前端react面试题指北

在调用setState 之后发生了什么

- 状态合并，触发调和：

  setState函数之后，会将传入的参数对象与当前的状态合并，然后出发调用过程

- 根据新的状态构建虚拟dom树

  经过调和过程，react会高效的根据新的状态构建虚拟DOM树，准备渲染整个UI页面

- 计算新老树节点差异，最小化渲染

  得倒新的虚拟DOM树后，会计算出新老树的节点差异，会根据差异对界面进行最小化渲染

- 按需更新

  在差异话计算中，react可以相对准确的知道哪些位置发生了改变以及该如何改变，这保证按需更新，而不是宣布重新渲染

展示组件(Presentational component)和容器组件(Container component)之间有何不同

展示组件关心组件看起来是什么。展示专门通过 props 接受数据和回调，并且几乎不会有自身的状态，但当展示组件拥有自身的状态时，通常也只关心 UI 状态而不是数据的状态。

容器组件则更关心组件是如何运作的。容器组件会为展示组件或者其它容器组件提供数据和行为(behavior)，它们会调用 `Flux actions` ，并将其作为回调提供给展示组件。容器组件经常是有状态的，因为它们是(其它组件的)数据源。

可以使用TypeScript写React应用吗？怎么操作?

（1）如果还未创建 Create React App 项目

- 直接创建一个具有 typescript 的 Create React App 项目：

```javascript 复制代码
npx create-react-app demo --typescript
```

（2）如果已经创建了 Create React App 项目，需要将 typescript 引入到已有项目中

- 通过命令将 typescript 引入项目：

```javascript 复制代码
npm install --save typescript @types/node @types/react @types/react-dom @types/jest
```

- 将项目中任何 后缀名为 '.js' 的 JavaScript 文件重命名为 TypeScript 文件即后缀名为 '.tsx'（例如 src/index.js 重命名为 src/index.tsx ）

HOC相比 mixins 有什么优点?

HOC 和 Vue 中的 mixins 作用是一致的，并且在早期 React 也是使用 mixins 的方式。但是在使用 class 的方式创建组件以后，mixins 的方式就不能使用了，并且其实 mixins 也是存在一些问题的，比如：

- 隐含了一些依赖，比如我在组件中写了某个 `state` 并且在 `mixin` 中使用了，就这存在了一个依赖关系。万一下次别人要移除它，就得去 `mixin` 中查找依赖
- 多个 `mixin` 中可能存在相同命名的函数，同时代码组件中也不能出现相同命名的函数，否则就是重写了，其实我一直觉得命名真的是一件麻烦事。。
- 雪球效应，虽然我一个组件还是使用着同一个 `mixin` ，但是一个 `mixin` 会被多个组件使用，可能会存在需求使得 `mixin` 修改原本的函数或者新增更多的函数，这样可能就会产生一个维护成本

HOC 解决了这些问题，并且它们达成的效果也是一致的，同时也更加的政治正确（毕竟更加函数式了）。

react 的优化

shouldcomponentUpdate pureCompoment setState

- CPU的瓶颈（当有大量渲染任务的时候，js线程和渲染线程互斥）

- IO的瓶颈 就是网络（如何在网络延迟客观存在的 情况下，减少用户对网络延 迟的感知）（Code Splitting • Data Fetching）比如react.lazy（组件懒加载） suspense(分包在网络上，用的时候在获取)
- Virtual DOM 快么（Virtual DOM的优势不在于单次的操作，而是在大量、频繁的数据更新下，能够对视图 进行合理、高效的更新。）

展示组件(Presentational component)和容器组件(Container component)之间有何不同?

React.Component 和 React.PureComponent 的区别

PureComponent表示一个纯组件，可以用来优化React程序，减少render函数执行的次数，从而提高组件的性能。

在React中，当prop或者state发生变化时，可以通过在shouldComponentUpdate生命周期函数中执行return false来阻止页面的更新，从而减少不必要的render执行。
React.PureComponent会自动执行 shouldComponentUpdate。

不过，pureComponent中的 shouldComponentUpdate() 进行的是**浅比较**，也就是说如果是引用数据类型的数据，只会比较不是同一个地址，而不会比较这个地址里面的数据是否一致。浅比较会忽略属性和或状态突变情况，其实也就是数据引用指针没有变化，而数据发生改变的时候render是不会执行的。如果需要重新渲染那么就需要重新开辟空间引用数据。
PureComponent一般会用在一些纯展示组件上。

使用pureComponent的**好处**：当组件更新时，如果组件的props或者state都没有改变，render函数就不会触发。省去虚拟DOM的生成和对比过程，达到提升性能的目的。这是因为react自动做了一层浅比较。

参考 前端进阶面试题详细解答

区分状态和 props

条件	State	Props
1. 从父组件中接收初始值	Yes	Yes
2. 父组件可以改变值	No	Yes
3. 在组件中设置默认值	Yes	Yes
4. 在组件的内部变化	Yes	No
5. 设置子组件的初始值	Yes	Yes
6. 在子组件的内部更改	No	Yes

React 的工作原理

React 会创建一个虚拟 DOM(virtual DOM)。当一个组件中的状态改变时，React 首先会通过 "diffing" 算法来标记虚拟 DOM 中的改变，第二步是调节(reconciliation)，会用 diff 的结果来更新 DOM。

什么是 Props

Props 是 React 中属性的简写。它们是只读组件，必须保持纯，即不可变。它们总是在整个应用中从父组件传递到子组件。子组件永远不能将 prop 送回父组件。这有助于维护单向数据流，通常用于呈现动态生成的数据。

React-Router 4的Switch有什么用?

Switch 通常被用来包裹 Route，用于渲染与路径匹配的第一个子 `<Route>` 或 `<Redirect>`，它里面不能放其他元素。

假如不加 `<Switch>` ：

```javascript 复制代码
import { Route } from 'react-router-dom'

<Route path="/" component={Home}></Route>
```

```
<Route path="/login" component={Login}></Route>
```

Route 组件的 path 属性用于匹配路径，因为需要匹配 `/` 到 `Home`，匹配 `/login` 到 `Login`，所以需要两个 Route，但是不能这么写。这样写的话，当 URL 的 path 为 “/login” 时，`<Route path="/" />` 和 `<Route path="/login" />` 都会被匹配，因此页面会展示 Home 和 Login 两个组件。这时就需要借助 `<Switch>` 来做到只显示一个匹配组件：

javascript 复制代码

```
import { Switch, Route} from 'react-router-dom'

<Switch>
    <Route path="/" component={Home}></Route>
    <Route path="/login" component={Login}></Route>
</Switch>
```

此时，再访问 “/login” 路径时，却只显示了 Home 组件。这是就用到了exact属性，它的作用就是精确匹配路径，经常与 `<Switch>` 联合使用。只有当 URL 和该 `<Route>` 的 path 属性完全一致的情况下才能匹配上：

javascript 复制代码

```
import { Switch, Route} from 'react-router-dom'

<Switch>
   <Route exact path="/" component={Home}></Route>
   <Route exact path="/login" component={Login}></Route>
</Switch>
```

constructor

text 复制代码

```
答案是：在 constructor 函数里面，需要用到props的值的时候，就需要调用 super(props)
```

1. class语法糖默认会帮你定义一个constructor，所以当你不需要使用constructor的时候，是可以不用自己定义的
2. 当你自己定义一个constructor的时候，就一定要写super()，否则拿不到this
3. 当你在constructor里面想要使用props的值，就需要传入props这个参数给super，调用super(props)，否则只需要写super()

Redux 和 Vuex 有什么区别，它们的共同思想

（1）Redux 和 Vuex区别

- Vuex改进了Redux中的Action和Reducer函数，以mutations变化函数取代Reducer，无需switch，只需在对应的mutation函数里改变state值即可
- Vuex由于Vue自动重新渲染的特性，无需订阅重新渲染函数，只要生成新的State即可
- Vuex数据流的顺序是：View调用store.commit提交对应的请求到Store中对应的mutation函数->store改变（vue检测到数据变化自动渲染）

通俗点理解就是，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;

（2）共同思想

- 单—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。

React如何获取组件对应的DOM元素?

可以用ref来获取某个子节点的实例，然后通过当前class组件实例的一些特定属性来直接获取子节点实例。ref有三种实现方法:

- **字符串格式**：字符串格式，这是React16版本之前用得最多的，例如：`<p ref="info">span</p>`
- **函数格式**：ref对应一个方法，该方法有一个参数，也就是对应的节点实例，例如：`<p ref={ele => this.info = ele}></p>`
- **createRef方法**：React 16提供的一个API，使用React.createRef()来实现

react 的渲染过程中，兄弟节点之间是怎么处理的？也就是key值不一样的时候

> 通常我们输出节点的时候都是map一个数组然后返回一个 `ReactNode` ，为了方便 `react` 内部进行优化，我们必须给每一个 `reactNode` 添加 `key` ，这个 `key prop` 在设计值处不是给开发者用的，而是给react用的，大概的作用就是给每一个 `reactNode` 添加一个身份标识，方便react进行识别，在重渲染过程中，如果key一样，若组件属性有所变化，则 `react` 只更新组件对应的属性；没有变化则不更新，如果key不一样，则react先销毁该组件，然后重新创建该组件

简述react事件机制

当用户在为onClick添加函数时，React并没有将Click时间绑定在DOM上面
而是在document处监听所有支持的事件，当事件发生并冒泡至document处时，React将事件内容封装交给中间层SyntheticEvent（负责所有事件合成）
所以当事件触发的时候，对使用统一的分发函数dispatchEvent将指定函数执行。React在自己的合成事件中重写了 stopPropagation方法，将 isPropagationStopped设置为 true，然后在遍历每一级事件的过程中根据此遍历判断是否继续执行。这就是 React自己实现的冒泡机制

概述下 React 中的事件处理逻辑

- 抹平浏览器差异，实现更好的跨平台。
- 避免垃圾回收，React 引入事件池，在事件池中获取或释放事件对象，避免频繁地去创建和销毁。
- 方便事件统一管理和事务机制。

> 为了解决跨浏览器兼容性问题， `React` 会将浏览器原生事件（ `Browser Native Event` ）封装为合成事件（ `SyntheticEvent` ）传入设置的事件处理器中。这里的合成事件提供了与原生事件相同的接口，不过它们屏蔽了底层浏览器的细节差异，保证了行为的一致性。另外有意思的是， `React` 并没有直接将事件附着到子元素上，而是以单一事件监听器的方式将所有的事件发送到顶层进行处理。这样 `React` 在更新 `DOM` 的时候就不需要考虑如何去处理附着在 `DOM` 上的事件监听器，最终达到优化性能的目的

mobox 和 redux 有什么区别?

（1）共同点

- 为了解决状态管理混乱，无法有效同步的问题统一维护管理应用状态;

- 某一状态只有一个可信数据来源（通常命名为store，指状态容器）；
- 操作更新状态方式统一，并且可控（通常以action方式提供更新状态的途径）；
- 支持将store与React组件连接，如react-redux，mobx- react;

（2）区别 Redux更多的是遵循Flux模式的一种实现，是一个 JavaScript库，它关注点主要是以下几方面：

- Action： 一个JavaScript对象，描述动作相关信息，主要包含type属性和payload属性：

```
o type： action 类型; o payload： 负载数据;
```

复制代码

- Reducer： 定义应用状态如何响应不同动作（action），如何更新状态;

- Store： 管理action和reducer及其关系的对象，主要提供以下功能：

```javascript
o 维护应用状态并支持访问状态(getState());
o 支持监听action的分发，更新状态(dispatch(action));
o 支持订阅store的变更(subscribe(listener));
```

javascript 复制代码

- 异步流： 由于Redux所有对store状态的变更，都应该通过action触发，异步任务（通常都是业务或获取数据任务）也不例外，而为了不将业务或数据相关的任务混入React组件中，就需要使用其他框架配合管理异步任务流程，如redux-thunk，redux-saga等;

Mobx是一个透明函数响应式编程的状态管理库，它使得状态管理简单可伸缩：

- Action：定义改变状态的动作函数，包括如何变更状态;
- Store： 集中管理模块状态（State）和动作(action)
- Derivation（衍生）： 从应用状态中派生而出，且没有任何其他影响的数据

对比总结:

- 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

哪些方法会触发 React 重新渲染？重新渲染 render 会做些什么？

（1）哪些方法会触发 react 重新渲染?

- **setState（）方法被调用**

setState 是 React 中最常用的命令，通常情况下，执行 setState 会触发 render。但是这里有个点值得关注，执行 setState 的时候不一定会重新渲染。当 setState 传入 null 时，并不会触发 render。

javascript 复制代码

```
class App extends React.Component {
  state = {
    a: 1
  };

  render() {
    console.log("render");
    return (
      <React.Fragement>
        <p>{this.state.a}</p>
        <button
          onClick={() => {            this.setState({ a: 1 }); // 这里并没有改变 a 的值          }}
        <button onClick={() => this.setState(null)}>setState null</button>
        <Child />
      </React.Fragement>
    );
  }
}
```

- **父组件重新渲染**

只要父组件重新渲染了，即使传入子组件的 props 未发生变化，那么子组件也会重新渲染，进而触发 render

（2）重新渲染 render 会做些什么?

- 会对新旧 VNode 进行对比，也就是我们所说的Diff算法。
- 对新旧两棵树进行一个深度优先遍历，这样每一个节点都会一个标记，在到深度遍历的时候，每遍历到一和个节点，就把该节点和新的节点树进行对比，如果有差异就放到一个对象里面
- 遍历差异对象，根据差异的类型，根据对应对规则更新VNode

React 的处理 render 的基本思维模式是每次一有变动就会去重新渲染整个应用。在 Virtual DOM 没有出现之前，最简单的方法就是直接调用 innerHTML。Virtual DOM厉害的地方并不是说它比直接操作 DOM 快，而是说不管数据怎么变，都会尽量以最小的代价去更新 DOM。React 将 render 函数返回的虚拟 DOM 树与老的进行比较，从而确定 DOM 要不要更新、怎么更新。当 DOM 树很大时，遍历两棵树进行各种比对还是相当耗性能的，特别是在顶层 setState 一个微小的修改，默认会去遍历整棵树。尽管 React 使用高度优化的 Diff 算法，但是这个过程仍然会损耗性能.

在哪个生命周期中你会发出Ajax请求？为什么?

Ajax请求应该写在组件创建期的第五个阶段，即 componentDidMount生命周期方法中。原因如下。 在创建期的其他阶段，组件尚未渲染完成。而在存在期的5个阶段，又不能确保生命周期方法一定会执行（如通过 shouldComponentUpdate方法优化更新等）。在销毁期，组件即将被销毁，请求数据变得无意义。因此在这些阶段发出Ajax请求显然不是最好的选择。 在组件尚未挂载之前，Ajax请求将无法执行完毕，如果此时发出请求，将意味着在组件挂载之前更新状态（如执行 setState），这通常是不起作用的。 在 componentDidMount方法中，执行Ajax即可保证组件已经挂载，并且能够正常更新组件。

React中的setState批量更新的过程是什么?

调用 `setState` 时，组件的 `state` 并不会立即改变， `setState` 只是把要修改的 `state` 放入一个队列， `React` 会优化真正的执行时机，并出于性能原因，会将 `React` 事件处理程序中的多次 `React` 事件处理程序中的多次 `setState` 的状态修改合并成一次状态修改。 最终更新只产生一次组件及其子组件的重新渲染，这对于大型应用程序中的性能提升至关重要。

```javascript 复制代码
this.setState({
  count: this.state.count + 1    ===>    入队，[count+1的任务]
});
this.setState({
  count: this.state.count + 1    ===>    入队，[count+1的任务，count+1的任务]
```

```
});
                                                    ↓
                                    合并 state，[count+1的任务]
                                                    ↓
                                    执行 count+1的任务
```

需要注意的是，只要同步代码还在执行，“攒起来”这个动作就不会停止。（注：这里之所以多次 +1 最终只有一次生效，是因为在同一个方法中多次 setState 的合并动作不是单纯地将更新累加。比如这里对于相同属性的设置，React 只会为其保留最后一次的更新）。