JavaScript

解释下为什么接下来这段代码不是IIFE(立即调用的函数表达式)?

```
function foo(){//code... }()
```

> **解析**
>
> 以function关键字开头的语句会被解析为函数声明，而函数声明是不允许直接运行的。只有当解析器把这句话解析为函数表达式，才能够直接运行，怎么办呢？以运算符开头就可以了。
>
> `(function foo(){// code.. })()`
>
> 更加详细的解释可参考以下链接：
>
> https://swordair.com/function-and-exclamation-mark/

Object.is() 与原来的比较操作符 “===”、“==” 的区别?

> (1) 两等号判等，会在比较时进行类型转换；
>
> (2)三等号判等（判断严格），比较时不进行隐式类型转换，（类型不同则会返回false）；
>
> (3)Object.is 在三等号判等的基础上特别处理了 NaN 、-0 和 +0 ，保证 -0 和 +0 不再相同，但
>
> Object.is(NaN, NaN) 会返回 true。
>
> Object.is 应被认为有其特殊的用途，而不能用它认为它比其它的相等对比更宽松或严格。

common.js和es6中模块引入的区别?

答案:

> CommonJS 是一种模块规范，最初被应用于 Nodejs，成为 Nodejs 的模块规范。运行在浏览器端的 JavaScript 由于也缺少类似的规范，在 ES6 出来之前，前端也实现了一套相同的模块规范 (例如: AMD)，用来对前端模块进行管理。自 ES6 起，引入了一套新的 ES6 Module 规范，在语言标准的层面上实现了模块功能，而且实现得相当简单，有望成为浏览器和服务器通用的模块解决方案。但目前浏览器对 ES6 Module 兼容还不太好，我们平时在 Webpack 中使用的 export 和 import，会经过 Babel 转换为 CommonJS 规范。在使用上的差别主要有：
>
> - CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用
> - CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。
> - CommonJs 是单个值导出，ES6 Module可以导出多个
> - CommonJs 是动态语法可以写在判断里，ES6 Module 静态语法只能写在顶层
> - CommonJs 的 this 是当前模块，ES6 Module的 this 是 undefined

请解释原型设计模式

> **答案:**

> 原型模式可用于创建新对象，但它创建的不是非初始化的对象，而是使用原型对象（或样本对象）的值进行初始化的对象。原型模式也称为属性模式。
>
> 原型模式在初始化业务对象时非常有用，业务对象的值与数据库中的默认值相匹配。原型对象中的默认值被复制到新创建的业务对象中。
>
> 经典的编程语言很少使用原型模式，但作为原型语言的 JavaScript 在构造新对象及其原型时使用了这个模式。

浏览器是怎样解析CSS选择器的?

> 解析：
>
> CSS选择器的解析是从右向左解析的。若从左向右的匹配，发现不符合规则，需要进行回溯，会损失很多性能。若从右向左匹配，先找到所有的最右节点，对于每一个节点，向上寻找其父节点直到找到根元素或满足条件的匹配规则，则结束这个分支的遍历。两种匹配规则的性能差别很大，是因为从右向左的匹配在第一步就筛选掉了大量的不符合条件的最右节点（叶子节点），而从左向右的匹配规则的性能都浪费在了失败的查找上面。 而在 CSS 解析完毕后，需要将解析的结果与 DOM Tree 的内容一起进行分析建立一棵 Render Tree，最终用来进行绘图。在建立 Render Tree 时（WebKit 中的「Attachment」过程），浏览器就要为每个 DOM Tree 中的元素根据 CSS 的解析结果（Style Rules）来确定生成怎样的 Render Tree。

浏览器缓存

缓存可以减少网络 IO 消耗，提高访问速度。浏览器缓存是一种操作简单、效果显著的前端性能优化手段 很多时候，大家倾向于将浏览器缓存简单地理解为“HTTP 缓存”。 但事实上，浏览器缓存机制有四个方面，它们按照获取资源时请求的优先级依次排列如下：

- Memory Cache
- Service Worker Cache
- HTTP Cache
- Push Cache

缓存它又分为强缓存和协商缓存。优先级较高的是强缓存，在命中强缓存失败的情况下，才会走协商缓存

- 实现强缓存：过去我们一直用 expires。当服务器返回响应时，在 Response Headers 中将过期时间写入 expires 字段，现在一般使用Cache-Control 两者同时出现使用Cache-Control
- 协商缓存：Last-Modified 是一个时间戳，如果我们启用了协商缓存，它会在首次请求时随着 Response Headers 返回：每次请求去判断这个时间戳是否发生变化。从而去决定你是304读取缓存还是给你返回最新的数据

浏览器是如何渲染页面的?

解析:

渲染的流程如下： 1.解析HTML文件，创建DOM树。

自上而下，遇到任何样式（link、style）与脚本（script）都会阻塞（外部样式不阻塞后续外部脚本的加载）。

2.解析CSS。优先级：浏览器默认设置<用户设置<外部样式<内联样式<HTML中的style样式；

3.将CSS与DOM合并，构建渲染树（Render Tree）

4.布局和绘制，重绘（repaint）和重排（reflow）

判断一个给定的字符串是否是重构的

> **答案:**
>
> 如果两个字符串是同构的，那么字符串 A 中所有出现的字符都可以用另一个字符替换，以便获得字符串 B，而且必须保留字符的顺序。字符串 A 中的每个字符必须与字符串 B 的每个字符一对一对应。
>
> - paper 和 title 将返回 true。
> - egg 和 sad 将返回 false。
> - dgg 和 add 将返回 true。

```
isIsomorphic("egg", 'add'); // true
isIsomorphic("paper", 'title'); // true
isIsomorphic("kick", 'side'); // false

function isIsomorphic(firstString, secondString) {

  // 检查长度是否相等，如果不相等，它们不可能是同构的
  if (firstString.length !== secondString.length) return false

  var letterMap = {};

  for (var i = 0; i < firstString.length; i++) {
    var letterA = firstString[i],
        letterB = secondString[i];

    // 如果 letterA 不存在，创建一个 map，并将 letterB 赋值给它
    if (letterMap[letterA] === undefined) {
      letterMap[letterA] = letterB;
    } else if (letterMap[letterA] !== letterB) {
      // 如果 letterA 在 map 中已存在，但不是与 letterB 对应，
      // 那么这意味着 letterA 与多个字符相对应。
      return false;
    }
  }, ,     // 迭代完毕，如果满足条件，那么返回 true。
  // 它们是同构的。
  return true;
}
```

下面代码的输出结果是什么？(D)

```
function sayHi() {
  console.log(name);
  console.log(age);
```

```
  var name = 'Lydia';
  let age = 21;
}
```

A. undefined 和 undefined;

B. Lydia 和 ReferenceError;

C. ReferenceError 和 21;

D. undefined 和 ReferenceError;

> **解析：**
>
> 本题的考点主要是var与let的区别以及var的预解析问题。var所声明的变量会被预解析，var name;提升到作用域最顶部，所以在开始的console.log(name)时，name已经存在，但是由于没有赋值，所以是undefined；而let会有暂时性死区，也就是在let声明变量之前，你都无法使用这个变量，会抛出一个错误，故选D。

下面代码的输出结果是什么？(B)

```
const arr = [1, 2, [3, 4, [5]]];
console.log(arr.flat(1));
```

A. [1, 2, [3, 4, [5]]];

B. [1, 2, 3, 4, [5]];

C. [1, 2, [3, 4, 5]];

D. [1, 2, 3, 4, 5];

> **解析：**
>
> 这里主要是考察Array.prototype.flat方法的使用，扁平化会创建一个新的，被扁平化的数组，扁平化的深度取决于传入的值；这里传入的是1也就是默认值，所以数组只会被扁平化一层，相当于[].concat([1, 2], [3, 4, [5]])，故选B。

下面代码的输出结果是什么？(C)

```
const name = 'Lydia Hallie';
const age = 21;

console.log(Number.isNaN(name));
console.log(Number.isNaN(age));

console.log(isNaN(name));
console.log(isNaN(age));
```

A. true false true false

B. true false false false

C. false false true false

D. false true false true

> **解析：**
>
> 本题主要考察isNaN和Number.isNaN的区别；首先isNaN在调用的时候，会先将传入的参数转换为数字类型，所以非数字值传入也有可能返回true，所以第三个和第四个打印分别是true false；Number.isNaN不同的地方是，他会首先判断传入的值是否为数字类型，如果不是，直接返回false，本题中传入的是字符串类型，所以第一个和第二个打印均为false，故选C。

请回答其他类型值转换为数字时的规则

> 答案：
>
> 有时我们需要将非数字值当作数字来使用，比如数学运算。为此 ES5 规范在 9.3 节定义了抽象操作 ToNumber。
>
> （1）Undefined 类型的值转换为 NaN。
>
> （2）Null 类型的值转换为 0。
>
> （3）Boolean 类型的值，true 转换为 1，false 转换为 0。
>
> （4）String 类型的值转换如同使用 Number() 函数进行转换，如果包含非数字值则转换为 NaN，空字符串为 0。
>
> （5）Symbol 类型的值不能转换为数字，会报错。
>
> （6）对象（包括数组）会首先被转换为相应的基本类型值，如果返回的是非数字的基本类型值，则再遵循以上规则将其强制转换为数字。
>
> 为了将值转换为相应的基本类型值，抽象操作 ToPrimitive 会首先（通过内部操作 DefaultValue）检查该值是否有valueOf() 方法。
>
> 如果有并且返回基本类型值，就使用该值进行强制类型转换。如果没有就使用toString() 的返回值（如果存在）来进行强制类型转换.如果 valueOf() 和 toString() 均不返回基本类型值，会产生 TypeError 错误。

JS有几种方法判断变量的类型?

> (1) 使用typeof检测 当需要变量是否是number, string, boolean, function, undefined, json类型时，可以使用typeof进行判断。 arr, json, nul, date, reg, error 全部被检测为object类型，，其他的变量能够被正确检测出来。
>
> (2) 使用instanceof检测 instanceof 运算符与 typeof 运算符相似，用于识别正在处理的对象的类型。与 typeof 方法不同的是，instanceof 方法要求开发者明确地确认对象为某特定类型。
>
> (3). 使用constructor检测 constructor本来是原型对象上的属性，指向构造函数。但是根据实例对象寻找属性的顺序，若实例对象上没有实例属性或方法时，就去原型链上寻找，因此，实例对象也是能使用

> constructor属性的。

js数组去重，能用几种方法实现?

> (1).使用es6 set方法 [...new Set(arr)] let arr = [1,2,3,4,3,2,3,4,6,7,6]; let unique = (arr)=> [...new Set(arr)]; unique(arr);//[1, 2, 3, 4, 6, 7] (2).利用新数组indexOf查找 indexOf() 方法可返回某个指定的元素在数组中首次出现的位置。如果没有就返回-1。 (3).for 双重循环 通过判断第二层循环，去重的数组中是否含有该元素，如果有就退出第二层循环，如果没有j==result.length就相等，然后把对应的元素添加到最后的数组里面。

```
let arr = [1,2,3,4,3,2,3,4,6,7,6];
let result = [];
for(var i = 0 ; i< arr.length; i++) {
    for(var j = 0 ; j < result.length ; j++) {
if( arr[i] === result[j]){
break;
}
    }
    if(j == result.length){
result.push(arr[i]);
    }
}
console.log(result);
```

(4).利用for嵌套for，然后splice去重

```
function unique(arr){
    for(vari=0; i<arr.length; i++){
    for(varj=i+1; j<arr.length; j++){
if(arr[i]==arr[j]){
//第一个等同于第二个，splice方法删除第二个 arr.splice(j,1); j--;
}
}
}
returnarr;
}
```

(5).

```
let arr = [1,2,3,4,3,2,3,4,6,7,6];
let unique = (arr) => {
    return arr.filter((item,index) => {
return arr.indexOf(item) === index;
    })
};
unique(arr);
```

(6).利用Map数据结构去重

```
let arr = [1,2,3,4,3,2,3,4,6,7,6];
let unique = (arr)=> {
let seen = new Map();
return arr.filter((item) => {
return !seen.has(item) && seen.set(item,1);
    });
};
unique(arr);
```

请你谈谈Cookie的弊端

1.Cookie数量和长度的限制。每个domain最多只能有20条cookie，每个cookie长度不能超过4KB，否则会被截掉。

2.安全性问题。如果cookie被人拦截了，那人就可以取得所有的session信息。即使加密也与事无补，因为拦截者并不需要知道cookie的意义，他只要原样转发cookie就可以达到目的了。

3.有些状态不可能保存在客户端。例如，为了防止重复提交表单，我们需要在服务器<端保存一个计数器。如果我们把这个计数器保存在客户端，那么它起不到任何作用。

JS实现九九乘法表

```
<style>
html,body,ul,li {
padding: 0;
margin: 0;
border: 0;
}
ul {
width: 900px;
overflow: hidden;
margin-top: 4px;
font-size: 12px;
line-height: 36px;
}
li {
float: left;
width: 90px;
margin: 0 4px;
display: inline-block;
text-align: center;
border: 1px solid #333;
background:yellowgreen;
}
</style>

<script>
```

```
for(var i = 1; i <= 9; i++){
var myUl = document.createElement('ul');
for(var j = 1; j <= i; j++){
var myLi = document.createElement('li');
var myText = document.createTextNode(j + " × " + i + " = " + i*j);
myLi.appendChild(myText);
myUl.appendChild(myLi);
}
document.getElementsByTagName('body')[0].appendChild(myUl);
}
</script>
```

以下代码运行输出为（A ）

```
var a = [1, 2, 3],
    b = [1, 2, 3],
    c = [1, 2, 4];
    console.log(a == b);
    console.log(a === b);
    console.log(a > c);
    console.log(a < c);
```

A: false, false, false, true B: false, false, false, false C: true, true, false, true D: other

解析:

> JavaScript中Array的本质也是对象，所以前两个的结果都是false， 而JavaScript中Array的'>'运算符和'<'运算符的比较方式类似于字符串比较字典序，会从第一个元素开始进行比较，如果一样比较第二个，还一样就比较第三个，如此类推，所以第三个结果为false，第四个为true。 综上所述，结果为false, false, false, true，选A

以下代码运行结果为(D)

```
var val = 'smtg';
console.log('Value is ' + (val === 'smtg') ? 'Something' : 'Nothing');
```

A: Value is Something B: Value is Nothing C: NaN D: other

解析:

> 这题考的javascript中的运算符优先级，这里'+'运算符的优先级要高于'?'所以运算符，实际上是 'Value is true'?'Something' : 'Nothing'，当字符串不为空时，转换为bool为true，所以结果为'Something'，选D

[,,,].join(", ")运行结果为 (C)

A: ", , , " B: "undefined, undefined, undefined, undefined" C: ", , " D: ""

解析:

> JavaScript中使用字面量创建数组时，如果最末尾有一个逗号‘,’，会被省略，所以实际上这个数组只有三个元素（都是undefined）： console.log([,].length);//输出结果：//3 而三个元素，使用join方法，只需要添加两次，所以结果为", , "，选C

以下代码运行结果为(D)

```
function sidEffecting(ary) {
  ary[0] = ary[2];
}
function bar(a,b,c) {
  c = 10
  sidEffecting(arguments);
  return a + b + c;
}
bar(1,1,1)
```

A: 3 B: 12 C: error D: other

解析:

> 这题考的是JS的函数arguments的概念： 在调用函数时，函数内部的arguments维护着传递到这个函数的参数列表。它看起来是一个数组，但实际上它只是一个有length属性的Object，不从Array.prototype继承。所以无法使用一些Array.prototype的方法。 arguments对象其内部属性以及函数形参创建getter和setter方法，因此改变形参的值会影响到arguments对象的值，反过来也是一样 具体例子可以参见Javascript秘密花园#arguments 所以，这里所有的更改都将生效，a和c的值都为10，a+b+c的值将为21，选D

以下代码运行结果为(A)

```
var name = 'World!';
(function () {
    if (typeof name === 'undefined') {
        var name = 'Jack';
        console.log('Goodbye ' + name);
    } else {
        console.log('Hello ' + name);
    }
})();
```

A: Goodbye Jack B: Hello Jack C: Hello undefined D: Hello World

解析:

> 这题考的是javascript作用域中的变量提升，javascript的作用于中使用var定义的变量都会被提升到所有代码的最前面，于是乎这段代码就成了： var name = ‘World!’; (function () { var name;//现在还是undefined if (typeof name === ‘undefined’) { name = ‘Jack’; console.log('Goodbye ’ + name); } else {

> console.log('Hello ' + name); } })(); 这样就很好理解了，typeof name === 'undefined'的结果为true，所以最后会输出'Goodbye Jack'，选A

以下代码运行结果为(B)

```
var a = [0];
if ([0]) {
  console.log(a == true);
} else {
  console.log("wut");
}
```

A: true B: false C: "wut" D: other

解析:

> 同样是一道隐式类型转换的题，不过这次考虑的是''运算符，a本身是一个长度为1的数组，而当数组不为空时，其转换成bool值为true。 而左右的转换，会使用如果一个操作值为布尔值，则在比较之前先将其转换为数值的规则来转换，Number([0])，也就是0，于是变成了0 == true，结果自然是false，所以最终结果为B

以下代码运行结果为(D)

```
var ary = Array(3);
ary[0]=2
ary.map(function(elem) { return '1'; });
```

A: [2, 1, 1] B: ["1", "1", "1"] C: [2, "1", "1"] D: other

解析:

> 又是考的Array.prototype.map的用法，map在使用的时候，只有数组中被初始化过元素才会被触发，其他都是undefined，所以结果为[“1”, undefined ,undefined]，选D

以下代码运行结果为（B）

```
var a = 11111111111111110000,
    b = 1111;
    console.log(a + b);
```

A: 11111111111111111111 B: 11111111111111110000 C: NaN D: Infinity

解析: 又是一道考查JavaScript数字的题，由于JavaScript实际上只有一种数字形式IEEE 754标准的64位双精度浮点数，其所能表示的整数范围为-253~253(包括边界值)。这里的11111111111111110000已经超过了2^53次方，所以会发生精度丢失的情况。综上选B

以下代码运行结果为(C)

```
(function(){
  var x = y = 1;
})();
console.log(y);
console.log(x);
```

A: 1, 1 B: error, error C: 1, error D: other

解析:

> 变量提升和隐式定义全局变量的题，也是一个JavaScript经典的坑... 还是那句话，在作用域内，变量定义和函数定义会先行提升，所以里面就变成了: (function(){ var x; y = 1; x = 1; })(); 这点会问了，为什么不是var x, y;，这就是坑的地方...这里只会定义第一个变量x，而y则会通过不使用var的方式直接使用，于是乎就隐式定义了一个全局变量y 所以，y是全局作用域下，而x则是在函数内部，结果就为1, error，选C

手写 call、apply 及 bind 函数

```
call函数实现

 Function.prototype.myCall = function (context) {
   // 判断调用对象
   if (typeof this !== "function") {
     console.error("type error");
   }

   // 获取参数
   let args = [...arguments].slice(1),
     result = null;

   // 判断 context 是否传入，如果未传入则设置为 window
   context = context || window;

   // 将调用函数设为对象的方法
   context.fn = this;

   // 调用函数
   result = context.fn(...args);

   // 将属性删除
   delete context.fn;

   return result;
 }


 apply 函数实现
```

```
Function.prototype.myApply = function (context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }

  let result = null;

  // 判断 context 是否存在，如果未传入则为 window
  context = context || window;

  // 将函数设为对象的方法
  context.fn = this;

  // 调用方法
  if (arguments[1]) {
    result = context.fn(...arguments[1]);
  } else {
    result = context.fn();
  }

  // 将属性删除
  delete context.fn;

  return result;
}


bind 函数实现

Function.prototype.myBind = function (context) {
  // 判断调用对象是否为函数
  if (typeof this !== "function") {
    throw new TypeError("Error");
  }

  // 获取参数
  var args = [...arguments].slice(1),
    fn = this;

  return function Fn() {

    // 根据调用方式，传入不同绑定值
    return fn.apply(this instanceof Fn ? this : context,
args.concat(...arguments));
  }

}
```

事件委托是什么?

答案

> 事件委托本质上是利用了浏览器事件冒泡的机制。因为事件在冒泡过程中会上传到父节点，并且父节点可以通过事件对象获取到 目标节点，因此可以把子节点的监听函数定义在父节点上，由父节点的监听函数统一处理多个子元素的事件，这种方式称为事件 代理。
>
> 使用事件代理我们可以不必要为每一个子元素都绑定一个监听事件，这样减少了内存上的消耗。并且使用事件代理我们还可以实 现事件的动态绑定，比如说新增了一个子节点，我们并不需要单独地为它添加一个监听事件，它所发生的事件会交给父元素中的 监听函数来处理。

javascript 代码中的 "use strict"; 是什么意思？使用它区别是什么?

> use strict 指的是严格运行模式，在这种模式对 js 的使用添加了一些限制。比如说禁止 this 指向全局对象，还有禁止使 用 with 语句等。设立严格模式的目的，主要是为了消除代码使用中的一些不安全的使用方式，也是为了消除 js 语法本身的一 些不合理的地方，以此来减少一些运行时的怪异的行为。同时使用严格运行模式也能够提高编译的效率，从而提高代码的运行速度。 我认为严格模式代表了 js 一种更合理、更安全、更严谨的发展方向。

相关知识点：

> use strict 是一种 ECMAscript5 添加的（严格）运行模式，这种模式使得 Javascript 在更严格的条件下运行。
>
> 设立"严格模式"的目的，主要有以下几个：
>
> - 消除 Javascript 语法的一些不合理、不严谨之处，减少一些怪异行为;
> - 消除代码运行的一些不安全之处，保证代码运行的安全；
> - 提高编译器效率，增加运行速度；
> - 为未来新版本的 Javascript 做好铺垫。
>
> 区别：
>
> （1）禁止使用 with 语句。 （2）禁止 this 关键字指向全局对象。 （3）对象不能有重名的属性。

手写async await

```
function asyncToGenerator(generatorFunc) {
    return function() {
      const gen = generatorFunc.apply(this, arguments)
      return new Promise((resolve, reject) => {
        function step(key, arg) {
          let generatorResult
          try {
            generatorResult = gen[key](arg)
          } catch (error) {
            return reject(error)
          }
          const { value, done } = generatorResult
          if (done) {
            return resolve(value)
```

```
            } else {
              return Promise.resolve(value).then(val => step('next', val), err =>
step('throw', err))
            }
          }
          step("next")
        })
      }
}
```

解析

```
function asyncToGenerator(generatorFunc) {
  // 返回的是一个新的函数
  return function() {

    // 先调用generator函数 生成迭代器
    // 对应 var gen = testG()
    const gen = generatorFunc.apply(this, arguments)

    // 返回一个promise 因为外部是用.then的方式 或者await的方式去使用这个函数的返回值的
    // var test = asyncToGenerator(testG)
    // test().then(res => console.log(res))
    return new Promise((resolve, reject) => {

      // 内部定义一个step函数 用来一步一步的跨过yield的阻碍
      // key有next和throw两种取值，分别对应了gen的next和throw方法
      // arg参数则是用来把promise resolve出来的值交给下一个yield
      function step(key, arg) {
        let generatorResult

        // 这个方法需要包裹在try catch中
        // 如果报错了 就把promise给reject掉 外部通过.catch可以获取到错误
        try {
          generatorResult = gen[key](arg)
        } catch (error) {
          return reject(error)
        }

        // gen.next() 得到的结果是一个 { value, done } 的结构
        const { value, done } = generatorResult

        if (done) {
          // 如果已经完成了 就直接resolve这个promise
          // 这个done是在最后一次调用next后才会为true
          // 以本文的例子来说 此时的结果是 { done: true, value: 'success' }
          // 这个value也就是generator函数最后的返回值
          return resolve(value)
        } else {
          // 除了最后结束的时候外，每次调用gen.next()
          // 其实是返回 { value: Promise, done: false } 的结构，
```

```
            // 这里要注意的是Promise.resolve可以接受一个promise为参数
            // 并且这个promise参数被resolve的时候，这个then才会被调用
            return Promise.resolve(
              // 这个value对应的是yield后面的promise
              value
            ).then(
              // value这个promise被resove的时候，就会执行next
              // 并且只要done不是true的时候 就会递归的往下解开promise
              // 对应gen.next().value.then(value => {
              //    gen.next(value).value.then(value2 => {
              //       gen.next()
              //
              //      // 此时done为true了 整个promise被resolve了
              //      // 最外部的test().then(res => console.log(res))的then就开始执行
了
              //    })
              // })
              function onResolve(val) {
                step("next", val)
              },
              // 如果promise被reject了 就再次进入step函数
              // 不同的是，这次的try catch中调用的是gen.throw(err)
              // 那么自然就被catch到 然后把promise给reject掉啦
              function onReject(err) {
                step("throw", err)
              },
            )
          }
        }
        step("next")
      })
    }
}
```

下列关于获取页面元素说法正确的是【多选】（AC）

A.document.getElementById ('a') 是通过 id 值为 a 获取页面中的一个元素

B.document.getElementsByName ("na") 是通过 name 属性值为 na 获取页面中的一个元素

C.document.getElementsByTagName ("div") 是通过标签名获取所有 div;

D.以上说法都不正确

解析：getElementsByName相似于getElementById，只不过在这里是通过na属性查询而不是通过name属性查询。

下面代码的输出是什么？（D）

```
function SayHi(){
    console.log(name);
    console.log(age);
    var name = 'xiannv';
    let age = '18'
}
SayHi()
```

A.TJH 和 undefined

B.TJH 和 ReferenceError

C.ReferenceError 和 24

D.undefined 和 ReferenceError

解析：这道题考点作用域，用var声明了name 所以把name声明提升到本作用域内最上面,在打印name时,已经声明了 但没赋值所以打印name时name为未定义，而let不会预解析则打印age时还未声明，就会报引用错误。

下面代码的输出是什么？ (C)

```
1   let obj1 = {
2     name:'obj1_name',
3     print:function(){
4         return ()=>console.log(this.name)
5     }
6   }
7   let obj2 = {name:'obj2_name'}
8   obj1.print()();
9   obj1.print().call(obj2);
10  obj1.print.call(obj2)();
```

A. obj1_name obj2_name obj2_name

B. obj2_name obj1_name obj2_name

C. obj1_name obj1_name obj2_name

D. obj2_name obj2_name obj1_name

解析：第8行代码打印为'obj1_name'，obj1.print()执行返回值为第4行的箭头函数，再加一个括号时obj1.print()()；就是第4行的函数执行了 打印为this.name，this指向看声明时作用域则this指向obj1，打印的this.name也就是obj1的name了。

第9行则是在obj1.print()执行返回值为第4行的整个函数，但在这里第4行为箭头函数 ，箭头函数特性this指向继承父级函数this指向 所以在这里call给箭头函数绑定this是没有用的，this.name依然为'obj1_name'。

第10行代码打印则直接把this指向绑定在obj1.print函数上面了 ，箭头函数执行时则 继承父级this指向就指向了obj2，打印的this.name为'obj2_name'。

最后结果为 obj1_name obj1_name obj2_name

window的主对象主要有几个：(B)

A. 4

B. 5

C. 6

D. 7

解析：window五大对象，Navigator 、Screen 、History 、Location 、Window。

将一个整数序列整理为升序，两趟处理后变为10,12,21,9,7,3,4,25，则采用的排序算法可能Z:（C）

A、插入排序

B、快速排序

C、选择排序

D、堆排序

解析：

插入排序：基于某序列已经有序排列的情况下，通过一次插入一个元素的方式按照原有排序方式增加元素。

快速排序：通过一趟排序算法把所需要排序的序列的元素分割成两大块，其中，一部分的元素都要小于或等于另外一部分的序列元素，然后仍根据该种方法对划分后的这两块序列的元素分别再次实行快速排序算法，排序实现的整个过程可以是递归的来进行调用，最终能够实现将所需排序的无序序列元素变为一个有序的序列。

选择排序：第一次从待排序的数据元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素的个数为零。选择排序是不稳定的排序方法。

堆排序：指利用堆这种数据结构所设计的一种排序算法。堆是一个近似完全二叉树的结构，并同时满足堆积的性质索引总是小于（或者大于）它的父节点。堆中的最大值总是位于根节点（在优先队列中使用堆的话堆中的最小值位于根节点）。

下面代码的输出是什么？(B)

```
let obj = {
  name: "Lydia".
  age: 21
}

for (let item in obj) {
  console.log(item)
}
```

A. {name: "Lydia"},{age: 21}

B. ”name", "age"

C. "Lydia",21

D. ["name", "Lydia"] ["age", 21]

解析：for-in是遍历对象自身属性和方法的一个方法，let key in object，这里的key对应的就是遍历对象的key值，所以应该选B；

下面代码的输出结果是什么？(B)

```
const person = {
    firstName: 'Lydia',
    lastname: 'Hallie',
    pet: {
        name: 'Mara',
        breed: 'Dutch Tulip Hound'
    },
    getFullName() {
        return `${this.firstName}/${this.lastName}`
    }
}

console.log(person.pet?.name)
console.log(person.pet?.family?.name)
console.log(person.getFullName?.())
console.log(person.getLastName?.())
```

A. undefined undefined undefined undefined

B. Mara undefined Lydia/Hallie undefined

C. Mara null Lydia/Hallie null

D. null ReferenceError null ReferenceError null

解析：首先?.是可选操作链，es2020新特性，当?前面的属性不为undefined或null时才会继续执行后续代码，否则直接返回左侧结果；所以看结果第一个person.pet?.name pet是已存在属性，name也是存在的属性，所以会返回'Mara' 排除A和D；其次对象上查找属性时若属性不存在则返回的是undefined，故选B；

下面代码的输出结果是什么？(A)

```
console.log(+true);
console.log(!'Lydia');
```

A. 1 false

B. false NaN

C. NaN false

D. 1 NaN

解析：这是两个类型转换的操作，第一个是把true转成数字类型，true转为数字类型返回1，所以排除B和C；其次对一个非空字符串取反，值为false，所以选A；

下面代码执行的输出结果是什么？(A)

```
const myLife = ['游戏', '食物', '睡觉', '工作'];

for (let item in myLife) {
  console.log(item);
}

for (let item of myLife) {
  console.log(item);
}
```

A. 0123 和 '游戏''食物' '睡觉''工作'

B. '游戏''食物' '睡觉''工作' 和 '游戏''食物' '睡觉''工作'

C. '游戏''食物' '睡觉''工作' 和 0123

D. 0123 和 {0: '游戏', 1: '食物', 2: '睡觉', 3: '工作'}

解析：for...in的作用是枚举对象 for...of的作用是遍历可遍历数据，使用for...in时 (let item in myLife) 这块的item代表的是被枚举对象的key值，当枚举的对象为数组时，key值对应索引值，所以for...in的结果是0123，而for...of中的item为被遍历的每一项，所以结果为'游戏' '食物' '睡觉' '工作'，故选A；

下面代码输出结果是什么？(D)

```
[1, 2, 3, 4].reduce((x, y) => console.log(x, y))
```

A. 1, 2 、3,3、6,4

B. 1,2、2,3、3,4

C. 1,undefined、2,undefined、3,undefined、4,undefined

D. 1,2、undefined,3、undefined，4

解析：首先4项 数组，由于没有传入初始值，所以第一次执行的时候会直接传入1、2，所以C直接排除，接下来每次执行的时候，由于回调中并没有返回任何值，所以后续的x值全部都是undefined，故选C；

使用哪个构造函数可以成功继承Dog类？(B)

```
class Dog {
  constructor(name) {
    this.name = name;
  }
};

class Labrador extends Dog {
  // 1
  constructor(name, size) {
    this.size = size;
  }
  // 2
  constructor(name, size) {
    super(name);
    this.size =  size;
  }
  // 3
  constructor(size) {
    super(name);
    this.size = size;
  }
  // 4
  constructor(name, size) {
    this.name = name;
    this.size = size;
  }

}
```

A. 1

B. 2

C. 3

D. 4

解析：在子类中，在调用super之前不能访问到this关键字，如果这样做将会抛出错误，所以A和D排除；使用super的时候，需要用给定的参数来调用父类的构造函数，父类的构造函数接受name参数，所以我们需要将name传给super；同时子类Labrador需要接受两个参数，一个是name继承子父类，另一个是size，是他的额外属性，故选B；

下面代码的输出结果是什么？(B)

```
const a = {};
const b = { key: 'b' };
const c = { key: 'c' };

a[b] = 123;
a[c] = 456;
```

```
    console.log(a[b]);
```

A. 123

B. 456

C. undefined

D. ReferenceError

解析：首先我们是想将一个对象设置为对象a的键，值为123，但是使用对象的时候会自动转换为字符串，而对象转成字符串是[Object object] ['[Object object]'] = 123；同理，a[c] [Object object]，所以后面的赋值会覆盖前一次操作；打印的时候a[b]也是同理，所以自然结果是B；

下面代码的输出结果是什么？请简述原因。

```
var a = 10;
(function () {
 console.log(a);
 a = 5;
 console.log(window.a);
 var a = 20;
 console.log(a);
})();
```

答案：打印顺序为undefined -> 10 -> 20；是由于var的预解析以及作用域导致。

解析：在全局中通过var声明的变量会挂载在window对象上，所以全局中的var a = 10;会直接挂载在window上，自执行函数内部有作用域问题，var的预解析会提前到第一个console之前，所以第一个console的结果是undefined，第二个是全局中挂载的10，继续向下执行，首先是a = 5的赋值，但是再最后还会有一个a = 20的赋值，所以最后一次打印结果是20；

将数组扁平化并去重，最终得到一个升序且不重复的数组

```
var arr = [[1, 2, 2] [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14]]]], 10];
```

参考答案：`[...new Set(arr.flat(Infinity))].sort((a, b) => a - b)`

解析：实现方法有很多，这里用一个相对最简单的方案，首先通过Array.prototype.flat()方法，传入Infinity将数组无限级降为一维数组，然后通过Set类型的不重复数据，将数组项去重，最后直接通过Array.prototype.sort方法将数组升序排列即可；

两个打印的结果分别是什么？

```
var scope = "global";
function fn(){
  console.log(scope);
```

```
  var scope = "local";
  console.log(scope);
}
fn();
```

参考答案：undefined scope 解析：

只要函数内定义了一个局部变量，函数在解析的时候都会将这个变量“提前声明”

等同于下面这样写：

```
var scope = "global";
function fn(){
  var scope;        //提前声明了局部变量
  console.log(scope); //undefined
  scope = "local";
  console.log(scope); //local;
}
fn();
```

foo = foo||bar，这行代码是什么意思？为什么要这样写？

解析：

如果foo存在，值不变，否则把bar的值赋给foo。

短路表达式：作为”&&”和”||”操作符的操作数表达式，这些表达式在进行求值时， 只 要最终的结果已经可以确定是真或假，求值过程便告终止，这称之为短路求值。

执行以下代码，输出内容分别是什么？

```
let x = 5;
function fn(x) {
    return function(y) {
        console.log(y + (++x));
    }
}
let f = fn(6);
f(7);
console.log(x);
```

A、14，5

B、13，5

C、14，6

D、13，6

解析：

答案：A

这道题考查知识点为作用域、闭包。

依次执行这段代码，首先调用**fn(6)**，结果赋值给**f**，那么**f**即为返回的这个函数，然后调用这个函数并传入7，则打印的**y**的值为7，而自增的**x**的值其实是调用**fn**时传入的值为5，原因是这个函数声明在**fn**函数内，根据作用域链得到了**x**的值为5，**x**自增后为6，加上**y**的值，最终打印结果为14。最后打印的**x**就是在全局声明的**x**值为5。所以正确答案为A选项。

ES6的class和构造函数的区别

解析

class 的写法只是语法糖，和之前 **prototype** 差不多，但还是有细微差别的，下面看看：

- **严格模式**

类和模块的内部，默认就是严格模式，所以不需要使用**use strict**指定运行模式。只要你的代码写在类或模块之中，就只有严格模式可用。考虑到未来所有的代码，其实都是运行在模块之中，所以 ES6 实际上把整个语言升级到了严格模式。

- **不存在变量提升**

类不存在变量提升（hoist），这一点与 ES5 完全不同。

```
new Foo(); // ReferenceError
class Foo {}
```

- **方法默认是不可枚举的**

ES6 中的 **class**，它的方法（包括静态方法和实例方法）默认是不可枚举的，而构造函数默认是可枚举的。细想一下，这其实是个优化，让你在遍历时候，不需要再判断 **hasOwnProperty**了

- class的所有方法（包括静态方法和实例方法）都没有原型对象**prototype**，不能用使用**new**调用
- class必须使用new调用，否则会报错。这是它跟普通构造函数的一个主要区别，后者不需要**new**也可以执行
- **ES5和ES6子类this生成顺序不同**

ES5 的继承先生成了子类实例，再调用父类的构造函数修饰子类实例。ES6 的继承先 生成父类实例，再调用子类的构造函数修饰父类实例。这个差别使得 ES6 可以继承内置对象。

- ES6能继承静态方法，而构造函数不能

介绍Promise以及Promise的几种状态

**介绍: **Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。所谓Promise，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，Promise 是一个对象，从它可以获取异步操作的消息。Promise 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

状态: pending（进行中）、fulfilled（已成功）和rejected（已失败）

使用Promise实现红绿灯交替重复亮

红灯3秒亮一次，黄灯2秒亮一次，绿灯1秒亮一次；如何让三个灯不断交替重复亮灯？（用Promise实现）

三个亮灯函数已经存在：

```
function red() {
    console.log('red');
}
function green() {
    console.log('green');
}
function yellow() {
    console.log('yellow');
}
```

解析：

红灯3秒亮一次，绿灯1秒亮一次，黄灯2秒亮一次，意思就是3秒执行一次red函数，2秒执行一次green函数，1秒执行一次yellow函数，不断交替重复亮灯，意思就是按照这个顺序一直执行这3个函数，这步可以利用递归来实现。

```
function red() {
console.log('red');
}
function green() {
console.log('green');
}
function yellow() {
console.log('yellow');
}

var light = function (timmer, cb) {
return new Promise(function (resolve, reject) {
  setTimeout(function () {
      cb();
      resolve();
  }, timmer);
});
};

var step = function () {
Promise.resolve().then(function () {
```

```
    return light(3000, red);
  }).then(function () {
    return light(2000, green);
  }).then(function () {
    return light(1000, yellow);
  }).then(function () {
    step();
  });
  }

  step();
```

为什么在 JavaScript中比较两个相似的对象时返回 false?

```
let a = { a: 1 };
let b = { a: 1 };
let c = a;
console.log(a === b); // 打印 false，即使它们有相同的属性
console.log(a == b);  // false
console.log(a === c); // true
```

解析：

JavaScript 以不同的方式比较对象和基本类型。在基本类型中，JavaScript 通过值对它们进行比较，而在对象中，JavaScript 通过引用或存储变量的内存中的地址对它们进行比较。这就是为什么第一个和第二个`console.log`语句返回`false`，而第二个`console.log`语句返回`true`。`a`和`c`有相同的引用地址，而`a`和`b`没有。

为什么在调用这个函数时，代码中的`b`会变成一个全局变量?

```
function myFunc() {
  let a = b = 0;
}
myFunc();
```

解析：

原因是赋值运算符是从右到左的求值的。这意味着当多个赋值运算符出现在一个表达式中时，它们是从右向左求值的。所以上面代码变成了这样：

```
function myFunc() {
let a = (b = 0);
}
myFunc();
```

> 首先，表达式b = 0求值，在本例中b没有声明。因此，JavaScript 引擎在这个函数外创建了一个全局变量b，之后表达式b = 0的返回值为0，并赋给新的局部变量a。
>
> 可以通过在赋值之前先声明变量来解决这个问题。

```
function myFunc() {
let a,b;
a = b = 0;
}
myFunc();
```

var，let和const`的区别是什么?

> 解析：
>
> (1) var声明的变量会挂载在window上，而let和const声明的变量不会：

```
var a = 100;
console.log(a,window.a);    // 100 100

let b = 10;
console.log(b,window.b);    // 10 undefined

const c = 1;
console.log(c,window.c);    // 1 undefined
```

> (2) var声明变量存在变量提升，let和const不存在变量提升:

```
console.log(a); // undefined  ===>  a已声明还没赋值，默认得到undefined值
var a = 100;

console.log(b);
// 报错：Cannot access 'b' before initialization
// => 无法在初始化之前访问“b”
let b = 10;

console.log(c);
// 报错：Cannot access 'c' before initialization
const c = 10;
```

> (3) let和const声明形成块作用域

```
if(1){
var a = 100;
let b = 10;
}
```

```
console.log(a); // 100
console.log(b)  // 报错: b is not defined  ===> 找不到b这个变量

------------------------------------------------------------

if(1){
var a = 100;
const c = 1;
}
console.log(a); // 100
console.log(c)  // 报错: c is not defined  ===> 找不到c这个变量
```

(4) 同一作用域下let和const不能声明同名变量，而var可以

```
var a = 100;
console.log(a); // 100

var a = 10;
console.log(a); // 10
-------------------------------------
let a = 100;
let a = 10;
//  控制台报错: Identifier 'a' has already been declared  ===> 标识符a已经被声明了。
```

(5) 暂存死区

```
var a = 100;

if(1){
a = 10;
// 在当前块作用域中存在a使用let/const声明的情况下，给a赋值10时，只会在当前作用域找变量a,
// 而这时，还未到声明时候，所以控制台Cannot access 'a' before initialization
let a = 1;
}
```

(6) const

```
/*
* 1、一旦声明必须赋值,不能使用null占位。
* 2、声明后不能再修改
* 3、如果声明的是复合类型数据，可以修改其属性
*
*/
```

```
const a = 100;

const list = [];
list[0] = 10;
console.log(list);   // [10]

const obj = {a:100};
obj.name = 'apple';
obj.a = 10000;
console.log(obj);  // {a:10000,name:'apple'}
```

什么时候不使用箭头函数? 说出三个或更多的例子?

解析:

（1）当想要函数被提升时(箭头函数是匿名的)

（2）要在函数中使用this/arguments时，由于箭头函数本身不具有this/arguments，因此它们取决于外部上下文

（3）使用命名函数(箭头函数是匿名的)

（4）使用函数作为构造函数时(箭头函数没有构造函数)

（5）当想在对象字面是以将函数作为属性添加并在其中使用对象时，因为咱们无法访问 this 即对象本身。

Object.freeze() 和 const 的区别是什么?

解析:

const和Object.freeze是两个完全不同的概念。

const 声明一个只读的变量，一旦声明，常量的值就不可改变:

```
const person = {
name: "Leonardo"
};
let animal = {
species: "snake"
};
person = animal; // ERROR "person" is read-only
```

Object.freeze适用于值，更具体地说，适用于对象值，它使对象不可变，即不能更改其属性。

```
let person = {
name: "Leonardo"
};
let animal = {
```

```
species: "snake"
};
Object.freeze(person);
person.name = "Lima"; //TypeError: Cannot assign to read only property
'name' of object
console.log(person);
```

如何在 JS 中创建对象?

解析:

(1) 使用对象字面量:

```
let obj = {
    name:"张三",
}
console.log(obj);          // {name: "张三"}
```

(2) 使用构造函数:

```
let obj = new Object();
obj.name = "张三";
console.log(obj);          // {name: "张三"}
```

(3) 使用 Object.create 方法:

```
let obj = Object.create({
name:"张三",
});
console.log(obj.name);     // 张三
```

{} 和 [] 的 valueOf 和 toString 的结果是什么?

解析:

{} 的 valueOf 结果为 {}，toString 的结果为 "[object Object]"

[] 的 valueOf 结果为 []，toString 的结果为 ""

js垃圾回收方法

标记清除（mark and sweep）

- 这是JavaScript最常见的垃圾回收方式，当变量进入执行环境的时候，比如函数中声明一个变量，垃圾回收器将其标记为“进入环境”，当变量离开环境的时候（函数执行结束）将其标记为“离开环境”。

- 垃圾回收器会在运行的时候给存储在内存中的所有变量加上标记，然后去掉环境中的变量以及被环境中变量所引用的变量（闭包），在这些完成之后仍存在标记的就是要删除的变量了

引用计数(reference counting)

- 在低版本IE中经常会出现内存泄露，很多时候就是因为其采用引用计数方式进行垃圾回收。引用计数的策略是跟踪记录每个值被使用的次数，当声明了一个 变量并将一个引用类型赋值给该变量的时候这个值的引用次数就加1，如果该变量的值变成了另外一个，则这个值得引用次数减1，当这个值的引用次数变为0的时 候，说明没有变量在使用，这个值没法被访问了，因此可以将其占用的空间回收，这样垃圾回收器会在运行的时候清理掉引用次数为0的值占用的空间。
- 在IE中虽然JavaScript对象通过标记清除的方式进行垃圾回收，但BOM与DOM对象却是通过引用计数回收垃圾的， 也就是说只要涉及BOM及DOM就会出现循环引用问题。

请实现一个方法将data结构转换为tree结构。

```
let data = [
  {"parent_id": null, "id": 'a', 'value': 'xxxx'},
  {"parent_id": 'a', "id": 'c', 'value': 'xxxx'},
  {"parent_id": 'd', "id": 'f', 'value': 'xxxx'},
  {"parent_id": 'c', "id": 'e', 'value': 'xxxx'},
  {"parent_id": 'b', "id": 'd', 'value': 'xxxx'},
  {"parent_id": 'a', "id": 'b', 'value': 'xxxx'},
];
let tree = {
  'a': {
    value: 'xxx',
    children: {
      'b': {
        value: 'xxx',
        children: {
          'd': {
            value: 'xxx',
            children: {
              'f': {
                value: 'xxx'
              }
            }
          }
        }
      },
      'c': {
        value: 'xxx',
        children: {
          'e': {
            value: 'xxx'
          }
        }
      }
    }
  }
};
```

参考答案：

```
function transformToTree(data) {
        let result = {};
        let newData;
        data.sort((a, b) => a.id.codePointAt() - b.id.codePointAt());
        newData = data.filter(item => !item.parent_id);
        newData.forEach(item => {
            result[item.id] = {
                value: item.value
            };
        });
        function loop(result) {
            for (const key in result) {
                let newData = data.filter(item => item.parent_id === key);
                if (newData.length) {
                    newData.forEach(item => {
                        result[key].children = result[key].children || {};
                        result[key].children[item.id] = {
                            value: item.value
                        };
                        loop(result[key].children);
                    });
                }
            }
        }
        loop(result);
        return result;
    }
```

下面代码的输出结果是什么?

```
function fn(n, o) {
  console.log(o);
  return {
    fn: function (m) {
      return fn(m, n);
    }
  }
}
fn(0).fn(1).fn(2).fn(3); // =>?
```

解析：

这块连续调用.fn只是一个迷惑作用，其实可以看做打印值都是上一次调用的传参，第一次调用没有第二个参数，所以首先打印一个undefined，之后每次打印都是前一次的参数，所以输出结果应该是 undefined 0 1 2。

原型

解析：

所有的函数数据类型都天生自带一个prototype属性，该属性的属性值是一个对象 prototype的属性值中天生自带一个constructor属性，其constructor属性值指向当前原型所属的类 所有的对象数据类型，都天生自带一个_proto_属性，该属性的属性值指向当前实例所属类的原型

什么是闭包，如何使用它，为什么要使用它?

> 闭包就是能够读取其他函数内部变量的函数。由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成"定义在一个函数内部的函数"。
>
> 它的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。
>
> 使用闭包的注意点：· 由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。
>
> 解决方法是，在退出函数之前，将不使用的局部变量全部删除。 ·
>
> 闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值

如何规避javascript多人开发函数重名问题?

> 命名空间 封闭空间 js模块化mvc（数据层、表现层、控制层） seajs（如果了解的呢，可以说） 变量转换成对象的属性 对象化

虚拟 DOM 的优缺点?

解析：

- 优点：

保证性能下限： 框架的虚拟 DOM 需要适配任何上层 API 可能产生的操作，它的一些 DOM 操作的实现必须是普适的，所以它的性能并不是最优的；但是比起粗暴的 DOM 操作性能要好很多，因此框架的虚拟 DOM 至少可以保证在你不需要手动优化的情况下，依然可以提供还不错的性能，即保证性能的下限； 无需手动操作 DOM： 我们不再需要手动去操作 DOM，只需要写好 View-Model 的代码逻辑，框架会根据虚拟 DOM 和 数据双向绑定，帮我们以可预期的方式更新视图，极大提高我们的开发效率； 跨平台： 虚拟 DOM 本质上是 JavaScript 对象,而 DOM 与平台强相关，相比之下虚拟 DOM 可以进行更方便地跨平台操作，例如服务器渲染、weex 开发等等。

- 缺点:

无法进行极致优化： 虽然虚拟 DOM + 合理的优化，足以应对绝大部分应用的性能需求，但在一些性能要求极高的应用中虚拟 DOM 无法进行针对性的极致优化。

VUE

虚拟DOM的优劣如何?

优点:

- 保证性能下限: 虚拟DOM可以经过diff找出最小差异,然后批量进行patch,这种操作虽然比不上手动优化,但是比起粗暴的DOM操作性能要好很多,因此虚拟DOM可以保证性能下限
- 无需手动操作DOM: 虚拟DOM的diff和patch都是在一次更新中自动进行的,我们无需手动操作DOM,极大提高开发效率
- 跨平台: 虚拟DOM本质上是JavaScript对象,而DOM与平台强相关,相比之下虚拟DOM可以进行更方便地跨平台操作,例如服务器渲染、移动端开发等等

缺点:

- 无法进行极致优化: 在一些性能要求极高的应用中虚拟DOM无法进行针对性的极致优化,比如VScode采用直接手动操作DOM的方式进行极端的性能优化

Vue中computed和watch的区别?

> 答案：computed 是计算属性，依赖其他属性计算值，并且 computed 的值有缓存，只有当计算值变化才会返回内容。 watch 监听到值的变化就会执行回调，在回调中可以进行一些逻辑操作。

Vue中$route和$router的区别?

> 答案：
>
> $route 是“路由信息对象”，包括 path，params，hash，query，fullPath，matched，name 等路由信息参数。
>
> $router是“路由实例”对象包括了路由的跳转方法，钩子函数等。

vue有什么生命周期？在new Vue 到 vm.$destory的过程经历了什么?

> **生命周期：**
>
> - **初始化阶段**：beforeCreate和create
> - **挂载阶段**：beforeMount和mounted
> - **更新阶段**：beforeUpdate和update
> - **卸载阶段**：beforeDestory和destory
>
> **过程**
>
> 当`new Vue()`后，首先会**初始化事**和**生命周期**，接着会执行**beforeCreate**生命周期钩子，在这个钩子里面还拿不到this.$el和this.$data`;
>
> 接着往下走会**初始化inject**和**将data的数据进行侦测也就是进行双向绑定**;
>
> 接着会执**`create钩子函数**，在这个钩子里面能够拿到this.$data还拿不到this.$el`;到这里初始化阶段就走完了。
>
> 然后会进入一个**模版编译阶段**，在这个阶段首先会判断有没有`el`选项如果有的话就继续往下走，如果没有的话会调用`vm.$mount(el)`;

> 接着继续判断有没有**template**选项，如果有的话，会将template提供的模版编译到render函数中；如果没有的话，会通过el选项选择模版；到这个编译阶段就结束了。（温馨提示：这个阶段只有完整版的Vue.js才会经历，也是就是通过cmd引入的方式；在单页面应用中，没有这个编译阶段，因为vue-loader已经提前帮编译好，因此，单页面使用的vue.js是运行时的版本）。
>
> 模版编译完之后（这里说的是完整版，如果是运行时的版本会在初始化阶段结束后直接就到挂载阶段），然后进入**挂载阶段**，在挂在阶段首先或**触发beforeMount**钩子，在这个钩子里面只能拿到this.$data还是拿不到this.$el;`
>
> 接着会执**mounted钩子**，在这个钩子里面就既能够拿到this.$el也能拿到this.$data`；到这个挂载阶段就已经走完了，整个实例也已经挂载好了。
>
> 当数据发生变更的时候，就会进入**更新阶段**，首先会触发**beforeUpdate钩子**，然后触**updated钩**，这个阶段会重新计算生成新的Vnode,然后通过patch函数里面的diff算法,将新生成的Vnode和缓存中的旧Vnode进行一个比对，最后将差异部分更新到视图中。
>
> 当**vm.$destory**被调用的时候，就会进入**卸载阶段**，在这个阶段，首先触发**beforeDestory钩子**接着触发**destoryed钩子**，在这个阶段Vue会将自身从父组件中删除，取消实例上的所有追踪并且移除所有的事件监听。
>
> 到这里Vue整个生命周期就结束了。

怎么理解vue的单向数据流?

解析

- 在**vue**中，父组件可以通过**prop**将数据传递给子组件，但这个**prop**只能由父组件来修改，子组件修改的话会抛出错误
- 如果是子组件想要修改数据，只能通过**$emit**由子组件派发事件，并由父组件接收事件进行修改

vue-Router 中 hash / history 两种模式有什么区别?

解析

hash模式会在**url**上显示'#'，而**history**模式没有

刷新页面时，**hash**模式可以正常加载到**hash**值对应的页面，**history**模式没有处理的话，会返回404，一般需要后端将所有页面都配置重定向到首页路由

兼容性上，**hash**模式可以支持低版本浏览器和IE

vue-router 中 hash / history 是如何实现的?

解析

hash模式

- **#后面hash**值的变化，不会导致浏览器向服务器发出请求，浏览器不发出请求，就不会刷新页面，同时通过监听**hashchange**事件可以知道**hash**发生了哪些变化。根据**hash**变化来实现页面的局部更新

history模式

- **history**模式的实现，主要是**HtmlS**标准发布的两个Api（**pushState**和**replaceState**），这两个Api可以改变**url**，但是不会发送请求，这样就可以监听**url**的变化来实现局部更新

vue中组件的data为什么要写成函数形式

在Vue中，组件都是可复用的，一个组件创建好后，可以在多个地方重复使用，而不管复用多少次，组件内的data都必须是相互隔离，互不影响的，如果data以对象的形式存在，由于Javascript中对象是引用类型，作用域没有隔离，因此data必须以函数的形式返回

为什么new Vue根中的data可以直接写

new Vue根组件不需要复用，因此不需要以函数方式返回

谈谈你对 keep-alive 的了解?

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，其有以下特性：

- 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。

V-model的原理是什么?

解析：

Vue的双向数据绑定是由数据劫持结合发布者订阅者实现的。 数据劫持是通过Object.defineProperty()来劫持对象数据的setter和getter操作。 在数据变动时作你想做的事

- 原理 通过Observer来监听自己的model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer和Compile之间的通信桥梁，达到数据变化->视图更新 在初始化vue实例时，遍历data这个对象，给每一个键值对利用Object.definedProperty对data的键值对新增get和set方法，利用了事件监听DOM的机制，让视图去改变数据

---

vuex的流程

解析：

页面通过mapAction异步提交事件到action。action通过commit把对应参数同步提交到mutation。 mutation会修改state中对于的值。 最后通过getter把对应值跑出去，在页面的计算属性中 通过mapGetter来动态获取state中的值

`$route`和`$router`的区别

解析：

- $route是“路由信息对象”，包括path，params，hash，query，fullPath，matched，name等路由信息参数。
- $router是“路由实例”对象包括了路由的跳转方法，钩子函数等。

react和vue的区别

解析：

=> 相同点： 1.数据驱动页面，提供响应式的试图组件 2.都有virtual DOM,组件化的开发，通过props参数进行父子之间组件传递数据，都实现了webComponents规范 3.数据流动单向，都支持服务器的渲染SSR 4.都有支持native的方法，react有React native， vue有wexx

=> 不同点： 1.数据绑定：Vue实现了双向的数据绑定，react数据流动是单向的 2.数据渲染：大规模的数据渲染，react更快 3.使用场景：React配合Redux架构适合大规模多人协作复杂项目，Vue适合小快的项目 4.开发风格：react推荐做法jsx + inline style把html和css都写在js了 vue是采用webpack + vue-loader单文件组件格式，html, js, css同一个文件

vuex中state,getter,mutation,action,module,plugins各自的用途，和用法?

解析：

- State:{ count: 0 } 保存着所有的全局变量
- Getter： 对state中的数据派生出一些状态，例如对数据进行过滤。（可以认为是store中的计算属性），会对state中的变量进行过滤再保存，只要state中的变量发生了改变，它也会发生变化，不变化的时候，读的缓存。
- Mutation：更改 Vuex 的 store 中的状态的唯一方法是提交 mutation。
- 一条重要的原则就是要记住 mutation 必须是同步函数。
- Action: Action 类似于 mutation, 不同点在于，Action 提交的是 mutation，而不是直接变更状态。Action 可以包含任意异步操作，mutation只能是同步。
- 有点不同的是Action 函数接受一个与 store 实例具有相同方法和属性的 context 对象，因此你可以调用 context.commit 提交一个 mutation，或者通过 context.state 和 context.getters 来获取 state 和 getters。
- Module： //模块，可以写很多模块，最后都引入到一个文件。分散管理。生成实例的时候 都放在Store的modules中
- plugins：插件（Plugins）是用来拓展webpack功能的，它们会在整个构建过程中生效，执行相关的任务。

Vue 2.0不再支持在v-html中使用过滤器怎么办?

解析：

①全局方法（推荐）

```
Vue.prototype.msg = function (msg) {
  return msg.replace ("\n", "<br>")
}
 <div v-html="msg(content)"></div>
```

②computed方法

```
computed: {
 content: function(msg){
  return msg.replace("\n", "<br>")
 }
}
<div>{{content}}</div>
```

③$options.filters(推荐)

```
filters: {
 msg: function(msg){
  return msg.replace(/\n/g, "<br>")
 }
},
data: {
 content: "XXXX"
}
<div v-html="$options.filters.msg(content)"></div>
```

怎么解决vue动态设置img的src不生效的问题?

解析:

```
<img class="logo" :src="logo" alt="公司logo">
  data() {
    return {
      logo:require("./../assets/images/logo.png"),
    };
  }
//因为动态添加src被当做静态资源处理了，没有进行编译，所以要加上require
```

vue中怎么重置data?

解析:

Object.assign() Object.assign（）方法用于将所有可枚举属性的值从一个或多个源对象复制到目标对象。它将返回目标对象。

```
var o1 = { a: 1 };
var o2 = { b: 2 };
var o3 = { c: 3 };
var obj = Object.assign(o1, o2, o3);
```

```
console.log(obj); // { a: 1, b: 2, c: 3 }
console.log(o1); // { a: 1, b: 2, c: 3 }, 注意目标对象自身也会改变。
```

注意：具有相同属性的对象，同名属性，后边的会覆盖前边的。

由于Object.assign()有上述特性，所以我们在Vue中可以这样使用：

Vue组件可能会有这样的需求：在某种情况下，需要重置Vue组件的data数据。此时，我们可以通过this.获取当前状态下的，通过options.data()获取该组件初始状态下的data。

然后只要使用Object.assign(this.options.data())就可以将当前状态的data重置为初始状态。

vue怎么实现强制刷新组件?

解析：

① v-if

```
当v-if的值发生变化时，组件都会被重新渲染一遍。因此，利用v-if指令的特性，可以达到强制
<comp v-if="update"></comp>
<button @click="reload()">刷新comp组件</button>
data() {
  return {
    update: true
  }
},
methods: {
  reload() {
     // 移除组件
    this.update = false
      // 在组件移除后，重新渲染组件
      // this.$nextTick可实现在DOM 状态更新后，执行传入的方法。
    this.$nextTick(() => {
      this.update = true
    })
  }
}
```

② this.$forceUpdate

```
<button @click="reload()">刷新当前组件</button>
methods: {
  reload() {
    this.$forceUpdate()
  }
}
```

vuex组件中访问state报错怎么解决?

解析：

TypeError: Cannot read property ‘state’ of undefined”

在组件中使用this.$store.state.test访问state的属性报错，是因为store的实例并未注入到所有的子组件，需修改main.js

```
new Vue({

    el: '#app',
    store, //将store注入到子组件
    router,
    components: { App },
    template: ''

})
```

Vue 的父组件和子组件生命周期钩子函数执行顺序?

解析：

Vue 的父组件和子组件生命周期钩子函数执行顺序可以归类为以下 4 部分：

- 加载渲染过程：父 beforeCreate -> 父 created -> 父 beforeMount -> 子 beforeCreate -> 子 created -> 子 beforeMount -> 子 mounted -> 父 mounted
- 子组件更新过程：父 beforeUpdate -> 子 beforeUpdate -> 子 updated -> 父 updated
- 父组件更新过程：父 beforeUpdate -> 父 updated
- 销毁过程：父 beforeDestroy -> 子 beforeDestroy -> 子 destroyed -> 父 destroyed

React

调用setState之后发生了什么?

> 答案：
>
> 在代码中调用 setState 函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发所谓的调和过程（Reconciliation）。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个 UI 界面。在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

react-router里的Link标签和a标签有什么区别

从渲染的DOM来看，这两者都是链接，都是a标签，

区别是：

Link是react-router里实现路由跳转的链接，配合Route使用，react-router拦截了其默认的链接跳转行为，区别于传统的页面跳转，

Link 的"跳转"行为只会触发相匹配的Route对应的页面内容更新，而不会刷新整个页面。 a标签是html原生的超链接，用于跳转到href指向的另一个页面或者锚点元素,跳转新页面会刷新页面。

react-hooks的优劣如何

React Hooks优点:

- 简洁: React Hooks解决了HOC和Render Props的嵌套问题,更加简洁
- 解耦: React Hooks可以更方便地把 UI 和状态分离,做到更彻底的解耦
- 组合: Hooks 中可以引用另外的 Hooks形成新的Hooks,组合变化万千
- 函数友好: React Hooks为函数组件而生,从而解决了类组件的几大问题:
    - this 指向容易错误
    - 分割在不同声明周期中的逻辑使得代码难以理解和维护
    - 代码复用成本高（高阶组件容易使代码量剧增）

React Hooks缺陷:

- 额外的学习成本（Functional Component 与 Class Component 之间的困惑）
- 写法上有限制（不能出现在条件、循环中），并且写法限制增加了重构成本
- 破坏了PureComponent、React.memo浅比较的性能优化效果（为了取最新的props和state，每次render()都要重新创建事件处函数）
- 在闭包场景可能会引用到旧的state、props值
- 内部实现上不直观（依赖一份可变的全局状态，不再那么"纯"）
- React.memo并不能完全替代shouldComponentUpdate（因为拿不到 state change，只针对 props change）

前后端交互

请解释 JSONP 的工作原理，以及它为什么不是真正的 Ajax?

解析

JSONPS 是利用当浏览器解析到该元素时，会暂停其他资源的下载和处理，直到将该资源加载、编译、执行完毕，图片和框架等元素也如此，类似于将所指向资源嵌入当前标签内。这也是为什么将js脚本放在底部而不是头部。

href是Hypertext Reference的缩写，指向网络资源所在位置，建立和当前元素（锚点）或当前文档（链接）之间的链接，如果我们在文档中添加那么浏览器会识别该文档为css文件，就会并行下载资源并且不会停止对当前文档的处理。

伪类和伪元素的区别

> 伪类和伪元素是为了修饰不在文档树中的部分，比如一句话的第一个字母，列表中第一个元素。
>
> **伪类**用于当已有元素处于某种状态时候，为其添加对应的样式，这个状态是根据用户行为变化而变化的。比如说hover。虽然他和普通的css类似，可以为已有的元素添加样式，但是他只有处于dom树无法描述的状态才能为元素添加样式，**所以称为伪类**
>
> **伪元素**用于创建一些原本不在文档树中的元素，并为其添加样式，比如说`:before`。虽然用户可以看到这些内容，但是其实他不在文档树中。

前端如何进行seo优化,以下说法错误的是：（C）

A.爬虫依赖于标签来确定上下文和各个关键字的权重；

B.重要内容HTML代码放在最前：搜索引擎抓取HTML顺序是从上到下，保证重要内容一定会被抓取

C.重要内容要用js输出：爬虫会执行js获取内容

D.非装饰性图片必须加alt

解析：重要内容不能用js输出：爬虫不会执行js获取内容

关于HTML语义化，以下哪个说法是正确的？（D）

A、语义化的HTML有利于机器的阅读，如PDA手持设备、搜索引擎爬虫；但不利于人的阅读

B、Table 属于过时的标签，遇到数据列表时，需尽量使用 div 来模拟表格

C、语义化是HTML5带来的新概念，此前版本的HTML无法做到语义化

D、header、article、address都属于语义化明确的标签

解析：A错误在于语义化就是为了利于人的阅读而产生的。B错误Table标签语义化明确本就是用来做数据列表的，用div来模拟则不满足标签语义化使用。C语义化是为了利于人的阅读不管html还是html5或者是xml都可尽量做到语义化。

以下哪个是块级元素【多选】(AD)

A. div

B. input

C. img

D. p

解析：块级元素特性占满整行，可见 div 和 p 标签是占满整行的。

img 上 title 与 alt区别?

解析：

title 指图片的信息（鼠标移到图片上显示）、alt 指图片不显示时显示的文字

title 与 h1 的区别、b 与 strong 的区别、i 与 em 的区别?

解析：

（1）title用于网站信息标题，突出网站标题或关键字，一个网站可以有多个title，seo

权重高于H1；H1概括的是文章主题，一个页面最好只用一个H1，seo权重低于

title。

A. 从网站角度而言，title则重于网站信息标题，突出网站标题或关键字用title，一

篇文章，一个页面最好只用一个H1，H1用得太多，会稀释主题；一个网站可以

有多个title，最好一个单页用一个title以便突出网站页面主题信息。

B. 从文章角度而言，H1则概括的是文章主题，突出文章主题，用H1，面对的用

户，要突出其视觉效果。

C. 从SEO角度而言，title的权重高于H1，其适用性要比H1广。

（2）b为了加粗而加粗，strong为了标明重点而加粗

A. b这个标签对应 bold，即文本加粗，其目的仅仅是为了加粗显示文本，是一种样

式 / 风格需求；

B. strong这个标签意思是加强字符的语气，表示该文本比较重要，提醒读者 / 终端

注意。为了达到这个目的，浏览器等终端将其加粗显示；

（3）同（2）i为了斜体而斜体，em为了标明重点而斜体，且对搜索引擎来说strong

和em比b和i要重视的多

px、em、rem、vw单位分别有什么区别?

解析：

（1）px：px就是pixel的缩写，意为像素。px就是一张图片最小的一个点，一张位图

就是千千万万的这样的点构成的，比如常常听到的电脑像素是1024x768的，表示

的是水平方向是1024个像素点，垂直方向是768个像素点。

（2）em：参考物是父元素的font-size，具有继承的特点。如果自身定义了font-size按自身来计算（浏览器默认字体是16px），整个页面内1em不是一个固定的值。

（3）rem：css3新单位，相对于根元素html（网页）的font-size，不会像em那样，依

赖于父元素的字体大小，而造成混乱。

（4）vw：css3新单位，viewpoint width的缩写，视窗宽度，1vw等于视窗宽度的1%。

举个例子：浏览器宽度1200px, 1 vw = 1200px/100 = 12 px。

如何优化网站的SEO

1. 网站结构布局优化：尽量简单, 提倡扁平化结构. 一般而言，建立的网站结构层次越少，越容易被“蜘蛛”抓取，也就容易被收录。
2. img标签必须添加“alt”和“title”属性，告诉搜索引擎导航的定位，做到即使图片未能正常显示时，用户也能看到提示文字。

3. 把重要内容HTML代码放在最前搜索引擎抓取HTML内容是从上到下，利用这一特点，可以让主要代码优先读取，广告等不重要代码放在下边。
4. 控制页面的大小，减少http请求，提高网站的加载速度。
5. 合理的设计title、description和keywords

- title标题：只强调重点即可，尽量把重要的关键词放在前面，关键词不要重复出现，尽量做到每个页面的title标题中不要设置相同的内容。
- meta keywords页面/网站的关键字。
- meta description网页描述，需要高度概括网页内容，切记不能太长，过分堆砌关键词，每个页面也要有所不同。

1. 语义化书写HTML代码，符合W3C标准尽量让代码语义化，在适当的位置使用适当的标签，用正确的标签做正确的事。让阅读源码者和“蜘蛛”都一目了然。
2. a标签：页面链接，要加 “title” 属性说明，链接到其他网站则需要加上 el="nofollow" 属性, 告诉 “蜘蛛” 不要爬，因为一旦“蜘蛛”爬了外部链接之后，就不会再回来了。
3. 图标使用IconFont替换
4. 使用CDN网络缓存，加快用户访问速度，减轻服务器压力
5. 启用GZIP压缩，浏览速度变快，搜索引擎的蜘蛛抓取信息量也会增大
6. SSR技术
7. 预渲染技术

前端性能优化

什么叫优雅降级和渐进增强?

> 渐进增强(progressive enhancement)：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(graceful degradation) ： 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。
>
> 区别：
>
> (1) 优雅降级是从复杂的现状开始，并试图减少用户体验的供给
>
> (2) 渐进增强则是从一个非常基础的，能够起作用的版本开始，并不断扩充，以适应未来环境的需要
>
> (3) 降级（功能衰减）意味着往回看；而渐进增强则意味着朝前看，同时保证其根基处于安全地带

如果需要手动写动画，你认为最小时间间隔是多久，为什么?

> 多数显示器默认频率是60Hz，即1秒刷新60次，所以理论上最小间隔为1/60*1000ms = 16.7ms

png、jpg、gif 这些图片格式解释一下，分别什么时候用。有没有了解过 webp?

> （1）BMP，是无损的、既支持索引色也支持直接色的、点阵图。这种图片格式几乎没有对数据进行压缩，所以BMP格式的图片通常具有较大的文件大小。
>
> （2）GIF是无损的、采用索引色的、点阵图。采用LZW压缩算法进行编码。文件小，是GIF格式的优点，同时，GIF格式还具有支持动画以及透明的优点。但，GIF格式仅支持8bit的索引色，所以GIF格式适用于对色彩要求不高同时需要文件体积较小的场景。

> （3）JPEG是有损的、采用直接色的、点阵图。JPEG的图片的优点，是采用了直接色，得益于更丰富的色彩，JPEG非常适合用来存储照片，与GIF相比，JPEG不适合用来存储企业Logo、线框类的图。因为有损压缩会导致图片模糊，而直接色的选用，又会导致图片文件较GIF更大。
>
> （4）PNG-8是无损的、使用索引色的、点阵图。PNG是一种比较新的图片格式，PNG-8是非常好的GIF格式替代者，在可能的情况下，应该尽可能的使用PNG-8而不是GIF，因为在相同的图片效果下，PNG-8具有更小的文件体积。除此之外，PNG-8还支持透明度的调节，而GIF并不支持。现在，除非需要动画的支持，否则我们没有理由使用GIF而不是PNG-8。
>
> （5）PNG-24是无损的、使用直接色的、点阵图。PNG-24的优点在于，它压缩了图片的数据，使得同样效果的图片，PNG-24格式的文件大小要比BMP小得多。当然，PNG24的图片还是要比JPEG、GIF、PNG-8大得多。
>
> （6）SVG是无损的、矢量图。SVG是矢量图。这意味着SVG图片由直线和曲线以及绘制它们的方法组成。当你放大一个SVG图片的时候，你看到的还是线和曲线，而不会出现像素点。这意味着SVG图片在放大时，不会失真，所以它非常适合用来绘制企业Logo、Icon等。
>
> （7）WebP是谷歌开发的一种新图片格式，WebP是同时支持有损和无损压缩的、使用直接色的、点阵图。使用webp格式的最大的优点是，在相同质量的文件下，它拥有更小的文件体积。因此它非常适合于网络图片的传输，因为图片体积的减少，意味着请求时间的减小，这样会提高用户的体验。这是谷歌开发的一种新的图片格式，目前在兼容性上还不是太好。

Virtual DOM的优势在哪里?

> 「Virtual Dom 的优势」其实这道题目面试官更想听到的答案不是上来就说「直接操作/频繁操作 DOM 的性能差」，如果 DOM 操作的性能如此不堪，那么 jQuery 也不至于活到今天。所以面试官更想听到 VDOM 想解决的问题以及为什么频繁的 DOM 操作会性能差。
>
> 首先我们需要知道：DOM 引擎、JS 引擎 相互独立，但又工作在同一线程（主线程） JS 代码调用 DOM API 必须 挂起 JS 引擎、转换传入参数数据、激活 DOM 引擎，DOM 重绘后再转换可能有的返回值，最后激活 JS 引擎并继续执行若有频繁的 DOM API 调用，且浏览器厂商不做"批量处理"优化， 引擎间切换的单位代价将迅速积累若其中有强制重绘的 DOM API 调用，重新计算布局、重新绘制图像会引起更大的性能消耗。其次是 VDOM 和真实 DOM 的区别和优化：
>
> - 虚拟 DOM 不会立马进行排版与重绘操作
> - 虚拟 DOM 进行频繁修改，然后一次性比较并修改真实 DOM 中需要改的部分，最后在真实 DOM 中进行排版与重绘，减少过多DOM节点排版与重绘损耗
> - 虚拟 DOM 有效降低大面积真实 DOM 的重绘与排版，因为最终与真实 DOM 比较差异，可以只渲染局部

为什么虚拟DOM会提高性能?

> 答案：虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 dom diff 算法避免了没有必要的 dom 操作，从而提高性能。用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异把 2 所记录的差异应用到步骤 1 所构建的真正的 DOM 树上，视图就更新了。

简单说一下图片的懒加载和预加载

> **懒加载**也叫延迟加载，指的是在长网页中延迟加载图片的时机，当用户需要访问时，再去加载这样可以提高网站的首屏加载速度，提升用户的体验，并且可以减少服务器的压力。
>
> 它适用于图片很多，页面很长的电商网站的场景。懒加载的实现原理是，将页面上的图片的 src 属性设置为空字符串，将图片的真实路径保存在一个自定义属性中，当页面滚动的时候，进行判断，如果图片进入页面可视区域内，则从自定义属性中取出真实路径赋值给图片的 src 属性，以此来实现图片的延迟加载。
>
> **预加载**指的是将所需的资源提前请求加载到本地，这样后面在需要用到时就直接从缓存取资源。通过预加载能够减少用户的等待时间，提高用户的体验。我了解的预加载的最常用的方式是使用 js 中的 image 对象，通过为 image 对象来设置 scr 属性，来实现图片的预加载。
>
> 这两种方式都是提高网页性能的方式，两者主要区别是一个是提前加载，一个是迟缓甚至不加载。懒加载对服务器前端有一定的缓 解压力作用，预加载则会增加服务器前端压力。

怎么做性能优化

> 1、JavaScript外联文件引用放在html文档底部；CSS外联文件引用在html文档头部，位于head内；
>
> 2、http静态资源尽量用多个子域名；
>
> 3、服务器端提供html文档和http静态资源时，尽量开启gzip压缩；
>
> 4、尽量减少HTTP Requests的数量；
>
> 5、使用雪碧图来减少CSS背景图片的HTTP请求次数；
>
> 6、首屏不需要展示的较大尺寸图片，请使用lazyload；
>
> 7、图片无损压缩的优化；
>
> 8、减少cookies的大小：尽量减少cookies的体积对减少用户获得响应的时间十分重要；
>
> 9、引入textarea/script元素做延迟解析异步渲染

什么是防抖和节流？有什么区别？

解析：

(1) 防抖(debounce)：触发高频事件后 n 秒内函数只会执行一次，如果 n 秒内高频事件

再次被触发，则重新计算时间。

举例：就好像在百度搜索时，每次输入之后都有联想词弹出，这个控制联想词的方法

就不可能是输入框内容一改变就触发的，他一定是当你结束输入一段时间之后才会触

发。

(2) 节流(thorttle)：高频事件触发，但在 n 秒内只会执行一次，所以节流会稀释函数的

执行频率。

举例：预定一个函数只有在大于等于执行周期时才执行，周期内调用不执行。就好像

你在淘宝抢购某一件限量热卖商品时，你不断点刷新点购买，可是总有一段时间你点

上是没有效果，这里就用到了节流，就是怕点的太快导致系统出现bug。

(3) 区别：防抖动是将多次执行变为最后一次执行，节流是将多次执行变成每隔一段时

间执行。

请说出三种减少网页加载时间的方法。

1. 尽量减少页面中重复http请求数量
2. 服务器开启gzip压缩
3. css样式的定义放置在文件的头部
4. JavaScrip/t脚本放置在文件末尾
5. 压缩合并JavaScript.css代码
6. 使用多域名负载网页内的多个文件.图片

前端性能优化方案

三个方面来说明前端性能优化 一： webapck优化与开启gzip压缩 1.babel-loader用 include 或 exclude 来帮我们避免不必要的转译，不转译node_moudules中的js文件 其次在缓存当前转译的js文件，设置loader: 'babel-loader?cacheDirectory=true' 2.文件采用按需加载等等 3.具体的做法非常简单，只需要你在你的 request headers 中加上这么一句： accept-encoding:gzip 4.图片优化，采用svg图片或者字体图标 5.浏览器缓存机制，它又分为强缓存和协商缓存 二：本地存储——从 Cookie 到 Web Storage、IndexedDB 说明一下SessionStorage和localStorage还有cookie的区别和优缺点 三：代码优化 1.事件代理 2.事件的节流和防抖 3.页面的回流和重绘 4.EventLoop事件循环机制 5.代码优化等等

重排与重绘的区别，什么情况下会触发?

解析：

简述重排的概念:

(1) 浏览器下载完页面中的所有组件（HTML、JavaScript、CSS、图片）之后会解析生成两个内部数据结构（DOM 树和渲染树），DOM 树表示页面结构，渲染树表示 DOM 节点如何显示。重排是 DOM 元素的几何属性变化，DOM 树的结构变化，渲染树需要重新计算。

简述重绘的概念:

(1) 重绘是一个元素外观的改变所触发的浏览器行为，例如改变 visibility、outline、背景色等属性。浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。由于浏览器的流布局，对渲染树的计算通常只需要遍历一次就可以完成。但 table 及其内部元素除外，它可能需要多次计算才能确定好其在渲染树中节点的属性值，比同等元素要多花两倍时间，这就是我们尽量避免使用 table 布局页面的原因之一。

简述重绘和重排的关系:

重绘不会引起重排，但重排一定会引起重绘，一个元素的重排通常会带来一系列的反应，甚至触发整个文档的重排和重绘，性能代价是高昂的。

什么情况下会触发重排?

```
* 页面渲染初始化时；（这个无法避免）
* 浏览器窗口改变尺寸；
* 元素尺寸改变时；
* 元素位置改变时；
* 元素内容改变时；
* 添加或删除可见的 DOM 元素时。
```

重排优化有如下五种方法:

```
* 将多次改变样式属性的操作合并成一次操作，减少 DOM 访问。
* 如果要批量添加 DOM，可以先让元素脱离文档流，操作完后再带入文档流，这样只会触发一次重排。（fragment 元素的应用）
* 将需要多次重排的元素，position 属性设为 absolute 或 fixed，这样此元素就脱离了文档流，它的变化不会影响到其他元素。例如有动画效果的元素就最好设置为绝对定位。
* 由于 display 属性为 none 的元素不在渲染树中，对隐藏的元素操作不会引发其他元素的重排。如果要对一个元素进行复杂的操作时，可以先隐藏它，操作完成后再显示。这样只在隐藏和显示时触发两次重排。
* 在内存中多次操作节点，完成后再添加到文档中去。例如要异步获取表格数据，渲染到页面。可以先取得数据后在内存中构建整个表格的 html 片段，再一次性添加到文档中去，而不是循环添加每一行。
```

什么是回流，什么是重绘，有什么区别?

> 当render tree中的一部分(或全部)因为元素的规模尺寸，布局，隐藏等改变而需要重新构建。这就称为**回流(reflow)**
>
> 当render tree中的一些元素需要更新属性，而这些属性只是影响元素的外观，风格，而不会影响布局的，比如background-color。则就叫称为**重绘**
>
> **区别:**
>
> 回流必将引起重绘，而重绘不一定会引起回流。比如：只有颜色改变的时候就只会发生重绘而不会引起回流 当页面布局和几何属性改变时就需要回流 比如：添加或者删除可见的DOM元素，元素位置改变，元素尺寸改变——边距、填充、边框、宽度和高度，内容改变

工程化

简单说一下babel的原理

> babel的转译过程分为三个阶段：**parsing、transforming、generating**，以ES6代码转译为ES5代码为例，babel转译的具体过程如下：
>
> 1. ES6代码输入
> 2. babylon 进行解析得到 AST
> 3. plugin 用 babel-traverse 对 AST 树进行遍历转译,得到新的AST树
> 4. 用 babel-generator 通过 AST 树生成 ES5 代码

webpack3和webpack4区别

1. mode

   webpack增加了一个mode配置，只有两种值development | production。对不同的环境他会启用不同的配置。

2. CommonsChunkPlugin

   CommonChunksPlugin已经从webpack4中移除。 可使用optimization.splitChunks进行模块划分（提取公用代码）。 但是需要注意一个问题，默认配置只会对异步请求的模块进行提取拆分，如果要对entry进行拆分 需要设置optimization.splitChunks.chunks = 'all'。

3. webpack4使用MiniCssExtractPlugin取代ExtractTextWebpackPlugin。

4. 代码分割。

   使用动态import，而不是用system.import或者require.ensure

5. vue-loader。

   使用vue-loader插件为.vue文件中的各部分使用相对应的loader，比如css-loader等

6. UglifyJsPlugin

   现在也不需要使用这个plugin了，只需要使用optimization.minimize为true就行，production mode下面自动为true

   optimization.minimizer可以配置你自己的压缩程序

对webpack的看法

- webpack是一个模块打包工具，可以使用webpack管理你的模块依赖，并编译输出模块们所需要的静态文件。能很好的管理、打包web开发中所用到的HTML、js、css以及各种静态文件（图片、字体等），让开发过程更加高效。对于不同类型的资源，webpack有对应的模块加载器。webpack模块打包器会分析模块间的依赖关系，最后生成了优化且合并后的静态资源。
- webpack两大特色：
  - code splitting（可以自动完成）
  - loader 可以处理各种类型的静态文件，并且支持串联操作 webpack是以commonJS的形式来书写脚本，但是AMD/CMD的支持也很全面，方便旧项目进行代码迁移
- webpck具有require和browserify的功能，但仍有很多自己的新特性：
  - 对 CommonJS、AMD、ES6的语法做了兼容
  - 对JS、css、图片等资源文件都支持打包
  - 串联式模块化加载器以及插件机制，让其具有更好的灵活性和扩展性，例如提供对conffeescript、ES6的支持
  - 有独立的配置文件webpck.config.js
  - 可以将代码切割成不同的chunk，实现按需加载，降低了初始化时间
  - 支持sourceUrls和sourceMaps，易于调试
  - 具有强大的plugin接口，大多是内部插件，使用起来比较灵活
  - webpack使用异步IO并具有多级缓存，在增亮编译上更加快

简单说一下webpack 的原理

> 初始化参数：从配置文件和 Shell 语句中读取与合并参数，得出最终的参数；
>
> 开始编译：用上一步得到的参数初始化 Compiler 对象，加载所有配置的插件，执行对象的 run 方法开始执行编译；
>
> 确定入口：根据配置中的 entry 找出所有的入口文件；
>
> 编译模块：从入口文件出发，调用所有配置的 Loader 对模块进行翻译，再找出该模块依赖的模块，再递归本步骤直到所有入口依赖的文件都经过了本步骤的处理；
>
> 完成模块编译：在经过第4步使用 Loader 翻译完所有模块后，得到了每个模块被翻译后的最终内容以及它们之间的依赖关系；
>
> 输出资源：根据入口和模块之间的依赖关系，组装成一个个包含多个模块的 Chunk，再把每个 Chunk 转换成一个单独的文件加入到输出列表，这步是可以修改输出内容的最后机会；
>
> 输出完成：在确定好输出内容后，根据配置确定输出的路径和文件名，把文件内容写入到文件系统。

移动端

移动端性能优化

- 尽量使用css3动画，开启硬件加速
- 适当使用touch时间代替click时间
- 避免使用css3渐变阴影效果
- 可以用transform: translateZ(0) 来开启硬件加速
- 不滥用float。float在渲染时计算量比较大，尽量减少使用
- 不滥用web字体。web字体需要下载，解析，重绘当前页面
- 合理使用requestAnimationFrame动画代替setTimeout
- css中的属性（css3 transitions、css3 3D transforms、opacity、webGL、video）会触发GUP渲染，耗电

移动端300ms延迟

解析：

300毫米延迟解决的是双击缩放。双击缩放，手指在屏幕快速点击两次。safari浏览器就会将网页缩放值原始比例。由于用户可以双击缩放或者是滚动的操作， 当用户点击屏幕一次之后，浏览器并不会判断用户确实要打开至这个链接，还是想要进行双击操作 因此，safair浏览器就会等待300ms，用来判断用户是否在次点击了屏幕 解决方案：1.禁用缩放，设置meta标签 user-scalable=no 2.fastclick.js 原理：FastClick的实现原理是在检查到touchend事件的时候，会通过dom自定义事件立即 发出click事件，并把浏览器在300ms之后真正的click事件阻止掉 fastclick.js还可以解决穿透问题

移动端白屏解决方案

解析

白屏的基本原因可以归结为网速和静态资源

- css文件加载需要时间，在加载过程中页面空白

【解决方案】可以考虑css代码前置或内联

- 首屏没有实际的数据内容，等待异步加载数据，再渲染页面导致白屏

【解决方案】首屏直接同步渲染html，后续的滚屏等操作再异步请求数据渲染html

- 首屏内联js执行，会阻塞页面渲染

【解决方案】尽量不要在首屏html代码中内联js脚本

【其他解决方案】

- 在服务器端使用模版引擎渲染所有页面
- 减少文件加载体积，压缩`html`、`js`
- 本地存储静态文件

首屏和白屏时间如何计算?

答案:

> 首屏时间的计算，可以由 Native WebView 提供的类似 onload 的方法实现，在 ios 下对应的是 webViewDidFinishLoad，在 android 下对应的是onPageFinished事件。白屏的定义有多种。可以认为“没有任何内容”是白屏，可以认为“网络或服务异常”是白屏，可以认为“数据加载中”是白屏，可以认为“图片加载不出来”是白屏。场景不同，白屏的计算方式就不相同。
>
> 方法1：当页面的元素数小于x时，则认为页面白屏。比如“没有任何内容”，可以获取页面的DOM节点数，判断DOM节点数少于某个阈值X，则认为白屏。 方法2：当页面出现业务定义的错误码时，则认为是白屏。比如“网络或服务异常”。 方法3：当页面出现业务定义的特征值时，则认为是白屏。比如“数据加载中”。

小程序和H5什么区别?

> 渲染方式与 H5 不同，小程序一般是通过 Native 原生渲染的，但是小程序同时也支持 web 渲染，如果使用 web 渲染的方式，我们需要初始化一个WebView 组件，然后在 WebView 中加载 H5 页面；
>
> 所以当我们开发一个小程序时，通常会使用 hybrid 的方式，即会根据具体情况选择部分功能用小程序原生的代码来开发，部分功能通过 WebView 加载 H5 页面来实现。Native 与 Web 渲染混合使用，以实现项目的最优解；
>
> 这里值得注意的是，小程序下，native 方式通常情况下性能要优于 web 方式。小程序特有的双线程设计。 H5 下我们所有资源通常都会打到一个 bundle.js 文件里（不考虑分包加载），而小程序编译后的结果会有两个bundle，index.js封装的是小程序项目的 view 层，以及 index.worker.js 封装的是项目的业务逻辑，在运行时，会有两条线程来分别处理这两个bundle，一个是主渲染线程，它负责加载并渲染 index.js 里的内容，另外一个是 Service Worker线 程，它负责执行 index.worker.js 里封装的业务逻辑，这里面会有很多对底层api调用。

网络协议和服务

一个tcp链接能发几个http请求?

> **答案:**
>
> 如果是 HTTP 1.0 版本协议，一般情况下，不支持长连接，因此在每次请求发送完毕之后，TCP 连接即会断开，因此一个 TCP 发送一个 HTTP 请求，但是有一种情况可以将一条 TCP 连接保持在活跃状态，那就是通过 Connection 和 Keep-Alive 首部，在请求头带上 Connection: Keep-Alive，并且可以通过 Keep-Alive 通用首部中指定的，用逗号分隔的选项调节 keep-alive 的行为，如果客户端和服务端都支持，那么其实也可以发送多条，不过此方式也有限制，可以关注《HTTP 权威指南》4.5.5 节对于 Keep-Alive 连接的限制和规则。
>
> 而如果是 HTTP 1.1 版本协议，支持了长连接，因此只要 TCP 连接不断开，便可以一直发送 HTTP 请求，持续不断，没有上限； 同样，如果是 HTTP 2.0 版本协议，支持多用复用，一个 TCP 连接是可以并发多个 HTTP 请求的，同样也是支持长连接，因此只要不断开 TCP 的连接，HTTP 请求数也是可以没有上限地持续发送

简述URL和URI的区别?

> 答案:
>
> URI: Uniform Resource Identifier 指的是统一资源标识符
>
> URL: Uniform Resource Location 指的是统一资源定位符
>
> URI 指的是统一资源标识符，用唯一的标识来确定一个资源，它是一种抽象的定义，也就是说，不管使用什么方法来定义，只要能唯 一的标识一个资源，就可以称为 URI。
>
> URL 指的是统一资源定位符，URN 指的是统一资源名称。URL 和 URN 是 URI 的子集，URL 可以理解为使用地址来标识资源。

浏览器输入url到页面呈现出来发生了什么?

> 1)、进行地址解析 解析出字符串地址中的主机，域名，端口号，参数等 2）、根据解析出的域名进行DNS解析 1.首先在浏览器中查找DNS缓存中是否有对应的ip地址，如果有就直 接使用，没有机执行第二步 2.在操作系统中查找DNS缓存是否有对应的ip地址，如果有就直接使 用，没有就执行第三步 3.向本地DNS服务商发送请求查找时候有DNS对应的ip地址。如果仍 然没有最后向Root Server服务商查询。
>
> 3)、根据查询到的ip地址寻找目标服务器
>
> 1.与服务器建立连接
>
> 2.进入服务器，寻找对应的请求
>
> 4)、浏览器接收到响应码开始处理。
>
> 5)、浏览器开始渲染dom，下载css、图片等一些资源。直到这次请求完成

https、http区别

- https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

- http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
- 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。
- http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

跨域

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，Ajax 请求会失败。防止CSRF攻击。 1.JSONP JSONP 的原理很简单，就是利用

```
<script>
    function jsonp(data) {
        console.log(data)
    }
</script>
```

JSONP 使用简单且兼容性不错，但是只限于 get 请求。

2. CORS CORS 需要浏览器和后端同时支持。IE 8 和 9 需要通过 XDomainRequest 来实现。
3. 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
4. webpack配置proxyTable设置开发环境跨域
5. nginx代理跨域
6. iframe跨域
7. postMessage 这种方式通常用于获取嵌入页面中的第三方页面数据。一个页面发送消息，另一个页面判断来源并接收消息

TCP的三次握手和四次挥手

解析：

> 三次握手

- 第一次握手：客户端发送一个SYN码给服务器，要求建立数据连接；
- 第二次握手： 服务器SYN和自己处理一个SYN（标志）；叫SYN+ACK（确认包）；发送给客户端，可以建立连接
- 第三次握手： 客户端再次发送ACK向服务器，服务器验证ACK没有问题，则建立起连接；

> 四次挥手

- 第一次挥手： 客户端发送FIN(结束)报文，通知服务器数据已经传输完毕；
- 第二次挥手: 服务器接收到之后，通知客户端我收到了SYN,发送ACK(确认)给客户端，数据还没有传输完成

- 第三次挥手： 服务器已经传输完毕，再次发送FIN通知客户端，数据已经传输完毕
- 第四次挥手： 客户端再次发送ACK,进入TIME_WAIT状态；服务器和客户端关闭连接；

CDN（内容分发网络）

解析：

1.尽可能的避开互联网有可能影响数据传输速度和稳定性的瓶颈和环节。使内容传输的更快更稳定。 2.关键技术：内容存储和分发技术中 3.基本原理：广泛采用各种缓存服务器，将这些缓存服务器分布到用户访问相对的地区或者网络中。当用户访问网络时利用全局负载技术 将用户的访问指向距离最近的缓存服务器，由缓存服务器直接相应用户的请求（全局负载技术）

WEB安全

web上传漏洞原理？如何进行？防御手段？

> 如何进行：用户上传了一个可执行的脚本文件，并通过此脚本文件获得了执行服务器端命令的能力。
>
> 主要原理：当文件上传时没有对文件的格式和上传用户做验证，导致任意用户可以上传任意文件，那么这就是一个上传漏洞。
>
> 防御手段：
>
> 1、最有效的，将文件上传目录直接设置为不可执行，对于Linux而言，撤销其目录的'x'权限；实际中很多大型网站的上传应用都会放置在独立的存储上作为静态文件处理，一是方便使用缓存加速降低能耗，二是杜绝了脚本执行的可能性；
>
> 2、文件类型检查：强烈推荐白名单方式，结合MIME Type、后缀检查等方式；此外对于图片的处理可以使用压缩函数或resize函数，处理图片的同时破坏其包含的HTML代码；
>
> 3、使用随机数改写文件名和文件路径，使得用户不能轻易访问自己上传的文件；
>
> 4、单独设置文件服务器的域名

Cookie如何防范XSS攻击？

> XSS（跨站脚本攻击）是指攻击者在返回的HTML中嵌入javascript脚本，为了减轻这些攻击，需要在HTTP头部配上，set-cookie：
>
> httponly-这个属性可以防止XSS,它会禁止javascript脚本来访问cookie。
>
> secure - 这个属性告诉浏览器仅在请求为https的时候发送cookie

网页验证码是干嘛的，是为了解决什么安全问题？

> （1）区分用户是计算机还是人的公共全自动程序。可以防止恶意破解密码、刷票、论坛灌水
>
> （2）有效防止黑客对某一个特定注册用户用特定程序暴力破解方式进行不断的登陆尝试