一.传输层

1.TCP

`tcp` 传输控制协议 `Transimision Control Protocal` 可靠、面向连接的协议,传输效率低 (在不可靠的`IP`层上建立可靠的传输层)。 TCP提供全双工服务，即数据可在同一时间双向传播。

1).TCP数据段格式

- 源端口号、目标端口号，指代的是发送方随机端口，目标端对应的端口
- 序列号：32位序列号是用于对数据包进行标记，方便重组
- 确认序列号：期望发送方下一个发送的数据的编号
- 4位首部长度：单位是字节，4位最大能表示15，所以头部长度最大为60
- **`URG`:紧急新号、`ACK`:确认信号、`PSH`:应该从TCP缓冲区读走数据、`RST`：断开重新连接、`SYN`:建立连接、`FIN`:表示要断开**
- 窗口大小：发送方期望接收的字节数，当网络通畅时将这个窗口值变大加快传输速度，当网络不稳定时减少这个值。在TCP中起到流量控制作用。
- 校验和：由发送方计算，接收方验证。用来做差错控制
- 紧急指针：用来发送紧急数据使用

> TCP 对数据进行分段打包传输，对每个数据包编号控制顺序。

2).TCP抓包

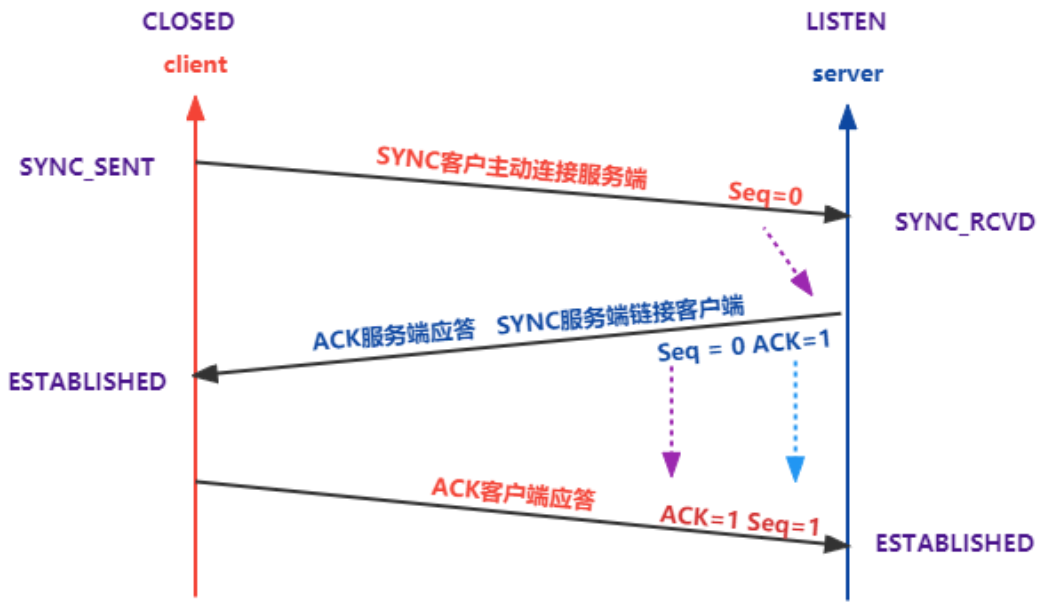
`client.js`

```
const net = require('net');
const socket = new net.Socket();
// 连接8080端口
socket.connect(8080, 'localhost');
```

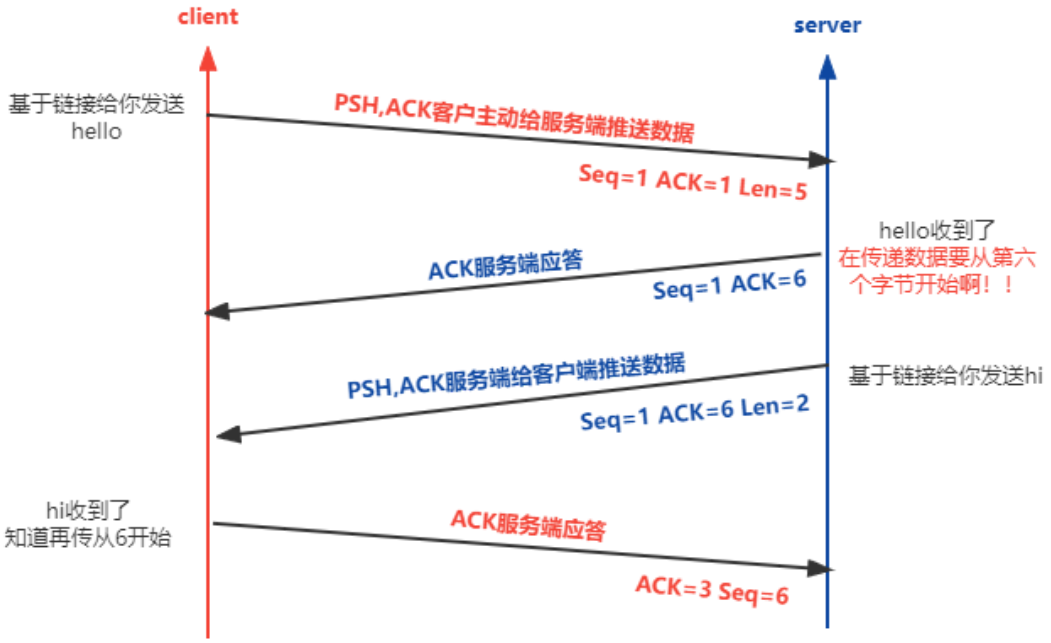
```
// 连接成功后给服务端发送消息
socket.on('connect', function(data) {
    socket.write('hello'); // 浏览器和客户端说 hello
    socket.end()
});
socket.on('data', function(data) {
    console.log(data.toString())
})
socket.on('error', function(error) {
    console.log(error);
});
```

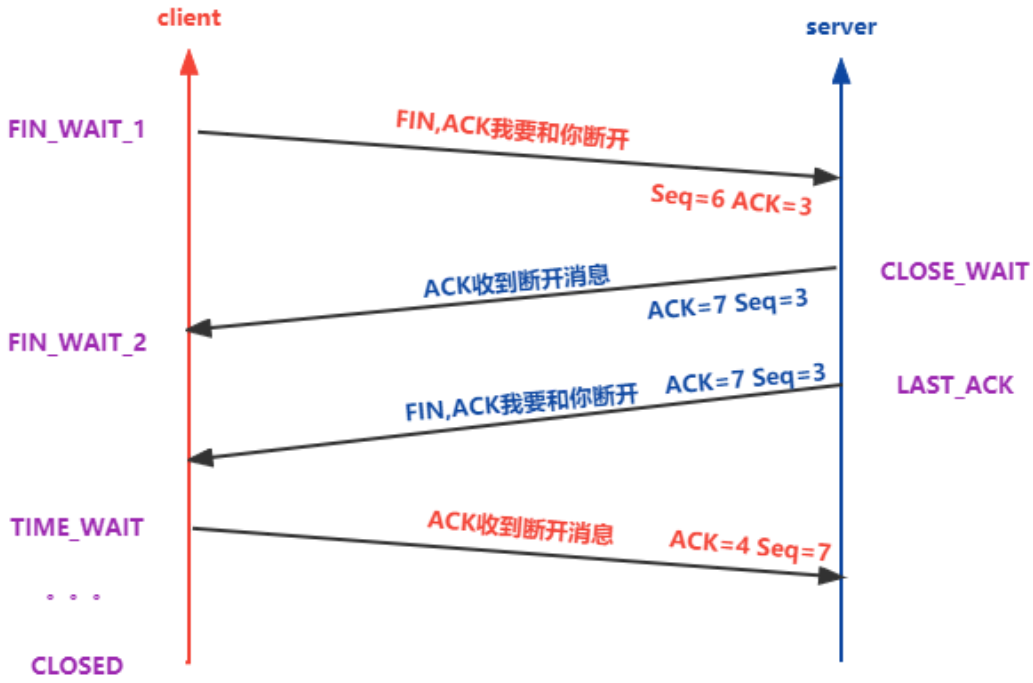
server.js

```
const net = require('net');
const server = net.createServer(function(socket){
    socket.on('data',function (data) { // 客户端和服务端
        socket.write('hi'); // 服务端和客户端说 hi
    });
    socket.on('end',function () {
        console.log('客户端关闭')
    })
})
server.on('error',function(err){
    console.log(err);
})
server.listen(8080); // 监听8080端口
```



- 1）我能主动给你打电话吗？ 2）当然可以啊！那我也能给你打电话吗？
- 3）可以的呢，建立连接成功！





- 1）我们分手吧 2）收到分手的信息
- 3）好吧，分就分吧 4）行，那就到这里了

3).TCP滑动窗口 (发送的数据要有有序 是从一组数据中发送某一部分)

- 滑动窗口：TCP是全双工的，所以发送端有发送缓存区；接收端有接收缓存区，要发送的数据都放 到发送者的缓存区，发送窗口（要被发送的数据）就是要发送缓存中的哪一部分

- 核心是流量控制：在建立连接时，接收端会告诉发送端自己的窗口大小（`rwnd`）,每次接收端收到数据后都会再次确认（`rwnd`）大小，如果值为0，停止发送数据。（并发送窗口探测包，持续监测窗口大小）

思考题:

- 1.求数组中连续k项最大的和。

```
let arr = [1, 3, 10, -2, 9, 8, -4];  // k = 3;
function subSum(arr,k = 3) {
    let max = 0;
    for(let i = 0 ; i < k;i++){
        max +=  arr[i]; // 假设最大值是第一组
    }
    let win = max
    for(let i = k; i< arr.length;i++ ){
        win += arr[i] - arr[i - k]; // 出一个进一个
        max = Math.max(win,max)
    }
    return max
}
console.log(subSum(arr, 3))
```

- 2.无重复字符的最长字符串长度

```
let str = 'abcabcbb'
function findStr(str) {
    let left = 0; // 滑动窗口边界
    let right = 0;
    let set = new Set(); // 创建一个set
    let maxLen = 0;
    while(right < str.length){
        while(set.has(str[right])){ // 集合中有我就全部删掉
            set.delete(str[left++]);
        }
        set.add(str[right]); // 重新添加算长度
        maxLen = Math.max(maxLen, set.size)
        right++;
    }
    return maxLen
}
console.log(findStr(str))
```

4).TCP粘包 （node中默认采用的就是nagle算法）

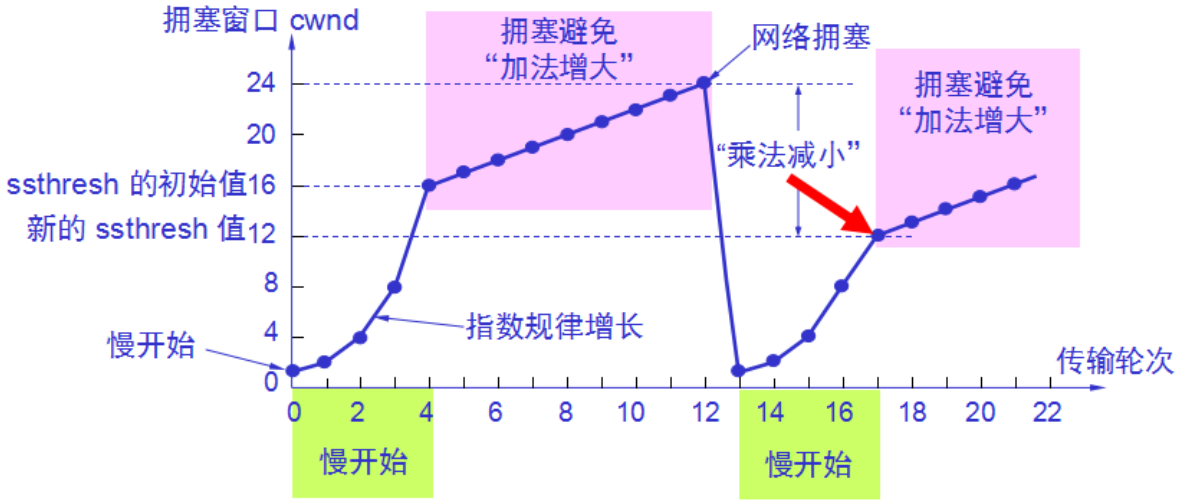
`Nagle`算法的基本定义是**任意时刻，最多只能有一个未被确认的小段** (TCP内部控制)

`Cork算法` 当达到`MSS`(Maximum Segment Size)值时统一进行发送（此值就是帧的大小 - `ip`头 - `tcp`头 = 1460个字节）理论值 不同的网络值不同

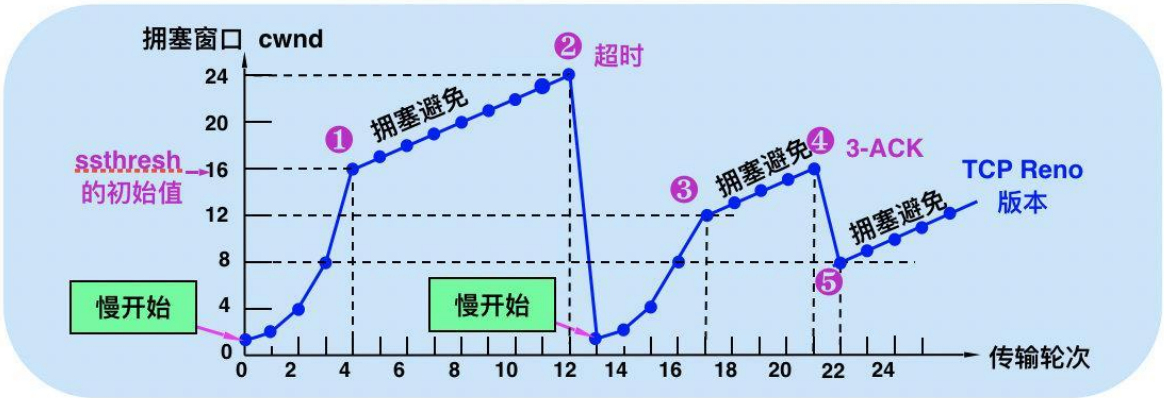
5). TCP拥塞处理

> 举例：假设接收方窗口大小是无限的，接收到数据后就能发送`ACK`包，那么传输数据主要是依赖于网络带宽，带宽的大小是有限的。

- TCP 维护一个拥塞窗口`cwnd` （congestion window）变量，在传输过程正没有拥塞就将此值增大。如果出现拥塞（超时重传`RTO(Retransmission TimeOut)`）就将窗口值减少。
- `cwnd < ssthresh` 使用慢开始算法
- `cwnd > ssthresh` 使用拥塞避免算法
- ROT时更新`ssthresh`值为当前窗口的一半，更新`cwnd` = 1



- 传输轮次：`RTT` (Round-trip time) ,从发送到确认信号的时间
- `cwnd`控制发送窗口的大小。



> 快重传，可能在发送的过程中出现丢包情况。此时不要立即回退到慢开始阶段，而是对已经收到的报文重复确认，如果确认次数达到3此，则立即进行重传 **快恢复算法** (减少超时重传机制的出现)，降低重置`cwnd`的频率。

6).TCP缺陷

- TCP队头阻塞问题
- TCP中慢启动问题
- TCP中短连接问题

二.应用层

1.HTTP发展历程

1990年HTTP/0.9为了便于服务器和客户端处理，采用了纯文本格式，只运行使用GET请求。在响应请求之后会立即关闭连接。

1996年HTTP/1.0增强了 0.9 版本，引入了 HTTP Header（头部）的概念，传输的数据不再仅限于文本，可以解析图片音乐等，增加了响应状态码和POST,HEAD等请求方法。

1999年广泛使用HTTP/1.1，正式标准，允许持久连接，允许响应数据分块，增加了缓存管理和控制，增加了 PUT、DELETE 等新的方法。

2015年HTTP/2，使用HPACK算法压缩头部，减少数据传输量。允许服务器主动向客户端推送数据，二进制协议可发起多个请求，使用时需要对请求加密通信。

2018年HTTP/3 基于UDP的QUIC协议。

2.HTTP1.1

1).内容协商

客户端和服务端进行协商，返回对应的结果

客户端和服务端进行协商，返回对应的结果

客户端 Header	服务端 Header	
Accept	Content-Type	我发送给你的数据是什么类型
Accept-encoding	Content-Encoding	我发送给你的数据是用什么格式压缩(gzip、deflate、br)
Accept-language		根据客户端支持的语言返回 （多语言）
Range	Content-Range	范围请求数据 206

2).长连接

`TCP` 的连接和关闭非常耗时间，所以我们可以复用`TCP`创建的连接。HTTP/1.1响应中默认会增加`Connection:keep-alive`

3).管线化

`HTTP1.1`支持管线化机制须通过长连接来实现，在一条`TCP`通道中可以同时提交多个`http`请求 （但是需要按照请求顺序依次响应，**管道的特点**） 只有幂等的请求才能够被管线化。 （默认都不支持管线化）

4).多个TCP链接

Head-of-line blocking队头阻塞指的是在同一个`tcp`链接中，如果先发送的`http`请求如果没有响应的话，后面的`http`请求也不会响应。 对一个域名同时发起多个长连接实现并发。 默认`chrome`为6个。

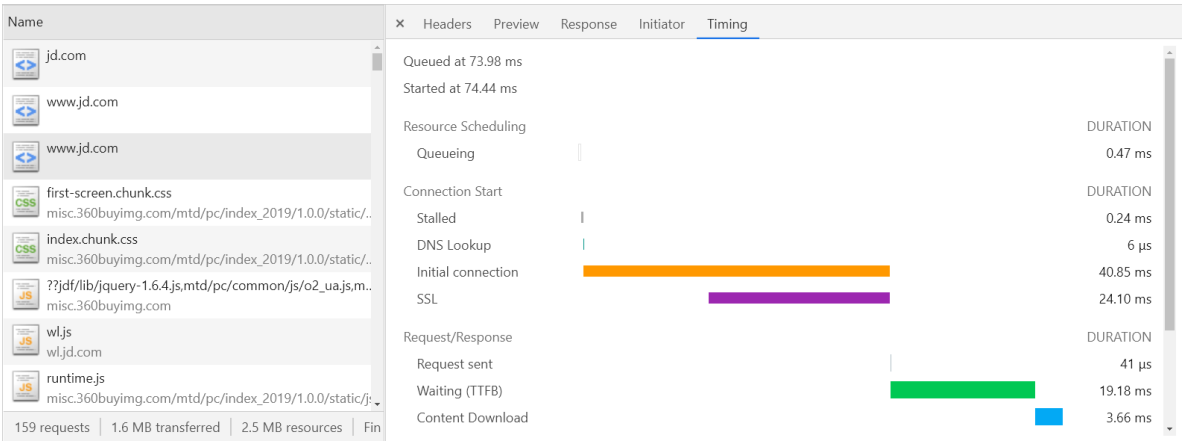
5).`Cookie`

Set-Cookie/Cookie用户第一次访问服务器的时候，服务器会写入身份标识，下次再请求的时候会携带`cookie`。通过Cookie可以实现有状态的会话

6).`HTTP`缓存

- **强缓存** 服务器会将数据和缓存规则一并返回，缓存规则信息包含在响应header中。`Cache-Control`
- **对比缓存** `if-Modified-Since/if-None-Match`、`Last-modified/Etag`

7).`HTTP1.1`优化



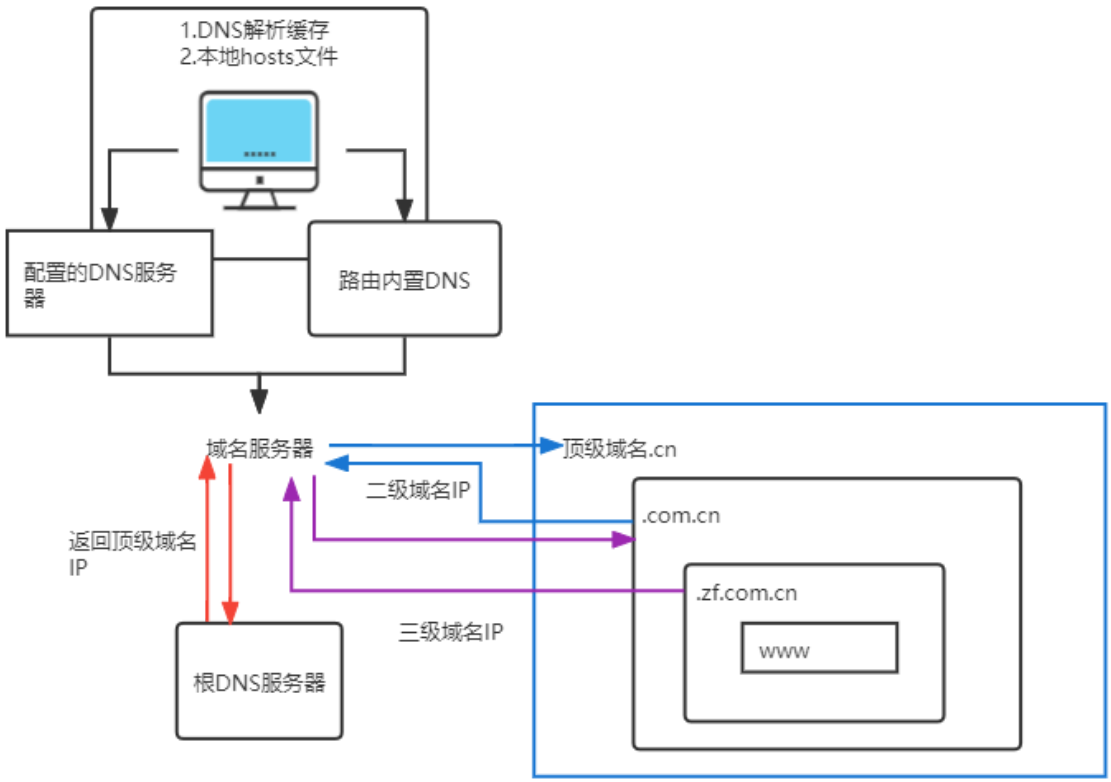
- `Queuing`: 请求发送前会根据优先级进行排队，同时每个域名最多处理6个TCP链接，超过的也会进行排队，并且分配磁盘空间时也会消耗一定时间。
- `Stalled` :请求发出前的等待时间（处理代理，链接复用）

- `DNS lookup` :查找`DNS`的时间
- `initial Connection` :建立TCP链接时间
- `SSL`:`SSL`握手时间（`SSL`协商）
- `Request Sent` :请求发送时间（可忽略）
- `Waiting(TTFB)`:等待响应的时间，等待返回首个字符的时间
- `Content Dowloaded` :用于下载响应的时间

> 手段:

- 减少网站中使用的域名`域名越多`，`DNS`解析花费的时间越多。
- 减少网站中的重定向操作，重定向会增加请求数量。
- 选用高性能的Web服务器`Nginx`代理静态资源 。
- 资源大小优化：对资源进行压缩、合并（合并可以减少请求，也会产生文件缓存问题）， 使用`gzip/br`压缩。
- 给资源添加强制缓存和协商缓存。
- 升级`HTTP/1.x`到`HTTP/2`
- 付费、将静态资源迁移至`CDN`

`DNS`解析流程

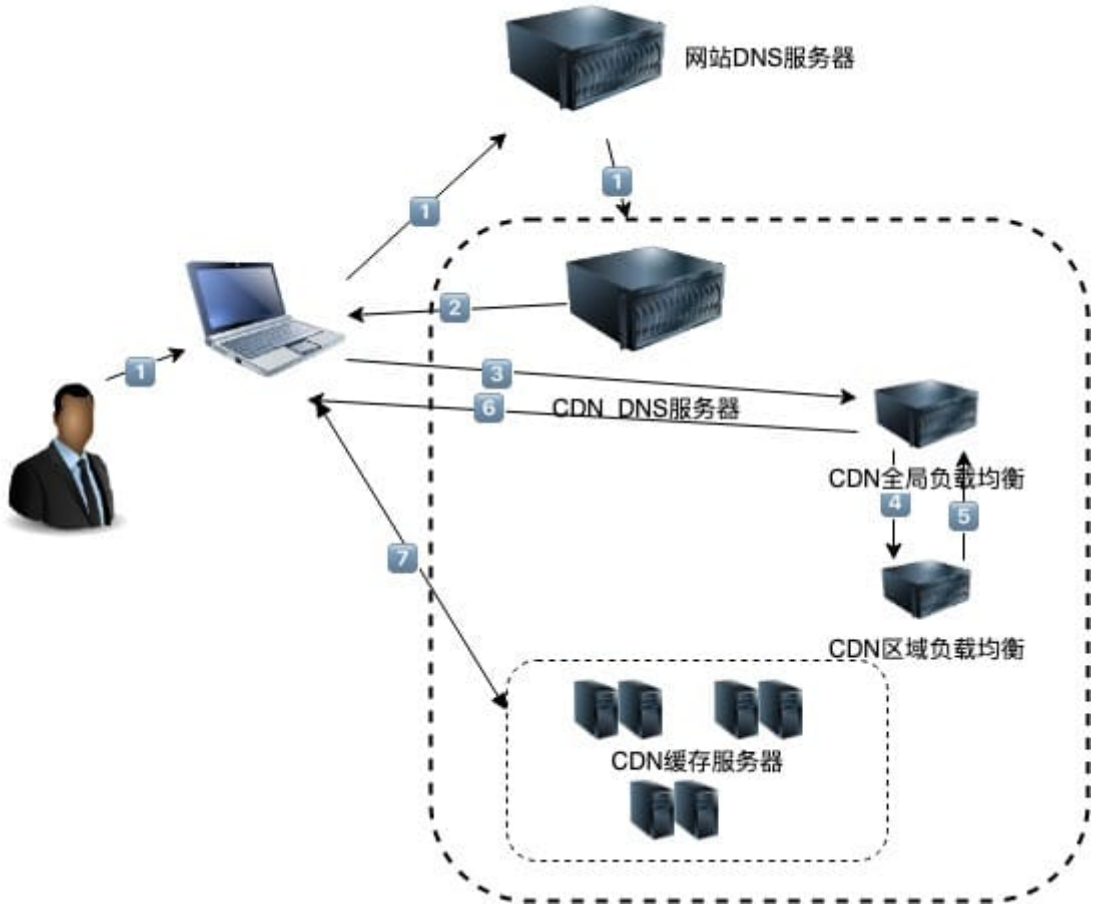


`CDN`解析流程

`CDN`的全称是Content Delivery Network，受制于网络的限制，访问者离服务器越远访问速度就越慢

核心就是**离你最近的服务器给你提供数据** （代理 + 缓存）

- 先在全国各地架设`CDN`服务器
- *正常访问网站会通过`DNS`解析，解析到对应的服务器*
- 解析1：我们通过`CDN`域名访问时，会被解析到`CDN`专用`DNS`服务器。并返回`CDN`全局负载均衡服务器的`IP`地址。
- 解析2：向全局负载均衡服务器发起请求，全局负载均衡服务器会根据用户`IP`分配用户所属区域的负载均衡服务器。并返回一台`CDN`服务器`IP`地址
- 用户向`CDN`服务器发起请求。如果服务器上不存在此文件。则向上一级缓存服务器请求，直至查找到源服务器，返回结果并缓存到`DNS`服务器上。



3.`HTTP/2`

> HTTP/2主要的目标就是改进性能，兼容HTTP/1.1

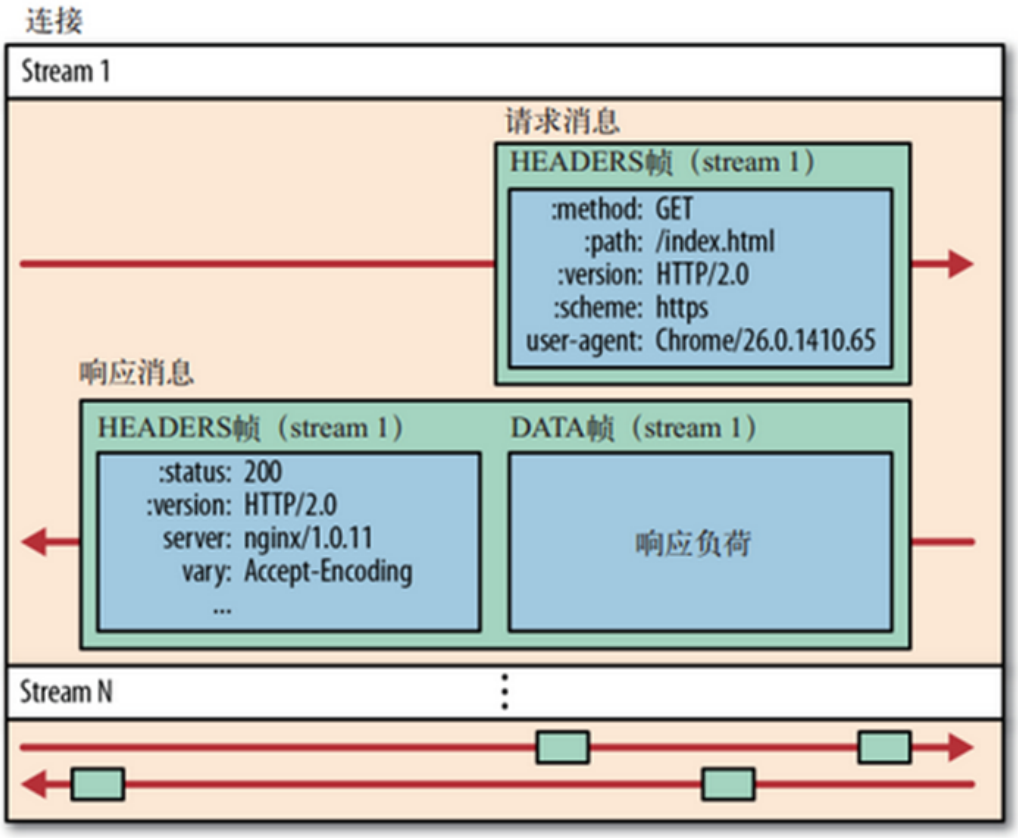
- 问题1：`HTTP/1.1`中只优化了`body`（`gzip`压缩）并没有对头部进行处理
- 问题2：`HTTP/1.1` 问题在于当前请求未得到响应时，不能复用通道再次发送请求。如果第一个请求没有返回会被阻塞 **HTTP队头阻塞问题。**

1).多路复用

> 在一条TCP链接上可以乱序收发请求和响应，多个请求和响应之间不再有顺序关系

- 同域下采用一个TCP链接传输数据
- 采用二进制格式， HTTP/1.1采用的是纯文本需要处理空行、大小写等。文本的表现形式有多样性，二进制则只有0和1的组合不在有歧义而且体积更小。把原来的

`Header+body`的方式转换为二进制帧。



- `HTTP/2`虚拟了流的概念（有序的帧），给每帧分配一个唯一的流ID，这样数据可以通过`ID`按照顺序组合起来

> 帧的组成及大小

```
+-----------------------------------------------+
|                 Length (24)                   |
+---------------+---------------+---------------+
|   Type (8)    |   Flags (8)   |
+-+-------------+---------------+-------------------+
|R|                 Stream Identifier (31)          |
+=+=================================================+
|                   Frame Payload (0...)        ...
+---------------------------------------------------+
```

- Length帧的大小，`2^24`帧最大不能超过`16M`
- Type帧的类型：常用的就是 HEADERS，DATA
- Flags标志位：常用的是`END_HEADERS`，`END_STREAM`，`PRIORITY`
- Stream Identifier 流的标号

2).头部压缩

> 使用`HPACK`算法压缩HTTP头

- 废除起始行，全部移入到Header中去，采用**静态表**的方式压缩字段
- 如果是自定义Header，在发送的过程中会添加到**静态表**后，也就是所谓的**动态表**
- 对内容进行哈夫曼编码来减小体积

3).服务端推送

> 服务端可以提前将可能会用到的资源主动推送到客户端。

4.HTTP/3

> **目前还处于草案阶段**　解决TCP中**队头阻塞**问题
>
> TCP为了保证可靠传输，如果在传输的过程中发生丢包，可能此时其他包已经接受完毕，但是仍要等待客户端重传丢失的包。这就是TCP协议本身**队头阻塞**的问题。

1).QUIC 协议

- HTTP/3中关键的改变，那就是把下层的 `TCP` 换成了 `UDP`。 `UDP` 无序从而解决了**队头阻塞**的问题
- `QUIC` 基于 `UDP` 实现了可靠传输、流量控制，引入流和多路复用
- `QUIC` 全面采用加密通信，`QUIC` 使用了 `TLS` 1.3，首次连接只需要 `1RTT`
- 支持**链接迁移**，不受 `IP` 及 `port` 影响而发生重连，通过 `ConnectionID` 进行链接
- 使用 `QPACK` 进行头部压缩， `HPACK` 要求传输过程有序（动态表），会导致队头阻塞。