

react面试题详解

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。

当调用 `setState`的时候，发生了什么操作？**

当调用 `setState`时，React做的第一件事是将传递给`setState`的对象合并到组件的当前状态，这将启动一个称为和解（`reconciliation`）的过程。和解的最终目标是，根据这个新的状态以最有效的方式更新DOM。为此，React将构建一个新的React虚拟DOM树（可以将其视为页面DOM元素的对象表示方式）。一旦有了这个DOM树，为了弄清DOM是如何响应新的状态而改变的，React会将这个新树与上一个虚拟DOM树比较。这样做，React会知道发生的确切变化，并且通过了解发生的变化后，在绝对必要的情况下进行更新DOM，即可将因操作DOM而占用的空间最小化。

为什么要使用 `React.Children.map (props.children, ()=>)`而不是`props.children.map (() =>)`?

因为不能保证 `props.children`将是一个数组。 以下面的代码为例。

```
<Parent>
  <h1>有课前端网</h1>
</Parent>
```

在父组件内部，如果尝试使用 `props.children.map` 映射子对象，则会抛出错误，因为 `props.children` 是一个对象，而不是一个数组。如果有多个子元素，React 会使 `props.children` 成为一个数组，如下所示。

```
<Parent>
  <h1>有课前端网</h1>
  <h2>前端技术学习平台</h2>
</Parent>;
//不建议使用如下方式，在这个案例中会抛出错误。
```

```
class Parent extends Component {
  render() {
    return <div> {this.props.children.map((obj) => obj)}</div>;
  }
}
```

建议使用如下方式，避免在上一个案例中抛出错误。

```
class Parent extends Component {
  render() {
    return <div> {React.Children.map(this.props.children, (obj) => obj)}</div>;
  }
}
```

概述一下 React 中的事件处理逻辑。

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（Synthetic Event）并传入设置的事件处理程序中。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，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（基于事件委托原理）。这样 React 在更新 DOM 时就不需要考虑如何处理附着在 DOM 上的事件监听器，最终达到优化性能的目的。

diff 算法是怎么运作

每一种节点类型有自己的属性，也就是prop，每次进行diff的时候，react会先比较该节点类型，假如节点类型不一样，那么react会直接删除该节点，然后直接创建新的节点插入到其中，假如节点类型一样，那么会比较prop是否有更新，假如有prop不一样，那么react会判定该节点有更新，那么重渲染该节点，然后在对其子节点进行比较，一层一层往下，直到没有子节点

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

- 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
- 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
- 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

虚拟DOM一定会提高性能吗？

很多人认为虚拟DOM一定会提高性能，一定会更快，其实这个说法有点片面，因为虚拟DOM虽然会减少DOM操作，但也无法避免DOM操作

- 它的优势是在于diff算法和批量处理策略,将所有的DOM操作搜集起来，一次性去改变真实的DOM,但在首次渲染上，虚拟DOM会多了一层计算，消耗一些性能，所以有可能会比html渲染的要慢
- 注意，虚拟DOM实际上是给我们找了一条最短，最近的路径，并不是说比DOM操作的更快，而是路径最简单

参考：[前端react面试题详细解答](#)

在 ReactNative中，如何解决8081端口号被占用而提示无法访问的问题？

在运行 react-native start时添加参数port 8082；在 package.json中修改“scripts”中的参数，添加端口号；修改项目下的 node_modules \react-native\local-cli\server\server.js文件配置中的 default端口值。

在 ReactNative中，如何解决 adb devices找不到连接设备的问题？

在使用 Genymotion时，首先需要在SDK的 platform-tools中加入环境变量，然后在 Genymotion中单击 Setting，选择ADB选项卡，单击 Use custom Android SDK tools，浏览本地SDK的位置，单击OK按钮就可以了。启动虚拟机后，在cmd中输入 adb devices可以查看设备。

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

createElement 与 cloneElement 的区别是什么

`createElement` 函数是 JSX 编译之后使用的创建 `React Element` 的函数，而 `cloneElement` 则是用于复制某个元素并传入新的 `Props`

在 Redux中使用 Action要注意哪些问题？

在Redux中使用 Action的时候， Action文件里尽量保持 Action文件的纯净，传入什么数据就返回什么数据，最好把请求的数据和 Action方法分离开，以保持 Action的纯净。

使用状态要注意哪些事情？

要注意以下几点。

- 不要直接更新状态
- 状态更新可能是异步的
- 状态更新要合并。
- 数据从上向下流动

如果创建了类似于下面的 Icketang 元素，那么该如何实现 Icketang 类？

javascript 复制代码

```
<Icketang username="雨夜清荷">{(user) => (user ? <Info user={user} /> : <Loading />)}</Icketang>;
import React, { Component } from "react";
export class Icketang extends Component {
  //请实现你的代码
}
```

在上面的案例中，一个组件接受一个函数作为它的子组件。Icketang 组件的子组件是一个函数，而不是一个常用的组件。这意味着在实现 Icketang 组件时，需要将 props.children 作为一个函数来处理。具体实现如下。

javascript 复制代码

```
import React, { Component } from "react";
class Icketang extends Component {
  constructor(props) {
    super(props);
    this.state = {
      user: props.user,
    };
  }
  componentDidMount() {
    //模拟异步获取数据操作，更新状态
    setTimeout(
      () =>
        this.setState({
          user: "有课前端网",
        }),
      2000
    );
  }
  render() {
    return this.props.children(this.state.user);
  }
}
class Loading extends Component {
  render() {
    return <p>Loading.</p>;
  }
}
class Info extends Component {
  render() {
    return <h1> {this.props.user}</h1>;
  }
}
```

调用 Icketang 组件，并传递给 user 属性数据，把 props.children 作为一个函数来处理。这种模式的好处是，我们已经将父组件与子组件分离了，父组件管理状态。父组件的使用者可以决定父组件以何种形式渲染子组件。为了演示这一点，在渲染 Icketang 组件时，分别传递和不传递 user 属性数据来观察渲染结果。

javascript 复制代码

```
import { render } from "react-dom";  
render(<Icketang>{(user) => (user ? <Info user={user} /> : <Loading />)}</Icketang>, ickt);
```

上述代码没有为 Icketang 组件传递 user 属性数据，因此将首先渲染 Loading 组件，当父组件的 user 状态数据发生改变时，我们发现 Info 组件可以成功地渲染出来。

javascript 复制代码

```
render(<Icketang user="雨夜清荷">{(user) => (user ? <Info user={user} /> : <Loading />)}</Icketang>, ickt);
```

上述代码为 Icketang 组件传递了 user 属性数据，因此将直接渲染 Info 组件，当父组件的 user 状态数据发生改变时，我们发现 Info 组件产生了更新，在整个过程中，Loading 组件都未渲染。

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。

React 高阶组件、Render props、hooks 有什么区别，为什么要不断迭代

这三者是目前react解决代码复用的主要方式：

- 高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。具体而言，高阶组件是参数为组件，返回值为新组件的函数。
- render props是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术，更具体的说，render prop 是一个用于告知组件需要渲染什么内容的函数 prop。
- 通常，render props 和高阶组件只渲染一个子节点。让 Hook 来服务这个使用场景更加简单。这两种模式仍有用武之地，（例如，一个虚拟滚动条组件或许会有一个 renderItem 属性，或是一个可见的容器组件或许会有它自己的 DOM 结构）。但在大部分场景下，Hook 足够了，并且能够帮助减少嵌套。

(1) HOC 官方解释：

高阶组件（HOC）是 React 中用于复用组件逻辑的一种高级技巧。HOC 自身不是 React API 的一部分，它是一种基于 React 的组合特性而形成的设计模式。

简言之，HOC是一种组件的设计模式，HOC接受一个组件和额外的参数（如果需要），返回一个新的组件。HOC 是纯函数，没有副作用。

javascript 复制代码

```
// hoc的定义
function withSubscription(WrappedComponent, selectData) {
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.state = {
        data: selectData(DataSource, props)
      };
    }
    // 一些通用的逻辑处理
    render() {
      // ... 并使用新数据渲染被包装的组件!
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  }
}
```

```
};
```

```
// 使用
```

```
const BlogPostWithSubscription = withSubscription(BlogPost,  
  (DataSource, props) => DataSource.getBlogPost(props.id));
```

HOC的优缺点：

- 优点：逻辑复用、不影响被包裹组件的内部逻辑。
- 缺点：hoc传递给被包裹组件的props容易和被包裹后的组件重名，进而被覆盖

(2) Render props 官方解释：

"render prop"是指一种在 React 组件之间使用一个值为函数的 prop 共享代码的简单技术

具有render prop 的组件接受一个返回React元素的函数，将render的渲染逻辑注入到组件内部。在这里，"render"的命名可以是任何其他有效的标识符。

javascript 复制代码

```
// DataProvider组件内部的渲染逻辑如下
```

```
class DataProvider extends React.Components {  
  state = {  
    name: 'Tom'  
  }  
  
  render() {  
    return (  
      <div>  
        <p>共享数据组件自己内部的渲染逻辑</p>  
        { this.props.render(this.state) }    </div>  
    );  
  }  
}
```

```
// 调用方式
```

```
<DataProvider render={data => (  
  <h1>Hello {data.name}</h1>  
)}>
```

由此可以看到，render props的优缺点也很明显：

- 优点：数据共享、代码复用，将组件内的state作为props传递给调用者，将渲染逻辑交给调用者。
- 缺点：无法在 return 语句外访问数据、嵌套写法不够优雅

(3) Hooks 官方解释：

Hook是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。通过自定义hook，可以复用代码逻辑。

javascript 复制代码

```
// 自定义一个获取订阅数据的hook
function useSubscription() {
  const data = DataSource.getComments();
  return [data];
}
//
function CommentList(props) {
  const {data} = props;
  const [subData] = useSubscription();
  ...
}
// 使用
<CommentList data='hello' />
```

以上可以看出，hook解决了hoc的prop覆盖的问题，同时使用的方式解决了render props的嵌套地狱的问题。hook的优点如下：

- 使用直观；
- 解决hoc的prop 重名问题；
- 解决render props 因共享数据 而出现嵌套地狱的问题；
- 能在return之外使用数据的问题。

需要注意的是：hook只能在组件顶层使用，不可在分支语句中使用。、

diff算法？



- 把树形结构按照层级分解，只比较同级元素。
- 给列表结构的每个单元添加唯一的 `key` 属性，方便比较。
- `React` 只会匹配相同 `class` 的 `component` (这里面的 `class` 指的是组件的名字)
- 合并操作，调用 `component` 的 `setState` 方法的时候, `React` 将其标记为 - `dirty` .到每一个事件循环结束, `React` 检查所有标记 `dirty` 的 `component` 重新绘制.
- 选择性子树渲染。开发人员可以重写 `shouldComponentUpdate` 提高 `diff` 的性能

react-router4的核心

- 路由变成了组件
- 分散到各个页面，不需要配置 比如 `<link> <route></route>`

React中的props为什么是只读的？

`this.props` 是组件之间沟通的一个接口，原则上来讲，它只能从父组件流向子组件。`React` 具有浓重的函数式编程的思想。

提到函数式编程就要提一个概念：纯函数。它有几个特点：

- 给定相同的输入，总是返回相同的输出。
- 过程没有副作用。
- 不依赖外部状态。

`this.props` 就是汲取了纯函数的思想。props的不可以变性就保证的相同的输入，页面显示的内容是一样的，并且不会产生副作用

React中refs的作用是什么？有哪些应用场景？

Refs 提供了一种方式，用于访问在 render 方法中创建的 React 元素或 DOM 节点。Refs 应该谨慎使用，如下场景使用 Refs 比较适合：

- 处理焦点、文本选择或者媒体的控制
- 触发必要的动画
- 集成第三方 DOM 库

Refs 是使用 `React.createRef()` 方法创建的，他通过 `ref` 属性附加到 React 元素上。要在整个组件中使用 Refs，需要将 `ref` 在构造函数中分配给其实例属性：

javascript 复制代码

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.myRef = React.createRef()
  }
  render() {
    return <div ref={this.myRef} />
  }
}
```

由于函数组件没有实例，因此不能在函数组件上直接使用 `ref`：

javascript 复制代码

```
function MyFunctionalComponent() {
  return <input />;
}
class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  render() {
    // 这将会不会工作！
    return (
      <MyFunctionalComponent ref={this.textInput} />
    );
  }
}
```

```
}
```

但可以通过闭合的帮助在函数组件内部进行使用 Refs:

javascript 复制代码

```
function CustomTextInput(props) {  
  // 这里必须声明 textInput, 这样 ref 回调才可以引用它  
  let textInput = null;  
  function handleClick() {  
    textInput.focus();  
  }  
  return (  
    <div>  
      <input  
        type="text"  
        ref={(input) => { textInput = input; }} />      <input  
        type="button"  
        value="Focus the text input"  
        onClick={handleClick}  
      />  
    </div>  
  );  
}
```

注意:

- 不应该过度的使用 Refs
- `ref` 的返回值取决于节点的类型:
 - 当 `ref` 属性被用于一个普通的 HTML 元素时, `React.createRef()` 将接收底层 DOM 元素作为他的 `current` 属性以创建 `ref`。
 - 当 `ref` 属性被用于一个自定义的类组件时, `ref` 对象将接收该组件已挂载的实例作为他的 `current`。
- 当在父组件中需要访问子组件中的 `ref` 时可使用传递 Refs 或回调 Refs。

React setState 调用的原理

具体的执行过程如下(源码级解析):

- 首先调用了 `setState` 入口函数, 入口函数在这里就是充当一个分发器的角色, 根据入参的不同, 将其分发到不同的功能函数中去;

```

ReactComponent.prototype.setState = function (partialState, callback) {
  this.updater.enqueueSetState(this, partialState);
  if (callback) {
    this.updater.enqueueCallback(this, callback, 'setState');
  }
};

```

- `enqueueSetState` 方法将新的 `state` 放进组件的状态队列里，并调用 `enqueueUpdate` 来处理将要更新的实例对象；

```

enqueueSetState: function (publicInstance, partialState) {
  // 根据 this 拿到对应的组件实例
  var internalInstance = getInternalInstanceReadyForUpdate(publicInstance, 'setState');
  // 这个 queue 对应的就是一个组件实例的 state 数组
  var queue = internalInstance._pendingStateQueue || (internalInstance._pendingStateQueue = []);
  queue.push(partialState);
  // enqueueUpdate 用来处理当前的组件实例
  enqueueUpdate(internalInstance);
}

```

- 在 `enqueueUpdate` 方法中引出了一个关键的对象——`batchingStrategy`，该对象所具备的 `isBatchingUpdates` 属性直接决定了当下是要走更新流程，还是应该排队等待；如果轮到执行，就调用 `batchedUpdates` 方法来直接发起更新流程。由此可以推测，`batchingStrategy` 或许正是 React 内部专门用于管控批量更新的对象。

```

function enqueueUpdate(component) {
  ensureInjected();
  // 注意这一句是问题的关键，isBatchingUpdates标识着当前是否处于批量创建/更新组件的阶段
  if (!batchingStrategy.isBatchingUpdates) {
    // 若当前没有处于批量创建/更新组件的阶段，则立即更新组件
    batchingStrategy.batchedUpdates(enqueueUpdate, component);
    return;
  }
  // 否则，先把组件塞入 dirtyComponents 队列里，让它“再等等”
  dirtyComponents.push(component);
  if (component._updateBatchNumber == null) {
    component._updateBatchNumber = updateBatchNumber + 1;
  }
}

```

注意： `batchingStrategy` 对象可以理解为“锁管理器”。这里的“锁”，是指 React 全局唯一的 `isBatchingUpdates` 变量，`isBatchingUpdates` 的初始值是 `false`，意味着“当前并未进行任何批量更新操作”。每当 React 调用 `batchedUpdate` 去执行更新动作时，会先把这个锁给“锁上”（置为 `true`），表明“现在正处于批量更新过程中”。当锁被“锁上”的时候，任何需要更新的组件都只能暂时进入 `dirtyComponents` 里排队等候下一次的批量更新，而不能随意“插队”。此处体现的“任务锁”的思想，是 React 面对大量状态仍然能够实现有序分批处理的基石。