高频react面试题自检

fetch封装

javascript 复制代码

```
npm install whatwg-fetch --save  // 适配其他浏览器
npm install es6-promise

export const handleResponse = (response) => {
  if (response.status === 403 || response.status === 401) {
    const oauthurl = response.headers.get('locationUrl');
    if (!_.isEmpty(oauthUrl)) {
      window.location.href = oauthurl;
      return;
    }
  }
  if (!response.ok) {
    return getErrorMessage(response).then(errorMessage => apiError(response.status, errorMessage));
  }
  if (isJson(response)) {
    return response.json();
  }
  if (isText(response)) {
    return response.text();
  }

  return response.blob();
};

const httpRequest = {
  request: ({
    method, headers, body, path, query,
  }) => {
    const options = {};
    let url = path;
    if (method) {
      options.method = method;
    }
    if (headers) {
      options.headers = {...options.headers,...headers};
    }
    if (body) {
      options.body = body;
    }
    if (query) {
```

```
      const params = Object.keys(query)
        .map(k => `${k}=${query[k]}`)
        .join('&');
      url = url.concat(`?${params}`);
    }
    return fetch(url, Object.assign({}, options, { credentials: 'same-origin' })).then(handleRespons
  },
};

export default httpRequest;
```

react-router里的 <Link> 标签和 <a> 标签有什么区别

> 对比 <a> , Link 组件避免了不必要的重渲染

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。

何为 redux

Redux 的基本思想是整个应用的 state 保持在一个单一的 store 中。store 就是一个简单的 javascript 对象，而改变应用 state 的唯一方式是在应用中触发 actions，然后为这些 actions 编写 reducers 来修改 state。整个 state 转化是在 reducers 中完成，并且不应该有任何副作用。

参考：前端react面试题详细解答

怎么阻止组件的渲染

在组件的 `render` 方法中返回 `null` 并不会影响触发组件的生命周期方法

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

对componentWillReceiveProps 的理解

该方法当 `props` 发生变化时执行，初始化 `render` 时不执行，在这个回调函数里面，你可以根据属性的变化，通过调用 `this.setState()` 来更新你的组件状态，旧的属性还是可以通过 `this.props` 来获取,这里调用更新状态是安全的，并不会触发额外的 `render` 调用。

使用好处： 在这个生命周期中，可以在子组件的render函数执行前获取新的props，从而更新子组件自己的state。 可以将数据请求放在这里进行执行，需要传的参数则从componentWillReceiveProps(nextProps)中获取。而不必将所有的请求都放在父组件中。于是该请求只会在该组件渲染时才会发出，从而减轻请求负担。

componentWillReceiveProps在初始化render的时候不会执行，它会在Component接受到新的状态(Props)时被触发，一般用于父组件状态更新时子组件的重新渲染。

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.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。

React中的setState和replaceState的区别是什么?

（1）setState() setState()用于设置状态对象，其语法如下：

```javascript
setState(object nextState[, function callback])
```


- nextState，将要设置的新状态，该状态会和当前的state合并
- callback，可选参数，回调函数。该函数会在setState设置成功，且组件重新渲染后调用。

合并nextState和当前state，并重新渲染组件。setState是React事件处理函数中和请求回调函数中触发UI更新的主要方法。

（2）replaceState() replaceState()方法与setState()类似，但是方法只会保留nextState中状态，原state不在nextState中的状态都会被删除。其语法如下：

```javascript
replaceState(object nextState[, function callback])
```


- nextState，将要设置的新状态，该状态会替换当前的state。
- callback，可选参数，回调函数。该函数会在replaceState设置成功，且组件重新渲染后调用。

总结： setState 是修改其中的部分状态，相当于 Object.assign，只是覆盖，不会减少原来的状态。而replaceState 是完全替换原来的状态，相当于赋值，将原来的 state 替换为另一个对象，如果新状态属性减少，那么 state 中就没有这个状态了。

对有状态组件和无状态组件的理解及使用场景

（1）有状态组件

特点：

- 是类组件
- 有继承
- 可以使用this

- 可以使用react的生命周期
- 使用较多，容易频繁触发生命周期钩子函数，影响性能
- 内部使用 state，维护自身状态的变化，有状态组件根据外部组件传入的 props 和自身的state进行渲染。

使用场景：

- 需要使用到状态的。
- 需要使用状态操作组件的（无状态组件的也可以实现新版本react hooks也可实现）

总结： 类组件可以维护自身的状态变量，即组件的 state ，类组件还有不同的生命周期方法，可以让开发者能够在组件的不同阶段（挂载、更新、卸载），对组件做更多的控制。类组件则既可以充当无状态组件，也可以充当有状态组件。当一个类组件不需要管理自身状态时，也可称为无状态组件。

（2）无状态组件 特点：

- 不依赖自身的状态state
- 可以是类组件或者函数组件。
- 可以完全避免使用 this 关键字。（由于使用的是箭头函数事件无需绑定）
- 有更高的性能。当不需要使用生命周期钩子时，应该首先使用无状态函数组件
- 组件内部不维护 state ，只根据外部组件传入的 props 进行渲染的组件，当 props 改变时，组件重新渲染。

使用场景：

- 组件不需要管理 state，纯展示

优点：

- 简化代码、专注于 render
- 组件不需要被实例化，无生命周期，提升性能。 输出（渲染）只取决于输入（属性），无副作用
- 视图和数据的解耦分离

缺点：

- 无法使用 ref
- 无生命周期方法

- 无法控制组件的重渲染，因为无法使用shouldComponentUpdate 方法，当组件接受到新的属性时则会重渲染

总结： 组件内部状态且与外部无关的组件，可以考虑用状态组件，这样状态树就不会过于复杂，易于理解和管理。当一个组件不需要管理自身状态时，也就是无状态组件，应该优先设计为函数组件。比如自定义的 `<Button/>` 、 `<Input />` 等组件。

什么是状态提升

使用 react 经常会遇到几个组件需要共用状态数据的情况。这种情况下，我们最好将这部分共享的状态提升至他们最近的父组件当中进行管理。我们来看一下具体如何操作吧。

javascript 复制代码

```
import React from 'react'
class Child_1 extends React.Component{
    constructor(props){
        super(props)
    }
    render(){
        return (
            <div>
                <h1>{this.props.value+2}</h1>
            </div>
        )
    }
}
class Child_2 extends React.Component{
    constructor(props){
        super(props)
    }
    render(){
        return (
            <div>
                <h1>{this.props.value+1}</h1>
            </div>
        )
    }
}
class Three extends React.Component {
    constructor(props){
        super(props)
        this.state = {
            txt:"牛逼"
        }
        this.handleChange = this.handleChange.bind(this)
    }
```

```
    handleChange(e){
        this.setState({
            txt:e.target.value
        })
    }
    render(){
       return (
            <div>
                <input type="text" value={this.state.txt} onChange={this.handleChange}/>
                <p>{this.state.txt}</p>
                <Child_1 value={this.state.txt}/>
                <Child_2 value={this.state.txt}/>
            </div>
       )
    }
}
export default Three
```

Redux 怎么实现属性传递，介绍下原理

react-redux 数据传输： view-->action-->reducer-->store-->view。看下点击事件的数据是如何通过redux传到view上：

- view 上的AddClick 事件通过mapDispatchToProps 把数据传到action ---> click: ()=>dispatch(ADD)
- action 的ADD 传到reducer上
- reducer传到store上 const store = createStore(reducer);
- store再通过 mapStateToProps 映射穿到view上text:State.text

代码示例：

javascript 复制代码

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux';
import { Provider, connect } from 'react-redux';
class App extends React.Component{
    render(){
        let { text, click, clickR } = this.props;
        return(
            <div>
                <div>数据:已有人{text}</div>
                <div onClick={click}>加人</div>
                <div onClick={clickR}>减人</div>
            </div>
```

```
        )
    }
}
const initialState = {
    text:5
}
const reducer = function(state,action){
    switch(action.type){
        case 'ADD':
            return {text:state.text+1}
        case 'REMOVE':
            return {text:state.text-1}
        default:
            return initialState;
    }
}

let ADD = {
    type:'ADD'
}
let Remove = {
    type:'REMOVE'
}

const store = createStore(reducer);

let mapStateToProps = function (state){
    return{
        text:state.text
    }
}

let mapDispatchToProps = function(dispatch){
    return{
        click:()=>dispatch(ADD),
        clickR:()=>dispatch(Remove)
    }
}

const App1 = connect(mapStateToProps,mapDispatchToProps)(App);

ReactDOM.render(
    <Provider store = {store}>
        <App1></App1>
    </Provider>,document.getElementById('root')
)
```

diff算法如何比较?

- 只对同级比较，跨层级的dom不会进行复用
- 不同类型节点生成的dom树不同，此时会直接销毁老节点及子孙节点，并新建节点
- 可以通过key来对元素diff的过程提供复用的线索
- 单节点diff
- 单点diff有如下几种情况：
- key和type相同表示可以复用节点
- key不同直接标记删除节点，然后新建节点
- key相同type不同，标记删除该节点和兄弟节点，然后新创建节点

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()

使用箭头函数(arrow functions)的优点是什么

- 作用域安全：在箭头函数之前，每一个新创建的函数都有定义自身的 `this` 值(在构造函数中是新对象；在严格模式下，函数调用中的 `this` 是未定义的；如果函数被称为“对象方法”，则为基础对象等)，但箭头函数不会，它会使用封闭执行上下文的 `this` 值。
- 简单：箭头函数易于阅读和书写
- 清晰：当一切都是一个箭头函数，任何常规函数都可以立即用于定义作用域。开发者总是可以查找 next-higher 函数语句，以查看 `this` 的值

虚拟 DOM 的引入与直接操作原生 DOM 相比，哪一个效率更高，为什么

虚拟DOM相对原生的DOM不一定是效率更高，如果只修改一个按钮的文案，那么虚拟 DOM 的操作无论如何都不可能比真实的 DOM 操作更快。在首次渲染大量DOM时，由于多了一层虚

拟DOM的计算，虚拟DOM也会比innerHTML插入慢。它能保证性能下限，在真实DOM操作的时候进行针对性的优化时，还是更快的。所以要根据具体的场景进行探讨。

在整个 DOM 操作的演化过程中，其实主要矛盾并不在于性能，而在于开发者写得爽不爽，在于研发体验/研发效率。虚拟 DOM 不是别的，正是前端开发们为了追求更好的研发体验和研发效率而创造出来的高阶产物。虚拟 DOM 并不一定会带来更好的性能，React 官方也从来没有把虚拟 DOM 作为性能层面的卖点对外输出过。**虚拟 DOM 的优越之处在于，它能够在提供更爽、更高效的研发模式（也就是函数式的 UI 编程方式）的同时，仍然保持一个还不错的性能。

React组件的构造函数有什么作用？它是必须的吗？

构造函数主要用于两个目的：

- 通过将对象分配给this.state来初始化本地状态
- 将事件处理程序方法绑定到实例上

所以，当在React class中需要设置state的初始值或者绑定事件时，需要加上构造函数，官方Demo：

javascript 复制代码

```
class LikeButton extends React.Component {
  constructor() {
    super();
    this.state = {
      liked: false
    };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState({liked: !this.state.liked});
  }
  render() {
    const text = this.state.liked ? 'liked' : 'haven\'t liked';
    return (
      <div onClick={this.handleClick}>
        You {text} this. Click to toggle.      </div>
    );
  }
}
ReactDOM.render(
  <LikeButton />,
  document.getElementById('example')
```

```
);
```

构造函数用来新建父类的this对象；子类必须在constructor方法中调用super方法；否则新建实例时会报错；因为子类没有自己的this对象，而是继承父类的this对象，然后对其进行加工。如果不调用super方法；子类就得不到this对象。

注意：

- constructor () 必须配上 super(), 如果要在constructor 内部使用 this.props 就要 传入 props , 否则不用
- JavaScript中的 bind 每次都会返回一个新的函数, 为了性能等考虑, 尽量在constructor中绑定事件

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）。

react-router 里的 Link 标签和 a 标签的区别

从最终渲染的 DOM 来看，这两者都是链接，都是 标签，区别是： `<Link>` 是react-router 里实现路由跳转的链接，一般配合 `<Route>` 使用，react-router接管了其默认的链接跳转行为，区别于传统的页面跳转， `<Link>` 的“跳转”行为只会触发相匹配的 `<Route>` 对应的页面内容更新，而不会刷新整个页面。

`<Link>` 做了3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(即是to)，用history (web前端路由两种方式之一，history & hash)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)。

a标签默认事件禁掉之后做了什么才实现了跳转?

javascript 复制代码

```
let domArr = document.getElementsByTagName('a')
[...domArr].forEach(item=>{
    item.addEventListener('click',function () {
        location.href = this.href
    })
})
```