最新的前端大厂面经（详解答案）

简单

1 从输入一个 URL 地址到浏览器完成渲染的整个过程

这个问题属于老生常谈的经典问题了 下面给出面试**简单版**作答

1. 浏览器地址栏输入 URL 并回车
2. 浏览器查找当前 URL 是否存在缓存，并比较缓存是否过期
3. DNS 解析 URL 对应的 IP
4. 根据 IP 建立 TCP 连接（三次握手）
5. 发送 http 请求
6. 服务器处理请求，浏览器接受 HTTP 响应
7. 浏览器解析并渲染页面
8. 关闭 TCP 连接（四次握手）

> 注意！ 面试官可以基于这道题进行前端很多知识点的考察 从 http 网络知识到浏览器原理再到前端性能优化 这个题目都可以作为引子开始

所以推荐大家可以看看这篇详细作答 史上最详细的经典面试题 从输入 URL 到看到页面发生了什么?

2 什么是事件代理（事件委托） 有什么好处

事件委托的原理：不给每个子节点单独设置事件监听器，而是设置在其父节点上，然后利用冒泡原理设置每个子节点。

优点：

- 减少内存消耗和 dom 操作，提高性能 在 JavaScript 中，添加到页面上的事件处理程序数量将直接关系到页面的整体运行性能，因为需要不断的操作 dom,那么引起浏览器重绘和回流的可能也就越多，页面交互的事件也就变的越长，这也就是为什么要**减少 dom 操作**的原因。每一个事件处理函数，都是一个对象，多一个事件处理函数，**内存**中就会被多占用一部分空间。如果要用事件委托，就会将所有的操作放到 js 程序里面，只对它的父级进行操

作，与 dom 的操作就只需要交互一次，这样就能大大的减少与 dom 的交互次数，提高性能；

- 动态绑定事件 因为事件绑定在父级元素 所以新增的元素也能触发同样的事件

3 addEventListener 默认是捕获还是冒泡

默认是**冒泡**

addEventListener**第三个参数**默认为 false 代表执行事件冒泡行为。

当为 true 时执行事件捕获行为。

4 css 的渲染层合成是什么 浏览器如何创建新的渲染层

在 DOM 树中每个节点都会对应一个渲染对象（RenderObject），当它们的渲染对象处于相同的坐标空间（z 轴空间）时，就会形成一个 RenderLayers，也就是渲染层。渲染层将保证页面元素以正确的顺序堆叠，这时候就会出现**层合成（composite）**，从而正确处理透明元素和重叠元素的显示。对于有位置重叠的元素的页面，这个过程尤其重要，因为一旦图层的合并顺序出错，将会导致元素显示异常。

浏览器如何创建新的渲染层

- 根元素 document
- 有明确的定位属性（relative、fixed、sticky、absolute）
- opacity < 1
- 有 CSS fliter 属性
- 有 CSS mask 属性
- 有 CSS mix-blend-mode 属性且值不为 normal
- 有 CSS transform 属性且值不为 none
- backface-visibility 属性为 hidden
- 有 CSS reflection 属性

- 有 CSS column-count 属性且值不为 auto 或者有 CSS column-width 属性且值不为 auto
- 当前有对于 opacity、transform、fliter、backdrop-filter 应用动画
- overflow 不为 visible

> 注意！不少人会将这些**合成层的条件和渲染层产生的条件**混淆，这两种条件发生在两个不同的层处理环节，是完全不一样的 具体可以看看这篇文章 浏览器层合成与页面渲染优化

5 webpack Plugin 和 Loader 的区别

- Loader：

  用于对模块源码的转换，loader 描述了 webpack 如何处理非 javascript 模块，并且在 buld 中引入这些依赖。loader 可以将文件从不同的语言（如 TypeScript）转换为 JavaScript，或者将内联图像转换为 data URL。比如说：CSS-Loader，Style-Loader 等。

- Plugin

  目的在于解决 loader 无法实现的其他事,它直接作用于 webpack，扩展了它的功能。在 webpack 运行的生命周期中会广播出许多事件，plugin 可以监听这些事件，在合适的时机通过 webpack 提供的 API 改变输出结果。插件的范围包括，从打包优化和压缩，一直到重新定义环境中的变量。插件接口功能极其强大，可以用来处理各种各样的任务。

> webpack 相关知识可以看鲨鱼哥这篇文章 前端进阶高薪必看-Webpack 篇 说的很通俗易懂 基本应对简单的面试足够了

6.apply call bind 区别

- 三者都可以改变函数的 this 对象指向。
- 三者第一个参数都是 this 要指向的对象，如果如果没有这个参数或参数为 undefined 或 null，则默认指向全局 window。
- 三者都可以传参，但是 **apply 是数组**，而 **call 是参数列表**，且 apply 和 call 是一次性传入参数，而 **bind 可以分为多次传入。**
- bind 是返回**绑定 this 之后的函数**，便于稍后调用；apply 、call 则是**立即执行** 。
- bind()会返回一个新的函数，如果这个返回的新的函数作为**构造函数**创建一个新的对象，那么此时 this **不再指向**传入给 bind 的第一个参数，而是指向用 new 创建的实例

> 注意！很多同学可能会忽略 bind 绑定的函数作为构造函数进行 new 实例化的情况

7 举出闭包实际场景运用的例子

比如常见的防抖节流

```js
// 防抖
function debounce(fn, delay = 300) {
  let timer; //闭包引用的外界变量
  return function () {
    const args = arguments;
    if (timer) {
      clearTimeout(timer);
    }
    timer = setTimeout(() => {
      fn.apply(this, args);
    }, delay);
  };
}
```

使用闭包可以在 JavaScript 中模拟块级作用域

```js
function outputNumbers(count) {
  (function () {
    for (var i = 0; i < count; i++) {
      alert(i);
    }
  })();
  alert(i); //导致一个错误！
}
```

闭包可以用于在对象中创建私有变量

```js
var aaa = (function () {
  var a = 1;
  function bbb() {
    a++;
    console.log(a);
  }
```

```
  function ccc() {
    a++;
    console.log(a);
  }
  return {
    b: bbb, //json结构
    c: ccc,
  };
})();
console.log(aaa.a); //undefined
aaa.b(); //2
aaa.c(); //3
```

8 css 优先级是怎么计算的

- 第一优先级：!important 会覆盖页面内任何位置的元素样式
- 1.内联样式，如 style="color: green"，权值为 1000
- 2.ID 选择器，如#app，权值为 0100
- 3.类、伪类、属性选择器，如.foo, :first-child, div[class="foo"]，权值为 0010
- 4.标签、伪元素选择器，如 div::first-line，权值为 0001
- 5.通配符、子类选择器、兄弟选择器，如*, >, +，权值为 0000
- 6.继承的样式没有权值

9 事件循环相关题目--必考（一般是代码输出顺序判断）


```
setTimeout(function () {
  console.log("1");
}, 0);
async function async1() {
  console.log("2");
  const data = await async2();
  console.log("3");
  return data;
}
async function async2() {
  return new Promise((resolve) => {
    console.log("4");
    resolve("async2的结果");
  }).then((data) => {
    console.log("5");
    return data;
  });
```

```
}
async1().then((data) => {
  console.log("6");
  console.log(data);
});
new Promise(function (resolve) {
  console.log("7");
  //   resolve()
}).then(function () {
  console.log("8");
});
```

输出结果：247536 async2 的结果 1

> 注意！我在最后一个 **Promise** 埋了个坑 我没有调用 resolve 方法 这个是在面试美团的时候遇到了 当时自己没看清楚 以为都是一样的套路 最后面试官说不对 找了半天才发现是这个坑 哈哈

10 http **状态码** 204 301 302 304 400 401 403 404 **含义**

- http 状态码 204 （无内容） 服务器成功处理了请求，但没有返回任何内容
- http 状态码 301 （永久移动） 请求的网页已永久移动到新位置。 服务器返回此响应（对 GET 或 HEAD 请求的响应）时，会自动将请求者转到新位置。
- http 状态码 302 （临时移动） 服务器目前从不同位置的网页响应请求，但请求者应继续使用原有位置来进行以后的请求。
- http 状态码 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。
- http 状态码 400 （错误请求） 服务器不理解请求的语法（一般为参数错误）。
- http 状态码 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。
- http 状态码 403 （禁止） 服务器拒绝请求。（一般为客户端的用户权限不够）
- http 状态码 404 （未找到） 服务器找不到请求的网页。

11 http2.0 **做了哪些改进** 3.0 **呢**

http2.0 特性如下

- 二进制分帧传输
- 多路复用
- 头部压缩

- 服务器推送

Http3.0 相对于 Http2.0 是一种脱胎换骨的改变！

http 协议是应用层协议，都是建立在传输层之上的。我们也都知道传输层上面不只有 TCP 协议，还有另外一个强大的协议 **UDP 协议**，2.0 和 1.0 都是基于 TCP 的，因此都会有 TCP 带来的硬伤以及局限性。而 Http3.0 则是建立在 UDP 的基础上。所以其与 Http2.0 之间有质的不同。

http3.0 特性如下

- 连接迁移
- 无队头阻塞
- 自定义的拥塞控制
- 前向安全和前向纠错

> 建议大家详细看看这篇文章Http2.0 的一些思考以及 Http3.0 的优势

12 position 有哪些值，作用分别是什么

- static

  static(没有定位)是 position 的默认值，元素处于正常的文档流中，会忽略 left、top、right、bottom 和 z-index 属性。

- relative

  relative(相对定位)是指给元素设置相对于原本位置的定位，元素并不脱离文档流，因此元素原本的位置会被保留，其他的元素位置不会受到影响。

  使用场景：子元素相对于父元素进行定位

- absolute absolute(绝对定位)是指给元素设置绝对的定位，相对定位的对象可以分为两种情况：

  1. 设置了 absolute 的元素如果存在有祖先元素设置了 position 属性为 relative 或者 absolute，则这时元素的定位对象为此已设置 position 属性的祖先元素。
  2. 如果并没有设置了 position 属性的祖先元素，则此时相对于 body 进行定位。 **使用场景**：跟随图标 图标使用不依赖定位父级的 absolute 和 margin 属性进行定位，这样，当文本的字符个数改变时，图标的位置可以自适应

- fixed 可以简单说 fixed 是特殊版的 absolute，fixed 元素总是相对于 body 定位的。 **使用场景**：侧边栏或者广告图

- inherit 继承父元素的 position 属性，但需要注意的是 IE8 以及往前的版本都不支持 inherit 属性。

- sticky 设置了 sticky 的元素，在屏幕范围（viewport）时该元素的位置并不受到定位影响（设置是 top、left 等属性无效），当该元素的位置将要移出偏移范围时，定位又会变成 fixed，根据设置的 left、top 等属性成固定位置的效果。 当元素在容器中被滚动超过指定的偏移值时，元素在容器内固定在指定位置。亦即如果你设置了 top: 50px，那么在 sticky 元素到达距离相对定位的元素顶部 50px 的位置时固定，不再向上移动（相当于此时 fixed 定位）。

  使用场景：跟随窗口

13 垂直水平居中实现方式

这道题基本也是 css 经典题目 但是网上已经有太多千篇一律的答案了 如果大家想在这道题加分

可以针对定宽高和不定宽高的实现多种不同的方案

建议大家直接看 面试官：你能实现多少种水平垂直居中的布局（定宽高和不定宽高）

14 vue 组件通讯方式有哪些方法

- props 和$emit 父组件向子组件传递数据是通过 prop 传递的，子组件传递数据给父组件是通过$emit 触发事件来做到的

- $parent,$children 获取当前组件的父组件和当前组件的子组件

- $attrs 和$listeners A->B->C。Vue 2.4 开始提供了$attrs 和$listeners 来解决这个问题

- 父组件中通过 provide 来提供变量，然后在子组件中通过 inject 来注入变量。(官方不推荐在实际业务中使用，但是写组件库时很常用)

- $refs 获取组件实例

- envetBus 兄弟组件数据传递 这种情况下可以使用事件总线的方式

- vuex 状态管理

15 Vue 响应式原理

整体思路是数据劫持+观察者模式

对象内部通过 defineReactive 方法，使用 Object.defineProperty 将属性进行劫持（只会劫持已经存在的属性），数组则是通过重写数组方法来实现。当页面使用对应属性时，每个属性都拥有自己的 dep 属性，存放他所依赖的 watcher（依赖收集），当属性变化后会通知自己对应的 watcher 去更新(派发更新)。

相关代码如下

javascript 复制代码

```
class Observer {
  // 观测值
  constructor(value) {
    this.walk(value);
  }
  walk(data) {
    // 对象上的所有属性依次进行观测
    let keys = Object.keys(data);
    for (let i = 0; i < keys.length; i++) {
      let key = keys[i];
      let value = data[key];
      defineReactive(data, key, value);
    }
  }
}
// Object.defineProperty数据劫持核心 兼容性在ie9以及以上
function defineReactive(data, key, value) {
  observe(value); // 递归关键
  // --如果value还是一个对象会继续走一遍odefineReactive 层层遍历一直到value不是对象才
  //    思考？如果Vue数据嵌套层级过深 >>性能会受影响
  Object.defineProperty(data, key, {
    get() {
      console.log("获取值");

      //需要做依赖收集过程 这里代码没写出来
      return value;
    },
    set(newValue) {
      if (newValue === value) return;
      console.log("设置值");
      //需要做派发更新过程 这里代码没写出来
      value = newValue;
    },
```

```
  });
}
export function observe(value) {
  // 如果传过来的是对象或者数组 进行属性劫持
  if (
    Object.prototype.toString.call(value) === "[object Object]" ||
    Array.isArray(value)
  ) {
    return new Observer(value);
  }
}
```

响应式数据原理详解 传送门

16 Vue nextTick 原理

nextTick 中的回调是在下次 DOM 更新循环结束之后执行的延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM。主要思路就是采用微任务优先的方式调用异步方法去执行 nextTick 包装的方法

相关代码如下

javascript 复制代码

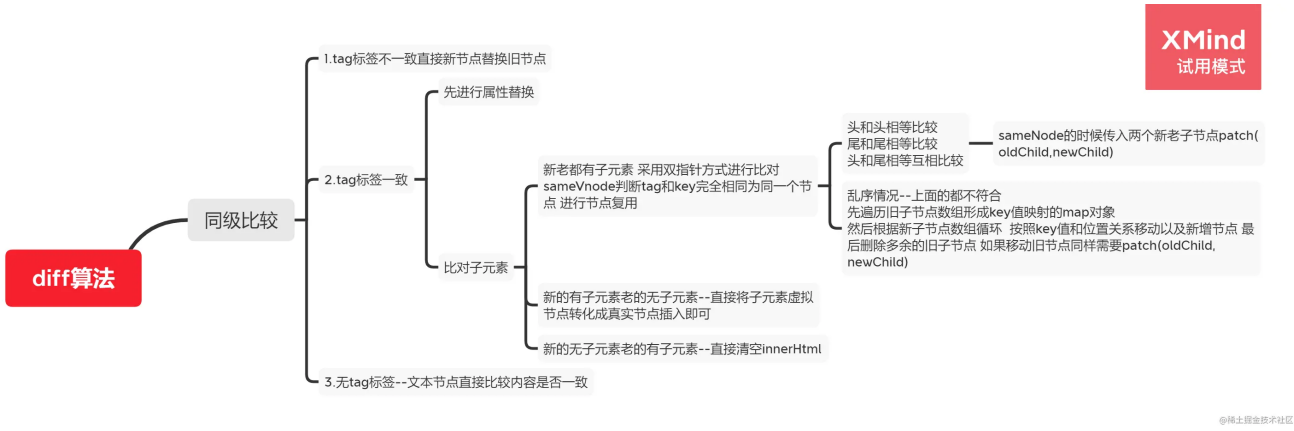
```
let callbacks = [];
let pending = false;
function flushCallbacks() {
  pending = false; //把标志还原为false
  // 依次执行回调
  for (let i = 0; i < callbacks.length; i++) {
    callbacks[i]();
  }
}
let timerFunc; //定义异步方法  采用优雅降级
if (typeof Promise !== "undefined") {
  // 如果支持promise
  const p = Promise.resolve();
  timerFunc = () => {
    p.then(flushCallbacks);
  };
} else if (typeof MutationObserver !== "undefined") {
  // MutationObserver 主要是监听dom变化 也是一个异步方法
  let counter = 1;
  const observer = new MutationObserver(flushCallbacks);
```

```
    const textNode = document.createTextNode(String(counter));
    observer.observe(textNode, {
      characterData: true,
    });
    timerFunc = () => {
      counter = (counter + 1) % 2;
      textNode.data = String(counter);
    };
  } else if (typeof setImmediate !== "undefined") {
    // 如果前面都不支持 判断setImmediate
    timerFunc = () => {
      setImmediate(flushCallbacks);
    };
  } else {
    // 最后降级采用setTimeout
    timerFunc = () => {
      setTimeout(flushCallbacks, 0);
    };
  }

  export function nextTick(cb) {
    // 除了渲染watcher  还有用户自己手动调用的nextTick 一起被收集到数组
    callbacks.push(cb);
    if (!pending) {
      // 如果多次调用nextTick  只会执行一次异步 等异步队列清空之后再把标志变为false
      pending = true;
      timerFunc();
    }
  }
```

nextTick 原理详解 传送门

17 Vue diff 原理



建议直接看 diff 算法详解 传送门

18 路由原理 history 和 hash 两种路由方式的特点

hash 模式

1. location.hash 的值实际就是 URL 中#后面的东西 它的特点在于：hash 虽然出现 URL 中，但不会被包含在 HTTP 请求中，对后端完全没有影响，因此改变 hash 不会重新加载页面。

2. 可以为 hash 的改变添加监听事件

```javascript
window.addEventListener("hashchange", funcRef, false);
```

每一次改变 hash（window.location.hash），都会在浏览器的访问历史中增加一个记录利用 hash 的以上特点，就可以来实现前端路由"更新视图但不重新请求页面"的功能了

> 特点：兼容性好但是不美观

history 模式

利用了 HTML5 History Interface 中新增的 pushState() 和 replaceState() 方法。

这两个方法应用于浏览器的历史记录站，在当前已有的 back、forward、go 的基础之上，它们提供了对历史记录进行修改的功能。这两个方法有个共同的特点：当调用他们修改浏览器历史记录栈后，虽然当前 URL 改变了，但浏览器不会刷新页面，这就为单页应用前端路由"更新视图但不重新请求页面"提供了基础。

> 特点：虽然美观，但是刷新会出现 404 需要后端进行配置

19 手写 bind

```js
//bind实现要复杂一点  因为他考虑的情况比较多  还要涉及到参数合并(类似函数柯里化)
Function.prototype.myBind = function (context, ...args) {
  if (!context || context === null) {
    context = window;
  }
  // 创造唯一的key值  作为我们构造的context内部方法名
```

```
  let fn = Symbol();
  context[fn] = this;
  let _this = this;
  //  bind情况要复杂一点
  const result = function (...innerArgs) {
    // 第一种情况 :若是将 bind 绑定之后的函数当作构造函数，通过 new 操作符使用，则不绑
    // 此时由于new操作符作用  this指向result实例对象  而result又继承自传入的_this 根据
    // this.__proto__ === result.prototype   //this instanceof result =>true
    // this.__proto__.__proto__ === result.prototype.__proto__ === _this.prototyp
    if (this instanceof _this === true) {
      // 此时this指向指向result的实例  这时候不需要改变this指向
      this[fn] = _this;
      this[fn](...[...args, ...innerArgs]); //这里使用es6的方法让bind支持参数合并
      delete this[fn];
    } else {
      // 如果只是作为普通函数调用  那就很简单了 直接改变this指向为传入的context
      context[fn](...[...args, ...innerArgs]);
      delete context[fn];
    }
  };
  // 如果绑定的是构造函数 那么需要继承构造函数原型属性和方法
  // 实现继承的方式: 使用Object.create
  result.prototype = Object.create(this.prototype);
  return result;
};

//用法如下

// function Person(name, age) {
//   console.log(name); //'我是参数传进来的name'
//   console.log(age); //'我是参数传进来的age'
//   console.log(this); //构造函数this指向实例对象
// }
// // 构造函数原型的方法
// Person.prototype.say = function() {
//   console.log(123);
// }
// let obj = {
//   objName: '我是obj传进来的name',
//   objAge: '我是obj传进来的age'
// }
// // 普通函数
// function normalFun(name, age) {
//   console.log(name);   //'我是参数传进来的name'
//   console.log(age);   //'我是参数传进来的age'
//   console.log(this); //普通函数this指向绑定bind的第一个参数 也就是例子中的obj
```

```
//   console.log(this.objName); //'我是obj传进来的name'
//   console.log(this.objAge); //'我是obj传进来的age'
// }

// 先测试作为构造函数调用
// let bindFun = Person.myBind(obj, '我是参数传进来的name')
// let a = new bindFun('我是参数传进来的age')
// a.say() //123

// 再测试作为普通函数调用
// let bindFun = normalFun.myBind(obj, '我是参数传进来的name')
//  bindFun('我是参数传进来的age')
```

19 手写 promise.all 和 race（京东）

js 复制代码

```
//静态方法
static all(promiseArr) {
  let result = [];
  //声明一个计数器 每一个promise返回就加一
  let count = 0;
  return new Mypromise((resolve, reject) => {
    for (let i = 0; i < promiseArr.length; i++) {
    //这里用 Promise.resolve包装一下 防止不是Promise类型传进来
      Promise.resolve(promiseArr[i]).then(
        (res) => {
          //这里不能直接push数组  因为要控制顺序一一对应(感谢评论区指正)
          result[i] = res;
          count++;
          //只有全部的promise执行成功之后才resolve出去
          if (count === promiseArr.length) {
            resolve(result);
          }
        },
        (err) => {
          reject(err);
        }
      );
    }
  });
}
//静态方法
static race(promiseArr) {
  return new Mypromise((resolve, reject) => {
```

```
      for (let i = 0; i < promiseArr.length; i++) {
        Promise.resolve(promiseArr[i]).then(
          (res) => {
            //promise数组只要有任何一个promise 状态变更  就可以返回
            resolve(res);
          },
          (err) => {
            reject(err);
          }
        );
      }
    });
  }
}
```

20 手写-实现一个寄生组合继承

js 复制代码

```
function Parent(name) {
  this.name = name;
  this.say = () => {
    console.log(111);
  };
}
Parent.prototype.play = () => {
  console.log(222);
};
function Children(name) {
  Parent.call(this);
  this.name = name;
}
Children.prototype = Object.create(Parent.prototype);
Children.prototype.constructor = Children;
// let child = new Children("111");
// // console.log(child.name);
// // child.say();
// // child.play();
```

21 手写-new 操作符

js 复制代码

```
function myNew(fn, ...args) {
  let obj = Object.create(fn.prototype);
```

```
  let res = fn.call(obj, ...args);
  if (res && (typeof res === "object" || typeof res === "function")) {
    return res;
  }
  return obj;
}
```

用法如下：

```
// // function Person(name, age) {
// //   this.name = name;
// //   this.age = age;
// // }
// // Person.prototype.say = function() {
// //   console.log(this.age);
// // };
// // let p1 = myNew(Person, "lihua", 18);
// // console.log(p1.name);
// // console.log(p1);
// // p1.say();
```

22 手写-setTimeout 模拟实现 setInterval（阿里）

js 复制代码

```
function mySetInterval(fn, time = 1000) {
  let timer = null,
    isClear = false;
  function interval() {
    if (isClear) {
      isClear = false;
      clearTimeout(timer);
      return;
    }
    fn();
    timer = setTimeout(interval, time);
  }
  timer = setTimeout(interval, time);
  return () => {
    isClear = true;
  };
}

// let a = mySettimeout(() => {
//   console.log(111);
// }, 1000)
```

```
// let cancel = mySettimeout(() => {
//   console.log(222)
// }, 1000)
// cancel()
```

23 手写-发布订阅模式（字节）

js 复制代码

```
class EventEmitter {
  constructor() {
    this.events = {};
  }
  // 实现订阅
  on(type, callBack) {
    if (!this.events[type]) {
      this.events[type] = [callBack];
    } else {
      this.events[type].push(callBack);
    }
  }
  // 删除订阅
  off(type, callBack) {
    if (!this.events[type]) return;
    this.events[type] = this.events[type].filter((item) => {
      return item !== callBack;
    });
  }
  // 只执行一次订阅事件
  once(type, callBack) {
    function fn() {
      callBack();
      this.off(type, fn);
    }
    this.on(type, fn);
  }
  // 触发事件
  emit(type, ...rest) {
    this.events[type] &&
      this.events[type].forEach((fn) => fn.apply(this, rest));
  }
}
// 使用如下
// const event = new EventEmitter();

// const handle = (...rest) => {
```

```
//   console.log(rest);
// };

// event.on("click", handle);

// event.emit("click", 1, 2, 3, 4);

// event.off("click", handle);

// event.emit("click", 1, 2);

// event.once("dbClick", () => {
//   console.log(123456);
// });
// event.emit("dbClick");
// event.emit("dbClick");
```

24 手写-防抖节流（京东）

js 复制代码

```
// 防抖
function debounce(fn, delay = 300) {
  //默认300毫秒
  let timer;
  return function () {
    const args = arguments;
    if (timer) {
      clearTimeout(timer);
    }
    timer = setTimeout(() => {
      fn.apply(this, args); // 改变this指向为调用debounce所指的对象
    }, delay);
  };
}

window.addEventListener(
  "scroll",
  debounce(() => {
    console.log(111);
  }, 1000)
);

// 节流
// 设置一个标志
```

```js
function throttle(fn, delay) {
  let flag = true;
  return () => {
    if (!flag) return;
    flag = false;
    timer = setTimeout(() => {
      fn();
      flag = true;
    }, delay);
  };
}

window.addEventListener(
  "scroll",
  throttle(() => {
    console.log(111);
  }, 1000)
);
```

25 手写-将虚拟 Dom 转化为真实 Dom（类似的递归题-必考）


```
{
  tag: 'DIV',
  attrs:{
  id:'app'
  },
  children: [
    {
      tag: 'SPAN',
      children: [
        { tag: 'A', children: [] }
      ]
    },
    {
      tag: 'SPAN',
      children: [
        { tag: 'A', children: [] },
        { tag: 'A', children: [] }
      ]
    }
  ]
}
```

把上诉虚拟**Dom**转化成下方真实**Dom**

```
<div id="app">
  <span>
    <a></a>
  </span>
  <span>
    <a></a>
    <a></a>
  </span>
</div>
```

答案

```js
// 真正的渲染函数
function _render(vnode) {
  // 如果是数字类型转化为字符串
  if (typeof vnode === "number") {
    vnode = String(vnode);
  }
  // 字符串类型直接就是文本节点
  if (typeof vnode === "string") {
    return document.createTextNode(vnode);
  }
  // 普通DOM
  const dom = document.createElement(vnode.tag);
  if (vnode.attrs) {
    // 遍历属性
    Object.keys(vnode.attrs).forEach((key) => {
      const value = vnode.attrs[key];
      dom.setAttribute(key, value);
    });
  }
  // 子数组进行递归操作 这一步是关键
  vnode.children.forEach((child) => dom.appendChild(_render(child)));
  return dom;
}
```

26 手写-实现一个对象的 flatten 方法（阿里）

题目描述

```css
const obj = {
 a: {
        b: 1,
        c: 2,
        d: {e: 5}
    },
 b: [1, 3, {a: 2, b: 3}],
 c: 3
}

flatten(obj) 结果返回如下
// {
//  'a.b': 1,
//  'a.c': 2,
//  'a.d.e': 5,
//  'b[0]': 1,
//  'b[1]': 3,
//  'b[2].a': 2,
//  'b[2].b': 3
//   c: 3
// }
```

答案

```js
function isObject(val) {
  return typeof val === "object" && val !== null;
}

function flatten(obj) {
  if (!isObject(obj)) {
    return;
  }
  let res = {};
  const dfs = (cur, prefix) => {
    if (isObject(cur)) {
      if (Array.isArray(cur)) {
        cur.forEach((item, index) => {
          dfs(item, `${prefix}[${index}]`);
        });
      } else {
        for (let k in cur) {
          dfs(cur[k], `${prefix}${prefix ? "." : ""}${k}`);
```

```
        }
      }
    } else {
      res[prefix] = cur;
    }
  };
  dfs(obj, "");

  return res;
}
flatten();
```

27 手写-判断括号字符串是否有效（小米）

题目描述

js 复制代码

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。
左括号必须以正确的顺序闭合。

示例 1：

输入：s = "()"
输出：true

示例 2：

输入：s = "()[]{}"
输出：true

示例 3：

输入：s = "(]"
输出：false

答案

```js
const isValid = function (s) {
  if (s.length % 2 === 1) {
    return false;
  }
  const regObj = {
    "{": "}",
    "(": ")",
    "[": "]",
  };
  let stack = [];
  for (let i = 0; i < s.length; i++) {
    if (s[i] === "{" || s[i] === "(" || s[i] === "[") {
      stack.push(s[i]);
    } else {
      const cur = stack.pop();
      if (s[i] !== regObj[cur]) {
        return false;
      }
    }
  }

  if (stack.length) {
    return false;
  }

  return true;
};
```

28 手写-查找数组公共前缀（美团）

题目描述

```
编写一个函数来查找字符串数组中的最长公共前缀。
如果不存在公共前缀，返回空字符串 ""。

示例 1:

输入: strs = ["flower","flow","flight"]
输出: "fl"

示例 2:
```

输入：strs = ["dog","racecar","car"]
输出：""
解释：输入不存在公共前缀。

答案

```js
const longestCommonPrefix = function (strs) {
  const str = strs[0];
  let index = 0;
  while (index < str.length) {
    const strCur = str.slice(0, index + 1);
    for (let i = 0; i < strs.length; i++) {
      if (!strs[i] || !strs[i].startsWith(strCur)) {
        return str.slice(0, index);
      }
    }
    index++;
  }
  return str;
};
```

29 手写-字符串最长的不重复子串

题目描述

给定一个字符串 s ，请你找出其中不含有重复字符的 最长子串 的长度。

示例 1:

输入: s = "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:

输入: s = "bbbbb"
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

示例 3:

输入: s = "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
     请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。

示例 4:

输入: s = ""
输出: 0

答案

```js
const lengthOfLongestSubstring = function (s) {
  if (s.length === 0) {
    return 0;
  }

  let left = 0;
  let right = 1;
  let max = 0;
  while (right <= s.length) {
    let lr = s.slice(left, right);
    const index = lr.indexOf(s[right]);

    if (index > -1) {
      left = index + left + 1;
    } else {
      lr = s.slice(left, right + 1);
      max = Math.max(max, lr.length);
    }
    right++;
  }
  return max;
};
```

30 手写-如何找到数组中第一个没出现的最小正整数 怎么优化（字节）

```ini
给你一个未排序的整数数组 nums ，请你找出其中没有出现的最小的正整数。
请你实现时间复杂度为 O(n) 并且只使用常数级别额外空间的解决方案。
```

```
示例 1:

输入: nums = [1,2,0]
输出: 3

示例 2:

输入: nums = [3,4,-1,1]
输出: 2

示例 3:

输入: nums = [7,8,9,11,12]
输出: 1
```

这是一道字节的算法题 目的在于不断地去优化算法思路

- 第一版 O(n^2) 的方法

```js
const firstMissingPositive = (nums) => {
  let i = 0;
  let res = 1;
  while (i < nums.length) {
    if (nums[i] == res) {
      res++;
      i = 0;
    } else {
      i++;
    }
  }
  return res;
};
```

- 第二版 时间空间均为 O(n)

```js
const firstMissingPositive = (nums) => {
  const set = new Set();
  for (let i = 0; i < nums.length; i++) {
    set.add(nums[i]);
  }
  for (let i = 1; i <= nums.length + 1; i++) {
    if (!set.has(i)) {
```

```
      return i;
    }
  }
};
```

- 最终版 时间复杂度为 O(n) 并且只使用常数级别空间

```js
const firstMissingPositive = (nums) => {
  for (let i = 0; i < nums.length; i++) {
    while (
      nums[i] >= 1 &&
      nums[i] <= nums.length && // 对1~nums.length范围内的元素进行安排
      nums[nums[i] - 1] !== nums[i] // 已经出现在理想位置的，就不用交换
    ) {
      const temp = nums[nums[i] - 1]; // 交换
      nums[nums[i] - 1] = nums[i];
      nums[i] = temp;
    }
  }
  // 现在期待的是 [1,2,3,...]，如果遍历到不是放着该放的元素
  for (let i = 0; i < nums.length; i++) {
    if (nums[i] != i + 1) {
      return i + 1;
    }
  }
  return nums.length + 1; // 发现元素 1~nums.length 占满了数组，一个没缺
};
```

31 手写-怎么在制定数据源里面生成一个长度为 n 的不重复随机数组 能有几种方法 时间复杂度多少（字节）

- 第一版 时间复杂度为 O(n^2)

```js
function getTenNum(testArray, n) {
  let result = [];
  for (let i = 0; i < n; ++i) {
    const random = Math.floor(Math.random() * testArray.length);
    const cur = testArray[random];
    if (result.includes(cur)) {
      i--;
      break;
    }
    result.push(cur);
```

```
  }
  return result;
}
const testArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14];
const resArr = getTenNum(testArray, 10);
```

- 第二版 标记法 / 自定义属性法 时间复杂度为 O(n)

js 复制代码

```
function getTenNum(testArray, n) {
  let hash = {};
  let result = [];
  let ranNum = n;
  while (ranNum > 0) {
    const ran = Math.floor(Math.random() * testArray.length);
    if (!hash[ran]) {
      hash[ran] = true;
      result.push(ran);
      ranNum--;
    }
  }
  return result;
}
const testArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14];
const resArr = getTenNum(testArray, 10);
```

- 第三版 交换法 时间复杂度为 O(n)

js 复制代码

```
function getTenNum(testArray, n) {
  const cloneArr = [...testArray];
  let result = [];
  for (let i = 0; i < n; i++) {
    debugger;
    const ran = Math.floor(Math.random() * (cloneArr.length - i));
    result.push(cloneArr[ran]);
    cloneArr[ran] = cloneArr[cloneArr.length - i - 1];
  }
  return result;
}
const testArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14];
const resArr = getTenNum(testArray, 14);
```

> 值得一提的是操作数组的时候使用交换法 这种思路在算法里面很常见

- 最终版 边遍历边删除 时间复杂度为 O(n)

```js
function getTenNum(testArray, n) {
  const cloneArr = [...testArray];
  let result = [];
  for (let i = 0; i < n; ++i) {
    const random = Math.floor(Math.random() * cloneArr.length);
    const cur = cloneArr[random];
    result.push(cur);
    cloneArr.splice(random, 1);
  }
  return result;
}
const testArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14];
const resArr = getTenNum(testArray, 14);
```

中等

1 Webpack 有哪些优化手段

随着项目越来越大，Webpack 构建速度可能会越来越慢，构建出来的 js 的体积也越来越大，此时就需要对 Webpack 的配置进行优化

这个知识点可以单独开一篇文章 大家请看 带你深度解锁 Webpack 系列(优化篇)

2 css 怎么开启硬件加速(GPU 加速)

浏览器在处理下面的 css 的时候，会使用 GPU 渲染

- transform（当 3D 变换的样式出现时会使用 GPU 加速）
- opacity
- filter
- will-change

```css
采用 transform: translateZ(0)
采用 transform: translate3d(0, 0, 0)
```

使用 CSS 的 will-change属性。 will-change 可以设置为opacity、transform、top、left、b

> 注意！层爆炸，由于某些原因可能导致产生大量不在预期内的合成层，虽然有浏览器的层压缩机制，但是也有很多无法进行压缩的情况，这就可能出现层爆炸的现象（简单理解就是，很多不需要提升为合成层的元素因为某些不当操作成为了合成层）。解决层爆炸的问题，最佳方案是打破 overlap 的条件，也就是说让其他元素不要和合成层元素重叠。简单直接的方式：使用 3D 硬件加速提升动画性能时，最好给元素增加一个 z-index 属性，人为干扰合成的排序，可以有效减少创建不必要的合成层，提升渲染性能，移动端优化效果尤为明显。

3 常用设计模式有哪些并举例使用场景

1.工厂模式 - 传入参数即可创建实例

虚拟 DOM 根据参数的不同返回基础标签的 Vnode 和组件 Vnode

2.单例模式 - 整个程序有且仅有一个实例

vuex 和 vue-router 的插件注册方法 install 判断如果系统存在实例就直接返回掉

3.发布-订阅模式 (vue 事件机制)

4.观察者模式 (响应式数据原理)

5.装饰模式: (@装饰器的用法)

6.策略模式 策略模式指对象有某个行为,但是在不同的场景中,该行为有不同的实现方案-比如选项的合并策略

...其他模式欢迎补充

4 浏览器缓存策略是怎样的（强缓存 协商缓存）具体是什么过程?

这个也是经典的前端缓存问题 知识点加起来是一篇文章了 推荐大家看 前端浏览器缓存知识梳理

5 https 加密过程是怎样的

使用了对称加密可非对称加密的混合方式

具体过程请看 前端进阶高薪必看-HTTPS 篇

6 flex:1 是哪些属性组成的

flex 实际上是 flex-grow、flex-shrink 和 flex-basis 三个属性的缩写。

flex-grow：定义项目的的放大比例；

```
默认为0，即 即使存在剩余空间，也不会放大；
所有项目的flex-grow为1：等分剩余空间（自动放大占位）；
flex-grow为n的项目，占据的空间（放大的比例）是flex-grow为1的n倍。
```


flex-shrink：定义项目的缩小比例；

```
默认为1，即 如果空间不足，该项目将缩小；
所有项目的flex-shrink为1：当空间不足时，缩小的比例相同；
flex-shrink为0：空间不足时，该项目不会缩小；
flex-shrink为n的项目，空间不足时缩小的比例是flex-shrink为1的n倍。
```


flex-basis： 定义在分配多余空间之前，项目占据的主轴空间（main size），浏览器根据此属性计算主轴是否有多余空间

```
默认值为auto，即 项目原本大小；
设置后项目将占据固定空间。
```


7 304 是什么意思 一般什么场景出现， 命中强缓存返回什么状态码

协商缓存命中返回 304

这种方式使用到了 headers 请求头里的两个字段，Last-Modified & If-Modified-Since 。服务器通过响应头 Last-Modified 告知浏览器，资源最后被修改的时间：

Last-Modified: Thu, 20 Jun 2019 15:58:05 GMT

当再次请求该资源时，浏览器需要再次向服务器确认，资源是否过期，其中的凭证就是请求头 If-Modified-Since 字段，值为上次请求中响应头 Last-Modified 字段的值：

If-Modified-Since: Thu, 20 Jun 2019 15:58:05 GMT

浏览器在发送请求的时候服务器会检查请求头 request header 里面的 If-modified-Since，如果最后修改时间相同则返回 304，否则给返回头(response header)添加 last-Modified 并且返回数据(response body)。

另外，浏览器在发送请求的时候服务器会检查请求头(request header)里面的 if-none-match 的值与当前文件的内容通过 hash 算法（例如 nodejs: cryto.createHash('sha1')）生成的内容摘要字符对比，相同则直接返回 304，否则给返回头(response header)添加 etag 属性为当前的内容摘要字符，并且返回内容。

综上总结为：

```sql
请求头last-modified的日期与响应头的last-modified一致
请求头if-none-match的hash与响应头的etag一致
这两种情况会返回Status Code: 304
```

强缓存命中返回 200 200（from cache）

8 手写 Vue.extend 实现

```js
//  src/global-api/initExtend.js
import { mergeOptions } from "../util/index";
export default function initExtend(Vue) {
  let cid = 0; //组件的唯一标识
  // 创建子类继承Vue父类 便于属性扩展
  Vue.extend = function (extendOptions) {
    // 创建子类的构造函数 并且调用初始化方法
    const Sub = function VueComponent(options) {
      this._init(options); //调用Vue初始化方法
    };
    Sub.cid = cid++;
    Sub.prototype = Object.create(this.prototype); // 子类原型指向父类
    Sub.prototype.constructor = Sub; //constructor指向自己
    Sub.options = mergeOptions(this.options, extendOptions); //合并自己的options和
    return Sub;
  };
}
```

具体可以看看这篇 手写 Vue2.0 源码（八）-组件原理

9 vue-router 中路由方法 pushState 和 replaceState 能否触发 popSate 事件

答案是：**不能**

pushState 和 replaceState

HTML5 新接口，可以改变网址(存在跨域限制)而不刷新页面，这个强大的特性后来用到了单页面应用如：vue-router，react-router-dom 中。

注意:仅改变网址,网页不会真的跳转,也不会获取到新的内容,本质上网页还停留在原页面

```js
window.history.pushState(state, title, targetURL);
@状态对象：传给目标路由的信息,可为空
@页面标题：目前所有浏览器都不支持,填空字符串即可
@可选url：目标url，不会检查url是否存在，且不能跨域。如不传该项,即给当前url添加data

window.history.replaceState(state, title, targetURL);
@类似于pushState,但是会直接替换掉当前url,而不会在history中留下记录
```

popstate 事件会在点击后退、前进按钮(或调用 history.back()、history.forward()、history.go()方法)时触发

> **注意:用 history.pushState()或者 history.replaceState()不会触发 popstate 事件**

10 tree shaking 是什么，原理是什么

Tree shaking 是一种通过**清除多余代码方式**来优化项目打包体积的技术，专业术语叫 Dead code elimination

tree shaking 的**原理**是什么?

```sql
ES6 Module引入进行静态分析，故而编译的时候正确判断到底加载了那些模块

静态分析程序流，判断那些模块和变量未被使用或者引用，进而删除对应代码
```

> 扩展：common.js 和 es6 中模块引入的区别?

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：

1、CommonJS 模块输出的是一个值的拷贝，ES6 模块输出的是值的引用。

2、CommonJS 模块是运行时加载，ES6 模块是编译时输出接口（静态编译）。

3、CommonJs 是单个值导出，ES6 Module 可以导出多个

4、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

5、CommonJs 的 this 是当前模块，ES6 Module 的 this 是 undefined

11 babel 是什么，原理了解吗

Babel 是一个 JavaScript 编译器。他把最新版的 javascript 编译成当下可以执行的版本，简言之，利用 babel 就可以让我们在当前的项目中随意的使用这些新最新的 es6，甚至 es7 的语法。

Babel 的三个主要处理步骤分别是： 解析（parse），转换（transform），生成（generate）。

- 解析 将代码解析成抽象语法树（AST），每个 js 引擎（比如 Chrome 浏览器中的 V8 引擎）都有自己的 AST 解析器，而 Babel 是通过 Babylon 实现的。在解析过程中有两个阶段：词法分析和语法分析，词法分析阶段把字符串形式的代码转换为令牌（tokens）流，令牌类似于 AST 中节点；而语法分析阶段则会把一个令牌流转换成 AST 的形式，同时这个阶段会把令牌中的信息转换成 AST 的表述结构。

- 转换 在这个阶段，Babel 接受得到 AST 并通过 babel-traverse 对其进行深度优先遍历，在此过程中对节点进行添加、更新及移除操作。这部分也是 Babel 插件介入工作的部分。

- 生成 将经过转换的 AST 通过 babel-generator 再转换成 js 代码，过程就是深度优先遍历整个 AST，然后构建可以表示转换后代码的字符串。

还想深入了解的可以看 [实践系列]Babel 原理

12 原型链判断

请写出下面的答案

```js
Object.prototype.__proto__;
Function.prototype.__proto__;
Object.__proto__;
Object instanceof Function;
Function instanceof Object;
Function.prototype === Function.__proto__;
```

```js
Object.prototype.__proto__; //null
Function.prototype.__proto__; //Object.prototype
Object.__proto__; //Function.prototype
Object instanceof Function; //true
Function instanceof Object; //true
Function.prototype === Function.__proto__; //true
```

这道题目深入考察了原型链相关知识点 尤其是 Function 和 Object 的之间的关系

强烈推荐大家看看这篇文章 看完就清楚了 JavaScript 原型系列（三）Function、Object、null 等等的关系和鸡蛋问题

13 RAF 和 RIC 是什么

requestAnimationFrame： 告诉浏览器在下次重绘之前执行传入的回调函数(通常是操纵 dom，更新动画的函数)；由于是每帧执行一次，那结果就是每秒的执行次数与浏览器屏幕刷新次数一样，通常是每秒 60 次。

requestIdleCallback：：会在浏览器空闲时间执行回调，也就是允许开发人员在主事件循环中执行低优先级任务，而不影响一些延迟关键事件。如果有多个回调，会按照先进先出原则执行，但是当传入了 timeout，为了避免超时，有可能会打乱这个顺序。

> 这个题目可以深入去问浏览器每一帧的渲染流程 具体可以看看这篇 requestIdleCallback 和 requestAnimationFrame 详解

困难

1 Es6 的 let 实现原理

原始 es6 代码

```js
var funcs = [];
for (let i = 0; i < 10; i++) {
  funcs[i] = function () {
    console.log(i);
  };
}
funcs[0](); // 0
```

babel 编译之后的 es5 代码（polyfill）

```js
var funcs = [];

var _loop = function _loop(i) {
  funcs[i] = function () {
    console.log(i);
  };
};

for (var i = 0; i < 10; i++) {
  _loop(i);
}
funcs[0](); // 0
```

其实我们根据 babel 编译之后的结果可以看得出来 let 是借助闭包和函数作用域来实现块级作用域的效果的 在不同的情况下 let 的编译结果是不一样的

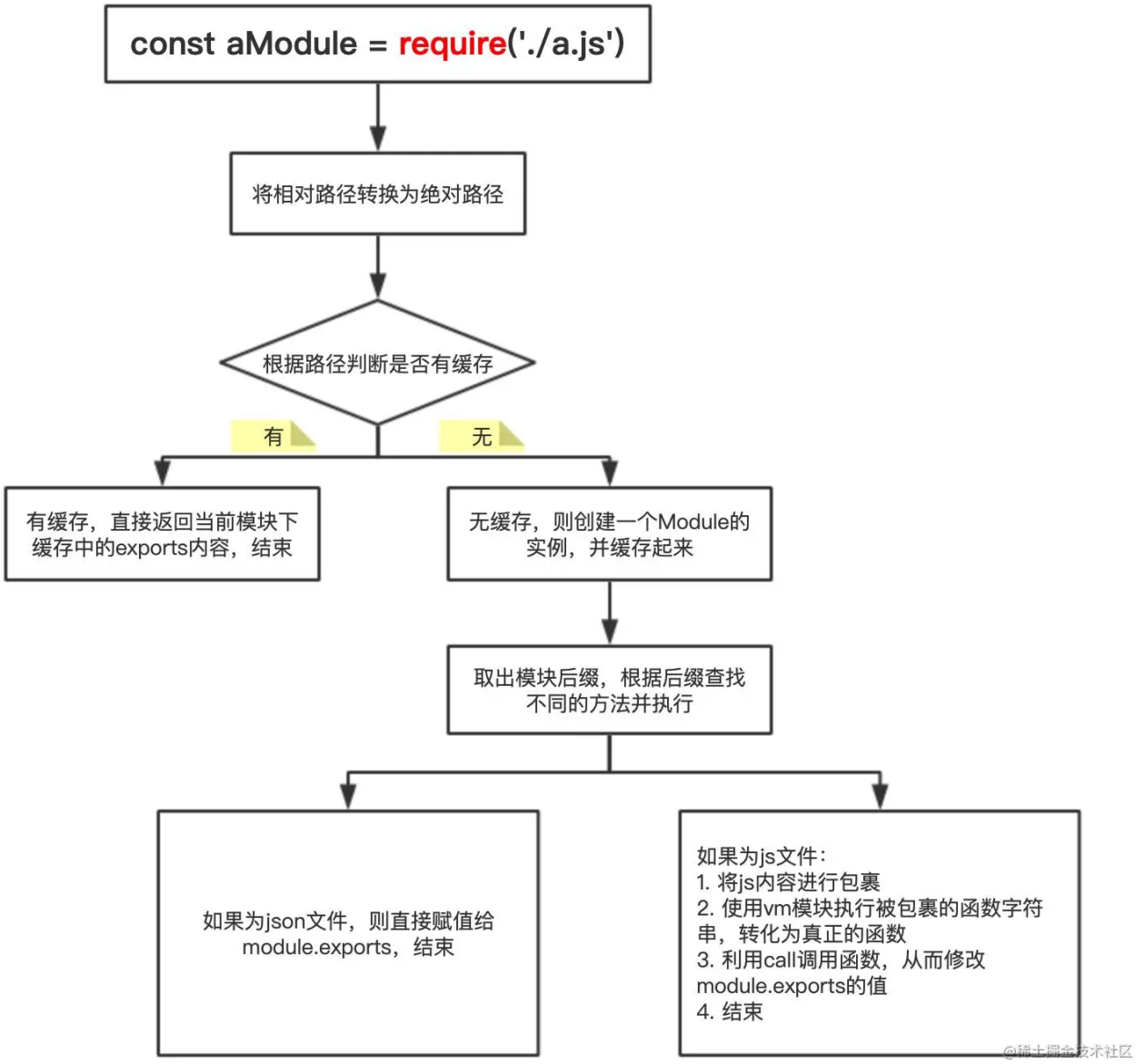
2 如何设计实现一个渲染引擎

这道题是字节终面的最后一个题目 属于**开放性问题** 没有固定答案 我当时觉得题目概念太大了 把我整懵了 我只是回答了下浏览器渲染原理啥的 貌似面试官不太满意 哈哈 如果叫你设计一个渲染引擎 应该从哪些方面着手呢

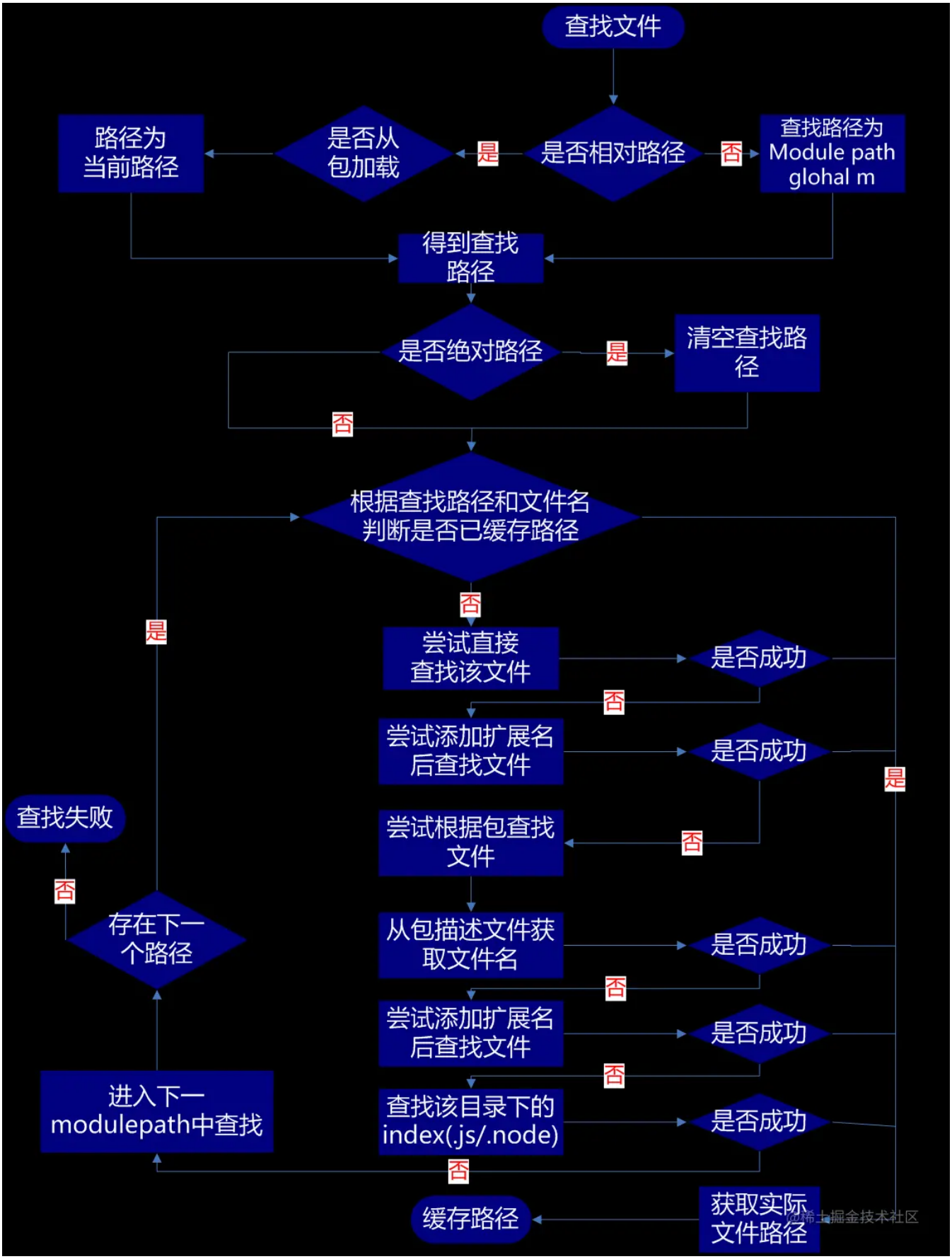
大家可以参考看看文章 你不知道的浏览器页面渲染机制

3 require 具体实现原理是什么

require 基本原理



require 查找路径



require 和 module.exports 干的事情并不复杂，我们先假设有一个全局对象{}，初始情况下是空的，当你 require 某个文件时，就将这个文件拿出来执行，如果这个文件里面存在 module.exports，当运行到这行代码时将 module.exports 的值加入这个对象，键为对应的文件名，最终这个对象就长这样：

```
{
  "a.js": "hello world",
  "b.js": function add(){},
  "c.js": 2,
  "d.js": { num: 2 }
}
```

当你再次 require 某个文件时，如果这个对象里面有对应的值，就直接返回给你，如果没有就重复前面的步骤，执行目标文件，然后将它的 module.exports 加入这个全局对象，并返回给调用者。这个全局对象其实就是我们经常听说的缓存。所以 require 和 module.exports 并没有什么黑魔法，就只是运行并获取目标文件的值，然后加入缓存，用的时候拿出来用就行。

4 前端性能定位以及优化指标

前端性能优化 已经是**老生常谈**的一项技术了 很多人说起性能优化方案的时候头头是道 但是真正的对于性能分析定位和性能指标这块却一知半解 所以这道题虽然和性能相关 但是考察点在于平常项目如何进行**性能定位和分析**