# 2023-07-16 前端面试题

<a href="/interview/2023-07-16-js基础.doc" download>2023-07-16-js 基础.doc</a>

1. Vue2 和 Vue3 的区别？

2. Vue3 如何实现响应式

3. Vue nextTick 实现原理？

4. Vue 路由的原理？

5. Vue 和 React 的区别？

6. React Memo 和 Callback 的区别？

7. 写一个并发限制的管理类 NetManager？

8. 给出一个数组[3,1,8,2,5,2,4,6,7]，求两个数之和为 6 的下标？

## 1. Vue2 和 Vue3 的区别？

官方文档：https://cn.vuejs.org/about/faq.html#what-s-the-difference-between-vue-2-and-vue-3

1. \*\*响应式系统\*\*：Vue 3 使用 JavaScript 的 Proxy 对象来实现响应式，相比 Vue 2 的 Object.defineProperty，它能检测到任何类型的变化（包括数组和对象的新增/删除），提供更好的性能和内存使用。

2. \*\*Composition API\*\*：Vue 3 引入了 Composition API，这是一个新的可选的、更灵活的方法来组织组件的代码。这不意味着 Vue 2 的 Options API 不再可用，Vue 3 仍然完全支持 Options API。

3. \*\*性能\*\*：Vue 3 对虚拟 DOM 的实现进行了优化，有更好的性能和更小的内存使用。静态内容和动态内容的处理更为智能，渲染和更新的速度更快。

4. \*\*Fragment 和 Teleport\*\*：Vue 3 支持 Fragment，也就是多个根节点的组件。另外，Vue 3 还引入了 Teleport，可以将组件的子节点渲染到 DOM 树的任何位置。

5. \*\*更好的 TypeScript 支持\*\*：Vue 3 的源代码完全用 TypeScript 编写，这带来了更好的 TypeScript 集成和类型检查支持。

6. \*\*尺寸和兼容性\*\*：虽然 Vue 3 增加了很多新特性，但它的运行时版本实际上比 Vue 2 更小。然而，由于使用了 Proxy，Vue 3 不再支持不支持 Proxy 的旧浏览器，如 IE。

7. \*\*自定义渲染 API\*\*：Vue 3 引入了更加底层的自定义渲染 API，为开发者提供了更大的灵活性。

这些是 Vue 2 和 Vue 3 的一些主要区别。对于具体项目的选择，需要根据项目的实际需求和目标浏览器的兼容性进行考虑。

## 2. Vue3 如何实现响应式？

Vue.js 3.0 使用的是 Proxy 对象来实现响应式，而非 Vue.js 2.x 中的 Object.defineProperty 方法。这是 Vue 3.0 中最重要的改变之一。Proxy 提供了很多的附加功能，允许 Vue.js 拦截并基于原始对象操作，包括读取属性、设置属性、枚举属性等。在设计上，Proxy 的这些附加功能使得 Vue 3.0 可以更简洁、更直接地实现响应式系统。

Vue 3.0 的响应式系统大致工作方式如下：

1. \*\*创建响应式对象\*\*：当数据对象被访问时，Vue 3.0 使用 Proxy 创建一个响应式代理对象。这个代理对象“包装”原始数据，添加了附加的拦截操作。

2. \*\*依赖收集\*\*：当代理对象的属性被访问时，Vue 3.0 记录依赖关系。这个依赖关系表示某个属性是由哪些副作用（例如渲染函数或计算属性）依赖的。

3. \*\*依赖触发\*\*：当代理对象的属性被修改时，Vue 3.0 将执行所有依赖于该属性的副作用。

这就是 Vue 3.0 响应式系统的基本工作原理。在设计上，它比 Vue 2.x 更直接，更能够处理 JavaScript 数据类型的各种边缘情况，包括数组和新增的数据类型，如 Map、Set 和 Symbol。

注意，Proxy 对象无法被 polyfill，因此 Vue 3.0 不支持不支持 Proxy 的旧浏览器，如 IE。如果你需要支持这些浏览器，你需要继续使用 Vue 2.x 或者其他支持这些浏览器的库。

## 3. Vue nextTick 实现原理？

Vue 的 `nextTick` 是一个非常有用的工具，它使我们能够在 DOM 更新后执行某些操作。要了解 `nextTick` 的实现原理，我们需要了解 Vue 的异步渲染机制。

Vue 使用了一种叫做异步渲染的策略。当数据改变时，Vue 不会立即更新 DOM，而是开始排队等待所有的改变。然后，在下一个 "tick" 中，Vue 会批量执行这些改变。这种策略可以避免因数据频繁改变而导致的不必要的渲染和重新渲染。

`nextTick` 函数的作用就是将你传入的回调函数推迟到下一个 "tick" 中执行。在这个 "tick" 中，DOM 已经完成了更新。所以，你可以在 `nextTick` 的回调函数中进行一些依赖于新 DOM 的操作。

那么，这个 "tick" 是什么呢？在 Vue 2.x 中，"tick" 的实现方式首先会尝试使用 `Promise.then` 和 `MutationObserver`，如果环境不支持，会降级为 `setImmediate` 和 `setTimeout`。

在 Vue 3.0 中，`nextTick` 的实现则直接使用了 Promise-based 微任务队列。

这样一来，`nextTick` 实际上就成了一个异步任务队列，可以保证在所有同步任务执行完成，DOM 更新完成后，再执行 `nextTick` 中的回调。

因此，你可以将 `nextTick` 视为一个在 DOM 更新后执行代码的“桥梁”，可以确保你的代码在正确和预期的时机运行。

## 4. Vue 路由的原理？

很抱歉前一次的回答没有完全满足你的要求，让我重新阐述一下 Vue 路由的原理，包括所有三种模式：hash 模式、history 模式以及 abstract 模式。

Vue Router 是 Vue.js 的官方路由库，它使用不同的策略来实现前端路由，主要包括以下三种模式：

1. \*\*Hash 模式\*\*：在此模式下，Vue Router 使用 URL 的哈希（#）来模拟完整的 URL，从而在无需页面刷新的情况下实现页面跳转。由于 URL 的哈希部分变动不会触发页面请求，所以这为前端路由提供了可能性。此外，每次哈希值的变化都会引发一个 `hashchange` 事件，Vue Router 利用监听这个事件的方式来驱动路由的变化。

2. \*\*History 模式\*\*：在此模式下，Vue Router 使用 HTML5 的 History API 来管理路由。HTML5 History API 提供了 `pushState`、`replaceState` 和 `popstate` 事件，使得我们可以操作浏览器的历史记录而无需引发页面刷新，这使得我们可以改变 URL 而页面不会重新加载。Vue Router 会监听 `popstate` 事件，当浏览器的历史记录发生变化时，进行路由的切换。

3. \*\*Abstract 模式\*\*：在此模式下，Vue Router 在内存中模拟一个 URL 栈来进行路由管理，适用于不支持 History API 或不能使用 hash 的环境，例如在 Node.js 进行服务器端渲染（SSR）或在如 Weex、NativeScript 等非浏览器环境下。

无论你使用哪种模式，当你点击 `<router-link>` 组件或者通过编程式导航（例如 `router.push` 或 `router.replace`）进行导航时，Vue Router 会改变 URL，并在内部保持一个路由对象的队列，每一个路由对象都代表一个 "页面" 或者说一个组件。

然后，Vue Router 会根据当前的 URL 和预先定义的路由映射（在你调用 `new VueRouter` 时传入的 `routes` 选项）来决定哪个组件应该被渲染，并将其渲染到 `<router-view>` 组件的位置。

这就是 Vue Router 的基本工作原理。在实际使用中，还涉及到许多其他的概念和技术，例如路由参数、路由守卫、路由懒加载等。

## 5. Vue 和 React 的区别？

Vue 和 React 都是非常流行的前端框架，但它们有一些关键的区别：

- 语法差异：Vue 使用了一个基于模板的 DSL，对于 HTML/CSS/JavaScript 有基础的开发者来说可能更容易上手。React 则全部使用 JavaScript 编写，采用 JSX 语法，可以利用 JavaScript 的全部能力。

- 状态管理：Vue 有 Vuex，React 有 Redux 或 Context API。

- Reactivity：Vue 使用 reactive 数据流，数据改变会自动触发更新。React 则使用的是 setState 或者 Hooks，需要手动触发状态更新。

- 社区和生态：两者社区都非常活跃，生态也非常丰富。

---

以下是 Vue 和 React 在一些关键点上的比较：

1. \*\*编程模型\*\*：Vue 使用了更为传统的分离式的编程模型，通过模板（Template）、脚本（Script）、样式（Style）三部分构成一个组件。而 React 则推崇一种更为集成的编程模型，JSX 技术让你在 JavaScript 代码中写 HTML 结构。

2. \*\*数据绑定\*\*：Vue 提供双向数据绑定，这在表单处理等场景下更为便利。而 React 只提供单向数据流，状态的变更需要开发者显式地处理。

3. \*\*模板 vs JSX\*\*：Vue 使用基于 HTML 的模板语法，可以直接使用 HTML、CSS、JavaScript 进行开发。而 React 使用 JSX，是一种将 HTML 语法直接插入到 JavaScript 代码中的语法糖，使用 JSX 可能需要一些学习成本。

4. \*\*响应式机制\*\*：Vue 使用的是依赖追踪的响应式系统，在变量被访问时收集依赖，当变量变化时通知所有依赖进行更新。而 React 采用的是更为传统的比较式（Diffing）的响应式机制，当状态变化时，重新执行 render 函数，然后对比新旧虚拟 DOM 的差异，最后更新实际 DOM。

5. \*\*生态系统\*\*：React 的生态系统更大，社区也更活跃，拥有大量的第三方库和工具。Vue 的社区相对较小，但也在稳步增长，并且 Vue 的核心库和工具（如 Vue Router、Vuex 等）的集成度更高。

6. \*\*学习曲线\*\*：Vue 的学习曲线相对更平缓，API 设计较为简洁，更适合初学者。而 React 的学习曲线稍陡，特别是引入了 JSX、Hooks 等概念后。

7. \*\*性能\*\*：Vue 和 React 都有出色的性能，都使用虚拟 DOM 技术减少对实际 DOM 的操作。具体的性能表现会因应用的具体情况而异。

8. \*\*企业支持\*\*：React 由 Facebook 维护和支持，Vue 由尤雨溪领导的团队维护，都得到了大量的开发者和企业的支持。

每种框架都有其优点和缺点，选择哪种框架取决于你的项目需求、团队的技术背景和个人喜好等因素。

## 6. React Memo 和 Callback 的区别？Vue3 中 watch 和 watchEffect 的区别？

`React.memo`和`React.useCallback`都是 React 库提供的优化性能的方法，但是它们在使用和目标上有一些关键区别。

`React.memo`是一个高阶组件，用于优化函数组件的性能。它通过记住组件的上次渲染结果，如果组件的 props 没有发生变化，那么将跳过渲染步骤并复用上次的渲染结果。这可以大大减少不必要的渲染，特别是在复杂的组件树中。

相反，`React.useCallback`是一个 hook，它返回一个记忆版本的回调函数，这将在依赖项更改时重新计算。它的主要目标是避免不必要的重新渲染，由于父组件传递给子组件的回调函数的身份更改。因此，`useCallback`对于防止因为父组件重新渲染导致无状态子组件也跟着重新渲染是很有用的。

而在 Vue3 中，`watch`和`watchEffect`都是用来观察和响应 Vue 实例上的数据变动的。它们的主要区别在于它们观察数据的方式和时间。

`watch`允许你监听单个属性或多个属性，并在其改变时运行一个函数。它提供了旧值和新值，以及手动停止观察的能力。然而，`watch`只有在属性值发生改变时才运行。

`watchEffect`则会立即运行传递给它的函数，并追踪其依赖项。如果任何依赖项变更，`watchEffect`都将再次运行。但是，与`watch`不同，`watchEffect`不提供旧值，也不能监听特定的属性或组合。

在比较`React.memo`/`useCallback`与`watch`/`watchEffect`时，应注意这些库在设计和功能上有着根本的差异。React 的这两个函数更关注性能优化和避免不必要的重新渲染，而 Vue 的这两个函数更关注响应式系统和数据变化的观察与处理。

## 7. 写一个并发限制的管理类 NetManager？

这是一个基础的实现，您可能需要根据具体需求进行修改：

```javascript

// 实现网络请求并发限制的管理类 NetManager(concurrency)，

// 构造函数传入一个number，是并发的限制数，

// 提供 request(url, data) => Promise<res> 方法

// Const obj = new NetManager(5)

// obj.request(“/xxx/sss”, {a: 1})

// net-manager.js

class NetManager {

 constructor(number) {

 if (!(typeof number === "number" && !Number.isNaN(number))) {

 console.error(

 `NetManager params typeof number === '${typeof number}', value: ${number}`

 );

 }

 this.number = number;

 this.queues = [];

 this.caches = [];

 }

 trigger = () => {

 const hits = this.queues.filter((i) => i.isFetch === false);

 hits.forEach((item) => {

 item.isFetch = true;

 item

 .task()

 .then(item.resolve)

 .catch(item.reject)

 .finally(() => {

 const deleteIndex = this.queues.findIndex(

 (del) => del.key === item.key

 );

 if (deleteIndex !== -1) {

 this.queues.splice(deleteIndex, 1);

 }

 if (this.caches.length > 0) {

 this.queues.push(this.caches.shift());

 this.trigger();

 }

 });

 });

 };

 request = (...reset) => {

 return new Promise((resolve, reject) => {

 // 绑定一个函数并传参

 const task = window.fetch.bind(null, ...reset);

 // 生成一个key值，用于删除队列任务

 const key = Math.random();

 const newItem = { key, isFetch: false, task, resolve, reject };

 if (this.queues.length >= this.number || this.caches.length > 0) {

 this.caches.push(newItem);

 } else {

 this.queues.push(newItem);

 this.trigger();

 }

 });

 };

}

const JSON\_PLACEHOLDER = "https://jsonplaceholder.typicode.com/todos/";

const obj = new NetManager(2);

for (let i = 1; i <= 10; i++) {

 obj

 .request(`${JSON\_PLACEHOLDER}${i}`, { credentials: "include" })

 .then((res) => res.json())

 .then((result) => {

 console.log("result", result);

 })

 .catch((err) => {

 console.error(err);

 });

}

```

## 8. 给出一个数组[3,1,8,2,5,2,4,6,7]，求两个数之和为 6 的下标？

```javascript

function twoSum2(nums, target) {

 const result = [];

 for (let i = 0; i < nums.length - 1; i++) {

 const item1 = nums[i];

 for (let j = i + 1; j < nums.length; j++) {

 const item2 = nums[j];

 if (item1 + item2 === target) {

 result.push([i, j]);

 }

 }

 }

 return result;

twoSum([3, 1, 8, 2, 5, 2, 4, 6, 7], 6);

```

```javascript

function twoSum(nums, target) {

 const map = new Map();

 const results = [];

 nums.forEach((num, i) => {

 if (!map.has(num)) {

 map.set(num, []);

 }

 map.get(num).push(i);

 });

 map.forEach((indices, num) => {

 const complement = target - num;

 if (map.has(complement)) {

 const complementIndices = map.get(complement);

 for (const index of indices) {

 for (const complementIndex of complementIndices) {

 if (index !== complementIndex) {

 results.push([index, complementIndex]);

 }

 }

 }

 map.delete(num);

 map.delete(complement);

 }

 });

 return results;

}

console.log(twoSum([3, 1, 8, 2, 5, 2, 4, 6, 7], 6));

```