对于现阶段HTML及浏览器知识的整理的知识的整理。

浏览器

Html基础

> Html语义化的了解?

html语义化让页面的内容结构化，结构更清晰，便于对浏览器、搜索引擎解析;

> 什么是 `<!DOCTYPE>`

DOCTYPE是html5标准网页声明，且必须声明在HTML文档的第一行。来告知浏览器的解析器用什么文档标准解析这个文档。

文档解析类型有：

- `BackCompat`：怪异模式，浏览器使用自己的怪异模式解析渲染页面。（如果没有声明DOCTYPE，默认就是这个模式）
- `CSS1Compat`：标准模式，浏览器使用W3C的标准解析渲染页面。

> 行内元素有哪些？块级元素有哪些？空元素有哪些?

行内元素：a b i em span input select strong等。

块级元素：h1~h6 p div ul ol dl li等。

空元素：meta link br hr img input area等。

> HTML中attribute和property的区别是什么?

- property是DOM中的属性，是JavaScript里的对象。
- attribute是HTML标签上的特性，它的值只能够是字符串。

事件机制

> 事件的触发过程是怎么样的?

事件触发有三个阶段：

- 事件捕获阶段
- 目标阶段
- 事件冒泡阶段

`window` 往事件触发处传播，遇到注册的捕获事件会触发。传播到事件触发处时触发注册的事件。从事件触发处往 `window` 传播，遇到注册的冒泡事件会触发。

事件触发一般来说会按照上面的顺序进行，但是也有特例，**如果给一个 body 中的子节点同时注册冒泡和捕获事件，事件触发会按照绑定的顺序执行。**

通常我们认为 `stopPropagation` 是用来阻止事件冒泡的，其实该函数也可以阻止捕获事件。

> DOM事件级别?

- DOM0 `element.onclick=function(){}`
- DOM2 `element.addEventListener('click', function(){},false)`
- DOM3 `element.addEventListener('keyup', function(){},false)` 事件类型增加很多，有鼠标，键盘事件

> 什么是事件代理?

如果一个节点中的子节点是动态生成的，那么子节点需要注册事件的话**应该注册在父节点上**

```
<ul id="ul">
  <li>1</li>
  <li>2</li>
  <li>3</li>
</ul>
<script>
  let ul = document.querySelector('#ul')
  ul.addEventListener('click', (event) => {
    console.log(event.target);
  })
</script>
```

优点：

- 节省内存
- 不需要给子节点注销事件

缺点:

- 基于冒泡，对于onfoucs和onblur等事件不支持
- 层级过多，冒泡过程中，可能会被某层阻止掉（建议就近委托）

存储

> 有几种方式可以实现存储功能，分别有什么优缺点?

cookie，localStorage，sessionStorage，indexDB

特性	cookie	localStorage	sessionStorage	indexDB
数据生命周期	一般由服务器生成，可以设置过期时间	除非被清理，否则一直存在	页面关闭就清理	除非被清理，否则一直存在
数据存储大小	4K	5M	5M	无限
与服务端通信	每次都会携带在 header 中，对于请求性能影响	不参与	不参与	不参与

> Cookie写不进去的情况?

- 服务端设置了http-only属性的Cookie，客户端JS无法读取，更别说更改了。
- 跨域的Cookie会存取失败（跨二级域名不包括在内）。
- 如果浏览器设置了**阻止网站设置任何数据**， 客户端无法接收Cookie，当然JS对Cookie的操作会失败。
- Cookie的数量超过最大限制，之前的Cookie被自动删除，JS无法读取到。
- Cookie过期被浏览器自动删除了。

浏览器渲染

> 从输入url到展示的过程?

- DNS 解析
- TCP 三次握手
- 发送请求，分析 url，设置请求报文(头，主体)
- 服务器返回请求的文件 (html)
- 浏览器渲染
  - 浏览器接收到 HTML 文件并转换为 DOM 树
    - 当数据转换为字符串以后，浏览器会先将这些字符串通过词法分析转换为**标记**（token），这一过程在词法分析中叫做**标记化**（tokenization）。
    - 字节数据 => 字符串 => Token => Node => DOM(CSSDOM)
  - 将 CSS 文件转换为 CSSOM 树
  - 结合 DOM 树与 CSSOM 树，生成渲染树
  - layout: 布局
  - GPU painting: 像素绘制页面
- TCP四次握手

> 插入几万个 DOM，如何实现页面不卡顿?

- 通过 `window.requestAnimationFrame` 的方式去循环的插入 DOM，

- 虚拟滚动，这种技术的原理就是只渲染可视区域内的内容，非可见区域的那就完全不渲染了，当用户在滚动的时候就实时去替换渲染的内容。

> 什么情况阻塞渲染?

- 首先渲染的前提是生成渲染树，所以 HTML 和 CSS 肯定会阻塞渲染。降低一开始需要渲染的文件大小，并且扁平层级，优化选择器。
- 当浏览器在解析到 `<script>` 标签时，会暂停构建 DOM，完成后才会从暂停的地方重新开始。
  - 将 `<script>` 标签放在 `body` 标签底部。
  - `<script>` 标签加上 `defer` 属性以后，表示该 JS 文件会并行下载，但是会放到 HTML 解析完成后顺序执行。
  - 对于没有任何依赖的 JS 文件可以加上 `async` 属性，表示 JS 文件下载和解析不会阻塞渲染。

通信

> 如何实现跨标签页通信?

- 通过父页面 `window.open()` 和子页面 `postMessage`
  - 异步下，通过 `window.open('about: blank')` 和 `tab.location.href = '*'`
- 设置同域下共享的 `localStorage` 与监听 `window.onstorage`
  - 重复写入相同的值无法触发
  - 会受到浏览器隐身模式等的限制
- 设置共享 `cookie` 与不断轮询脏检查(`setInterval`)
- 借助服务端或者中间层实现

EVENT LOOP

> 进程与线程区别？JS 单线程带来的好处?

进程描述了 CPU 在**运行指令及加载和保存上下文所需的时间**，放在应用上来说就代表了一个程序。

线程是进程中的更小单位，描述了执行一段指令所需的时间。

好处:

- 安全的渲染 UI
- 可以达到节省内存，节约上下文切换时间，没有锁的问题。

> 什么是执行栈?

可以把执行栈认为是一个存储函数调用的**栈结构**，遵循先进后出的原则。

当开始执行 JS 代码时，首先会执行一个 `main` 函数，然后执行代码。根据先进后出的原则，后执行的函数会先弹出栈。

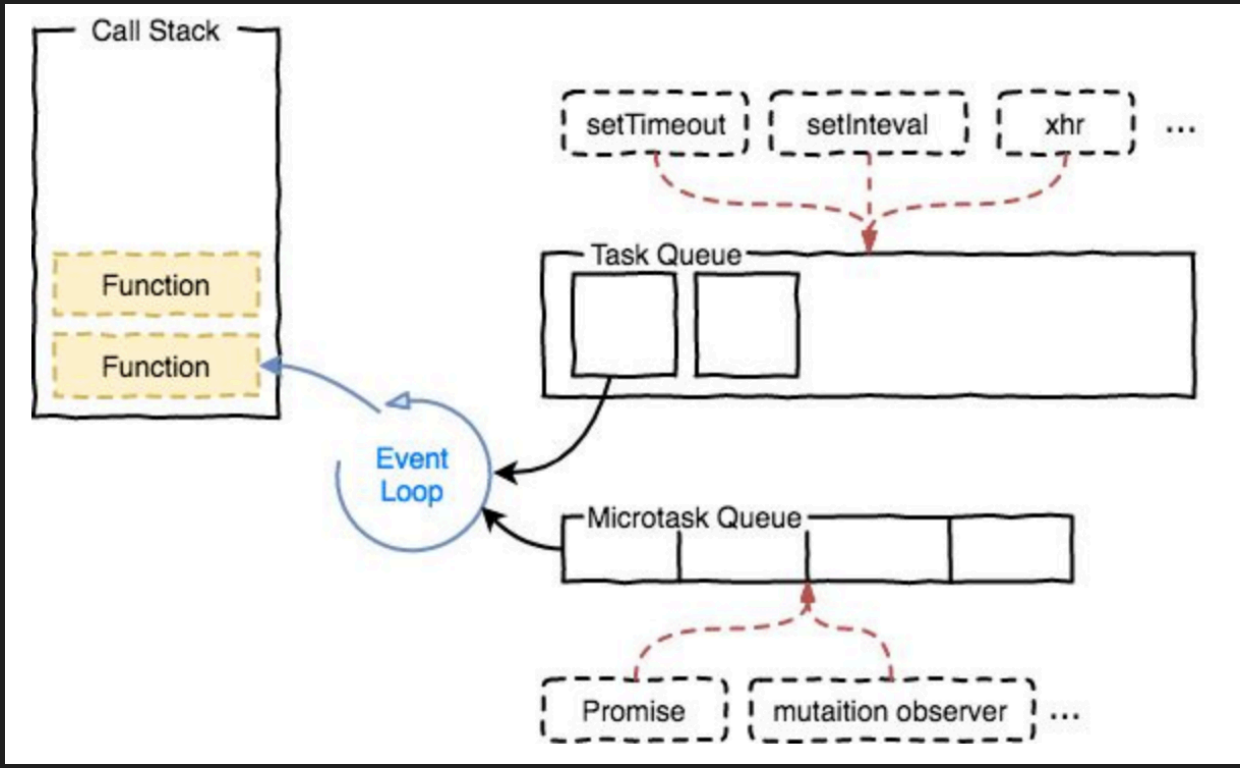
> 什么是微任务（microtask）和宏任务（macrotask）？

- **宏任务，macrotask，也叫tasks。** 一些异步任务的回调会依次进入macro task queue，等待后续被调用，这些异步任务包括：
  - setTimeout
  - setInterval
  - setImmediate (Node独有)
  - requestAnimationFrame (浏览器独有)
  - I/O
  - UI Rendering
- **微任务，microtask，也叫jobs。** 另一些异步任务的回调会依次进入micro task queue，等待后续被调用，这些异步任务包括：
  - process.nextTick (Node独有)
  - Promise
  - MutationObserver

> 异步代码执行顺序？什么是event loop?

当我们执行 JS 代码的时候其实就是往执行栈中放入函数，当遇到异步的代码时，会被**挂起**并在需要执行的时候加入到 Task（有多种 Task） 队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为。

主线程从task队列中读取事件，这个过程是循环不不断的，所以整个的这种运行机制又称为Event Loop(事件循环)



Event Loop执行顺序：

- 首先执行同步代码。这属于宏任务。
- 当执行完所有同步代码后，执行栈为空，查询是否有异步代码需要执行。

- 执行所有微任务。
- 当执行完所有微任务后，如有必要会渲染页面。
- 然后开始下一轮 Event Loop，执行宏任务中的异步代码，也就是 `setTimeout` 中的回调函数

另一种说法：

- 执行一个宏任务（栈中没有就从事件队列中获取）
- 执行过程中如果遇到微任务，就将它添加到微任务的任务队列中
- 宏任务执行完毕后，立即执行当前微任务队列中的所有微任务（依次执行）
- 当前宏任务执行完毕，开始检查渲染，然后GUI线程接管渲染
- 渲染完毕后，JS线程继续接管，开始下一个宏任务（从事件队列中获取）

事件循环是指: 执行一个宏任务，然后执行清空微任务列表，循环再执行宏任务，再清微任务列表

我们只需记住**当前执行栈执行完毕时会立刻先处理所有微任务队列中的事件，然后再去宏任务队列中取出一个事件。同一次事件循环中，微任务永远在宏任务之前执行**。

https://github.com/Advanced-Frontend/Daily-Interview-Question/issues/7

重绘&回流

> 什么是重绘和回流?

当元素的样式发生变化时，浏览器需要触发更新，重新绘制元素。这个过程中，有两种类型的操作，即重绘与回流

- **重绘(repaint)**: 当元素样式的改变不影响布局时，浏览器将使用重绘对元素进行更新，此时由于只需要UI层面的重新像素绘制，因此 损耗较少。
- **回流(reflow)**: 当元素的尺寸、结构或触发某些属性时，浏览器会重新渲染页面，称为回流。此时，浏览器需要重新经过计算，计算后还需要重新页面布局，因此是较重的操作。会触发回流的操作:
    - 页面初次渲染
    - 浏览器窗口大小改变
    - 元素尺寸、位置、内容发生改变
    - 元素字体大小变化
    - 添加或者删除可见的 dom 元素
    - 激活 CSS 伪类（例如：:hover）
    - 查询某些属性或调用某些方法
        - clientWidth、clientHeight、clientTop、clientLeft
        - offsetWidth、offsetHeight、offsetTop、offsetLeft
        - scrollWidth、scrollHeight、scrollTop、scrollLeft
        - getComputedStyle()
        - getBoundingClientRect()
        - scrollTo()

回流必定触发重绘，重绘不一定触发回流。重绘的开销较小，回流的代价较高。

> 如何减少重绘和回流?

- 使用 `transform` 替代 `top`
- 使用 `visibility` 替换 `display: none` ，因为前者只会引起重绘，后者会引发回流（改变了布局）
- 不要把节点的属性值放在一个循环里当成循环里的变量
- 不要使用 `table` 布局，可能很小的一个小改动会造成整个 `table` 的重新布局
- 动画实现的速度的选择，动画速度越快，回流次数越多，也可以选择使用 `requestAnimationFrame`
- CSS 选择符从右往左匹配查找，避免节点层级过多
- 将频繁重绘或者回流的节点设置为图层，图层能够阻止该节点的渲染行为影响别的节点。
    - `will-change`
    - `video` 、 `iframe` 标签

垃圾回收

> V8垃圾回收机制?

将内存中不再使用的数据进行清理，释放出内存空间。V8 将内存分成 **新生代空间** 和 **老生代空间** 。

- **新生代空间**: 用于存活较短的对象
    - 又分成两个空间: from 空间 与 to 空间
    - Scavenge GC算法: 当 from 空间被占满时，启动 GC 算法
        - 存活的对象从 from space 转移到 to space
        - 清空 from space
        - from space 与 to space 互换
        - 完成一次新生代GC
- **老生代空间**: 用于存活时间较长的对象
    - 从 新生代空间 转移到 老生代空间 的条件
        - 经历过一次以上 Scavenge GC 的对象
        - 当 to space 体积超过25%
    - **标记清除算法**: 标记存活的对象，未被标记的则被释放
        - 经历过一次以上 Scavenge GC 的对象
        - 当 to space 体积超过25%
    - **压缩算法**: 将内存中清除后导致的碎片化对象往内存堆的一端移动，解决 **内存的碎片化**

内存泄漏

> 什么情况下会有内存泄漏?

- 意外的**全局变量**: 无法被回收
- **定时器**: 未被正确关闭，导致所引用的外部变量无法被释放

- **事件监听**: 没有正确销毁 (低版本浏览器可能出现)
- **闭包**: 会导致父级中的变量无法被释放
- **dom 引用**: dom 元素被删除时，内存中的引用未被正确清空

Http

Http/Https协议

> Http和Https的区别?

1. HTTP 的URL 以http:// 开头，而HTTPS 的URL 以https:// 开头
2. HTTP 是不安全的，而 HTTPS 是安全的
3. HTTP 标准端口是80 ，而 HTTPS 的标准端口是443
4. 在OSI 网络模型中，HTTP工作于应用层，而HTTPS 的安全传输机制工作在传输层
5. HTTP 无法加密，而HTTPS 对传输的数据进行加密
6. HTTP无需证书，而HTTPS 需要CA机构wosign的颁发的SSL证书

> URI和URL的区别?

- URI，是uniform resource identifier，统一资源标识符，用来唯一的标识一个资源。
- URL是uniform resource locator，统一资源定位器，它是一种具体的URI，即URL可以用来标识一个资源，而且还指明了如何locate这个资源。

> Http请求方法有哪些?

- GET:获取资源
- POST:传输资源
- PUT:更更新资源
- DELETE:删除资源
- HEAD:获得报文文首首部

> POST和GET的区别？

	GET	POST
后退按钮/刷新	无害	数据会被重新提交（浏览器应该告知用户数据会被重新提交）。
书签	可收藏为书签	不可收藏为书签
缓存	能被缓存	不能缓存
编码类型	application/x-www-form-urlencoded	application/x-www-form-urlencoded 或 multipart/form-data。为二进制数据使用多重编码。
历史	参数保留在浏览器历史中。	参数不会保存在浏览器历史中。
对数据长度的限制	是的。当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。	无限制。
对数据类型的限制	只允许 ASCII 字符。	没有限制。也允许二进制数据。
安全性	与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。在发送密码或其他敏感信息时绝不要使用 GET ！	POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。
可见性	数据在 URL 中对所有人都是可见的。	数据不会显示在 URL 中。

> 谈谈Http1.1和http2.0

- http1.1
  - **缓存处理**，在HTTP1.0中主要使用header里的If-Modified-Since,Expires来做为缓存判断的标准，HTTP1.1则引入了更多的缓存控制策略例如Entity tag，If-Unmodified-Since, If-Match, If-None-Match等更多可供选择的缓存头来控制缓存策略。
  - **带宽优化及网络连接的使用**，HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了，并且不支持断点续传功能，HTTP1.1则在请求头引入了range头域，它允许只请求资源的某个部分，即返回码是206（Partial Content），这样就方便了开发者自由的选择以便于充分利用带宽和连接。
  - **错误通知的管理**，在HTTP1.1中新增了24个错误状态响应码，如409（Conflict）表示请求的资源与资源的当前状态发生冲突；410（Gone）表示服务器上的某个资源被永久性的删除。
  - **Host头处理**，在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发展，在一台物理服务器上

可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。
    - **长连接**，HTTP 1.1支持长连接（PersistentConnection）和请求的流水线（Pipelining）处理，在一个TCP连接上可以传送多个HTTP请求和响应，减少了建立和关闭连接的消耗和延迟，在HTTP1.1中默认开启Connection： keep-alive，一定程度上弥补了HTTP1.0每次请求都要创建连接的缺点。
- http2.0
    - **新的二进制格式**（Binary Format），HTTP1.x的解析是基于文本。基于文本协议的格式解析存在天然缺陷，文本的表现形式有多样性，要做到健壮性考虑的场景必然很多，二进制则不同，只认0和1的组合。基于这种考虑HTTP2.0的协议解析决定采用二进制格式，实现方便且健壮。
    - **多路复用**（MultiPlexing），即连接共享，即每一个request都是是用作连接共享机制的。一个request对应一个id，这样一个连接上可以有多个request，每个连接的request可以随机的混杂在一起，接收方可以根据request的 id将request再归属到各自不同的服务端请求里面。
    - **header压缩**，如上文中所言，对前面提到过HTTP1.x的header带有大量信息，而且每次都要重复发送，HTTP2.0使用encoder来减少需要传输的header大小，通讯双方各自cache一份header fields表，既避免了重复header的传输，又减小了需要传输的大小。
    - **服务端推送**（server push），同SPDY一样，HTTP2.0也具有server push功能。

> 简单讲解一下 HTTP2 的多路复用?

在 HTTP/1 中，每次请求都会建立一次TCP连接，也就是我们常说的3次握手4次挥手，这在一次请求过程中占用了相当长的时间，即使开启了 Keep-Alive ，解决了多次连接的问题，但是依然有两个效率上的问题：

- 第一个：串行的文件传输。当请求a文件时，b文件只能等待，等待a连接到服务器、服务器处理文件、服务器返回文件，这三个步骤。我们假设这三步用时都是1秒，那么a文件用时为3秒，b文件传输完成用时为6秒，依此类推。（注：此项计算有一个前提条件，就是浏览器和服务器是单通道传输）
- 第二个：连接数过多。我们假设Apache设置了最大并发数为300，因为浏览器限制，浏览器发起的最大请求数为6（Chrome），也就是服务器能承载的最高并发为50，当第51个人访问时，就需要等待前面某个请求处理完成。

HTTP2采用二进制格式传输，取代了HTTP1.x的文本格式，二进制格式解析更高效。 多路复用代替了HTTP1.x的序列和阻塞机制，所有的相同域名请求都通过同一个TCP连接并发完成。在HTTP1.x中，并发多个请求需要多个TCP连接，浏览器为了控制资源会有6-8个TCP连接都限制。 HTTP2中

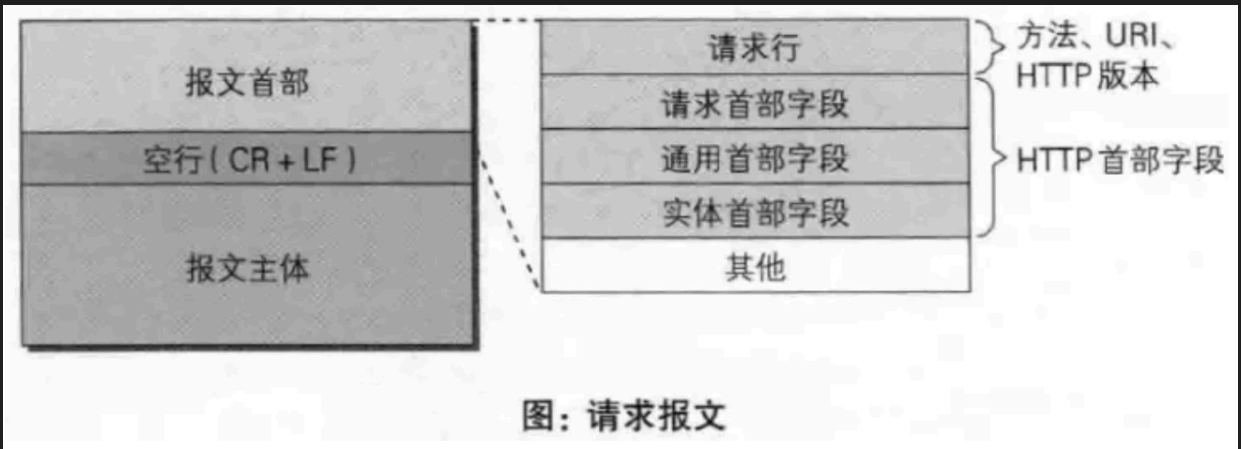
- 同域名下所有通信都在单个连接上完成，消除了因多个 TCP 连接而带来的延时和内存消耗。
- 单个连接上可以并行交错的请求和响应，之间互不干扰

请求报文与响应报文格式

> HTTP请求报文与响应报文格式?

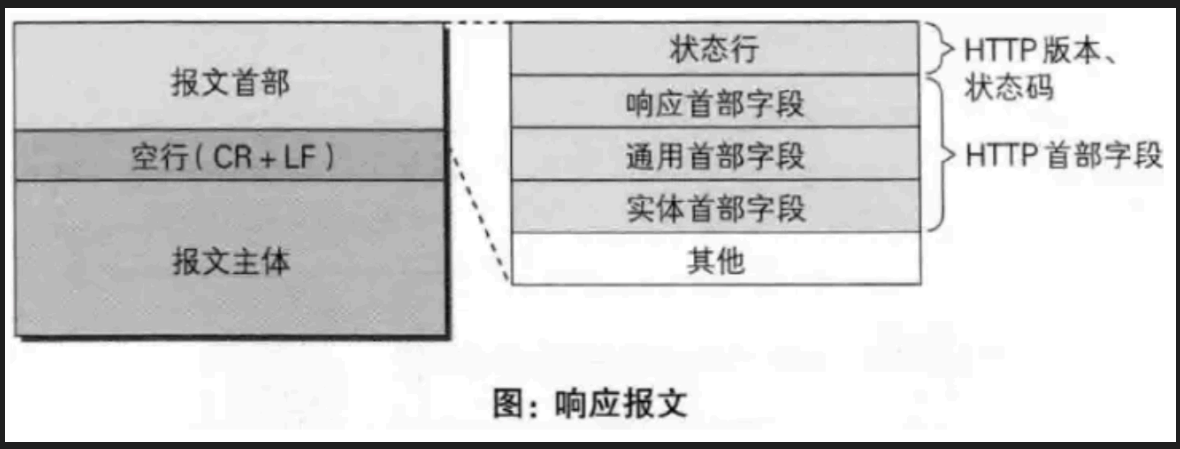
- 请求报文：
    - 请求行：包含请求方法、URI、HTTP版本信息

- 请求首部字段
- 请求内容实体
- 空行



图：请求报文

- 相应报文：
  - 状态行：包含HTTP版本、状态码、状态码的原因短语
  - 响应首部字段
  - 响应内容实体
  - 空行



图：响应报文

常见的首部：

- 通用首部字段（请求报文与响应报文都会使用的首部字段）
  - Date：创建报文时间
  - Connection：连接的管理
  - Cache-Control：缓存的控制
  - Transfer-Encoding：报文主体的传输编码方式
- 请求首部字段（请求报文会使用的首部字段）
  - Host：请求资源所在服务器
  - Accept：可处理的媒体类型
  - Accept-Charset：可接收的字符集
  - Accept-Encoding：可接受的内容编码
  - Accept-Language：可接受的自然语言
- 响应首部字段（响应报文会使用的首部字段）
  - Accept-Ranges：可接受的字节范围
  - Location：令客户端重新定向到的URI

    - Server：HTTP服务器的安装信息
- 实体首部字段（请求报文与响应报文的的实体部分使用的首部字段）
    - Allow：资源可支持的HTTP方法
    - Content-Type：实体主类的类型
    - Content-Encoding：实体主体适用的编码方式
    - Content-Language：实体主体的自然语言
    - Content-Length：实体主体的的字节数
    - Content-Range：实体主体的位置范围，一般用于发出部分请求时使用

TCP

> tcp属于哪一层?

- tcp属于哪一层（1 物理层 -> 2 数据链路层 -> 3 网络层(ip)-> 4 传输层(tcp) -> 5 应用层(http)）

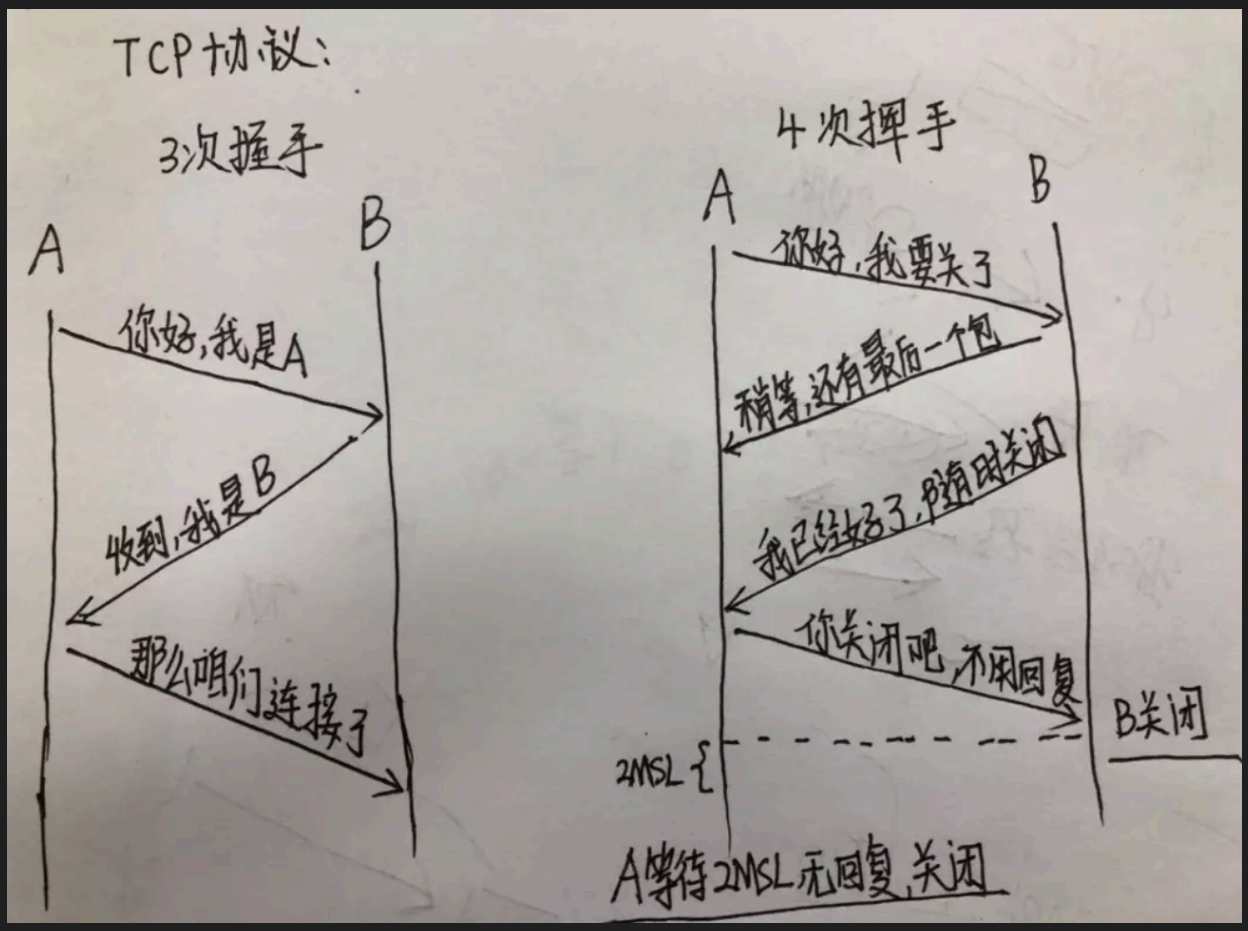
> TCP三次握手?

- 客户端发送 syn(同步序列编号) 请求，进入 syn_send 状态，等待确认
- 服务端接收并确认 syn 包后发送 syn+ack 包，进入 syn_recv 状态
- 客户端接收 syn+ack 包后，发送 ack 包，双方进入 established 状态

> TCP四次握手?

- 客户端 -- FIN --> 服务端， FIN—WAIT
- 服务端 -- ACK --> 客户端， CLOSE-WAIT
- 服务端 -- ACK,FIN --> 客户端， LAST-ACK
- 客户端 -- ACK --> 服务端，CLOSED

> TCP握手图示?



状态码

> 常见状态码?

- 2xx 成功
  - 200 OK，表示从客户端发来的请求在服务器端被正确处理
  - 204 No content，表示请求成功，但响应报文不含实体的主体部分
  - 205 Reset Content，表示请求成功，但响应报文不含实体的主体部分，但是与 204 响应不同在于要求请求方重置内容
  - 206 Partial Content，进行范围请求
- 3xx 重定向
  - 301 moved permanently，永久性重定向，表示资源已被分配了新的 URL
  - 302 found，临时性重定向，表示资源临时被分配了新的 URL
  - 303 see other，表示资源存在着另一个 URL，应使用 GET 方法获取资源
  - 304 not modified，表示服务器允许访问资源，但因发生请求未满足条件的情况
  - 307 temporary redirect，临时重定向，和302含义类似，但是期望客户端保持请求方法不变向新的地址发出请求
- 4xx 客户端错误
  - 400 bad request，请求报文存在语法错误
  - 401 unauthorized，表示发送的请求需要有通过 HTTP 认证的认证信息
  - 403 forbidden，表示对请求资源的访问被服务器拒绝
  - 404 not found，表示在服务器上没有找到请求的资源
- 5xx 服务器错误

- 500 internal sever error，表示服务器端在执行请求时发生了错误
- 501 Not Implemented，表示服务器不支持当前请求所需要的某个功能
- 502 Bad Gateway：作为网关或者代理工作的服务器尝试执行请求时，从上游服务器接收到无效的响应
- 503 service unavailable，表明服务器暂时处于超负载或正在停机维护，无法处理请求
- 504 Gateway Timeout：服务器响应超时

缓存策略

> 浏览器中的缓存策略?

通常浏览器缓存策略分为两种：**强缓存**和**协商缓存**，并且缓存策略都是通过设置 HTTP Header 来实现的。

- 强缓存，强缓存可以通过设置两种 HTTP Header 实现：`Expires` 和 `Cache-Control` 。强缓存表示在缓存期间不需要请求，`state code` 为 200。
  - Expires

    ```
    Expires: Wed, 22 Oct 2018 08:41:00 GMT
    ```

    `Expires` 是 HTTP/1 的产物，表示资源会在 `Wed, 22 Oct 2018 08:41:00 GMT` 后过期，需要再次请求。并且 `Expires` **受限于本地时间**，如果修改了本地时间，可能会造成缓存失效。
  - Cache-control

    ```
    Cache-control: max-age=30
    ```

    `Cache-Control` 出现于 HTTP/1.1，**优先级高于 Expires** 。该属性值表示资源会在 30 秒后过期，需要再次请求。`Cache-Control` **可以在请求头或者响应头中设置**，并且可以组合使用多种指令。

指令	作用
public	表示响应可以被客户端和代理服务器缓存
private	表示响应只可以被客户端缓存
max-age=30	缓存 30 秒后就过期，需要重新请求
s-maxage=30	覆盖 max-age，作用一样，只在代理服务器中生效
no-store	不缓存任何响应
no-cache	资源被缓存，但是立即失效，下次会发起请求验证资源是否过期
max-stale=30	30 秒内，即使缓存过期，也使用该缓存
min-fresh=30	希望在 30 秒内获取最新的响应

- 协商缓存，如果缓存过期了，就需要发起请求验证资源是否有更新。协商缓存可以通过设置两种 HTTP Header 实现：`Last-Modified` 和 `ETag` 。当浏览器发起请求验证资源时，如果资源没有做改变，那么服务端就会返回 304 状态码，并且更新浏览器缓存有效期。
  - Last-Modified
    - `Last-Modified` 表示本地文件最后修改日期，`If-Modified-Since` 会将 `Last-Modified` 的值发送给服务器，询问服务器在该日期后资源是否有更新，有更新的话就会将新的资源发送回来，否则返回 304 状态码。
    - 缺点：
      - 如果本地打开缓存文件，即使没有对文件进行修改，但还是会造成 `Last-Modified` 被修改，服务端不能命中缓存导致发送相同的资源
      - 因为 `Last-Modified` 只能以秒计时，如果在不可感知的时间内修改完成文件，那么服务端会认为资源还是命中了，不会返回正确的资源
  - Etag
    - `ETag` 类似于文件指纹，`If-None-Match` 会将当前 `ETag` 发送给服务器，询问该资源 `ETag` 是否变动，有变动的话就将新的资源发送回来。并且 `ETag` 优先级比 `Last-Modified` 高。

> 如果什么缓存策略都没设置，那么浏览器会怎么处理?

对于这种情况，浏览器会采用一个启发式的算法，通常会取响应头中的 `Date` 减去 `Last-Modified` 值的 10% 作为缓存时间。

> 实际场景应用缓存策略?

- 频繁变动的资源
  - 对于频繁变动的资源，首先需要使用 `Cache-Control: no-cache` 使浏览器每次都请求服务器，然后配合 `ETag` 或者 `Last-Modified` 来验证资源是否有效。这样的做法虽然不能节省请求数量，但是能显著减少响应数据大小。
- 代码文件
  - 这里特指除了 HTML 外的代码文件，因为 HTML 文件一般不缓存或者缓存时间很短。

    一般来说，现在都会使用工具来打包代码，那么我们就可以对文件名进行哈希处理，只有当代码修改后才会生成新的文件名。基于此，我们就可以给代码文件设置缓存有效期一年 `Cache-Control: max-age=31536000` ，这样只有当 HTML 文件中引入的文件名发生了改变才会去下载最新的代码文件，否则就一直使用缓存。

跨域

> 什么是跨域？为什么浏览器要使用同源策略？你有几种方式可以解决跨域问题？了解预检请求嘛?

- 因为浏览器出于安全考虑，有同源策略。也就是说，如果协议、域名或者端口有一个不同就是跨域。
- 同源政策的目的，是为了保证用户信息的安全，防止恶意的网站窃取数据。主要是用来防止 CSRF 攻击的。

- 跨域的解决方式
  - Jsonp：利用 `<script>` 标签不受跨域限制的特点，缺点是只能支持 get 请求
    - 封装一个 JSONP

```
function jsonp(url, jsonpCallback, success) {
  let script = document.createElement('script')
  script.src = url
  script.async = true
  script.type = 'text/javascript'
  window[jsonpCallback] = function(data) {
    success && success(data)
  }
  document.body.appendChild(script)
}
jsonp('http://xxx', 'callback', function(value) {
  console.log(value)
})
```

  - CORS：服务端设置 `Access-Control-Allow-Origin: *` 就可以开启 CORS。该属性表示哪些域名可以访问资源，如果设置通配符则表示所有网站都可以访问资源。
  - document.domain：该方式只能用于**二级域名相同**的情况下，比如 `a.test.com` 和 `b.test.com` 适用于该方式。只需要给页面添加 `document.domain = 'test.com'` 表示二级域名都相同就可以实现跨域
  - postMessage

```
// 发送消息端
window.parent.postMessage('message', 'http://test.com')
// 接收消息端
var mc = new MessageChannel()
mc.addEventListener('message', event => {
  var origin = event.origin || event.originalEvent.origin
  if (origin === 'http://test.com') {
    console.log('验证通过')
  }
})
```