2024 最新 React 面试题

1. **React18 新特性**

18 不在支持 IE

批处理

18 之前，批处理 `只限于 React 原生事件内部` 的更新。

18 中，批处理支持处理的操作范围扩大了： `Promise，setTimout，native event handler` 等这些非 React 原生事件。

Transitions

- `starTransition`：用于标记非紧急的更新，用 starTransition 包裹起来就是告诉 React，这部分代码渲染的优先级不高，可以优先处理其它更重要的渲染。
- `useTransition`：除了能提供 startTransition 以外，还能提供一个变量来跟踪当前渲染的执行状态。

Suspense

ReactDom.createRoot

ReactDom.hydrateRoot

New Hooks

- `useTransition`：用来标记低优先的 state 更新
- `useDeferredValue`：可以用来标记低优先的变量

架构演进

- React 15 主要分为 `Reconciler 协调器` 和 `Renderer 渲染器` 两部分：
  - Reconciler 负责生成虚拟 DOM 并进行 diff，找出变动的虚拟 DOM，
  - 然后 Renderer 负责将变化的组件渲染到不同的宿主环境中。
- React 16 多了一层 `Scheduler 调度器`，并且 `Reconciler 协调器` 的部分基于 `Fiber` 完成了重构。
- React 17 是一个用以稳定 `concurrent mode 并行模式` 的过渡版本，另外，它使用 `Lanes` 重构了优先级算法。

- Lane 用二进制位表示任务的优先级，方便优先级的计算（位运算），不同优先级占用不同位置的“赛道”，而且存在批的概念，优先级越低，“赛道”越多。高优先级打断低优先级，新建的任务需要赋予什么优先级等问题都是 Lane 所要解决的问题。

流程

整个 `Reconciliation` 的流程可以简单地分为两个阶段：

- `Render 阶段`：当 React 需要进行 re-render 时，会遍历 Fiber 树的节点，根据 diff 算法将变化应用到 `workInProgress` 树上，这个阶段是随时可中断的。
- `Commit 阶段`：当 `workInProgress` 树构建完成之后，将其作为 `Current` 树，并把 DOM 变动绘制到页面上，这个阶段是不可中断的，必须一气呵成，类似操作系统中「原语」的概念。
- workInProgress tree 代表当前正在执行更新的 Fiber 树
- currentFiber tree 表示上次渲染构建的 Filber 树

Scheduler

对于大部分浏览器来说，`每 1s 会有 60 帧`，所以每一帧差不多是 `16.6 ms`，如果 Reconciliation 的 Render 阶段的更新时间过长，挤占了主线程其它任务的执行时间，就会导致页面卡顿。

思路：

- 将 re-render 时的 JS 计算拆分成更小粒度的任务，可以随时暂停、继续和丢弃执行的任务。
- 当 JS 计算的时间达到 16 毫秒之后使其暂停，把主线程让给 UI 绘制，防止出现渲染掉帧的问题。
- 在浏览器空闲的时候继续执行之前没执行完的小任务。

React 给出的解决方案是将整次 Render 阶段的长任务拆分成多个小任务：

- 每个任务执行的时间控制在 5ms。
- 把每一帧 5ms 内未执行的任务分配到后面的帧中。
- 给任务划分优先级，同时进行时优先执行高优任务。

2. **如何把每个任务执行的时间控制在 5ms?**

Scheduler 提供的 `shouldYield` 方法在 源码 中叫 `shouldYieldToHost`，它通过综合判断已消耗的时间（是否超过 5ms）、是否有用户输入等高优事件来决定是否需要中断遍历，给浏览器渲染和处理其它任务的时间，防止页面卡顿。

3. **如何把每一帧 5ms 内未执行的任务分配到后面的帧中?**

时间切片

如果任务的执行因为超过了 5ms 等被中断了，那么 React Scheduler 会借助一种 `效果接近于 setTimeout` 的方式来开启一个宏任务，预定下一次的更新。

React 是在借助 `MessageChannel 模拟 setTimeout` 的行为，将未完成的任务以宏任务的形式发放给浏览器，被动地让浏览器自行安排执行时间。

而 requestIdleCallback 是主动从浏览器处获取空闲信息并执行任务，个人感觉不太像是一种对 requestIdleCallback 的 polyfill。

4. **如何给任务划分优先级?**

基于 `Lanes` 的优先级控制。

不同的 Lanes 可以简单理解为不同的数值，数值越小，表明优先级越高。比如：

- 用户事件比较紧急，那么可以对应比较高的优先级如 SyncLane；
- UI 界面过渡的更新不那么紧急，可以对应比较低的优先级如 TransitionLane；
- 网络加载的更新也不那么紧急，可以对应低优先级 RetryLane。

5. **对 React Hook 的闭包陷阱的理解?**

React Hooks 的闭包陷阱发生在 `useState` 钩子函数中的示例：

```
function Counter() {
    const [count, setCount] = useState(0);
    const handleClick = () => {
      setTimeout(() => {
        setCount(count + 1);
      }, 1000);
    };
    const handleReset = () => {
      setCount(0);
    };
    return (
      <div>
        <p>Count: {count}</p>
        <button onClick={handleClick}>Increment</button>
        <button onClick={handleReset}>Reset</button>
      </div>
    );
}
```

改进方法：

```
const handleClick = () => {
    setTimeout(() => {
        setCount(count => count + 1);
    }, 1000);
};
```

React Hooks 中的闭包陷阱通常发生在 `useEffect` 钩子函数中的示例：

```
function App() {
    const [count, setCount] = useState(0);
    useEffect(() => {
        const timer = setInterval(() => {
        console.log(count);
        }, 1000);
        return () => clearInterval(timer);
    }, []);
    const handleClick = () => {
        setCount(count + 1);
    };
    return (
        <div>
        <p>Count: {count}</p>
        <button onClick={handleClick}>Increment</button>
        </div>
    );
}
```

改进方法：

```
useEffect(() => {
    const timer = setInterval(() => {
      console.log(count);
    }, 1000);
    return () => clearInterval(timer);
}, [count]);
```

为什么不能将hooks写到if else语句中了把?

react 用链表来严格保证hooks的顺序。

因为这样可能会导致顺序错乱，导致当前 hooks 拿到的不是自己对应的 Hook 对象。

6. 让 useEffect 支持 async/await ?

- 创建一个异步函数（async...await 的方式），然后执行该函数。

```
useEffect(() => {
  const asyncFun = async () => {
```

```
    setPass(await mockCheck());
  };
  asyncFun();
}, []);
```

- 也可以使用 IIFE，如下所示：

```
useEffect(() => {
  (async () => {
    setPass(await mockCheck());
  })();
}, []);
```

- ahooks useAsyncEffect

```
function useAsyncEffect(
    effect: (isCanceled: () => boolean) => Promise<void>,
    dependencies?: any[]
  ) {
    return useEffect(() => {
      let canceled = false;
      effect(() => canceled);

      return () => {
        canceled = true;
      }
    }, dependencies)
  }
```

7. **React 性能优化**

减少计算量

- 减少渲染的节点/降低渲染计算量(复杂度)
- 不要在渲染函数都进行不必要的计算
  - 比如不要在渲染函数(render)中进行数组排序、数据转换、订阅事件、创建事件处理器等等.
- 减少不必要的嵌套
- 虚拟列表
- 惰性渲染

- CSS > 大部分 CSS-in-js > inline style

利用缓存

- 避免重新渲染
  - shouldComponentUpdate
  - React.memo
- 简化的 props 更容易理解, 且可以提高组件缓存的命中率
- 不变的事件处理器
  - useCallback
- 不可变数据
  - Immutable.js、Immer、immutability-helper 以及 seamless-immutable。
- 简化 state

精确重新计算的范围

- 响应式数据的精细化渲染
- 不要滥用 Context
  - 一旦 Context 的 Value 变动，所有依赖该 Context 的组件会全部 forceUpdate.

8. **为什么 useState 返回的是数组而不是对象?**

因为解构赋值的原因：

- 返回数组，可以对数组中的变量命名，代码看起来也比较干净。
- 返回对象，那就必须和返回的值同名，不能重复使用了。

9. **React 懒加载的实现原理?**

React.lazy

React 16.6 之后，React 提供了 `React.lazy` 方法来支持组件的懒加载。配合 webpack 的 code-splitting 特性，可以实现按需加载。

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
```

```
      </Suspense>
    </div>
  );
}
```

如上代码中，通过 `import()、React.lazy 和 Suspense` 共同一起实现了 React 的懒加载，也就是我们常说了运行时动态加载，即 OtherComponent 组件文件被拆分打包为一个新的包（bundle）文件，并且只会在 OtherComponent 组件渲染时，才会被下载到本地。

> React.lazy 需要配合 Suspense 组件一起使用，在 Suspense 组件中渲染 React.lazy 异步加载的组件。如果单独使用 React.lazy，React 会给出错误提示。

Webpack 动态加载

`import() 函数` 是由 TS39 提出的一种动态加载模块的规范实现，其返回是一个 `promise` 。

webpack 检测到这种 `import() 函数` 语法会自动代码分割。使用这种动态导入语法代替以前的静态引入，可以让组件在渲染的时候，再去加载组件对应的资源，

webpack 通过 `创建 script 标签` 来实现动态加载的，找出依赖对应的 `chunk` 信息，然后生成 script 标签来动态加载 chunk，每个 chunk 都有对应的状态： `未加载、加载中、已加载` 。

Suspense 组件

Suspense 内部主要通过捕获组件的状态去判断如何加载，React.lazy 创建的动态加载组件具有 `Pending、Resolved、Rejected` 三种状态，当这个组件的状态为 Pending 时显示的是 Suspense 中 fallback 的内容，只有状态变为 resolve 后才显示组件。

10. React VS Vue

- 组件化方式不同
  - React 组件包含状态和行为，所有组件共享一个状态树
  - Vue 每个组件都有自己的状态和行为，并且可以很容易将数据和行为绑定在一起
- 数据驱动方式不同
  - React 单项数据流
  - Vue 双向数据绑定
- 模板语法不同
  - React 模板语法是 JSX，all in js
  - Vue 模板语法是 Template、js、css，支持指令
- 生命周期不同
  - React 生命周期：初始化、更新、卸载
  - Vue 生命周期：创建、挂载、更新、销毁

- 状态管理方式不同
  - React 状态管理：Redux、Mobx、zustand
  - Vue 状态管理：Vuex、Pinia
- 性能优化方式不同
  - React 性能优化：React.memo、shouldComponentUpdate
  - Vue 性能优化：keep-alive、v-if

11. **React 组件通信**

父组件调子组件

- 如果是类组件，可以在子组件类中定义一个方法，并将其挂载到实例上
- 如果是类组件，可以使用 `createRef 创建一个 ref 对象`，并将其传递给子组件的 ref prop
- 如果是函数式组件，可以使用 `useImperativeHandle` Hook 将指定的方法暴露给父组件
- 如果是函数式组件，可以使用 `useRef` 创建一个 ref 对象，并将其传递给子组件的 ref prop

12. **React 中，Element、Component、Node、Instance 是四个重要的概念。**

- `Element`：Element 是 React 应用中最基本的构建块，它是一个普通的 JavaScript 对象，用来描述 UI 的一部分。Element 可以是原生的 DOM 元素，也可以是自定义的组件。它的作用是用来向 React 描述开发者想在页面上 render 什么内容。Element 是不可变的，一旦创建就不能被修改。
- `Component`：Component 是 React 中的一个概念，它是由 Element 构成的，可以是函数组件或者类组件。Component 可以接收输入的数据（props），并返回一个描述 UI 的 Element。Component 可以被复用，可以在应用中多次使用。分为 Class Component 以及 Function Component。
- `Node`：Node 是指 React 应用中的一个虚拟节点，它是 Element 的实例。Node 包含了 Element 的所有信息，包括类型、属性、子节点等。Node 是 React 内部用来描述 UI 的一种数据结构，它可以被渲染成真实的 DOM 元素。
- `Instance`：Instance 是指 React 应用中的一个组件实例，它是 Component 的实例。每个 Component 在应用中都会有一个对应的 Instance，它包含了 Component 的所有状态和方法。Instance 可以被用来操作组件的状态，以及处理用户的交互事件等。

13. **Redux**

核心描述：

- `单一数据源`：整个应用的全局 state 被存储在一棵 object tree 中，并且这个 object tree 只存在于唯一一个 store 中。
- `State 是只读的`：唯一改变 state 的方法就是触发 action，action 是一个用于描述已发生事情的普通对象。

- `使用纯函数来执行修改`：为了描述 action 如何改变 state tree，你需要编写纯的 reducers。

14. **React Hooks 实现生命周期?**

相对于传统class， Hooks 有哪些优势?

- State Hook 使得组件内的状态的设置和更新相对独立，这样便于对这些状态单独测试并复用。
- Hook 将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据），而并非强制按照生命周期划分，这样使得各个逻辑相对独立和清晰。

生命周期方法	Hooks 组件
constructor	useState
getDerivedStateFromProps	useEffect 手动对比 props， 配合 useState 里面 update 函数
shouldComponentUpdate	React.memo
render	函数本身
componentDidMount	useEffect 第二个参数为[]
componentDidUpdate	useEffect 配合useRef
componentWillUnmount	useEffect 里面返回的函数
componentDidCatch	无
getDerivedStateFromError	无

```
1 import React, { useState, useEffect, useRef, memo } from 'react';
2
3 // 使用 React.memo 实现类似 shouldComponentUpdate 的优化,
4 // React.memo 只对 props 进行浅比较
5 const UseEffectExample = memo((props) => {
```

```
    console.log("===== UseStateExample render=======");
    // 声明一个叫 “count” 的 state 变量。
    const [count, setCount] = useState(0);
    const [count2, setCount2] = useState(0);
    const [fatherCount, setFatherCount] = useState(props.fatherCount)

    console.log(props);

    // 模拟 getDerivedStateFromProps
    useEffect(() => {
        // props.fatherCount 有更新，才执行对应的修改，没有更新执行另外的逻辑
        if(props.fatherCount == fatherCount ){
            console.log("======= 模拟 getDerivedStateFromProps=======");
            console.log(props.fatherCount, fatherCount);
        }else{
            setFatherCount(props.fatherCount);
            console.log(props.fatherCount, fatherCount);
        }
    })

    // 模拟 componentDidMount
    useEffect(() => {
        console.log("=======只渲染一次(相当于DidMount)=======");
        console.log(count);
    }, [])

    // 模拟 componentDidUpdate
    const mounted = useRef();
    useEffect(() => {
        console.log(mounted);
        if (!mounted.current) {
            mounted.current = true;
          } else {
            console.log("======count 改变时才执行(相当于DidUpdate)=========");
            console.log(count);
          }
    }, [count])

    // 模拟 componentDidMount 和 componentDidUpdate、componentWillUnmount
    useEffect(() => {
        // 在 componentDidMount，以及 count 更改时 componentDidUpdate 执行的内容
        console.log("======初始化、或者 count 改变时才执行(相当于Didmount和
DidUpdate)=========");
        console.log(count);
        return () => {
            console.log("====unmount=======");
            console.log(count);
```

```
        }
    }, [count])

    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>

            <button onClick={() => setCount2(count2 + 1)}>
                Click me2
            </button>
        </div>
    );
});

export default UseEffectExample;
```

注意事项：

- `useState` 只在初始化时执行一次，后面不再执行；
- `useEffect` 相当于是 componentDidMount，componentDidUpdate 和 componentWillUnmount 这三个函数的组合，可以通过传参及其他逻辑，分别模拟这三个生命周期函数；
- `useEffect` 第二个参数是一个数组，
  - 如果数组为空时，则只执行一次（相当于 componentDidMount）；
  - 如果数组中有值时，则该值更新时，useEffect 中的函数才会执行；
  - 如果没有第二个参数，则每次 render 时，useEffect 中的函数都会执行；
- React 保证了每次运行 effect 的同时，DOM 都已经更新完毕，也就是说 effect 中的获取的 state 是最新的，但是需要注意的是，effect 中返回的函数（其清除函数）中，获取到的 state 是更新前的。
- 传递给 useEffect 的函数在每次渲染中都会有所不同，这是刻意为之的。事实上这正是我们可以在 effect 中获取最新的 count 的值，而不用担心其过期的原因。每次我们重新渲染，都会生成新的 effect，替换掉之前的。某种意义上讲，effect 更像是渲染结果的一部分 —— 每个 effect “属于”一次特定的渲染。
- effect 的清除阶段（返回函数）在每次重新渲染时都会执行，而不是只在卸载组件的时候执行一次。它会在调用一个新的 effect 之前对前一个 effect 进行清理，从而避免了我们手动去处理一些逻辑。

15. **React.memo() 和 useMemo()**

`React.memo()` 随 React v16.6 一起发布。

虽然类组件已经允许您使用 `PureComponent` 或 `shouldComponentUpdate` 来控制重新渲染，但 React 16.6 引入了对函数组件执行相同操作的能力。

React.memo() 是一个 `高阶组件 （HOC）`，它接收一个组件 A 作为参数并返回一个组件 B，如果组件 B 的 props（或其中的值）没有改变，则组件 B 会阻止组件 A 重新渲染 。

useMemo() 是一个 React Hook。

- 可以依赖 useMemo() 作为性能优化，而不是语义保证
- 函数内部引用的每个值也应该出现在依赖项数组中

React.memo() 和 useMemo() 之间的主要区别：

- React.memo() 是一个高阶组件，可以使用它来包装不想重新渲染的组件，除非其中的 props 发生变化
- useMemo() 是一个 React Hook，可以使用它在组件中包装函数。 可以使用它来确保该函数中的值仅在其依赖项之一发生变化时才重新计算。

虽然 memoization 似乎是一个可以随处使用的巧妙小技巧，但只有在绝对需要这些性能提升时才应该使用它。 Memoization 会占用运行它的机器上的内存空间，因此可能会导致意想不到的效果。

16. 实现 useTimeout hook

useTimeout 是可以在函数式组件中，处理 setTimeout 计时器函数

```
// callback 回调函数， delay 延迟时间
function useTimeout(callback, delay) {
  const memorizeCallback = useRef();

  useEffect(() => {
    memorizeCallback.current = callback;
  }, [callback]);

  useEffect(() => {
    if (delay !== null) {
      const timer = setTimeout(() => {
        memorizeCallback.current();
      }, delay);
      return () => {
        clearTimeout(timer);
      };
    }
  }, [delay]);
};
```

17. 对 useReducer 的理解

useReducer 是 React Hooks 中的一个函数，用于管理和更新组件的状态。它可以被视为 useState 的一种替代方案，适用于处理更复杂的状态逻辑。

```jsx
import { useReducer } from 'react';

const initialState = {
  count: 0,
};

const reducer = (state, action) => {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error('Unsupported action type');
  }
};

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);

  const increment = () => {
    dispatch({ type: 'increment' });
  };

  const decrement = () => {
    dispatch({ type: 'decrement' });
  };

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}
```

相比于 useState，useReducer 在处理复杂状态逻辑时更有优势，因为它允许我们将状态更新的逻辑封装在 reducer 函数中，并根据不同的动作类型执行相应的逻辑。这样可以使代码更具可读性和可维护性，并且更容易进行状态追踪和调试。

18. **React.memo() VS JS 的 memorize 函数**

19. `适用范围不同`：

- React.memo() 主要适用于优化 React 组件的性能表现，
- 而 memorize 函数可以用于任何 JavaScript 函数的结果缓存。

1. `实现方式不同`：

- React.memo() 是一个 React 高阶组件（HOC），通过浅层比较 props 是否发生变化来决定是否重新渲染组件。- 而 memorize 函数则是通过将函数的输入参数及其计算结果保存到一个缓存对象中，以避免重复计算相同的结果。

1. `缓存策略不同`：

- React.memo() 的缓存策略是浅比较（shallow compare），只比较 props 的第一层属性值是否相等，不会递归比较深层嵌套对象或数组的内容。
- 而 memorize 函数的缓存策略是将输入参数转换成字符串后，作为缓存的键值。如果传入的参数不是基本类型时，则需要自己实现缓存键值的计算。
- `应用场景不同`：
  - `React.memo() 主要适用于对不经常变化的组件进行性能优化，
    - 对于状态不变的组件或纯函数，可以使用 React.memo() 进行优化；
  - 而 memorize 函数则主要适用于对计算量大、执行时间长的函数进行结果缓存。
    - 对于递归计算、复杂数学运算等耗时操作，可以使用 memorize 函数进行结果缓存。

2. **类组件 VS React Hooks**

- `跨组件复用`：
  - 其实 render props / HOC 也是为了复用，相比于它们，Hooks 作为官方的底层 API，最为轻量，而且改造成本小，不会影响原来的组件层次结构和传说中的嵌套地狱；
- 相比而言，`类组件的实现更为复杂`
  - 不同的生命周期会使逻辑变得分散且混乱，不易维护和管理；
  - 时刻需要关注 this 的指向问题；
  - 代码复用代价高，高阶组件的使用经常会使整个组件树变得臃肿；
- `状态与 UI 隔离`：
  - 正是由于 Hooks 的特性，状态逻辑会变成更小的粒度，并且极容易被抽象成一个自定义 Hooks，组件中的状态和 UI 变得更为清晰和隔离。

注意：

- 避免在 循环/条件判断/嵌套函数 中调用 hooks，保证调用顺序的稳定；
- 不能在 useEffect 中使用 useState，React 会报错提示；

- 类组件不会被替换或废弃，不需要强制改造类组件，两种方式能并存

3. **setState 是同步还是异步?**

react 18 之前:

- 在Promise的状态更新、js原生事件、setTimeout、setInterval..中是同步的。
- 在react的合成事件中，是异步的。

> setState的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形式了所谓的“异步”，当然可以通过第二个参数 setState(partialState, callback) 中的callback拿到更新后的结果。

react 18 之后:

- setState都会表现为异步（即批处理）。

4. **React-Router 的 `<Link />` 组件和 `<a>` 有什么区别?**

Link 的 “跳转” 行为只会触发相匹配的对应的页面内容更新，而不会刷新整个页面。

Link 跳转做了三件事情:

- 有onclick那就执行onclick
- click的时候阻止a标签默认事件
- 根据跳转 href，用 history 跳转，此时只是链接变了，并没有刷新页面

而 a 标签就是普通的超链接了，用于从当前页面跳转到href指向的另一个页面（非锚点情况）。

5. **PureComponent 和 Component的区别是?**

- Component 需要手动实现 shouldComponentUpdate，
- PureComponent 通过浅对比默认实现了 shouldComponentUpdate 方法。

> 注意： PureComponent 不仅会影响本身，而且会影响子组件，所以 PureComponent 最佳情况是展示组件。

6. **React 事件和原生事件的执行顺序**

为什么要有合成事件?

- 在传统的事件里，不同的浏览器需要兼容不同的写法，在合成事件中React提供统一的事件对象，抹平了浏览器的兼容性差异
- React通过顶层监听的形式，通过事件委托的方式来统一管理所有的事件，可以在事件上区分事件优先级，优化用户体验

事件委托

- 事件委托的意思就是可以通过给父元素绑定事件委托，通过事件对象的target属性可以获取到当前触发目标阶段的dom元素，来进行统一管理
- 比如写原生dom循环渲染的时候，我们要给每一个子元素都添加dom事件，这种情况最简单的方式就是通过事件委托在父元素做一次委托，通过target属性判断区分做不同的操作

事件监听

事件监听主要用到了addEventListener这个函数，事件监听和事件绑定的最大区别就是事件监听可以给一个事件监听多个函数操作，而事件绑定只有一次

```
// 可以监听多个，不会被覆盖
eventTarget.addEventListener('click', () => {});
eventTarget.addEventListener('click', () => {});

eventTarget.onclick = function () {};
eventTarget.onclick = function () {}; // 第二个会把第一个覆盖
```

- 16版本先执行原生事件，当冒泡到document时，统一执行合成事件，
- 17版本在原生事件执行前先执行合成事件捕获阶段，原生事件执行完毕执行冒泡阶段的合成事件，通过根节点来管理所有的事件

原生的阻止事件流会阻断合成事件的执行，合成事件阻止后也会影响到后续的原生执行

7. **Redux VS Vuex**

相同点

- state 共享数据
- 流程一致：定义全局state，触发，修改state
- 原理相似，通过全局注入store。

不同点

- 从实现原理上来说：
  - Redux 使用的是不可变数据，
  - 而Vuex的数据是可变的。
  - Redux每次都是用新的state替换旧的state，
  - 而Vuex是直接修改
  - Redux 在检测数据变化的时候，是通过 diff 的方式比较差异的，
  - 而Vuex其实和Vue的原理一样，是通过 getter/setter来比较的
- 从表现层来说：

    - vuex定义了state、getter、mutation、action四个对象；
    - redux定义了state、reducer、action。
    - vuex中state统一存放，方便理解；
    - redux中state依赖所有reducer的初始值
    - vuex有getter,目的是快捷得到state；
    - redux没有这层，react-redux mapStateToProps参数做了这个工作。
    - vuex中mutation只是单纯赋值(很浅的一层)；
    - redux中reducer只是单纯设置新state(很浅的一层)。
    - vuex中action有较为复杂的异步ajax请求；
    - redux中action中可简单可复杂,简单就直接发送数据对象（{type:xxx, your-data}）,复杂需要调用异步ajax（依赖redux-thunk插件）。
    - vuex触发方式有两种commit同步和dispatch异步；
    - redux同步和异步都使用dispatch
- vuex 弱化 dispatch，通过commit进行 store状态的一次更变；
- 取消了action概念，不必传入特定的 action形式进行指定变更；
- 弱化reducer，基于commit参数直接对数据进行转变，使得框架更加简易；

共同思想

- 单一的数据源
- 变化可以预测
- 本质上： redux与vuex都是对mvvm思想的服务，将数据从视图中抽离的一种方案。

8. **Mobx VS Redux**

- redux将数据保存在单一的store中，
    - mobx将数据保存在分散的多个store中
- redux使用plain object保存数据，需要手动处理变化后的操作;
    - mobx适用observable保存数据，数据变化后自动处理响应的操作
- redux使用不可变状态，这意味着状态是只读的，不能直接去修改它，而是应该返回一个新的状态，同时使用纯函数;
    - mobx中的状态是可变的，可以直接对其进行修改
- mobx相对来说比较简单，在其中有很多的抽象，mobx更多的使用面向对象的编程思维;
    - redux会比较复杂，因为其中的函数式编程思想掌握起来不是那么容易，同时需要借助一系列的中间件来处理异步和副作用

- mobx中有更多的抽象和封装，调试会比较困难，同时结果也难以预测;
  - 而redux提供能够进行时间回溯的开发工具，同时其纯函数以及更少的抽象，让调试变得更加的容易

9. **React 中，父子组件的生命周期执行顺序**

观察父子组件的挂载生命周期函数，可以发现：

- 挂载时，子组件的挂载钩子先被触发；
- 卸载时，子组件的卸载钩子后被触发。

我们经常在挂载函数上注册监听器，说明此时是可以与页面交互的，也就是说其实所有挂载钩子都是在父组件实际挂载到dom树上才触发的，不过是在父组件挂载后依次触发子组件的 componentDidmount ，最后再触发自身的挂载钩子。

相反，卸载的时候父节点先被移除，再从上至下依次触发子组件的卸载钩子；

但是我们也经常在卸载钩子上卸载监听器，这说明 componentWillUnmount 其实在父组件从dom树上卸载前触发的，先触发自身的卸载钩子，但此时并未从dom树上剥离，然后依次尝试触发所有子组件的卸载钩子，最后，父组件从dom树上完成实际卸载。

10. **React render 方法原理？在什么时候触发？**

render函数里面可以编写JSX，转化成createElement这种形式，用于生成虚拟DOM，最终转化成真实DOM

在 React 中，类组件只要执行了 setState 方法，就一定会触发 render 函数执行，函数组件使用 useState更改状态不一定导致重新render

组件的 props 改变了，不一定触发 render 函数的执行，

但是如果 props 的值来自于父组件或者祖先组件的 state，在这种情况下，父组件或者祖先组件的 state 发生了改变，就会导致子组件的重新渲染

所以，一旦执行了setState就会执行render方法，useState 会判断当前值有无发生改变确定是否执行render方法，一旦父组件发生渲染，子组件也会渲染

11. **React-router 几种模式，以及实现原理?**

主要分成了两种模式：

- `hash 模式`：在url后面加上#，如http://127.0.0.1:5500/home/#/page1
- `history 模式`：允许操作浏览器的曾经在标签页或者框架里访问的会话历史记录

hash 原理

hash 值改变，触发全局 window 对象上的 `hashchange` 事件。所以 hash 模式路由就是利用 hashchange 事件监听 URL 的变化，从而进行 DOM 操作来模拟页面跳转

通过 `window.addEventListener('hashChange',callback)` 监听hash值的变化，并传递给其嵌套的组件

12. **React JSX 转换成真实 DOM 的过程?**

- 使用 `React.createElement` 或JSX编写React组件，实际上所有的 JSX 代码最后都会转换成 React.createElement(...)， `Babel` 帮助我们完成了这个转换的过程。
- createElement函数对key和ref等特殊的props进行处理，并获取defaultProps对默认props进行赋值，并且对传入的孩子节点进行处理，最终构造成一个虚拟DOM对象
- `ReactDOM.render` 将生成好的虚拟DOM渲染到指定容器上，其中采用了 `批处理、事务等机制` 并且对特定浏览器进行了性能优化，最终转换为真实DOM

13. **React 服务端渲染（SSR）原理?**

14. node server 接收客户端请求，得到当前的请求 url 路径，然后在已有的路由表内查找到对应的组件，拿到需要请求的数据，将数据作为 props、context或者store 形式传入组件

15. 然后基于 react 内置的服务端渲染方法 renderToString() 把组件渲染为 html 字符串在把最终的 html 进行输出前需要将数据注入到浏览器端

16. 浏览器开始进行渲染和节点对比，然后执行完成组件内事件绑定和一些交互，浏览器重用了服务端输出的 html 节点，整个流程结束

17. **常用的 React Hooks**

- `状态钩子 （useState）`: 用于定义组件的 State，类似类定义中 this.state 的功能
- `useReducer`： 用于管理复杂状态逻辑的替代方案，类似于 Redux 的 reducer。
- `生命周期钩子 （useEffect）`: 类定义中有许多生命周期函数，而在 React Hooks 中也提供了一个相应的函数 (useEffect)，这里可以看做componentDidMount、componentDidUpdate和componentWillUnmount的结合。
- `useLayoutEffect`： 与 useEffect 类似，但在浏览器完成绘制之前同步执行。
- `useContext`: 获取 context 对象，用于在组件树中获取和使用共享的上下文。
- `useCallback`: 缓存回调函数，避免传入的回调每次都是新的函数实例而导致依赖组件重新渲染，具有性能优化的效果；
- `useMemo`: 用于缓存传入的 props，避免依赖的组件每次都重新渲染；
- `useRef`: 获取组件的真实节点；用于在函数组件之间保存可变的值，并且不会引发重新渲染。
- `useImperativeHandle`： 用于自定义暴露给父组件的实例值或方法。
- `useDebugValue`： 用于在开发者工具中显示自定义的钩子相关标签。

18. **useEffect VS useLayoutEffect**

- `使用场景`：
  - `useEffect` 在 React 的渲染过程中是被异步调用的，用于绝大多数场景；

    - `useLayoutEffect` 会在所有的 DOM 变更之后同步调用，主要用于处理 DOM 操作、调整样式、避免页面闪烁等问题。
    - 也正因为是同步处理，所以需要避免在 useLayoutEffect 做计算量较大的耗时任务从而造成阻塞。
- `使用效果`：
    - `useEffect` 是按照顺序执行代码的，改变屏幕像素之后执行（先渲染，后改变DOM），当改变屏幕内容时可能会产生闪烁；
    - `useLayoutEffect` 是改变屏幕像素之前就执行了（会推迟页面显示的事件，先改变DOM后渲染），不会产生闪烁。useLayoutEffect总是比useEffect先执行。

在未来的趋势上，两个 API 是会长期共存的，暂时没有删减合并的计划，需要开发者根据场景去自行选择。React 团队的建议非常实用，如果实在分不清，先用 useEffect，一般问题不大；如果页面有异常，再直接替换为 useLayoutEffect 即可。