

Front-End Interview

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) 两等号判断，会在比较时进行类型转换；
(2)三等号判断（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 在三等号判断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

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 < 内部样式 < 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同构的

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 当需要判断变量是否是 number, string, boolean, function, undefined 等类型时，可以使用 typeof 进行判断。
- (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。

side projects

```
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 (var i = 0; i < arr.length; i++) {
        for (var j = i + 1; j < arr.length; j++) {
            if (arr[i] == arr[j]) {
                arr.splice(j, 1); j--;
            }
        }
    }
    return arr;
};
```

(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 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'?运算符，实际上是 'Value is true?'Something' : 'Nothing'，当字符串不为空时，转换为boolean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 sideEffecting(ary) {
    ary[0] = ary[2];
}
function bar(a,b,c) {
    c = 10
    sideEffecting(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boolean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", 空 × 2]`，选D

以下代码运行结果为 (B)

```
var a = 111111111111111110000,
    b = 1111;
console.log(a + b);
```

A: 11111111111111111111111111111111 B: 111111111111111110000 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111111111111111110000已经超过了2⁵³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, B: 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

```
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resolve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)  
    },  
)
```

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name属性查询而不是通过name属性查询。

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
算法可能是：(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
- 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, 4

B. 1, 2, 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 D;

网络协议和服务

一个 TCP 连接能发几个 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。

简述一下跨域

因为浏览器出于安全考虑，有同源策略。

同源策略是浏览器最核心也最基本的安全功能，如果缺少了同源策略，浏览器很容易受到XSS、CSFR等攻击。

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。

也就是说，如果协议、域名或者端口有一个不同就是跨域

同源策略限制以下几种行为：

- Cookie、LocalStorage 和 IndexDB 无法读取
- DOM和JS对象无法获得
- AJAX 请求不能发送

跨域解决方案

1.JSONP跨域

JSONP的原理就是利用 `<script>` 标签没有跨域限制，通过 `<script>` 标签src属性，发送带有callback参数的GET请求，服务端将接口返回数据拼凑到callback函数中，返回给浏览器，浏览器解析执行，从而前端拿到callback函数返回的数据。

```
<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(跨域资源共享)

CORS是一个W3C标准，全 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.CDN的全称是Content Delivery Network，即(内容分发网络)。CDN是构建在现有网络基础之上的智能虚拟网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。CDN的关键技术主要有内容存储和分发技术。

2. 归纳起来，CDN具有以下主要功能：

- (1)节省骨干网带宽，减少带宽需求量；
- (2)提供服务器端加速，解决由于用户访问量大造成的服务器过载问题；
- (3)服务商能使用Web Cache技术在本地缓存用户访问过的Web页面和对象，实现相同对象的访问无须占用主干的出口带宽，并提高用户访问因特网页面的响应时间的需求；
- (4)能克服网站分布不均的问题，并且能降低网站自身建设和维护成本；
- (5)降低“通信风暴”的影响，提高网络访问的稳定性。

3.CDN的基本原理是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集中的地区或网络中，在用户访问网站时，利用全局负载技术将用户的访问指向距离最近的工作正常的缓存服务器上，由缓存服务器直接响应用户请求。

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. 网页的图片验证码是用于人机识别的，用于区分人的操作行为和机器行为，防止恶意机器盗刷，恶意占票，
2. 网页的登录验证码，属于双因素认证应用到账号安全的范畴，作用是确保是用户本人进行登录，大大降低了由于账号被盗，导致的非授权登录行为
3. 网页的注册验证码，是验证注册者的身份，防止恶意注册，确保用户的有效性

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

JSONP 是利用 标签没有同源策略的限制（算是一个漏洞），来达到与第三方通信，从而实现跨域。

我们通常使用js代码动态创建一个script标签，src引用第三方api的地址，并提供一个回调函数的function name，例如<http://www.smartsstudy.com/api?callback=functionName>，query的key是约定好的，比如这里就叫callback，functionName是我们前端定义好的函数，后端通过query取到函数名，并把你所需要的数据以参数的形式传给这个函数，返回给浏览器。浏览器解析服务器返回数据，functionName函数，参数为资源数据。

JSONP并不使用XMLHttpRequest对象加载资源，而是通过script标签把资源当做普通的javascript脚本来加载，所以不存在跨域问题，也不是真正的AJAX。

get 和 post 请求在缓存方面的区别

缓存一般只适用于那些不会更新服务端数据的请求。

一般 get 请求都是查找请求，不会对服务器资源数据造成修改

而 post 请求一般都会对服务器数据造成修改，所以，一般会对 get 请求进行缓存，很少会对 post 请求进行缓存。

websocket和ajax轮询

- websocket是html5中提出的新的协议，可以实现客户端与服务器的通信，实现服务器的推送功能
- 优点是，只要简历一次连接，就可以连续不断的得到服务器推送消息，节省带宽和服务器端的压力。
- ajax轮询模拟常连接就是每隔一段时间（0.5s）就向服务器发起ajax请求，查询服务器是否有数据更新
- 缺点就是，每次都要建立HTTP连接，即使需要传输的数据非常少，浪费带宽

CSS

IOS手机浏览器字体齿轮

修改`-webkit-font-smoothing`属性，结果是：

```
-webkit-font-smoothing: none; 无抗锯齿  
-webkit-font-smoothing: antialiased | subpixel-antialiased | default; 灰度平滑
```

calc, support, media各自的含义及用法？

@support主要是用于检测浏览器是否支持CSS的某个属性，其实就是条件判断，如果支持某个属性，你可以写一套样式，如果不支持某个属性，你也可以提供另外一套样式作为替补。

calc() 函数用于动态计算长度值。 calc()函数支持 "+"、"-"、"*"、"/" 运算；

@media 查询，你可以针对不同的媒体类型定义不同的样式。

1rem、1em、1vh、1px各自代表的含义？

rem：是全部的长度都相对于根元素元素。通常做法是给html元素设置一个字体大小，然后其他元素的长度单位就为rem。

em:

- 子元素字体大小的em是相对于父元素字体大小
- 元素的width/height/padding/margin用em的话是相对于该元素的font-size

vw/vh :

全称是 Viewport Width 和 Viewport Height，视窗的宽度和高度，相当于 屏幕宽度和高度的 1%，不过，处理宽度的时候%单位更合适，处理高度的话 vh 单位更好。

px:

px像素（Pixel）。相对长度单位。像素px是相对于显示器屏幕分辨率而言的。

一般电脑的分辨率有{1920*1024}等不同的分辨率

怎么画一条0.5px的直线？

```
height: 1px;
transform: scale(0.5);
```

解释css sprites，如何使用？

CSS Sprites其实就是把网页中一些背景图片整合到一张图片文件中，再利用CSS的“background-image”，“background-repeat”，“background-position”的组合进行背景定位，background-position可以用数字能精确的定位出背景图片的位置。

CSS Sprites为一些大型的网站节约了带宽，让提高了用户的加载速度和用户体验，不需要加载更多的图片。

CSS中link和@import的区别是什么？

(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方式的样式权重高于@import的权重

在 css 选择器当中，优先级排序正确的是 (D)

- A、id选择器>标签选择器>类选择器
- B、标签选择器>类选择器>id选择器
- C、类选择器>标签选择器>id选择器
- D、id选择器>类选择器>标签选择器

解析：style权重若为1000，id选择器权重为100，class选择器权重为10，div权重则为1。

rgba()和 opacity 的透明效果有什么不同？

解析：

rgba()和 opacity 都能实现透明效果，但最大的不同是 opacity 作用于元素，以及元素内的所有内容的透明度；而 rgba()只作用于元素的颜色或其背景色。（设置 rgba 透明的元素的子元素及内容都不会继承透明效果！）

请列举几种隐藏元素的方法

解析：

- (1) visibility: hidden; 这个属性只是简单的隐藏某个元素，但是元素占用的空间仍然存在
- (2) opacity: 0; CSS3属性，设置0可以使一个元素完全透明，但不会改变页面布局，并且，如果该元素已经绑定一些事件，如click事件，那么点击该区域，也能触发点击事件的
- (3) position: absolute; 设置一个很大的 left 负值定位，使元素定位在可见区域之外
- (4) display: none; 元素会变得不可见，并且不会再占用元素位置。会改变页面布局。
- (5) transform: scale(0); 将一个元素设置为缩放无限小，元素将不可见，元素原来所在的位置将被保留。
- (6)

side projects