高级前端一面必会react面试题（持续更新中）

为什么虚拟 dom 会提高性能

> 虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 dom diff 算法避免了没有必要 的 dom 操作，从而提高性能

具体实现步骤如下:

1. 用 JavaScript 对象结构表示 DOM 树的结构;然后用这个树构建一个真正的 DOM 树， 插到文档当中;
2. 当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记 录两棵树差异;
3. 把 2 所记录的差异应用到步骤 1 所构建的真正的 DOM 树上，视图就更新了。

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。

React 数据持久化有什么实践吗?

封装数据持久化组件：

javascript 复制代码

```
let storage={
    // 增加
    set(key, value){
        localStorage.setItem(key, JSON.stringify(value));
    },
    // 获取
    get(key){
        return JSON.parse(localStorage.getItem(key));
    },
    // 删除
    remove(key){
        localStorage.removeItem(key);
    }
};
export default Storage;
```

在React项目中，通过redux存储全局数据时，会有一个问题，如果用户刷新了网页，那么通过redux存储的全局数据就会被全部清空，比如登录信息等。这时就会有全局数据持久化存储的需求。首先想到的就是localStorage，localStorage是没有时间限制的数据存储，可以通过它来实现数据的持久化存储。

但是在已经使用redux来管理和存储全局数据的基础上，再去使用localStorage来读写数据，这样不仅是工作量巨大，还容易出错。那么有没有结合redux来达到持久数据存储功能的框架呢？当然，它就是**redux-persist**。redux-persist会将redux的store中的数据缓存到浏览器的localStorage中。其使用步骤如下：

（1）首先要安装redux-persist：

javascript 复制代码

```
npm i redux-persist
```

（2）对于reducer和action的处理不变，只需修改store的生成代码，修改如下：

javascript 复制代码

```
import {createStore} from 'redux'
import reducers from '../reducers/index'
import {persistStore, persistReducer} from 'redux-persist';
import storage from 'redux-persist/lib/storage';
import autoMergeLevel2 from 'redux-persist/lib/stateReconciler/autoMergeLevel2';
const persistConfig = {
    key: 'root',
    storage: storage,
    stateReconciler: autoMergeLevel2 // 查看 'Merge Process' 部分的具体情况
};
const myPersistReducer = persistReducer(persistConfig, reducers)
const store = createStore(myPersistReducer)
export const persistor = persistStore(store)
export default store
```

（3）在index.js中，将PersistGate标签作为网页内容的父标签：

javascript 复制代码

```
import React from 'react';
import ReactDOM from 'react-dom';
import {Provider} from 'react-redux'
import store from './redux/store/store'
import {persistor} from './redux/store/store'
import {PersistGate} from 'redux-persist/lib/integration/react';
ReactDOM.render(<Provider store={store}>
            <PersistGate loading={null} persistor={persistor}>
                {/*网页内容*/}            </PersistGate>
        </Provider>, document.getElementById('root'));
```

这就完成了通过redux-persist实现React持久化本地数据存储的简单应用。

hooks父子传值

javascript 复制代码

```
父传子
在父组件中用useState声明数据
 const [ data, setData ] = useState(false)
```

把数据传递给子组件

```
<Child data={data} />
```

子组件接收

```
export default function (props) {
    const { data } = props
    console.log(data)
}
```

子传父

子传父可以通过事件方法传值，和父传子有点类似。

在父组件中用useState声明数据

```
 const [ data, setData ] = useState(false)
```

把更新数据的函数传递给子组件

```
<Child setData={setData} />
```

子组件中触发函数更新数据，就会直接传递给父组件

```
export default function (props) {
    const { setData } = props
    setData(true)
}
```

如果存在多个层级的数据传递，也可依照此方法依次传递

```
// 多层级用useContext
const User = () => {
 // 直接获取，不用回调
 const { user, setUser } = useContext(UserContext);
 return <Avatar user={user} setUser={setUser} />;
};
```

为什么 React 要用 JSX?

JSX 是一个 JavaScript 的语法扩展，或者说是一个类似于 XML 的 ECMAScript 语法扩展。它本身没有太多的语法定义，也不期望引入更多的标准。

其实 React 本身并不强制使用 JSX。在没有 JSX 的时候，React 实现一个组件依赖于使用 React.createElement 函数。代码如下：

javascript 复制代码

```
class Hello extends React.Component {
  render() {
    return React.createElement(
        'div',
        null,
        `Hello ${this.props.toWhat}`
```

```
      );
    }
  }
  ReactDOM.render(
    React.createElement(Hello, {toWhat: 'World'}, null),
    document.getElementById('root')
  );
```

而 JSX 更像是一种语法糖，通过类似 XML 的描述方式，描写函数对象。在采用 JSX 之后，这段代码会这样写：

javascript 复制代码

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}
ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);
```

通过对比，可以清晰地发现，代码变得更为简洁，而且代码结构层次更为清晰。

因为 React 需要将组件转化为虚拟 DOM 树，所以在编写代码时，实际上是在手写一棵结构树。而**XML 在树结构的描述上天生具有可读性强的优势。**

但这样可读性强的代码仅仅是给写程序的同学看的，实际上在运行的时候，会使用 Babel 插件将 JSX 语法的代码还原为 React.createElement 的代码。

总结： JSX 是一个 JavaScript 的语法扩展，结构类似 XML。JSX 主要用于声明 React 元素，但 React 中并不强制使用 JSX。即使使用了 JSX，也会在构建过程中，通过 Babel 插件编译为 React.createElement。所以 JSX 更像是 React.createElement 的一种语法糖。

React 团队并不想引入 JavaScript 本身以外的开发体系。而是希望通过合理的关注点分离保持组件开发的纯粹性。

useEffect 与 useLayoutEffect 的区别

（1）共同点

- **运用效果：** useEffect 与 useLayoutEffect 两者都是用于处理副作用，这些副作用包括改变 DOM、设置订阅、操作定时器等。在函数组件内部操作副作用是不被允许的，所以需要使用这两个函数去处理。
- **使用方式：** useEffect 与 useLayoutEffect 两者底层的函数签名是完全一致的，都是调用的 mountEffectImpl方法，在使用上也没什么差异，基本可以直接替换。

(2) 不同点

- **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；而 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。

参考 前端进阶面试题详细解答

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

hooks 常用的

```text
useEffct使用：
如果不传参数：相当于render之后就会执行
传参数为空数组：相当于componentDidMount
```

如果传数组：相当于componentDidUpdate
如果里面返回：相当于componentWillUnmount
会在组件卸载的时候执行清除操作。effect 在每次渲染的时候都会执行。React 会在执行当前 effect 之前对上一个 eff

useLayoutEffect:
useLayoutEffect在浏览器渲染前执行
useEffect在浏览器渲染之后执行

当父组件引入子组件以及在更新某一个值的状态的时候，往往会造成一些不必要的浪费，
而useMemo和useCallback的出现就是为了减少这种浪费，提高组件的性能，
不同点是：useMemo返回的是一个缓存的值，即memoized 值，而useCallback返回的是一个memoized 回调函数。

useCallback
父组件更新子组件会渲染,针对方法不重复执行，包装函数返回函数；

useMemo:
const memoizedValue =useMemo(callback,array)
callback是一个函数用于处理逻辑
array 控制useMemo重新执行行的数组，array改变时才会 重新执行useMemo
不传数组，每次更新都会重新计算
空数组，只会计算一次
依赖对应的值，当对应的值发生变化时，才会重新计算(可以依赖另外一个 useMemo 返回的值)
不能在useMemo里面写副作用逻辑处理，副作用的逻辑处理放在 useEffect内进行处理

自定义hook
自定义 Hook 是一个函数，其名称以 “use” 开头，函数内部可以调用其他的 Hook，
自定义 Hook 是一种自然遵循 Hook 设计的约定，而并不是 React 的特性
在我看来，自定义hook就是把一块业务逻辑单独拿出去写。

```
 const [counter, setCounter] = useState(0);
 const counterRef = useRef(counter);  // 可以保存上一次的变量
```

useRef 获取节点

```
function App() {
    const inputRef = useRef(null);

    return <div>
        <input type="text" ref={inputRef}/>
        <button onClick={() => inputRef.current.focus()}>focus</button>
    </div>
}
```

对 React 和 Vue 的理解，它们的异同

相似之处：

- 都将注意力集中保持在核心库，而将其他功能如路由和全局状态管理交给相关的库
- 都有自己的构建工具，能让你得到一个根据最佳实践设置的项目模板。
- 都使用了Virtual DOM（虚拟DOM）提高重绘性能
- 都有props的概念，允许组件间的数据传递
- 都鼓励组件化应用，将应用分拆成一个个功能明确的模块，提高复用性

不同之处：

1）数据流

Vue默认支持数据双向绑定，而React一直提倡单向数据流

2）虚拟DOM

Vue2.x开始引入"Virtual DOM"，消除了和React在这方面的差异，但是在具体的细节还是有各自的特点。

- Vue宣称可以更快地计算出Virtual DOM的差异，这是由于它在渲染过程中，会跟踪每一个组件的依赖关系，不需要重新渲染整个组件树。
- 对于React而言，每当应用的状态被改变时，全部子组件都会重新渲染。当然，这可以通过PureComponent/shouldComponentUpdate这个生命周期方法来进行控制，但Vue将此视为默认的优化。

3）组件化

React与Vue最大的不同是模板的编写。

- Vue鼓励写近似常规HTML的模板。写起来很接近标准 HTML元素，只是多了一些属性。
- React推荐你所有的模板通用JavaScript的语法扩展——JSX书写。

具体来讲：React中render函数是支持闭包特性的，所以我们import的组件在render中可以直接调用。但是在Vue中，由于模板中使用的数据都必须挂在 this 上进行一次中转，所以 import 完组件之后，还需要在 components 中再声明下。

4）监听数据变化的实现原理不同

- Vue 通过 getter/setter 以及一些函数的劫持，能精确知道数据变化，不需要特别的优化就能达到很好的性能

- React 默认是通过比较引用的方式进行的，如果不优化（PureComponent/shouldComponentUpdate）可能导致大量不必要的vDOM的重新渲染。这是因为 Vue 使用的是可变数据，而React更强调数据的不可变。

5）高阶组件

react可以通过高阶组件（Higher Order Components-- HOC）来扩展，而vue需要通过mixins来扩展。

原因高阶组件就是高阶函数，而React的组件本身就是纯粹的函数，所以高阶函数对React来说易如反掌。相反Vue.js使用HTML模板创建视图组件，这时模板无法有效的编译，因此Vue不采用HOC来实现。

6）构建工具

两者都有自己的构建工具

- React ==> Create React APP
- Vue ==> vue-cli

7）跨平台

- React ==> React Native
- Vue ==> Weex

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。

React中constructor和getInitialState的区别?

两者都是用来初始化state的。前者是ES6中的语法，后者是ES5中的语法，新版本的React中已经废弃了该方法。

getInitialState是ES5中的方法，如果使用createClass方法创建一个Component组件，可以自动调用它的getInitialState方法来获取初始化的State对象，

```javascript 复制代码
var APP = React.creatClass ({
  getInitialState() {
    return {
        userName: 'hi',
        userId: 0
     };
  }
})
```

React在ES6的实现中去掉了getInitialState这个hook函数，规定state在constructor中实现，如下：

```javascript 复制代码
Class App extends React.Component{
    constructor(props){
      super(props);
      this.state={};
    }
  }
```

React必须使用JSX吗?

React 并不强制要求使用 JSX。当不想在构建环境中配置有关 JSX 编译时，不在 React 中使用 JSX 会更加方便。

每个 JSX 元素只是调用 `React.createElement(component, props, ...children)` 的语法糖。因此，使用 JSX 可以完成的任何事情都可以通过纯 JavaScript 完成。

例如，用 JSX 编写的代码：

```javascript 复制代码
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}
ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);
```

可以编写为不使用 JSX 的代码：

```javascript 复制代码
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}
ReactDOM.render(
  React.createElement(Hello, {toWhat: 'World'}, null),
  document.getElementById('root')
);
```

高阶组件

高阶函数：如果一个函数**接受一个或多个函数作为参数或者返回一个函数**就可称之为**高阶函数。**

高阶组件：如果一个函数 **接受一个或多个组件作为参数并且返回一个组件** 就可称之为 **高阶组件。**

react 中的高阶组件

React 中的高阶组件主要有两种形式：**属性代理**和**反向继承。**

属性代理 Proxy

- 操作 `props`
- 抽离 `state`
- 通过 `ref` 访问到组件实例
- 用其他元素包裹传入的组件 `WrappedComponent`

反向继承

会发现其属性代理和反向继承的实现有些类似的地方，都是返回一个继承了某个父类的子类，只不过属性代理中继承的是 `React.Component` ，反向继承中继承的是传入的组件 `WrappedComponent` 。

反向继承可以用来做什么：

1.操作 `state`

高阶组件中可以读取、编辑和删除 `WrappedComponent` 组件实例中的 `state` 。甚至可以增加更多的 `state` 项，但是**非常不建议这么做**因为这可能会导致 `state` 难以维护及管理。

javascript 复制代码

```
function withLogging(WrappedComponent) {
    return class extends WrappedComponent {
        render() {
            return (
                <div>;
                    <h2>;Debugger Component Logging...<h2>;
                    <p>;state:<p>;
                    <pre>;{JSON.stringify(this.state, null, 4)}<pre>;
                    <p>props:<p>;
```

```
                    <pre>{JSON.stringify(this.props, null, 4)}<pre>;
                    {super.render()}
                <div>;
            );
        }
    };
}
```

2.渲染劫持（Render Highjacking）

条件渲染通过 props.isLoading 这个条件来判断渲染哪个组件。

修改由 render() 输出的 React 元素树

如何告诉 React 它应该编译生产环境版

> 通常情况下我们会使用 `Webpack` 的 `DefinePlugin` 方法来将 `NODE_ENV` 变量值设置为 `production` 。编译版本中 `React` 会忽略 `propType` 验证以及其他的告警信息，同时还会降低代码库的大小， `React` 使用了 `Uglify` 插件来移除生产环境下不必要的注释等信息

这段代码有什么问题?

javascript 复制代码

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      username: "有课前端网",
      msg: " ",
    };
  }
  render() {
    return <div> {this.state.msg}</div>;
  }
  componentDidMount() {
    this.setState((oldState, props) => {
      return {
        msg: oldState.username + " - " + props.intro,
      };
    });
```

```
  }
}
```

render (< App intro=" 前端技术专业学习平台">，ickt) 在页面中正常输出“有课前端网-前端技术专业学习平台”。但是这种写法很少使用，并不是常用的写法。React允许对 setState方法传递一个函数，它接收到先前的状态和属性数据并返回一个需要修改的状态对象，正如我们在上面所做的那样。它不但没有问题，而且如果根据以前的状态（ state）以及属性来修改当前状态，推荐使用这种写法。

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性能优化是哪个周期函数

> `shouldComponentUpdate` 这个方法用来判断是否需要调用render方法重新描绘dom。因为dom的描绘非常消耗性能，如果我们能在 `shouldComponentUpdate方` 法中能够写出更优化的 `dom diff` 算法，可以极大的提高性能

react 生命周期

初始化阶段：

- getDefaultProps:获取实例的默认属性
- getInitialState:获取每个实例的初始化状态
- componentWillMount：组件即将被装载、渲染到页面上
- render:组件在这里生成虚拟的 DOM 节点
- componentDidMount:组件真正在被装载之后

运行中状态：

- componentWillReceiveProps:组件将要接收到属性的时候调用
- shouldComponentUpdate:组件接受到新属性或者新状态的时候（可以返回 false，接收数据后不更新，阻止 render 调用，后面的函数不会被继续执行了）

- componentWillUpdate:组件即将更新不能修改属性和状态
- render:组件重新描绘
- componentDidUpdate:组件已经更新

销毁阶段：

- componentWillUnmount:组件即将销毁

shouldComponentUpdate 是做什么的，（react 性能优化是哪个周期函数？）

shouldComponentUpdate 这个方法用来判断是否需要调用 render 方法重新描绘 dom。因为 dom 的描绘非常消耗性能，如果我们能在 shouldComponentUpdate 方法中能够写出更优化的 dom diff 算法，可以极大的提高性能。

在react17 会删除以下三个生命周期
componentWillMount，componentWillReceiveProps， componentWillUpdate

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。

React 设计思路，它的理念是什么?

（1）编写简单直观的代码

React最大的价值不是高性能的虚拟DOM、封装的事件机制、服务器端渲染，而是声明式的直观的编码方式。react文档第一条就是声明式，React 使创建交互式 UI 变得轻而易举。为应用的每一个状态设计简洁的视图，当数据改变时 React 能有效地更新并正确地渲染组件。 以声明式编写 UI，可以让代码更加可靠，且方便调试。

（2）简化可复用的组件

React框架里面使用了简化的组件模型，但更彻底地使用了组件化的概念。React将整个UI上的每一个功能模块定义成组件，然后将小的组件通过组合或者嵌套的方式构成更大的组件。React的组件具有如下的特性：

- 可组合：简单组件可以组合为复杂的组件
- 可重用：每个组件都是独立的，可以被多个组件使用
- 可维护：和组件相关的逻辑和UI都封装在了组件的内部，方便维护
- 可测试：因为组件的独立性，测试组件就变得方便很多。

(3) Virtual DOM

真实页面对应一个 DOM 树。在传统页面的开发模式中，每次需要更新页面时，都要手动操作 DOM 来进行更新。 DOM 操作非常昂贵。在前端开发中，性能消耗最大的就是 DOM 操作，而且这部分代码会让整体项目的代码变得难 以维护。React 把真实 DOM 树转换成 JavaScript 对象树，也就是 Virtual DOM，每次数据更新后，重新计算 Virtual DOM，并和上一次生成的 Virtual DOM 做对比，对发生变化的部分做批量更新。React 也提供了直观的 shouldComponentUpdate 生命周期回调，来减少数据变化后不必要的 Virtual DOM 对比过程，以保证性能。

（4）函数式编程

React 把过去不断重复构建 UI 的过程抽象成了组件，且在给定参数的情况下约定渲染对应的 UI 界面。React 能充分利用很多函数式方法去减少冗余代码。此外，由于它本身就是简单函数，所以易于测试。

（5）一次学习，随处编写

无论现在正在使用什么技术栈，都可以随时引入 React来开发新特性，而不需要重写现有代码。

React 还可以使用 Node 进行服务器渲染，或使用 React Native 开发原生移动应用。因为 React 组件可以映射为对应的原生控件。在输出的时候，是输出 Web DOM，还是 Android 控件，还是 iOS 控件，就由平台本身决定了。所以，react很方便和其他平台集成