应用层协议

应用层介绍

应用层并不关心数据在底层（如传输层、网络层）是如何传输的，它只关心**“通信的目的是什么”**和**“数据的具体格式是什么”**。

- **角色定位**：就像一个公司的CEO，他只负责制定战略和目标（例如“我们要开拓欧洲市场”），而具体怎么坐飞机去、走哪条路线、货物怎么运输（TCP/IP的下三层）则交给下属部门去完成。
- **主要任务**：为特定的应用程序提供通信规则和标准。它定义了：
  - **消息格式**：数据应该长什么样。比如HTTP协议定义了请求头和响应头的格式。
  - **交互逻辑**：客户端和服务器之间如何对话。比如“三次握手”建立连接、发送请求、等待响应、关闭连接。
  - **所需服务**：它需要下层（主要是传输层）提供什么样的服务。例如，是需要可靠的TCP连接，还是不需要连接的UDP服务。

2. 主要功能和特点

1. **用户服务的直接提供者**：所有为用户服务的网络功能都在这一层。
2. **基于协议**：应用层的功能通过各种**应用层协议**来实现。每个协议都对应着一类特定的网络应用。
3. **与传输层接口**：应用层协议通常会使用一个**端口号（Port Number）**来与传输层（TCP或UDP）进行对接。端口号就像公司里的分机号，确保数据能被交给正确的应用程序处理。
4. **无处不在**：只要你在使用网络软件，你就在和应用层打交道。

工程流程

4. 如何工作？（以访问网页为例）

1. 你在浏览器输入网址 `www.example.com`。
2. 浏览器首先**调用DNS协议**，向DNS服务器查询 `www.example.com` 对应的IP地址。
3. 得到IP地址后，浏览器**使用HTTP协议**，生成一个HTTP请求消息（内容包括要访问的页面等）。
4. 浏览器将HTTP请求消息交给下层的**TCP协议**，要求建立一个可靠的连接并将数据发送到目标IP地址的**80端口**（HTTP的默认端口）。
5. TCP及其下面的各层负责完成数据的实际传输。
6. 服务器端的Web服务器程序（如Apache, Nginx）一直在**监听80端口**，它收到请求后，处理请求，并同样通过HTTP协议生成一个HTTP响应消息（内含网页的HTML代码）。
7. 响应消息沿网络传回你的电脑，浏览器接收到后，解析HTML代码并渲染出完整的网页给你看。

协议

总结表

协议	全称	主要功能	默认端口	传输层协议
HTTP	超文本传输协议	传输网页内容	80	TCP
HTTPS	安全超文本传输协议	安全地传输网页内容	443	TCP
DNS	域名系统协议	将域名解析为IP地址	53	**UDP**/TCP
DHCP	动态主机配置协议	自动分配IP地址	67/68	**UDP**
FTP	文件传输协议	传输文件	20(数据), 21(控制)	TCP
SMTP	简单邮件传输协议	**发送**邮件	25	TCP
POP3	邮局协议	**下载**邮件到本地	110	TCP
IMAP4	互联网消息访问协议	**在服务器上管理**邮件	143	TCP
SSH	安全外壳协议	加密的远程终端管理	22	TCP
Telnet	远程终端协议	明文的远程终端管理（不安全）	23	TCP

HTTP协议

1 HTTP 协议概念

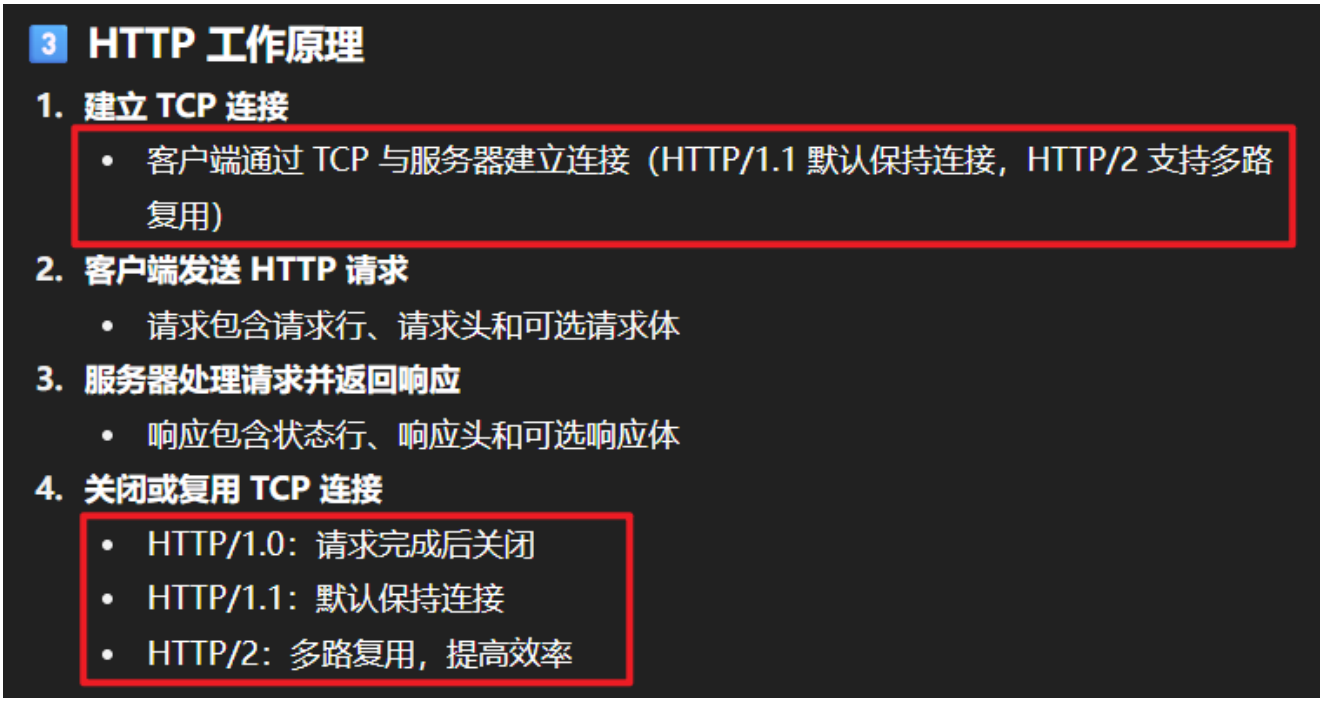
- **全称**：HyperText Transfer Protocol（超文本传输协议）
- **类型**：应用层协议
- **用途**：用于 **浏览器与服务器之间传输超文本（网页内容）**
- **传输方式**：通常依赖 **TCP**（端口 80）
- **特点**：
  1. **无状态**：HTTP 不会记录客户端和服务器的状态，每次请求都是独立的。
  2. **面向请求-响应**：客户端发请求，服务器返回响应。
  3. **灵活**：可以传输文本、图片、视频、JSON 等各种数据。

协议特点

2 HTTP 的特点

特点	说明
无状态	每次请求都是独立的，服务器不会记住前一次的请求（可以通过 Cookie/Session 保存状态）
灵活	可以传输任意类型的数据，靠 MIME 类型描述
简单	请求和响应都是 **ASCII 文本**，易于调试
面向对象	HTTP 把万维网上的资源视为对象，每个对象由 URL 唯一标识
扩展性	可以通过自定义请求头和响应头扩展功能

工作流程



HTTP结构

请求结构

http的请求结构由三个部分组成：

请求行（request line） + 请求头（request headers） + 请求体（request body）

```
请求行（Request Line）
请求头部（Request Headers）
空行
请求体（Request Body，可选）
```

请求行

（request line）

结构如下

2 请求行（Request Line）

请求行是 HTTP 请求的第一行，告诉服务器客户端想做什么操作。

```mathematica
方法 请求URL 协议版本
```

```
POST /api/login HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: application/json
Content-Type: application/json
Content-Length: 48
Authorization: Bearer abc123xyz

{
  "username": "pinkQQx",
  "password": "123456"
}
```

请求行

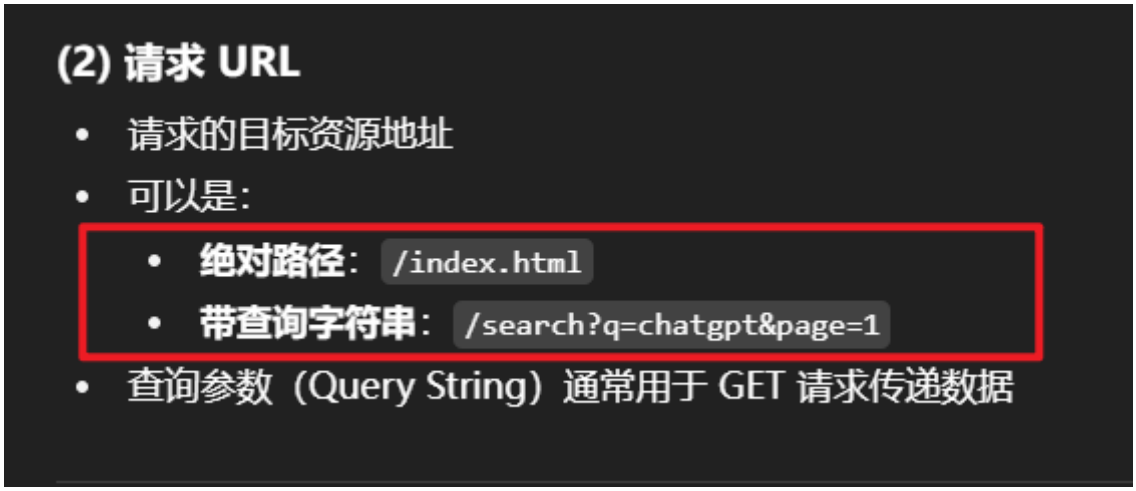
请求方法+URL地址+协议版本

方法 (HTTP Method)

表示客户端对资源的操作类型，常用方法有：

方法	作用	说明
GET	获取资源	请求数据，不改变服务器状态，参数通常放在 URL 中
POST	提交数据	向服务器提交数据（表单、JSON 等），会改变服务器状态
PUT	更新资源	完全替换服务器上的资源
PATCH	部分更新	更新资源的部分内容
DELETE	删除资源	删除服务器上的资源
HEAD	获取资源头信息	和 GET 类似，但只返回响应头，不返回响应体
OPTIONS	查询支持的请求方法	用于跨域请求预检（CORS）

请求URL



版本协议

- **HTTP版本（HTTP Version）**：指定使用的HTTP协议版本，例如：
  - `HTTP/1.1`：最常用的版本，支持持久连接。
  - `HTTP/2`：引入了二进制分帧和多路复用。
  - `HTTP/3`：基于UDP的QUIC协议，优化了性能。

协议版本

HTTP/1.1

优点性能

HTTP/1.1 针对前一版本进行了多项关键改进，显著提高了效率：

- **持久连接（Persistent Connections）：** 这是一个革命性的改进。它允许客户端和服务器在发送和接收多个请求和响应时，**复用同一个 TCP 连接。**这大大减少了每次请求时建立和关闭连接的开销，从而降低了网络延迟。
- **管线化（Pipelining）：** 在持久连接的基础上，客户端可以在收到前一个响应之前，**连续发送多个请求。**这进一步减少了网络往返时间（RTT），提升了通信效率。
- **缓存控制（Caching）：** 协议提供了更精细的缓存机制，允许客户端和服务器通过 `Cache-Control`、`Expires` 等头部字段，更有效地管理浏览器缓存，减少不必要的资源下载。
- **虚拟主机支持：** 强制在请求头中包含 `Host` 字段，使得一台服务器可以托管多个域名，并根据 `Host` 字段将请求路由到正确的网站上。
- **新增请求方法：** 除了 `GET` 和 `POST`，HTTP/1.1 还引入了 `PUT`、`DELETE`、`OPTIONS` 等更多方法，使得协议更具语义化。

缺点

- **队头阻塞（Head-of-Line Blocking）：** 这是 HTTP/1.1 最著名的缺点。尽管管线化允许同时发送多个请求，但服务器必须按照请求的顺序来返回响应。如果队列中的第一个请求由于某种原因（例如，处理时间过长）被阻塞，那么它后面的所有请求，即使已经处理完毕，也必须等待，直到它返回响应。这极大地影响了并行处理的能力。
- **头部冗余：** 在每一个请求中，都会携带大量的重复信息，例如 `User-Agent`、`Cookie`、`Accept` 等。这导致了不必要的网络带宽浪费，尤其是在发送大量小请求时。
- **不支持服务器推送：** 客户端必须主动发起请求才能获取资源。服务器无法在客户端需要之前，主动将资源（例如，相关的 CSS 或 JavaScript 文件）推送给客户端，这增加了加载页面的往返次数。

实现基础

实现基础

HTTP/1.1 协议是基于 **TCP 协议**实现的。具体来说：

1. **连接建立：** 客户端首先通过 **TCP 的三次握手**与服务器建立一个可靠的连接。
2. **数据传输：** 一旦连接建立，客户端便可以发送 HTTP 请求，服务器则返回 HTTP 响应。这些请求和响应都是以**文本**形式封装在 TCP 数据包中进行传输的。
3. **连接管理：** 在 HTTP/1.1 的持久连接模式下，这个 TCP 连接会保持打开，直到一方明确表示关闭。在 HTTP/1.0 中，每次请求-响应后，TCP 连接都会被关闭。

简而言之，HTTP/1.1 依赖于 TCP 提供的可靠、有序的数据流服务，并在此基础上定义了客户端和服务器之间如何交换应用层数据。

HTTP/2.0

特点以及改进

1. 二进制分帧（Binary Framing）

HTTP/2.0 的最大创新在于它不再使用纯文本传输数据，而是将所有请求和响应数据分割成更小的、独立的数据帧（frames），并以二进制格式传输。

- **优点：** 这种格式更紧凑，解析更高效，并且为后续的多路复用、优先级控制等特性提供了基础。

2. 多路复用（Multiplexing）

这是 HTTP/2.0 解决 HTTP/1.1 **“队头阻塞”**问题的关键。

- **工作原理：** 在 HTTP/2.0 中，客户端和服务器之间只需要建立一个 **TCP 连接**。所有的请求和响应都可以通过这个连接同时进行传输，每个请求和响应都拥有唯一的流 ID（Stream ID）。服务器可以交叉地发送这些帧，而客户端能够根据流 ID 重新组装它们。
- **优点：** 彻底解决了 HTTP/1.1 的队头阻塞问题。即使一个请求的处理时间很长，也不会阻塞其他请求的响应。这极大地提升了页面加载速度，尤其是在网页包含大量资源时。

3. 头部压缩（Header Compression）

HTTP/1.1 的另一个缺点是每个请求都会携带大量重复的头部信息，如 `User-Agent`、`Cookie` 等，造成带宽浪费。

- **工作原理：** HTTP/2.0 使用 **HPACK 算法**对头部进行压缩。HPACK 维护一个静态字典和一个动态字典，用于存储重复出现的头部字段。在传输过程中，它只发送字典中字段的索引，而不是完整的头部字符串。
- **优点：** 显著减少了传输的数据量，尤其是在大量请求同时发送时。

4. 服务器推送（Server Push）

这是 HTTP/2.0 引入的一项新功能。在 HTTP/1.1 中，浏览器必须先下载 HTML 页面，然后解析它，再发起对 CSS、JavaScript、图片等资源的请求。

- **工作原理：** 服务器在接收到客户端对 HTML 页面的请求后，可以主动将一些它认为客户端马上会需要的资源（如 CSS 文件或脚本）**“推送”**给客户端的缓存，而无需等待客户端发出请求。
- **优点：** 减少了客户端-服务器之间的往返次数（RTT），进一步加快了页面加载速度。

5. 优先级和依赖（Stream Prioritization）

HTTP/2.0 允许为不同的流（请求）设置优先级。

- **工作原理：** 开发者可以告诉服务器哪些资源更重要（例如，CSS 和 JavaScript 文件），服务器会优先处理和发送这些资源，确保关键内容更快地加载。

缺点

突出缺点

尽管 HTTP/2 带来了巨大的性能提升，但它仍然存在一个核心的缺点：

- **底层的 TCP 队头阻塞：** 尽管 HTTP/2 解决了应用层的队头阻塞问题，但它依然基于 TCP 协议。TCP 是一个按顺序交付的协议，如果在传输过程中某个 TCP 包丢失，即使是位于不同 HTTP/2 流中的数据包，所有后续的数据包都会被阻塞，直到丢失的包被重传成功。这在网络丢包率较高的环境下，仍然会影响性能。

实现基础

实现基础

HTTP/2 协议是基于 **TCP 协议** 实现的。它并没有改变 HTTP 的基本工作模式（即请求-响应模型），而是在传输层和应用层之间增加了一个**二进制分帧层**。

具体来说：

1. **连接建立：** 客户端与服务器之间首先通过 TCP 协议建立一个可靠的连接。
2. **分帧传输：** HTTP/2 将 HTTP 请求和响应的所有数据（包括头部和主体）都拆分成一个个独立的**二进制帧**。
3. **流管理：** 每个请求和响应都被分配一个唯一的**流 ID**。这些帧会乱序地通过同一个 TCP 连接进行传输。
4. **组装数据：** 接收方会根据每个帧的流 ID 将它们重新组装成完整的请求或响应。

简而言之，HTTP/2 在 TCP 提供的可靠连接之上，通过引入一个全新的二进制分帧层，实现了多路复用和头部压缩等高级功能，从而解决了 HTTP/1.1 的大部分性能问题。

HTTP/3.0

它并没有像 HTTP/2 那样在应用层进行优化，而是将核心突破放在了**传输层**。HTTP/3 最大的变革是放弃了已沿用多年的 **TCP 协议**，转而使用全新的 **QUIC 协议**作为底层传输协议，**从而彻底解决了 HTTP/2 遗留的性能瓶颈。**

我们都知道**TCP协议是顺序交付的协议**，**HTTP/2.0做了数据流/多路复用的优化**，依然会因为**TCP协议本身的瓶颈，导致数据阻塞**

特点和改进

基于UDP开发

1. 基于 QUIC 协议

这是 HTTP/3 与前两个版本最本质的区别。QUIC（Quick UDP Internet Connections）协议最初由 Google 开发，是一个基于 **UDP** 协议的传输层协议。

- **优点：**
  - **彻底解决队头阻塞问题：** HTTP/2 虽然解决了应用层的队头阻塞，但底层 TCP 协议的队头阻塞依然存在。QUIC 协议在 UDP 之上实现了自己的多路复用，每个数据流都是独立的，如果一个数据包丢失，只会影响该数据流，而不会阻塞其他数据流，从根本上解决了问题。
  - **更快的连接建立：** QUIC 协议集成了 TLS 加密握手，可以在**1个RTT（往返时间）**甚至 **0 个RTT** 内完成连接建立和加密协商。相比之下，TCP 加 TLS 握手通常需要2到3个RTT，HTTP/3 大大减少了连接建立的延迟。
  - **更好的连接迁移：** QUIC 使用连接 ID 来标识连接，而不是 IP 地址和端口号。这意味着当客户端从 Wi-Fi 切换到移动网络时，只要连接 ID 不变，就不会中断正在进行的传输，这对于移动设备来说是一个巨大的优势。

2. 继承 HTTP/2 的优点

HTTP/3 并没有丢弃 HTTP/2 带来的优秀特性，而是将它们整合到了新的协议栈中：

- **多路复用：** HTTP/3 仍然支持多路复用，但它是在 QUIC 层实现的，解决了 HTTP/2 底层 TCP 带来的队头阻塞问题。
- **头部压缩：** HTTP/3 依然使用头部压缩技术，但它采用了一个新的压缩算法 **QPACK**，相比 HTTP/2 的 HPACK 算法，它更好地适应了 QUIC 协议的异步和乱序特性。
- **服务器推送：** 这一特性在 HTTP/3 中也得到了保留。

对比和小结

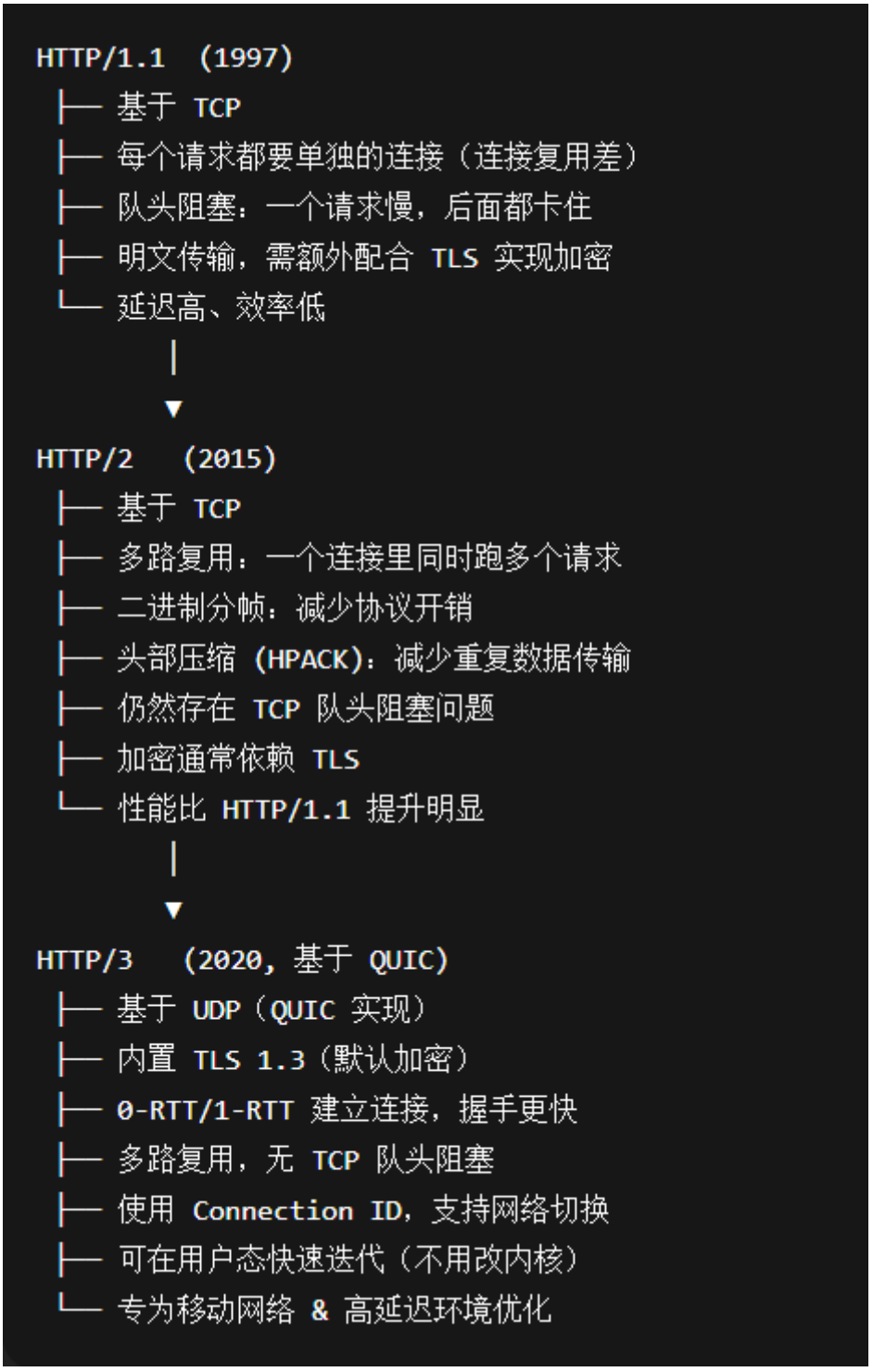
总结

特性	HTTP/1.1	HTTP/2	HTTP/3
底层协议	TCP	TCP	**QUIC (UDP)**
多路复用	不支持	支持（有TCP队头阻塞）	**原生支持（无队头阻塞）**
连接建立	多个TCP连接	单个TCP连接	**单个QUIC连接**
连接时延	2-3个RTT/连接	2-3个RTT	**0-1个RTT**
头部压缩	不支持	支持（HPACK）	**支持（QPACK）**
服务器推送	不支持	支持	支持

导出到 Google 表格

简而言之，HTTP/3 通过更换底层传输协议，从根本上解决了 **HTTP/1.1 和 HTTP/2 所面临的队头阻塞和连接建立延迟问题。**它特别适用于**高延迟和网络不稳定的环境**，如移动互联网，被认为是未来互联网通信的主流协议。

HTTP演化



🔑 **一眼对比：**

- **HTTP/1.1** → 基于 TCP，简单但效率低。
- **HTTP/2** → 依旧 TCP，但优化了多路复用、头部压缩。
- **HTTP/3** → 脱离 TCP，基于 UDP + QUIC，解决 TCP 固有瓶颈。

二进制分帧

这个问题问得很好 👍 ——"二进制分帧"其实是 **HTTP/2 的核心机制**之一。
我给你慢慢拆开讲，避免一下子太抽象。

1️⃣ 背景：为什么要有二进制分帧？

- **HTTP/1.1 的问题**：
  1. **文本协议**：HTTP/1.1 的请求和响应是纯文本（ASCII），体积大，解析效率低。
  2. **队头阻塞**：一个 TCP 连接里，请求必须按顺序返回，前面的慢了，后面的也得等。
  3. **多资源请求效率低**：网页加载需要很多 CSS/JS/图片，HTTP/1.1 通常要开多个 TCP 连接。

➡ 为了解决这些问题，HTTP/2 引入了 **二进制分帧**。

帧是最小的数据传输单位

2️⃣ 什么是二进制分帧？

二进制分帧（Binary Framing Layer）：
HTTP/2 把 **所有请求和响应消息** 拆分成更小的单位，叫 **帧（Frame）**，每个帧都用二进制格式编码。

- **帧（Frame）**：HTTP/2 通信的最小单位。
- **消息（Message）**：由多个帧组成，请求或响应整体就是一条消息。
- **流（Stream）**：双向的帧序列，每个请求/响应就是一个独立的流，可以多个流并行。

帧------(组成)----消息-----(组成)--------流

帧的结构

字段	作用
Length	帧内容长度
Type	帧类型（HEADERS、DATA、SETTINGS 等）
Flags	控制标志位（如 END_STREAM）
Stream Identifier	流 ID，表示属于哪个请求/响应
Frame Payload	实际数据（内容）

帧的类型

常见帧类型：

- **HEADERS**：携带请求/响应头
- **DATA**：传输实际数据（HTML、JSON 等）
- **SETTINGS**：交换连接配置
- **PING**：心跳检测
- **GOAWAY**：关闭连接

帧的优势

4 二进制分帧的优势

1. **高效解析**
   - 机器解析二进制比解析文本快得多。
2. **多路复用（Multiplexing）**
   - 同一个 TCP 连接里可以同时传输多个流（多个请求/响应并行），不会互相阻塞。
3. **头部压缩（HPACK）**
   - 用二进制编码+压缩算法减少重复头部，降低开销。
4. **更强的扩展性**
   - 帧类型可扩展，协议更灵活。

请求信息会被拆**成至少一个 HEADERS 帧（头部字段）+ 0 个或多个 DATA 帧（消息体数据）。**

- **HEADERS 帧** = 用来装 **HTTP 请求/响应头字段** 的"容器"。
- **DATA 帧** = 用来装 **HTTP 请求体或响应体数据** 的"容器"。

HTTP/2 中（分帧）

它会被拆分成 **两个主要的帧**（简化表示）：

这样做的好处

4 为什么要拆成 HEADERS 和 DATA?

因为 HTTP/2 要支持 **多路复用**。

- 在同一个 TCP 连接里，你可能同时请求 `/index.html`、`/style.css`、`/logo.png`。
- 每个请求对应一个 **流（Stream）**，有唯一的流 ID。
- 请求和响应会被拆成帧，然后交错发送：
  - 先发一部分 `/index.html` 的 HEADERS
  - 然后发一点 `/logo.png` 的 DATA
  - 再发 `/style.css` 的 HEADERS
  - .....
- 接收方根据 **帧头里的流 ID**，把属于同一个流的帧重新组装。

这样就避免了 HTTP/1.1 的 **队头阻塞**。

请求信息会被拆**成至少一个 HEADERS 帧（头部字段）+ 0 个或多个 DATA 帧（消息体数据）。**

数据传输结构

1. 帧（Frame）

- HTTP/2 最小的传输单元。
- 每个帧都有 **帧头（Frame Header）**，里面标记了这个帧属于哪个 **流 ID**。
- 常见帧类型：`HEADERS`（请求/响应头）、`DATA`（消息体）、`SETTINGS`、`RST_STREAM` 等。

👉 可以理解为「快递箱子里的小包裹」。

2. 消息（Message）

- 消息 = **一组帧的集合**，用来表达一个完整的 **HTTP 请求 或 响应**。
- **请求消息**：至少包含一个 `HEADERS` 帧（请求头），可能跟着 `DATA` 帧（请求体）。
- **响应消息**：至少包含一个 `HEADERS` 帧（响应头），可能跟着 `DATA` 帧（响应体）。

👉 可以理解为「一个完整的快递（寄件人、收件人、内容都齐全）」。

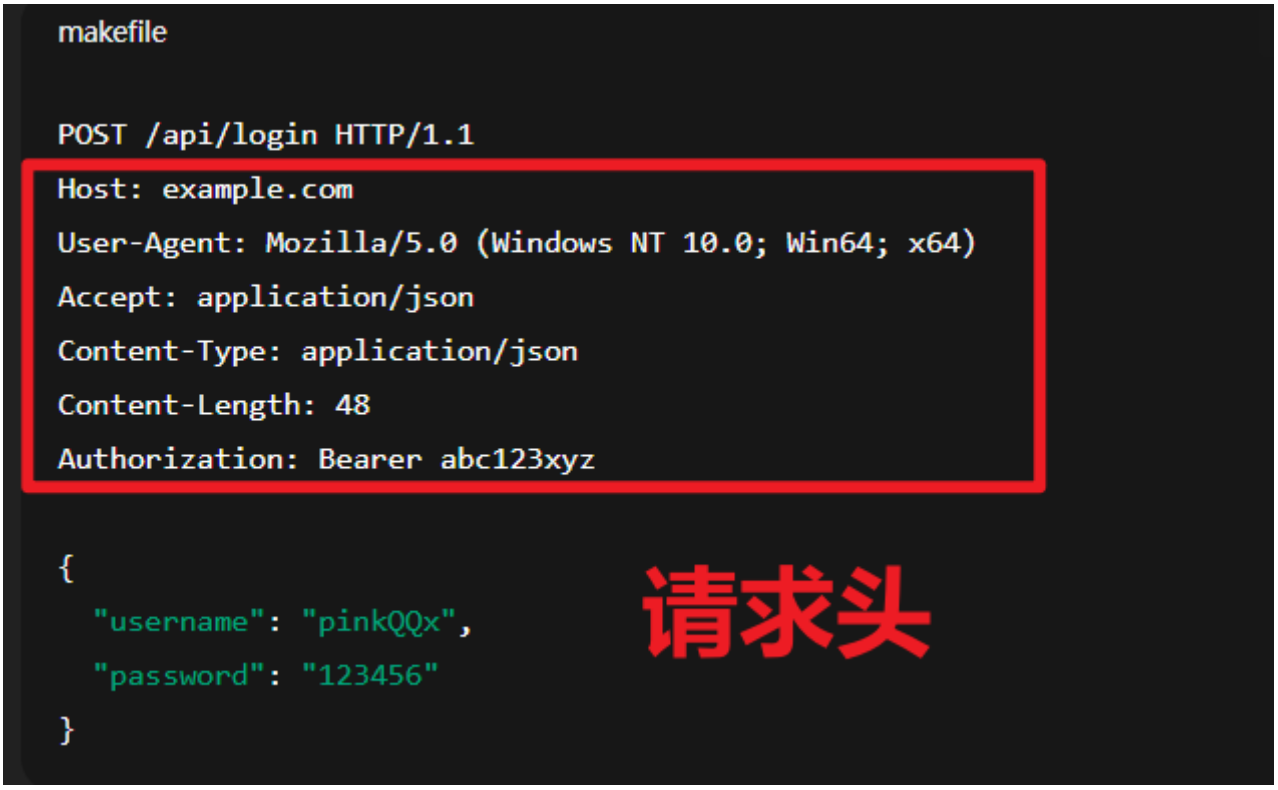
3. 流（Stream）

- 一个 **逻辑上的双向数据通道**，由 **流 ID** 标识。
- 在一个流里，可以传输多个帧（这些帧属于同一个消息或多个消息的不同阶段）。
- 一个流对应 **一次 HTTP 请求-响应的交互**。

👉 可以理解为「快递运输的专属轨道/车厢」。

请求头

(request headers)



请求头的基本格式

每个请求头都是一行文本，格式如下：

```pgsql
Header-Name: Header-Value
```

- **Header-Name**：头字段名，不区分大小写
- **Header-Value**：头字段值，可能是单值，也可能是逗号分隔的多值
- 每个请求头占一行，末尾以 **回车换行（CRLF）** 结束

示例：

```makefile
Host: www.example.com\r\n
User-Agent: Mozilla/5.0\r\n
Accept: text/html,application/xhtml+xml\r\n
```

4 请求头的结束标志

- 请求头部结束必须有 **一个空行**（CRLF）
- 空行后面就是 **请求体**（如果有的话）
- 如果请求没有体（比如 GET 请求），空行就标志请求头的结束

示例：

```pgsql
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0
```

请求头的分类

1 通用头（General Headers）

- **作用**：可用于请求和响应，描述报文整体信息，而不专属于消息体或请求本身。
- **特点**：在 HTTP/2 中仍然存在，但通过二进制帧传输。
- **示例**：
  - `Cache-Control: no-cache`
  - `Connection: keep-alive`
  - `Date: Tue, 22 Aug 2025 11:30:00 GMT`
  - `Via: 1.1 proxy`

解释

- **Cache-Control: no-cache** → 在使用缓存前必须向服务器验证，不直接用旧缓存。
- **Connection: keep-alive** → 保持 TCP 长连接，避免每次请求都重新建立连接。
- **Date: Tue, 22 Aug 2025 11:30:00 GMT** → 报文生成的时间，用于缓存和日志参考。
- **Via: 1.1 proxy** → 报文经过的代理或网关标识，用于调试和防止循环。

2 请求头（Request Headers）

- **作用**：专门出现在请求中，用来描述客户端、请求资源及环境信息。
- **示例**：
  - `Host`：目标主机名（HTTP/1.1 必须）
  - `User-Agent`：客户端软件信息
  - `Accept`：客户端可接收的 MIME 类型
  - `Accept-Encoding`：可接收的压缩算法
  - `Accept-Language`：语言偏好
  - `Authorization`：认证信息
  - `Cookie`：客户端发送给服务器的 Cookie

3 实体头（Entity Headers / Body Headers）

- **作用**：描述请求体或响应体的属性（比如类型、长度、编码方式）。
- **特点**：只有在请求体或响应体存在时才有意义。
- **示例**：
  - `Content-Type`：请求体/响应体类型（JSON、表单、XML、图片等）
  - `Content-Length`：请求体/响应体长度
  - `Content-Encoding`：请求体/响应体压缩方式
  - `Content-Language`：请求体/响应体语言
  - `Content-Disposition`：请求体/响应体如何处理（通常用于文件上传/下载）

🔹 小结

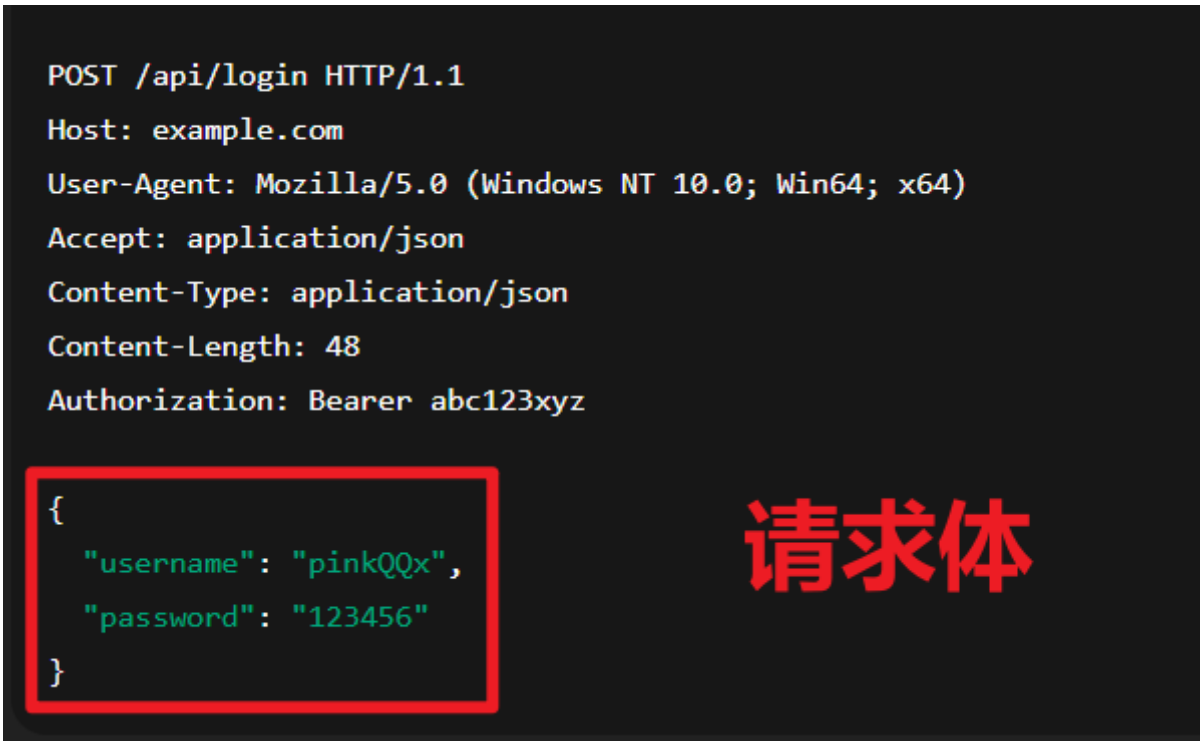
1. **通用头**：可用于请求或响应，描述报文整体
2. **请求头**：只出现在请求里，描述客户端和请求信息
3. **实体头**：描述请求体或响应体的属性

文本总结：

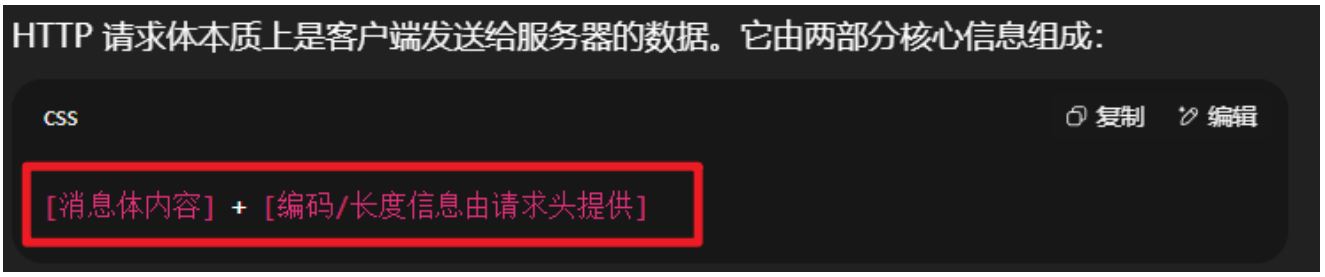
1. **通用头**：不依赖请求方法，通常都会出现，但不是必须。
2. **请求头**：几乎所有请求都会有，Host 是必需的。
3. **实体头**：**只有请求体存在时才出现**，GET/DELETE 一般没有。

请求体

```
请求行（Request Line）
请求头部（Request Headers）
空行（CRLF）
请求体（Request Body，可选）
```



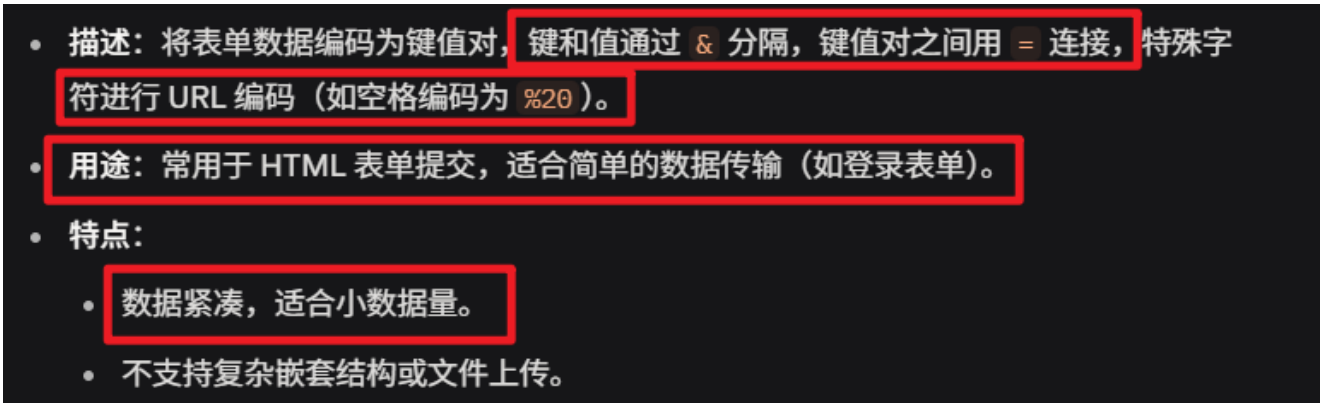
请求体的基本组成



消息体内容（Body Content）

HTTP 请求体的数据类型由 Content-Type 头定义

1.application/x-www-form-urlencoded



示例

```
POST /login HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

username=john&password=12345
```

- 解析：`username=john` 和 `password=12345`，服务器按键值对解析。
- URL 编码示例：如果 `username` 是 `john doe`，则编码为 `username=john%20doe`。

2.application/json

- **描述**：以 JSON（JavaScript Object Notation）格式编码数据，结构化强，支持嵌套对象和数组。
- **用途**：广泛用于 API 请求，适合传输复杂数据结构（如用户 profile、配置信息）。
- **特点**：
  - 易于解析，支持复杂数据。
  - 跨语言兼容（JSON 是通用的数据交换格式）。
  - 数据量较大时比 `urlencoded` 更冗长。

示例

```
POST /api/users HTTP/1.1
Host: example.com
Content-Type: application/json
Content-Length: 44

{"username": "john", "password": "12345"}
```

- 解析：服务器将 JSON 字符串反序列化为对象，访问 `username` 和 `password`。

3.multipart/form-data

- **描述**：将请求体分为多个部分（part），每部分包含键值对或文件，部分之间通过边界（boundary）分隔。常用于文件上传。
- **用途**：适合上传文件（如图片、视频）或混合表单数据和文件。
- **特点**：
  - 支持文件和非文件数据混合传输。
  - 每个部分可以有自己的 `Content-Type`（如 `text/plain` 或 `image/jpeg`）。
  - 数据量较大，格式较复杂。

示例

```
POST /upload HTTP/1.1
Host: example.com
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YW
Content-Length: 314

------WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="username"

john
------WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="file"; filename="example.jpg"
Content-Type: image/jpeg

[二进制文件数据]
------WebKitFormBoundary7MA4YWxkTrZu0gW--
```

解析：服务器按 boundary 分割，提取 username 和文件数据。

4.text/plain

- **描述**：纯文本格式，数据不进行特殊编码，直接发送。
- **用途**：用于简单的文本数据传输，较少用于复杂场景。
- **特点**：
  - 简单，但缺乏结构化，解析不灵活。
  - 不适合复杂数据或文件。

示例

```
POST /submit HTTP/1.1
Host: example.com
Content-Type: text/plain
Content-Length: 11

Hello World
```

- 解析：服务器直接读取文本内容，需自定义解析逻辑。

5.application/octet-stream

- **描述**：表示任意二进制数据流，服务器不假设数据的具体格式。
- **用途**：上传任意格式的文件或数据（如未知类型的二进制数据）。
- **特点**：
  - 通用性强，但缺乏语义，服务器需额外处理。

示例

```
POST /upload HTTP/1.1
Host: example.com
Content-Type: application/octet-stream
Content-Length: 1024

[二进制数据]
```

6.multipart/related

- **描述**：类似 `multipart/form-data`，但用于传输相关联的数据（如 MHTML 格式的网页内容）。
- **用途**：较少用于常规 Web 开发，多见于邮件或复杂资源聚合。
- **特点**：支持嵌套内容，复杂度高。
- **示例**：（简化）

示例

```
Content-Type: multipart/related; boundary=boundary123

--boundary123
Content-Type: text/html

<html>...</html>
--boundary123
Content-Type: image/png

[图像数据]
--boundary123--
```

7.其他类型

- **application/xml 或 text/xml**：传输 XML 格式数据，常见于 SOAP 或老式 API。

```
text                                  收起  自动换行  复制

<user><name>john</name><password>12345</password></user>
```

- **application/pdf, image/jpeg, video/mp4 等**：直接传输特定格式的文件，通常与 `Content-Disposition: attachment` 结合用于下载。
- **multipart/byteranges**：用于范围请求的响应（较少用于请求体）。

数据类型的选择

选择请求体数据类型取决于：

- **数据结构：**
  - 简单键值对：`application/x-www-form-urlencoded`。
  - 复杂结构：`application/json`。
  - 文件上传：`multipart/form-data`。
- **性能：**
  - `urlencoded` 紧凑但不支持文件。
  - `multipart/form-data` 灵活但数据量较大。
- **服务器支持：**确保服务器能解析指定的 `Content-Type`。
- **语义：**JSON 适合 API，`multipart/form-data` 适合表单和文件。

相关请求头

与请求体数据类型相关的头字段：

- **Content-Type**：指定数据类型（如 `application/json`）。
- **Content-Length**：请求体的字节长度（如 `44`）。
- **Content-Encoding**：指定压缩方式（如 `gzip`、`br`），与数据类型无关但影响传输。
- **Content-Disposition**：在 `multipart/form-data` 中定义每个部分的名称和文件名。

完整的请求报文

```
POST /api/login HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
Accept: application/json
Content-Type: application/json
Content-Length: 48
Authorization: Bearer abc123xyz

{
  "username": "pinkQQx",
  "password": "123456"
}
```

响应结构

HTTP 响应报文整体结构

```
状态行（Status Line）
响应头部（Response Headers）
空行（CRLF）
响应体（Response Body，可选）
```

示例

简单示意：

```yaml
HTTP/1.1 200 OK
Date: Tue, 22 Aug 2025 12:00:00 GMT
Server: Apache/2.4.41 (Ubuntu)
Content-Type: text/html
Content-Length: 1256
Connection: keep-alive

<html>
  <body>Hello, world!</body>
</html>
```

响应行

描述服务器的响应状态

（response line）

基本格式

```
HTTP-Version Status-Code Reason-Phrase
```

示例

```
HTTP/1.1 200 OK
```

含义

- 含义：
  - **HTTP-Version**：协议版本（如 HTTP/1.1、HTTP/2）
  - **Status-Code**：状态码（如 200、404、500）
  - **Reason-Phrase**：状态描述文字（如 OK、Not Found）

状态码信息

1. 1xx - 信息性响应（Informational）

- **定义**：临时响应，表示服务器已接收请求，客户端需继续操作或等待。
- **用途**：用于协议协商或通知客户端请求正在处理，较少在常规 Web 应用中使用。
- **常见示例**：
  - **100 Continue**：客户端可继续发送请求体。
  - **101 Switching Protocols**：服务器同意切换协议（如 WebSocket）。
- **特点**：通常无响应体，客户端需进一步行动。

2. 2xx - 成功响应（Success）

- **定义**：请求被服务器成功接收、理解并处理。
- **用途**：表示请求目标达成，返回数据或确认操作完成。
- **常见示例**：
  - **200 OK**：请求成功，返回数据（如网页、JSON）。
  - **201 Created**：新资源创建成功（如 POST 提交）。
  - **204 No Content**：请求成功，无响应体（如 DELETE）。
- **特点**：通常包含响应体（除 `204`），表示操作成功。

3. 3xx - 重定向响应（Redirection）

- **定义**：客户端需采取进一步行动（如跳转到新 URL）以完成请求。
- **用途**：用于资源重定位、缓存控制或临时跳转。
- **常见示例**：
  - **301 Moved Permanently**：资源永久移动，需更新书签。
  - **302 Found**：资源临时移动。
  - **304 Not Modified**：资源未变，客户端使用缓存。
- **特点**：通常包含 `Location` 头，响应体常为空。

4. 4xx - 客户端错误（Client Error）

- 定义：请求有误，服务器无法处理（如语法错误、权限不足）。
- 用途：通知客户端请求问题，需修改请求。
- 常见示例：
  - 400 Bad Request：请求格式错误。
  - 401 Unauthorized：需要认证。
  - 404 Not Found：资源不存在。
  - 429 Too Many Requests：请求超限。
- 特点：通常包含错误描述（如 HTML 或 JSON），客户端需修复。

5. 5xx - 服务器错误（Server Error）

- 定义：服务器在处理请求时发生内部错误。
- 用途：通知客户端服务器端问题，客户端无法直接解决。
- 常见示例：
  - 500 Internal Server Error：服务器通用错误。
  - 502 Bad Gateway：网关收到无效响应。
  - 503 Service Unavailable：服务器暂时不可用（如维护）。
- 特点：通常包含错误信息，需服务器端修复。

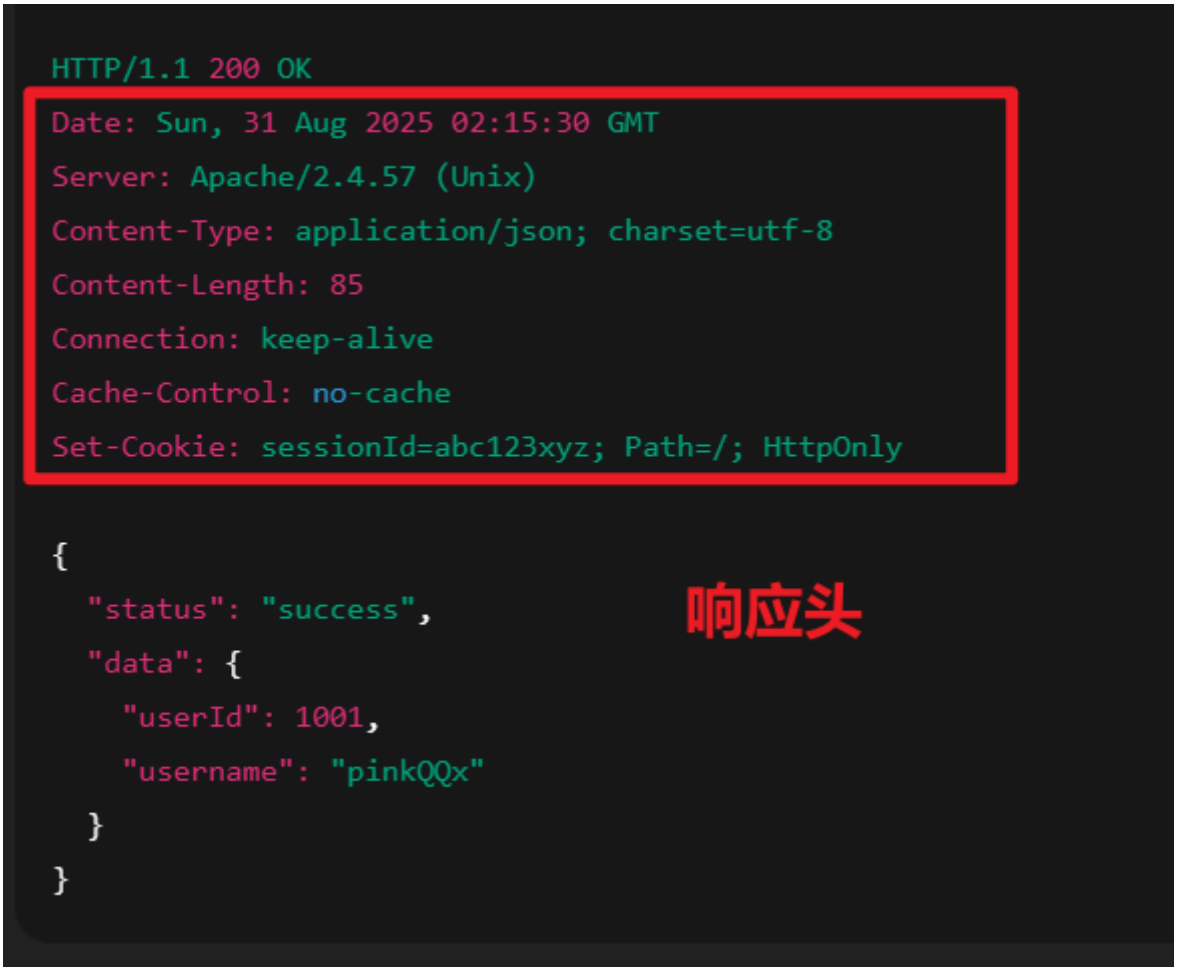
总结

- 1xx：临时响应，少见，用于协议交互。
- 2xx：请求成功，操作完成。
- 3xx：需重定向或缓存处理。
- 4xx：客户端错误，需修改请求。
- 5xx：服务器错误，需服务器修复。

响应头

（response headers）

描述响应报文的元信息，类似请求头



分类

1.通用头

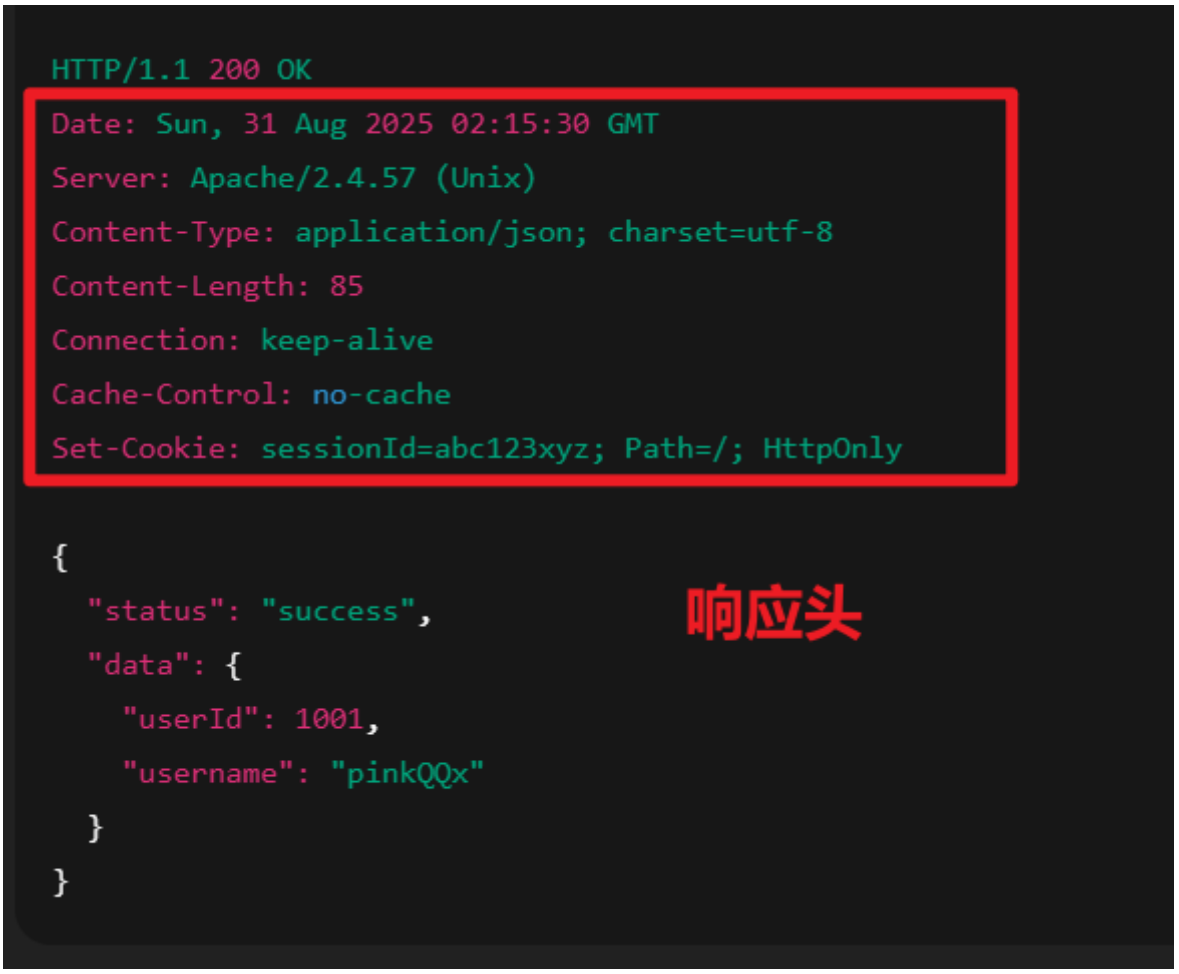
作用：用于请求或响应，描述报文整体信息，与具体请求或响应体无关

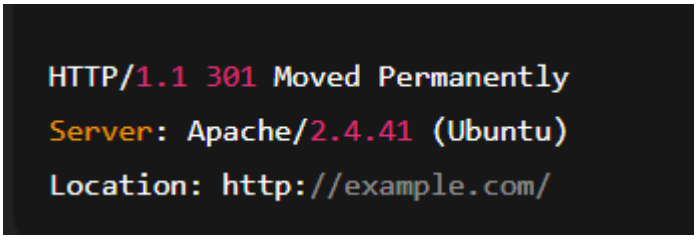
```
HTTP/1.1 200 OK
Date: Tue, 22 Aug 2025 12:00:00 GMT
Cache-Control: no-cache
Connection: keep-alive
Via: 1.1 proxy
```

字段	示例	作用
`Cache-Control`	`Cache-Control: no-cache`	指示缓存策略，`no-cache` 表示必须向服务器验证再使用缓存
`Connection`	`Connection: keep-alive`	控制 TCP 连接管理，`keep-alive` 表示保持长连接
`Date`	`Date: Tue, 22 Aug 2025 12:00:00 GMT`	报文生成时间，用于缓存和调试
`Via`	`Via: 1.1 proxy`	报文经过的代理/网关标识，用于追踪或防止循环

2.响应头

作用：专门用于响应，描述服务器信息和资源状态，**客户端根据这些头做处理。**


HTTP/1.1 200 OK
Date: Sun, 31 Aug 2025 02:15:30 GMT
Server: Apache/2.4.57 (Unix)
Content-Type: application/json; charset=utf-8
Content-Length: 85
Connection: keep-alive
Cache-Control: no-cache
Set-Cookie: sessionId=abc123xyz; Path=/; HttpOnly
{
"status": "success",
"data": {
"userId": 1001,
"username": "pinkQQx"
}
}
响应头


HTTP/1.1 301 Moved Permanently
Server: Apache/2.4.41 (Ubuntu)
Location: http://example.com/

字段	示例	作用
`Server`	`Server: Apache/2.4.41 (Ubuntu)`	服务器软件信息
`Location`	`Location: http://example.com/`	重定向 URL（状态码 3xx 时使用）
`WWW-Authenticate`	`WWW-Authenticate: Basic realm="example"`	认证信息，要求客户端提供凭证

3.实体头

作用：描述响应体（Response Body）的属性，如类型、长度、编码方式。

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 1256
Content-Encoding: gzip
Content-Language: en
```

字段	示例	作用
`Content-Type`	`Content-Type: text/html; charset=UTF-8`	响应体格式，如 HTML、JSON、图片等
`Content-Length`	`Content-Length: 1256`	响应体长度（字节数）
`Content-Encoding`	`Content-Encoding: gzip`	响应体是否经过压缩
`Content-Language`	`Content-Language: en`	响应体语言
`Content-Disposition`	`Content-Disposition: attachment; filename="file.pdf"`	响应体如何处理（如下载文件）

响应体

```mathematica
状态行（Status Line）
响应头（Response Headers）
空行（CRLF）
响应体（Response Body，可选）
```

- **响应体**紧跟在空行后面。
- 不是所有响应都有响应体：
  - `204 No Content`、`304 Not Modified`、HEAD 请求 → 没有响应体
  - 普通网页、API、文件下载 → 有响应体

2 响应体的内容类型

响应体可以是任何数据，类型由 **实体头（Entity Headers）** 指定：

Content-Type 类型	描述	示例
`text/html`	HTML 页面	`<html><body>Hello</body></html>`
`application/json`	JSON 数据	`{"name":"Alice","age":18}`
`application/xml`	XML 数据	`<user><name>Alice</name></user>`
`text/plain`	纯文本	`Hello world`
`image/png`	图片	二进制数据流
`application/octet-stream`	二进制流	文件下载、压缩包等

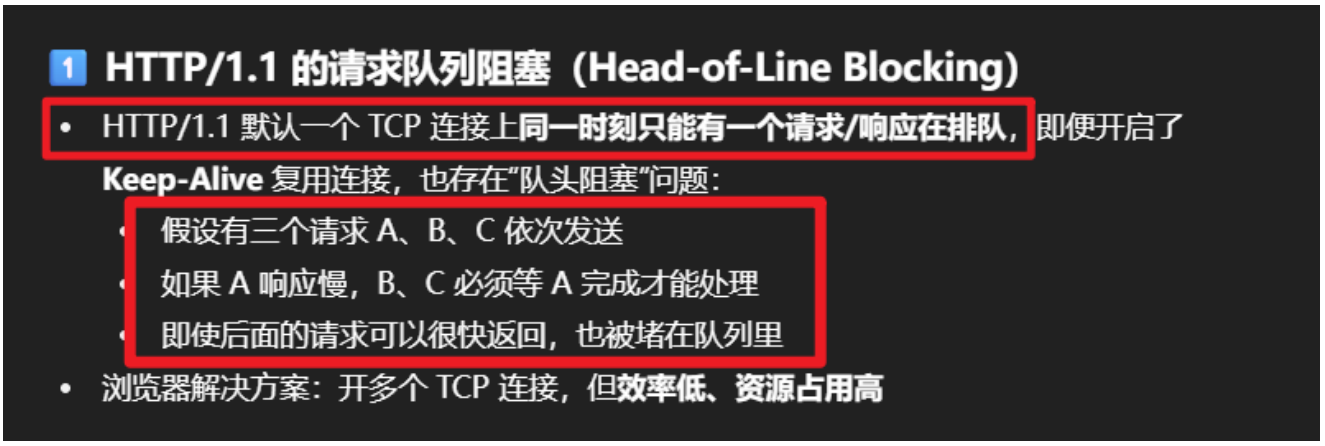
3 响应体与响应头的关系

- **Content-Type** → 告诉客户端数据格式，决定如何解析
- **Content-Length** → 响应体字节长度
- **Content-Encoding** → 响应体是否压缩（gzip、deflate）
- **Content-Language** → 响应体语言
- **Transfer-Encoding: chunked** → 分块传输时，客户端需按块拼接响应体

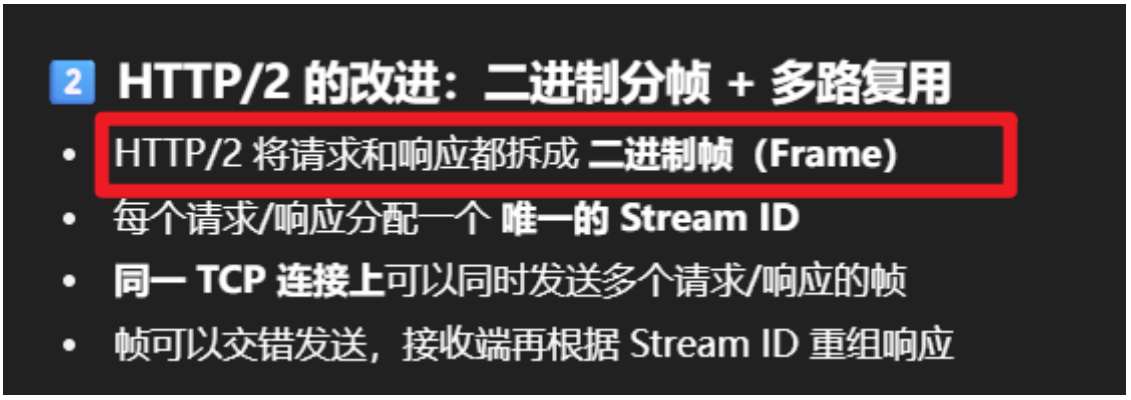
问题解决

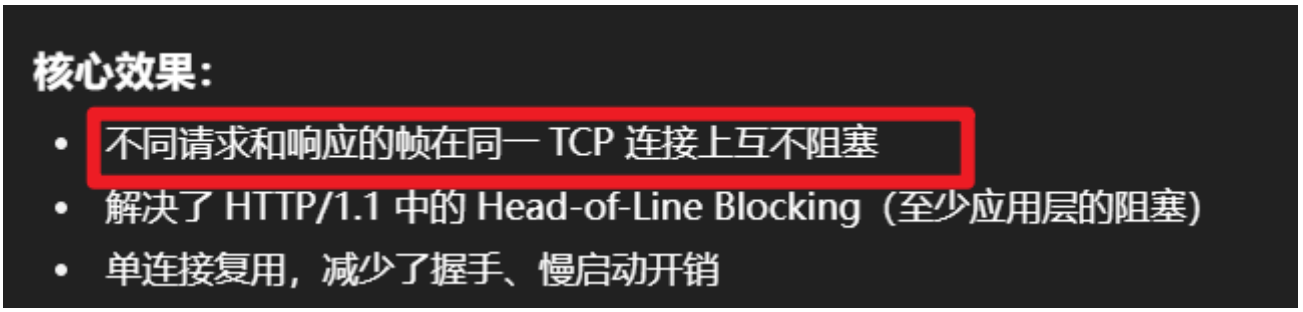
主要问题

请求队列阻塞



HTTP/2.0使用 **唯一id/交错发送/多路复用**





客户端根据唯一id对数据进行组装

浏览器想同时请求 A.js、B.css、C.png

```
┌──────────┐
| 单 TCP 连接 |
└──────────┘
```

帧交错发送：

[A1][B1][C1][A2][B2][C2] ...

服务器按 Stream ID 返回：

[A1][A2] → A.js

[B1][B2] → B.css

[C1][C2] → C.png

浏览器根据 Stream ID 组装完整响应

- A、B、C 的响应不会互相阻塞
- TCP 层仍然有顺序（低层可能有包重传），但 HTTP 层已经可以多路复用

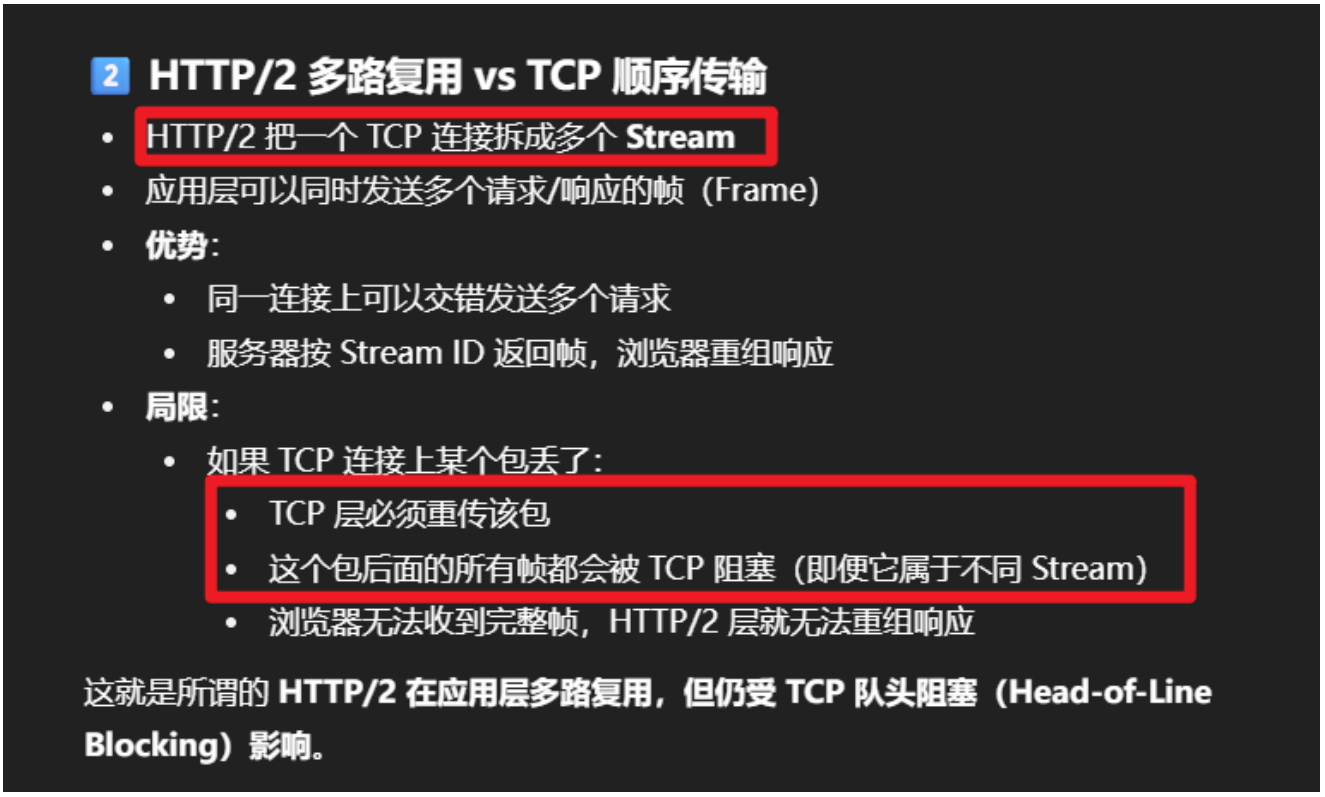
- **HTTP/2 并不能完全消除 TCP 层的 Head-of-Line Blocking**
  - 如果 TCP 某个包丢了，TCP 层会重传，仍会阻塞后续包
  - 这就是 HTTP/3 引入 QUIC（基于 UDP）解决的目标
- **优势**：
  - 单连接复用，减少多连接开销
  - 帧交错，提高响应并发效率
  - 头部压缩（HPACK）减少重复数据

TCP协议的局限性

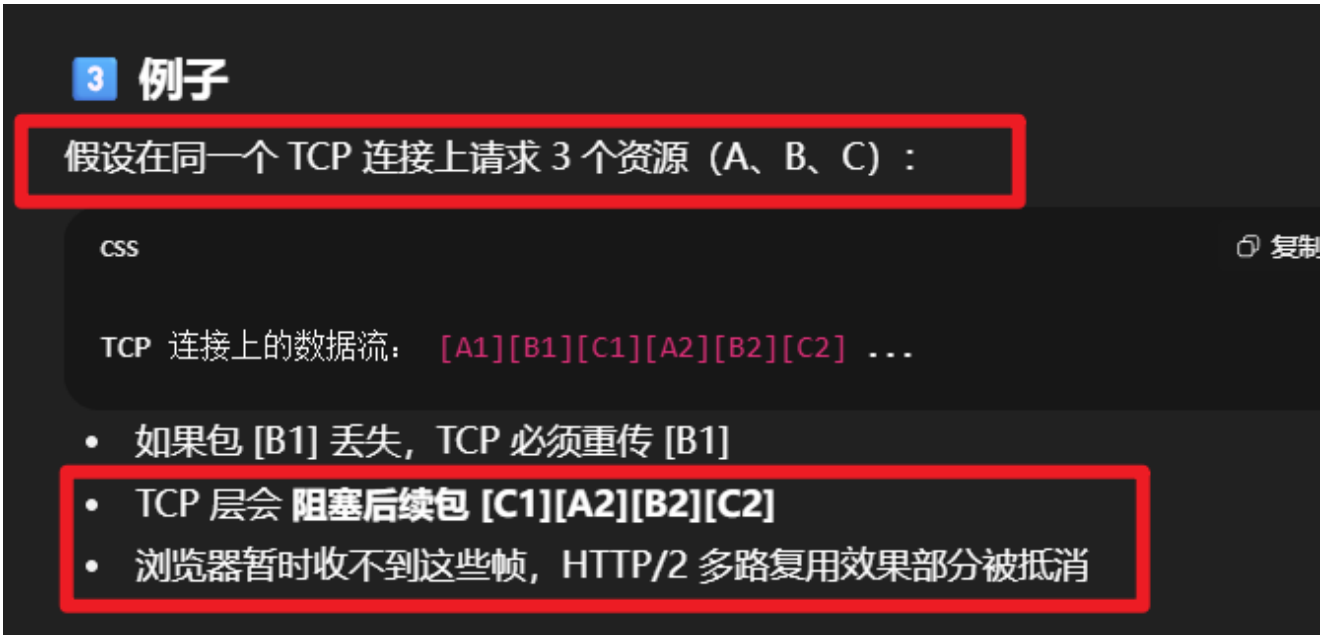
1 TCP 的特性：可靠顺序传输

- **TCP 协议**保证 **数据包可靠且顺序到达**
- 如果一个 TCP 包在传输过程中丢失：
  1. TCP 层会检测到丢包（通过 ACK 超时）
  2. 重新传输丢失的包
  3. 后续到达的包必须在 TCP 层 **等待丢失包重传完成** 才能交付给应用层

⚠ 重点：这就是 TCP 的 **顺序传输限制**，即便 HTTP/2 在应用层做多路复用，也受底层 TCP 约束。



举例



HTTP/3 的改进

使用了UDP作为传输层协议

1 UDP 的特性

1. **无连接**
   - 不建立 TCP 三次握手，不维护连接状态
2. **不保证顺序**
   - 数据包可以乱序到达
3. **不保证可靠性**
   - 数据包可能丢失，不重传
4. **轻量高效**
   - 头部开销小（8 字节），传输延迟低

✅ 重点：UDP 不保证顺序和可靠性，这正好解决了 **TCP 的队头阻塞问题。**

2 TCP 队头阻塞问题回顾

- TCP 保证顺序，所以丢包重传会阻塞整个连接的后续数据
- HTTP/2 即便应用层多路复用，也受 TCP 层限制

3 UDP 如何避免队头阻塞

- UDP 没有顺序约束 → 丢一个包不会阻塞其他包
- 可以把不同请求的数据放在不同的 UDP 数据包里发送
- 接收端可以根据 **包头信息或 Stream ID** 独立处理每个包
- 因此 **丢失一个包不会影响其他流**，应用层可以选择重传或忽略

> 简单理解：TCP = 排队买票（前面的人没到，后面的人必须等），UDP = 自助买票（每个人独立，前面慢不影响后面）

4 HTTP/3 + QUIC 的实现

- QUIC 运行在 UDP 上，实现了：
  1. **多路复用**：每个 HTTP/3 Stream 独立传输
  2. **可靠传输**：只对丢失的包重传，不阻塞其他 Stream
  3. **连接管理**：类似 TCP，但无需传统三次握手（减少延迟）

流程示意：

arduino 复制 编辑

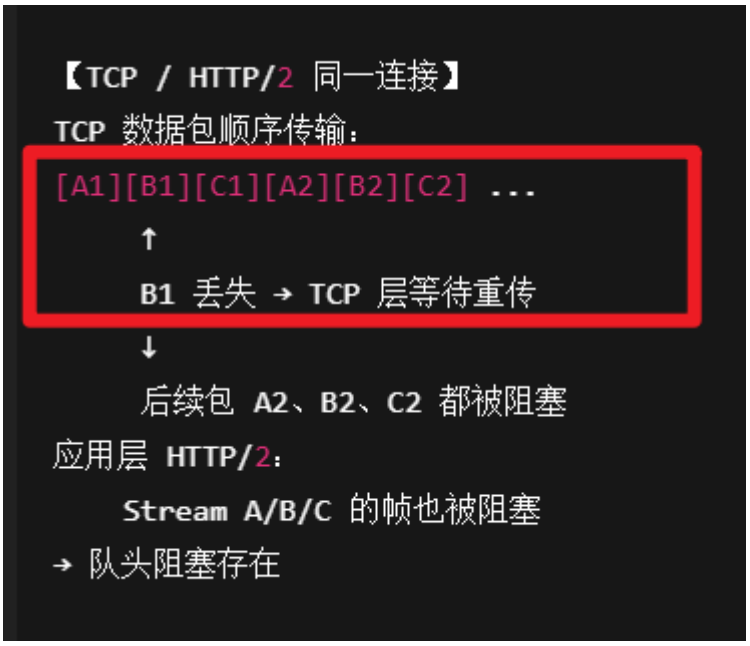
```
HTTP/3 Stream A ┐
HTTP/3 Stream B ┼─ UDP 数据包独立传输 → 丢包只重传本 Stream
HTTP/3 Stream C ┘
```

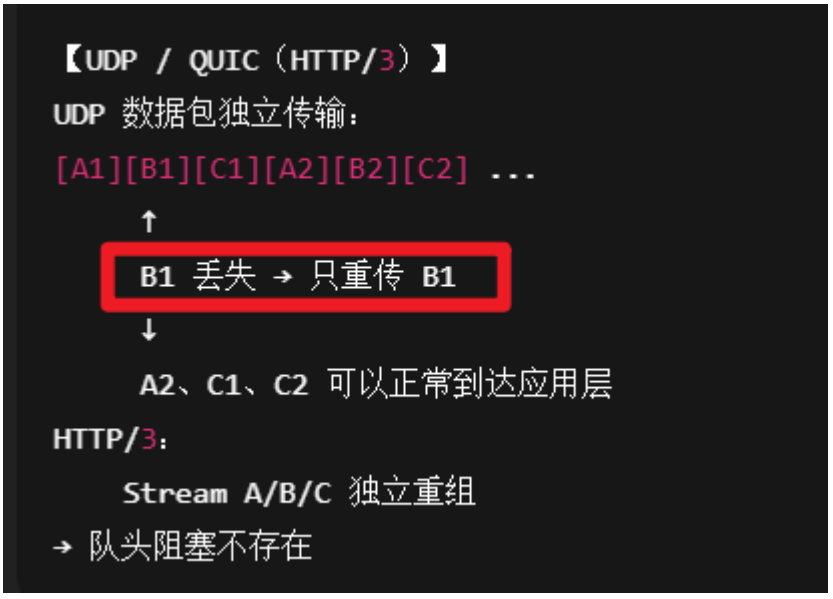
- 丢包重传只针对 A/B/C 中具体丢失的包
- 其他 Stream 不受影响 → 真正解决了 HTTP/2 的 TCP 队头阻塞问题

5 总结

特性	TCP	UDP	QUIC（HTTP/3）
可靠性	高，顺序传输	不保证	高，按 Stream 重传
顺序传输	必须顺序	可乱序	Stream 内顺序，Stream 间独立
队头阻塞	存在	不存在	不存在（每个 Stream 独立）
多路复用	HTTP/2 层解决，受 TCP 限制	本身轻量，可自定义	完全支持，应用层无阻塞

示意图





1

HTTPS协议

HTTPS 也是 HTTP 协议，它的请求报文和响应报文结构 **和 HTTP 一模一样**，只是传输时加密了，不是明文。

1 协议层面的区别

项目	HTTP	HTTPS
全称	HyperText Transfer Protocol（超文本传输协议）	HyperText Transfer Protocol Secure（安全超文本传输协议）
传输方式	明文传输	加密传输（通过 TLS/SSL）
端口	默认 80	默认 443
协议层	TCP/IP	TCP/IP + TLS/SSL

解释：

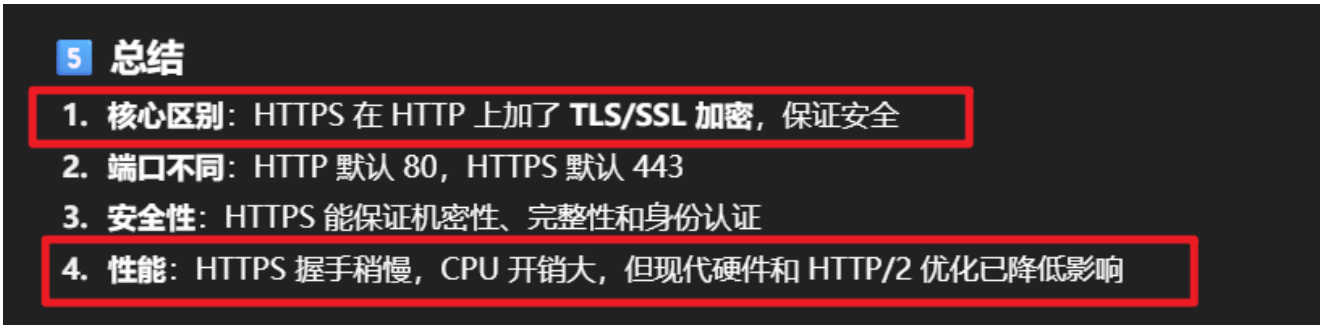
- HTTPS 在 HTTP 基础上增加了一层 **加密层（TLS/SSL）**，保证数据在传输过程中无法被窃听或篡改。

2 安全性区别

项目	HTTP	HTTPS
数据加密	不加密，明文传输	对称加密 + 非对称加密，保证数据安全
身份认证	无法保证服务器身份	通过证书（CA 签发）验证服务器身份，防止中间人攻击
数据完整性	无保障	有消息摘要（MAC），保证数据未被篡改

3 性能和资源

项目	HTTP	HTTPS
握手	直接建立 TCP 连接	需要 TCP + TLS/SSL 握手，稍慢
CPU 消耗	较低	加密/解密需要消耗 CPU
缓存	与 HTTP 一样	可以缓存，但需要正确配置证书和 HSTS



DNS协议

把域名转换成协议

2 DNS 的工作原理（简化）

1. 用户在浏览器输入域名 `www.example.com`
2. 浏览器向操作系统请求解析域名
3. 操作系统查询 **本地 DNS 缓存**，没有就向 **DNS 服务器** 发起请求
4. DNS 服务器查询：
   - 根域名服务器 → 顶级域名服务器 → 权威域名服务器
   - 最终返回域名对应的 IP 地址
5. 浏览器拿到 IP 地址后发起 HTTP/HTTPS 请求

3 DNS 协议的基本特征

特性	说明
协议类型	应用层协议
传输层协议	UDP（默认 53 端口）、TCP（大数据或 zone 传输时使用）
数据格式	DNS 消息，包括 **请求报文** 和 **响应报文**
查询类型	A（IPv4 地址）、AAAA（IPv6 地址）、CNAME（别名）、MX（邮件服务器）等

4 DNS 消息结构（简化）

请求报文

- **头部（Header）**：ID、标志位（查询/响应）、问题数等
- **问题（Question）**：要查询的域名、类型（A、AAAA、CNAME 等）
- **可选字段**：附加信息（Additional）

响应报文

- **头部（Header）**：同请求头，标记响应状态
- **问题（Question）**：和请求报文一致
- **回答（Answer）**：域名对应的 IP 或别名
- **授权（Authority）**：权威 DNS 信息
- **附加（Additional）**：额外信息，如备用 IP

5 端口和传输方式

- **UDP 53**：主要用于普通查询，快速且轻量
- **TCP 53**：用于大数据传输或区域传送（zone transfer）

6 总结

- **DNS 协议 = 域名解析协议**，在应用层工作
- 作用是 **域名 ↔ IP 地址** 转换
- 支持多种查询类型（A、AAAA、CNAME、MX 等）
- 消息结构：**请求报文 + 响应报文**，头部、问题、回答、权威信息、附加信息
- 使用 UDP 53 为主，TCP 53 为辅

SSH协议

为什么不直接登录 要用SSH登录

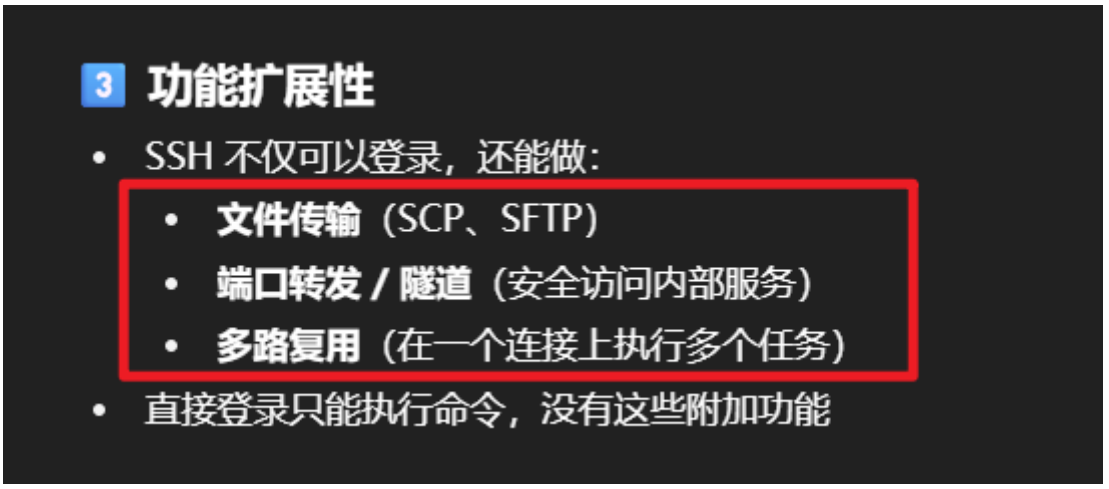
这是一个非常实际的问题，我们可以从 **安全性、管理和便捷性**几个方面来解释为什么不直接登录，而要使用 **SSH 登录**。

1 安全性

1. **加密传输**
   - 直接登录（比如 Telnet 或普通远程终端）数据是 **明文传输**
   - 密码和命令可能被抓包、窃听
   - **SSH** 使用 **对称加密 + 非对称加密**，所有通信数据都是加密的
2. **防止中间人攻击**
   - SSH 可以通过服务器公钥认证，防止假冒服务器
   - 直接登录无法验证对方身份，容易被冒充
3. **数据完整性**
   - SSH 使用 MAC（消息认证码）防止数据在传输过程中被篡改
   - 明文协议无法保证命令和数据完整性

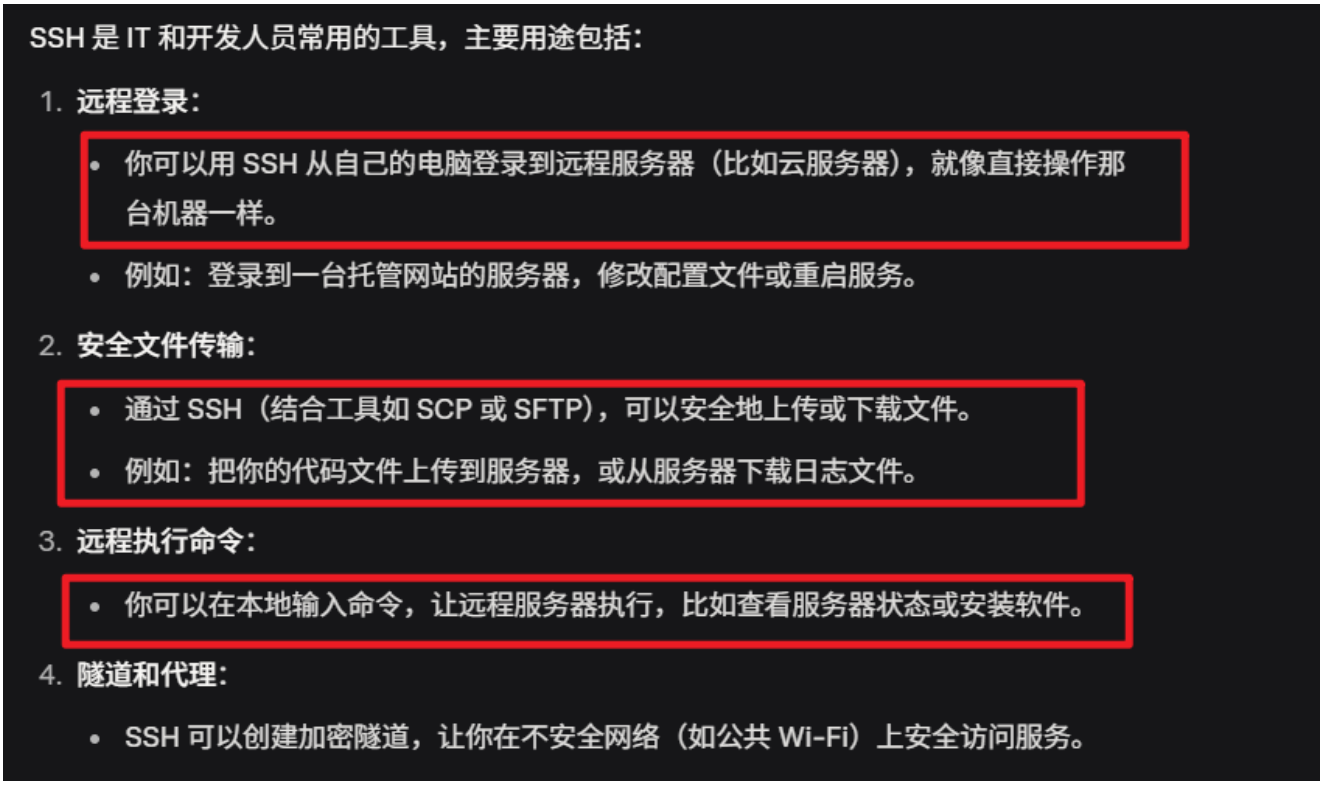
2 认证机制

- SSH 支持多种 **安全认证方式**：
  - 密码认证
  - 公钥认证（更安全、无需每次输入密码）
  - Kerberos / GSSAPI 等
- 直接登录一般只有明文密码，安全性低



SSH（Secure Shell，安全壳协议）是一种用于安全远程访问和传输数据的网络协议。

简单来说，它就像一个安全的“通道”，**让你的电脑可以安全地连接到另一台电脑或服务器（比如远程操作一台服务器或传输文件），保护数据不被窃听或篡改。**



2 SSH 协议的核心特性

特性	说明
安全性	使用加密算法（对称加密 + 非对称加密 + 哈希）保护数据传输
身份认证	支持密码、密钥、Kerberos 等认证方式
完整性	通过消息认证码（MAC）防止数据被篡改
端口	默认 TCP 22
协议层	应用层（运行在 TCP 之上）

3 SSH 协议组成（分层）

SSH 协议通常由 **三层结构**组成：

1. **传输层协议（Transport Layer Protocol）**
   - 建立加密安全通道
   - 负责认证服务器、协商加密算法、数据完整性
   - 特性：加密 + 完整性 + 防重放攻击
2. **用户认证协议（User Authentication Protocol）**
   - 验证客户端身份
   - 支持：
     - 密码认证（password）
     - 公钥认证（public key）
     - Kerberos / GSSAPI 等
3. **连接协议（Connection Protocol）**
   - 支持多路复用（multiplexing）
   - 可以在一个 SSH 连接上同时打开多个 **虚拟通道**
     - 远程 Shell
     - 文件传输（SCP / SFTP）
     - 端口转发（local/remote forwarding）

4 SSH 的连接流程（简化）

1. **客户端发起 TCP 连接** → 默认 22 端口
2. **建立安全通道（Transport Layer）** → 协商加密算法、生成密钥
3. **身份认证（User Authentication）** → 密码或公钥验证
4. **建立连接通道（Connection Layer）** → 可以执行命令、传输文件或做端口转发

例如使用SSH链接仓库

1. 安装 Git

- Linux / Mac:

```
bash

git --version
```

- Windows：安装 Git for Windows，带 Git Bash

2. 检查本地 SSH

- 查看本地是否已有 SSH Key:

```
bash                                                    复制  编辑

ls ~/.ssh/id_rsa.pub
```

- 如果存在公钥文件（`.pub`），说明已有；如果没有，需要生成新的 SSH Key。

2 生成 SSH Key（如果没有）

bash

```
ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
```

- `-t rsa` → 使用 RSA 算法
- `-b 4096` → 密钥长度 4096 位
- `-C` → 注释（一般写邮箱）
- 默认会生成两个文件：
  - **私钥**：`~/.ssh/id_rsa` → **不要泄露**
  - **公钥**：`~/.ssh/id_rsa.pub` → 上传到 GitLab

3 添加公钥到 GitLab

1. 登录 GitLab → 用户头像 → **Settings** → **SSH Keys**
2. 打开本地公钥：

bash

```
cat ~/.ssh/id_rsa.pub
```

3. 复制内容到 GitLab 的 **Key** 区域，设置标题，点击 **Add key**

4 配置 SSH 客户端

1. 配置 `~/.ssh/config`（可选，但推荐）

text

```
Host gitlab.com
    HostName gitlab.com
    User git
    IdentityFile ~/.ssh/id_rsa
```

- 解释：
  - `Host gitlab.com` → 匹配远程主机
  - `HostName` → 实际地址
  - `User git` → GitLab 使用 git 用户进行 SSH 访问
  - `IdentityFile` → 指定私钥路径

2. 测试 SSH 连接

bash 复制 编辑

```
ssh -T git@gitlab.com
```

- 正常情况会显示：

css 复制 编辑

```
Welcome to GitLab, @your_username!
```

- 如果提示 `Are you sure you want to continue connecting?` → 输入 `yes`，将主机加入 `known_hosts`

5 拉取仓库流程

1. 克隆仓库

bash

```
git clone git@gitlab.com:username/repository.git
```

- `git@gitlab.com` → 使用 SSH 协议
- `username/repository.git` → 仓库路径