2022前端面试大全

Rudy_ LV.3
2021年12月14日 17:26 · 阅读 11523

关注

前端面试大全

HTML

html语义化

html语义化就是用合适的标签来标记和合适的内容

语义化的好处

- 有利于SEO：和搜索引擎建立良好沟通，有助于爬虫抓取更多的有效信息。
- 方便其他设备（如屏幕阅读器、盲人阅读器、移动设备）更好的解析页面。
- 使代码更具可读性，便于团队开发和维护
- 网页加载慢导致CSS文件还未加载时（没有CSS），页面仍然清晰、可读、好看

语义化标签

- `header footer nav(导航链接区域)`
- `aside` 通常被包含在article元素中作为主要内容的附属信息部分
- `section` section元素代表文档中的“节”或“段”
- `article` article代表一个在文档，页面或者网站中自成一体的内容,比如论坛帖子
- `h1 - h6` 标题，根据级别依次递减 h1一个页面最好只有一个
- `p` 内容段落

行内元素和块级元素

- 行内元素的margin和padding只在水平方向有效果

标签： `a img input label select strong textarea`

块级元素

- 会独占一行,默认情况下,其宽度自动填满其父元素宽度
- 块级元素可以设置width,height属性.
- **块级元素即使设置了宽度,仍然是独占一行.**
- 块级元素可以设置margin和padding属性

标签： `div p form h1-h6 ul ol`

行内块级元素

结合的行内和块级的优点，既可以设置长宽，可以让padding和margin生效，又可以和其他行内元素并排

meta标签

`<meta>` 标签提供关于 HTML 文档的元数据。它不会显示在页面上，但是对于机器是可读的。可用于浏览器（如何显示内容或重新加载页面），搜索引擎（关键词），或其他 web 服务

常见meta标签作用

```html 复制代码
// 声明文档编码
<meta charset="utf-8">
//页面关键字
<meta name="keywords" content="your keywords">
// 页面描述内容
<meta name="description" content="your description">
// 定义网页作者
<meta name="author" content="author,email address">
//移动端布局参数设置
<meta name="viewport" content="width=device-width, initial-scale=1.0">
content参数如下：
-   width viewport 宽度(数值/device-width)
-   height viewport 高度(数值/device-height)
```

```
//各个浏览器的meta标签有不同的自定义功能，包括 chrome 360 uc 等等

<!-- 优先使用最新的chrome版本 -->
<meta http-equiv="X-UA-Compatible" content="chrome=1" />
<!-- 禁止自动翻译 -->
<meta name="google" value="notranslate">
```

html如何做SEO优化

- 尽量简单，导航最好不要超过3级
- 控制首页链接数量
  网站首页是权重最高的地方，如果首页链接太少,影响网站收录数量。太多影响用户体验，也会降低网站首页的权重，收录效果也不好
- 图片必须添加alt和title，在每个菜单考虑增加面包屑导航
- 控制页面的大小，减少http请求，提高网站的加载速度
- 突出重要内容，合理的设计网站 `title、description` 和 `keywords`
- 尽量让代码语义化，用正确的标签做正确的事

Doctype作用? 严格模式与混杂模式如何区分?它们有何意义?

- 声明位于文档中的最前面，处于html标签之前。告知浏览器以何种模式来渲染文档。
- 严格模式的排版和 JS 运作模式是 以该浏览器支持的最高标准运行。
- 在混杂模式中，页面以宽松的向后兼容的方式显示。模拟老式浏览器的行为以防止站点无法工作。
- DOCTYPE不存在或格式不正确会导致文档以混杂模式呈现。

HTML 4.01 规定了三种文档类型:Strict、Transitional 以及 Frameset

分别表示严格版本、过渡版本以及基于框架的 HTML 文档

DOM操作

1. 创建新节点

```
js 复制代码
```

2. 添加、移除、替换、插入

```js
appendChild()
removeChild()
replaceChild()
insertBefore() //并没有insertAfter()
```

3. 查找

```js
getElementsByTagName() //通过标签名称

getElementsByName() //通过元素的Name属性的值(IE容错能力较强， 会得到一个数组，其中包括id等于name值的)

getElementById() //通过元素Id，唯一性
```

JS

事件循环机制

「硬核JS」一次搞懂JS运行机制

面试题：说说事件循环机制(满分答案来了)

一次弄懂EventLoop

Chrome V8内存回收机制

深入理解chromeV8内存回收机制

script标签中的 defer和async属性

图解 script 标签中的 async 和 defer 属性

相同点：

- async属性是异步加载，加载完成后立即执行，标记为async的脚本不保证按照出现顺序执行
- 标记为defer属性的脚本执行顺序就是按照出现顺序

从输入URL到看到页面发生了什么?

首先我们需要通过 DNS（域名解析系统）将 URL 解析为对应的 IP 地址，然后与这个 IP 地址确定的那台服务器建立起 TCP 网络连接，随后我们向服务端抛出我们的 HTTP 请求，服务端处理完我们的请求之后，把目标数据放在 HTTP 响应里返回给客户端，拿到响应数据的浏览器就可以开始走一个渲染的流程。渲染完毕，页面便呈现给了用户，并时刻等待响应用户的操作

对于这个问题的更详细的解答可以从浏览器的原理方向回答，比如：

1. UI线程负责处理用户的输入，首先会判断用户输入的是关键词还是一个域名地址，如果是关键词就交给搜索引擎处理，如果是URL地址就通知网络进程负责DNS解析和TCP连接以及最终的请求和响应交互

2. 浏览器接收到响应后，网络进行还会进行安全处理，检测请求域名是否和某个已知的病毒网站相同，相同会给出提示

3. 如果网站安全，请求数据已经返回到了浏览器端，此时浏览器进程会申请一个渲染进程负责页面的渲染

4. 在渲染进程和数据都准备好了的前提下，浏览器进程会通知渲染进程进行提交导航操作，一旦浏览器进程收到渲染进程的回复说导航已经提交了，那导航过程就结束了，此时导航栏被更新，当前tab也的会话历史也被更新，你就可以通过前进后退按钮进入访问的页面

5. 导航提交完成后，渲染进程开始着手进行资源加载和渲染页面，这里可以在回答一下渲染过程，就是HTML解析成DOM树，CSS解析成CSSOM树....（等等）

6. 一旦完成渲染，会通过IPC告知浏览器进程，然后UI线程就会停止导航栏上旋转的圈圈

类型判断

1. `typeof`
2. `instanceof`

类型转换

JavaScript 深入之头疼的类型转换(上)

1. 原始值转布尔 `Boolean()`
2. 原始值转数字
   undefined转数字是NaN null是0 string类型忽略所有的前导0，只要有一个字符不是数字就NaN
3. 原始值转字符
4. 原始值转对象
5. 对象转布尔（都是true）
6. 对象转字符串和数字

JavaScript 深入之头疼的类型转换(下)

- 一元操作符 + 会调用 `ToNumber` 处理该值，相当于 `Number()`

- 二元操作符 +

1. lprim = ToPrimitive(value1)
2. rprim = ToPrimitive(value2)
3. 如果 lprim 是字符串或者 rprim 是字符串，那么返回 ToString(lprim) 和 ToString(rprim) 的拼接结果
4. 返回 ToNumber(lprim) 和 ToNumber(rprim)的运算结果

一句话：+号左右两侧分别转成基本类型，如果其中之一是字符串，则调用ToString否则调用ToNumber

- ==相等

规则如下：

1. 如果x与y是同一类型：
   1. x是Undefined，返回true

1. x是NaN，返回false
2. y是NaN，返回false
3. x与y相等，返回true
4. x是+0，y是-0，返回true
5. x是-0，y是+0，返回true
6. 返回false

4. x是字符串，完全相等返回true,否则返回false
5. x是布尔值，x和y都是true或者false，返回true，否则返回false
6. x和y指向同一个对象，返回true，否则返回false

2. x是null并且y是undefined，返回true
3. x是undefined并且y是null，返回true
4. x是数字，y是字符串，判断x == ToNumber(y)
5. x是字符串，y是数字，判断ToNumber(x) == y
6. x是布尔值，判断ToNumber(x) == y
7. y是布尔值，判断x ==ToNumber(y)
8. x是字符串或者数字，y是对象，判断x == ToPrimitive(y)
9. x是对象，y是字符串或者数字，判断ToPrimitive(x) == y
10. 返回false

闭包

对闭包的看法，为什么要用闭包？说一下闭包原理以及应用场景

使用闭包主要是为了设计私有的方法和变量。在js中，函数即闭包，只有函数才会产生作用域的概念

变量)

闭包原理

利用了函数作用域链的特性，一个函数内部定义的函数会将包含外部函数的活动对象添加到它的作用域链中，函数执行完毕，其执行作用域链销毁，但因内部函数的作用域链仍然在引用这个活动对象，所以其活动对象不会被销毁，直到内部函数被销毁后才被销毁

优点

1. 可以从内部函数访问外部函数的作用域中的变量，且访问到的变量长期驻扎在内存中，可供之后使用
2. 避免变量污染全局
3. 把变量存到独立的作用域，作为私有成员存在

缺点

1. 对内存消耗有影响，因为内部函数保存了对外部变量的引用，导致无法被垃圾回收，增大内存的使用量，所以使用不当会造成内存泄漏。
2. 对处理速度有负面影响。闭包的层级决定了引用的外部变量在查找时经过的作用域链长度

应用场景

1. 模块封装，在各个模块规范出来之前，都是用这样的方式防止变量污染全局

js 复制代码

```
var Yideng = (function () {
    // 这样声明为模块私有变量，外界无法直接访问
    var foo = 0;

    function Yideng() {}
    Yideng.prototype.bar = function bar() {
        return foo;
    };
    return Yideng;
}());
```

2. 柯里化函数
3. 使用闭包实现私有化方法和变量

```
    function funcTwo(i){
        memo.push(i)
        console.log(memo.join(','))
    }
    return funcTwo
}
```

继承

原型链继承

js 复制代码

```
function Parent () {
    this.name = 'kevin';
}

Parent.prototype.getName = function () {
    console.log(this.name);
}

function Child () {

}

Child.prototype = new Parent();

var child1 = new Child();

console.log(child1.getName()) // kevin
```

缺点：

1. 引用类型的属性被所有实例共享
2. 在创建Child的时候，不能向Parent传参

借用构造函数（经典继承）

js 复制代码

```
function Parent () {
    this.names = ['kevin', 'daisy'];
```

```
}

var child1 = new Child();

child1.names.push('yayu');

console.log(child1.names); // ["kevin", "daisy", "yayu"]

var child2 = new Child();

console.log(child2.names); // ["kevin", "daisy"]
```

优点：

1.避免了引用类型的属性被所有实例共享

2.可以在 Child 中向 Parent 传参

缺点： 方法都在构造函数中定义，每次创建实例都会自动创建一遍方法

组合继承（常用）

js 复制代码

```
function Parent (name) {
    this.name = name;
    this.colors = ['red', 'blue', 'green'];
}

Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child (name, age) {

    Parent.call(this, name);

    this.age = age;

}

Child.prototype = new Parent();
Child.prototype.constructor = Child;
```

```
console.log(child1.age); // 18
console.log(child1.colors); // ["red", "blue", "green", "black"]

var child2 = new Child('daisy', '20');

console.log(child2.name); // daisy
console.log(child2.age); // 20
console.log(child2.colors); // ["red", "blue", "green"]
```

组合继承最大的缺点是会调用两次父构造函数

一次是设置子类型实例的原型的时候：

js 复制代码

```
Child.prototype = new Parent();
```

一次在创建子类型实例的时候：

js 复制代码

```
var child1 = new Child('kevin', '18');
```

回想下 new 的模拟实现，其实在这句中，我们会执行：

js 复制代码

```
Parent.call(this, name);
```

原型式继承

js 复制代码

```
function createObj(o){
    function F(){}
    F.prototype = o
    return new F()
}
```

包含引用类型的属性值始终都会共享相应的值，这点跟原型链继承一样

寄生式继承

```
    clone.sayName = function () {
        console.log('hi');
    }
    return clone;
}
```

缺点：跟借用构造函数模式一样，每次创建对象都会创建一遍方法

寄生组合式继承

主要目的是间接的让 `Child.prototype` 访问到 `Parent.prototype`

js 复制代码

```
function Parent (name) {
    this.name = name;
    this.colors = ['red', 'blue', 'green'];
}

Parent.prototype.getName = function () {
    console.log(this.name)
}

function Child (name, age) {
    Parent.call(this, name);
    this.age = age;
}

// 关键的三步
var F = function () {};

F.prototype = Parent.prototype;

Child.prototype = new F();


var child1 = new Child('kevin', '18');

console.log(child1);
```

Class继承

- `extends` 后面接着的目标不一定是 `class` ，只要是个有 `prototype` 属性的函数就可以了

Super相关

- 在实现继承时，如果子类中有 `constructor` 函数，必须得在 `constructor` 中调用一下 `super` 函数，因为它就是用来产生实例 `this` 的。
- `super` 有两种调用方式：当成函数调用和当成对象来调用。
- `super` 当成函数调用时，代表父类的构造函数，且返回的是子类的实例，也就是此时 `super` 内部的 `this` 指向子类。在子类的 `constructor` 中 `super()` 就相当于是 `Parent.constructor.call(this)` 。
- `super` 当成对象调用时，普通函数中 `super` 对象指向父类的原型对象，静态函数中指向父类。且通过 `super` 调用父类的方法时， `super` 会绑定子类的 `this` ，就相当于是 `Parent.prototype.fn.call(this)` 。

模块化规范

模块化的好处

- 避免命名冲突(减少命名空间污染)
- 更好的分离, 按需加载
- 高复用性
- 高可维护性

IIFE（匿名立即执行函数）

js 复制代码

```
;(function (root) {
  var api = 'https://github.com/ronffy';
  var config = {
    api: api,
  };
  root.config = config;
}(window));
```

> 加载。这和浏览器的异步加载模块的环境刚好适应（浏览器同步加载模块会导致性能、可用性、调试和跨域访问等问题）。

本规范只定义了一个函数 `define` ，它是全局变量

```
/**
 * @param {string} id 模块名称
 * @param {string[]} dependencies 模块所依赖模块的数组
 * @param {function} factory 模块初始化要执行的函数或对象
 * @return {any} 模块导出的接口
 */
function define(id?, dependencies?, factory): any
```

RequireJS

AMD是一种异步模块规范，RequireJS是AMD规范的实现。

先有 RequireJS，后有 AMD 规范，随着 RequireJS 的推广和普及，AMD 规范才被创建出来

CMD CMD 和 AMD 一样，都是 JS 的模块化规范，也主要应用于浏览器端。

AMD 是 RequireJS 在的推广和普及过程中被创造出来。

CMD 是 SeaJS 在的推广和普及过程中被创造出来。

二者的的主要区别是 CMD 推崇依赖就近，AMD 推崇依赖前置。

```
// AMD
// 依赖必须一开始就写好
define(['./utils'], function(utils) {
  utils.request();
});

// CMD
define(function(require) {
  // 依赖可以就近书写
  var utils = require('./utils');
  utils.request();
});
```

CommonJS

服务端模块规范

CommonJS模块的加载机制是，**输入的是被输出的值的拷贝**。也就是说，一旦输出一个值，模块内部 的变化就影响不到这个值

CommonJS 和 AMD 都是运行时加载，换言之：都是在**运行时**确定模块之间的依赖关系

区别：

1. CommonJS 是服务器端模块规范，AMD 是浏览器端模块规范。
2. CommonJS 加载模块是**同步**的，即执行 `var a = require('./a.js');` 时，在 a.js 文件加载完成后，才执行后面的代码。AMD 加载模块是异步的，所有依赖加载完成后以回调函数的形式执行代码

UMD

UMD 是一种通用模块定义规范，支持全局变量的形式，也符合 AMD 规范，还符合 CommonJS 规范

UMD 解决了 JS 模块跨模块规范、跨平台使用的问题，它是非常好的解决方案。

ES6 Module

AMD 、 CMD 等都是在原有JS语法的基础上二次封装的一些方法来解决模块化的方案，ES6 module（在很多地方被简写为 ESM）是语言层面的规范，ES6 module 旨在为浏览器和服务器提供通用的模块解决方案。长远来看，未来无论是基于 JS 的 WEB 端，还是基于 node 的服务器端或桌面应用，模块规范都会统一使用 ES6 module

import和export都只能用在模块顶级，否则报错可以通过import()动态加载

CommonJS 和 ES6 module

CommonJS 和 AMD 是运行时加载，在运行时确定模块的依赖关系。CommonJS输出的是值的拷贝，并且是同步模块加载规范

前端JavaScript模块化规范进化论

Promise

Promise规范

Promise规范

Pormise特点

手写Promise

史上最最最详细的手写Promise教程

如何解决跨域问题

九种跨域方式实现原理（完整版）

解决跨域问题需要先明确浏览器的同源策略，所谓同源是指"协议+域名+端口"三者相同，即便两个不同的域名指向同一个ip地址，也非同源。

同源策略限制内容有：

- Cookie、LocalStorage、IndexedDB 等存储性内容
- DOM 节点
- AJAX 请求发送后，结果被浏览器拦截了

但是有三个标签是允许跨域加载资源：

- `<img src=XXX>`
- `<link href=XXX>`
- `<script src=XXX>`

JSONP

优点是兼容性好，简单易用，支持浏览器与服务器双向通信。

缺点是只支持GET请求，可能遭受XSS的攻击

js 复制代码

```
<script>
  functioncreateJs(sUrl){
    var oScript =document.createElement('script');
    oScript.type = 'text/javascript';
    oScript.src= sUrl;
    document.getElementsByTagName('head')[0].appendChild(oScript);
  }
  createJs('jsonp.js');
  box({
    'name': 'test'
  });
  functionbox(json){
    alert(json.name);
} </script>
```

js 复制代码

```
$.ajax({
    url:"http://crossdomain.com/jsonServerResponse",
    dataType:"jsonp",
    type:"get",//可以省略
    jsonpCallback:"show",//->自定义传递给服务器的函数名
    jsonp:"callback",//->把传递函数名的那个形参callback
    success:function (data){
    console.log(data);}
});
```

CORS

服务器端对于CORS的支持，主要就是通过设置Access-Control-Allow-Origin来进行的。如果浏览器检测到相应的设置，就可以允许Ajax进行跨域的访问

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

window.name

window对象有个name属性，该属性有个特征:即在一个窗口(window)的生命周期内,窗口载入的所有 的页面都是共享一个window.name的，每个页面对window.name都有读写的权限，window.name是 持久存在一个窗口载入过的所有页面中的

html5 postMessage

postMessage()方法允许来自不同源的脚本采用异步方式进行有限的通信，可以实现跨文本档、多窗口、跨域消息传递

```
otherWindow.postMessage(message, targetOrigin, [transfer]);
```

js 复制代码

```
// a.html
  <iframe src="http://localhost:4000/b.html" frameborder="0" id="frame" onload="load()"></iframe> //
  //内嵌在http://localhost:3000/a.html
    <script>
      function load() {
        let frame = document.getElementById('frame')
        frame.contentWindow.postMessage('我爱你', 'http://localhost:4000') //发送数据
        window.onmessage = function(e) { //接受返回数据
          console.log(e.data) //我不爱你
        }
      }
    </script>

// b.html
  window.onmessage = function(e) {
    console.log(e.data) //我爱你
    e.source.postMessage('我不爱你', e.origin)
 }
```

DOM事件模型和事件流

一个事件发生后，会在子元素和父元素之间传播，分成三个阶段。

冒泡阶段:事件从目标节点自下而上向window对象传播的阶段。

设计模式

JavaScript 设计模式核心原理与应用实践

Symbol有什么用处

1. 可以用来表示一个独一无二的变量**防止命名冲突**
2. Symbol不会被常规方法（除Object.getOwnPropertySymbols）遍历到，可以模拟私有变量

创建ajax的过程（低频考点）

1. 创建 XMLHttpRequest 对象,也就是创建一个异步调用对象.
2. 创建一个新的 HTTP 请求,并指定该 HTTP 请求的方法、 URL 及验证信息.
3. 设置响应 HTTP 请求状态变化的函数.
4. 发送 HTTP 请求.
5. 获取异步调用返回的数据.
6. 使用JavaScript和DOM实现局部刷新.

```js 复制代码
var xmlHttp = new XMLHttpRequest();

xmlHttp.open('GET','demo.php','true');

xmlHttp.send()
xmlHttp.onreadystatechange = function(){
    if(xmlHttp.readyState === 4 & xmlHttp.status=== 200){
} }
```

判断NaN

- typeof 判断为number并且isNaN为true
- 利用NaN不等于自身的特点
- ES6 Object.is(value1,value2)

类数组和数组的区别

数组本质是一个特殊的对象，与常规对象的区别是：

- 当由新元素添加到列表中时，自动更新length属性
- 设置length属性，可以截断数组
- 从Array.protoype中继承了方法
- 属性为'Array'

类数组是一个拥有length属性的普通对象，类数组不能直接调用数组方法

区别：类数组是简单对象，它的原型关系与数组不同

类数组转数组方法：

1. `Array.from()`
2. `Array.prototype.slice.call()`
3. `Array.prototype.forEach()` 遍历后转成新的数组

转换须知：

1. 转换后数组长度由length属性决定，索引不连续时，转换是连续的，会自动补位
2. 仅考虑0或者是正整数的索引

js 复制代码

```
let al1 = {
    length: 4,
    0: 0,
    1: 1,
    a: 3,
    4: 4,
    5: 5,
};
console.log(Array.from(al1)) // [0, 1, undefined, undefined]
```

内存泄露以及定位

深入了解 JavaScript 内存泄露

1. 箭头函数不能绑定arguments,取而代之的是rest的...解决
2. 箭头函数是匿名函数，不能作为构造函数，不能使用new
3. 箭头函数没有原型属性
4. 箭头函数不能绑定this，会将离自己最近的一个普通函数的this作为自己的this
5. call、apply、bind都无法改变箭头函数中this的指向

requestAnimationFrame和requestIdleCallback

`requestAnimationFrame`：在下次重绘前调用指定的动画，方法传入一个回调函数，函数在下次重绘之前执行

页面是一帧一帧绘制出来的，当每秒绘制的帧数（FPS）达到 60 时，页面是流畅的，小于这个值时，用户会感觉到卡顿。1s 60帧，所以每一帧分到的时间是 1000/60 ≈ 16 ms

浏览器的每一帧都会完成什么任务

- 处理用户的交互
- JS 解析执行
- 帧开始。处理窗口尺寸变更，页面滚动等
- requestAnimationFrame(rAF)
- 布局
- 绘制

如果上面6个步骤执行时间没超过16ms，即时间有富余，就执行 `requestIdleCallback` 里面注册的任务。与 `requestAnimationFrame` 每一帧必定会执行不同，requestIdleCallback是**捡浏览器空闲来执行任务**

前端跨页面通信的方案

前端跨页面通信

CSS

flex布局

flex: 1 flex: auto flex: none flex: 0到底有什么 区别？使用场景?

盒模型

标准盒模型：width = content IE盒模型： width = content+padding+border

设置盒模型： `box-sizing:content-box / border-box`

元素的宽高

详解各种获取元素宽高及位置的属性

水平垂直居中

- **水平居中**
  - **margin:0 auto + width:fit-content**： 全部元素
  - **块级元素 + margin:0 auto + width**： 块级元素
    - 若节点不是块级元素需声明 `display:block`
    - 若节点宽度已隐式声明则无需显式声明 `width`
  - **行内元素 + text-aligin:center**： 行内元素
    - 父节点上声明 `text-align`
    - 若节点不是行内元素需声明 `display:inline/inline-block`
  - **position + left/right + margin-left/right + width**： 全部元素
  - **position + left/right + transform:translateX(-50%)**： 全部元素
  - **display:flex + justify-content:center**： 全部元素
    - 父节点上声明 `display` 和 `justify-content`

- 父节点高度未声明或自适应
- 若节点不是块级元素需声明 `display:block`

- **行内元素 + line-height**： 行内元素
  - 父节点上声明 `line-height`
  - 若节点不是行内元素需声明 `display:inline/inline-block`
- **display:table + display:table-cell + vertical-align:middle**： 全部元素
  - 父节点上声明 `display:table`
- **display:table-cell + vertical-align:middle**： 全部元素
  - 父节点上声明 `display` 和 `vertical-align`
- **position + top/bottom + margin-top/bottom + height**： 全部元素
- **position + top/bottom + transform:translateY(-50%)** ： 全部元素
- **display:flex + align-items:center**： 全部元素
  - 父节点上声明 `display` 和 `align-items`
- **display:flex + margin:auto 0**： 全部元素
  - 父节点上声明 `display`

CSS 中解决浮动中高度塌陷的方案有哪些

1. 清除浮动

- 添加标签增加 `clear:both属性`
- css伪元素

2. BFC：计算BFC的高度时，浮动元素也参与计算

3. 添加伪元素

```css
.box::after {
    content: '.';
    height: 0;
    display: block;
    clear: both;
}
```

什么是BFC

BFC原理

选择器的权重

- **10000**： !important
- **1000**： 内联样式 、 外联样式
- **100**： ID选择器
- **10**： 类选择器 、 伪类选择器 、 属性选择器
- **1**： 标签选择器 、 伪元素选择器
- **0**： 通配选择器 、 后代选择器 、 兄弟选择器

重绘&回流（重排）

回流和重排

回流又名**重排**，指 几何属性 需改变的渲染。但感觉回流这个词比较高大上，后续统称回流吧。

可理解成，将整个网页填白，对内容重新渲染一次。只不过以人眼的感官速度去看浏览器回流是不会有任何变化的，若你拥有 闪电侠 的感官速度去看浏览器回流(实质是将时间调慢)，就会发现每次回流都会将页面清空，再从左上角第一个像素点从左到右从上到下这样一点一点渲染，直至右下角最后一个像素点。每次回流都会呈现该过程，只是感受不到而已。

渲染树的节点发生改变，影响了该节点的几何属性，导致该节点位置发生变化，此时就会触发浏览器回流并重新生成渲染树。回流意味着节点的几何属性改变，需重新计算并生成渲染树，导致渲染树的全部或部分发生变化

渲染树的节点发生改变，但不影响该节点的几何属性。由此可见，回流对浏览器性能的消耗是高于重绘的，而且回流一定会伴随重绘，重绘却不一定伴随回流

如何减少和避免回流重绘

1. 使用visibility:hidden替换display:none
2. 使用transform代替top
3. 避免使用Table布局
4. 避免规则层级过多
5. 避免节点属性值放在循环里当成循环变量

   比如：

```js 复制代码
for (let i = 0; i < 10000; i++) {
    const top = document.getElementById("css").style.top;
    console.log(top);
}

const top = document.getElementById("css").style.top;
for (let i = 0; i < 10000; i++) {
    console.log(top);
}
```

6. 动态改变类而不改变样式

postion值得含义

- absoulte:绝对定位，相对于最近一级不是static的父元素定位
- fix:绝对定位，相对浏览器窗口或者frame
- relative:生产相对定位，相对其在普通文档流中的位置
- static 默认定位
- sticky 生成粘性定位的元素，容器的位置根据正常文档流计算得出

1. link属于HTML标签，而@import是CSS提供的;
2. 页面被加载的时，link会同时被加载，而@import被引用的CSS会等到引用它的CSS文件被加载完再加载
3. import只在IE5以上才能识别，而link是HTML标签，无兼容问题;
4. link方式的样式的权重 高于@import的权重.

vertical-align

一种简单的 CSS 属性，用来指定行内元素（inline）或表格单元格（table-cell）元素的垂直对齐方式

起作用的前提：元素为 inline 水平元素或 table-cell 元素，包括 `span` ，`img` ，`input` ，`button` ，`td` 以及通过 display 改变了显示水平为 inline 水平或者 table-cell 的元素。这也意味着，默认情况下， `div` ，`p` 等元素设置 vertical-align 无效

值得注意的是：例如 float 和 position: absolute ，一旦设置了这两个属性之一，元素的 display 值被忽略，强制当成 block 方式处理，因此，vertical-align 也就失去了作用

px em rem 的区别

px：绝对长度单位，像素 px 是相对于显示器屏幕分辨率来说的

em: 相对长度单位，相对于当前对象内文本的字体尺寸

- em 的值并不是固定的
- em 会继承父级元素的字体大小(参考物是父元素的 font-size) em 中所有的字体都是相对于父元素的大小决定的

rem 相对于 html 根元素的 font-size

1em=1rem=16px

在 body 中加入 font-size:62.5% 这样直接就是原来的 px 数值除以10加上 em 就可以

常见布局方式

css 伪类与伪元素区别

1. 伪类表示被选择元素的某种状态，例如 `:hover`

2. 伪元素表示的是被选择元素的某个部分，这个部分看起来像一个独立的元素，但是是"假元素"，只存在于css中，所以叫"伪"的元素，例如 `:before` 和 `:after`

核心区别在于，是否创造了“新的元素”

- 定义不同
  - 伪类即假的类，可以添加类来达到效果
  - 伪元素即假元素，需要通过添加元素才能达到效果
- 总结:
  - 伪类和伪元素都是用来表示文档树以外的"元素"。
  - 伪类和伪元素分别用单冒号:和双冒号::来表示。
  - 伪类和伪元素的区别，关键点在于是否在虚拟空间上增加了元素（类）
  - 是否需要添加元素才能达到效果，如果是则是伪元素，反之则是伪类。

哪些样式可以被继承

1. 文本相关属性 `font-family font-size font-weight line-height`
2. 列表相关的属性 `list-style-image, list-style-position,list-style-type, list-style`
3. color

css实现各种图形

用 css 画三角形、梯形、扇形、箭头和椭圆几种基本形状

Vue2

1. name属性的作用

- 项目使用keep-alive时，可以搭配组件的name属性进行过滤
- DOM做递归组件时需要调用自身name
- vue-devtools 调试工具里显示的组见名称是由vue中组件name决定的

2. keep-alive

- keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，避免重新渲染
- 一般结合路由和动态组件一起使用，用于缓存组件；
- 提供 include 和 exclude 属性，两者都支持字符串或正则表达式， include 表示只有名称匹配的组件会被缓存，exclude 表示任何名称匹配的组件都不会被缓存 ，其中 exclude 的优先级比 include 高；
- 对应两个钩子函数 activated 和 deactivated ，当组件被激活时，触发钩子函数 activated，当组件被移除时，触发钩子函数 deactivated

参考链接

v-for中key的作用

`v-for` 默认使用就地复用策略，列表数据修改的时候，他会根据key值去判断某个值是否修改，如果修改，则重新渲染这一项，否则复用之前的元素

为什么不推荐使用下标作为key，比如我一个数组有3条数据，我在第一条后面插入一个元素，那么实际上从第二条开始的数据key都变了，那么我本来只需要渲染一条数据，现在会渲染3条

Vue中的mixin和组件冲突时的合并策略

1. 当组件选项和混入冲突时，以组件优先
2. 当组件和混入都定义生命周期，都会触发，而且mixin会先触发

v-model的实现原理

Vue性能优化手段

```
编码阶段
尽量减少data中的数据，data中的数据都会增加getter和setter，会收集对应的watcher
v-if和v-for不能连用
如果需要使用v-for给每项元素绑定事件时使用事件代理
SPA 页面采用keep-alive缓存组件
在更多的情况下，使用v-if替代v-show
key保证唯一
使用路由懒加载、异步组件
防抖、节流
第三方模块按需导入
长列表滚动到可视区域动态加载
图片懒加载
SEO优化
预渲染
服务端渲染SSR
打包优化
压缩代码
Tree Shaking/Scope Hoisting
使用cdn加载第三方模块
多线程打包happypack
splitChunks抽离公共文件
sourceMap优化
用户体验
骨架屏
PWA
```

js 复制代码

Vue3

Vue3的新特性有哪些

聊一聊 Vue3 的 9 个知识点

1. 组合式API（Composition API）
2. setup 包括setup的参数以及 `ref reactive`
3. 声明周期的变化
4. watch和watchEffet
5. 新增三个组件 Fragment 支持多个根节点 Suspense 可以在组件渲染之前的等待时间是

Proxy对比Object.defineProperty

- Object.defineProperty 是 Es5 的方法，Proxy 是 Es6 的方法
- defineProperty 不能监听到数组下标变化和对象新增属性，Proxy 可以
- defineProperty 是劫持对象属性，Proxy 是代理整个对象
- defineProperty 局限性大，只能针对单属性监听，所以在一开始就要全部递归监听。Proxy 对象嵌套属性运行时递归，用到才代理，也不需要维护特别多的依赖关系，性能提升很大，且首次渲染更快
- defineProperty 会污染原对象，修改时是修改原对象，Proxy 是对原对象进行代理并会返回一个新的代理对象，修改的是代理对象
- defineProperty 不兼容 IE8，Proxy 不兼容 IE11

其他

浏览器缓存策略

浏览器缓存策略

前端安全

前端安全

TCP和UDP的区别

TCP：向上提供面向连接的可靠服务

UDP：向上提供无连接的不可靠服务

虽然 `UDP` 并没有 `TCP` 传输来的准确，但是也能在很多实时性要求高的地方有所作为 对数据准

是否连接	无连接	面向连接
是否可靠	不可靠传输，不使用流量控制和拥塞控制	可靠传输，使用流量控制和拥塞控制
连接对象个数	支持一对一，一对多，多对一和多对多交互通信	只能是一对一通信
传输方式	面向报文	面向字节流
首部开销	首部开销小，仅8字节	首部最小20字节，最大60字节
适用场景	适用于实时应用（IP电话、视频会议、直播等）	适用于要求可靠传输的应用，例如文件传输

@稀土掘金技术社区

TCP三次握手和四次挥手

三次握手：

发送端首先发送一个带SYN标志的数据包给对方。接收端收到后，回传一个带有SYN/ACK标志的数据包 以示传达确认信息。最后，发送端再回传一个带ACK标志的数据包，代表"握手"结束。 若在握手过程中某个阶段莫名中断，TCP协议会再次以相同的顺序发送相同的数据包

四次挥手：

1. 第一次挥手:主动关闭方发送一个FIN，用来关闭主动方到被动关闭方的数据传送，也就是主动关闭 方告诉被动关闭方: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，主动关闭方依然会重发这些数据)，但是，此时主动关闭方还可 以接受数据。

2. 第二次挥手:被动关闭方收到FIN包后，发送一个ACK给对方，确认序号为收到序号+1(与SYN相 同，一个FIN占用一个序号)。

3. 第三次挥手:被动关闭方发送一个FIN，用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉 主动关闭方，我的数据也发送完了，不会再给你发数据了。

4. 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

2XX Success

```
200  （成功） 服务器已成功处理了请求。 通常，这表示服务器提供了请求的网页。
204  （无内容） 服务器成功处理了请求，但没有返回任何内容。
206  （部分内容） 服务器成功处理了部分 GET 请求。
```

3XX 重定向状态码

```
301  （永久重定向） 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
302  （临时重定向） 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。
```

4XX Client Error

```
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。
```

5XX Server Error

```
500  （服务器内部错误） 服务器遇到错误，无法完成请求。
502  （错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。
503  （服务不可用） 服务器目前无法使用（由于超载或停机维护）
```

http1.0/1.1/2.0 之间的区别

http1.0

每次TCP连接都只能发送一个请求，当服务器响应后就会关闭此连接，下次发送请求还需要再次建立链接

http1.1

很多请求排队，造成队头阻塞。

`http 2.0`

加入了双工模式，客户端可以同时发送多个请求，服务器也可以同时响应多个请求，解决了HTTP的队头阻塞问题，使用了多路复用技术 同一个TCP连接可以处理多个请求

服务端可以主动向客户端推送数据

http1.1和http2.0多路复用的区别

http 1.1 同一个TCP连接只能处理一个请求采用一问一答的形式，上一个请求响应处理后才能处理下一个请求
http 2.0 同域名下的所有请求都在单个连接上完成

原因：http2.0是二进制帧协议(有标识)，http1.1：基于文本分割的协议

http1.1存在的问题

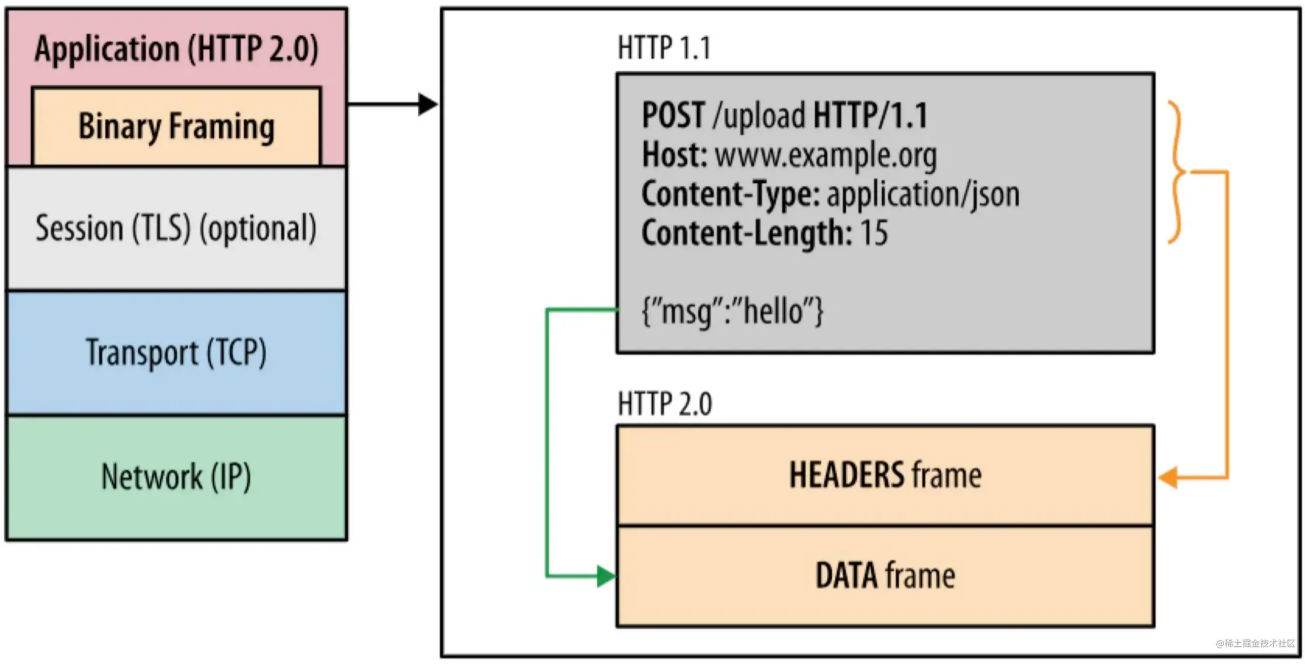
- 线头阻塞：TCP连接上只能发送一个请求，前面的请求未完成前，后续的请求都在排队等待。
- 多个TCP连接

  虽然HTTP/1.1管线化可以支持请求并发，但是浏览器很难实现，chrome、firefox等都禁用了管线化。所以1.1版本请求并发依赖于多个TCP连接，建立TCP连接成本很高，还会存在慢启动的问题。
- 头部冗余，采用文本格式

  HTTP/1.X版本是采用文本格式，首部未压缩，而且每一个请求都会带上cookie、user-agent等完全相同的首部。
- 客户端需要主动请求

HTTP2性能提升的核心就在于二进制分帧层。HTTP2是二进制协议，他采用二进制格式传输数据而不是http1.x的文本格式

制来编码。



多路复用

HTTP2建立一个TCP连接，一个连接上面可以有任意多个流（stream），消息分割成一个或多个帧在流里面传输。帧传输过去以后，再进行重组，形成一个完整的请求或响应，这实现了真正的并发

头部压缩

服务端推送

了解HTTPS吗？请介绍一下

详细解析HTTP和HTTPs的区别

看完这篇 HTTPS，和面试官扯皮就没问题了

《大前端进阶 安全》系列 HTTPS详解（通俗易懂）

1. HTTPS解决了什么问题？

HTTPS出现了

2. 什么是HTTPS

HTTP：超文本传输协议，它是一个在计算机世界里专门在两点之间传输文字、图片、音频、视频等超文本数据的约定和规范

HTTPS：HTTPS是一个在计算机世界里专门在两点直之间安全的传输文字、图片、音频和视频等超文本数据的约定和规范

HTTPS = HTTP + SSL(TLS)

3. HTTPS做了什么

- 加密：HTTPS 通过对数据加密来使其免受窃听者对数据的监听，这就意味着当用户在浏览网站时，没有人能够监听他和网站之间的信息交换，或者跟踪用户的活动，访问记录等，从而窃取用户信息
- 数据一致性：数据在传输的过程中不会被窃听者所修改，用户发送的数据会完整的传输到服务端，保证用户发的是什么，服务器接收的就是什么
- 身份认证：是指确认对方的真实身份，防止中间人攻击并建立用户信任

SSL和TLS：SSL（安全套接字层），TLS传输安全层，TLS实际上是SSL的标准化，TLS目前存在3个版本1.1，1.2，1.3

通常来说，HTTP 会先直接和 TCP 进行通信。在使用 SSL 的 HTTPS 后，则会先演变为和 SSL 进行通信，然后再由 SSL 和 TCP 进行通信

更多SSL/TLS协议内容，SSL/TLS协议运行机制的概述

SSL/TLS的基本运行过程

1. 客户端向服务器端索要并验证公钥。

   为了防止公钥被篡改，将公钥放在数字证书中。只要证书是可信的，公钥就是可信的

容，服务端的公钥加密只用来加密 对话秘钥本身

3. 双方采用"对话密钥"进行加密通信。

其他区别

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

GET和POST区别

1.GET在浏览器回退不会再次请求，POST会再次提交请求

2.GET请求会被浏览器主动缓存，POST不会，要手动设置

3.GET请求参数会被完整保留在浏览器历史记录里，POST中的参数不会

4.GET请求在URL中传送的参数是有长度限制的，而POST没有限制

5.GET参数通过URL传递，POST放在Request body中

6.GET参数暴露在地址栏不安全，POST放在报文内部更安全

7.GET一般用于查询信息，POST一般用于提交某种信息进行某些修改操作

8.GET产生一个TCP数据包；POST产生两个TCP数据包

- GET请求，浏览器会把request header和data一起发送出去，

CDN

程序员要搞明白CDN，这篇应该够了

渐进增强和优雅降级

渐进增强 :针对低版本浏览器进行构建页面，保证最基本的功能，然后再针对高级浏览器进行效果、交 互等改进和追加功能达到更好的用户体验。

优雅降级 :一开始就构建完整的功能，然后再针对低版本浏览器进行兼容

浏览器存储

共同点:都是保存在浏览器端，且都遵循同源策略

- Cookie 跟随浏览器请求携带，大小不能超过4k，只在设置的过期时间内有效，即使窗口或者浏览器关闭
- webstorage：大小更大 2.5M-10M之间
  - localStorage：始终有效，即使浏览器关闭，**数据在浏览器所有同源窗口共享**
  - sessionStorage:仅在当前窗口有效
- indexDB：indexedDB 允许储存大量数据，提供查找接口，还能建立索引

性能优化

前端性能优化

前端性能优化之旅

前端工程化

`devDependencies` 、 `dependencies` 、 `optionalDependencies` 和 `peerDependencies` 区别

optionalDependencies 是可选模块，安不安装均可，即使安装失败，包的安装过程也不会报错

peerDependencies 一般用在大型框架和库的插件上，例如我们写 webpack--xx-plugin 的时候，对于 使用者而言，他一定会先有 webpack 再安装我们的这个模块，这里的 peerDependencies 就是约束了 这个例子中 webpack 的版本

npm 中 --save-dev 和 --save 之间的区别

save-dev 和 save 都会把模块安装到 node_modules 目录下，但 save-dev 会将依赖名称和版本写到 devDependencies 下，而 save 会将依赖名称和版本写到 dependencies 下。如果我们使用 npm -- production install 这样的命令安装模块的话，就只会安装 save 安装的包

webpack优化

待补充

webpack plugin和loader内部实现机制是怎么样的?

plugin

插件基本结构就是一个有apply方法的构造函数，里面执行一些插件的钩子，apply方法是用来获取compiler的

js 复制代码

```
module.exports = class ZipPlugin {
    // 接收参数
    constructor(options) {
        this.options = options;
    }
    // apply函数可以用来访问webpack的核心
    apply(compiler) {
        //  tap ： 以同步方式触发钩子;
        // tapAsync ： 以异步方式触发钩子;
        // tapPromise ： 以异步方式触发钩子，返回 Promise
        compiler.hooks.emit.tapAsync('ZipPlugin', (compilation, callback) => {
```

```
                const source = compilation.assets[filename].source();
                folder.file(filename, source);
            }

            zip.generateAsync({
                type: 'nodebuffer'
            }).then((content) => {
                // 设置输出路径
                const outputPath = path.join(__dirname,'..',this.options.filename + '.zip')
                const outputRelativePath = path.relative(
                    compilation.options.output.path,
                    outputPath
                );
                compilation.assets[outputRelativePath] = new RawSource(content);
                // 继续向下执行打包操作
                callback();
            });
        });
    }
}
```

插件具体的参数和钩子可以查看 webpack官网

loader

loader就是一个导出为函数的js模块，函数接收源文件内容或者是被其他loader处理过的内容，然后进行自己的一些处理操作，在将结果暴露出去

loader-runner：允许在不安装webpack的情况下运行loader

自己实现一个plugin和loader

实现一个压缩dist的资源为zip包的插件

js 复制代码

```
// jszip是一个用于创建、读取和编辑.zip文件的JavaScript库
const JSZip = require('jszip');
const path = require('path');
const RawSource = require('webpack-sources').RawSource;
const zip = new JSZip();
```

```
        this.options = options;
    }

    apply(compiler) {
        // emit是compiler的一个钩子 emit： 输出 asset到output 目录之前执行。这个钩子不会被复制到子编译器
        // 异步钩子用tapAsync执行，同步用tap执行
        compiler.hooks.emit.tapAsync('ZipPlugin', (compilation, callback) => {
            // 添加一个文件夹
            const folder = zip.folder(this.options.filename);
            // 遍历打包的资源
            for (let filename in compilation.assets) {
                // 获取每个文件的source
                const source = compilation.assets[filename].source();
                // 将资源文件添加到文件夹里
                folder.file(filename, source);
            }
            // 生成zip文件
            zip.generateAsync({
                type: 'nodebuffer'
            }).then((content) => {
                const outputPath = path.join(
                    compilation.options.output.path,
                    this.options.filename + '.zip'
                );
                // 确定文件输出路径
                const outputRelativePath = path.relative(
                    compilation.options.output.path,
                    outputPath
                );
                // 通过 RawSource 向 compilation.assets写入zip包
                compilation.assets[outputRelativePath] = new RawSource(content);
                // 继续执行后面的操作
                callback();
            });
        });
    }
}
```

webpack 4.x 对比3.x版本进行了哪些优化，为什么提高了构建速度?

1. webpack4增加了mode配置项，可以对不同的环境开启不同的配置

- 默认使用更快的 md4 hash 算法。
- webpack AST 可以直接从loader传递给 AST，减少解析时间。
- 使用字符串方法代替正则 表达式。

webpack 5有哪些新特性

1. 内置静态资源的构建能力

在webpack5之前，我们一般都会使用file-loader、url-loader、raw-loader来处理静态资源，而webpack提供了内置的静态资源构建能力，我们不需要安装额外的 loader，仅需要简单的配置就能实现静态资源的打包和分目录存放。

js 复制代码

```
// webpack.config.js
module.exports = {
    ...,
    module: {
      rules: [
          {
            test: /\.(png|jpg|svg|gif)$/,
            type: 'asset/resource',
            generator: {
                // [ext]前面自带"."
                filename: 'assets/[hash:8].[name][ext]',
            },
        },
      ],
    },
}
```

2. 内置文件系统缓存

Webpack5 之前，我们会使用 cache-loader 缓存一些性能开销较大的 loader ，或者是使用 hard-source-webpack-plugin 为模块提供一些中间缓存。在 Webpack5 之后，默认就为我们集成了一种自带的缓存能力（对 module 和 chunks 进行缓存）

js 复制代码

```
// webpack.config.js
module.exports = {
```

```
        buildDependencies: {
            config: [__filename],  // 当构建依赖的config文件（通过 require 依赖）内容发生变化时，缓存失效
        },
        name: '',  // 配置以name为隔离，创建不同的缓存文件，如生成PC或mobile不同的配置缓存
        ...,
    },
}
```

3. 内置worker的构建能力

以前使用webpack，如果在项目中使用了webworker，需要手动安装work-loader并配置，还需要针对worker配置特别的work.js的文件名。

在 Webpack5 中，我们不需要添加 loader 的处理方式，并且不需要针对 worker 配置特定的 .worker.js 之类的文件名，借助于 new URL， 便能实现 worker 的创建

js 复制代码

```
// master.js
const worker = new Worker(new URL('./calc.js', import.meta.url), {
    name: "calc"
  /* webpackEntryOptions: { filename: "workers/[name].js" } */
});
worker.onmessage = e => {
  console.log(e.data.value);
};
```

4. 不在为nodeJS模块提供自动引用Polyfills

在早期，webpack 的目的是为了让大多数的 Node.js 模块运行在浏览器中，但如今模块的格局已经发生了变化，现在许多模块主要是为前端而编写。Webpack <= 4 的版本中提供了许多 Node.js 核心模块的 polyfills，一旦某个模块引用了任何一个核心模块（如 `cypto` 模块），webpack 就会自动引用这些 polyfills。

尽管这会使得使用为 Node.js 编写模块变得容易，但它在构建时给 bundle 附加了庞大的 polyfills。在大部分情况下，这些 polyfills 并非必须

5. 支持命名代码块ID

现在可以在生产环境中使用 `chunkIds: "named"` 命名

6. 对Tree-shaking进行了优化

Webpack5 能够跟踪对导出的嵌套属性的访问，所以支持嵌套的Tree-shaking分析

webpack-dev-server 实现原理

webpack和gulp的区别

gulp的核心是task，通过配置一系列task，并且定义task要处理的事务（比如js压缩，css压缩，less编译）,之后让gulp来依次执行这些task，让整个**流程自动化。**本质是一个工具链，所以gulp是一种前端自动化任务管理工具。

webpack：是静态文件打包工具，可以把项目的各种js文、css文件等打包合并成一个或多个文件，主要用于模块化方案，预编译模块的方案

如果工程模块依赖非常简单，甚至没有用到模块化的概念。只需要进行简单的合并、压缩，就使用grunt/gulp即可。但是如果整个项目使用了模块化管理，而且相互以来非常强，我们就可以使用更加强大的webpack了

微前端有了解过吗？简单介绍下

微前端基础知识了解

低代码平台有了解过吗？是怎么实现的

待补充

业务

如何设计一个通用的组件

1. 要易用（至少让使用者能够简单方便的引入到程序当中）

5. 要谨慎（要考虑多种情况）
6. 要易读（写的东西要能给别人讲清楚）

如何做?

1.如何做到易用，所谓众口难调，你觉得好用，其他人未必这样觉得。做一个组件之前，先了解各方使用者的需求或许是最好的选择，不要让别人适应你而是应该适应别人，并且多采用一些公认的设计模式方法，也可以提高自己的设计能力。

2.如何做到稳定，系统关键点可以多编写一些单元测试适应不同的场景，在有外部访问的组件当中，可以适当增加压力测试。

3.如何做到灵活，系统设计的时候，先要确定“变化”和“不变”，对于变化除了默认值可以增加一些配置文件选项，让使用者能够自己配置，或者开放一些抽象类和接口，让调用者自己实现。

4.如何全面，当你做一个组件的时候，别人可能也会关心，发生异常或者变化的情况，这个时候适当的增加一些日志，更进一步增加一些回调方法，或许能够为排查问题，处理异常以及后续其他操作提供一些便利条件。

5.怎么谨慎，做组件的时候，考虑到被各种不同的使用者使用，这里就需要将异常情况或者各种条件分钟考虑清楚，当变量是非法值的时候需要系统能够有效的处理，或者在输入的时候提前判断，或者做好异常处理。

6.要容易读，很多人在写组件的时候，都不注重代码的注释和更新的履历，殊不知这样的情况就是当时间比较长之后，很难理解自己的代码也很难让别人看懂，不能因为只是给别人中的组件不涉及到业务就不去写注释，良好的习惯也是程序员价值的衡量标准

算法

快速排序

实现一个斐波那契数列

js 复制代码

```
F(0) = 0;
```

```js
function fib(n){
    if(n < 0) throw new Error('输入的数字不能小于0');
    if (n < 2) {
      return n;
    }
    return fib(n - 1) + fib(n - 2);
}
```

递归的缺点是有大量的重复计算，比如计算fib(5) = fib(3)+fib(4) fib(4) = fib(3)+fib(2) 这样fib(3)就被重复计算了两次

2. 迭代

```js
function fib(n){
    let f0 = 0,f1 = 1
    let res = 0
    for(let i=1;i<n;i++){
        res = f0+f1
        f0 = f1
        f1 = res
    }
    return res
}
```

3. 递归（去除重复计算版本）

```js
function fib(n){
    if(n<0) throw new  Error('error')
    if(n<2) return n
    function _fib(n,a,b){
        if(n===0) return a
        return _fib(n-1,b,a+b)
    }
    return _fib(n,0,1)
}
```

分类: 前端 标签: 面试

前端面试

记录前端面试的相关文章

关注专栏

评论

输入评论（Enter换行，Ctrl + Enter发送）

全部评论 1

最新 最热

继国缘一 JY.3

2月前

吃个桃桃?

点赞 回复

相关推荐

奇舞精选 22天前 前端

2022年国内外前端发展态势

1.3w 118 15

捌玖ki 4月前 面试 JavaScript

网易前端面试（灵犀部门）

3.3w 369 34

寻找海蓝96 3年前 面试 前端

面试官到底想看什么样的简历?

6.9w 1651 121

前端早早聊 23天前 前端

2022 年，前端深水区的裁员结局

3.8w 222 64

时光足迹 20天前 面试 Vue.js

2022年前端面试集锦

1.2w 173 26

Gaby 2月前 JavaScript 架构 面试

你还在直接用 localStorage 么？该提升下逼格了

7.1w 854 210

进军的王小二 9月前 面试 前端

2021年我的前端面试小结(70题)

8.9w 1138 158

扫地盲僧 9月前 JavaScript 程序员

2022年如何成为一名优秀的大前端Leader?

2.2w 709 87

摸鱼的春哥 7月前 前端 JavaScript

2022，前端的天🌩要怎么变?

15.0w 1463 651

法医 3月前 前端 CSS

2022高频前端面试题——CSS篇

3486 78 20

海明月 8月前 面试 前端

三十七个常见Vue面试题，背就完事了。

9.1w 1671 109

涡流 10月前 面试

一尾流莺 6月前 前端 面试

【🐯初/中级前端面经】中小型公司面试时都会问些什么?

8.8w 1832 177

伊人a 1年前 前端 面试

2021年我的前端面试准备

28.1w 5637 558

vortesnail 6月前 前端 面试

做了一份前端面试复习计划，保熟～

31.2w 6332 345

杰出D 1年前 前端 算法

面试了十几个高级前端，竟然连（扁平数据结构转Tree）都写不出来

21.0w 3283 1713

宅神king 2年前 面试

五月中级前端面试报告

2.7w 352 74

字节前端 6月前 前端 JavaScript GitHub

来自未来，2022 年的前端人都在做什么?

3.3w 719 42

神三元 2年前 JavaScript 面试

（建议精读）HTTP灵魂之问，巩固你的 HTTP 知识体系

20.8w 4584 173