# 作业帮前端一面+二面)(vue2+vue3)

## 八股文

### href 和 src 的区别?

`href` 和 `src` 都是HTML中的属性，但它们的用途和行为有所不同：

- `href` 是Hypertext Reference的缩写，表示超文本引用。它用于在当前元素和文档之间建立链接。常见的使用场景包括：link、a等元素。例如，当我们在link元素中使用href属性来链接CSS文件时，浏览器会识别该文档为CSS文档，并行下载该文档，同时**不会停止**对当前文档的处理。
- `src` 是source的缩写，表示资源的来源。它用于将指向的内容嵌入到文档中当前标签所在的位置。常见的使用场景包括：img、script、iframe等元素。例如，当我们在script元素中使用src属性来链接JavaScript文件时，浏览器在解析到该元素时，**会暂停**浏览器的渲染，直到该资源加载完毕。

简单来说， `src` 用于替换当前元素，而 `href` 用于在当前文档和引用资源之间建立联系。

### 如何放大图片并保证宽高比不变?

#### 方法一

只设置宽或高的其中一项，另一项设置为auto（或者不设置，默认就是auto）,这样是不会改变图片宽高比的。

```
1 img {
2   width: 100%;
3   height: auto; /* 高度自动调整以保持宽高比 */
4 }
5
```

#### 方法二

使用 `object-fit` ，其中 `cover` 属性和 `container` 属性都可以保持宽高比不变，其区别在于

- `object-fit: cover;` ：被替换的内容在保持其宽高比的同时填充元素的整个内容框。如果对象的宽高比与容器不同，那么该对象将被剪裁以填充容器。（容器内不会留下任何空白）
- `object-fit: contain;` ：被替换的内容在保持其宽高比的同时，将被缩放，并尽可能地将其内容在填充元素的内容框中。在保持宽高比的同时缩放图片，意味着某些方向上可能无法完全填充容器（即，如果图片的宽高比与容器的宽高比不同，那么会在容器的一个方向上留下空白）。

```
div {
  width: 1000px;
  height: 1000px;
  overflow: hidden;
}
div > img {
  width: 100%;
  height: 100%;
  object-fit: cover; /* object-fit: contain; */
}
: 100%;  object-fit: cover; /* object-fit: contain; */}
```

这里提一下，把图片用作背景图，设置 `background-size` 也是一样的。

### 方法三

通过 `zoom` 或者 `transform:scale` 直接缩放图片。

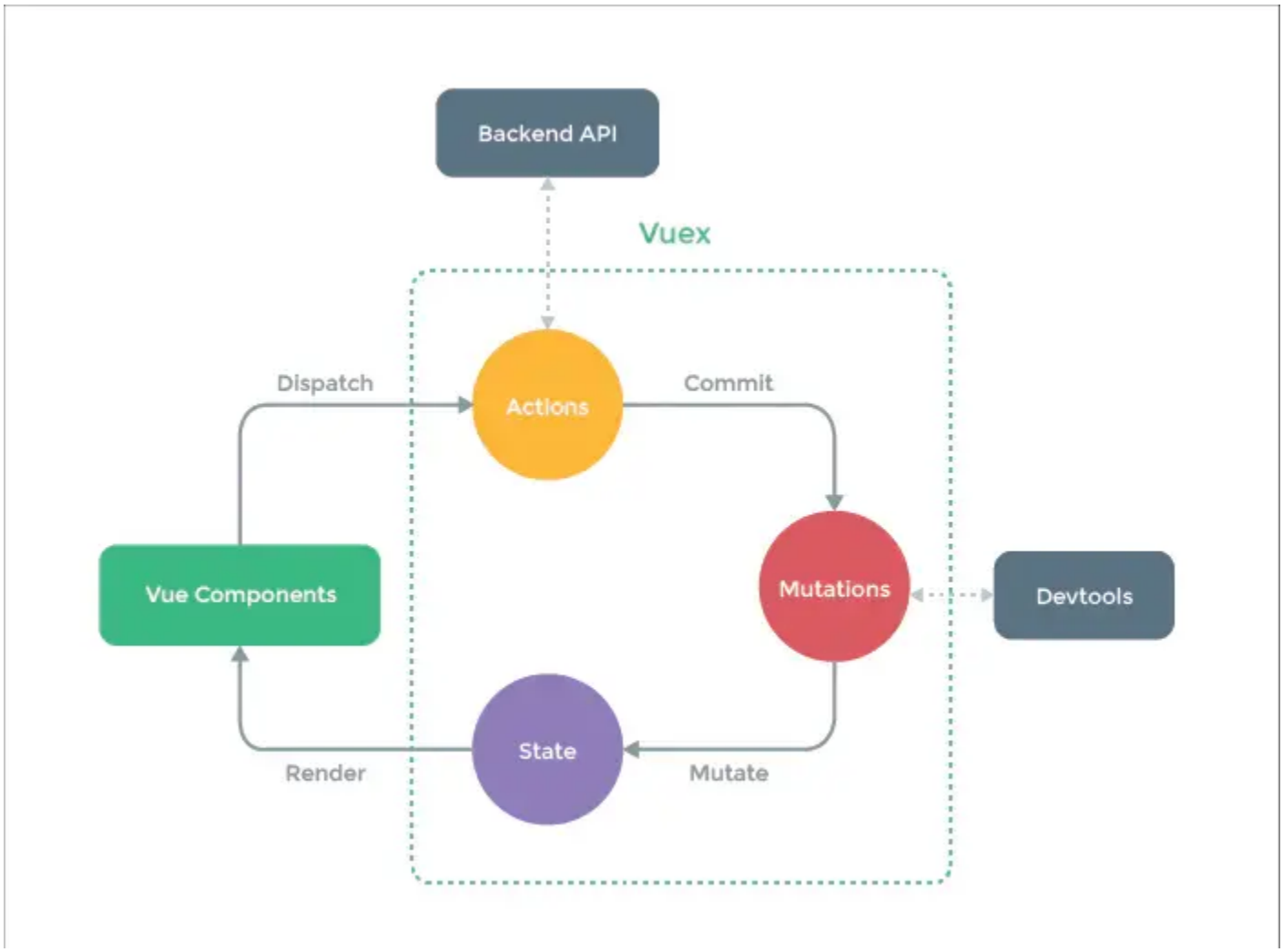
```
img {
  zoom: 1.5 /* 根据需要调整放大比例 */
  transform: scale(1.5); /* 根据需要调整放大比例 */
}

```

## 图片懒加载的原理？如果用户快速下拉到页面底部，会不会导致所有的图片懒加载被触发？如何避免？

## vuex的工作流程？

Vuex 实现了一个单向数据流，在全局拥有一个 State 存放数据，当组件要更改 State 中的数据时，必须通过 Mutation 提交修改信息， Mutation 同时提供了订阅者模式供外部插件调用获取 State 数据的更新。而当所有异步操作(常见于调用后端接口异步获取更新数据)或批量的同步操作需要走 Action ，但 Action 也是无法直接修改 State 的，还是需要通过Mutation 来修改State的数据。最后，根据 State 的变化，渲染到视图上。




## v-if 和 v-show 的区别？什么场景使用 v-if？什么场景使用 v-show？

`v-if` 是动态的向 DOM 树内添加或者删除 DOM 元素，初始为假的时候不会生成对应的 `VNODE`；`v-show` 是通过设置 DOM 元素的display样式属性控制显隐，DOM 元素一开始就会被渲染。这导致了 `v-if` 有更高的切换消耗；`v-show` 有更高的初始渲染消耗。故而 `v-if` 适合不大可能改变的场景，比如根据用户权限展示的元素，用户没有权限就没有必要渲染对应的DOM了；`v-show` 适合频繁切换的场景，比如折叠面板。

## computed 和 watch 的区别？watch 和 watcheffect 的区别？

1. **`computed` 和 `watch` 的区别**：
   - **`computed`**：
     - `computed` 是一个计算属性，它依赖于一个或多个响应式数据，并根据这些依赖自动计算一个新的派生值。
     - `computed` 的结果会被缓存，只有当依赖的响应式数据发生变化时，才会重新计算结果。
     - `computed` 的值是同步获取的，可以像访问普通属性一样使用，不需要显式地调用函数。
   - **`watch`**：
     - `watch` 用于观察一个或多个数据的变化，并在数据变化时执行指定的回调函数。

- `watch` 的回调函数是异步执行的，默认情况下在数据变化后的下一个事件循环周期中执行。
- `watch` 可以用于监听多个数据的变化，也可以执行一些异步操作，比如发起网络请求或执行动画等。

2. **`watch` 和 `watchEffect` 的区别：**
   - **`watch`：**
     - `watch` 需要显式地指定要观察的响应式数据，并在回调函数中处理数据变化。
     - `watch` 的回调函数接收两个参数，新值和旧值，以便你可以比较它们的差异。
     - `watch` 可以监听多个数据，通过配置选项来进行更复杂的操作。
   - **`watchEffect`：**
     - `watchEffect` 是一个更简化的 API，它会自动追踪在其内部使用的响应式数据，并在这些数据变化时自动运行回调函数。
     - `watchEffect` 的回调函数不需要显式地指定要观察的数据，它会自动检测依赖并运行。
     - `watchEffect` 的回调函数不接收新值和旧值，因为它只关心执行代码块时的数据状态。

简单地讲，`computed` 用于计算派生值，`watch` 用于执行自定义操作以响应数据变化，而 `watchEffect` 用于执行具有副作用的代码块。你可以根据具体的需求选择使用其中的一个或多个。

## vue-router路由守卫判断路径不存在跳到404要怎么做?

遍历路由表，比较路径名即可。

```
router.beforeEach((to, from, next) => {
  // 遍历路由表
  const match = router.options.routes.some(route => {
    return route.path === to.path;
  });
  // 如果没有匹配的路由
  if (!match) {
    next('/404'); // 重定向到404页面
  } else {
    next();
  }
});
.some(route => {    return route.path === to.path;  });  // 如果没有匹配的路由  if (!match) {    next('/404'); // 重定向到404页面  } else {    next();  }});
```

拓展： 我们也可以直接在路由表中处理这个问题

```
const routes = [
  // ... 其他路由
  { path: '/:pathMatch(.*)*', name: 'NotFound', component: NotFoundComponent }
]

```

## 箭头函数的this指向哪里?

箭头函数没有自己的 `this`，箭头函数会捕获其在创建时它所在的词法作用域（即外部函数或全局作用域）的 `this` 值。

下面是一个帮助理解的小例子：

```
// 对象并不会创建作用域，所以这里的箭头函数实在全局作用域中创建的
const obj1 = {
  name: "John",
  sayName: function () {
    console.log(this.name);
  },
  sayNameArrow: () => {
    console.log(this.name);
  },
};

obj1.sayName(); // 输出 "John"，普通函数的 this 指向 obj
obj1.sayNameArrow(); // 输出空值，箭头函数的 this 指向全局作用域

// 有外部函数的情况
function outerFunction() {
  this.name = "John";

  const innerArrow = () => {
    console.log(this.name);
  };

  function innerRegular() {
    console.log(this.name);
  }

  innerArrow();
  innerRegular();
}
const obj2 = new outerFunction();
// `this` 在构造函数内部指向新创建的对象，innerArrow 输出 John， innerRegular 输出空值
outerFunction();
```

```
33 // 独立调用普通函数，`this` 会指向全局作用域，innerArrow 输出 John， innerRegular 输出 Jhon
34 ame: function () {    console.log(this.name);  },  sayNameArrow: () => {  console.log(this.name);  },};obj1.sayName(); // 输出 "John"，普通函数的 this 指向objobj1.sayNameArrow(); // 输出空值，箭头函数的 this 指向全局作用域// 有外部函数的情况function outerFunction() {  this.name = "John";  const innerArrow = () => {  console.log(this.name);  };  function innerRegular() {  console.log(this.name);  }  innerArrow();  innerRegular();}const obj2 = new outerFunction();// this 在构造函数内部指向新创建的对象，innerArrow 输出 John，innerRegular 输出空值outerFunction(); // 独立调用普通函数，this 会指向全局作用域，innerArrow 输出 John，  innerRegular 输出 Jhon
```

总结：

- 普通函数的 `this` 是动态的，作为方法被调用的时候，指向调用它的对象；独立被调用的时候，严格模式指向 `undefined`，非严格模式指向 `window`；
- 箭头函数的 `this` 是静态的，它创建时在哪个作用域里，它的 `this` 就和那个作用域的 `this` 一致。
-

## 讲讲 Promise 上的方法?

- **`Promise.resolve(value)`**：返回一个已解决（resolved）的 Promise 对象，其结果值为指定的值 `value`。
- **`Promise.reject(reason)`**：返回一个已拒绝（rejected）的 Promise 对象，其拒绝原因为指定的值 `reason`。
- **`Promise.all(iterable)`**：接收一个可迭代对象（通常是数组），并返回一个新的 Promise 对象，该对象在可迭代对象中的所有 Promise 都已解决时才解决，结果值是一个包含所有 Promise 结果的数组。如果可迭代对象中的任何一个 Promise 被拒绝，它会立即拒绝，并返回拒绝原因。
- **`Promise.race(iterable)`**：接收一个可迭代对象，并返回一个新的 Promise 对象，该对象在可迭代对象中的任何一个 Promise 解决或拒绝时立即解决或拒绝，并采用第一个解决或拒绝的 Promise 的结果或原因。
- **`Promise.allSettled(iterable)`**：接收一个可迭代对象，并返回一个新的 Promise 对象，该对象在可迭代对象中的所有 Promise 都已解决或拒绝时才解决，结果是一个包含所有 Promise 的状态和结果的对象数组，每个对象包含 `status`（解决状态）和 `value` 或 `reason`（结果值或拒绝原因）。

- `Promise.prototype.then(onFulfilled, onRejected)` ：用于添加解决和拒绝时的回调函数。`onFulfilled` 回调在 Promise 解决时调用，接收解决的结果作为参数；`onRejected` 回调在 Promise 拒绝时调用，接收拒绝的原因作为参数。`then` 方法返回一个新的 Promise，允许链式调用。
- `Promise.prototype.catch(onRejected)` ：用于添加拒绝时的回调函数，相当于 `then(null, onRejected)` 。用于处理 Promise 链中的错误。
- `Promise.prototype.finally(onFinally)` ：用于添加一个回调函数，不管 Promise 是解决还是拒绝，都会在最后执行。通常用于执行清理操作。
- `Promise.prototype.catch()` ：该方法没有静态版本，它是通过 `Promise.prototype.then()` 方法来捕获 Promise 链中的错误。如果 `then` 方法的 `onRejected` 回调抛出异常或返回一个拒绝的 Promise，则会被 `.catch()` 方法捕获。

## 代码题

### 圣杯布局+高度满屏

外层div设置 `display: flex` 加上 `height: 100vh` ，内层的三个div左右的固定宽度，中间的设置 `flex: 1` 即可。

### 数组中没出现的最小正整数

输入：[2, 3, 4]， 返回：1；输入：[1，2, 3, 4]， 返回：5；输入：[1，2, 4, 5]， 返回：3；

```
function findNum(arr) {
  let len = arr.length;
  for (let i = 0; i < len; i++) {
      if(arr[i] !== i + 1) {
        return i + 1;
      }
  }

  return arr[arr.length - 1] + 1;
}

```

### 字符串中的乱码处理

“I'm?□□□driving□??□to□?beijing□?□□after□breakfast”

1.只需要大小写英文字母和“’ ”单引号

2.如果乱码的末尾是?则它的下一位字母肯定是大写;

示例结果：I'm driving to Beijing after breakfast.

```
let string = "I’m?   driving ?? to ?beijing ?  after breakfast";

// 字母转大写
string = string.replace(/ \?([a-z])/g, (match) => match.toUpperCase());

// 只保留英文和单引号
string = string.replace(/[^a-zA-Z’]/g, ' ');
string = string.replace(/ {2,}/g, ' ');

console.log(string);
g.replace(/ \?([a-z])/g, (match) => match.toUpperCase());// 只保留英文和单引号
string = string.replace(/[^a-zA-Z’]/g, ' ');string = string.replace(/ {2,}/g,
' ');console.log(string);
```

当时的第一反应就是：完了，我不会正则，这怎么处理？遍历了半天字符串，给面试官都整无语了。
那么不用正则怎么做呢：

```
let string = "I'm?   driving ?? to ?beijing ?  after breakfast";

arr = string.split(" ")

const res = []
for (let i of arr) {
  if (i.length !== 0) {
    if (i === '?') {
      res.push(i)
    } else if (i.charAt(0) === '?') {
      res.push(i.charAt(1).toUpperCase() + i.slice(2))
    } else {
      res.push(i)
    }
  }
}

console.log(res.join(" ").split("?").map((el) => el.trim()).join(" "))
st res = []for (let i of arr) {  if (i.length !== 0) {    if (i === '?') {
 res.push(i)    } else if (i.charAt(0) === '?') {
res.push(i.charAt(1).toUpperCase() + i.slice(2))    } else {      res.push(i)
  }  }}console.log(res.join(" ").split("?").map((el) => el.trim()).join(" "))
```

但其实这里的做法隐含条件是比较多的，比如乱码段必然包含和 `?` 和 ` `，`?` 不会单独出现；以及乱码的结尾如果是 `?`，`?` 是单独的而不是连续的。但当时面试官给的题目也没有特别详细的说明条件，我也没有录屏，记忆大概就是这样了。

## 和最大的三个子数组

有一个数组 **[[1, 2, 3], [4, 5, 6], [-1, 12, 13], [6, 18, 0], [5, 5, 5], [6, 9, 3]]**,找出其中和最大的三个子数组

```
function findMaxSubArr(array) {

  // 找出和最大的三个子数组
  const sortedArray = array.sort((a, b) => calculateSum(b) - calculateSum(a));
  const topThreeArrays = sortedArray.slice(0, 3);

  return topThreeArrays;
}

// 计算子数组的和
function calculateSum(arr) {
  return arr.reduce((sum, val) => sum + val, 0);
}
, b) => calculateSum(b) - calculateSum(a));  const topThreeArrays = sortedArray.slice(0, 3);  return topThreeArrays;}// 计算子数组的和function calculateSum(arr) {  return arr.reduce((sum, val) => sum + val, 0);}
```

这题当时我开了个数组，把每个子数组加上它的和作为一个对象的两个属性，对这个对象数组按照和进行排序，还把 `sort` 回调函数的返回值和升降序的关系搞反了，老丢人了。

| compareFn(a, b) 返回值 | 排序顺序 |
| --- | --- |
| > 0 | a 在 b 后，如 [b, a] |
| < 0 | a 在 b 前，如 [a, b] |
| === 0 | 保持 a 和 b 原来的顺序 |