web前端经典react面试题

redux有什么缺点

- 一个组件所需要的数据，必须由父组件传过来，而不能像 `flux` 中直接从 `store` 取。
- 当一个组件相关数据更新时，即使父组件不需要用到这个组件，父组件还是会重新 `render` ，可能会有效率影响，或者需要写复杂的 `shouldComponentUpdate` 进行判断。

React-Router的路由有几种模式?

React-Router 支持使用 hash（对应 HashRouter）和 browser（对应 BrowserRouter） 两种路由规则， react-router-dom 提供了 BrowserRouter 和 HashRouter 两个组件来实现应用的 UI 和 URL 同步：

- BrowserRouter 创建的 URL 格式：xxx.com/path
- HashRouter 创建的 URL 格式：xxx.com/#/path

（1）BrowserRouter

它使用 HTML5 提供的 history API（pushState、replaceState 和 popstate 事件）来保持 UI 和 URL 的同步。由此可以看出，**BrowserRouter 是使用 HTML 5 的 history API 来控制路由跳转的：**

```javascript 复制代码
<BrowserRouter
    basename={string}
    forceRefresh={bool}
    getUserConfirmation={func}
    keyLength={number}
/>
```

其中的属性如下：

- basename 所有路由的基准 URL。basename 的正确格式是前面有一个前导斜杠，但不能有尾部斜杠；

```javascript 复制代码
<BrowserRouter basename="/calendar">
    <Link to="/today" />
</BrowserRouter>
```

等同于

```javascript 复制代码
<a href="/calendar/today" />
```

- forceRefresh 如果为 true，在导航的过程中整个页面将会刷新。一般情况下，只有在不支持 HTML5 history API 的浏览器中使用此功能；
- getUserConfirmation 用于确认导航的函数，默认使用 window.confirm。例如，当从 /a 导航至 /b 时，会使用默认的 confirm 函数弹出一个提示，用户点击确定后才进行导航，否则不做任何处理；

```javascript 复制代码
// 这是默认的确认函数
const getConfirmation = (message, callback) => {
  const allowTransition = window.confirm(message);
  callback(allowTransition);
}
<BrowserRouter getUserConfirmation={getConfirmation} />
```

> 需要配合 `<Prompt>` 一起使用。

- KeyLength 用来设置 Location.Key 的长度。

(2) HashRouter

使用 URL 的 hash 部分（即 window.location.hash）来保持 UI 和 URL 的同步。由此可以看出，**HashRouter 是通过 URL 的 hash 属性来控制路由跳转的：**

```javascript 复制代码
<HashRouter
    basename={string}
    getUserConfirmation={func}
    hashType={string}
/>
```

其参数如下：

- basename, getUserConfirmation 和 `BrowserRouter` 功能一样；
- hashType window.location.hash 使用的 hash 类型，有如下几种：
  - slash - 后面跟一个斜杠，例如 #/ 和 #/sunshine/lollipops；
  - noslash - 后面没有斜杠，例如 # 和 #sunshine/lollipops；
  - hashbang - Google 风格的 ajax crawlable，例如 #!/ 和 #!/sunshine/lollipops。

对于store的理解

Store 就是把它们联系到一起的对象。Store 有以下职责：

- 维持应用的 state；
- 提供 getState() 方法获取 state；
- 提供 dispatch(action) 方法更新 state；
- 通过 subscribe(listener)注册监听器;
- 通过 subscribe(listener)返回的函数注销监听器

React中的状态是什么？它是如何使用的

状态是 React 组件的核心，是数据的来源，必须尽可能简单。基本上状态是确定组件呈现和行为的对象。与props 不同，它们是可变的，并创建动态和交互式组件。可以通过 `this.state()` 访问它们。

React声明组件有哪几种方法，有什么不同?

React 声明组件的三种方式：

- 函数式定义的 无状态组件
- ES5原生方式 `React.createClass` 定义的组件
- ES6形式的 `extends React.Component` 定义的组件

（1）无状态函数式组件 它是为了创建纯展示组件，这种组件只负责根据传入的props来展示，不涉及到state状态的操作 组件不会被实例化，整体渲染性能得到提升，不能访问this对象，不能访问生命周期的方法

（2）ES5 原生方式 React.createClass // RFC React.createClass会自绑定函数方法，导致不必要的性能开销，增加代码过时的可能性。

（3）E6继承形式 React.Component // RCC 目前极为推荐的创建有状态组件的方式，最终会取代React.createClass形式；相对于 React.createClass可以更好实现代码复用。

无状态组件相对于于后者的区别： 与无状态组件相比，React.createClass和React.Component都是创建有状态的组件，这些组件是要被实例化的，并且可以访问组件的生命周期方法。

React.createClass与React.Component区别：

① 函数this自绑定

- React.createClass创建的组件，其每一个成员函数的this都有React自动绑定，函数中的this会被正确设置。
- React.Component创建的组件，其成员函数不会自动绑定this，需要开发者手动绑定，否则this不能获取当前组件实例对象。

② 组件属性类型propTypes及其默认props属性defaultProps配置不同

- React.createClass在创建组件时，有关组件props的属性类型及组件默认的属性会作为组件实例的属性来配置，其中defaultProps是使用getDefaultProps的方法来获取默认组件属性的
- React.Component在创建组件时配置这两个对应信息时，他们是作为组件类的属性，不是组件实例的属性，也就是所谓的类的静态属性来配置的。

③ 组件初始状态state的配置不同

- React.createClass创建的组件，其状态state是通过getInitialState方法来配置组件相关的状态；
- React.Component创建的组件，其状态state是在constructor中像初始化组件属性一样声明的。

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

React Hook 的使用限制有哪些?

React Hooks 的限制主要有两条：

- 不要在循环、条件或嵌套函数中调用 Hook；
- 在 React 的函数组件中调用 Hook。

那为什么会有这样的限制呢？Hooks 的设计初衷是为了改进 React 组件的开发模式。在旧有的开发模式下遇到了三个问题。

- 组件之间难以复用状态逻辑。过去常见的解决方案是高阶组件、render props 及状态管理框架。
- 复杂的组件变得难以理解。生命周期函数与业务逻辑耦合太深，导致关联部分难以拆分。
- 人和机器都很容易混淆类。常见的有 this 的问题，但在 React 团队中还有类难以优化的问题，希望在编译优化层面做出一些改进。

这三个问题在一定程度上阻碍了 React 的后续发展，所以为了解决这三个问题，Hooks **基于函数组件**开始设计。然而第三个问题决定了 Hooks 只支持函数组件。

那为什么不要在循环、条件或嵌套函数中调用 Hook 呢？因为 Hooks 的设计是基于数组实现。在调用时按顺序加入数组中，如果使用循环、条件或嵌套函数很有可能导致数组取值错位，执行错误的 Hook。当然，实质上 React 的源码里不是数组，是链表。

这些限制会在编码上造成一定程度的心智负担，新手可能会写错，为了避免这样的情况，可以引入 ESLint 的 Hooks 检查插件进行预防。

React.forwardRef是什么？它有什么作用？

React.forwardRef 会创建一个React组件，这个组件能够将其接受的 ref 属性转发到其组件树下的另一个组件中。这种技术并不常见，但在以下两种场景中特别有用：

- 转发 refs 到 DOM 组件
- 在高阶组件中转发 refs

React 废弃了哪些生命周期？为什么？

被废弃的三个函数都是在render之前，因为fber的出现，很可能因为高优先级任务的出现而打断现有任务导致它们会被执行多次。另外的一个原因则是，React想约束使用者，好的框架能够让人不得已写出容易维护和扩展的代码，这一点又是从何谈起，可以从新增加以及即将废弃的生命周期分析入手

1) componentWillMount

首先这个函数的功能完全可以使用componentDidMount和 constructor来代替，异步获取的数据的情况上面已经说明了，而如果抛去异步获取数据，其余的即是初始化而已，这些功能都可以在constructor中执行，除此之外，如果在 willMount 中订阅事件，但在服务端这并不会执行 willUnMount事件，也就是说服务端会导致内存泄漏所以componentWillMount完全可以不使用，但使用者有时候难免因为各 种各样的情况在 componentWilMount中做一些操作，那么React为了约束开发者，干脆就抛掉了这个API

2) componentWillReceiveProps

在老版本的 React 中，如果组件自身的某个 state 跟其 props 密切相关的话，一直都没有一种很优雅的处理方式去更新 state，而是需要在 componentWilReceiveProps 中判断前后两个 props 是否相同，如果不同再将新的 props更新到相应的 state 上去。这样做一来会破坏 state 数据的单一数据源，导致组件状态变得不可预测，另一方面也会增加组件的重绘次数。类似的业务需求也有很多，如一个可以横向滑动的列表，当前高亮的 Tab 显然隶属于列表自身的时，根据传入的某个值，直接定位到某个 Tab。为了解决这些问题，React引入了第一个新的生命周期：getDerivedStateFromProps。它有以下的优点：

- getDSFP是静态方法，在这里不能使用this，也就是一个纯函数，开发者不能写出副作用的代码
- 开发者只能通过prevState而不是prevProps来做对比，保证了state和props之间的简单关系以及不需要处理第一次渲染时prevProps为空的情况
- 基于第一点，将状态变化（setState）和昂贵操作（tabChange）区分开，更加便于 render 和 commit 阶段操作或者说优化。

3) componentWillUpdate

与 componentWillReceiveProps 类似，许多开发者也会在 componentWillUpdate 中根据 props 的变化去触发一些回调 。 但不论是 componentWilReceiveProps 还 是 componentWilUpdate，都有可能在一次更新中被调用多次，也就是说写在这里的回调函数也有可能会被调用多次，这显然是不可取的。与 componentDidMount 类 似，componentDidUpdate 也不存在这样的问题，一次更新中 componentDidUpdate 只会被调用一次，所以将原先写在 componentWillUpdate 中 的 回 调 迁 移 至 componentDidUpdate 就可以解决这个问题。

另外一种情况则是需要获取DOM元素状态，但是由于在fber中，render可打断，可能在wilMount中获取到的元素状态很可能与实际需要的不同，这个通常可以使用第二个新增的生命函数的解决 getSnapshotBeforeUpdate(prevProps, prevState)

4) getSnapshotBeforeUpdate(prevProps, prevState)

返回的值作为componentDidUpdate的第三个参数。与willMount不同的是，getSnapshotBeforeUpdate会在最终确定的render执行之前执行，也就是能保证其获取到的元素状态与didUpdate中获取到的元素状态相同。官方参考代码：

javascript 复制代码

```
class ScrollingList extends React.Component {
  constructor(props) {
    super(props);
    this.listRef = React.createRef();
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
    // 我们是否在 list 中添加新的 items ？
    // 捕获滚动位置以便我们稍后调整滚动位置。
    if (prevProps.list.length < this.props.list.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
```

```
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    // 如果我们 snapshot 有值，说明我们刚刚添加了新的 items，
    // 调整滚动位置使得这些新 items 不会将旧的 items 推出视图。
    //（这里的 snapshot 是 getSnapshotBeforeUpdate 的返回值）
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }

  render() {
    return (
      <div ref={this.listRef}>{/* ...contents... */}</div>
    );
  }
}
```

state 和 props 触发更新的生命周期分别有什么区别?

state 更新流程： 这个过程当中涉及的函数：

1. shouldComponentUpdate: 当组件的 state 或 props 发生改变时，都会首先触发这个生命周期函数。它会接收两个参数：nextProps, nextState——它们分别代表传入的新 props 和新的 state 值。拿到这两个值之后，我们就可以通过一些对比逻辑来决定是否有 re-render（重渲染）的必要了。如果该函数的返回值为 false，则生命周期终止，反之继续；

> 注意：此方法仅作为**性能优化的方式**而存在。不要企图依靠此方法来"阻止"渲染，因为这可能会产生 bug。应该**考虑使用内置的 PureComponent 组件**，而不是手动编写 `shouldComponentUpdate()`

2. componentWillUpdate：当组件的 state 或 props 发生改变时，会在渲染之前调用 componentWillUpdate。componentWillUpdate **是 React16 废弃的三个生命周期之一。**过去，我们可能希望能在这个阶段去收集一些必要的信息（比如更新前的 DOM 信息等等），现在我们完全可以在 React16 的 getSnapshotBeforeUpdate 中去做这些事；
3. componentDidUpdate：componentDidUpdate() 会在UI更新后会被立即调用。它接收 prevProps（上一次的 props 值）作为入参，也就是说在此处我们仍然可以进行 props 值对比（再次说明 componentWillUpdate 确实鸡肋哈）。

props 更新流程： 相对于 state 更新，props 更新后唯一的区别是增加了对 componentWillReceiveProps 的调用。关于 componentWillReceiveProps，需要知道这些事情：

- componentWillReceiveProps：它在Component接受到新的 props 时被触发。componentWillReceiveProps 会接收一个名为 nextProps 的参数（对应新的 props 值）。**该生命周期是 React16 废弃掉的三个生命周期之一。**在它被废弃前，可以用它来比较 this.props 和 nextProps 来重新setState。在 React16 中，用一个类似的新生命周期 getDerivedStateFromProps 来代替它。

React-Router怎么设置重定向?

使用 `<Redirect>` 组件实现路由的重定向：

javascript 复制代码

```
<Switch>
  <Redirect from='/users/:id' to='/users/profile/:id'/>
  <Route path='/users/profile/:id' component={Profile}/>
</Switch>
```

当请求 `/users/:id` 被重定向去 `'/users/profile/:id'` ：

- 属性 `from: string` ： 需要匹配的将要被重定向路径。
- 属性 `to: string` ： 重定向的 URL 字符串
- 属性 `to: object` ： 重定向的 location 对象
- 属性 `push: bool` ： 若为真，重定向操作将会把新地址加入到访问历史记录里面，并且无法回退到前面的页面。

如何将两个或多个组件嵌入到一个组件中?

可以通过以下方式将组件嵌入到一个组件中：

javascript 复制代码

```
class MyComponent extends React.Component{
    render(){
        return(
            <div>
                <h1>Hello</h1>
```

```
                <Header/>
            </div>
        );
    }
}
class Header extends React.Component{
    render(){
        return
            <h1>Header Component</h1>
   };
}
ReactDOM.render(
    <MyComponent/>, document.getElementById('content')
);
```

在React中如何避免不必要的render?

React 基于虚拟 DOM 和高效 Diff 算法的完美配合，实现了对 DOM 最小粒度的更新。大多数情况下，React 对 DOM 的渲染效率足以业务日常。但在个别复杂业务场景下，性能问题依然会困扰我们。此时需要采取一些措施来提升运行性能，其很重要的一个方向，就是避免不必要的渲染（Render）。这里提下优化的点：

- **shouldComponentUpdate 和 PureComponent**

在 React 类组件中，可以利用 shouldComponentUpdate或者 PureComponent 来减少因父组件更新而触发子组件的 render，从而达到目的。shouldComponentUpdate 来决定是否组件是否重新渲染，如果不希望组件重新渲染，返回 false 即可。

- **利用高阶组件**

在函数组件中，并没有 shouldComponentUpdate 这个生命周期，可以利用高阶组件，封装一个类似 PureComponet 的功能

- **使用 React.memo**

React.memo 是 React 16.6 新的一个 API，用来缓存组件的渲染，避免不必要的更新，其实也是一个高阶组件，与 PureComponent 十分类似，但不同的是， React.memo只能用于函数组件。

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

在React中组件的this.state和setState有什么区别?

this.state通常是用来初始化state的，this.setState是用来修改state值的。如果初始化了state之后再使用this.state，之前的state会被覆盖掉，如果使用this.setState，只会替换掉相应的state值。所以，如果想要修改state的值，就需要使用setState，而不能直接修改state，直接修改state之后页面是不会更新的。

react router

javascript 复制代码

```
import React from 'react'
import { render } from 'react-dom'
import { browserHistory, Router, Route, IndexRoute } from 'react-router'

import App from '../components/App'
import Home from '../components/Home'
import About from '../components/About'
import Features from '../components/Features'

render(
  <Router history={browserHistory}>  // history 路由
    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      <Route path='about' component={About} />
      <Route path='features' component={Features} />
    </Route>
  </Router>,
  document.getElementById('app')
)
render(
  <Router history={browserHistory} routes={routes} />,
  document.getElementById('app')
)
```

React Router 提供一个routerWillLeave生命周期钩子，这使得 React组件可以拦截正在发生的跳转，或在离开route前提示用户。routerWillLeave返回值有以下两种：

`return false` 取消此次跳转
`return` 返回提示信息，在离开 route 前提示用户进行确认。

对虚拟 DOM 的理解？虚拟 DOM 主要做了什么？虚拟 DOM 本身是什么？

从本质上来说，Virtual Dom是一个JavaScript对象，通过对象的方式来表示DOM结构。将页面的状态抽象为JS对象的形式，配合不同的渲染工具，使跨平台渲染成为可能。通过事务处理机制，将多次DOM修改的结果一次性的更新到页面上，从而有效的减少页面渲染的次数，减少修改DOM的重绘重排次数，提高渲染性能。

虚拟DOM是对DOM的抽象，这个对象是更加轻量级的对DOM的描述。它设计的最初目的，就是更好的跨平台，比如node.js就没有DOM，如果想实现SSR，那么一个方式就是借助虚拟dom，因为虚拟dom本身是js对象。 在代码渲染到页面之前，vue或者react会把代码转换成一个对象（虚拟DOM）。以对象的形式来描述真实dom结构，最终渲染到页面。在每次数据发生变化前，虚拟dom都会缓存一份，变化之时，现在的虚拟dom会与缓存的虚拟dom进行比较。在vue或者react内部封装了diff算法，通过这个算法来进行比较，渲染时修改改变的变化，原先没有发生改变的通过原先的数据进行渲染。

另外现代前端框架的一个基本要求就是无须手动操作DOM，一方面是因为手动操作DOM无法保证程序性能，多人协作的项目中如果review不严格，可能会有开发者写出性能较低的代码，另一方面更重要的是省略手动DOM操作可以大大提高开发效率。

为什么要用 Virtual DOM：

（1）保证性能下限，在不进行手动优化的情况下，提供过得去的性能

下面对比一下修改DOM时真实DOM操作和Virtual DOM的过程，来看一下它们重排重绘的性能消耗：

- 真实DOM：生成HTML字符串 + 重建所有的DOM元素
- Virtual DOM：生成vNode + DOMDiff + 必要的DOM更新

Virtual DOM的更新DOM的准备工作耗费更多的时间，也就是JS层面，相比于更多的DOM操作它的消费是极其便宜的。尤雨溪在社区论坛中说道：框架给你的保证是，你不需要手动优化

的情况下，我依然可以给你提供过得去的性能。 **（2）跨平台** Virtual DOM本质上是JavaScript的对象，它可以很方便的跨平台操作，比如服务端渲染、uniapp等。

在 React 中，refs 的作用是什么

Refs 可以用于获取一个 DOM 节点或者 React 组件的引用。何时使用 refs 的好的示例有管理焦点/文本选择，触发命令动画，或者和第三方 DOM 库集成。你应该避免使用 String 类型的 Refs 和内联的 ref 回调。Refs 回调是 React 所推荐的。

React中可以在render访问refs吗？为什么？

javascript 复制代码

```
<>
  <span id="name" ref={this.spanRef}>{this.state.title}</span>
  <span>{     this.spanRef.current ? '有值' : '无值'  }</span>
</>
```

不可以，render 阶段 DOM 还没有生成，无法获取 DOM。DOM 的获取需要在 pre-commit 阶段和 commit 阶段：

**

React 与 Vue 的 diff 算法有何不同?

diff 算法是指生成更新补丁的方式，主要应用于虚拟 DOM 树变化后，更新真实 DOM。所以 diff 算法一定存在这样一个过程：触发更新 → 生成补丁 → 应用补丁。

React 的 diff 算法，触发更新的时机主要在 state 变化与 hooks 调用之后。此时触发虚拟 DOM 树变更遍历，采用了深度优先遍历算法。但传统的遍历方式，效率较低。为了优化效率，使用了分治的方式。将单一节点比对转化为了 3 种类型节点的比对，分别是树、组件及元素，以此提升效率。

- 树比对：由于网页视图中较少有跨层级节点移动，两株虚拟 DOM 树只对同一层次的节点进行比较。
- 组件比对：如果组件是同一类型，则进行树比对，如果不是，则直接放入到补丁中。
- 元素比对：主要发生在同层级中，通过标记节点操作生成补丁，节点操作对应真实的 DOM 剪裁操作。

以上是经典的 React diff 算法内容。自 React 16 起，引入了 Fiber 架构。为了使整个更新过程可随时暂停恢复，节点与树分别采用了 FiberNode 与 FiberTree 进行重构。fiberNode 使用了双链表的结构，可以直接找到兄弟节点与子节点。整个更新过程由 current 与 workInProgress 两株树双缓冲完成。workInProgress 更新完成后，再通过修改 current 相关指针指向新节点。

Vue 的整体 diff 策略与 React 对齐，虽然缺乏时间切片能力，但这并不意味着 Vue 的性能更差，因为在 Vue 3 初期引入过，后期因为收益不高移除掉了。除了高帧率动画，在 Vue 中其他的场景几乎都可以使用防抖和节流去提高响应性能。