux 下怎么打开 TCP Fast Open 功能呢?

在 Linux 系统中，可以通过**设置 tcp_fastopn 内核参数，来打开 Fast Open 功能**:

```
# 无论作为客户端还是服务器，都可以使用 Fast Open 功能
$ echo 3 > /proc/sys/net/ipv4/tcp_fastopen
```

tcp_fastopn 各个值的意义:

- 0 关闭
- 1 作为客户端使用 Fast Open 功能
- 2 作为服务端使用 Fast Open 功能
- 3 无论作为客户端还是服务器，都可以使用 Fast Open 功能

TCP Fast Open 功能需要客户端和服务端同时支持，才有效果。

小结

本小结主要介绍了关于优化 TCP 三次握手的几个 TCP 参数。

优化三次握手的策略	
策略	**TCP 内核参数**
调整 SYN 报文的重传次数	tcp_syn_retries
调整 SYN 半连接队列长度	tcp_max_syn_backlog、somaxconn、backlog
调整 SYN+ACK 报文的重传次数	tcp_synack_retries
调整 accpet 队列长度	min(backlog, somaxconn)
绕过三次握手	tcp_fastopen

> 客户端的优化

当客户端发起 SYN 包时，可以通过 `tcp_syn_retries` 控制其重传的次数。

> 服务端的优化

当服务端 SYN 半连接队列溢出后，会导致后续连接被丢弃，可以通过 `netstat -s` 观察半连接队列溢出的情况，如果 SYN 半连接队列溢出情况比较严重，可以通过 `tcp_max_syn_backlog、somaxconn、backlog` 参数来调整 SYN 半连接队列的大小。

服务端回复 SYN+ACK 的重传次数由 `tcp_synack_retries` 参数控制。如果遭受 SYN 攻击，应把 `tcp_syncookies` 参数设置为 1，表示仅在 SYN 队列满后开启 syncookie 功能，可以保证正常的连接成功建立。

服务端收到客户端返回的 ACK，会把连接移入 accpet 队列，等待进行调用 accpet() 函数取出连接。

可以通过 `ss -lnt` 查看服务端进程的 accept 队列长度，如果 accept 队列溢出，系统默认丢弃 ACK，如果可以把 `tcp_abort_on_overflow` 设置为 1 ，表示用 RST 通知客户端连接建立失败。

如果 accpet 队列溢出严重，可以通过 listen 函数的 `backlog` 参数和 `somaxconn` 系统参数提高队列大小，accept 队列长度取决于 min(backlog, somaxconn)。

> 绕过三次握手

TCP Fast Open 功能可以绕过三次握手，使得 HTTP 请求减少了 1 个 RTT 的时间，Linux 下可以通过 `tcp_fastopen` 开启该功能，同时必须保证服务端和客户端同时支持。

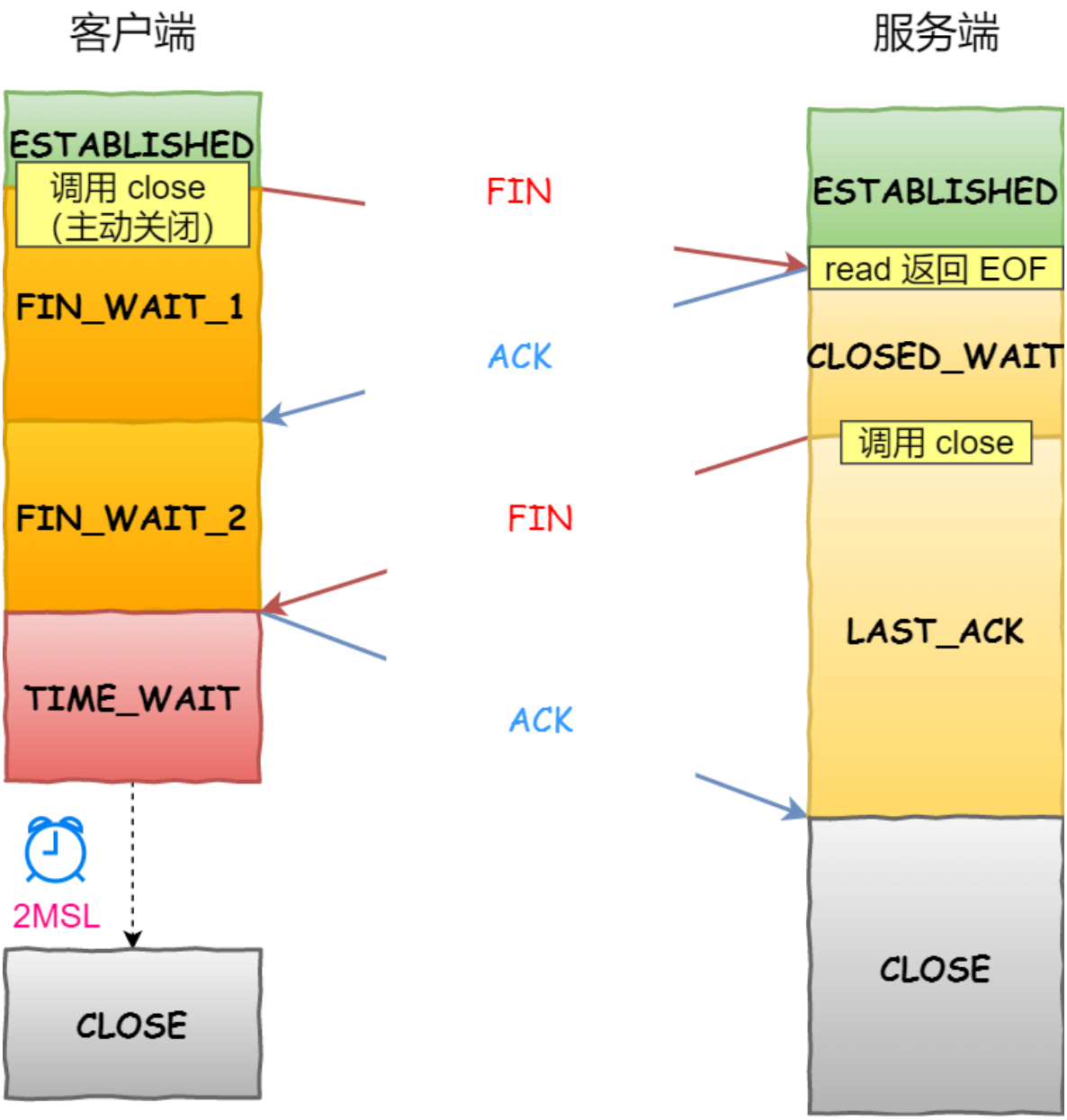
---

02 TCP 四次挥手的性能提升

接下来，我们一起看看针对 TCP 四次挥手关闭连接时，如何优化性能。

在开始之前，我们得先了解四次挥手状态变迁的过程。

客户端和服务端双方都可以主动断开连接，**通常先关闭连接的一方称为主动方，后关闭连接的一方称为被动方。**



可以看到，**四次挥手过程只涉及了两种报文，分别是 FIN 和 ACK**：

- FIN 就是结束连接的意思，谁发出 FIN 报文，就表示它将不会再发送任何数据，关闭这一方向上的传输通道；
- ACK 就是确认的意思，用来通知对方：你方的发送通道已经关闭；

四次挥手的过程：

- 当主动方关闭连接时，会发送 FIN 报文，此时发送方的 TCP 连接将从 ESTABLISHED 变成 FIN_WAIT1。
- 当被动方收到 FIN 报文后，内核会自动回复 ACK 报文，连接状态将从 ESTABLISHED 变成 CLOSE_WAIT，表示被动方在等待进程调用 close 函数关闭连接。
- 当主动方收到这个 ACK 后，连接状态由 FIN_WAIT1 变为 FIN_WAIT2，也就是表示**主动方的发送通道就关闭了**。
- 当被动方进入 CLOSE_WAIT 时，被动方还会继续处理数据，等到进程的 read 函数返回 0 后，应用程序就会调用 close 函数，进而触发内核发送 FIN 报文，此时被动方的连接状态变为 LAST_ACK。
- 当主动方收到这个 FIN 报文后，内核会回复 ACK 报文给被动方，同时主动方的连接状态由 FIN_WAIT2 变为 TIME_WAIT，**在 Linux 系统下大约等待 1 分钟后，TIME_WAIT 状态的连接才**

会彻底关闭。

- 当被动方收到最后的 ACK 报文后，**被动方的连接就会关闭**。

你可以看到，每个方向都需要**一个 FIN 和一个 ACK**，因此通常被称为**四次挥手**。

这里一点需要注意是：**主动关闭连接的，才有 TIME_WAIT 状态。**

主动关闭方和被动关闭方优化的思路也不同，接下来分别说说如何优化他们。

主动方的优化

关闭连接的方式通常有两种，分别是 RST 报文关闭和 FIN 报文关闭。

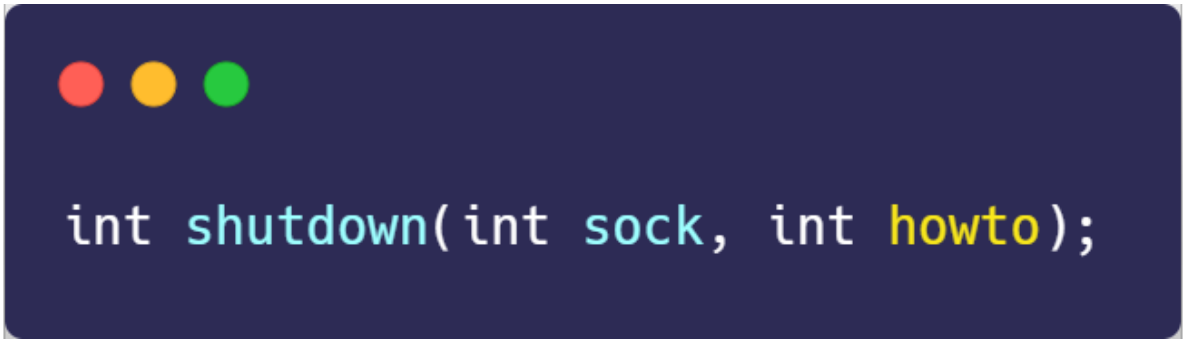
如果进程异常退出了，内核就会发送 RST 报文来关闭，它可以不走四次挥手流程，是一个暴力关闭连接的方式。

安全关闭连接的方式必须通过四次挥手，它由进程调用 `close` 和 `shutdown` 函数发起 FIN 报文（shutdown 参数须传入 SHUT_WR 或者 SHUT_RDWR 才会发送 FIN）。

> 调用 close 函数和 shutdown 函数有什么区别?

调用了 close 函数意味着完全断开连接，**完全断开不仅指无法传输数据，而且也不能发送数据。 此时，调用了 close 函数的一方的连接叫做「孤儿连接」，如果你用 netstat -p 命令，会发现连接对应的进程名为空。**

使用 close 函数关闭连接是不优雅的。于是，就出现了一种优雅关闭连接的 `shutdown` 函数，**它可以控制只关闭一个方向的连接**：



第二个参数决定断开连接的方式，主要有以下三种方式：

- SHUT_RD(0)：**关闭连接的「读」这个方向**，如果接收缓冲区有已接收的数据，则将会被丢弃，并且后续再收到新的数据，会对数据进行 ACK，然后悄悄地丢弃。也就是说，对端还是会接收到 ACK，在这种情况下根本不知道数据已经被丢弃了。
- SHUT_WR(1)：**关闭连接的「写」这个方向**，这就是常被称为「半关闭」的连接。如果发送缓冲区还有未发送的数据，将被立即发送出去，并发送一个 FIN 报文给对端。
- SHUT_RDWR(2)：相当于 SHUT_RD 和 SHUT_WR 操作各一次，**关闭套接字的读和写两个方向**。

close 和 shutdown 函数都可以关闭连接，但这两种方式关闭的连接，不只功能上有差异，控制它们的 Linux 参数也不相同。

> FIN_WAIT1 状态的优化

主动方发送 FIN 报文后，连接就处于 FIN_WAIT1 状态，正常情况下，如果能及时收到被动方的 ACK，则会很快变为 FIN_WAIT2 状态。

但是当迟迟收不到对方返回的 ACK 时，连接就会一直处于 FIN_WAIT1 状态。此时，**内核会定时重发 FIN 报文，其中重发次数由 tcp_orphan_retries 参数控制**（注意，orphan 虽然是孤儿的意思，该参数却不只对孤儿连接有效，事实上，它对所有 FIN_WAIT1 状态下的连接都有效），默认值是 0。

```
# 调整 FIN 报文重传次数为 5 次，默认值是 0，特指 8 次
$ echo 5 > /proc/sys/net/ipv4/tcp_orphan_retries
```

你可能会好奇，这 0 表示几次？**实际上当为 0 时，特指 8 次**，从下面的内核源码可知：

```
/* Calculate maximal number or retries on an orphaned socket. */
static int tcp_orphan_retries(struct sock *sk, int alive)
{
        int retries = sysctl_tcp_orphan_retries; /* May be zero. */

         /* We know from an ICMP that something is wrong. */
         if (sk->sk_err_soft && !alive)
                 retries = 0;

         /* However, if socket sent something recently, select some safe
          * number of retries. 8 corresponds to >100 seconds with minimal
          * RTO of 200msec. */
         if (retries == 0 && alive)
                 retries = 8;
         return retries;
}
```

如果 FIN_WAIT1 状态连接很多，我们就需要考虑降低 tcp_orphan_retries 的值，当重传次数超过 tcp_orphan_retries 时，连接就会直接关闭掉。

对于普遍正常情况时，调低 tcp_orphan_retries 就已经可以了。如果遇到恶意攻击，FIN 报文根本无法发送出去，这由 TCP 两个特性导致的：

- 首先，TCP 必须保证报文是有序发送的，FIN 报文也不例外，当发送缓冲区还有数据没有发送时，FIN 报文也不能提前发送。
- 其次，TCP 有流量控制功能，当接收方接收窗口为 0 时，发送方就不能再发送数据。所以，当攻击者下载大文件时，就可以通过接收窗口设为 0 ，这就会使得 FIN 报文都无法发送出去，那么连接会一直处于 FIN_WAIT1 状态。

解决这种问题的方法，是**调整 tcp_max_orphans 参数，它定义了「孤儿连接」的最大数量**：

当进程调用了 `close` 函数关闭连接，此时连接就会是「孤儿连接」，因为它无法再发送和接收数据。Linux 系统为了防止孤儿连接过多，导致系统资源长时间被占用，就提供了 `tcp_max_orphans` 参数。如果孤儿连接数量大于它，新增的孤儿连接将不再走四次挥手，而是直接发送 RST 复位报文强制关闭。

> FIN_WAIT2 状态的优化

当主动方收到 ACK 报文后，会处于 FIN_WAIT2 状态，就表示主动方的发送通道已经关闭，接下来将等待对方发送 FIN 报文，关闭对方的发送通道。

这时，**如果连接是用 shutdown 函数关闭的，连接可以一直处于 FIN_WAIT2 状态，因为它可能还可以发送或接收数据。但对于 close 函数关闭的孤儿连接，由于无法再发送和接收数据，所以这个状态不可以持续太久，而 tcp_fin_timeout 控制了这个状态下连接的持续时长**，默认值是 60 秒：

它意味着对于孤儿连接（调用 close 关闭的连接），如果在 60 秒后还没有收到 FIN 报文，连接就会直接关闭。

这个 60 秒不是随便决定的，它与 TIME_WAIT 状态持续的时间是相同的，后面我们再来说说为什么是 60 秒。

> TIME_WAIT 状态的优化

TIME_WAIT 是主动方四次挥手的最后一个状态，也是最常遇见的状态。

当收到被动方发来的 FIN 报文后，主动方会立刻回复 ACK，表示确认对方的发送通道已经关闭，接着就处于 TIME_WAIT 状态。在 Linux 系统，TIME_WAIT 状态会持续 60 秒后才会进入关闭状态。

TIME_WAIT 状态的连接，在主动方看来确实快已经关闭了。然后，被动方没有收到 ACK 报文前，还是处于 LAST_ACK 状态。如果这个 ACK 报文没有到达被动方，被动方就会重发 FIN 报文。重发次数仍然由前面介绍过的 tcp_orphan_retries 参数控制。

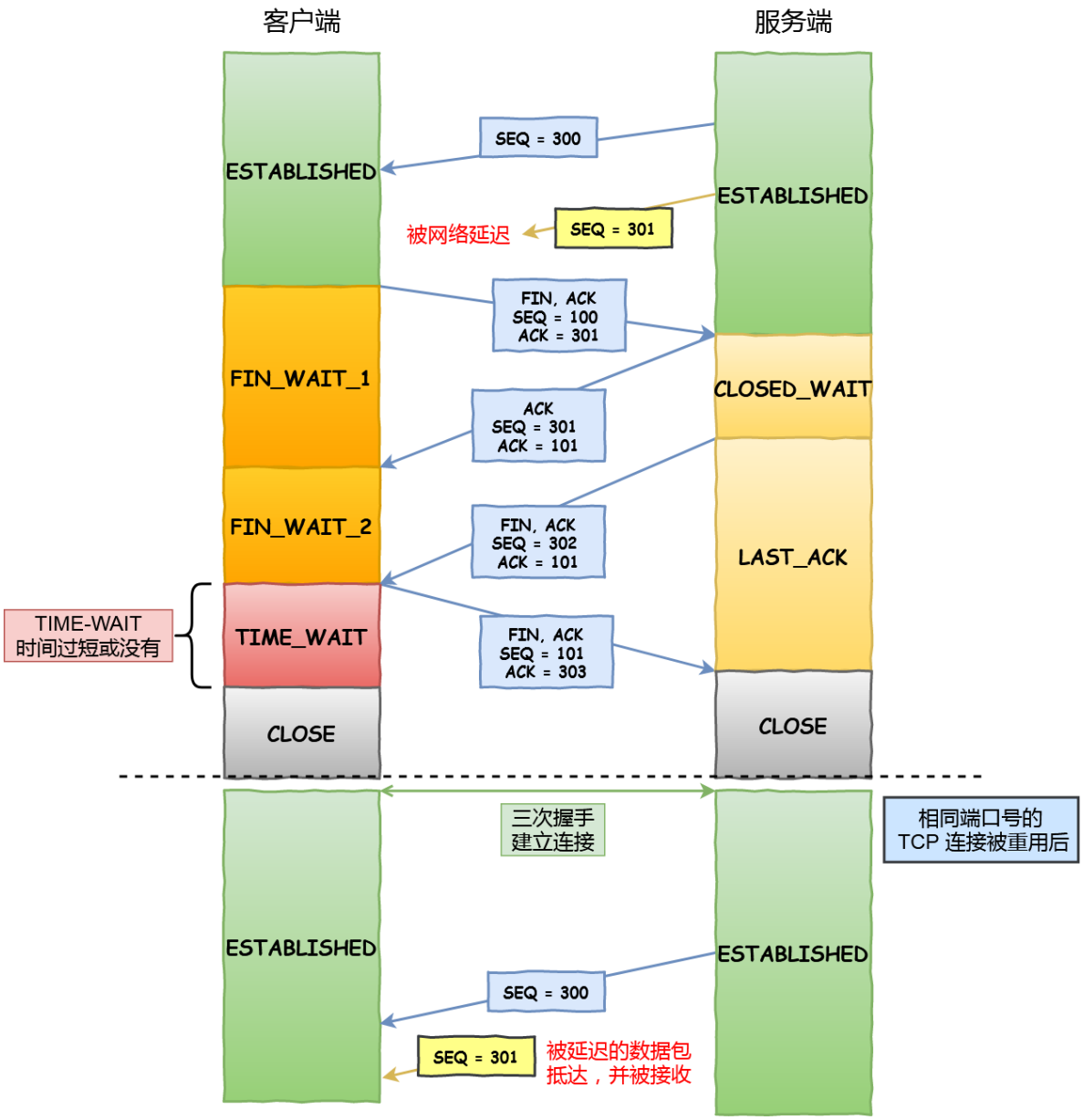
TIME-WAIT 的状态尤其重要，主要是两个原因：

- 防止具有相同「四元组」的「旧」数据包被收到；
- 保证「被动关闭连接」的一方能被正确的关闭，即保证最后的 ACK 能让被动关闭方接收，从而帮助其正常关闭；

原因一：防止旧连接的数据包

TIME-WAIT 的一个作用是**防止收到历史数据，从而导致数据错乱的问题。**

假设 TIME-WAIT 没有等待时间或时间过短，被延迟的数据包抵达后会发生什么呢？



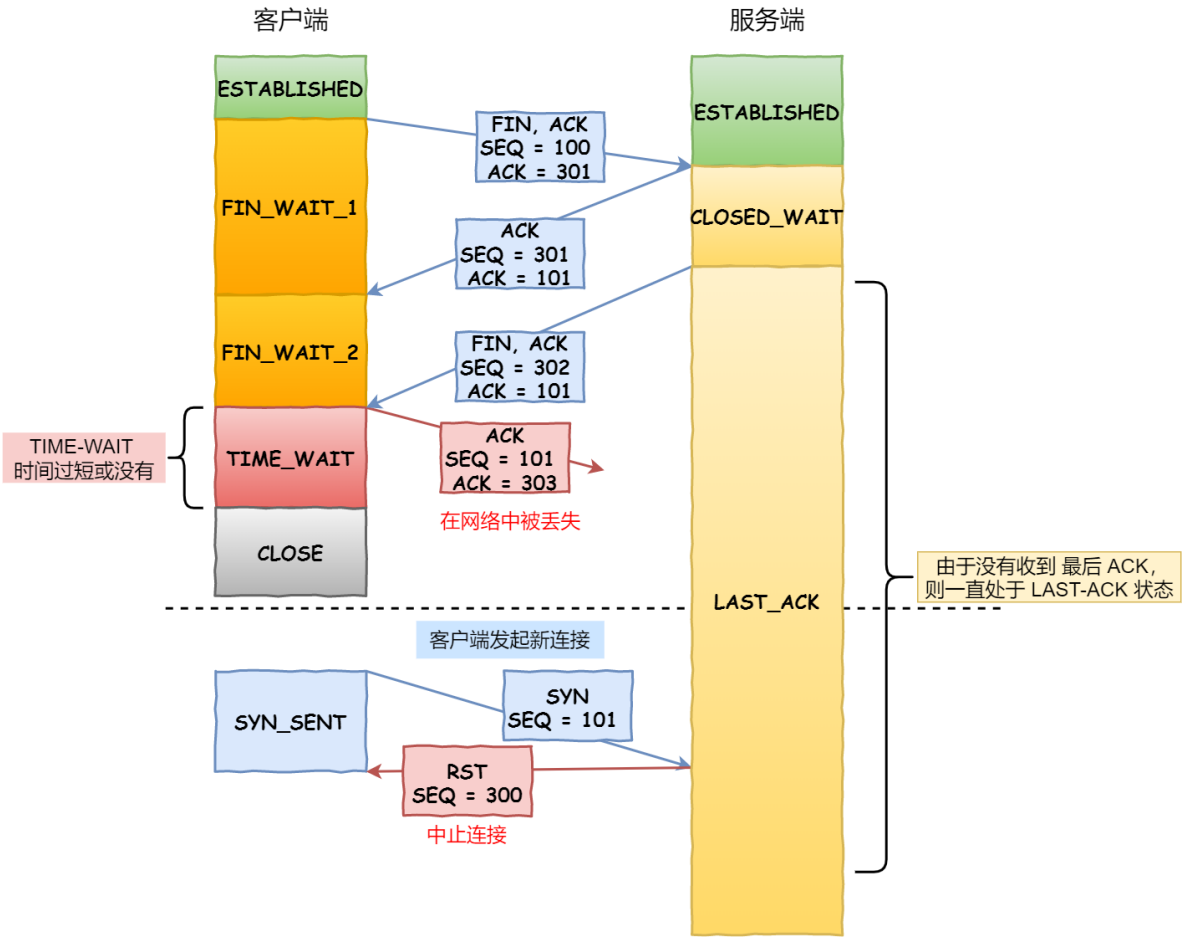
- 如上图黄色框框服务端在关闭连接之前发送的 `SEQ = 301` 报文，被网络延迟了。
- 这时有相同端口的 TCP 连接被复用后，被延迟的 `SEQ = 301` 抵达了客户端，那么客户端是有可能正常接收这个过期的报文，这就会产生数据错乱等严重的问题。

所以，TCP 就设计出了这么一个机制，经过 `2MSL` 这个时间，**足以让两个方向上的数据包都被丢弃，使得原来连接的数据包在网络中都自然消失，再出现的数据包一定都是新建立连接所产生的。**

原因二：保证连接正确关闭

TIME-WAIT 的另外一个作用是**等待足够的时间以确保最后的 ACK 能让被动关闭方接收，从而帮助其正常关闭。**

假设 TIME-WAIT 没有等待时间或时间过短，断开连接会造成什么问题呢？



- 如上图红色框框客户端四次挥手的最后一个 `ACK` 报文如果在网络中被丢失了，此时如果客户端 `TIME-WAIT` 过短或没有，则就直接进入了 `CLOSE` 状态了，那么服务端则会一直处在 `LAST-ACK` 状态。
- 当客户端发起建立连接的 `SYN` 请求报文后，服务端会发送 `RST` 报文给客户端，连接建立的过程就会被终止。

我们再回过头来看看，为什么 TIME_WAIT 状态要保持 60 秒呢？这与孤儿连接 FIN_WAIT2 状态默认保留 60 秒的原理是一样的，**因为这两个状态都需要保持 2MSL 时长。MSL 全称是 Maximum Segment Lifetime，它定义了一个报文在网络中的最长生存时间**（报文每经过一次路由器的转发，IP 头部的 TTL 字段就会减 1，减到 0 时报文就被丢弃，这就限制了报文的最长存活时间）。

为什么是 2 MSL 的时长呢？这其实是相当于**至少允许报文丢失一次**。比如，若 ACK 在一个 MSL 内丢失，这样被动方重发的 FIN 会在第 2 个 MSL 内到达，TIME_WAIT 状态的连接可以应对。

为什么不是 4 或者 8 MSL 的时长呢？你可以想象一个丢包率达到百分之一的糟糕网络，连续两次丢包的概率只有万分之一，这个概率实在是太小了，忽略它比解决它更具性价比。

因此，TIME_WAIT 和 FIN_WAIT2 状态的最大时长都是 2 MSL，由于在 Linux 系统中，MSL 的值固定为 30 秒，所以它们都是 60 秒。

虽然 TIME_WAIT 状态有存在的必要，但它毕竟会消耗系统资源。**如果发起连接一方的 TIME_WAIT 状态过多，占满了所有端口资源，则会导致无法创建新连接。**

- **客户端受端口资源限制**：如果客户端 TIME_WAIT 过多，就会导致端口资源被占用，因为端口就65536个，被占满就会导致无法创建新的连接；
- **服务端受系统资源限制**：由于一个四元组表示TCP连接，理论上服务端可以建立很多连接，服务端确实只监听一个端口，但是会把连接扔给处理线程，所以理论上监听的端口可以继续监听。但是线程池处理不了那么多一直不断的连接了。所以当服务端出现大量 TIME_WAIT 时，系统资源被占满时，会导致处理不过来新的连接；

另外，**Linux 提供了 tcp_max_tw_buckets 参数，当 TIME_WAIT 的连接数量超过该参数时，新关闭的连接就不再经历 TIME_WAIT 而直接关闭：**

```
# 调整 timewait 最大个数
$ echo 5000 > /proc/sys/net/ipv4/tcp_max_tw_buckets
```

当服务器的并发连接增多时，相应地，同时处于 TIME_WAIT 状态的连接数量也会变多，此时就应当调大 `tcp_max_tw_buckets` 参数，减少不同连接间数据错乱的概率。

tcp_max_tw_buckets 也不是越大越好，毕竟内存和端口都是有限的。

有一种方式可以在建立新连接时，复用处于 TIME_WAIT 状态的连接，那就是打开 tcp_tw_reuse 参数。但是需要注意，该参数是只用于客户端（建立连接的发起方），因为是在调用 connect() 时起作用的，而对于服务端（被动连接方）是没有用的。

```
# 打开 tcp_tw_reuse 功能
$ echo 1 > /proc/sys/net/ipv4/tcp_tw_reuse
```

tcp_tw_reuse 从协议角度理解是安全可控的，可以复用处于 TIME_WAIT 的端口为新的连接所用。

什么是协议角度理解的安全可控呢？主要有两点：

- 只适用于连接发起方，也就是 C/S 模型中的客户端；
- 对应的 TIME_WAIT 状态的连接创建时间超过 1 秒才可以被复用。

使用这个选项，还有一个前提，需要打开对 TCP 时间戳的支持（对方也要打开 ）：

```
# 打开时间戳功能，默认值为 1
$ echo 1 > /proc/sys/net/ipv4/tcp_timestamps
```

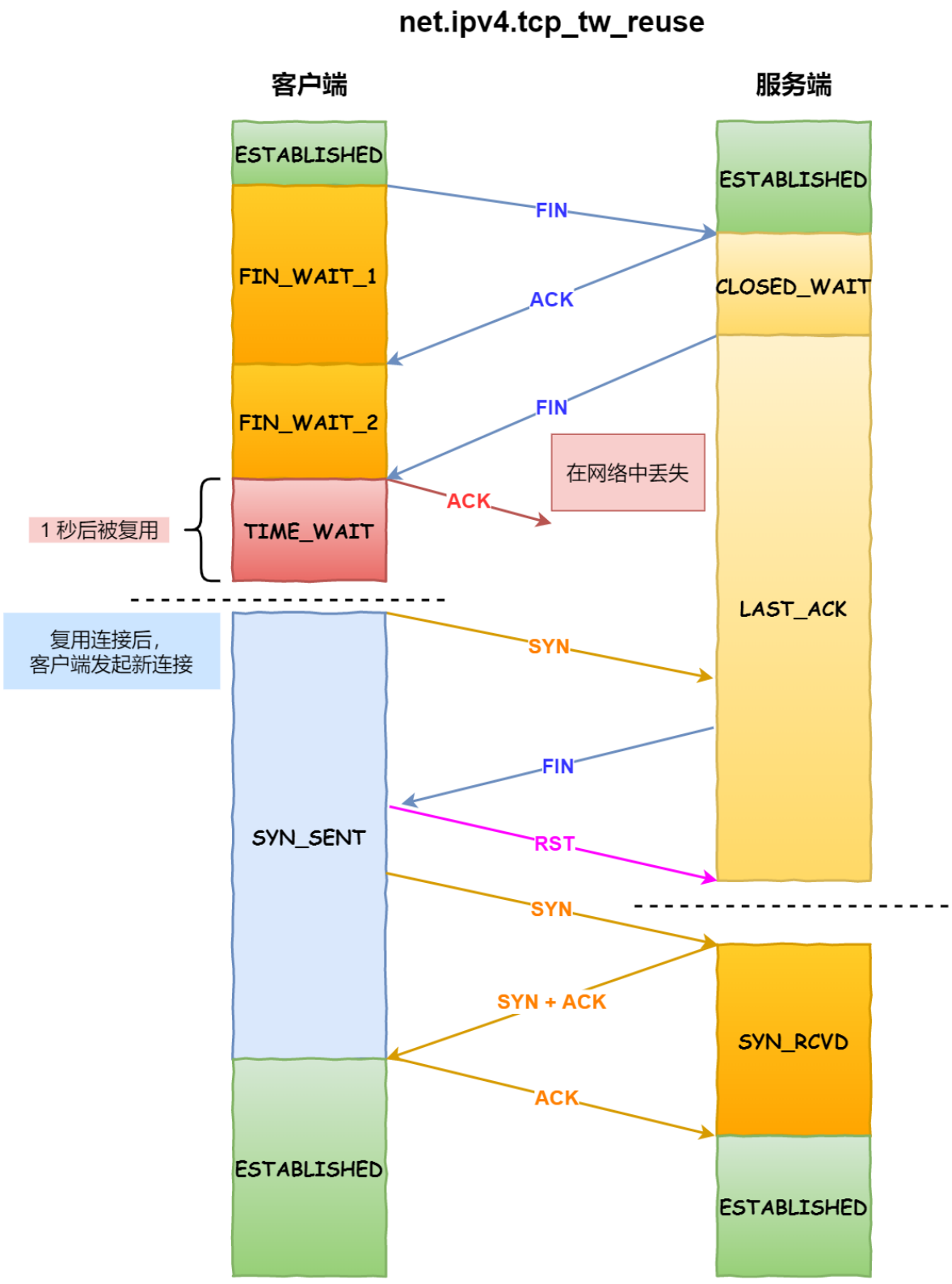
由于引入了时间戳，它能带来了些好处：

- 我们在前面提到的 2MSL 问题就不复存在了，因为重复的数据包会因为时间戳过期被自然丢弃；
- 同时，它还可以防止序列号绕回，也是因为重复的数据包会由于时间戳过期被自然丢弃；

时间戳是在 TCP 的选项字段里定义的，开启了时间戳功能，在 TCP 报文传输的时候会带上发送报文的时间戳。

TCP option 字段			
类型	总长度（字节）	数据	描述
0	-	-	选项列表末尾标识
1	-	-	没有意义，用于 32 位对齐使用
2	4	MSS 值	三次握手时，发送端告知可以接收的最大报文段大小
3	3	窗口移位	指明最大窗口扩展后的大小
4	2	-	表明支持 SACK 选择性确认功能
5	可变	确认报文段	选择性确认窗口中间的报文段
8	10	Timestamps 时间戳	用于更精准的计算 RTT，以及解决序列号绕回的问题
14	3	校验和算法	双方认可后，可使用新的校验和算法
15	可变	校验和	当 16 位标准校验和放不下时，放置在这里
34	可变	FOC	TFO中Cookie

我们来看看开启了 tcp_tw_reuse 功能，如果四次挥手中的最后一次 ACK 在网络中丢失了，会发生什么？



上图的流程：

- 四次挥手中的最后一次 ACK 在网络中丢失了，服务端一直处于 LAST_ACK 状态；
- 客户端由于开启了 tcp_tw_reuse 功能，客户端再次发起新连接的时候，会复用超过 1 秒后的 time_wait 状态的连接。**但客户端新发的 SYN 包会被忽略（由于时间戳），因为服务端比较了客户端的上一个报文与 SYN 报文的时间戳，过期的报文就会被服务端丢弃**；
- 服务端 FIN 报文迟迟没有收到四次挥手的最后一次 ACK，于是超时重发了 FIN 报文给客户端；
- **处于 SYN_SENT 状态的客户端，由于收到了 FIN 报文，则会回 RST 给服务端，于是服务端就离开了 LAST_ACK 状态；**

- **最初的客户端 SYN 报文超时重发了（1 秒钟后）**，此时就与服务端能正确的三次握手了。

所以大家都会说开启了 tcp_tw_reuse，可以在复用了 time_wait 状态的 1 秒过后成功建立连接，这 1 秒主要是花费在 SYN 包重传。

另外，老版本的 Linux 还提供了 tcp_tw_recycle 参数，但是当开启了它，就有两个坑：

- **Linux 会加快客户端和服务端 TIME_WAIT 状态的时间**，也就是它会使得 TIME_WAIT 状态会小于 60 秒，很容易导致数据错乱；
- 另外，**Linux 会丢弃所有来自远端时间戳小于上次记录的时间戳（由同一个远端发送的）的任何数据包**。就是说要使用该选项，则必须保证数据包的时间戳是单调递增的。那么，问题在于，此处的时间戳并不是我们通常意义上面的绝对时间，而是一个相对时间。很多情况下，我们是没法保证时间戳单调递增的，比如使用了 NAT、LVS 等情况；

所以，不建议设置为 1 ，在 Linux 4.12 版本后，Linux 内核直接取消了这一参数，建议关闭它：

```
# 关闭 tcp_tw_recycle 功能
$ echo 0 > /proc/sys/net/ipv4/tcp_tw_recycle
```

另外，我们可以在程序中设置 socket 选项，来设置调用 close 关闭连接行为。

```
struct linger so_linger;
so_linger.l_onoff = 1;
so_linger.l_linger = 0;
setsockopt(s, SOL_SOCKET, SO_LINGER, &so_linger,sizeof(so_linger));
```

如果 `l_onoff` 为非 0， 且 `l_linger` 值为 0，**那么调用 close 后，会立该发送一个 RST 标志给对端，该 TCP 连接将跳过四次挥手，也就跳过了 TIME_WAIT 状态，直接关闭。**

但这为跨越 TIME_WAIT 状态提供了一个可能，不过是一个非常危险的行为，不值得提倡。

被动方的优化

当被动方收到 FIN 报文时，内核会自动回复 ACK，同时连接处于 CLOSE_WAIT 状态，顾名思义，它表示等待应用进程调用 close 函数关闭连接。

内核没有权利替代进程去关闭连接，因为如果主动方是通过 shutdown 关闭连接，那么它就是想在半关闭连接上接收数据或发送数据。因此，Linux 并没有限制 CLOSE_WAIT 状态的持续时间。

当然，大多数应用程序并不使用 shutdown 函数关闭连接。所以，**当你用 netstat 命令发现大量 CLOSE_WAIT 状态。就需要排查你的应用程序，因为可能因为应用程序出现了 Bug，read 函数返回 0 时，没有调用 close 函数。**

处于 CLOSE_WAIT 状态时，调用了 close 函数，内核就会发出 FIN 报文关闭发送通道，同时连接进入 LAST_ACK 状态，等待主动方返回 ACK 来确认连接关闭。

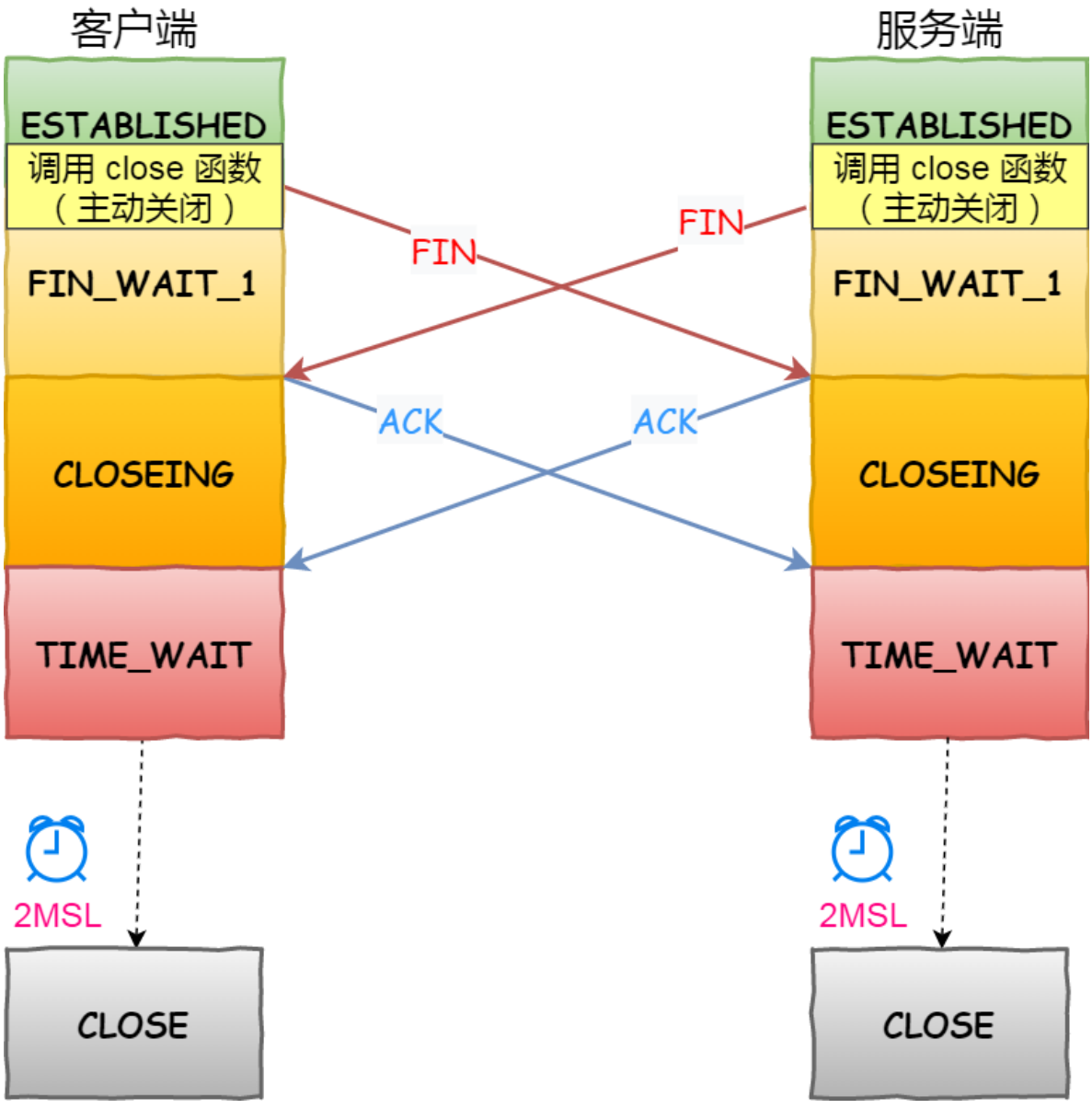
如果迟迟收不到这个 ACK，内核就会重发 FIN 报文，重发次数仍然由 tcp_orphan_retries 参数控制，这与主动方重发 FIN 报文的优化策略一致。

还有一点我们需要注意的，**如果被动方迅速调用 close 函数，那么被动方的 ACK 和 FIN 有可能在一个报文中发送，这样看起来，四次挥手会变成三次挥手，这只是一种特殊情况，不用在意。**

> 如果连接双方同时关闭连接，会怎么样?

由于 TCP 是双全工的协议，所以是会出现两方同时关闭连接的现象，也就是同时发送了 FIN 报文。

此时，上面介绍的优化策略仍然适用。两方发送 FIN 报文时，都认为自己是主动方，所以都进入了 FIN_WAIT1 状态，FIN 报文的重发次数仍由 tcp_orphan_retries 参数控制。



接下来，**双方在等待 ACK 报文的过程中，都等来了 FIN 报文。这是一种新情况，所以连接会进入一种叫做 CLOSING 的新状态，它替代了 FIN_WAIT2 状态**。接着，双方内核回复 ACK 确认对方发送通道的关闭后，进入 TIME_WAIT 状态，等待 2MSL 的时间后，连接自动关闭。

小结

针对 TCP 四次挥手的优化，我们需要根据主动方和被动方四次挥手状态变化来调整系统 TCP 内核参数。

优化四次挥手的策略	
策略	**TCP 内核参数**
调整 FIN 报文重传次数	tcp_orphan_retries
调整 FIN_WAIT2 状态的时间 (只适用 close 函数关闭的连接)	tcp_fin_timeout
调整孤儿连接的上限个数 (只适用 close 函数关闭的连接)	tcp_max_orphans
调整 time_wait 状态的上限个数	tcp_max_tw_buckets
复用 time_wait 状态的连接 (只适用客户端)	tcp_tw_reuse 、tcp_timestamps

> 主动方的优化

主动发起 FIN 报文断开连接的一方，如果迟迟没收到对方的 ACK 回复，则会重传 FIN 报文，重传的次数由 `tcp_orphan_retries` 参数决定。

当主动方收到 ACK 报文后，连接就进入 FIN_WAIT2 状态，根据关闭的方式不同，优化的方式也不同：

- 如果这是 close 函数关闭的连接，那么它就是孤儿连接。如果 `tcp_fin_timeout` 秒内没有收到对方的 FIN 报文，连接就直接关闭。同时，为了应对孤儿连接占用太多的资源， `tcp_max_orphans` 定义了最大孤儿连接的数量，超过时连接就会直接释放。
- 反之是 shutdown 函数关闭的连接，则不受此参数限制；

当主动方接收到 FIN 报文，并返回 ACK 后，主动方的连接进入 TIME_WAIT 状态。这一状态会持续 1 分钟，为了防止 TIME_WAIT 状态占用太多的资源， `tcp_max_tw_buckets` 定义了最大数量，超过时连接也会直接释放。

当 TIME_WAIT 状态过多时，还可以通过设置 `tcp_tw_reuse` 和 `tcp_timestamps` 为 1 ，将 TIME_WAIT 状态的端口复用于作为客户端的新连接，注意该参数只适用于客户端。

> 被动方的优化

被动关闭的连接方应对非常简单，它在回复 ACK 后就进入了 CLOSE_WAIT 状态，等待进程调用 close 函数关闭连接。因此，出现大量 CLOSE_WAIT 状态的连接时，应当从应用程序中找问题。

当被动方发送 FIN 报文后，连接就进入 LAST_ACK 状态，在未等到 ACK 时，会在 `tcp_orphan_retries` 参数的控制下重发 FIN 报文。

---

03 TCP 传输数据的性能提升

在前面介绍的是三次握手和四次挥手的优化策略，接下来主要介绍的是 TCP 传输数据时的优化策略。

TCP 连接是由内核维护的，内核会为每个连接建立内存缓冲区：

- 如果连接的内存配置过小，就无法充分使用网络带宽，TCP 传输效率就会降低；
- 如果连接的内存配置过大，很容易把服务器资源耗尽，这样就会导致新连接无法建立；

因此，我们必须理解 Linux 下 TCP 内存的用途，才能正确地配置内存大小。

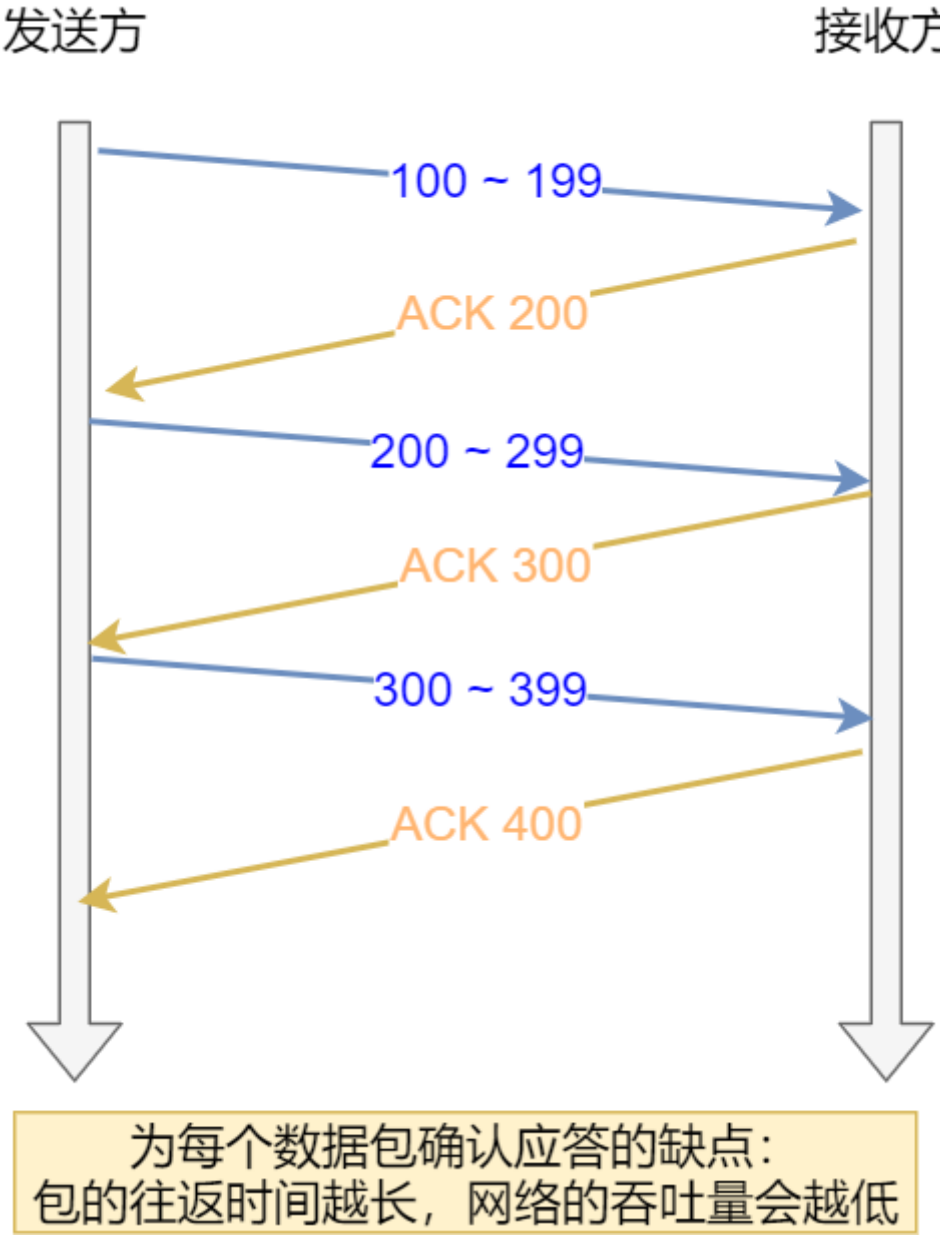
滑动窗口是如何影响传输速度的?

TCP 会保证每一个报文都能够抵达对方，它的机制是这样：报文发出去后，必须接收到对方返回的确认报文 ACK，如果迟迟未收到，就会超时重发该报文，直到收到对方的 ACK 为止。

所以，TCP 报文发出去后，并不会立马从内存中删除，因为重传时还需要用到它。

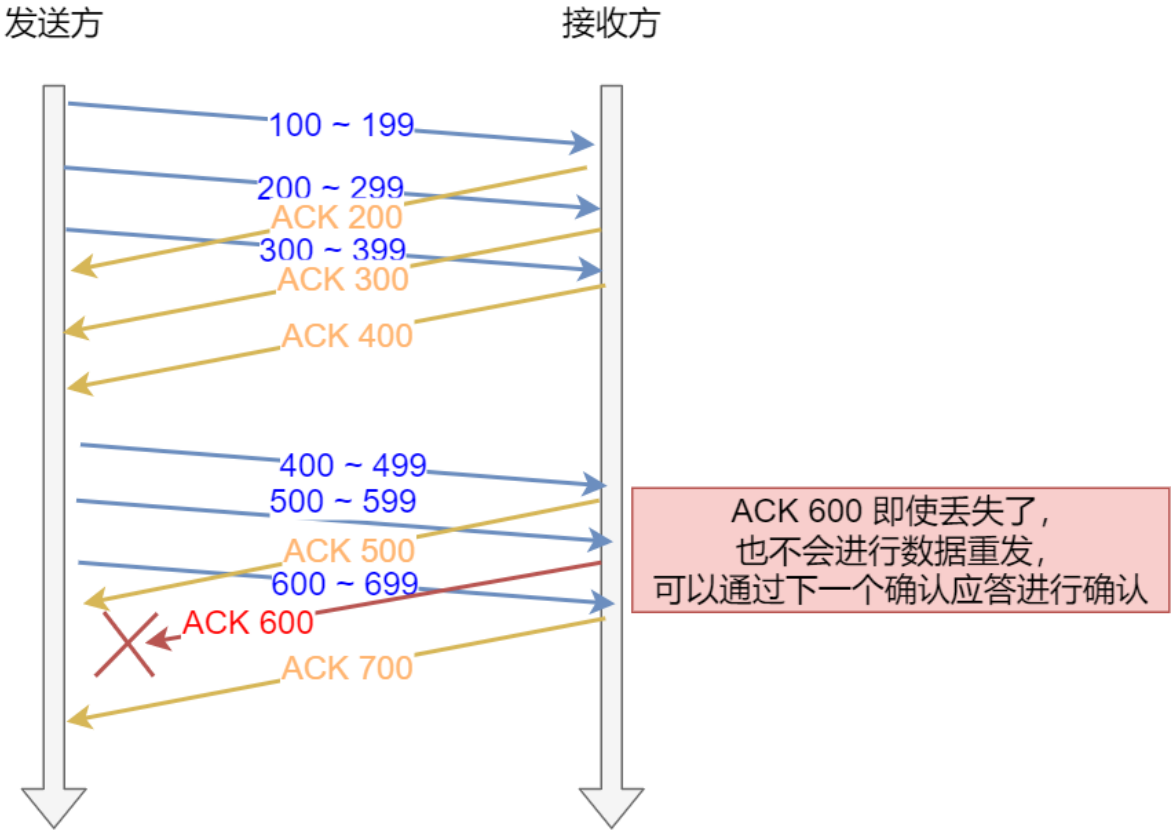
由于 TCP 是内核维护的，所以报文存放在内核缓冲区。如果连接非常多，我们可以通过 free 命令观察到 `buff/cache` 内存是会增大。

如果 TCP 是每发送一个数据，都要进行一次确认应答。当上一个数据包收到了应答了， 再发送下一个。这个模式就有点像我和你面对面聊天，你一句我一句，但这种方式的缺点是效率比较低的。



所以，这样的传输方式有一个缺点：数据包的**往返时间越长，通信的效率就越低**。

要解决这一问题不难，并行批量发送报文，再批量确认报文即可。



然而，这引出了另一个问题，发送方可以随心所欲的发送报文吗？**当然这不现实，我们还得考虑接收方的处理能力。**

当接收方硬件不如发送方，或者系统繁忙、资源紧张时，是无法瞬间处理这么多报文的。于是，这些报文只能被丢掉，使得网络效率非常低。

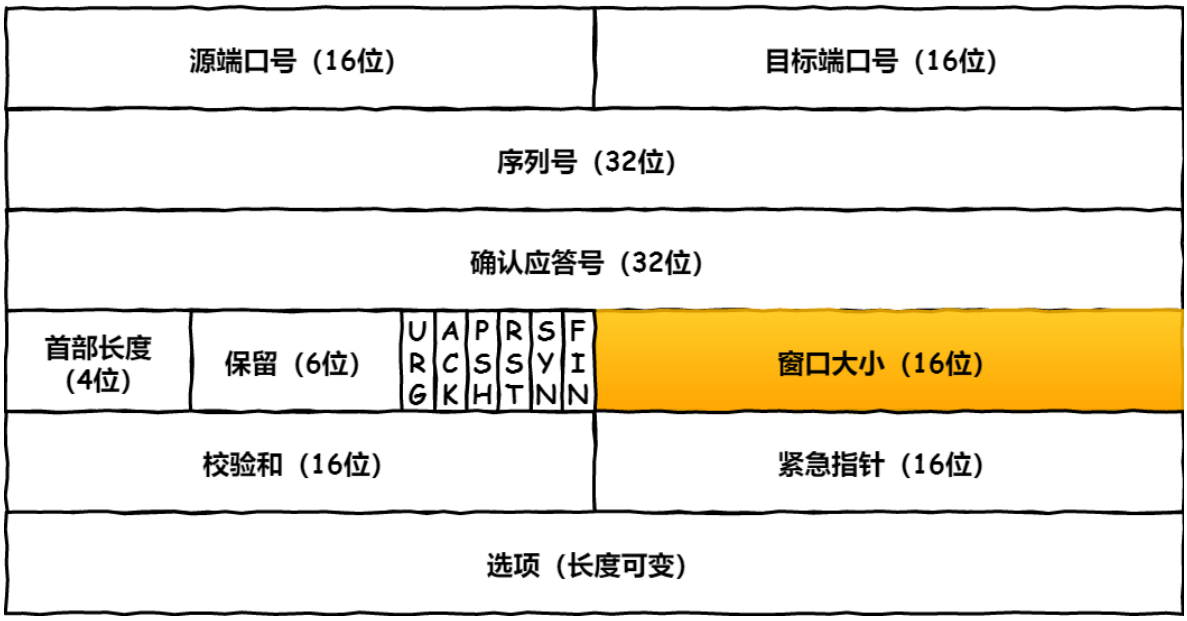
为了解决这种现象发生，TCP 提供一种机制可以让「发送方」根据「接收方」的实际接收能力控制发送的数据量，这就是滑动窗口的由来。

接收方根据它的缓冲区，可以计算出后续能够接收多少字节的报文，这个数字叫做接收窗口。当内核接收到报文时，必须用缓冲区存放它们，这样剩余缓冲区空间变小，接收窗口也就变小了；当进程调用 read 函数后，数据被读入了用户空间，内核缓冲区就被清空，这意味着主机可以接收更多的报文，接收窗口就会变大。

因此，接收窗口并不是恒定不变的，接收方会把当前可接收的大小放在 TCP 报文头部中的**窗口字段**，这样就可以起到窗口大小通知的作用。

发送方的窗口等价于接收方的窗口吗？如果不考虑拥塞控制，发送方的窗口大小「约等于」接收方的窗口大小，因为窗口通知报文在网络传输是存在时延的，所以是约等于的关系。

TCP 头部格式



从上图中可以看到，窗口字段只有 2 个字节，因此它最多能表达 65535 字节大小的窗口，也就是 64KB 大小。

这个窗口大小最大值，在当今高速网络下，很明显是不够用的。所以后续有了扩充窗口的方法：**在 TCP 选项字段定义了窗口扩大因子，用于扩大 TCP 通告窗口，其值大小是 2^14，这样就使 TCP 的窗口大小从 16 位扩大为 30 位（2^16 * 2^ 14 = 2^30），所以此时窗口的最大值可以达到 1GB。**

TCP option 字段			
类型	总长度（字节）	数据	描述
0	-	-	选项列表末尾标识
1	-	-	没有意义，用于 32 位对齐使用
2	4	MSS 值	三次握手时，发送端告知可以接收的最大报文段大小
3	3	窗口移位	指明最大窗口扩展后的大小
4	2	-	表明支持 SACK 选择性确认功能
5	可变	确认报文段	选择性确认窗口中间的报文段
8	10	Timestamps 时间戳	用于更精准的计算 RTT ，以及解决序列号绕回的问题
14	3	校验和算法	双方认可后，可使用新的校验和算法
15	可变	校验和	当 16 位标准校验和放不下时，放置在这里
34	可变	FOC	TFO中Cookie

Linux 中打开这一功能，需要把 tcp_window_scaling 配置设为 1（默认打开）：

```
# 启用窗口扩大因子功能，默认即打开
$ echo 1 > /proc/sys/net/ipv4/tcp_window_scaling
```

要使用窗口扩大选项，通讯双方必须在各自的 SYN 报文中发送这个选项：

- 主动建立连接的一方在 SYN 报文中发送这个选项；
- 而被动建立连接的一方只有在收到带窗口扩大选项的 SYN 报文之后才能发送这个选项。

这样看来，只要进程能及时地调用 read 函数读取数据，并且接收缓冲区配置得足够大，那么接收窗口就可以无限地放大，发送方也就无限地提升发送速度。

这是不可能的，因为网络的传输能力是有限的，当发送方依据发送窗口，发送超过网络处理能力的报文时，路由器会直接丢弃这些报文。因此，缓冲区的内存并不是越大越好。

如何确定最大传输速度?

在前面我们知道了 TCP 的传输速度，受制于发送窗口与接收窗口，以及网络设备传输能力。其中，窗口大小由内核缓冲区大小决定。如果缓冲区与网络传输能力匹配，那么缓冲区的利用率就达到了最大化。

问题来了，如何计算网络的传输能力呢?

相信大家都知道网络是有「带宽」限制的，带宽描述的是网络传输能力，它与内核缓冲区的计量单位不同:

- 带宽是单位时间内的流量，表达是「速度」，比如常见的带宽 100 MB/s；
- 缓冲区单位是字节，当网络速度乘以时间才能得到字节数；

这里需要说一个概念，就是带宽时延积，它决定网络中飞行报文的大小，它的计算方式：

比如最大带宽是 100 MB/s，网络时延（RTT）是 10ms 时，意味着客户端到服务端的网络一共可以存放 100MB/s * 0.01s = 1MB 的字节。

这个 1MB 是带宽和时延的乘积，所以它就叫「带宽时延积」（缩写为 BDP，Bandwidth Delay Product）。同时，这 1MB 也表示「飞行中」的 TCP 报文大小，它们就在网络线路、路由器等网络设备上。如果飞行报文超过了 1 MB，就会导致网络过载，容易丢包。

由于发送缓冲区大小决定了发送窗口的上限，而发送窗口又决定了「已发送未确认」的飞行报文的上限。因此，发送缓冲区不能超过「带宽时延积」。

发送缓冲区与带宽时延积的关系：

- 如果发送缓冲区「超过」带宽时延积，超出的部分就没办法有效的网络传输，同时导致网络过载，容易丢包；
- 如果发送缓冲区「小于」带宽时延积，就不能很好的发挥出网络的传输效率。

所以，发送缓冲区的大小最好是往带宽时延积靠近。

怎样调整缓冲区大小?

在 Linux 中发送缓冲区和接收缓冲都是可以用参数调节的。设置完后，Linux 会根据你设置的缓冲区进行**动态调节**。

> 调节发送缓冲区范围

先来看看发送缓冲区，它的范围通过 tcp_wmem 参数配置；

```
# 调整 TCP 发送缓冲区范围
$ echo "4096 16384 4194304" > /proc/sys/net/ipv4/tcp_wmem
```

上面三个数字单位都是字节，它们分别表示：

- 第一个数值是动态范围的最小值，4096 byte = 4K；
- 第二个数值是初始默认值，87380 byte ≈ 86K；
- 第三个数值是动态范围的最大值，4194304 byte = 4096K（4M）；

发送缓冲区是自行调节的，当发送方发送的数据被确认后，并且没有新的数据要发送，就会把发送缓冲区的内存释放掉。

> 调节接收缓冲区范围

而接收缓冲区的调整就比较复杂一些，先来看看设置接收缓冲区范围的 tcp_rmem 参数：

```
# 调整 TCP 接收缓冲区范围
$ echo "4096 87380 6291456" > /proc/sys/net/ipv4/tcp_rmem
```

上面三个数字单位都是字节，它们分别表示：

- 第一个数值是动态范围的最小值，表示即使在内存压力下也可以保证的最小接收缓冲区大小，4096 byte = 4K；
- 第二个数值是初始默认值，87380 byte ≈ 86K；

- 第三个数值是动态范围的最大值，6291456 byte = 6144K（6M）；

接收缓冲区可以根据系统空闲内存的大小来调节接收窗口：

- 如果系统的空闲内存很多，就可以自动把缓冲区增大一些，这样传给对方的接收窗口也会变大，因而提升发送方发送的传输数据数量；
- 反之，如果系统的内存很紧张，就会减少缓冲区，这虽然会降低传输效率，可以保证更多的并发连接正常工作；

发送缓冲区的调节功能是自动开启的，**而接收缓冲区则需要配置 tcp_moderate_rcvbuf 为 1 来开启调节功能**：

```
# 启动 TCP 接收缓冲区自动调节功能
$ echo 1 > /proc/sys/net/ipv4/tcp_moderate_rcvbuf
```

> 调节 TCP 内存范围

接收缓冲区调节时，怎么知道当前内存是否紧张或充分呢？这是通过 tcp_mem 配置完成的：

```
# 调整 TCP 内存范围
$ echo "88560 118080 177120" > /proc/sys/net/ipv4/tcp_mem
```

上面三个数字单位不是字节，而是「页面大小」，1 页表示 4KB，它们分别表示：

- 当 TCP 内存小于第 1 个值时，不需要进行自动调节；
- 在第 1 和第 2 个值之间时，内核开始调节接收缓冲区的大小；
- 大于第 3 个值时，内核不再为 TCP 分配新内存，此时新连接是无法建立的；

一般情况下这些值是在系统启动时根据系统内存数量计算得到的。根据当前 tcp_mem 最大内存页面数是 177120，当内存为 (177120 * 4) / 1024K ≈ 692M 时，系统将无法为新的 TCP 连接分配内存，即 TCP 连接将被拒绝。

> 根据实际场景调节的策略

在高并发服务器中，为了兼顾网速与大量的并发连接，**我们应当保证缓冲区的动态调整的最大值达到带宽时延积，而最小值保持默认的 4K 不变即可。而对于内存紧张的服务而言，调低默认值是提高并发的有效手段。**

同时，如果这是网络 IO 型服务器，那么，**调大 tcp_mem 的上限可以让 TCP 连接使用更多的系统内存，这有利于提升并发能力**。需要注意的是，tcp_wmem 和 tcp_rmem 的单位是字节，而 tcp_mem 的单位是页面大小。而且，**千万不要在 socket 上直接设置 SO_SNDBUF 或者 SO_RCVBUF，这样会关闭缓冲区的动态调整功能。**

小结

本节针对 TCP 优化数据传输的方式，做了一些介绍。

数据传输的优化策略	
策略	**TCP 内核参数**
扩大窗口大小	tcp_window_scaling
调整发送缓冲区范围	tcp_wmem
调整接收缓冲区范围	tcp_rmem
打开接收缓冲区动态调节	tcp_moderate_rcvbuf
调整内存范围	tcp_mem

TCP 可靠性是通过 ACK 确认报文实现的，又依赖滑动窗口提升了发送速度也兼顾了接收方的处理能力。

可是，默认的滑动窗口最大值只有 64 KB，不满足当今的高速网络的要求，要想提升发送速度必须提升滑动窗口的上限，在 Linux 下是通过设置 `tcp_window_scaling` 为 1 做到的，此时最大值可高达 1GB。

滑动窗口定义了网络中飞行报文的最大字节数，当它超过带宽时延积时，网络过载，就会发生丢包。而当它小于带宽时延积时，就无法充分利用网络带宽。因此，滑动窗口的设置，必须参考带宽时延积。

内核缓冲区决定了滑动窗口的上限，缓冲区可分为：发送缓冲区 tcp_wmem 和接收缓冲区 tcp_rmem。

Linux 会对缓冲区动态调节，我们应该把缓冲区的上限设置为带宽时延积。发送缓冲区的调节功能是自动打开的，而接收缓冲区需要把 tcp_moderate_rcvbuf 设置为 1 来开启。其中，调节的依据是 TCP 内存范围 tcp_mem。

但需要注意的是，如果程序中的 socket 设置 SO_SNDBUF 和 SO_RCVBUF，则会关闭缓冲区的动态整功能，所以不建议在程序设置它俩，而是交给内核自动调整比较好。

有效配置这些参数后，既能够最大程度地保持并发性，也能让资源充裕时连接传输速度达到最大值。

巨人的肩膀

[1] 系统性能调优必知必会.陶辉.极客时间.

[2] 网络编程实战专栏.盛延敏.极客时间.

[3] http://www.blogjava.net/yongboy/archive/2013/04/11/397677.html

[4] http://blog.itpub.net/31559359/viewspace-2284113/

[5] https://blog.51cto.com/professor/1909022

[6] https://vincent.bernat.ch/en/blog/2014-tcp-time-wait-state-linux

---

唠嗑唠嗑

跟大家说个沉痛的事情。

我想大部分小伙伴都发现了，最近公众号改版，**订阅号里的信息流不再是以时间顺序了，而是以推荐算法方式显示顺序。**

这对小林这种「**周更**」的作者，真的一次重重打击，非常的不友好。

因为长时间没发文，公众号可能会把推荐的权重降低，这就会导致很多读者，会收不到我的「最新」的推文，如此下去，那小林文章不就无人问津了？（抱头痛哭 ...）

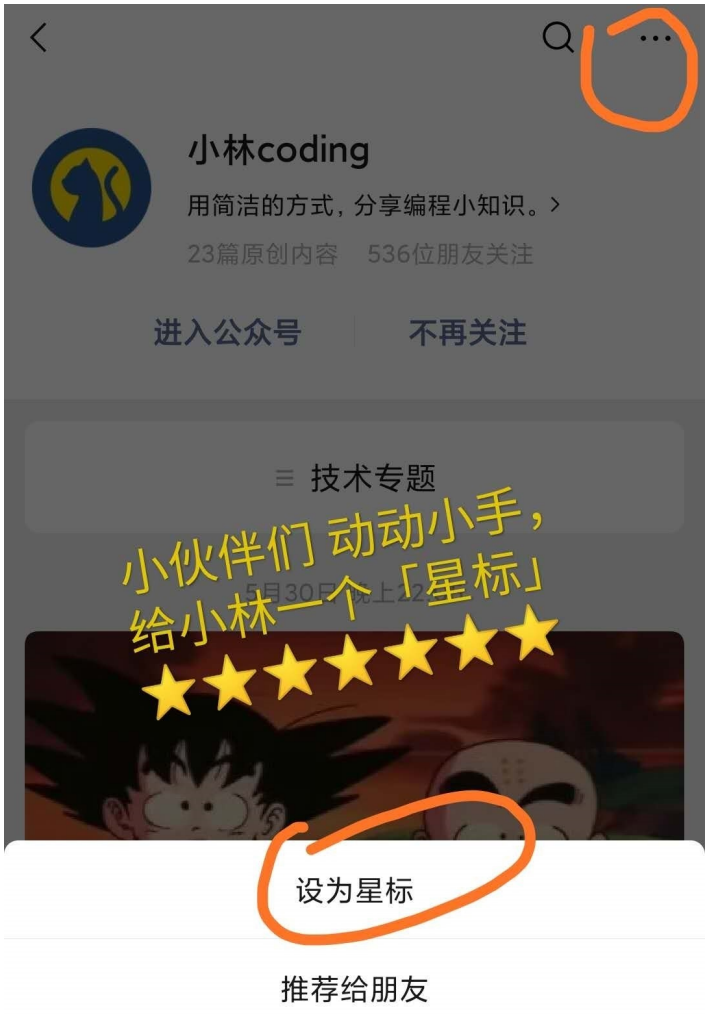
另外，小林更文时间长的原因，不是因为偷懒。

而是为了把知识点「写的更清楚，画的更清晰」，所以这必然会花费更多更长的时间。

如果你认可和喜欢小林的文章，不想错过文章的第一时间推送，可以动动你的小手手，给小林公众号一个「**星标**」。

平时没事，就让「小林coding」静静地躺在你的订阅号底部，但是你要知道它在这其间并非无所事事，而是在努力地准备着更好的内容，等准备好了，它自然会「蹦出」在你面前。

小林是专为大家图解的工具人，Goodbye，我们下次见！



读者问答

> 读者问：“小林，请教个问题，somaxconn和backlog是不是都是指的是accept队列？然后somaxconn是内核参数，backlog是通过系统调用间隔地修改somaxconn，比如Linux中listen()函数？”

两者取最小值才是 accpet 队列。

> 读者问：“小林，还有个问题要请教下，“如果 accept 队列满了，那么 server 扔掉 client 发过来的 ack”，也就是说该TCP连接还是位于半连接队列中，没有丢弃吗？”

1. 当 accept 队列满了，后续新进来的syn包都会被丢失
2. 我文章的突发流量例子是，那个连接进来的时候 accept 队列还没满，但是在第三次握手的时候，accept 队列突然满了，就会导致 ack 被丢弃，就一直处于半连接队列。

小林的赞赏码

本系列「图解网络」的所有图片都是小林纯手打的，平均每一篇都是万字 + 30 张图，这么做的原因很简单，就是为了大家突破计算机网络的痛点。

如果对你有帮助，可以给小林一个小小的赞赏，金额多少不重要，重要的是你们的小小心意，会助力小林输出更多优质的文章。

微信赞赏码

“希望对你有帮助”

智荣林#🎥 的赞赏码

支付宝赞赏码

支付宝
ALIPAY

生活好 支付宝

智荣(**荣)

打开支付宝[扫一扫]

免费寄送收钱码：拨打95188-6