从输入 `URL` 到看到页面发生了什么?

原文链接

`URL` 的输入到浏览器解析的一系列事件

很多大公司面试喜欢问这样一道面试题，输入 `URL` 到看见页面发生了什么？,今天我们来总结一下。简单来说，共有以下几个过程

- `DNS` 解析
- 发起 `TCP` 连接
- 发送 `HTTP` 请求
- 服务器处理请求并返回 `HTTP` 报文
- 浏览器解析渲染页面
- 连接结束。

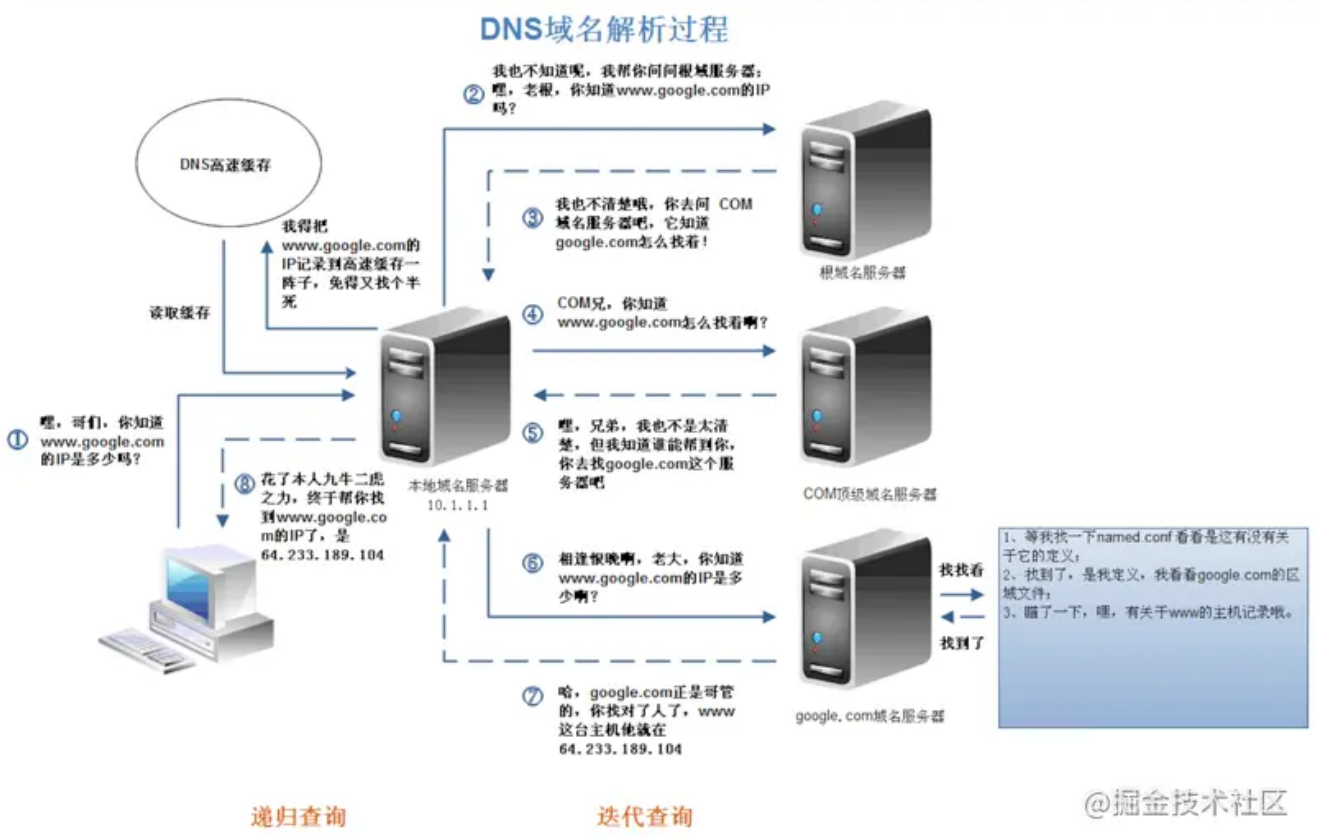
下面我们来看看具体的细节

1. `DNS` 解析

`DNS` 解析实际上就是寻找你所需要的资源的过程。假设你输入 `www.baidu.com`，而这个网址并不是百度的真实地址，互联网中每一台机器都有唯一标识的IP地址，这个才是关键，但是它不好记，乱七八糟一串数字谁记得住啊，所以就需要一个网址和IP地址的转换，也就是 `DNS` 解析。下面看看具体的解析过程

具体解析

`DNS` 解析其实是一个递归的过程



输入 `www.google.com` 网址后，首先在本地的域名服务器中查找，没找到去根域名服务器查找，没有再去 `com` 顶级域名服务器查找，如此的类推下去，直到找到 `IP` 地址，然后把它记录在本地，供下次使用。大致过程就是 `. -> .com -> google.com. -> www.google.com.` 。(你可能觉得我多写 `.`，并木有，这个 `.` 对应的就是根域名服务器，默认情况下所有的网址的最后一位都是 `.`，既然是默认情况下，为了方便用户，通常都会省略，浏览器在请求 `DNS` 的时候会自动加上)

DNS优化

既然已经懂得了解析的具体过程，我们可以看到上述一共经过了 `N` 个过程，每个过程有一定的消耗和时间的等待，因此我们得想办法解决一下这个问题！

DNS缓存

`DNS` 存在着多级缓存，从离浏览器的距离排序的话，有以下几种: 浏览器缓存，系统缓存，路由器缓存， `IPS` 服务器缓存，根域名服务器缓存，顶级域名服务器缓存，主域名服务器缓存。

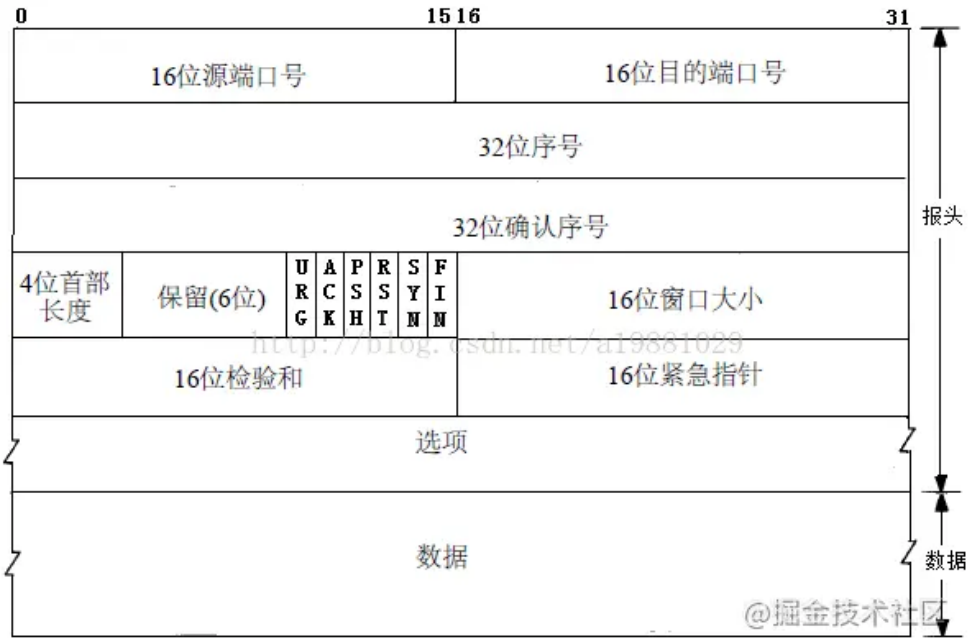
- 在你的 `chrome` 浏览器中输入 `chrome://net-internals/#dns`，你可以看到 `chrome` 浏览器的 `DNS` 缓存。
- 系统缓存主要存在 `/etc/hosts``(Linux` 系统)中

DNS负载均衡

不知道你们有没有注意这样一件事，你访问 `baidu.com` 的时候，每次响应的并非是同一个服务器（`IP` 地址不同），一般大公司都有成百上千台服务器来支撑访问，假设只有一个服务器，那它的性能和存储量要多大才能支撑这样大量的访问呢? `DNS` 可以返回一个合适的机器的 `IP` 给用户，例如可以根据每台机器的负载量，该机器离用户地理位置的距离等等，这种过程就是 `DNS` 负载均衡

2. 发起TCP连接

`TCP` 提供一种可靠的传输，这个过程涉及到三次握手，四次挥手，下面我们详细看看 `TCP` 提供一种面向连接的，可靠的字节流服务。其首部的数据格式如下



字段分析

- **源端口**：源端口和 `IP` 地址的作用是标识报文的返回地址。
- **目的端口**：端口指明接收方计算机上的应用程序接口。

`TCP` 报头中的源端口号和目的端口号同 `IP` 数据报中的源 `IP` 与目的 `IP` 唯一确定一条 `TCP` 连接。

- **序号**：是 `TCP` 可靠传输的关键部分。序号是该报文段发送的数据组的第一个字节的序号。在 `TCP` 传送的流中，每一个字节都有一个序号。比如一个报文段的序号为 `300`，报文段数据部分共有 `100` 字节，则下一个报文段的序号为 `400`。所以序号确保了 `TCP` 传输的有序性。
- **确认号**：即 `ACK`，指明下一个期待收到的字节序号，表明该序号之前的所有数据已经正确无误的收到。确认号只有当 `ACK` 标志为 `1` 时才有效。比如建立连接时，`SYN` 报文的 `ACK` 标志位为 `0`。
- **首部长度/数据偏移**：占 `4` 位，它指出 `TCP` 报文的数据距离 `TCP` 报文段的起始处有多远。由于首部可能含有可选项内容，因此 `TCP` 报头的长度是不确定的，报头不包含任何任选字段则长度为 `20` 字节，`4` 位首部长度字段所能表示的最大值为 `1111`，转化为 `10` 进制为 `15`，`15 * 32 / 8 = 60`，故报头最大长度为 `60` 字节。首部长度也叫数据偏移，是因为首部长度实际上指示了数据区在报文段中的起始偏移值。
- **保留**：占 `6` 位，保留今后使用，但目前应都为 `0`。
- **控制位**：`URG ACK PSH RST SYN FIN`，共 `6` 个，每一个标志位表示一个控制功能。
- **紧急URG**：当 `URG = 1`，表明紧急指针字段有效。告诉系统此报文段中有紧急数据
- **确认ACK**：仅当 `ACK = 1` 时，确认号字段才有效。`TCP` 规定，在连接建立后所有报文的传输都必须把 `ACK` 置 `1`。
- **推送PSH**：当两个应用进程进行交互式通信时，有时在一端的应用进程希望在键入一个命令后立即就能收到对方的响应，这时候就将 `PSH = 1`。
- **复位RST**：当 `RST = 1`，表明 `TCP` 连接中出现严重差错，必须释放连接，然后再重新建立连接。
- **同步SYN**：在连接建立时用来同步序号。当 `SYN = 1`，`ACK = 0`，表明是连接请求报文，若同意连接，则响应报文中应该使 `SYN = 1`，`ACK = 1`。
- **终止FIN**：用来释放连接。当 `FIN = 1`，表明此报文的发送方的数据已经发送完毕，并且要求释放。
- **窗口**：滑动窗口大小，用来告知发送端接受端的缓存大小，以此控制发送端发送数据的速率，从而达到流量控制。窗口大小时一个 `16bit` 字段，因而窗口大小最大为 `65535`。
- **校验和**：奇偶校验，此校验和是对整个的 `TCP` 报文段，包括 `TCP` 头部和 `TCP` 数据，以 `16` 位字进行计算所得。由发送端计算和存储，并由接收端进行验证。
- **紧急指针**：只有当 `URG` 标志置 `1` 时紧急指针才有效。紧急指针是一个正的偏移量，和顺序号字段中的值相加表示紧急数据最后一个字节的序号。`TCP` 的紧急方式是发送端向另一端发送紧急数据的一种方式。
- **选项和填充**：最常见的可选字段是最长报文大小，又称为 `MSS (Maximum Segment Size)`，每个连接方通常都在通信的第一个报文段（为建立连接而设置 `SYN` 标志为 `1` 的那个段）中指明这个选项，它表示本端所能接受的最大报文段的长度。选项长度不一定是 `32` 位的整数倍，所以要加填充位，即在这个字段中加入额外的零，以保证 `TCP` 头是 `32` 的整数倍。
- **数据部分**：`TCP` 报文段中的数据部分是可选的。在一个连接建立和一个连接终止时，双方交换的报文段仅有 `TCP` 首部。如果一方没有数据要发送，也使用没有任何数据的首部来确认收到的数据。在处理超时的许多情况中，也会发送不带任何数据的报文段。

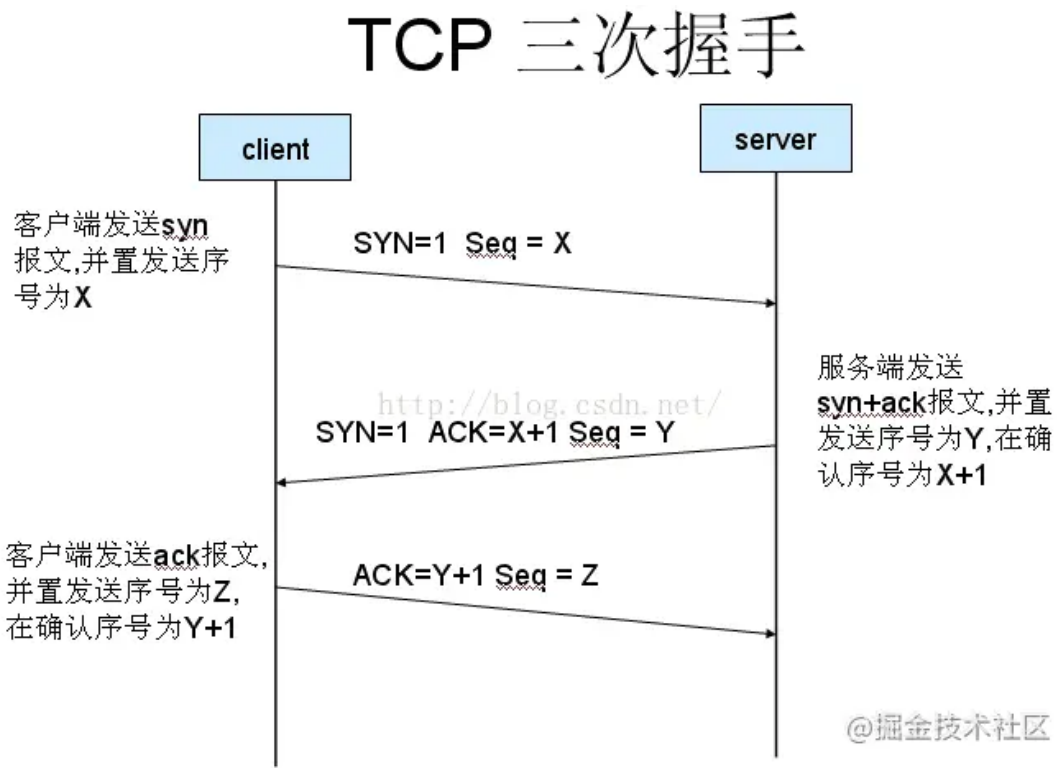
需要注意的是：

- **不要将确认序号 `Ack` 与标志位中的 `ACK` 搞混了。**
- **确认方 `Ack` = 发起方 `Req + 1`，两端配对。**

三次握手

- 第一次握手：客户端发送 `syn` 包(`Seq = x`)到服务器，并进入 `SYN_SEND` 状态，等待服务器确认；
- 第二次握手：服务器收到 `syn` 包，必须确认客户的 `SYN (ack = x + 1)`，同时自己也发送一个 `SYN` 包（`Seq = y`），即 `SYN + ACK` 包，此时服务器进入 `SYN_RECV` 状态；
- 第三次握手：客户端收到服务器的 `SYN + ACK` 包，向服务器发送确认包 `ACK (ack = y + 1)`，此包发送完毕，客户端和服务器进入 `ESTABLISHED` 状态，完成三次握手。

  握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，`TCP` 连接一旦建立，在通信双方中的任何一方主动关闭连接之前，`TCP` 连接都将被一直保持下去。



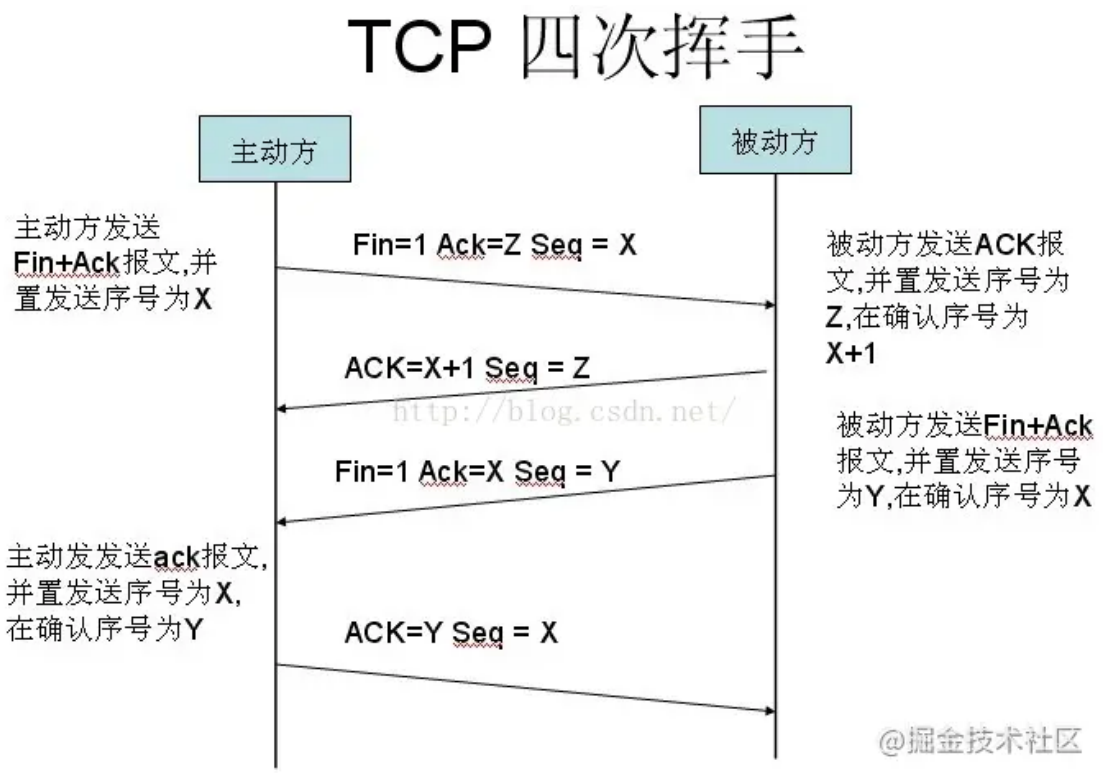
为什么会采用三次握手，若采用二次握手可以吗？ 四次呢?

建立连接的过程是利用客户服务器模式，假设主机 `A` 为客户端，主机 `B` 为服务器端。

采用三次握手是为了防止失效的连接请求报文段突然又传送到主机 `B`，因而产生错误。失效的连接请求报文段是指：主机 `A` 发出的连接请求没有收到主机 `B` 的确认，于是经过一段时间后，主机 `A` 又重新向主机 `B` 发送连接请求，且建立成功，顺序完成数据传输。考虑这样一种特殊情况，主机 `A` 第一次发送的连接请求并没有丢失，而是因为网络节点导致延迟达到主机 `B`，主机 `B` 以为是主机 `A` 又发起的新连接，于是主机 `B` 同意连接，并向主机 `A` 发回确认，但是此时主机 `A` 根本不会理会，主机 `B` 就一直在等待主机 `A` 发送数据，导致主机 `B` 的资源浪费。
采用两次握手不行，原因就是上面说的失效的连接请求的特殊情况。而在三次握手中， `client` 和 `server` 都有一个发 `syn` 和收 `ack` 的过程，双方都是发后能收，表明通信则准备工作OK。
为什么不是四次握手呢？大家应该知道通信中著名的蓝军红军约定，这个例子说明，通信不可能100%可靠，而上面的三次握手已经做好了通信的准备工作，再增加握手，并不能显著提高可靠性，而且也没有必要。

四次挥手

数据传输完毕后，双方都可释放连接。最开始的时候，客户端和服务器都是处于 `ESTABLISHED` 状态，假设客户端主动关闭，服务器被动关闭。



- 第一次挥手：客户端发送一个 `FIN`，用来关闭客户端到服务器的数据传送，也就是客户端告诉服务器：我已经不会再给你发数据了(当然，在 `fin` 包之前发送出去的数据，如果没有收到对应的 `ack` 确认报文，客户端依然会重发这些数据)，但是，此时客户端还可以接受数据。`FIN = 1`，其序列号为 `seq = u`（等于前面已经传送过来的数据的最后一个字节的序号加 `1`），此时，客户端进入 `FIN-WAIT-1`（终止等待 `1`）状态。`TCP` 规定，`FIN` 报文段即使不携带数据，也要消耗一个序号。
- 第二次挥手：服务器收到 `FIN` 包后，发送一个 `ACK` 给对方并且带上自己的序列号 `seq`，确认序号为收到序号 `+1`（与 `SYN` 相同，一个 `FIN` 占用一个序号）。此时，服务端就进入了 `CLOSE-WAIT`（关闭等待）状态。`TCP` 服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个 `CLOSE-WAIT` 状态持续的时间。
  此时，客户端就进入 `FIN-WAIT-2`（终止等待 `2`）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。
- 第三次挥手：服务器发送一个 `FIN`，用来关闭服务器到客户端的数据传送，也就是告诉客户端，我的数据也发送完了，不会再给你发数据了。由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为 `seq = w`，此时，服务器就进入了 `LAST-ACK`（最后确认）状态，等待客户端的确认。
- 第四次挥手：主动关闭方收到 `FIN` 后，发送一个 `ACK` 给被动关闭方，确认序号为收到序号 `+1`，此时，客户端就进入了 `TIME-WAIT`（时间等待）状态。注意此时 `TCP` 连接还没有释放，必须经过 `2 * MSL`（最长报文段寿命）的时间后，当客户端撤销相应的 `TCB` 后，才进入 `CLOSED` 状态。
  服务器只要收到了客户端发出的确认，立即进入 `CLOSED` 状态。同样，撤销 `TCB` 后，就结束了这次的 `TCP` 连接。可以看到，服务器结束 TC`P 连接的时间要比客户端早一些。

为什么客户端最后还要等待 `2MSL`？

`MSL（Maximum Segment Lifetime）`，`TCP` 允许不同的实现可以设置不同的 `MSL` 值。

- **第一，保证客户端发送的最后一个 `ACK` 报文能够到达服务器，因为这个 `ACK` 报文可能丢失，站在服务器的角度看来，我已经发送了 `FIN + ACK` 报文请求断开了，客户端还没有给我回应，应该是我发送的请求断开报文它没有收到，于是服务器又会重新发送一次，而客户端就能在这个 `2MSL` 时间段内收到这个重传的报文，接着给出回应报文，并且会重启 `2MSL` 计时器。**

- **第二，防止类似与“三次握手”中提到了的“已经失效的连接请求报文段”出现在本连接中。客户端发送完最后一个确认报文后，在这个 `2MSL` 时间中，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样新的连接中不会出现旧连接的请求报文。**

为什么建立连接是三次握手，关闭连接确是四次挥手呢?

建立连接的时候， 服务器在 `LISTEN` 状态下，收到建立连接请求的 `SYN` 报文后，把 `ACK` 和 `SYN` 放在一个报文里发送给客户端。而关闭连接时，服务器收到对方的 `FIN` 报文时，仅仅表示对方不再发送数据了但是还能接收数据，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送 `FIN` 报文给对方来表示同意现在关闭连接，因此，己方 `ACK` 和 `FIN` 一般都会分开发送，从而导致多了一次。

3. 发送 `HTTP` 请求

`HTTP` 的端口为 `80 / 8080` ，而 `HTTPS` 的端口为 `443`
发送 `HTTP` 请求的过程就是构建 `HTTP` 请求报文并通过 `TCP` 协议中发送到服务器指定端口请求报文由**请求行，请求报头，请求正文**组成。

- **请求行**
  请求行的格式为 `Method Request-URL HTTP-Version CRLF`
  `eg: GET index.html HTTP/1.1`
  常用的方法有: `GET, POST, PUT, DELETE, OPTIONS, HEAD` 。
  常见的请求方法区别（主要展示 `POST` 和 `GET` 的区别）

  常见的区别
  - `GET` 在浏览器回退时是无害的，而 `POST` 会再次提交请求。
  - `GET` 产生的 `URL` 地址可以被 `Bookmark` ，而 `POST` 不可以。
  - `GET` 请求会被浏览器主动 `cache` ，而 `POST` 不会，除非手动设置。
  - `GET` 请求只能进行 `url` 编码，而 `POST` 支持多种编码方式。
  - `GET` 请求参数会被完整保留在浏览器历史记录里，而 `POST` 中的参数不会被保留。
  - `GET` 请求在 `URL` 中传送的参数是有长度限制的，而 `POST` 么有。
  - 对参数的数据类型， `GET` 只接受 `ASCII` 字符，而 `POST` 没有限制。
  - `GET` 比 `POST` 更不安全，因为参数直接暴露在 `URL` 上，所以不能用来传递敏感信息。
  - `GET` 参数通过 `URL` 传递， `POST` 放在 `Request body` 中。

  注意一点你也可以在 `GET` 里面藏 `body` ， `POST` 里面带参数
  重点区别
  - `GET` 会产生一个 `TCP` 数据包，而 `POST` 会产生两个 `TCP` 数据包。
    详细的说就是:
    - 对于 `GET` 方式的请求，浏览器会把 `http header` 和 `data` 一并发送出去，服务器响应 `200` (返回数据);
    - 而对于 `POST` ，浏览器先发送 `header` ，服务器响应 `100 continue` ，浏览器再发送 `data` ，服务器响应 `200 ok` (返回数据)。

    注意一点，并不是所有的浏览器都会发送两次数据包， `Firefox` 就发送一次

- **请求报头**
  请求报头允许客户端向服务器传递请求的附加信息和客户端自身的信息。

从图中可以看出，请求报头中使用了 `Accept, Accept-Encoding, Accept-Language, Cache-Control, Connection, Cookie` 等字段。`Accept` 用于指定客户端用于接受哪些类型的信息，`Accept-Encoding` 与 `Accept` 类似，它用于指定接受的编码方式。`Connection` 设置为 `Keep-alive` 用于告诉客户端本次 `HTTP` 请求结束之后并不需要关闭 `TCP` 连接，这样可以使下次 `HTTP` 请求使用相同的 `TCP` 通道，节省 `TCP` 连接建立的时间。

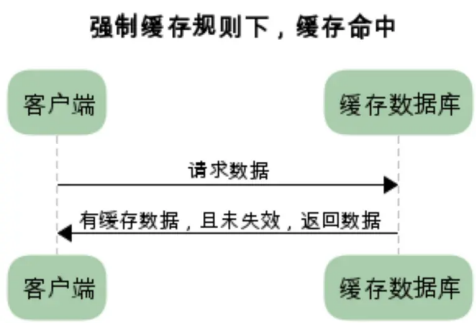
- **请求正文**
  当使用 `POST`， `PUT` 等方法时，通常需要客户端向服务器传递数据。这些数据就储存在请求正文中。在请求包头中有一些与请求正文相关的信息，例如: 现在的 `Web` 应用通常采用 `Rest` 架构，请求的数据格式一般为 `json`。这时就需要设置 `Content-Type: application/json`。

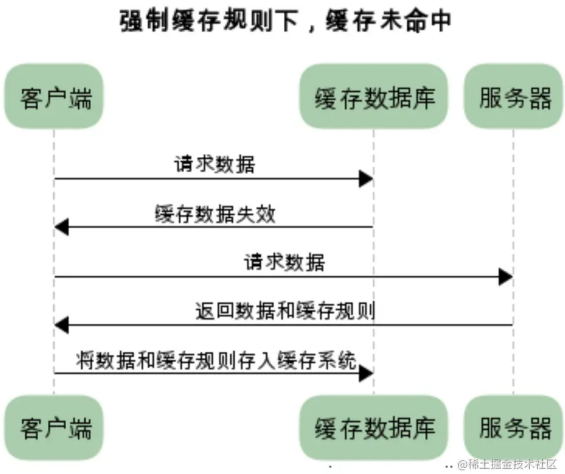
`HTTP` 缓存

`HTTP` 属于客户端缓存，我们常认为浏览器有一个缓存数据库，用来保存一些静态文件，下面我们分为以下几个方面来简单介绍 `HTTP` 缓存

- 缓存的规则
- 缓存的方案
- 缓存的优点
- 不同刷新的请求执行过程

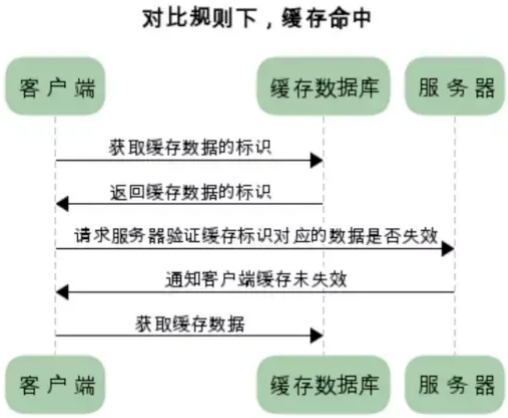
- **缓存的规则**
  缓存规则分为 **强制缓存** 和 **协商缓存**
  - **强制缓存**
    当缓存数据库中有客户端需要的数据，客户端直接将数据从其中拿出来使用（如果数据未失效），当缓存服务器没有需要的数据时，客户端才会向服务端请求。

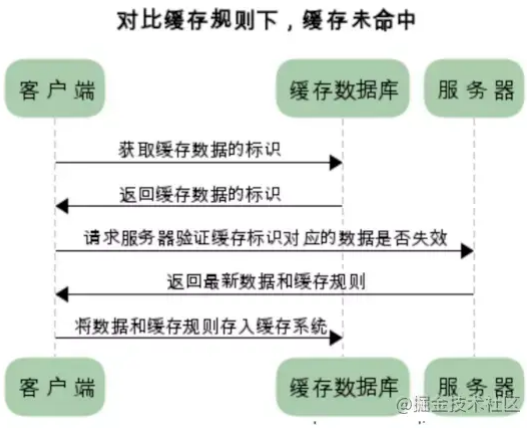




- **协商缓存**

  又称 **对比缓存**。客户端会先从缓存数据库拿到一个缓存的标识，然后向服务端验证标识是否失效，如果没有失效服务端会返回 `304`，这样客户端可以直接去缓存数据库拿出数据，如果失效，服务端会返回新的数据





强制缓存的优先级高于协商缓存，若两种缓存皆存在，且强制缓存命中目标，则协商缓存不再验证标识。

- **缓存的方案**

  上面的内容让我们大概了解了缓存机制是怎样运行的，但是，服务器是如何判断缓存是否失效呢？我们知道浏览器和服务器进行交互的时候会发送一些请求数据和响应数据，我们称之为 `HTTP` 报文。报文中包含首部 `header` 和主体部分 `body`。与缓存相关的规则信息就包含在 `header` 中。`boby` 中的内容是 `HTTP` 请求真正要传输的部分。举个 `HTTP` 报文 `header` 部分的例子如下：

我们依旧分为强制缓存和协商缓存来分析。

强制缓存

对于强制缓存，服务器响应的 `header` 中会用两个字段来表明——`Expires` 和 `Cache-Control`。

- **`Expires`**
  `Exprires` 的值为服务端返回的数据到期时间。当再次请求时的请求时间小于返回的此时间，则直接使用缓存数据。但由于服务端时间和客户端时间可能有误差，这也将导致缓存命中的误差，另一方面，`Expires` 是 `HTTP1.0` 的产物，故现在大多数使用 `Cache-Control` 替代。
- **`Cache-Control`**
  `Cache-Control` 有很多属性，不同的属性代表的意义也不同。
  - `private`： 客户端可以缓存
  - `public`： 客户端和代理服务器都可以缓存
  - `max-age = t`： 缓存内容将在 `t` 秒后失效
  - `no-cache`： 需要使用协商缓存来验证缓存数据
  - `no-store`： 所有内容都不会缓存。

协商缓存

协商缓存需要进行对比判断是否可以使用缓存。浏览器第一次请求数据时，服务器会将缓存标识与数据一起响应给客户端，客户端将它们备份至缓存中。再次请求时，客户端会将缓存中的标识发送给服务器，服务器根据此标识判断。若未失效，返回 `304` 状态码，浏览器拿到此状态码就可以直接使用缓存数据了。

- **`Last-Modified`**
  `Last-Modified`： 服务器在响应请求时，会告诉浏览器资源的最后修改时间。
  - `If-Modified-Since`： 浏览器再次请求服务器的时候，请求头会包含此字段，后面跟着在缓存中获得的最后修改时间。服务端收到此请求头发现有 `If-Modified-Since`，则与被请求资源的最后修改时间进行对比，如果一致则返回 `304` 和响应报文头，浏览器只需要从缓存中获取信息即可。
  - 从字面上看，就是说：从某个时间节点算起，是否文件被修改了
  - 如果真的被修改：那么开始传输响应一个整体，服务器返回： `200 OK`
  - 如果没有被修改：那么只需传输响应 `header`，服务器返回： `304 Not Modified`
  - `If-Unmodified-Since`： 从字面上看, 就是说: 从某个时间点算起, 是否文件没有被修改
    - 如果没有被修改：则开始‘继续’传送文件：服务器返回: `200 OK`
    - 如果文件被修改：则不传输，服务器返回： `412 Precondition failed` (预处理错误)

  这两个的区别是一个是修改了才下载一个是没修改才下载。
  `Last-Modified` 说好却也不是特别好，因为如果在服务器上，一个资源被修改了，但其实际内容根本没发生改变，会因为 `Last-Modified` 时间匹配不上而返回了整个实体给客户端（即使客户端缓存里有个一模一样的资源）。为了解决这个问题， `HTTP1.1` 推出了 `Etag`。

- **`Etag`**
  `Etag`： 服务器响应请求时，通过此字段告诉浏览器当前资源在服务器生成的唯一标识（生成规则由服务器决定）
  - `If-None-Match`： 再次请求服务器时，浏览器的请求报文头部会包含此字段，后面的值为在缓存中获取的标识。服务器接收到次报文后发现 `If-None-Match` 则与被请求资源的唯一标识进行对比。
    - 不同，说明资源被改动过，则响应整个资源内容，返回状态码 `200`。
    - 相同，说明资源无心修改，则响应 `header`，浏览器直接从缓存中获取数据信息。返回状态码 `304`。

  但是实际应用中由于 `Etag` 的计算是使用算法来得出的，而算法会占用服务端计算的资源，所有服务端的资源都是宝贵的，所以就很少使用 `Etag` 了。

- **缓存的优点**
  - 减少了冗余的数据传递，节省宽带流量
  - 减少了服务器的负担，大大提高了网站性能
  - 加快了客户端加载网页的速度 这也正是 `HTTP` 缓存属于客户端缓存的原因。

- **不同刷新的请求执行过程**
  - **浏览器地址栏中写入 `URL`，回车**
    浏览器发现缓存中有这个文件了，不用继续请求了，直接去缓存拿（最快）
  - **`F5`**
    `F5` 就是告诉浏览器，别偷懒，好歹去服务器看看这个文件是否有过期了。于是浏览器就战战兢兢的发送一个请求带上 `If-Modify-Since`。
  - **`Ctrl + F5`**
    告诉浏览器，你先把你缓存中的这个文件给我删了，然后再去服务器请求个完整的资源文件下来。于是客户端就完成了强行更新的操作.

4. 服务器处理请求并返回 `HTTP` 报文

它会对 `TCP` 连接进行处理，对 `HTTP` 协议进行解析，并按照报文格式进一步封装成 `HTTP Request` 对象，供上层使用。这一部分工作一般是由 `Web` 服务器去进行，我使用过的 `Web` 服务器有 `Tomcat, Nginx` 和 `Apache` 等等
`HTTP` 报文也分成三份，**状态码，响应报头和响应报文**

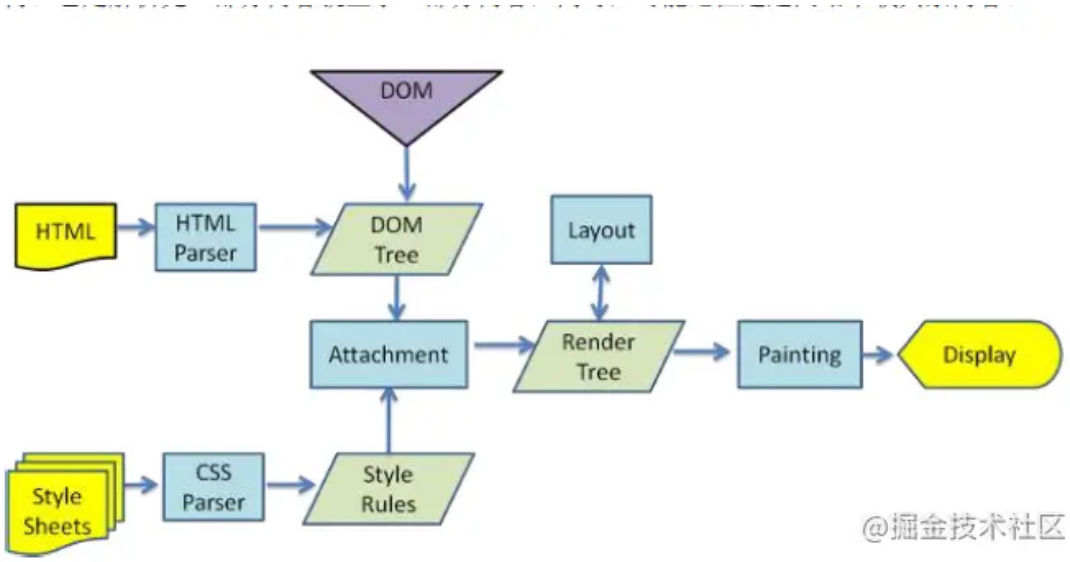
- **状态码**
  状态码是由 `3` 位数组成，第一个数字定义了响应的类别，且有五种可能取值:
  - `1xx`： 指示信息–表示请求已接收，继续处理。
  - `2xx`： 成功–表示请求已被成功接收、理解、接受。
  - `3xx`： 重定向–要完成请求必须进行更进一步的操作。
  - `4xx`： 客户端错误–请求有语法错误或请求无法实现。
  - `5xx`： 服务器端错误–服务器未能实现合法的请求。

  常见状态码区别
  - **`200` 成功**
    请求成功，通常服务器提供了需要的资源。
  - **`204` 无内容**
    服务器成功处理了请求，但没有返回任何内容。
  - **`301` 永久移动**
    请求的网页已永久移动到新位置。服务器返回此响应（对 `GET` 或 `HEAD` 请求的响应）时，会自动将请求者转到新位置。
  - **`302` 临时移动**
    服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。
  - **`304` 未修改**
    自从上次请求后，请求的网页未修改过。服务器返回此响应时，不会返回网页内容。
  - **`400` 错误请求**
    服务器不理解请求的语法。
  - **`401` 未授权**
    请求要求身份验证。对于需要登录的网页，服务器可能返回此响应。
  - **`403` 禁止**
    服务器拒绝请求。
  - **`404` 未找到**
    服务器找不到请求的网页。
  - **`422` 无法处理**
    请求格式正确，但是由于含有语义错误，无法响应
  - **`500` 服务器内部错误**
    服务器遇到错误，无法完成请求。

- **响应报头**
  常见的响应报头字段有：`Server, Connection...`。

- **响应报文**
  你从服务器请求的 `HTML, CSS, JS` 文件就放在这里面

5. 浏览器解析渲染页面



这个图就是 `webkit` 解析渲染页面的过程。

- 解析 `HTML` 形成 `DOM` 树
- 解析 `CSS` 形成 `CSSOM` 树
- 合并 `DOM` 树和 `CSSOM` 树形成渲染树
- 浏览器开始渲染并绘制页面

这个过程涉及两个比较重要的概念 **回流和重绘，`DOM` 结点都是以盒模型形式存在，需要浏览器去计算位置和宽度等，这个过程就是回流。等到页面的宽高，大小，颜色等属性确定下来后，浏览器开始绘制内容，这个过程叫做重绘。** 浏览器刚打开页面一定要经过这两个过程的，但是这个过程非常非常非常消耗性能，所以我们应该尽量减少页面的回流和重绘

- **性能优化之回流重绘**
  - 回流
    当 `Render Tree` 中部分或全部元素的尺寸、结构、或某些属性发生改变时，浏览器重新渲染部分或全部文档的过程称为回流。

    会导致回流的操作：
    - 页面首次渲染
    - 浏览器窗口大小发生改变
    - 元素尺寸或位置发生改变
    - 元素内容变化（文字数量或图片大小等等）
    - 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

    一些常用且会导致回流的属性和方法：
    - `clientWidth、clientHeight、clientTop、clientLeft`

        - `offsetWidth`、`offsetHeight`、`offsetTop`、`offsetLeft`
        - `scrollWidth`、`scrollHeight`、`scrollTop`、`scrollLeft`
        - `scrollIntoView()`、`scrollIntoViewIfNeeded()`
        - `getComputedStyle()`
        - `getBoundingClientRect()`
        - `scrollTo()`

    - 重绘
      当页面中元素样式的改变并不影响它在文档流中的位置时（例如：`color`、`background-color`、`visibility` 等），浏览器会将新样式赋予给元素并重新绘制它，这个过程称为重绘。

    - 优化
        - **`CSS`**
            - 避免使用 `table` 布局。
            - 尽可能在 `DOM` 树的最末端改变 `class`。
            - 避免设置多层内联样式。
            - 将动画效果应用到 `position` 属性为 `absolute` 或 `fixed` 的元素上。
            - 避免使用 `CSS` 表达式（例如：`calc()`）。
        - **`JavaScript`**
            - 避免频繁操作样式，最好一次性重写 `style` 属性，或者将样式列表定义为 `class` 并一次性更改 `class` 属性。
            - 避免频繁操作 `DOM`，创建一个 `documentFragment`，在它上面应用所有 `DOM` 操作，最后再把它添加到文档中。
            - 也可以先为元素设置 `display: none`，操作结束后再把它显示出来。因为在 `display` 属性为 `none` 的元素上进行的 `DOM` 操作不会引发回流和重绘。
            - 避免频繁读取会引发回流/重绘的属性，如果确实需要多次使用，就用一个变量缓存起来。
            - 对具有复杂动画的元素使用绝对定位，使它脱离文档流，否则会引起父元素及后续元素频繁回流。

- **`JS` 的解析**
  `JS` 的解析是由浏览器的 `JS` 引擎完成的。由于 `JavaScript` 是单线程运行，也就是说一个时间只能干一件事，干这件事情时其他事情都有排队，但是有些人物比较耗时（例如 `IO` 操作），所以将任务分为同步任务和异步任务，所有的同步任务放在主线程上执行，形成执行栈，而异步任务等待，当执行栈被清空时才去看看异步任务有没有东西要搞，有再提取到主线程执行，这样往复循环（冤冤相报何时了，阿弥陀佛），就形成了 `Event Loop` 事件循环