

初识HTTP

1.1 什么是HTTP

HTTP：超文本传输协议（Hyper Text Transfer Protocol）

HTTP协议是应用层协议，其底层是基于 TCP 协议做的传输层协议，经过了 TLS/SSL 加密

对称加密：加密和解密都是使用同一个密钥

非对称加密：加密和解密需要使用两个不同的密钥—公钥和私钥

每个HTTP请求都分为请求和响应两部分，HTTP请求都是语义比较简单的请求，在设计上可以提供很多扩展的能力，即HTTP协议是一个简单可扩展的协议

HTTP是一个无状态的协议，每一个请求之间都是相互孤立的，当前的请求是无法获取之前携带过什么信息，做过什么事情

1.2 协议分析

发展

- HTTP/0.9- 单行协议
 - 只有 GET 和目标地址一行
 - 只能承载 HTML 文档
- HTTP/1.0- 构建可扩展性
 - 增加了 header 和状态码
 - 可以支持多种文档
 -
- HTTP/1.1- 标准化协议
 - 链接复用
 - 缓存
 - 内容协商
 -
- HTTP/2- 更优异的表现
 - 二进制协议
 - 压缩 header
 - 服务器推送
 -

帧：HTTP/2 通信的最小单位，每个帧都包含帧头，至少也会识别出当前帧所属的数据流

消息：与逻辑请求或响应消息对应的完整的一系列帧

数据流：已建立的连接内的双向字节流，可以承载一条或多条消息

交错发送，接收方重组织

HTTP/2 连接都是永久的，而且仅需要每个来源一个连接

流控制：阻止发送方向接收方发送大量数据的机制

- HTTP/3- 草案

报文

Method

GET	请求一个指定资源的表示形式，使用 GET 请求应该只被用于获取数据
POST	用于将实体提交到指定的资源，通常导致服务器上的状态变化或副作用
PUT	用请求有效载荷替换目标资源的所有当前表示
DELETE	删除指定的资源
HEAD	请求一个与 GET 请求的响应相同的响应，但没有响应体
CONNECT	建立一个到由目标资源标识的服务器的隧道
OPTIONS	用于描述目标资源的通信选项
TRACE	沿着目标资源的路径执行一个消息环回测试
PATCH	用于对资源应用部分修改

特性

- **Safe**：不会修改服务器的数据的方法，如 GET , HEAD , OPTIONS
- **Idempotent**（幂等）：同样的请求被执行一次与连续执行多次的效果是一样的，服务器的状态也是一样的，所有 safe 的方法都是幂等的，如 GET , HEAD , OPTIONS , PUT , DELETE

状态码

用三位数字来表示，开头从 1~5，不同的开头有不同的含义

- **1xx**：指示信息，表示请求已接收，继续处理，一般很少见
- **2xx**：成功，表示请求已被成功接收、理解、接受
 - 200 OK —客户端请求成功
- **3xx**：重定向，要完成请求必须进行更进一步的操作
 - 301 —资源（网页等）被永久转移到其它URL
 - 302 —临时跳转
- **4xx**：客户端错误，请求有语法错误或请求无法实现
 - 401 Unauthorized —请求未经授权
 - 404 —请求资源不存在，可能是输入了错误的URL
- **5xx**：服务器端错误，服务器未能实现合法的请求
 - 500 —服务器内部发生了不可预期的错误
 - 504 Gateway Timeout —网关或者代理的服务器无法在规定的时间内获得想要的响应

RESTful APL

RESTful APL：一种API设计风格

REST—Representational State Transfer

设计规则

1. 每个URI代表一种资源
2. 客户端和服务端之间，传递这种资源的某种表现层
3. 客户端通过 HTTP method，对服务器端资源进行操作，实现“表现层状态转化”

请求	返回码	含义
GET/zoos	200 OK	列出所有动物园，服务器成功返回了
POST/zoos	201 CREATED	新建一个动物园，服务器创建成功
PUT/zoos/ID	400 INVALID REQUEST	更新某个指定动物园的信息（提供该动物园的全部信息） 用户发出的请求有错误，服务器没有进行新建或修改数据的操作
DELETE/zoos/ID	204 NO CONTENT	删除某个动物园，删除数据成功

常用请求头

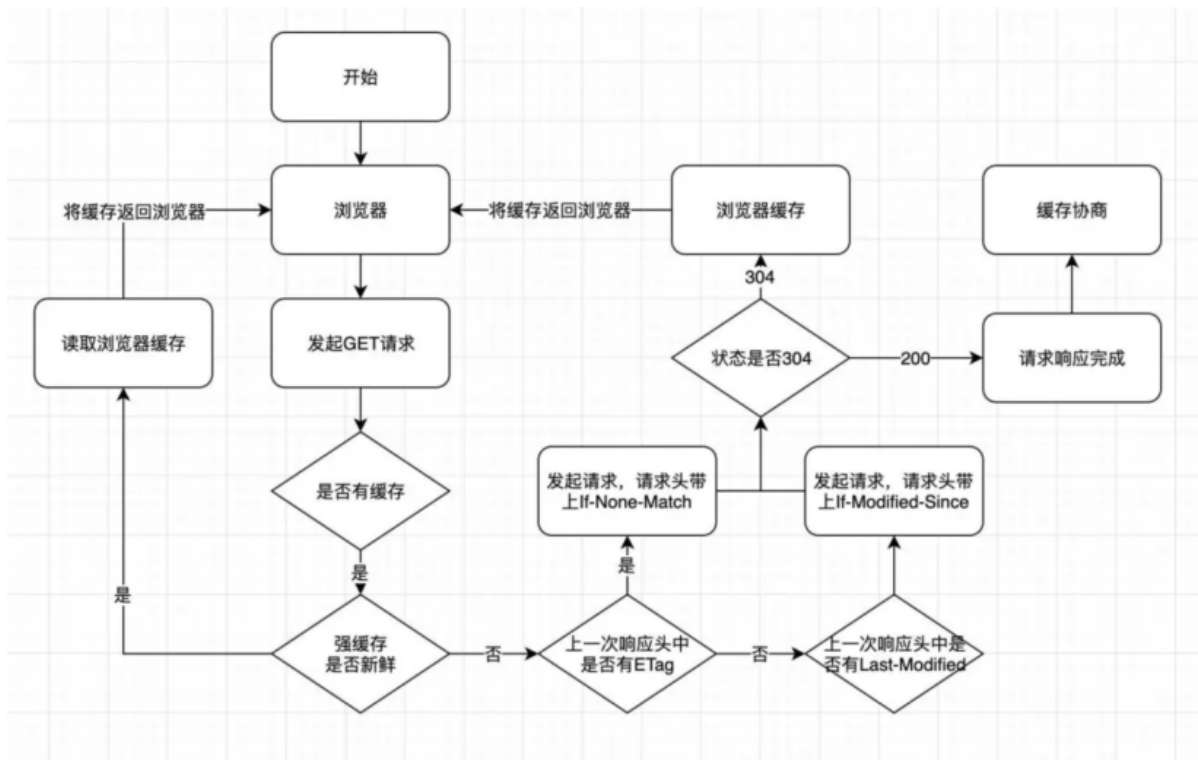
Accept	接收类型，表示浏览器支持的MIME类型（对标服务器返回的 Content-Type）
Content-Type	客户端发送出去实体内容的类型
Cache-Control	指定请求和响应遵循的缓存机制，如 no-cache
If-Modified-Since	对应服务端的 Last-Modified，用来匹配看文件是否变动，只能精确到1s之内
Expires	缓存控制，在这个时间内不会请求，直接使用缓存，服务端时间
Max-age	代表资源在本地缓存多少秒，有效期内不会请求而是使用缓冲
If-None-Match	对应服务端的 ETag，用来匹配文件内容是否改变（非常精准）
Cookie	有 cookie 并且同域访问时会自动带上
Referer	该页面的来源URL（适用于所有类型的请求，会精确到详细页面地址，csrf 拦截常用到这个字段）
Origin	最初的请求是从哪里发起的（只会精确到端口），Origin 比 Referer 更尊重隐私
User-Agent	用户客户端的一些必要信息，如 UA 头部等

常用响应头

Content-Type	服务端返回的实体内容的类型
Cache-Control	指定请求和响应遵循的缓存机制，如 no-cache
Last-Modified	请求资源的最后修改时间
Expires	应该在什么时候认为文档已经过期，从而不再缓存它
Max-age	客户端的本地资源应该缓存多少秒，开启了 Cache-Control 后有效
ETag	资源的特定版本的标识符，Etags 类似于指纹
Set-Cookie	设置和页面关联的 cookie，服务器通过这个头部把 cookie 传给客户端
Server	服务器的一些相关信息
Access-Control-Allow-Origin	服务器允许的请求 origin 头部（譬如为 *）

缓存

- 强缓存
 - Expires，时间戳
 - Cache-Control
 - 可缓存性
 - no-cache：协商缓存验证
 - no-store：不使用任何缓存
 - 到期
 - max-age：单位是秒，存储的最大周期，相对于请求的时间
 - 重新验证*重新加载
 - must-revalidate：一旦资源过期，在成功向原始服务器验证之前，不能使用
- 协商缓存
 - Etag/If-None-Match：资源的特定版本的标识符，类似于指纹
 - Last-Modified/If-Modified-Since：最后修改时间



cookie

Set-Cookie-response

Name=value	各种 cookie 的名称和值
Expires=Date	Cookie 的有效期，缺省时 cookie 仅在浏览器关闭之前有效
Path=Path	限制指定 cookie 的发送范围的文件目录，默认为当前
Domain=domain	限制 cookie 生效的域名，默认创建 cookie 的服务
secure	仅在 HTTP 安全连接时，才可以发送 cookie
HttpOnly	JavaScript 脚本无法获得 cookie
SameSite=[None Strict Lax]	None 同站、跨站请求都可以发送 Strict 仅在同站发送 允许与顶级导航一起发送，并将与第三方网站发起的 GET 请求一起发送

场景分析

2.1 静态资源

今日头条

1. 打开 chrome
2. 打开网站链接
3. 打开控制台
 - 右键 → 检查
 - F12

4. 切换至 network (网络)

状态码200也不一定就发起了请求

```
access-control-allow-origin: *
age: 2230335
ali-swift-global-savetime: 1625397577
cache-control: max-age=31536000
content-encoding: br
content-length: 26816
content-md5: Cf/SyvzWui5CSr8cu25TFQ==
content-type: text/css; charset=utf-8
date: Sun, 04 Jul 2021 11:19:37 GMT
eagleid: 276076a016276279126953122e
last-modified: Sun, 04 Jul 2021 11:11:19 GMT
server: Tengine
server-timing: cdn-cache;desc=HIT,edge;dur=1
timing-allow-origin: *
vary: Accept-Encoding
```

- 缓存策略
 - 强缓存
 - `Cache-control`: 一年
- 允许所有域名访问
- 资源类型: CSS

静态资源方案: 缓存+CDN+文件名hash

CDN: Content Delivery Network

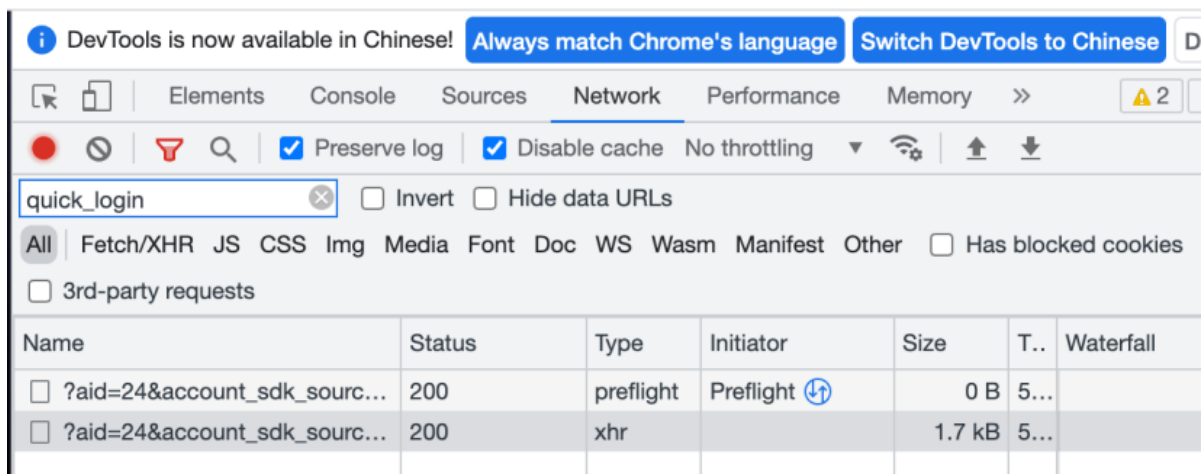
通过用户就近性和服务器负载的判断, CDN确保内容以一种极为高效的方式为用户的请求提供服务

2.2 登录

业务场景

- 表单登录
- 扫码登录

技术方式: SSO



- 账号密码登录
- 打开控制台 → network → 勾选 preserve log → 过滤 quick_login
- 观察请求



跨域, cross-origin 导致了 options 的请求

流程

1. 向什么地址做了上面动作
 - 使用 POST 方法
 - 目标域名 <https://sso.toutiao.com>
 - 目标 path/quick_login/v2/
2. 携带了哪些信息, 返回了哪些信息
 - 携带信息
 - Post body, 数据格式为 form
 - 希望获取的数据格式为 json
 - 已有的 cookie
 - 返回信息
 - 数据格式 json
 - 种 cookie 的信息

“same-origin” “cross-origin”

https:// www.example.com : 443

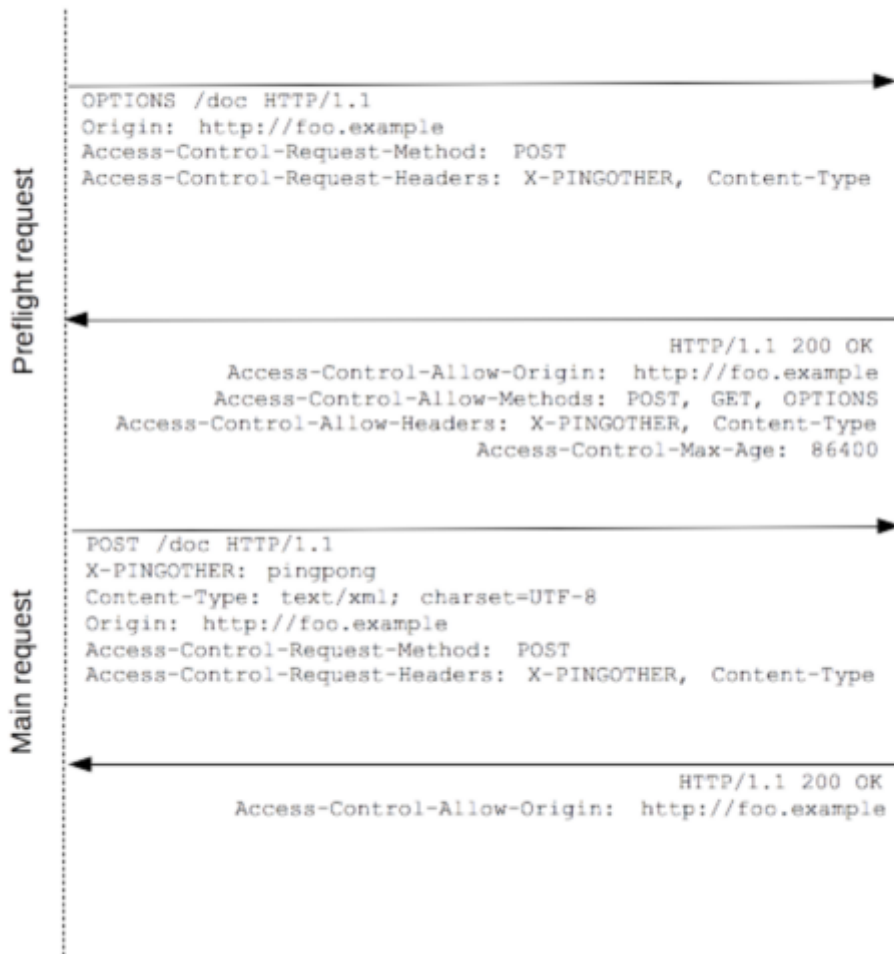
scheme host name port

Origin A	Origin B	
https://www.example.com:443	https://www.evil.com:443	cross-origin: different domains
	https://example.com:443	cross-origin: different subdomains
	https://login.example.com:443	cross-origin: different subdomains
	http://www.example.com:443	cross-origin: different schemes
	https://www.example.com:80	cross-origin: different ports
	https://www.example.com:443	same-origin: exact match
	https://www.example.com	same-origin: implicit port number (443) matches

- CORS (Cross-Origin Resource Sharing)
- 预请求：获取服务端是否允许该跨源请求（复杂请求）
- 相关协议头
 - Access-Control-Allow-Origin
 - Access-Control-Expose-Headers
 - Access-Control-Max-Age
 - Access-Control-Allow-Credentials
 - Access-Control-Allow-Methods
 - Access-Control-Allow-Headers
 - Access-Control-Request-Method
 - Access-Control-Request-Headers
 - Origin

Client

Server

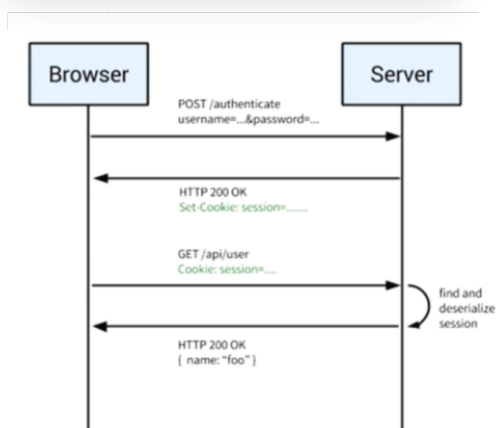


跨域解决方案

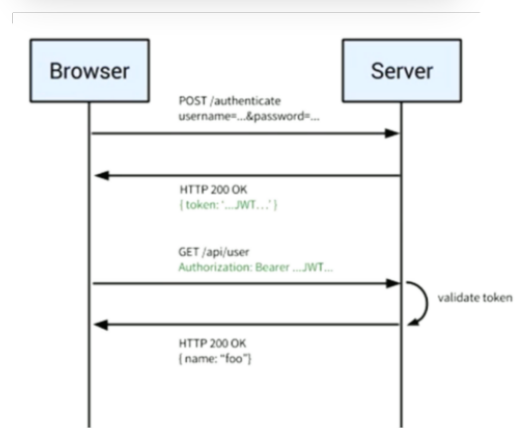
- 代理服务器
同源策略是浏览器的安全策略，不是HTTP的
- `Iframe`
诸多不便

鉴权

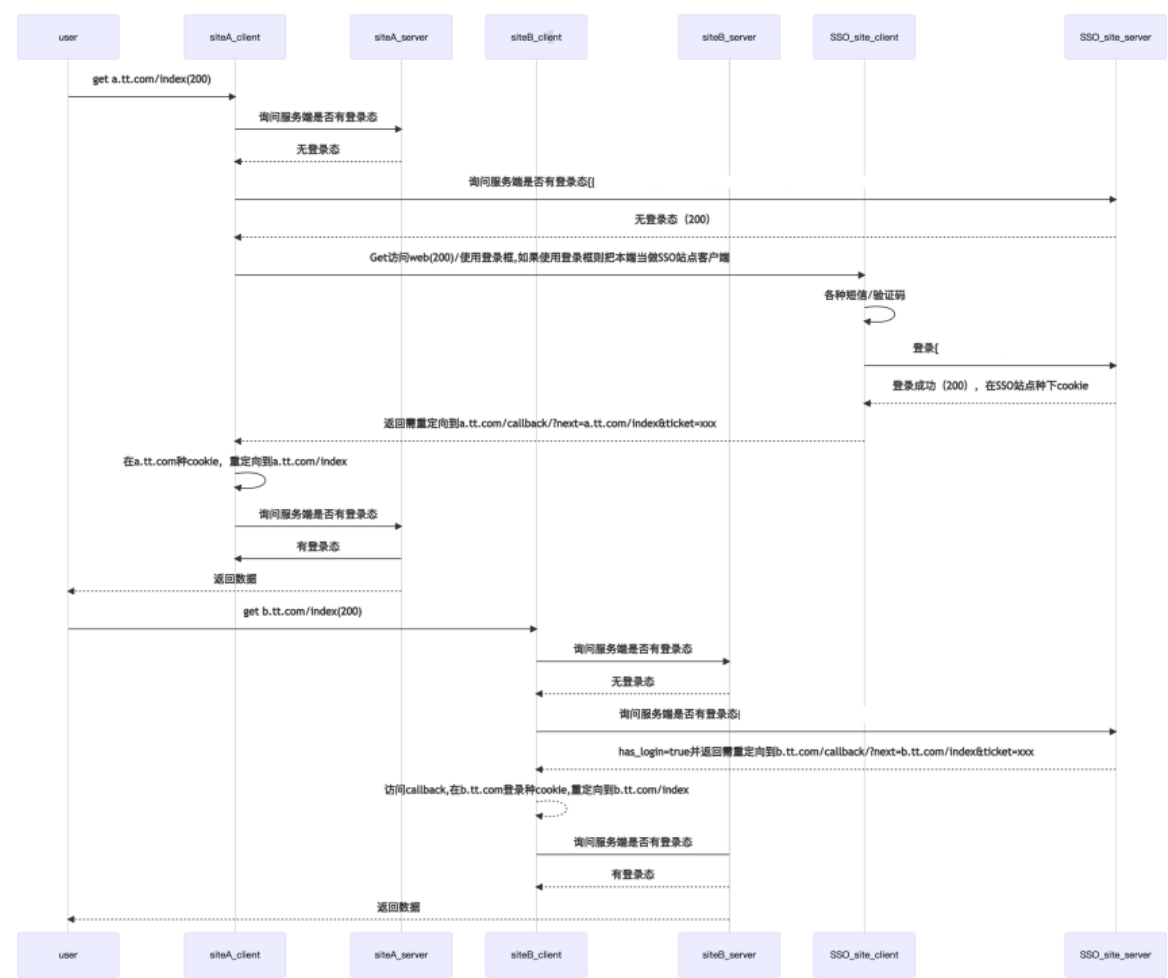
- Session + cookie



- JWT (JSON web token)



SSO-单点登录 (Single Sign On)



实战

3.1 浏览器篇

AJAX之XHR

XHR: XMLHttpRequest

readyState

0	UNSENT	代理被创建，但尚未调用 open() 方法
1	OPENED	open() 方法已经被调用
2	HEADERS_RECEIVED	send() 方法已经被调用，并且头部和状态已经可获得
3	LOADING	下载中；responseText 属性已经包含部分数据
4	DONE	下载操作已完成

```

function request(option) {
  if (String(option) !== '[object Object]') return undefined
  option.method = option.method ? option.method.toUpperCase() : 'GET'
  option.data = option.data || {}
  var formData = []
  for (var key in option.data) {
    formData.push(''.concat(key, '=', option.data[key]))
  }
  option.data = formData.join('&')

  if (option.method === 'GET') {
    option.url += location.search.length === 0 ? ''.concat('?', option.data) : ''.concat('&', option.data)
  }

  var xhr = new XMLHttpRequest()
  xhr.responseType = option.responseType || 'json'
  xhr.onreadystatechange = function () {
    if (xhr.readyState === 4) {
      if (xhr.status === 200) {
        if (option.success && typeof option.success === 'function') {
          option.success(xhr.response)
        }
      } else {
        if (option.error && typeof option.error === 'function') {
          option.error()
        }
      }
    }
  }
  xhr.open(option.method, option.url, true)
  if (option.method === 'POST') {
    xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded')
  }
  xhr.send(option.method === 'POST' ? option.data : null)
}

```

AJAX之Fetch

- XMLHttpRequest的升级版
- 使用Promise
- 模块化设计，Response，Request，Header对象
- 通过数据流处理对象，支持分块读取

```

postData('http://example.com/answer', {answer: 42})
  .then(data => console.log(data)) // JSON from `response.json()` call
  .catch(error => console.error(error))

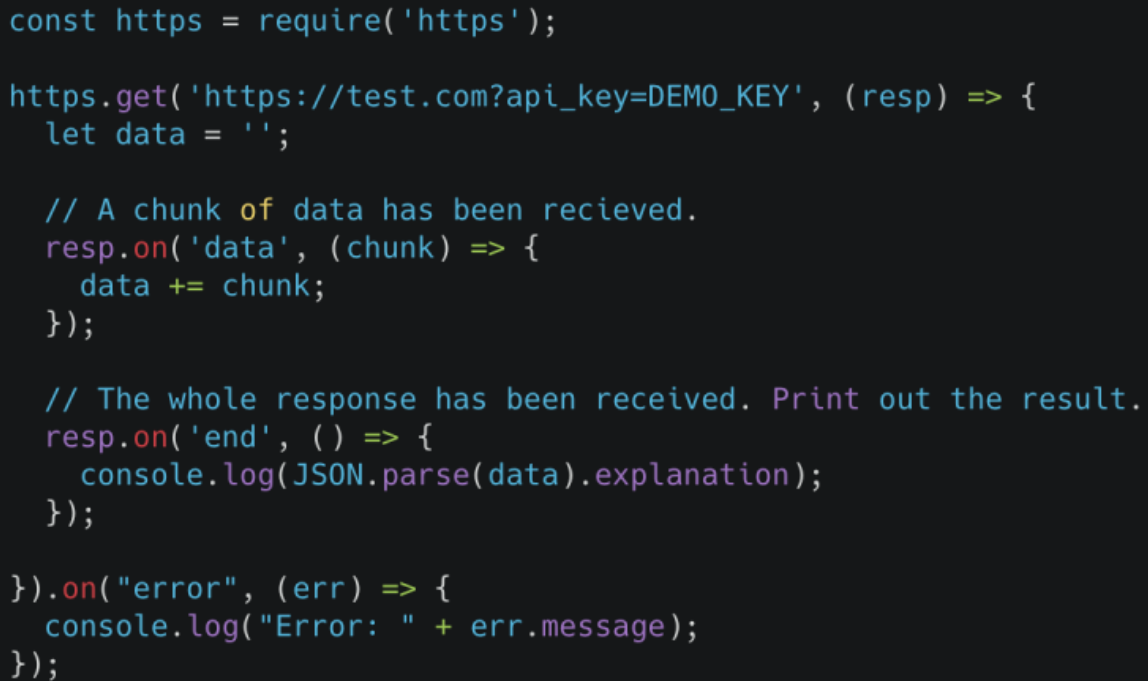
function postData(url, data) {
  // Default options are marked with *
  return fetch(url, {
    body: JSON.stringify(data), // must match 'Content-Type' header
    cache: 'no-cache', // *default, no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin', // include, same-origin, *omit
    headers: {
      'user-agent': 'Mozilla/4.0 MDN Example',
      'content-type': 'application/json'
    },
    method: 'POST', // *GET, POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors, cors, *same-origin
    redirect: 'follow', // manual, *follow, error
    referrer: 'no-referrer', // *client, no-referrer
  })
  .then(response => response.json()) // parses response to JSON
}

```

3.2 node篇

标准库: HTTP/HTTPS

- 默认模块, 无需安装其他依赖
- 功能有限/不是十分友好



```
const https = require('https');

https.get('https://test.com?api_key=DEMO_KEY', (resp) => {
  let data = '';

  // A chunk of data has been recieved.
  resp.on('data', (chunk) => {
    data += chunk;
  });

  // The whole response has been received. Print out the result.
  resp.on('end', () => {
    console.log(JSON.parse(data).explanation);
  });
}).on("error", (err) => {
  console.log("Error: " + err.message);
});
```

常用的请求库: axios

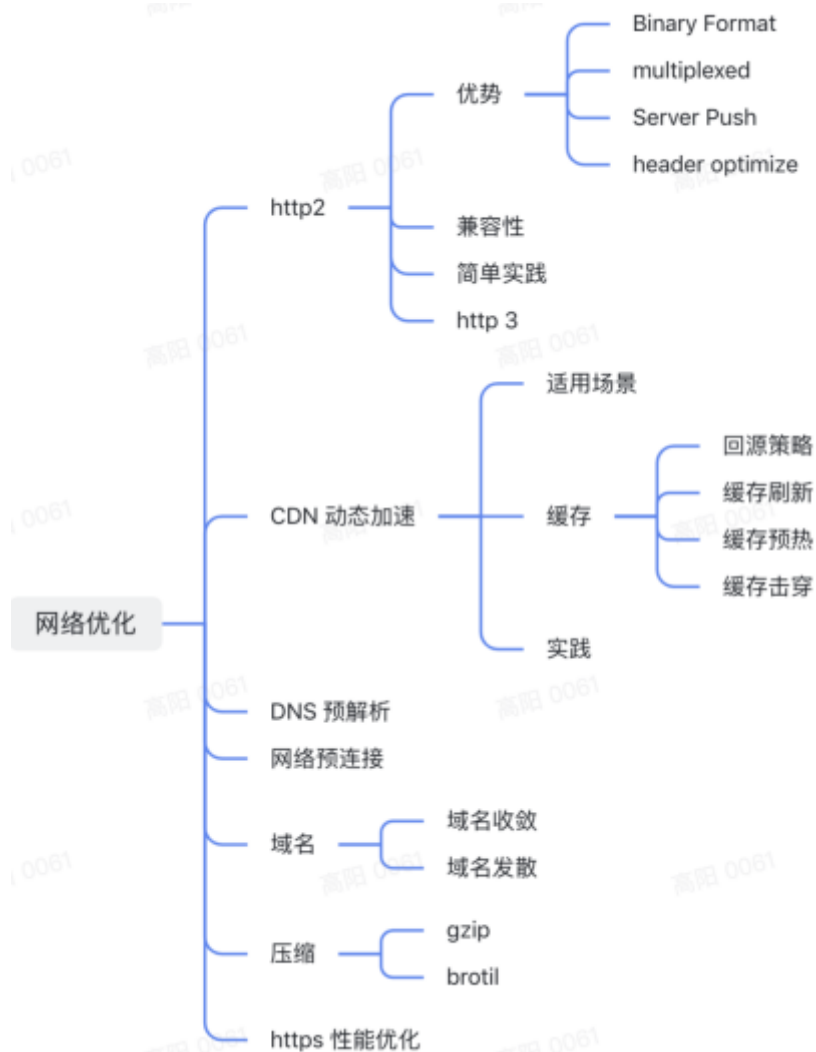
- 支持浏览器、`nodejs` 环境
- 丰富的拦截器

```
// 全局配置
axios.defaults.baseURL = "https://api.example.com";
// 添加请求拦截器
axios.interceptors.request.use(function (config) {
    // 在发送请求之前做点什么
    return config;
}, function (error) {
    // 对请求错误做点什么
    return Promise.reject(error);
});

// 发送请求
axios({
    method: 'get',
    url: 'http://test.com',
    responseType: 'stream'
}).then(function(response) {
    response.data.pipe(fs.createWriteStream('ada_lovelace.jpg'))
});
```

3.3 用户体验

网络优化



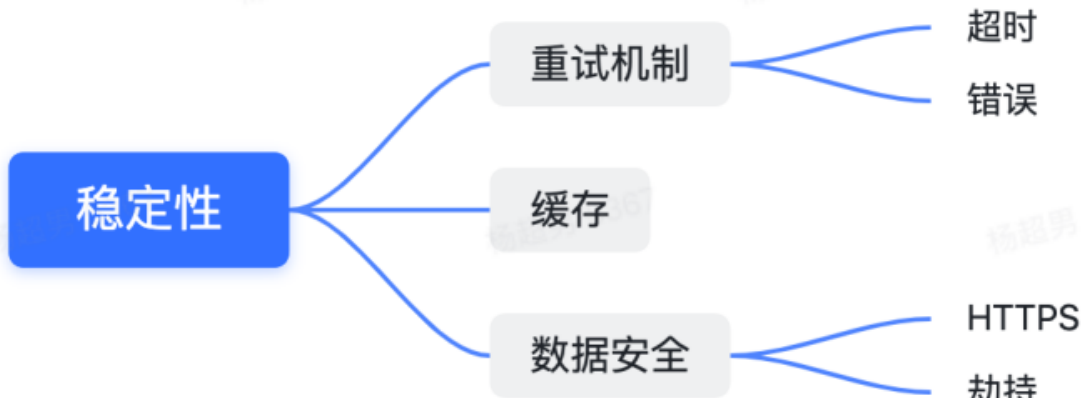
CDN开启H2的性能会更好

预解析、预连接

```

1 | <link rel="dns-prefetch" href="//example.com">
2 | <link rel="preconnect" href="//cdn.example.com" crossorigin>
  
```

稳定性



- 重试是保证稳定的有效手段，但要防止加剧恶劣情况

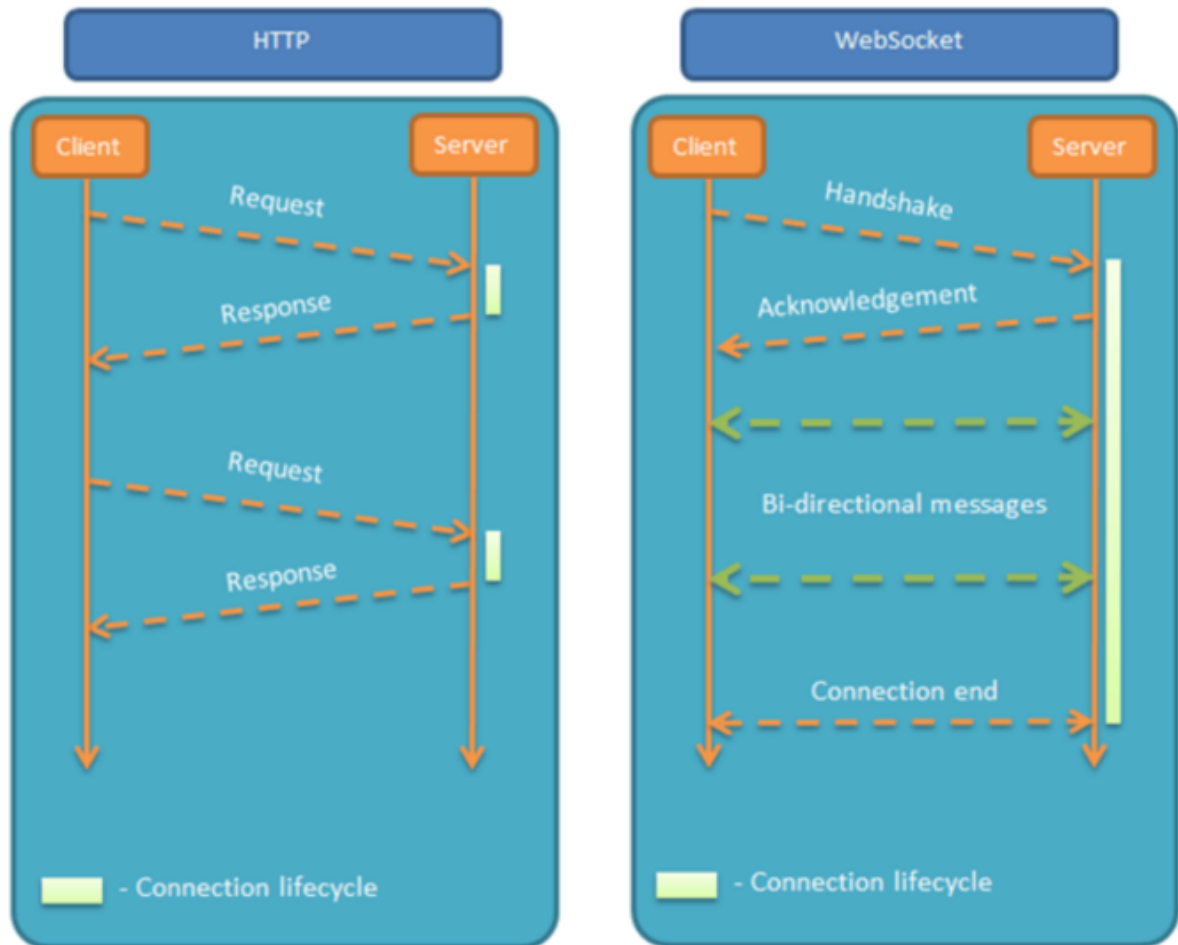
- 缓存合理使用，作为最后一道防线

3.4 扩展

通信方式

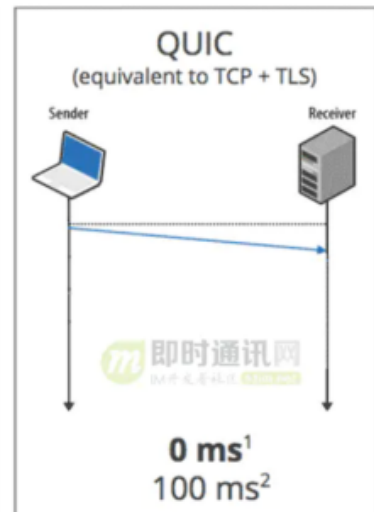
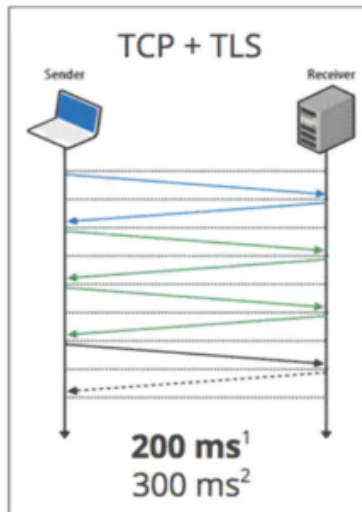
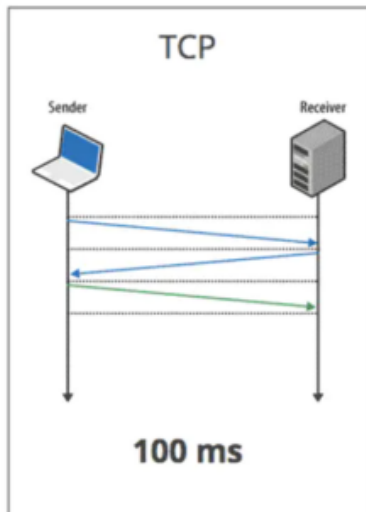
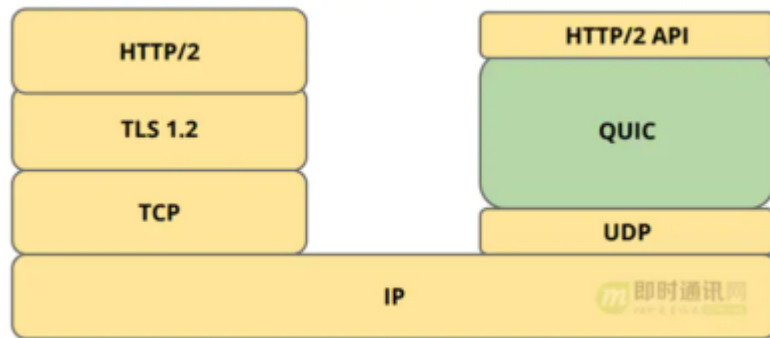
WebSocket

- 浏览器与服务器进行全双工通讯的网络技术
- 典型场景：实时性要求高，例如聊天室
- URL使用 `ws://` 或 `wss://` 等开头



QUIC (Quick UDP Internet Connection)

- 0-RTT 建联（首次建联除外）
- 类似TCP的可靠传输
- 类似TLS的加密传输，支持完美前向安全
- 用户空间的拥塞控制，最新的BBR算法
- 支持h2的基于流的多路复用，但没有TCP的HOL问题
- 前向纠错FEC
- 类似MPTCP的 `connection migration`



1. Repeat connection
2. Never talked to server before