前端核心面试题：React框架

@HOC与RenderProps

答题要点

- 两者都是React复用逻辑的主要方式，另外个主要方式是大名鼎鼎的天王【自定义Hook】
- HOC的逻辑是子组件进去，父组件出来，父组件在直接渲染子组件之余，为子组件注入了**新的props、新的生命周期、新的DOM事件监听、新的副作用**...这一大堆东西即你想复用的数据与逻辑
- RenderProps的逻辑是组件进去，怀了孕(Baby)的组件出来，Baby组件自带**响应式数据与变化逻辑**，即你想复用的数据与逻辑
- 案例：以【实时显式鼠标移动位置】这一复用逻辑为例：

HOC案例

js 复制代码

```
/* 子组件App进去 增强型的父组件XApp出来 */
const withMouse = (App) => {
  class XApp extends Component {
    constructor(props) {
      super(props);

      // 为子组件注入响应式数据x,y
      this.state = {
        x: 0,
        y: 0,
      };
    }

    /* 鼠标移动动态修改状态x,y的值 */
    onMouseMove = (e) => {
      this.setState(() => ({
        x: e.pageX,
        y: e.pageY,
      }));
    };

    render() {
      // 在此以外做的一切事情都是增强
      // return <App {...this.props}></App>
```

```
      return (
        // 给子组件的顶层DOM添加鼠标移动监听器
        <div onMouseMove={this.onMouseMove}>

          {/* 为子组件注入响应式数据x,y */}
          {/* <Com name={props.name} x={this.state.x} y={this.state.y} /> */}
          <App {...this.props} {...this.state} />
        </div>
      );
    }
  }

  // 普通的App进来 增强型App出去
  // 增强1：在顶层DOM身上绑定了鼠标移动事件监听器
  // 增强2：注入了响应式数据x,y即鼠标实时移动到的位置
  return XApp;
};
```

RenderProps案例

js 复制代码

```
import React, { Component } from 'react'

/* 对鼠标移动的state与事件的封装提取 */
class Mouse extends Component {
    constructor(props) {
        super(props);

        // 老鼠baby自带响应式数据
        this.state = {
            x: 0,
            y: 0
        }
    }

    /* 在DOM事件中改变响应式数据 */
    onMouseMove = (e) => {
        this.setState({
            x: e.pageX,
            y: e.pageY
        })
    }

    render() {
        return (
            // 老鼠baby自带DOM事件监听——用以动态改变响应式数据
            <div onMouseMove={this.onMouseMove}>
```

```
                {/* 拿出我妈给我的renderFn(x,y)执行得到DOM */}
                {this.props.renderFn(this.state.x, this.state.y)}
            </div>
        )
    }
}

class App extends Component {
    render() {
        return (
            <div>
                {this.props.name}
                <br />

                {/* 将一只老鼠子组件放在此处 即相当于把【实时显式鼠标移动位置】放在这 */}
                {/* <div style={{ height: "1000px", backgroundColor: "#eee" }}>鼠标实时位置：{x},{y}<
                <Mouse

                    // 告诉我的baby拿到响应式数据x,y后如何渲染JSX
                    renderFn={
                        (x, y) => <div style={{ height: "1000px", backgroundColor: "#eee" }}>鼠标实时
                    }
                />

            </div>
        )
    }
}

export default App
```

@React中怎么在父组件里使用子组件的函数

- 父组件给子组件一个callback,子组件在使用该callback时入参一个自己的函数，让父组件去调用 *PS：纯脑洞问题 没有普适性*

js 复制代码

```
function Parent(props){
    const callMe = (sonFn)=>sonFn()
    return (
        <>
            <Son callback={callMe}>
        </>
    )
}

function Son({callback}){
```

```
    const sonFn = (sonFn)=>alert('老爸调用到我了')
    return (
        <>
            <button onClick={callback(sonFn)}>让父组件调用我的函数</button>
        </>
    )
}
```

@递归组件

- 自己作为自己的子组件，例如：无穷子菜单，博文的无穷回复；
- 核心逻辑，以无穷回复为例：
- 每篇博文有它的回复数据，假设叫replies
- 组件CommentItem正常渲染出每个回复item的用户名、回复时间
- 每个回复的item依然有它的子回复replies
- 将replies中的每一项，继续映射为一个新的CommentItem

```
function CommentItem({item}){
    return <>
        <h3>item.username</h3>
        <span>item.date</span>

        {/*渲染item的子回复replies*/}
        {
            item.replies.map(
                it=><CommentItem item={it}/>
            )
        }
    </>
}
```

[参考链接] (juejin.cn/post/708790...)

@首屏渲染优化

代码层面

- 组件库的按需引入
- 路由的异步加载
- 使用服务端渲染SSR，直接由服务端返回渲染好的HTML
- 事件防抖节流，减少事件处理次数

- 减少DOM层级
- 减少DOM渲染次数（批量渲染documengFragment，差量渲染Vue/React）

通信层面

- 缓存网路数据在localStorage里或vuex/redux里，减少请求次数
- 小图片直接使用base64以减少网络请求次数
- 使用HTTP协议的强缓，服务端给更新频率低的数据一个较长的缓存时间（响应头Cache-Control:maxAge(3600*24*7)）
- 从CDN或静态资源服务器去获取图片、js、css、图标字体等静态资源

打包层面

- 对图片进行物理压缩，如在线压缩工具TinyPNG
- 打包时对HTML,CSS,JS进行压缩处理
- 开启Gzip压缩；
- 合理分包，将第三方包、轮子包、业务代码包分开，这样在前端版本升级时，往往还需要更新业务代码包，而第三方包和轮子包由于内容未变导致hash未变，因此直接被服务端判断为304，从而直接使用本地缓存；
- 打包时开启Tree-Shaking，摇掉无用代码，减少包体积；
- 从HTML中抽离CSS文件，以便单独下载与缓存；

@前端权限控制

- 登录成功后，服务端返回该用户的角色/权限；
- 可以将该角色/权限等级数据存储在全局（例如全局状态管理）；
- 路由层面A计划-动态生成路由表：根据该用户的等级动态添加它有权访问的路由（一旦用户访问自己无权访问的路由时命中404）；
- 路由层面B计划-路由守卫：使用路由守卫，当用户访问越权的路由时一脚踹到登录页；
- 界面层面A计划-条件渲染：根据用户的权限条件渲染它能访问的Link，组件、具体元素等；
- 界面层面B计划-封装：
- Vue中可以自定义指令如 `v-auth="3"` 当用户的等级不足3时将该按钮disable掉或隐藏掉;
- React中可以使用HOC实现例如 `WithCustomAuth(MyButton,3)` 在返回JSX的时候使用条件渲染

```
function WithCustomAuth(Com,level){
    function Parent(props){
```

```
        reurn ({
            authFromRedux >= level
            ? <Com {...props} />
            : <a href="/login">登录</a>
        })
    }
    return Parent
}
```