React 面试题 & 回答

> 本项目 **翻译版本**，源于 **sudheerj/reactjs-interview-questions** 这个项目。一时兴起就动起了翻译的念头，由于本人的 React 功力尚浅，翻译的内容难免有误或不妥的地方，望请各位见谅。如果你喜欢这个项目，请 Star，更感谢你的 Pull Request。

以下是现阶段本项目的短期计划：

1. 完成前期的翻译工作
2. 为 React 16 新特性，添加在线示例或完整的示例代码

这里再次感谢 **liaoyongfu** 的大力支持🌹！

目录

16	React 中的合成事件是什么?
17	什么是内联条件表达式?
18	什么是 "key" 属性，在元素数组中使用它们有什么好处?
19	refs 有什么用?
20	如何创建 refs?
21	什么是 forward refs?
22	callback refs 和 findDOMNode() 哪一个是首选选项?
23	为什么 String Refs 被弃用?
24	什么是 Virtual DOM?
25	Virtual DOM 如何工作?
26	Shadow DOM 和 Virtual DOM 之间有什么区别?
27	什么是 React Fiber?
28	React Fiber 的主要目标是什么?
29	什么是受控组件?
30	什么是非受控组件?
31	createElement 和 cloneElement 有什么区别?
32	在 React 中的提升状态是什么?
33	组件生命周期的不同阶段是什么?
34	React 生命周期方法有哪些?
35	什么是高阶组件（HOC）?
36	如何为高阶组件创建属性代理?
37	什么是上下文（Context）?
38	children 属性是什么?
39	怎样在 React 中写注释?
40	构造函数使用带 props 参数的目的是什么?
41	什么是调解?
42	如何使用动态属性名设置 state ?
43	每次组件渲染时调用函数的常见错误是什么?
44	为什么有组件名称要首字母大写?

45	为什么 React 使用 `className` 而不是 `class` 属性?
46	什么是 Fragments ?
47	为什么使用 Fragments 比使用容器 div 更好?
48	在 React 中什么是 Portal ?
49	什么是无状态组件?
50	什么是有状态组件?
51	在 React 中如何校验 props 属性?
52	React 的优点是什么?
53	React 的局限性是什么?
54	在 React v16 中的错误边界是什么?
55	在 React v15 中如何处理错误边界?
56	静态类型检查推荐的方法是什么?
57	`react-dom` 包的用途是什么?
58	`react-dom` 中 render 方法的目的是什么?
59	ReactDOMServer 是什么?
60	在 React 中如何使用 innerHTML?
61	如何在 React 中使用样式?
62	在 React 中事件有何不同?
63	如果在构造函数中使用 `setState()` 会发生什么?
64	索引作为键的影响是什么?
65	在 `componentWillMount()` 方法中使用 `setState()` 好吗?
66	如果在初始状态中使用 props 属性会发生什么?
67	如何有条件地渲染组件?
68	为什么在 DOM 元素上展开 props 需要小心?
69	在 React 中如何使用装饰器?
70	如何 memoize（记忆）组件?
71	如何实现 Server Side Rendering 或 SSR?
72	如何在 React 中启用生产模式?

73	什么是 CRA 及其好处?
74	在 mounting 阶段生命周期方法的执行顺序是什么?
75	在 React v16 中，哪些生命周期方法将被弃用?
76	生命周期方法 `getDerivedStateFromProps()` 的目的是什么?
77	生命周期方法 `getSnapshotBeforeUpdate()` 的目的是什么?
78	createElement() 和 cloneElement() 方法有什么区别?
79	推荐的组件命名方法是什么?
80	在组件类中方法的推荐顺序是什么?
81	什么是 switching 组件?
82	为什么我们需要将函数传递给 setState() 方法?
83	在 React 中什么是严格模式?
84	React Mixins 是什么?
85	为什么 `isMounted()` 是一个反模式，而正确的解决方案是什么?
86	React 中支持哪些指针事件?
87	为什么组件名称应该以大写字母开头?
88	在 React v16 中是否支持自定义 DOM 属性?
89	constructor 和 getInitialState 有什么区别?
90	是否可以在不调用 setState 方法的情况下，强制组件重新渲染?
91	在使用 ES6 类的 React 中 `super()` 和 `super(props)` 有什么区别?
92	在 JSX 中如何进行循环?
93	如何在 attribute 引号中访问 props 属性?
94	什么是 React proptype 数组?
95	如何有条件地应用样式类?
96	React 和 ReactDOM 之间有什么区别?
97	为什么 ReactDOM 从 React 分离出来?
98	如何使用 React label 元素?
99	如何合并多个内联的样式对象?
100	如何在调整浏览器大小时重新渲染视图?

101	`setState()` 和 `replaceState()` 方法之间有什么区别?
102	如何监听状态变化?
103	在 React 状态中删除数组元素的推荐方法是什么?
104	在 React 中是否可以不在页面上渲染 HTML 内容?
105	如何用 React 漂亮地显示 JSON?
106	为什么你不能更新 React 中的 props?
107	如何在页面加载时聚焦一个输入元素?
108	更新状态中的对象有哪些可能的方法?
109	为什么函数比对象更适合于 `setState()` ?
110	我们如何在浏览器中找到当前正在运行的 React 版本?
111	在 `create-react-app` 项目中导入 polyfills 的方法有哪些?
112	如何在 create-react-app 中使用 https 而不是 http?
113	如何避免在 create-react-app 中使用相对路径导入?
114	如何为 React Router 添加 Google Analytics?
115	如何每秒更新一个组件?
116	如何将 vendor prefixes 应用于 React 中的内联样式?
117	如何使用 React 和 ES6 导入和导出组件?
118	为什么 React 组件名称必须以大写字母开头?
119	为什么组件的构造函数只被调用一次?
120	在 React 中如何定义常量?
121	在 React 中如何以编程方式触发点击事件?
122	在 React 中是否可以使用 async/await?
123	React 项目常见的文件结构是什么?
124	最流行的动画软件包是什么?
125	模块化样式文件有什么好处?
126	什么是 React 流行的特定 linters?
127	如何发起 AJAX 调用以及应该在哪些组件生命周期方法中进行 AJAX 调用?
128	什么是渲染属性?

	React Router
129	什么是 React Router?
130	React Router 与 history 库的区别?
131	在 React Router v4 中的 `<Router>` 组件是什么?
132	`history` 中的 `push()` 和 `replace()` 方法的目的是什么?
133	如何使用在 React Router v4 中以编程的方式进行导航?
134	如何在 React Router v4 中获取查询字符串参数?
135	为什么你会得到 "Router may have only one child element" 警告?
136	如何在 React Router v4 中将 params 传递给 `history.push` 方法?
137	如何实现默认页面或 404 页面?
138	如何在 React Router v4 上获取历史对象?
139	登录后如何执行自动重定向?
	React Internationalization
140	什么是 React Intl?
141	React Intl 的主要特性是什么?
142	在 React Intl 中有哪两种格式化方式?
143	在 React Intl 中如何使用 `<FormattedMessage>` 作为占位符使用?
144	如何使用 React Intl 访问当前语言环境?
145	如何使用 React Intl 格式化日期?
	React Testing
146	在 React 测试中什么是浅层渲染（Shallow Renderer）?
147	在 React 中 `TestRenderer` 包是什么?
148	ReactTestUtils 包的目的是什么?
149	什么是 Jest?
150	Jest 对比 Jasmine 有什么优势?
151	举一个简单的 Jest 测试用例
	React Redux
152	什么是 Flux?

153	什么是 Redux?
154	Redux 的核心原则是什么？？
155	与 Flux 相比，Redux 的缺点是什么?
156	`mapStateToProps()` 和 `mapDispatchToProps()` 之间有什么区别?
157	我可以在 reducer 中触发一个 Action 吗?
158	如何在组件外部访问 Redux 存储的对象?
159	MVW 模式的缺点是什么?
160	Redux 和 RxJS 之间是否有任何相似之处?
161	如何在加载时触发 Action?
162	在 React 中如何使用 Redux 的 `connect()` ?
163	如何在 Redux 中重置状态?
164	Redux 中连接装饰器的 `at` 符号的目的是什么?
165	React 上下文和 React Redux 之间有什么区别?
166	为什么 Redux 状态函数称为 reducers ?
167	如何在 Redux 中发起 AJAX 请求?
168	我应该在 Redux Store 中保留所有组件的状态吗?
169	访问 Redux Store 的正确方法是什么?
170	React Redux 中展示组件和容器组件之间的区别是什么?
171	Redux 中常量的用途是什么?
172	编写 `mapDispatchToProps()` 有哪些不同的方法?
173	在 `mapStateToProps()` 和 `mapDispatchToProps()` 中使用 `ownProps` 参数有什么用?
174	如何构建 Redux 项目目录?
175	什么是 redux-saga?
176	redux-saga 的模型概念是什么?
177	在 redux-saga 中 `call()` 和 `put()` 之间有什么区别?
178	什么是 Redux Thunk?
179	`redux-saga` 和 `redux-thunk` 之间有什么区别?
180	什么是 Redux DevTools?

181	Redux DevTools 的功能有哪些?
182	什么是 Redux 选择器以及使用它们的原因?
183	什么是 Redux Form?
184	Redux Form 的主要功能有哪些?
185	如何向 Redux 添加多个中间件?
186	如何在 Redux 中设置初始状态?
187	Relay 与 Redux 有何不同?
	React Native
188	React Native 和 React 有什么区别?
189	如何测试 React Native 应用程序?
190	如何在 React Native 查看日志?
191	怎么调试 React Native 应用?
	React supported libraries & Integration
192	什么是 Reselect 以及它是如何工作的?
193	什么是 Flow?
194	Flow 和 PropTypes 有什么区别?
195	在 React 中如何使用 Font Awesome 图标?
196	什么 是 React 开发者工具?
197	在 Chrome 中为什么 DevTools 没有加载本地文件?
198	如何在 React 中使用 Polymer?
199	与 Vue.js 相比，React 有哪些优势?
200	React 和 Angular 有什么区别?
201	为什么 React 选项卡不会显示在 DevTools 中?
202	什么是 Styled Components?
203	举一个 Styled Components 的例子?
204	什么是 Relay?
205	如何在 `create-react-app` 中使用 TypeScript?
	Miscellaneous

206	Reselect 库的主要功能有哪些?
207	举一个 Reselect 用法的例子?
208	Redux 中的 Action 是什么?
209	在 React 中 statics 对象是否能与 ES6 类一起使用?
210	Redux 只能与 React 一起使用么?
211	您是否需要使用特定的构建工具来使用 Redux ?
212	Redux Form 的 `initialValues` 如何从状态更新?
213	React 是如何为一个属性声明不同的类型?
214	我可以导入一个 SVG 文件作为 React 组件么?
215	为什么不建议使用内联引用回调或函数?
216	在 React 中什么是渲染劫持?
217	什么是 HOC 工厂实现?
218	如何传递数字给 React 组件?
219	我需要将所有状态保存到 Redux 中吗? 我应该使用 react 的内部状态吗?
220	在 React 中 registerServiceWorker 的用途是什么?
221	React memo 函数是什么?
222	React lazy 函数是什么?
223	如何使用 setState 防止不必要的更新?
224	如何在 React 16 版本中渲染数组、字符串和数值?
225	如何在 React 类中使用类字段声明语法?
226	什么是 hooks?
227	Hooks 需要遵循什么规则?
228	如何确保钩子遵循正确的使用规则?
229	Flux 和 Redux 之间有什么区别?
230	React Router V4 有什么好处?
231	您能描述一下 componentDidCatch 生命周期方法签名吗?
232	在哪些情况下，错误边界不会捕获错误?
233	为什么事件处理器不需要错误边界?
234	try catch 与错误边界有什么区别?

235	React 16 中未捕获的错误的行为是什么?
236	放置错误边界的正确位置是什么?
237	从错误边界跟踪组件堆栈有什么好处?
238	在定义类组件时，什么是必须的方法?
239	render 方法可能返回的类型是什么?
240	构造函数的主要目的是什么?
241	是否必须为 React 组件定义构造函数?
242	什么是默认属性?
243	为什么不能在 componentWillUnmount 中调用 setState() 方法?
244	getDerivedStateFromError 的目的是什么?
245	当组件重新渲染时顺序执行的方法有哪些?
246	错误处理期间调用哪些方法?
247	displayName 类属性的用途是什么?
248	支持 React 应用程序的浏览器有哪一些?
249	unmountComponentAtNode 方法的目的是什么?
250	什么是代码拆分?
251	严格模式有什么好处?
252	什么是 Keyed Fragments ?
253	React 支持所有的 HTML 属性么?
254	HOC 有哪些限制?
255	如何在 DevTools 中调试 forwardRefs?
256	什么时候组件的 props 属性默认为 true?
257	什么是 NextJS 及其主要特征?
258	如何将事件处理程序传递给组件?
259	在渲染方法中使用箭头函数好么?
260	如何防止函数被多次调用?
261	JSX 如何防止注入攻击?
262	如何更新已渲染的元素?

263	你怎么说 props 是只读的?
264	你认为状态更新是如何合并的?
265	如何将参数传递给事件处理程序?
266	如何防止组件渲染?
267	安全地使用索引作为键的条件是什么?
268	keys 是否需要全局唯一?
269	用于表单处理的流行选择是什么?
270	formik 相对于其他 redux 表单库有什么优势?
271	为什么不需要使用继承?
272	我可以在 React 应用程序中可以使用 web components 么?
273	什么是动态导入?
274	什么是 loadable 组件?
275	什么是 suspense 组件?
276	什么是基于路由的代码拆分?
277	举例说明如何使用 context?
278	在 context 中默认值的目的是什么?
279	你是怎么使用 contextType?
280	什么是 consumer?
281	在使用 context 时，如何解决性能方面的问题?
282	在 HOCs 中 forward ref 的目的是什么?
283	ref 参数对于所有函数或类组件是否可用?
284	在组件库中当使用 forward refs 时，你需要额外的注意?
285	如何在没有 ES6 的情况下创建 React 类组件
286	是否可以在没有 JSX 的情况下使用 React?
287	什么是差异算法?
288	差异算法涵盖了哪些规则?
289	你什么时候需要使用 refs?
290	对于渲染属性来说是否必须将 prop 属性命名为 render?

Core React

1. 什么是 React?

React 是一个 **开源前端 JavaScript 库**，用于构建用户界面，尤其是单页应用程序。它用于处理网页和移动应用程序的视图层。React 是由 Facebook 的软件工程师 Jordan Walke 创建的。在 2011 年 React 应用首次被部署到 Facebook 的信息流中，之后于 2012 年被应用到 Instagram 上。

阅读资源：

1. React 中文文档
2. 掘金 - 图解 React
3. 掘金 - 200行代码实现简版react

⬆ 返回顶部

2. React 的主要特点是什么?

React 的主要特性有：

- 考虑到真实的 DOM 操作成本很高，它使用 VirtualDOM 而不是真实的 DOM。
- 支持服务端渲染。
- 遵循单向数据流或数据绑定。
- 使用可复用/可组合的 UI 组件开发视图。

⬆ 返回顶部

3. 什么是 JSX?

JSX 是 ECMAScript 一个类似 XML 的语法扩展。基本上，它只是为 `React.createElement()` 函数提供语法糖，从而让在我们在 JavaScript 中，使用类 HTML 模板的语法，进行页面描述。

在下面的示例中， `<h1>` 内的文本标签会作为 JavaScript 函数返回给渲染函数。

```
class App extends React.Component {
  render() {
    return(
      <div>
        <h1>{'Welcome to React world!'}</h1>
      </div>
    )
  }
}
```

以上示例 render 方法中的 JSX 将会被转换为以下内容：

```
React.createElement("div", null, React.createElement(
  "h1", null, 'Welcome to React world!'));
```

这里你可以访问 **Babeljs** 在线体验一下。

阅读资源：

1. **从零开始实现一个React（一）：JSX和虚拟DOM**

⬆ 返回顶部

4. 元素和组件有什么区别?

一个 *Element* 是一个简单的对象，它描述了你希望在屏幕上以 DOM 节点或其他组件的形式呈现的内容。*Elements* 在它们的属性中可以包含其他 *Elements* 。创建一个 React 元素是很轻量的。一旦元素被创建后，它将不会被修改。

React Element 的对象表示如下：

```
const element = React.createElement(
  'div',
  {id: 'login-btn'},
  'Login'
)
```

上面的 `React.createElement()` 函数会返回一个对象。

```
{
  type: 'div',
  props: {
    children: 'Login',
    id: 'login-btn'
  }
}
```

最后使用 `ReactDOM.render()` 方法渲染到 DOM：

```
<div id='login-btn'>Login</div>
```

而一个组件可以用多种不同方式声明。它可以是一个含有 `render()` 方法的类。或者，在简单的情况中，它可以定义为函数。无论哪种情况，它都将 props 作为输入，并返回一个 JSX 树作为输出：

```
const Button = ({ onLogin }) =>
  <div id={'login-btn'} onClick={onLogin} />
```

然后 JSX 被转换成 `React.createElement()` 函数：

```
const Button = ({ onLogin }) => React.createElement(
  'div',
  { id: 'login-btn', onClick: onLogin },
  'Login'
)
```

阅读资源：

1. **为什么React元素有一个$$typeof属性?**

⬆ 返回顶部

5. 如何在 React 中创建组件?

有两种可行的方法来创建一个组件：

1. **Function Components:** 这是创建组件最简单的方式。这些是纯 JavaScript 函数，接受 props 对象作为第一个参数并返回 React 元素：

   ```
   function Greeting({ message }) {
     return <h1>{`Hello, ${message}`}</h1>
   }
   ```

2. **Class Components:** 你还可以使用 ES6 类来定义组件。上面的函数组件若使用 ES6 的类可改写为：

   ```
   class Greeting extends React.Component {
     render() {
       return <h1>{`Hello, ${this.props.message}`}</h1>
     }
   }
   ```

通过以上任意方式创建的组件，可以这样使用：

```
<Greeting message="semlinker"/>
```

在 React 内部对函数组件和类组件的处理方式是不一样的，如：

```
// 如果 Greeting 是一个函数
const result = Greeting(props); // <p>Hello</p>

// 如果 Greeting 是一个类
const instance = new Greeting(props); // Greeting {}
const result = instance.render(); // <p>Hello</p>
```

阅读资源：

1. React 如何区分 Class 和 Function?

⬆ 返回顶部

6. 何时使用类组件和函数组件?

如果组件需要使用**状态或生命周期方法**，那么使用类组件，否则使用函数组件。

⬆ 返回顶部

7. 什么是 Pure Components?

`React.PureComponent` 与 `React.Component` 完全相同，只是它为你处理了 `shouldComponentUpdate()` 方法。当属性或状态发生变化时，PureComponent 将对属性和状态进行**浅比较**。另一方面，一般的组件不会将当前的属性和状态与新的属性和状态进行比较。因此，在默认情况下，每当调用 `shouldComponentUpdate` 时，默认返回 true，所以组件都将重新渲染。

⬆ 返回顶部

8. React 的状态是什么?

组件的状态是一个对象，它包含某些信息，这些信息可能在组件的生命周期中发生更改。我们应该尽量使状态尽可能简单，并尽量减少有状态组件的数量。让我们创建一个包含消息状态的 User 组件：

```
class User extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      message: 'Welcome to React world'
    }
  }

  render() {
    return (
      <div>
        <h1>{this.state.message}</h1>
      </div>
    )
  }
}
```

状态（State）与属性（Props）类似，但它是私有的，完全由组件控制。也就是说，除了它所属的组件外，任何组件都无法访问它。

⬆ 返回顶部

9. React 中的 props 是什么?

Props 是组件的输入。它们是单个值或包含一组值的对象，这些值在创建时使用类似于 HTML 标记属性的命名约定传递给组件。它们是从父组件传递到子组件的数据。

Props 的主要目的是提供以下组件功能：

1. 将自定义数据传递到组件。
2. 触发状态更改。
3. 在组件的 `render()` 方法中通过 `this.props.reactProp` 使用。

例如，让我们使用 `reactProp` 属性创建一个元素：

```
<Element reactProp={'1'} />
```

然后， `reactProp` 将成为附加到 React props 对象的属性，该对象最初已存在于使用 React 库创建的所有组件上。

```
props.reactProp
```

⬆ 返回顶部

10. 状态和属性有什么区别?

state 和 props 都是普通的 JavaScript 对象。虽然它们都保存着影响渲染输出的信息，但它们在组件方面的功能不同。Props 以类似于函数参数的方式传递给组件，而状态则类似于在函数内声明变量并对它进行管理。

States vs Props

Conditions	States	Props
可从父组件接收初始值	是	是
可在父组件中改变其值	否	是
在组件内设置默认值	是	是
在组件内可改变	是	否
可作为子组件的初始值	是	是

⬆ 返回顶部

11. 我们为什么不能直接更新状态?

如果你尝试直接改变状态，那么组件将不会重新渲染。

```
//Wrong
this.state.message = 'Hello world'
```

正确方法应该是使用 `setState()` 方法。它调度组件状态对象的更新。当状态更改时，组件通将会重新渲染。

```
//Correct
this.setState({ message: 'Hello World' })
```

注意： 你可以在 *constructor* 中或使用最新的 JavaScript 类属性声明语法直接设置状态对象。

⬆ 返回顶部

12. 回调函数作为 `setState()` 参数的目的是什么?

当 setState 完成和组件渲染后，回调函数将会被调用。由于 `setState()` 是异步的，回调函数用于任何后续的操作。

注意： 建议使用生命周期方法而不是此回调函数。

```
setState({ name: 'John' }, () => console.log('The name has updated and component re-rendered'))
```

阅读资源：

1. 掘金 - 揭密React setState
2. setState 如何知道该做什么?

⬆ 返回顶部

13. HTML 和 React 事件处理有什么区别?

1. 在 HTML 中事件名必须小写：

```
<button onclick='activateLasers()'>
```

而在 React 中它遵循 *camelCase* (驼峰) 惯例：

```
<button onClick={activateLasers}>
```

2. 在 HTML 中你可以返回 `false` 以阻止默认的行为：

```
<a href='#' onclick='console.log("The link was clicked."); return false;' />
```

而在 React 中你必须地明确地调用 `preventDefault()` ：

```
function handleClick(event) {
  event.preventDefault()
  console.log('The link was clicked.')
}
```

⬆ 返回顶部

14. 如何在 JSX 回调中绑定方法或事件处理程序?

实现这一点有三种可能的方法：

1. **Binding in Constructor:** 在 JavaScript 类中，方法默认不被绑定。这也适用于定义为类方法的 React 事件处理程序。通常我们在构造函数中绑定它们。

```
class Component extends React.Componenet {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
  }

  handleClick() {
    // ...
  }
}
```

2. **Public class fields syntax:** 如果你不喜欢 bind 方案，则可以使用 *public class fields syntax* 正确绑定回调。

```
handleClick = () => {
  console.log('this is:', this)
}
```

```
<button onClick={this.handleClick}>
  {'Click me'}
</button>
```

3. **Arrow functions in callbacks:** 你可以在回调函数中直接使用 *arrow functions* 。

```
<button onClick={(event) => this.handleClick(event)}>
  {'Click me'}
</button>
```

注意： 如果回调函数作为属性传给子组件，那么这些组件可能触发一个额外的重新渲染。在这些情况下，考虑到性能，最好使用 `.bind()` 或 *public class fields syntax* 方案。

⬆ 返回顶部

15. 如何将参数传递给事件处理程序或回调函数?

你可以使用箭头函数来包装事件处理器并传递参数：

```
<button onClick={() => this.handleClick(id)} />
```

这相当于调用 `.bind` ：

```
<button onClick={this.handleClick.bind(this, id)} />
```

⬆ 返回顶部

16. React 中的合成事件是什么?

`SyntheticEvent` 是对浏览器原生事件的跨浏览器包装。它的 API 与浏览器的原生事件相同，包括 `stopPropagation()` 和 `preventDefault()` ，除了事件在所有浏览器中的工作方式相同。

⬆ 返回顶部

17. 什么是内联条件表达式?

在 JS 中你可以使用 if 语句或三元表达式，来实现条件判断。除了这些方法之外，你还可以在 JSX 中嵌入任何表达式，方法是将它们用大括号括起来，然后再加上 JS 逻辑运算符 `&&` 。

```
<h1>Hello!</h1>
{
    messages.length > 0 && !isLogin ?
      <h2>
          You have {messages.length} unread messages.
      </h2>
      :
      <h2>
          You don't have unread messages.
      </h2>
}
```

当然如果只是想判断 if，可以如下直接判断：

```
{
    isLogin && <span>Your have been login!</span>
}
```

在上面的代码中，不需要使用 `isLogin ? <span>Your have been login!</span> : null` 这样的形式。

⬆ 返回顶部

18. 什么是 "key" 属性，在元素数组中使用它们有什么好处?

`key` 是一个特殊的字符串属性，你在创建元素数组时需要包含它。 *Keys* 帮助 React 识别哪些项已更改、添加或删除。

我们通常使用数据中的 IDs 作为 *keys*：

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
)
```

在渲染列表项时，如果你没有稳定的 IDs，你可能会使用 *index* 作为 *key*：

```
const todoItems = todos.map((todo, index) =>
  <li key={index}>
    {todo.text}
  </li>
)
```

注意：

1. 由于列表项的顺序可能发生改变，因此并不推荐使用 *indexes* 作为 *keys* 。这可能会对性能产生负面影响，并可能导致组件状态出现问题。
2. 如果将列表项提取为单独的组件，则在列表组件上应用 *keys* 而不是 `li` 标签。
3. 如果在列表项中没有设置 `key` 属性，在控制台会显示警告消息。

⬆ 返回顶部

19. ## refs 有什么用?

ref 用于返回对元素的引用。但在大多数情况下，应该避免使用它们。当你需要直接访问 DOM 元素或组件的实例时，它们可能非常有用。

20. ## 如何创建 refs?

这里有两种方案

1. 这是最近增加的一种方案。*Refs* 是使用 `React.createRef()` 方法创建的，并通过 `ref` 属性添加到 React 元素上。为了在整个组件中使用 *refs* ，只需将 *ref* 分配给构造函数中的实例属性。

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

2. 你也可以使用 ref 回调函数的方案，而不用考虑 React 版本。例如，访问搜索栏组件中的 `input` 元素如下：

```
class SearchBar extends Component {
   constructor(props) {
      super(props);
      this.txtSearch = null;
      this.state = { term: '' };
      this.setInputSearchRef = e => {
         this.txtSearch = e;
      }
   }

   onInputChange(event) {
      this.setState({ term: this.txtSearch.value });
   }

   render() {
      return (
         <input
            value={this.state.term}
            onChange={this.onInputChange.bind(this)}
            ref={this.setInputSearchRef} />
      );
   }
}
```

你也可以在使用 **closures** 的函数组件中使用 *refs* 。

注意： 你也可以使用内联引用回调，尽管这不是推荐的方法。

⬆ 返回顶部

21. 什么是 forward refs?

Ref forwarding 是一个特性，它允许一些组件获取接收到 *ref* 对象并将它进一步传递给子组件。

```
const ButtonElement = React.forwardRef((props, ref) => (
  <button ref={ref} className="CustomButton">
    {props.children}
  </button>
));

// Create ref to the DOM button:
const ref = React.createRef();
<ButtonElement ref={ref}>{'Forward Ref'}</ButtonElement>
```

⬆ 返回顶部

22. callback refs 和 findDOMNode() 哪一个是首选选项?

最好是使用 *callback refs* 而不是 `findDOMNode()` API。因为 `findDOMNode()` 阻碍了将来对 React 的某些改进。

使用 `findDOMNode` 已弃用的方案：

```
class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this).scrollIntoView()
  }

  render() {
    return <div />
  }
}
```

推荐的方案是：

```
class MyComponent extends Component {
  componentDidMount() {
    this.node.scrollIntoView()
  }

  render() {
    return <div ref={node => this.node = node} />
  }
}
```

⬆ 返回顶部

23. 为什么 String Refs 被弃用?

如果你以前使用过 React，你可能会熟悉旧的 API，其中的 `ref` 属性是字符串，如 `ref={'textInput'}`，并且 DOM 节点的访问方式为 `this.refs.textInput`。我们建议不要这样做，因为字符串引用有以下问题，并且被认为是遗留问题。字符串 refs 在 React v16 版本中被移除。

1. 它们强制 React 跟踪当前执行的组件。这是有问题的，因为它使 React 模块有状态，这会导致在 bundle 中复制 React 模块时会导致奇怪的错误。
2. 它们是不可组合的 - 如果一个库把一个 ref 传给子元素，则用户无法对其设置另一个引用。
3. 它们不能与静态分析工具一起使用，如 Flow。Flow 无法猜测出 `this.refs` 上的字符串引用的作用及其类型。Callback refs 对静态分析更友好。
4. 使用 "render callback" 模式（比如： ），它无法像大多数人预期的那样工作。

```
class MyComponent extends Component {
  renderRow = (index) => {
    // This won't work. Ref will get attached to DataTable rather than
MyComponent:
    return <input ref={'input-' + index} />;

    // This would work though! Callback refs are awesome.
    return <input ref={input => this['input-' + index] = input} />;
  }

  render() {
    return <DataTable data={this.props.data} renderRow={this.renderRow} />
  }
}
```

⬆ 返回顶部

24. 什么是 Virtual DOM?

Virtual DOM (VDOM) 是 *Real DOM* 的内存表示形式。UI 的展示形式被保存在内存中并与真实的 DOM 同步。这是在调用的渲染函数和在屏幕上显示元素之间发生的一个步骤。整个过程被称为 *reconciliation* 。

Real DOM vs Virtual DOM

Real DOM	Virtual DOM
更新较慢	更新较快
可以直接更新 HTML	无法直接更新 HTML
如果元素更新，则创建新的 DOM	如果元素更新，则更新 JSX
DOM 操作非常昂贵	DOM 操作非常简单
较多的内存浪费	没有内存浪费

阅读资源：

1. 知乎 - 如何理解虚拟DOM?
2. edureka - react-interview-questions

⬆ 返回顶部

25. Virtual DOM 如何工作?

Virtual DOM 分为三个简单的步骤。

1. 每当任何底层数据发生更改时，整个 UI 都将以 Virtual DOM 的形式重新渲染。
2. 然后计算先前 Virtual DOM 对象和新的 Virtual DOM 对象之间的差异。
3. 一旦计算完成，真实的 DOM 将只更新实际更改的内容。

⬆ 返回顶部

26. Shadow DOM 和 Virtual DOM 之间有什么区别?

Shadow DOM 是一种浏览器技术，它解决了构建网络应用的脆弱性问题。Shadow DOM 修复了 CSS 和 DOM。它在网络平台中引入作用域样式。 无需工具或命名约定，你即可使用原生 JavaScript 捆绑 CSS 和标记、隐藏实现详情以及编写独立的组件。 *Virtual DOM* 是一个由 JavaScript 库在浏览器 API 之上实现的概念。

⬆ 返回顶部

27. 什么是 React Fiber?

Fiber 是 React v16 中新的 *reconciliation* 引擎，或核心算法的重新实现。React Fiber 的目标是提高对动画，布局，手势，暂停，中止或者重用任务的能力及为不同类型的更新分配优先级，及新的并发原语等领域的适用性。

⬆ 返回顶部

28. React Fiber 的主要目标是什么?

React Fiber 的目标是提高其在动画、布局和手势等领域的适用性。它的主要特性是 **incremental rendering**: 将渲染任务拆分为小的任务块并将任务分配到多个帧上的能力。

⬆ 返回顶部

29. 什么是受控组件?

在随后的用户输入中，能够控制表单中输入元素的组件被称为受控组件，即每个状态更改都有一个相关联的处理程序。

例如，我们使用下面的 handleChange 函数将输入框的值转换成大写：

```
handleChange(event) {
  this.setState({value: event.target.value.toUpperCase()})
}
```

⬆ 返回顶部

30. 什么是非受控组件?

非受控组件是在内部存储其自身状态的组件，当需要时，可以使用 ref 查询 DOM 并查找其当前值。这有点像传统的 HTML。

在下面的 UserProfile 组件中，我们通过 ref 引用 `name` 输入框：

```
class UserProfile extends React.Component {
  constructor(props) {
    super(props)
    this.handleSubmit = this.handleSubmit.bind(this)
    this.input = React.createRef()
  }
```

```
    handleSubmit(event) {
      alert('A name was submitted: ' + this.input.current.value)
      event.preventDefault()
    }

    render() {
      return (
        <form onSubmit={this.handleSubmit}>
          <label>
            {'Name:'}
            <input type="text" ref={this.input} />
          </label>
          <input type="submit" value="Submit" />
        </form>
      );
    }
  }
```

在大多数情况下，建议使用受控组件来实现表单。

⬆ 返回顶部

31. createElement 和 cloneElement 有什么区别?

JSX 元素将被转换为 `React.createElement()` 函数来创建 React 元素，这些对象将用于表示 UI 对象。而 `cloneElement` 用于克隆元素并传递新的属性。

⬆ 返回顶部

32. 在 React 中的提升状态是什么?

当多个组件需要共享相同的更改数据时，建议将共享状态提升到最接近的共同祖先。这意味着，如果两个子组件共享来自其父组件的相同数据，则将状态移动到父组件，而不是在两个子组件中维护局部状态。

⬆ 返回顶部

33. 组件生命周期的不同阶段是什么?

组件生命周期有三个不同的生命周期阶段：

1. **Mounting:** 组件已准备好挂载到浏览器的 DOM 中. 此阶段包含来自 `constructor()`，`getDerivedStateFromProps()`，`render()`，和 `componentDidMount()` 生命周期方法中的初始化过程。
2. **Updating:** 在此阶段，组件以两种方式更新，发送新的属性并使用 `setState()` 或 `forceUpdate()` 方法更新状态. 此阶段包含 `getDerivedStateFromProps()`，`shouldComponentUpdate()`，`render()`，`getSnapshotBeforeUpdate()` 和 `componentDidUpdate()` 生命周期方法。
3. **Unmounting:** 在这个最后阶段，不需要组件，它将从浏览器 DOM 中卸载。这个阶段包含 `componentWillUnmount()` 生命周期方法。

值得一提的是，在将更改应用到 DOM 时，React 内部也有阶段概念。它们按如下方式分隔开：

1. **Render** 组件将会进行无副作用渲染。这适用于纯组件（Pure Component），在此阶段，React 可以暂停，中止或重新渲染。
2. **Pre-commit** 在组件实际将更改应用于 DOM 之前，有一个时刻允许 React 通过

`getSnapshotBeforeUpdate()` 捕获一些 DOM 信息（例如滚动位置）。

3. **Commit** React 操作 DOM 并分别执行最后的生命周期： `componentDidMount()` 在 DOM 渲染完成后调用，`componentDidUpdate()` 在组件更新时调用，`componentWillUnmount()` 在组件卸载时调用。 React 16.3+ 阶段 (也可以看 **交互式版本**)

React 16.3 之前

⬆ 返回顶部

34. React 生命周期方法有哪些?

React 16.3+

- **getDerivedStateFromProps:** 在调用 `render()` 之前调用，并在 *每次* 渲染时调用。 需要使用派生状态的情况是很罕见得。值得阅读 **如果你需要派生状态**.
- **componentDidMount:** 首次渲染后调用，所有得 Ajax 请求、DOM 或状态更新、设置事件监听器都应该在此处发生。
- **shouldComponentUpdate:** 确定组件是否应该更新。 默认情况下，它返回 `true` 。 如果你确定在更新状态或属性后不需要渲染组件，则可以返回 `false` 值。 它是一个提高性能的好地方，因为它允许你在组件接收新属性时阻止重新渲染。
- **getSnapshotBeforeUpdate:** 在最新的渲染输出提交给 DOM 前将会立即调用，这对于从 DOM 捕获信息（比如：滚动位置）很有用。
- **componentDidUpdate:** 它主要用于更新 DOM 以响应 prop 或 state 更改。 如果 `shouldComponentUpdate()` 返回 `false` ，则不会触发。
- **componentWillUnmount** 当一个组件被从 DOM 中移除时，该方法被调用，取消网络请求或者移除与该组件相关的事件监听程序等应该在这里进行。

Before 16.3

- **componentWillMount:** 在组件 `render()` 前执行，用于根组件中的应用程序级别配置。应该避免在该方法中引入任何的副作用或订阅。
- **componentDidMount:** 首次渲染后调用，所有得 Ajax 请求、DOM 或状态更新、设置事件监听器都应该在此处发生。
- **componentWillReceiveProps:** 在组件接收到新属性前调用，若你需要更新状态响应属性改变（例如，重置它），你可能需对比 `this.props` 和 `nextProps` 并在该方法中使用 `this.setState()` 处理状态改变。
- **shouldComponentUpdate:** 确定组件是否应该更新。 默认情况下，它返回 `true` 。 如果你确定在更新状态或属性后不需要渲染组件，则可以返回 `false` 值。 它是一个提高性能的好地方，因为它允许你在组件接收新属性时阻止重新渲染。
- **componentWillUpdate:** 当 `shouldComponentUpdate` 返回 `true` 后重新渲染组件之前执行，注意你不能在这调用 `this.setState()`
- **componentDidUpdate:** 它主要用于更新 DOM 以响应 prop 或 state 更改。 如果 `shouldComponentUpdate()` 返回 `false` ，则不会触发。
- **componentWillUnmount:** 当一个组件被从 DOM 中移除时，该方法被调用，取消网络请求或者移除与该组件相关的事件监听程序等应该在这里进行。

⬆ 返回顶部

35. 什么是高阶组件（HOC）？

高阶组件（*HOC*）就是一个函数，且该函数接受一个组件作为参数，并返回一个新的组件，它只是一种模式，这种模式是由 `react` 自身的组合性质必然产生的。

我们将它们称为**纯组件**，因为它们可以接受任何动态提供的子组件，但它们不会修改或复制其输入组件中的任何行为。

```
const EnhancedComponent = higherOrderComponent(WrappedComponent)
```

HOC 有很多用例：

1. 代码复用，逻辑抽象化
2. 渲染劫持
3. 抽象化和操作状态（`state`）
4. 操作属性（`props`）

> 译注：更详细用法请参考 **高阶组件的使用**

⬆ 返回顶部

36. 如何为高阶组件创建属性代理?

你可以使用 *属性代理* 模式向输入组件增加或编辑属性（props）：

```
function HOC(WrappedComponent) {
  return class Test extends Component {
    render() {
      const newProps = {
        title: 'New Header',
        footer: false,
        showFeatureX: false,
        showFeatureY: true
      };

      return <WrappedComponent {...this.props} {...newProps} />
    }
  }
}
```

⬆ 返回顶部

37. 什么是上下文（Context）?

Context 通过组件树提供了一个传递数据的方法，从而避免了在每一个层级手动的传递 `props`。比如，需要在应用中许多组件需要访问登录用户信息、地区偏好、UI主题等。

```
// 创建一个 theme Context,  默认 theme 的值为 light
const ThemeContext = React.createContext('light');

function ThemedButton(props) {
  // ThemedButton 组件从 context 接收 theme
  return (
    <ThemeContext.Consumer>
      {theme => <Button {...props} theme={theme} />}
    </ThemeContext.Consumer>
```

```
    );
  }

  // 中间组件
  function Toolbar(props) {
    return (
      <div>
        <ThemedButton />
      </div>
    );
  }

  class App extends React.Component {
    render() {
      return (
        <ThemeContext.Provider value="dark">
          <Toolbar />
        </ThemeContext.Provider>
      );
    }
  }
```

⬆ 返回顶部

38. children 属性是什么?

Children 是一个属性（`this.props.children`），它允许你将组件作为数据传递给其他组件，就像你使用的任何其他组件一样。在组件的开始和结束标记之间放置的组件树将作为 `children` 属性传递给该组件。

React API 中有许多方法中提供了这个不透明数据结构的方法，包括：`React.Children.map`、`React.Children.forEach`、`React.Children.count`、`React.Children.only`、`React.Children.toArray`。

```
  const MyDiv = React.createClass({
    render: function() {
      return <div>{this.props.children}</div>
    }
  })

  ReactDOM.render(
    <MyDiv>
      <span>{'Hello'}</span>
      <span>{'World'}</span>
    </MyDiv>,
    node
  )
```

⬆ 返回顶部

39. 怎样在 React 中写注释?

React/JSX 中的注释类似于 JavaScript 的多行注释，但是是用大括号括起来。

单行注释：

```
<div>
  {/* 单行注释（在原生 JavaScript 中，单行注释用双斜杠（//）表示） */}
  {`Welcome ${user}, let's play React`}
</div>
```

多行注释：

```
<div>
  {/* 多行注释超过
   一行 */}
  {`Welcome ${user}, let's play React`}
</div>
```

⬆ 返回顶部

40. 构造函数使用带 props 参数的目的是什么？

在调用 `super()` 方法之前，子类构造函数不能使用 `this` 引用。这同样适用于ES6子类。将 `props` 参数传递给 `super()` 的主要原因是为了在子构造函数中访问 `this.props` 。

带 props 参数：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)

    console.log(this.props) // prints { name: 'John', age: 42 }
  }
}
```

不带 props 参数：

```
class MyComponent extends React.Component {
  constructor(props) {
    super()

    console.log(this.props) // prints undefined

    // but props parameter is still available
    console.log(props) // prints { name: 'John', age: 42 }
  }

  render() {
    // no difference outside constructor
    console.log(this.props) // prints { name: 'John', age: 42 }
  }
}
```

上面的代码片段显示 `this.props` 仅在构造函数中有所不同。 它在构造函数之外是相同的。

⬆ 返回顶部

41. 什么是调解?

当组件的 `props` 或 `state` 发生更改时，React 通过将新返回的元素与先前呈现的元素进行比较来确定是否需要实际的 DOM 更新。当它们不相等时，React 将更新 DOM 。此过程称为 *reconciliation* 。

⬆ 返回顶部

42. 如何使用动态属性名设置 state ?

如果你使用 ES6 或 Babel 转换器来转换你的 JSX 代码，那么你可以使用 *计算属性名称* 来完成此操作。

```
handleInputChange(event) {
  this.setState({ [event.target.id]: event.target.value })
}
```

⬆ 返回顶部

43. 每次组件渲染时调用函数的常见错误是什么?

你需要确保在将函数作为参数传递时未调用该函数。

```
render() {
  // Wrong: handleClick is called instead of passed as a reference!
  return <button onClick={this.handleClick()}>{'Click Me'}</button>
}
```

相反地，传递函数本身应该没有括号：

```
render() {
  // Correct: handleClick is passed as a reference!
  return <button onClick={this.handleClick}>{'Click Me'}</button>
}
```

⬆ 返回顶部

44. 为什么有组件名称要首字母大写?

这是必要的，因为组件不是 DOM 元素，它们是构造函数。 此外，在 JSX 中，小写标记名称是指 HTML 元素，而不是组件。

⬆ 返回顶部

45. 为什么 React 使用 `className` 而不是 `class` 属性?

`class` 是 JavaScript 中的关键字，而 JSX 是 JavaScript 的扩展。这就是为什么 React 使用 `className` 而不是 `class` 的主要原因。传递一个字符串作为 `className` 属性。

```
render() {
  return <span className={'menu navigation-menu'}>{'Menu'}</span>
}
```

在实际项目中，我们经常使用 classnames 来方便我们操作 `className` 。

⬆ 返回顶部

46. 什么是 Fragments ?

它是 React 中的常见模式，用于组件返回多个元素。 *Fragments* 可以让你聚合一个子元素列表，而无需向 DOM 添加额外节点。

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  )
}
```

以下是简洁语法，但是在一些工具中还不支持：

```
render() {
  return (
    <>
      <ChildA />
      <ChildB />
      <ChildC />
    </>
  )
}
```

> 译注： `React 16` 以前， `render` 函数的返回必须有一个根节点，否则报错。

⬆ 返回顶部

47. 为什么使用 Fragments 比使用容器 div 更好?

1. 通过不创建额外的 DOM 节点，Fragments 更快并且使用更少的内存。这在非常大而深的节点树时很有好处。
2. 一些 CSS 机制如 *Flexbox* 和 *CSS Grid* 具有特殊的父子关系，如果在中间添加 div 将使得很难保持所需的结构。
3. 在 DOM 审查器中不会那么的杂乱。

⬆ 返回顶部

48. 在 React 中什么是 Portal ?

Portal 提供了一种很好的将子节点渲染到父组件以外的 DOM 节点的方式。

```
ReactDOM.createPortal(child, container)
```

第一个参数是任何可渲染的 React 子节点，例如元素，字符串或片段。第二个参数是 DOM 元素。

⬆ 返回顶部

49. 什么是无状态组件?

如果行为独立于其状态，则它可以是无状态组件。你可以使用函数或类来创建无状态组件。但除非你需要在组件中使用生命周期钩子，否则你应该选择函数组件。无状态组件有很多好处： 它们易于编写，理解和测试，速度更快，而且你可以完全避免使用 `this` 关键字。

⬆ 返回顶部

50. 什么是有状态组件?

如果组件的行为依赖于组件的 *state* ，那么它可以被称为有状态组件。这些 *有状态组件* 总是 *类组件* ，并且具有在 `constructor` 中初始化的状态。

```
class App extends Component {
  constructor(props) {
    super(props)
    this.state = { count: 0 }
  }

  render() {
    // ...
  }
}
```

⬆ 返回顶部

51. 在 React 中如何校验 props 属性?

当应用程序以开发模式运行的时，React 将会自动检查我们在组件上设置的所有属性，以确保它们具有正确的类型。如果类型不正确，React 将在控制台中生成警告信息。由于性能影响，它在生产模式下被禁用。使用 `isRequired` 定义必填属性。

预定义的 prop 类型：

1. `PropTypes.number`
2. `PropTypes.string`
3. `PropTypes.array`
4. `PropTypes.object`
5. `PropTypes.func`
6. `PropTypes.node`
7. `PropTypes.element`
8. `PropTypes.bool`
9. `PropTypes.symbol`
10. `PropTypes.any`

我们可以为 `User` 组件定义 `propTypes` ，如下所示：

```
import React from 'react'
import PropTypes from 'prop-types'

class User extends React.Component {
  static propTypes = {
    name: PropTypes.string.isRequired,
    age: PropTypes.number.isRequired
  }
```

```
    render() {
      return (
        <>
          <h1>{`Welcome, ${this.props.name}`}</h1>
          <h2>{`Age, ${this.props.age}`}</h2>
        </>
      )
    }
  }
```

注意: 在 React v15.5 中， *PropTypes* 从 `React.PropTypes` 被移动到 `prop-types` 库中。

⬆ 返回顶部

52. React 的优点是什么?

1. 使用 *Virtual DOM* 提高应用程序的性能。
2. JSX 使代码易于读写。
3. 它支持在客户端和服务端渲染。
4. 易于与框架（Angular，Backbone）集成，因为它只是一个视图库。
5. 使用 Jest 等工具轻松编写单元与集成测试。

⬆ 返回顶部

53. React 的局限性是什么?

1. React 只是一个视图库，而不是一个完整的框架。
2. 对于 Web 开发初学者来说，有一个学习曲线。
3. 将 React 集成到传统的 MVC 框架中需要一些额外的配置。
4. 代码复杂性随着内联模板和 JSX 的增加而增加。
5. 如果有太多的小组件可能增加项目的庞大和复杂。

⬆ 返回顶部

54. 在 React v16 中的错误边界是什么?

错误边界是在其子组件树中的任何位置捕获 JavaScript 错误、记录这些错误并显示回退 UI 而不是崩溃的组件树的组件。

如果一个类组件定义了一个名为 `componentDidCatch(error, info)` 或 `static getDerivedStateFromError()` 新的生命周期方法，则该类组件将成为错误边界：

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props)
    this.state = { hasError: false }
  }

  componentDidCatch(error, info) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, info)
  }

  static getDerivedStateFromError(error) {
     // Update state so the next render will show the fallback UI.
```

```
      return { hasError: true };
    }

    render() {
      if (this.state.hasError) {
        // You can render any custom fallback UI
        return <h1>{'Something went wrong.'}</h1>
      }
      return this.props.children
    }
  }
```

之后，将其作为常规组件使用：

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

⬆ 返回顶部

55. 在 React v15 中如何处理错误边界？

React v15 使用 `unstable_handleError` 方法为错误边界提供了非常基础的支持。已在 React v16 中，将其重命名为 `componentDidCatch` 。

⬆ 返回顶部

56. 静态类型检查推荐的方法是什么？

通常，我们使用 PropTypes 库（在 React v15.5 之后 `React.PropTypes` 被移动到了 `prop-types` 包中），在 React 应用程序中执行类型检查。对于大型项目，建议使用静态类型检查器，比如 Flow 或 TypeScript，它们在编译时执行类型检查并提供 auto-completion 功能。

⬆ 返回顶部

57. `react-dom` 包的用途是什么？

`react-dom` 包提供了特定的 DOM 方法，可以在应用程序的顶层使用。大多数的组件不需要使用此模块。该模块中提供的一些方法如下：

1. `render()`
2. `hydrate()`
3. `unmountComponentAtNode()`
4. `findDOMNode()`
5. `createPortal()`

⬆ 返回顶部

58. `react-dom` 中 render 方法的目的是什么？

此方法用于将 React 元素渲染到所提供容器中的 DOM 结构中，并返回对组件的引用。如果 React 元素之前已被渲染到容器中，它将对其执行更新，并且只在需要时改变 DOM 以反映最新的更改。

```
ReactDOM.render(element, container[, callback])
```

如果提供了可选的回调函数，该函数将在组件被渲染或更新后执行。

⬆ 返回顶部

59. ReactDOMServer 是什么?

`ReactDOMServer` 对象使你能够将组件渲染为静态标记（通常用于 Node 服务器中），此对象主要用于服务端渲染（SSR）。以下方法可用于服务器和浏览器环境：

1. `renderToString()`
2. `renderToStaticMarkup()`

例如，你通常运行基于 Node 的 Web 服务器，如 Express，Hapi 或 Koa，然后你调用 `renderToString` 将根组件渲染为字符串，然后作为响应进行发送。

```
// using Express
import { renderToString } from 'react-dom/server'
import MyPage from './MyPage'

app.get('/', (req, res) => {
  res.write('<!DOCTYPE html><html><head><title>My Page</title></head><body>')
  res.write('<div id="content">')
  res.write(renderToString(<MyPage/>))
  res.write('</div></body></html>')
  res.end()
})
```

⬆ 返回顶部

60. 在 React 中如何使用 innerHTML?

`dangerouslySetInnerHTML` 属性是 React 用来替代在浏览器 DOM 中使用 `innerHTML` 。与 `innerHTML` 一样，考虑到跨站脚本攻击（XSS），使用此属性也是有风险的。使用时，你只需传递以 `__html` 作为键，而 HTML 文本作为对应值的对象。

在本示例中 MyComponent 组件使用 `dangerouslySetInnerHTML` 属性来设置 HTML 标记：

```
function createMarkup() {
  return { __html: 'First &middot; Second' }
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />
}
```

⬆ 返回顶部

61. 如何在 React 中使用样式?

`style` 属性接受含有 camelCased（驼峰）属性的 JavaScript 对象，而不是 CSS 字符串。这与 DOM 样式中的 JavaScript 属性一致，效率更高，并且可以防止 XSS 安全漏洞。

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')'
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>
}
```

为了与在 JavaScript 中访问 DOM 节点上的属性保持一致，样式键采用了 camelcased（例如 `node.style.backgroundImage` ）。

⬆ 返回顶部

62. 在 React 中事件有何不同?

处理 React 元素中的事件有一些语法差异：

1. React 事件处理程序是采用驼峰而不是小写来命名的。
2. 使用 JSX，你将传递一个函数作为事件处理程序，而不是字符串。

⬆ 返回顶部

63. 如果在构造函数中使用 `setState()` 会发生什么?

当你使用 `setState()` 时，除了设置状态对象之外，React 还会重新渲染组件及其所有的子组件。你会得到这样的错误： *Can only update a mounted or mounting component.* 。因此我们需要在构造函数中使用 `this.state` 初始化状态。

⬆ 返回顶部

64. 索引作为键的影响是什么?

Keys 应该是稳定的，可预测的和唯一的，这样 React 就能够跟踪元素。

在下面的代码片段中，每个元素的键将基于列表项的顺序，而不是绑定到即将展示的数据上。这将限制 React 能够实现的优化。

```
{todos.map((todo, index) =>
  <Todo
    {...todo}
    key={index}
  />
)}
```

假设 `todo.id` 对此列表是唯一且稳定的，如果将此数据作为唯一键，那么 React 将能够对元素进行重新排序，而无需重新创建它们。

```
{todos.map((todo) =>
  <Todo {...todo}
    key={todo.id} />
)}
```

⬆ 返回顶部

65. 在 `componentWillMount()` 方法中使用 `setState()` 好吗?

建议避免在 `componentWillMount()` 生命周期方法中执行异步初始化。在 mounting 发生之前会立即调用 `componentWillMount()`，且它在 `render()` 之前被调用，因此在此方法中更新状态将不会触发重新渲染。应避免在此方法中引入任何副作用或订阅操作。我们需要确保对组件初始化的异步调用发生在 `componentDidMount()` 中，而不是在 `componentWillMount()` 中。

```
componentDidMount() {
  axios.get(`api/todos`)
    .then((result) => {
      this.setState({
        messages: [...result.data]
      })
    })
}
```

⬆ 返回顶部

66. 如果在初始状态中使用 props 属性会发生什么？

如果在不刷新组件的情况下更改组件上的属性，则不会显示新的属性值，因为构造函数函数永远不会更新组件的当前状态。只有在首次创建组件时才会用 props 属性初始化状态。

以下组件将不显示更新的输入值：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      records: [],
      inputValue: this.props.inputValue
    };
  }

  render() {
    return <div>{this.state.inputValue}</div>
  }
}
```

在 render 方法使用使用 props 将会显示更新的值：

```
```jsx
class MyComponent extends React.Component {
  constructor(props) {
    super(props)

    this.state = {
      record: []
    }
  }

  render() {
    return <div>{this.props.inputValue}</div>
  }
```
```

```
}
```

[⬆ 返回顶部](#目录)

67. 如何有条件地渲染组件?

在某些情况下，你希望根据某些状态渲染不同的组件。 JSX 不会渲染 `false` 或 `undefined`，因此你可以使用 `&&` 运算符，在某个条件为 true 时，渲染组件中指定的内容。

```
const MyComponent = ({ name, address }) => (
  <div>
    <h2>{name}</h2>
    {address &&
      <p>{address}</p>
    }
  </div>
)
```

如果你需要一个 `if-else` 条件，那么使用三元运算符：

```
const MyComponent = ({ name, address }) => (
  <div>
    <h2>{name}</h2>
    {address
      ? <p>{address}</p>
      : <p>{'Address is not available'}</p>
    }
  </div>
)
```

阅读资源：

1. 掘金 - 精读《React 八种条件渲染》

⬆ 返回顶部

68. 为什么在 DOM 元素上展开 props 需要小心?

当我们展开属性时，我们会遇到添加未知 HTML 属性的风险，这是一种不好的做法。相反，我们可以使用属性解构和 `...rest` 运算符，因此它只添加所需的 props 属性。例如，

```
const ComponentA = () =>
  <ComponentB isDisplay={true} className={'componentStyle'} />

const ComponentB = ({ isDisplay, ...domProps }) =>
  <div {...domProps}>{'ComponentB'}</div>
```

⬆ 返回顶部

69. 在 React 中如何使用装饰器?

你可以装饰你的类组件，这与将组件传递到函数中是一样的。 装饰器是修改组件功能灵活且易读的方式。

```
@setTitle('Profile')
class Profile extends React.Component {
    //....
}

/*
  title is a string that will be set as a document title
  WrappedComponent is what our decorator will receive when
  put directly above a component class as seen in the example above
*/
const setTitle = (title) => (WrappedComponent) => {
  return class extends React.Component {
    componentDidMount() {
      document.title = title
    }

    render() {
      return <WrappedComponent {...this.props} />
    }
  }
}
```

⬆ 返回顶部

70. 如何 memoize（记忆）组件?

有可用于函数组件的 memoize 库。例如 `moize` 库可以将组件存储在另一个组件中。

```
import moize from 'moize'
import Component from './components/Component' // this module exports a non-memoized component

const MemoizedFoo = moize.react(Component)

const Consumer = () => {
  <div>
    {'I will memoize the following entry:'}
    <MemoizedFoo/>
  </div>
}
```

⬆ 返回顶部

71. 如何实现 Server Side Rendering 或 SSR?

React 已经配备了用于处理 Node 服务器上页面渲染的功能。你可以使用特殊版本的 DOM 渲染器，它遵循与客户端相同的模式。

```
import ReactDOMServer from 'react-dom/server'
import App from './App'

ReactDOMServer.renderToString(<App />)
```

此方法将以字符串形式输出常规 HTML，然后将其作为服务器响应的一部分放在页面正文中。在客户端，React 检测预渲染的内容并无缝地衔接。

⬆ 返回顶部

72. 如何在 React 中启用生产模式?

你应该使用 Webpack 的 `DefinePlugin` 方法将 `NODE_ENV` 设置为 `production`，通过它你可以去除 propType 验证和额外警告等内容。除此之外，如果你压缩代码，如使用 Uglify 的死代码消除，以去掉用于开发的代码和注释，它将大大减少包的大小。

⬆ 返回顶部

73. 什么是 CRA 及其好处?

`create-react-app` CLI 工具允许你无需配置步骤，快速创建和运行 React 应用。

让我们使用 *CRA* 来创建 Todo 应用：

```
# Installation
$ npm install -g create-react-app

# Create new project
$ create-react-app todo-app
$ cd todo-app

# Build, test and run
$ npm run build
$ npm run test
$ npm start
```

它包含了构建 React 应用程序所需的一切：

1. React, JSX, ES6, 和 Flow 语法支持。
2. ES6 之外的语言附加功能，比如对象扩展运算符。
3. Autoprefixed CSS，因此你不在需要 -webkit- 或其他前缀。
4. 一个快速的交互式单元测试运行程序，内置了对覆盖率报告的支持。
5. 一个实时开发服务器，用于警告常见错误。
6. 一个构建脚本，用于打包用于生产中包含 hashes 和 sourcemaps 的 JS、CSS 和 Images 文件。

⬆ 返回顶部

74. 在 mounting 阶段生命周期方法的执行顺序是什么?

在创建组件的实例并将其插入到 DOM 中时，将按以下顺序调用生命周期方法。

1. `constructor()`
2. `static getDerivedStateFromProps()`
3. `render()`
4. `componentDidMount()`

⬆ 返回顶部

75. 在 React v16 中，哪些生命周期方法将被弃用?

以下生命周期方法将成为不安全的编码实践，并且在异步渲染方面会更有问题。

1. `componentWillMount()`

2. `componentWillReceiveProps()`
3. `componentWillUpdate()`

从 React v16.3 开始，这些方法使用 `UNSAFE_` 前缀作为别名，未加前缀的版本将在 React v17 中被移除。

⬆ 返回顶部

76. 生命周期方法 `getDerivedStateFromProps()` 的目的是什么?

新的静态 `getDerivedStateFromProps()` 生命周期方法在实例化组件之后以及重新渲染组件之前调用。它可以返回一个对象用于更新状态，或者返回 `null` 指示新的属性不需要任何状态更新。

```
class MyComponent extends React.Component {
  static getDerivedStateFromProps(props, state) {
    // ...
  }
}
```

此生命周期方法与 `componentDidUpdate()` 一起涵盖了 `componentWillReceiveProps()` 的所有用例。

⬆ 返回顶部

77. 生命周期方法 `getSnapshotBeforeUpdate()` 的目的是什么?

新的 `getSnapshotBeforeUpdate()` 生命周期方法在 DOM 更新之前被调用。此方法的返回值将作为第三个参数传递给 `componentDidUpdate()` 。

```
class MyComponent extends React.Component {
  getSnapshotBeforeUpdate(prevProps, prevState) {
    // ...
  }
}
```

此生命周期方法与 `componentDidUpdate()` 一起涵盖了 `componentWillUpdate()` 的所有用例。

⬆ 返回顶部

78. createElement() 和 cloneElement() 方法有什么区别?

JSX 元素将被转换为 `React.createElement()` 函数来创建 React 元素，这些对象将用于表示 UI 对象。而 `cloneElement` 用于克隆元素并传递新的属性。

⬆ 返回顶部

79. 推荐的组件命名方法是什么?

建议通过引用命名组件，而不是使用 `displayName` 。

使用 `displayName` 命名组件:

```
export default React.createClass({
  displayName: 'TodoApp',
  // ...
})
```

推荐的方式：

```
export default class TodoApp extends React.Component {
  // ...
}
```

⬆ 返回顶部

80. 在组件类中方法的推荐顺序是什么？

从 *mounting* 到 *render stage* 阶段推荐的方法顺序：

1. `static` 方法
2. `constructor()`
3. `getChildContext()`
4. `componentWillMount()`
5. `componentDidMount()`
6. `componentWillReceiveProps()`
7. `shouldComponentUpdate()`
8. `componentWillUpdate()`
9. `componentDidUpdate()`
10. `componentWillUnmount()`
11. 点击处理程序或事件处理程序，如 `onClickSubmit()` 或 `onChangeDescription()`
12. 用于渲染的getter方法，如 `getSelectReason()` 或 `getFooterContent()`
13. 可选的渲染方法，如 `renderNavigation()` 或 `renderProfilePicture()`
14. `render()`

⬆ 返回顶部

81. 什么是 switching 组件？

switching 组件是渲染多个组件之一的组件。我们需要使用对象将 prop 映射到组件中。

例如，以下的 switching 组件将基于 `page` 属性显示不同的页面：

```
import HomePage from './HomePage'
import AboutPage from './AboutPage'
import ServicesPage from './ServicesPage'
import ContactPage from './ContactPage'

const PAGES = {
  home: HomePage,
  about: AboutPage,
  services: ServicesPage,
  contact: ContactPage
}

const Page = (props) => {
```

```
  const Handler = PAGES[props.page] || ContactPage

  return <Handler {...props} />
}

// The keys of the PAGES object can be used in the prop types to catch dev-time
errors.
Page.propTypes = {
  page: PropTypes.oneOf(Object.keys(PAGES)).isRequired
}
```

⬆ 返回顶部

82. 为什么我们需要将函数传递给 setState() 方法?

这背后的原因是 `setState()` 是一个异步操作。出于性能原因，React 会对状态更改进行批处理，因此在调用 `setState()` 方法之后，状态可能不会立即更改。这意味着当你调用 `setState()` 方法时，你不应该依赖当前状态，因为你不能确定当前状态应该是什么。这个问题的解决方案是将一个函数传递给 `setState()` ，该函数会以上一个状态作为参数。通过这样做，你可以避免由于 `setState()` 的异步性质而导致用户在访问时获取旧状态值的问题。

假设初始计数值为零。在连续三次增加操作之后，该值将只增加一个。

```
// assuming this.state.count === 0
this.setState({ count: this.state.count + 1 })
this.setState({ count: this.state.count + 1 })
this.setState({ count: this.state.count + 1 })
// this.state.count === 1, not 3
```

如果将函数传递给 `setState()` ，则 count 将正确递增。

```
this.setState((prevState, props) => ({
  count: prevState.count + props.increment
}))
// this.state.count === 3 as expected
```

⬆ 返回顶部

83. 在 React 中什么是严格模式?

`React.StrictMode` 是一个有用的组件，用于突出显示应用程序中的潜在问题。就像 `<Fragment>` ， `<StrictMode>` 一样，它们不会渲染任何额外的 DOM 元素。它为其后代激活额外的检查和警告。这些检查仅适用于开发模式。

```
import React from 'react'

function ExampleApplication() {
  return (
    <div>
      <Header />
      <React.StrictMode>
        <div>
          <ComponentOne />
```

```
            <ComponentTwo />
          </div>
        </React.StrictMode>
        <Footer />
      </div>
    )
  }
```

在上面的示例中，*strict mode* 检查仅应用于 `<ComponentOne>` 和 `<ComponentTwo>` 组件。

⬆ 返回顶部

84. React Mixins 是什么?

Mixins 是一种完全分离组件通用功能的方法。 Mixins 不应该被继续使用，可以用高阶组件或装饰器来替换。

最常用的 mixins 是 `PureRenderMixin` 。当 props 和状态与之前的 props 和状态相等时，你可能在某些组件中使用它来防止不必要的重新渲染：

```
const PureRenderMixin = require('react-addons-pure-render-mixin')

const Button = React.createClass({
  mixins: [PureRenderMixin],
  // ...
})
```

⬆ 返回顶部

85. 为什么 `isMounted()` 是一个反模式，而正确的解决方案是什么?

`isMounted()` 的主要场景是避免在组件卸载后调用 `setState()` ，因为它会发出警告。

```
if (this.isMounted()) {
  this.setState({...})
}
```

在调用 `setState()` 之前检查 `isMounted()` 会消除警告，但也会破坏警告的目的。使用 `isMounted()` 有一种代码味道，因为你要检查的唯一原因是你认为在卸载组件后可能持有引用。

最佳解决方案是找到在组件卸载后调用 `setState()` 的位置，并修复它们。这种情况最常发生在回调中，即组件正在等待某些数据并在数据到达之前卸载。理想情况下，在卸载之前，应在 `componentWillUnmount()` 中取消任何回调。

⬆ 返回顶部

86. React 中支持哪些指针事件?

Pointer Events 提供了处理所有输入事件的统一方法。在过去，我们有一个鼠标和相应的事件监听器来处理它们，但现在我们有许多与鼠标无关的设备，比如带触摸屏的手机或笔。我们需要记住，这些事件只能在支持 *Pointer Events* 规范的浏览器中工作。

目前以下事件类型在 *React DOM* 中是可用的：

1. `onPointerDown`

2. `onPointerMove`
3. `onPointerUp`
4. `onPointerCancel`
5. `onGotPointerCapture`
6. `onLostPointerCaptur`
7. `onPointerEnter`
8. `onPointerLeave`
9. `onPointerOver`
10. `onPointerOut`

⬆ 返回顶部

87. 为什么组件名称应该以大写字母开头?

如果使用 JSX 渲染组件，则该组件的名称必须以大写字母开头，否则 React 将会抛出无法识别标签的错误。这种约定是因为只有 HTML 元素和 SVG 标签可以以小写字母开头。

定义组件类的时候，你可以以小写字母开头，但在导入时应该使用大写字母。

```
class myComponent extends Component {
  render() {
    return <div />
  }
}

export default myComponent
```

当在另一个文件导入时，应该以大写字母开头：

```
import MyComponent from './MyComponent'
```

⬆ 返回顶部

88. 在 React v16 中是否支持自定义 DOM 属性?

是的，在过去 React 会忽略未知的 DOM 属性。如果你编写的 JSX 属性 React 无法识别，那么 React 将跳过它。例如：

```
<div mycustomattribute={'something'} />
```

在 React 15 中将在 DOM 中渲染一个空的 div：

```
<div />
```

在 React 16 中，任何未知的属性都将会在 DOM 显示：

```
<div mycustomattribute='something' />
```

这对于应用特定于浏览器的非标准属性，尝试新的 DOM APIs 与集成第三方库来说非常有用。

⬆ 返回顶部

89. constructor 和 getInitialState 有什么区别?

当使用 ES6 类时，你应该在构造函数中初始化状态，而当你使用 `React.createClass()` 时，就需要使用 `getInitialState()` 方法。

使用 ES6 类:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state = { /* initial state */ }
  }
}
```

使用 `React.createClass()`:

```
const MyComponent = React.createClass({
  getInitialState() {
    return { /* initial state */ }
  }
})
```

注意： 在 React v16 中 `React.createClass()` 已被弃用和删除，请改用普通的 JavaScript 类。

⬆ 返回顶部

90. 是否可以在不调用 setState 方法的情况下，强制组件重新渲染?

默认情况下，当组件的状态或属性改变时，组件将重新渲染。如果你的 `render()` 方法依赖于其他数据，你可以通过调用 `forceUpdate()` 来告诉 React，当前组件需要重新渲染。

```
component.forceUpdate(callback)
```

建议避免使用 `forceUpdate()`，并且只在 `render()` 方法中读取 `this.props` 和 `this.state`。

⬆ 返回顶部

91. 在使用 ES6 类的 React 中 `super()` 和 `super(props)` 有什么区别?

当你想要在 `constructor()` 函数中访问 `this.props`，你需要将 props 传递给 `super()` 方法。

使用 `super(props)`:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    console.log(this.props) // { name: 'John', ... }
  }
}
```

使用 `super()`:

```
class MyComponent extends React.Component {
  constructor(props) {
    super()
    console.log(this.props) // undefined
  }
}
```

在 `constructor()` 函数之外，访问 `this.props` 属性会显示相同的值。

阅读资源：

1. 为什么我们要写 super(props)?

⬆ 返回顶部

92. 在 JSX 中如何进行循环?

你只需使用带有 ES6 箭头函数语法的 `Array.prototype.map` 即可。例如， `items` 对象数组将会被映射成一个组件数组：

```
<tbody>
  {items.map(item => <SomeComponent key={item.id} name={item.name} />)}
</tbody>
```

你不能使用 `for` 循环进行迭代：

```
<tbody>
  for (let i = 0; i < items.length; i++) {
    <SomeComponent key={items[i].id} name={items[i].name} />
  }
</tbody>
```

这是因为 JSX 标签会被转换成函数调用，并且你不能在表达式中使用语句。但这可能会由于 `do` 表达式而改变，它们是第一阶段提案。

⬆ 返回顶部

93. 如何在 attribute 引号中访问 props 属性?

React (或 JSX) 不支持属性值内的变量插值。下面的形式将不起作用：

```
<img className='image' src='images/{this.props.image}' />
```

但你可以将 JS 表达式作为属性值放在大括号内。所以下面的表达式是有效的：

```
<img className='image' src={'images/' + this.props.image} />
```

使用模板字符串也是可以的：

```
<img className='image' src={`images/${this.props.image}`} />
```

⬆ 返回顶部

94. 什么是 React proptype 数组?

如果你要规范具有特定对象格式的数组的属性，请使用 `React.PropTypes.shape()` 作为 `React.PropTypes.arrayOf()` 的参数。

```
ReactComponent.propTypes = {
  arrayWithShape: React.PropTypes.arrayOf(React.PropTypes.shape({
    color: React.PropTypes.string.isRequired,
    fontSize: React.PropTypes.number.isRequired
  })).isRequired
}
```

⬆ 返回顶部

95. 如何有条件地应用样式类?

你不应该在引号内使用大括号，因为它将被计算为字符串。

```
<div className="btn-panel {this.props.visible ? 'show' : 'hidden'}">
```

相反，你需要将大括号移到外部（不要忘记在类名之间添加空格）：

```
<div className={'btn-panel ' + (this.props.visible ? 'show' : 'hidden')}>
```

模板字符串也可以工作：

```
<div className={`btn-panel ${this.props.visible ? 'show' : 'hidden'}`}>
```

⬆ 返回顶部

96. React 和 ReactDOM 之间有什么区别?

`react` 包中包含 `React.createElement()`，`React.Component`，`React.Children`，以及与元素和组件类相关的其他帮助程序。你可以将这些视为构建组件所需的同构或通用帮助程序。`react-dom` 包中包含了 `ReactDOM.render()`，在 `react-dom/server` 包中有支持服务端渲染的 `ReactDOMServer.renderToString()` 和 `ReactDOMServer.renderToStaticMarkup()` 方法。

⬆ 返回顶部

97. 为什么 ReactDOM 从 React 分离出来?

React 团队致力于将所有的与 DOM 相关的特性抽取到一个名为 ReactDOM 的独立库中。React v0.14 是第一个拆分后的版本。通过查看一些软件包，`react-native`，`react-art`，`react-canvas`，和 `react-three`，很明显，React 的优雅和本质与浏览器或 DOM 无关。为了构建更多 React 能应用的环境，React 团队计划将主要的 React 包拆分成两个：`react` 和 `react-dom`。这为编写可以在 React 和 React Native 的 Web 版本之间共享的组件铺平了道路。

⬆ 返回顶部

98. 如何使用 React label 元素?

如果你尝试使用标准的 `for` 属性将 `<label>` 元素绑定到文本输入框，那么在控制台将会打印缺少 HTML 属性的警告消息。

```
<label for={'user'}>{'User'}</label>
<input type={'text'} id={'user'} />
```

因为 `for` 是 JavaScript 的保留字，请使用 `htmlFor` 来替代。

```
<label htmlFor={'user'}>{'User'}</label>
<input type={'text'} id={'user'} />
```

⬆ 返回顶部

99. 如何合并多个内联的样式对象?

在 React 中，你可以使用扩展运算符:

```
<button style={{...styles.panel.button, ...styles.panel.submitButton}}>
{'Submit'}</button>
```

如果你使用的是 React Native，则可以使用数组表示法：

```
<button style={[styles.panel.button, styles.panel.submitButton]}>{'Submit'}
</button>
```

⬆ 返回顶部

100. 如何在调整浏览器大小时重新渲染视图?

你可以在 `componentDidMount()` 中监听 `resize` 事件，然后更新尺寸（`width` 和 `height`）。你应该在 `componentWillUnmount()` 方法中移除监听。

```
class WindowDimensions extends React.Component {
  componentWillMount() {
    this.updateDimensions()
  }

  componentDidMount() {
    window.addEventListener('resize', this.updateDimensions)
  }

  componentWillUnmount() {
    window.removeEventListener('resize', this.updateDimensions)
  }

  updateDimensions() {
    this.setState({width: $(window).width(), height: $(window).height()})
  }

  render() {
    return <span>{this.state.width} x {this.state.height}</span>
  }
}
```

⬆ 返回顶部

101. `setState()` 和 `replaceState()` 方法之间有什么区别?

当你使用 `setState()` 时，当前和先前的状态将被合并。 `replaceState()` 会抛出当前状态，并仅用你提供的内容替换它。通常使用 `setState()` ，除非你出于某种原因确实需要删除所有以前的键。你还可以在 `setState()` 中将状态设置为 `false` / `null` ，而不是使用 `replaceState()` 。

⬆ 返回顶部

102. 如何监听状态变化?

当状态更改时将调用以下生命周期方法。你可以将提供的状态和属性值与当前状态和属性值进行比较，以确定是否发生了有意义的改变。

```
componentWillUpdate(object nextProps, object nextState)
componentDidUpdate(object prevProps, object prevState)
```

⬆ 返回顶部

103. 在 React 状态中删除数组元素的推荐方法是什么?

更好的方法是使用 `Array.prototype.filter()` 方法。

例如，让我们创建用于更新状态的 `removeItem()` 方法。

```
removeItem(index) {
  this.setState({
    data: this.state.data.filter((item, i) => i !== index)
  })
}
```

⬆ 返回顶部

104. 在 React 中是否可以不在页面上渲染 HTML 内容?

可以使用最新的版本 (>=16.2)，以下是可能的选项：

```
render() {
  return false
}
```

```
render() {
  return null
}
```

```
render() {
  return []
}
```

```
render() {
  return <React.Fragment></React.Fragment>
}
```

```
render() {
  return <></>
}
```

返回 `undefined` 是无效的。

⬆ 返回顶部

105. 如何用 React 漂亮地显示 JSON?

我们可以使用 `<pre>` 标签，以便保留 `JSON.stringify()` 的格式：

````
```jsx
const data = { name: 'John', age: 42 }

class User extends React.Component {
  render() {
    return (
      <pre>
        {JSON.stringify(data, null, 2)}
      </pre>
    )
  }
}

React.render(<User />, document.getElementById('container'))
```

**[⬆ 返回顶部](#目录)**
````

106. 为什么你不能更新 React 中的 props?

React 的哲学是 props 应该是 *immutable* 和 *top-down* 。这意味着父级可以向子级发送任何属性值，但子级不能修改接收到的属性。

⬆ 返回顶部

107. 如何在页面加载时聚焦一个输入元素?

你可以为 `input` 元素创建一个 `ref`，然后在 `componentDidMount()` 方法中使用它：

```jsx
class App extends React.Component{
  componentDidMount() {
    this.nameInput.focus()
  }

  render() {
    return (
      <div>
        <input
          defaultValue={'Won\'t focus'}
        />
        <input
          ref={(input) => this.nameInput = input}
          defaultValue={'Will focus'}
        />
      </div>
    )
  }
```

```
}

ReactDOM.render(<App />, document.getElementById('app'))
```

⬆ 返回顶部

108. 更新状态中的对象有哪些可能的方法?

1. 用一个对象调用 `setState()` 来与状态合并:
   - 使用 `Object.assign()` 创建对象的副本:

     ```
     const user = Object.assign({}, this.state.user, { age: 42 })
     this.setState({ user })
     ```

   - 使用扩展运算符:

     ```
     const user = { ...this.state.user, age: 42 }
     this.setState({ user })
     ```

2. 使用一个函数调用 `setState()`:

```
this.setState(prevState => ({
  user: {
    ...prevState.user,
    age: 42
  }
}))
```

⬆ 返回顶部

109. 为什么函数比对象更适合于 `setState()`?

出于性能考虑，React 可能将多个 `setState()` 调用合并成单个更新。这是因为我们可以异步更新 `this.props` 和 `this.state`，所以不应该依赖它们的值来计算下一个状态。

以下的 counter 示例将无法按预期更新:

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
})
```

首选方法是使用函数而不是对象调用 `setState()`。该函数将前一个状态作为第一个参数，当前时刻的 props 作为第二个参数。

```
// Correct
this.setState((prevState, props) => ({
  counter: prevState.counter + props.increment
}))
```

⬆ 返回顶部

110. 我们如何在浏览器中找到当前正在运行的 React 版本?

你可以使用 `React.version` 来获取版本:

```
const REACT_VERSION = React.version

ReactDOM.render(
  <div>{`React version: ${REACT_VERSION}`}</div>,
  document.getElementById('app')
)
```

⬆ 返回顶部

111. 在 `create-react-app` 项目中导入 polyfills 的方法有哪些?

1. 从 `core-js` 中手动导入:

   创建一个名为 `polyfills.js` 文件，并在根目录下的 `index.js` 文件中导入它。运行 `npm install core-js` 或 `yarn add core-js` 并导入你所需的功能特性:

   ```
   import 'core-js/fn/array/find'
   import 'core-js/fn/array/includes'
   import 'core-js/fn/number/is-nan'
   ```

2. 使用 Polyfill 服务:

   通过将以下内容添加到 `index.html` 中来获取自定义的特定于浏览器的 polyfill:

   ```
   <script src='https://cdn.polyfill.io/v2/polyfill.min.js?
   features=default,Array.prototype.includes'></script>
   ```

   在上面的脚本中，我们必须显式地请求 `Array.prototype.includes` 特性，因为它没有被包含在默认的特性集中。

⬆ 返回顶部

112. 如何在 create-react-app 中使用 https 而不是 http?

你只需要使用 `HTTPS=true` 配置。你可以编辑 `package.json` 中的 scripts 部分:

```
"scripts": {
  "start": "set HTTPS=true && react-scripts start"
}
```

或直接运行 `set HTTPS=true && npm start`

⬆ 返回顶部

113. 如何避免在 create-react-app 中使用相对路径导入?

在项目的根目录中创建一个名为 `.env` 的文件，并写入导入路径:

```
NODE_PATH=src/app
```

然后重新启动开发服务器。现在，你应该能够在没有相对路径的情况下导入 `src/app` 内的任何内容。

⬆ 返回顶部

114. 如何为 React Router 添加 Google Analytics?

在 `history` 对象上添加一个监听器以记录每个页面的访问：

```
history.listen(function (location) {
  window.ga('set', 'page', location.pathname + location.search)
  window.ga('send', 'pageview', location.pathname + location.search)
})
```

⬆ 返回顶部

115. 如何每秒更新一个组件?

你需要使用 `setInterval()` 来触发更改，但也需要在组件卸载时清除计时器，以防止错误和内存泄漏。

```
componentDidMount() {
  this.interval = setInterval(() => this.setState({ time: Date.now() }), 1000)
}

componentWillUnmount() {
  clearInterval(this.interval)
}
```

⬆ 返回顶部

116. 如何将 vendor prefixes 应用于 React 中的内联样式?

React不会自动应用 *vendor prefixes* ，你需要手动添加 *vendor prefixes* 。

```
<div style={{
  transform: 'rotate(90deg)',
  WebkitTransform: 'rotate(90deg)', // note the capital 'W' here
  msTransform: 'rotate(90deg)' // 'ms' is the only lowercase vendor prefix
}} />
```

⬆ 返回顶部

117. 如何使用 React 和 ES6 导入和导出组件?

导出组件时，你应该使用默认导出：

```
import React from 'react'
import User from 'user'

export default class MyProfile extends React.Component {
  render(){
    return (
      <User type="customer">
        //...
      </User>
    )
  }
}
```

使用 export 说明符，MyProfile 将成为成员并导出到此模块，此外在其他组件中你无需指定名称就可以导入相同的内容。

⬆ 返回顶部

118. 为什么 React 组件名称必须以大写字母开头?

在 JSX 中，小写标签被认为是 HTML 标签。但是，含有 `.` 的大写和小写标签名却不是。

1. `<component />` 将被转换为 `React.createElement('component')` (i.e, HTML 标签)
2. `<obj.component />` 将被转换为 `React.createElement(obj.component)`
3. `<Component />` 将被转换为 `React.createElement(Component)`

⬆ 返回顶部

119. 为什么组件的构造函数只被调用一次?

React 协调算法假设如果自定义组件出现在后续渲染的相同位置，则它与之前的组件相同，因此重用前一个实例而不是创建新实例。

⬆ 返回顶部

120. 在 React 中如何定义常量?

你可以使用 ES7 的 `static` 来定义常量。

```
class MyComponent extends React.Component {
  static DEFAULT_PAGINATION = 10
}
```

⬆ 返回顶部

121. 在 React 中如何以编程方式触发点击事件?

你可以使用 ref 属性通过回调函数获取对底层的 `HTMLinputeElement` 对象的引用，并将该引用存储为类属性，之后你就可以利用该引用在事件回调函数中， 使用 `HTMLElement.click` 方法触发一个点击事件。这可以分为两个步骤：

1. 在 render 方法创建一个 ref：

```
<input ref={input => this.inputElement = input} />
```

2. 在事件处理器中触发点击事件

```
this.inputElement.click()
```

⬆ 返回顶部

122. 在 React 中是否可以使用 async/await?

如果要在 React 中使用 `async` / `await` ，则需要 *Babel* 和 transform-async-to-generator 插件。

⬆ 返回顶部

123. React 项目常见的文件结构是什么?

React 项目文件结构有两种常见的实践。

1. **按功能或路由分组:**

   构建项目的一种常见方法是将 CSS，JS 和测试用例放在一起，按功能或路由分组。

```
common/
├─ Avatar.js
├─ Avatar.css
├─ APIUtils.js
└─ APIUtils.test.js
feed/
├─ index.js
├─ Feed.js
├─ Feed.css
├─ FeedStory.js
├─ FeedStory.test.js
└─ FeedAPI.js
profile/
├─ index.js
├─ Profile.js
├─ ProfileHeader.js
├─ ProfileHeader.css
└─ ProfileAPI.js
```

2. **按文件类型分组:**

   另一种流行的项目结构组织方法是将类似的文件组合在一起。

```
api/
├─ APIUtils.js
├─ APIUtils.test.js
├─ ProfileAPI.js
└─ UserAPI.js
components/
├─ Avatar.js
├─ Avatar.css
├─ Feed.js
├─ Feed.css
├─ FeedStory.js
├─ FeedStory.test.js
├─ Profile.js
├─ ProfileHeader.js
└─ ProfileHeader.css
```

⬆ 返回顶部

124. 最流行的动画软件包是什么?

React Transition Group 和 *React Motion* 是React生态系统中流行的动画包。

⬆ 返回顶部

125. 模块化样式文件有什么好处?

建议避免在组件中对样式值进行硬编码。任何可能在不同的 UI 组件之间使用的值都应该提取到它们自己的模块中。

例如，可以将这些样式提取到单独的组件中：

```
export const colors = {
  white,
  black,
  blue
}

export const space = [
  0,
  8,
  16,
  32,
  64
]
```

然后在其他组件中单独导入：

```
import { space, colors } from './styles'
```

⬆ 返回顶部

126. 什么是 React 流行的特定 linters?

ESLint 是一个流行的 JavaScript linter。有一些插件可以分析特定的代码样式。在 React 中最常见的一个是名为 `eslint-plugin-react` npm 包。默认情况下，它将使用规则检查许多最佳实践，检查内容从迭代器中的键到一组完整的 prop 类型。另一个流行的插件是 `eslint-plugin-jsx-a11y`，它将帮助修复可访问性的常见问题。由于 JSX 提供的语法与常规 HTML 略有不同，因此常规插件无法获取 `alt` 文本和 `tabindex` 的问题。

⬆ 返回顶部

127. 如何发起 AJAX 调用以及应该在哪些组件生命周期方法中进行 AJAX 调用?

你可以使用 AJAX 库，如 Axios，jQuery AJAX 和浏览器内置的 `fetch` API。你应该在 `componentDidMount()` 生命周期方法中获取数据。这样当获取到数据的时候，你就可以使用 `setState()` 方法来更新你的组件。

例如，从 API 中获取员工列表并设置本地状态：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      employees: [],
      error: null
    }
  }

  componentDidMount() {
    fetch('https://api.example.com/items')
      .then(res => res.json())
      .then(
        (result) => {
```

```
            this.setState({
              employees: result.employees
            })
          },
          (error) => {
            this.setState({ error })
          }
        )
    }

    render() {
      const { error, employees } = this.state
      if (error) {
        return <div>Error: {error.message}</div>;
      } else {
        return (
          <ul>
            {employees.map(item => (
              <li key={employee.name}>
                {employee.name}-{employees.experience}
              </li>
            ))}
          </ul>
        )
      }
    }
  }
```

⬆ 返回顶部

128. 什么是渲染属性?

Render Props 是一种简单的技术，用于使用值为函数的 prop 属性在组件之间共享代码。下面的组件使用返回 React 元素的 render 属性：

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

像 React Router 和 DownShift 这样的库使用了这种模式。

⬆ 返回顶部

React Router

129. 什么是 React Router?

React Router 是一个基于 React 之上的强大路由库，可以帮助您快速地向应用添加视图和数据流，同时保持 UI 与 URL 同步。

⬆ 返回顶部

130. React Router 与 history 库的区别?

React Router 是 `history` 库的包装器，它处理浏览器的 `window.history` 与浏览器和哈希历史的交互。它还提供了内存历史记录，这对于没有全局历史记录的环境非常有用，例如移动应用程序开发（React Native）和使用 Node 进行单元测试。

⬆ 返回顶部

131. 在 React Router v4 中的 `<Router>` 组件是什么?

React Router v4 提供了以下三种类型的 `<Router>` 组件:

1. `<BrowserRouter>`
2. `<HashRouter>`
3. `<MemoryRouter>`

以上组件将创建 *browser*， *hash* 和 *memory* 的 history 实例。React Router v4 通过 `router` 对象中的上下文使与您的路由器关联的 `history` 实例的属性和方法可用。

⬆ 返回顶部

132. `history` 中的 `push()` 和 `replace()` 方法的目的是什么?

一个 history 实例有两种导航方法:

1. `push()`
2. `replace()`

如果您将 history 视为一个访问位置的数组，则 `push()` 将向数组添加一个新位置， `replace()` 将用新的位置替换数组中的当前位置。

⬆ 返回顶部

133. 如何使用在 React Router v4 中以编程的方式进行导航?

在组件中实现操作路由/导航有三种不同的方法。

1. **使用 `withRouter()` 高阶函数:**

   `withRouter()` 高阶函数将注入 history 对象作为组件的 prop。该对象提供了 `push()` 和 `replace()` 方法，以避免使用上下文。

```
import { withRouter } from 'react-router-dom' // this also works with
'react-router-native'

const Button = withRouter(({ history }) => (
  <button
    type='button'
    onClick={() => { history.push('/new-location') }}
  >
    {'Click Me!'}
  </button>
))
```

2. **使用 `<Route>` 组件和渲染属性模式:**

   `<Route>` 组件传递与 `withRouter()` 相同的属性，因此您将能够通过 history 属性访问到操作历史记录的方法。

```
import { Route } from 'react-router-dom'

const Button = () => (
  <Route render={({ history }) => (
    <button
      type='button'
      onClick={() => { history.push('/new-location') }}
    >
      {'Click Me!'}
    </button>
  )} />
)
```

3. **使用上下文：**

   建议不要使用此选项，并将其视为不稳定的API。

```
const Button = (props, context) => (
  <button
    type='button'
    onClick={() => {
      context.history.push('/new-location')
    }}
  >
    {'Click Me!'}
  </button>
)

Button.contextTypes = {
  history: React.PropTypes.shape({
    push: React.PropTypes.func.isRequired
  })
}
```

⬆ 返回顶部

134. 如何在 React Router v4 中获取查询字符串参数?

在 React Router v4 中并没有内置解析查询字符串的能力，因为多年来一直有用户希望支持不同的实现。因此，使用者可以选择他们喜欢的实现方式。建议的方法是使用 query-string 库。

```
const queryString = require('query-string');
const parsed = queryString.parse(props.location.search);
```

如果你想要使用原生 API 的话，你也可以使用 `URLSearchParams` ：

```
const params = new URLSearchParams(props.location.search)
const foo = params.get('name')
```

如果使用 `URLSearchParams` 的话您应该为 IE11 使用 *polyfill* 。

⬆ 返回顶部

135. 为什么你会得到 "Router may have only one child element" 警告?

此警告的意思是 `Router` 组件下仅能包含一个子节点。

你必须将你的 Route 包装在 `<Switch>` 块中，因为 `<Switch>` 是唯一的，它只提供一个路由。

首先，您需要在导入中添加 `Switch` ：

```
import { Switch, Router, Route } from 'react-router'
```

然后在 `<Switch>` 块中定义路由：

```
<Router>
  <Switch>
    <Route {/* ... */} />
    <Route {/* ... */} />
  </Switch>
</Router>
```

⬆ 返回顶部

136. 如何在 React Router v4 中将 params 传递给 `history.push` 方法?

在导航时，您可以将 props 传递给 `history` 对象：

```
this.props.history.push({
  pathname: '/template',
  search: '?name=sudheer',
  state: { detail: response.data }
})
```

`search` 属性用于在 `push()` 方法中传递查询参数。

⬆ 返回顶部

137. 如何实现默认页面或 404 页面?

`<Switch>` 呈现匹配的第一个孩子 `<Route>` 。 没有路径的 `<Route>` 总是匹配。所以你只需要简单地删除 path 属性，如下所示：

```
<Switch>
  <Route exact path="/" component={Home}/>
  <Route path="/user" component={User}/>
  <Route component={NotFound} />
</Switch>
```

⬆ 返回顶部

138. 如何在 React Router v4 上获取历史对象?

1. 创建一个导出 `history` 对象的模块，并在整个项目中导入该模块。

   例如， 创建 `history.js` 文件:

```
import { createBrowserHistory } from 'history'

export default createBrowserHistory({
  /* pass a configuration object here if needed */
})
```

2. 您应该使用 `<Router>` 组件而不是内置路由器。在 `index.js` 文件中导入上面的 `history.js`：

```
import { Router } from 'react-router-dom'
import history from './history'
import App from './App'

ReactDOM.render((
  <Router history={history}>
    <App />
  </Router>
), holder)
```

3. 您还可以使用类似于内置历史对象的 `history` 对象的push方法：

```
// some-other-file.js
import history from './history'

history.push('/go-here')
```

⬆ 返回顶部

139. 登录后如何执行自动重定向?

`react-router` 包在 React Router 中提供了 `<Redirect>` 组件。渲染 `<Redirect>` 将导航到新位置。与服务器端重定向一样，新位置将覆盖历史堆栈中的当前位置。

```
import React, { Component } from 'react'
import { Redirect } from 'react-router'

export default class LoginComponent extends Component {
  render() {
    if (this.state.isLoggedIn === true) {
      return <Redirect to="/your/redirect/page" />
    } else {
      return <div>{'Login Please'}</div>
    }
  }
}
```

⬆ 返回顶部

React Internationalization

140. 什么是 React Intl?

React Intl 库使 React 中的内部化变得简单，使用现成的组件和 API ，可以处理从格式化字符串，日期和数字到复数的所有功能。React Intl 是 FormatJS 的一部分，它通过其组件和 API 提供与 React 的绑定。

⬆ 返回顶部

141. React Intl 的主要特性是什么?

1. 用分隔符显示数字
2. 正确显示日期和时间
3. 显示相对于“现在”的日期
4. 将标签转换为字符串
5. 支持 150 多种语言
6. 支持在浏览器和 Node 中运行
7. 建立在标准之上

⬆ 返回顶部

142. 在 React Intl 中有哪两种格式化方式?

该库提供了两种格式化字符串，数字和日期的方法：React 组件或 API。

```
<FormattedMessage
  id={'account'}
  defaultMessage={'The amount is less than minimum balance.'}
/>
```

```
const messages = defineMessages({
  accountMessage: {
    id: 'account',
    defaultMessage: 'The amount is less than minimum balance.',
  }
})

formatMessage(messages.accountMessage)
```

⬆ 返回顶部

143. 在 React Intl 中如何使用 `<FormattedMessage>` 作为占位符使用?

`react-intl` 的 `<Formatted ... />` 组件返回元素，而不是纯文本，因此它们不能用于占位符，替代文本等。在这种情况下，您应该使用较低级别的 API `formatMessage()` 。您可以使用 `injectIntl()` 高阶函数将 `intl` 对象注入到组件中，然后使用该对象上使用 `formatMessage()` 格式化消息。

```
import React from 'react'
import { injectIntl, intlShape } from 'react-intl'

const MyComponent = ({ intl }) => {
  const placeholder = intl.formatMessage({id: 'messageId'})
  return <input placeholder={placeholder} />
}

MyComponent.propTypes = {
  intl: intlShape.isRequired
}

export default injectIntl(MyComponent)
```

⬆ 返回顶部

144. 如何使用 React Intl 访问当前语言环境?

您可以在应用的任何组件中使用 `injectIntl()` 获取的当前语言环境:

```
import { injectIntl, intlShape } from 'react-intl'

const MyComponent = ({ intl }) => (
  <div>{`The current locale is ${intl.locale}`}</div>
)

MyComponent.propTypes = {
  intl: intlShape.isRequired
}

export default injectIntl(MyComponent)
```

⬆ 返回顶部

145. 如何使用 React Intl 格式化日期?

`injectIntl()` 高阶组件将允许您通过组件中的 props 访问 `formatDate()` 方法。 该方法由 `FormattedDate` 实例在内部使用，它返回格式化日期的字符串表示。

```
import { injectIntl, intlShape } from 'react-intl'

const stringDate = this.props.intl.formatDate(date, {
  year: 'numeric',
  month: 'numeric',
  day: 'numeric'
})

const MyComponent = ({intl}) => (
  <div>{`The formatted date is ${stringDate}`}</div>
)

MyComponent.propTypes = {
  intl: intlShape.isRequired
```

```
}

export default injectIntl(MyComponent)
```

⬆ 返回顶部

React Testing

146. 在 React 测试中什么是浅层渲染（Shallow Renderer）？

浅层渲染 对于在 React 中编写单元测试用例很有用。它允许您渲染一个 *一级深的组件* 并断言其渲染方法返回的内容，而不必担心子组件未实例化或渲染。

例如，如果您有以下组件：

```
function MyComponent() {
  return (
    <div>
      <span className={'heading'}>{'Title'}</span>
      <span className={'description'}>{'Description'}</span>
    </div>
  )
}
```

然后你可以如下断言：

```
import ShallowRenderer from 'react-test-renderer/shallow'

// in your test
const renderer = new ShallowRenderer()
renderer.render(<MyComponent />)

const result = renderer.getRenderOutput()

expect(result.type).toBe('div')
expect(result.props.children).toEqual([
  <span className={'heading'}>{'Title'}</span>,
  <span className={'description'}>{'Description'}</span>
])
```

⬆ 返回顶部

147. 在 React 中 `TestRenderer` 包是什么?

此包提供了一个渲染器，可用于将组件渲染为纯 JavaScript 对象，而不依赖于 DOM 或原生移动环境。该包可以轻松获取由 ReactDOM 或 React Native 平台所渲染的视图层次结构（类似于DOM树）的快照，而无需使用浏览器或 `jsdom` 。

```
import TestRenderer from 'react-test-renderer'

const Link = ({page, children}) => <a href={page}>{children}</a>
```

```
const testRenderer = TestRenderer.create(
  <Link page={'https://www.facebook.com/'}>{'Facebook'}</Link>
)

console.log(testRenderer.toJSON())
// {
//   type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ]
// }
```

⬆ 返回顶部

148. ReactTestUtils 包的目的是什么?

ReactTestUtils 由 `with-addons` 包提供，允许您对模拟 DOM 执行操作以进行单元测试。

⬆ 返回顶部

149. 什么是 Jest?

Jest 是一个由 Facebook 基于 Jasmine 创建的 JavaScript 单元测试框架，提供自动模拟创建和 `jsdom` 环境。它通常用于测试组件。

⬆ 返回顶部

150. Jest 对比 Jasmine 有什么优势?

与 Jasmine 相比，有几个优点：

- 自动查找在源代码中要执行测试。
- 在运行测试时自动模拟依赖项。
- 允许您同步测试异步代码。
- 使用假的 DOM 实现（通过 `jsdom` ）运行测试，以便可以在命令行上运行测试。
- 在并行流程中运行测试，以便更快完成。

⬆ 返回顶部

151. 举一个简单的 Jest 测试用例

让我们为 `sum.js` 文件中添加两个数字的函数编写一个测试：

```
const sum = (a, b) => a + b

export default sum
```

创建一个名为 `sum.test.js` 的文件，其中包含实际测试：

```
import sum from './sum'

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3)
})
```

然后将以下部分添加到 `package.json` ：

```
{
  "scripts": {
    "test": "jest"
  }
}
```

最后，运行 `yarn test` 或 `npm test` ，Jest 将打印结果：

```
$ yarn test
PASS ./sum.test.js
✓ adds 1 + 2 to equal 3 (2ms)
```

⬆ 返回顶部

React Redux

152. 什么是 Flux?

Flux 是 *应用程序设计范例* ，用于替代更传统的 MVC 模式。它不是一个框架或库，而是一种新的体系结构，它补充了 React 和单向数据流的概念。在使用 React 时，Facebook 会在内部使用此模式。

在 dispatcher，stores 和视图组件具有如下不同的输入和输出：

⬆ 返回顶部

153. 什么是 Redux?

Redux 是基于 *Flux设计模式* 的 JavaScript 应用程序的可预测状态容器。Redux 可以与 React 一起使用，也可以与任何其他视图库一起使用。它很小（约2kB）并且没有依赖性。

⬆ 返回顶部

154. Redux 的核心原则是什么？？

Redux 遵循三个基本原则：

1. **单一数据来源：** 整个应用程序的状态存储在单个对象树中。单状态树可以更容易地跟踪随时间的变化并调试或检查应用程序。
2. **状态是只读的：** 改变状态的唯一方法是发出一个动作，一个描述发生的事情的对象。这可以确保视图和网络请求都不会直接写入状态。
3. **使用纯函数进行更改：** 要指定状态树如何通过操作进行转换，您可以编写reducers。Reducers 只是纯函数，它将先前的状态和操作作为参数，并返回下一个状态。

⬆ 返回顶部

155. 与 Flux 相比，Redux 的缺点是什么?

我们应该说使用 Redux 而不是 Flux 几乎没有任何缺点。这些如下：

1. **您将需要学会避免突变：** Flux 对变异数据毫不吝啬，但 Redux 不喜欢突变，许多与 Redux 互补的包假设您从不改变状态。您可以使用 dev-only 软件包强制执行此操作，例如 `redux-immutable-state-invariant` ，Immutable.js，或指示您的团队编写非变异代码。

2. **您将不得不仔细选择您的软件包：** 虽然 Flux 明确没有尝试解决诸如撤消/重做，持久性或表单之类的问题，但 Redux 有扩展点，例如中间件和存储增强器，以及它催生了丰富的生态系统。
3. **还没有很好的 Flow 集成：** Flux 目前可以让你做一些非常令人印象深刻的静态类型检查，Redux 还不支持。

⬆ 返回顶部

156. `mapStateToProps()` 和 `mapDispatchToProps()` 之间有什么区别?

`mapStateToProps()` 是一个实用方法，它可以帮助您的组件获得最新的状态（由其他一些组件更新）：

```
const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

`mapDispatchToProps()` 是一个实用方法，它可以帮助你的组件触发一个动作事件（可能导致应用程序状态改变的调度动作）：

```
const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}
```

⬆ 返回顶部

157. 我可以在 reducer 中触发一个 Action 吗?

在 reducer 中触发 Action 是**反模式**。您的 reducer 应该 *没有副作用* ，只是接收 Action 并返回一个新的状态对象。在 reducer 中添加侦听器和调度操作可能会导致链接的 Action 和其他副作用。

⬆ 返回顶部

158. 如何在组件外部访问 Redux 存储的对象?

是的，您只需要使用 `createStore()` 从它创建的模块中导出存储。此外，它不应污染全局窗口对象。

```
store = createStore(myReducer)

export default store
```

⬆ 返回顶部

159. MVW 模式的缺点是什么?

1. DOM 操作非常昂贵，导致应用程序行为缓慢且效率低下。
2. 由于循环依赖性，围绕模型和视图创建了复杂的模型。

3. 协作型应用程序（如Google Docs）会发生大量数据更改。
4. 无需添加太多额外代码就无法轻松撤消（及时回退）。

⬆ 返回顶部

160. Redux 和 RxJS 之间是否有任何相似之处?

这些库的目的是不同的，但是存在一些模糊的相似之处。

Redux 是一个在整个应用程序中管理状态的工具。它通常用作 UI 的体系结构。可以将其视为（一半）Angular 的替代品。 RxJS 是一个反应式编程库。它通常用作在 JavaScript 中完成异步任务的工具。把它想象成 Promise 的替代品。 Redux 使用 Reactive 范例，因为Store是被动的。Store 检测到 Action，并自行改变。RxJS也使用 Reactive 范例，但它不是一个体系结构，它为您提供了基本构建块 Observables 来完成这种模式。

⬆ 返回顶部

161. 如何在加载时触发 Action?

您可以在 `componentDidMount()` 方法中触发 Action，然后在 `render()` 方法中可以验证数据。

```
class App extends Component {
  componentDidMount() {
    this.props.fetchData()
  }

  render() {
    return this.props.isLoaded
      ? <div>{'Loaded'}</div>
      : <div>{'Not Loaded'}</div>
  }
}

const mapStateToProps = (state) => ({
  isLoaded: state.isLoaded
})

const mapDispatchToProps = { fetchData }

export default connect(mapStateToProps, mapDispatchToProps)(App)
```

⬆ 返回顶部

162. 在 React 中如何使用 Redux 的 `connect()` ?

您需要按照两个步骤在容器中使用您的 Store：

1. **使用 `mapStateToProps()`：** 它将 Store 中的状态变量映射到您指定的属性。
2. **将上述属性连接到容器：** `mapStateToProps` 函数返回的对象连接到容器。你可以从 `react-redux` 导入 `connect()` 。

```
import React from 'react'
import { connect } from 'react-redux'
```

```
class App extends React.Component {
  render() {
    return <div>{this.props.containerData}</div>
  }
}

function mapStateToProps(state) {
  return { containerData: state.data }
}

export default connect(mapStateToProps)(App)
```

⬆ 返回顶部

163. 如何在 Redux 中重置状态?

你需要在你的应用程序中编写一个 *root reducer* ，它将处理动作委托给 `combineReducers()` 生成的 reducer。

例如，让我们在 `USER_LOGOUT` 动作之后让 `rootReducer()` 返回初始状态。我们知道，无论 Action 怎么样，当使用 `undefined` 作为第一个参数调用它们时，reducers 应该返回初始状态。

```
const appReducer = combineReducers({
  /* your app's top-level reducers */
})

const rootReducer = (state, action) => {
  if (action.type === 'USER_LOGOUT') {
    state = undefined
  }

  return appReducer(state, action)
}
```

如果使用 `redux-persist` ，您可能还需要清理存储空间。 `redux-persist` 在 storage 引擎中保存您的状态副本。首先，您需要导入适当的 storage 引擎，然后在将其设置为 `undefined` 之前解析状态并清理每个存储状态键。

```
const appReducer = combineReducers({
  /* your app's top-level reducers */
})

const rootReducer = (state, action) => {
  if (action.type === 'USER_LOGOUT') {
    Object.keys(state).forEach(key => {
      storage.removeItem(`persist:${key}`)
    })

    state = undefined
  }

  return appReducer(state, action)
}
```

⬆ 返回顶部

164. Redux 中连接装饰器的 `at` 符号的目的是什么?

@ 符号实际上是用于表示装饰器的 JavaScript 表达式。 *装饰器* 可以在设计时注释和修改类和属性。

让我们举个例子，在没有装饰器的情况下设置 Redux 。

- **未使用装饰器:**

```
import React from 'react'
import * as actionCreators from './actionCreators'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

function mapDispatchToProps(dispatch) {
  return { actions: bindActionCreators(actionCreators, dispatch) }
}

class MyApp extends React.Component {
  // ...define your main app here
}

export default connect(mapStateToProps, mapDispatchToProps)(MyApp)
```

- **使用装饰器:**

```
import React from 'react'
import * as actionCreators from './actionCreators'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'

function mapStateToProps(state) {
  return { todos: state.todos }
}

function mapDispatchToProps(dispatch) {
  return { actions: bindActionCreators(actionCreators, dispatch) }
}

@connect(mapStateToProps, mapDispatchToProps)
export default class MyApp extends React.Component {
  // ...define your main app here
}
```

除了装饰器的使用外，上面的例子几乎相似。装饰器语法尚未构建到任何 JavaScript 运行时中，并且仍然是实验性的并且可能会发生变化。您可以使用 `babel` 来获得装饰器支持。

⬆ 返回顶部

165. React 上下文和 React Redux 之间有什么区别?

您可以直接在应用程序中使用**Context**，这对于将数据传递给深度嵌套的组件非常有用。而**Redux**功能更强大，它还提供了 Context API 无法提供的大量功能。此外，React Redux 在内部使用上下文，但它不会在公共 API 中有所体现。

⬆ 返回顶部

166. 为什么 Redux 状态函数称为 reducers ?

Reducers 总是返回状态的累积（基于所有先前状态和当前 Action）。因此，它们充当了状态的 Reducer。每次调用 Redux reducer 时，状态和 Action 都将作为参数传递。然后基于该 Action 减少（或累积）该状态，然后返回下一状态。您可以 *reduce* 一组操作和一个初始状态（Store），在该状态下执行这些操作以获得最终的最终状态。

⬆ 返回顶部

167. 如何在 Redux 中发起 AJAX 请求?

您可以使用 `redux-thunk` 中间件，它允许您定义异步操作。

让我们举个例子，使用 *fetch API* 将特定帐户作为 AJAX 调用获取：

```
export function fetchAccount(id) {
  return dispatch => {
    dispatch(setLoadingAccountState()) // Show a loading spinner
    fetch(`/account/${id}`, (response) => {
      dispatch(doneFetchingAccount()) // Hide loading spinner
      if (response.status === 200) {
        dispatch(setAccount(response.json)) // Use a normal function to set the
received state
      } else {
        dispatch(someError)
      }
    })
  }
}

function setAccount(data) {
 return { type: 'SET_Account', data: data }
}
```

⬆ 返回顶部

168. 我应该在 Redux Store 中保留所有组件的状态吗?

将数据保存在 Redux 存储中，并在组件内部保持 UI 相关状态。

⬆ 返回顶部

169. 访问 Redux Store 的正确方法是什么?

在组件中访问 Store 的最佳方法是使用 `connect()` 函数，该函数创建一个包裹现有组件的新组件。此模式称为 *高阶组件* ，通常是在 React 中扩展组件功能的首选方式。这允许您将状态和 Action 创建者映射到组件，并在 Store 更新时自动传递它们。

我们来看一个使用 connect 的 `<FilterLink>` 组件的例子：

```
import { connect } from 'react-redux'
import { setVisibilityFilter } from '../actions'
import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => ({
  active: ownProps.filter === state.visibilityFilter
})

const mapDispatchToProps = (dispatch, ownProps) => ({
  onClick: () => dispatch(setVisibilityFilter(ownProps.filter))
})

const FilterLink = connect(
  mapStateToProps,
  mapDispatchToProps
)(Link)

export default FilterLink
```

由于它具有相当多的性能优化并且通常不太可能导致错误，因此 Redux 开发人员几乎总是建议使用 `connect()` 直接访问 Store（使用上下文API）。

```
class MyComponent {
  someMethod() {
    doSomethingWith(this.context.store)
  }
}
```

⬆ 返回顶部

170. React Redux 中展示组件和容器组件之间的区别是什么?

展示组件是一个类或功能组件，用于描述应用程序的展示部分。

容器组件是连接到 Redux Store的组件的非正式术语。容器组件 *订阅* Redux 状态更新和 *dispatch* 操作，它们通常不呈现 DOM 元素；他们将渲染委托给展示性的子组件。

⬆ 返回顶部

171. Redux 中常量的用途是什么?

常量允许您在使用 IDE 时轻松查找项目中该特定功能的所有用法。它还可以防止你拼写错误，在这种情况下，你会立即得到一个 `ReferenceError` 。

通常我们会将它们保存在一个文件中（ `constants.js` 或 `actionTypes.js` ）。

```
export const ADD_TODO = 'ADD_TODO'
export const DELETE_TODO = 'DELETE_TODO'
export const EDIT_TODO = 'EDIT_TODO'
export const COMPLETE_TODO = 'COMPLETE_TODO'
export const COMPLETE_ALL = 'COMPLETE_ALL'
export const CLEAR_COMPLETED = 'CLEAR_COMPLETED'
```

在 Redux 中，您可以在两个地方使用它们：

1. **在 Action 创建时:**

   让我们看看 `actions.js`:

```
import { ADD_TODO } from './actionTypes';

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

2. **在 reducers 里:**

   让我们创建 `reducer.js` 文件:

```
import { ADD_TODO } from './actionTypes'

export default (state = [], action) => {
  switch (action.type) {
    case ADD_TODO:
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ];
    default:
      return state
  }
}
```

⬆ 返回顶部

172. 编写 `mapDispatchToProps()` 有哪些不同的方法?

有一些方法可以将 *action creators* 绑定到 `mapDispatchToProps()` 中的 `dispatch()`。以下是可能的写法:

```
const mapDispatchToProps = (dispatch) => ({
 action: () => dispatch(action())
})
```

```
const mapDispatchToProps = (dispatch) => ({
 action: bindActionCreators(action, dispatch)
})
```

```
const mapDispatchToProps = { action }
```

第三种写法只是第一种写法的简写。

⬆ 返回顶部

173. 在 `mapStateToProps()` 和 `mapDispatchToProps()` 中使用 `ownProps` 参数有什么用?

如果指定了 `ownProps` 参数，React Redux 会将传递给该组件的 props 传递给你的 *connect* 函数。因此，如果您使用连接组件：

```
import ConnectedComponent from './containers/ConnectedComponent';

<ConnectedComponent user={'john'} />
```

你的 `mapStateToProps()` 和 `mapDispatchToProps()` 函数里面的 `ownProps` 将是一个对象：

```
{ user: 'john' }
```

您可以使用此对象来决定从这些函数返回的内容。

⬆ 返回顶部

174. 如何构建 Redux 项目目录?

大多数项目都有几个顶级目录，如下所示：

1. **Components**: 用于 *dumb* 组件，Redux 不必关心的组件。
2. **Containers**: 用于连接到 Redux 的 *smart* 组件。
3. **Actions**: 用于所有 Action 创建器，其中文件名对应于应用程序的一部分。
4. **Reducers**: 用于所有 reducer，其中文件名对应于state key。
5. **Store**: 用于 Store 初始化。

这种结构适用于中小型项目。

⬆ 返回顶部

175. 什么是 redux-saga?

`redux-saga` 是一个库，旨在使 React/Redux 项目中的副作用（数据获取等异步操作和访问浏览器缓存等可能产生副作用的动作）更容易，更好。

这个包在 NPM 上有发布:

```
$ npm install --save redux-saga
```

⬆ 返回顶部

176. redux-saga 的模型概念是什么?

Saga 就像你的项目中的一个单独的线程，它独自负责副作用。`redux-saga` 是一个 redux *中间件*，这意味着它可以在项目启动中使用正常的 Redux 操作，暂停和取消该线程，它可以访问完整的 Redux 应用程序状态，并且它也可以调度 Redux 操作。

⬆ 返回顶部

177. 在 redux-saga 中 `call()` 和 `put()` 之间有什么区别?

`call()` 和 `put()` 都是 Effect 创建函数。`call()` 函数用于创建 Effect 描述，指示中间件调用 promise。`put()` 函数创建一个 Effect，指示中间件将一个 Action 分派给 Store。

让我们举例说明这些 Effect 如何用于获取特定用户数据。

```
 function* fetchUserSaga(action) {
   // `call` function accepts rest arguments, which will be passed to
`api.fetchUser` function.
   // Instructing middleware to call promise, it resolved value will be assigned
to `userData` variable
   const userData = yield call(api.fetchUser, action.userId)

   // Instructing middleware to dispatch corresponding action.
   yield put({
     type: 'FETCH_USER_SUCCESS',
     userData
   })
 }
```

⬆ 返回顶部

178. 什么是 Redux Thunk?

Redux Thunk 中间件允许您编写返回函数而不是 Action 的创建者。 thunk 可用于延迟 Action 的发送，或仅在满足某个条件时发送。内部函数接收 Store 的方法 `dispatch()` 和 `getState()` 作为参数。

⬆ 返回顶部

179. `redux-saga` 和 `redux-thunk` 之间有什么区别?

Redux Thunk 和 *Redux Saga* 都负责处理副作用。在大多数场景中，Thunk 使用 *Promises* 来处理它们，而 Saga 使用 *Generators* 。Thunk 易于使用，因为许多开发人员都熟悉 Promise，Sagas/Generators 功能更强大，但您需要学习它们。但是这两个中间件可以共存，所以你可以从 Thunks 开始，并在需要时引入 Sagas。

⬆ 返回顶部

180. 什么是 Redux DevTools?

Redux DevTools 是 Redux 的实时编辑的时间旅行环境，具有热重新加载，Action 重放和可自定义的 UI。如果您不想安装 Redux DevTools 并将其集成到项目中，请考虑使用 Chrome 和 Firefox 的扩展插件。

⬆ 返回顶部

181. Redux DevTools 的功能有哪些?

1. 允许您检查每个状态和 action 负载。
2. 让你可以通过 *撤销* 回到过去。
3. 如果更改 reducer 代码，将重新评估每个 *已暂存* 的 Action。
4. 如果 Reducers 抛出错误，你会看到这发生了什么 Action，以及错误是什么。
5. 使用 `persistState()` 存储增强器，您可以在页面重新加载期间保持调试会话。

⬆ 返回顶部

182. 什么是 Redux 选择器以及使用它们的原因?

选择器 是将 Redux 状态作为参数并返回一些数据以传递给组件的函数。

例如，要从 state 中获取用户详细信息：

```
const getUserData = state => state.user.data
```

⬆ 返回顶部

183. 什么是 Redux Form?

Redux Form 与 React 和 Redux 一起使用，以使 React 中的表单能够使用 Redux 来存储其所有状态。Redux Form 可以与原始 HTML5 输入一起使用，但它也适用于常见的 UI 框架，如 Material UI，React Widgets和React Bootstrap。

⬆ 返回顶部

184. Redux Form 的主要功能有哪些?

```
1. 字段值通过 Redux 存储持久化。
2. 验证（同步/异步）和提交。
3. 字段值的格式化，解析和规范化。
```

⬆ 返回顶部

185. 如何向 Redux 添加多个中间件?

你可以使用 `applyMiddleware()` 。

例如，你可以添加 `redux-thunk` 和 `logger` 作为参数传递给 `applyMiddleware()` ：

```
import { createStore, applyMiddleware } from 'redux'
const createStoreWithMiddleware = applyMiddleware(ReduxThunk, logger)
(createStore)
```

⬆ 返回顶部

186. 如何在 Redux 中设置初始状态?

您需要将初始状态作为第二个参数传递给 createStore：

```
const rootReducer = combineReducers({
  todos: todos,
  visibilityFilter: visibilityFilter
})

const initialState = {
  todos: [{ id: 123, name: 'example', completed: false }]
}

const store = createStore(
  rootReducer,
  initialState
)
```

⬆ 返回顶部

187. Relay 与 Redux 有何不同?

Relay 与 Redux 类似，因为它们都使用单个 Store。主要区别在于 relay 仅管理源自服务器的状态，并且通过 *GraphQL* 查询（用于读取数据）和突变（用于更改数据）来使用对状态的所有访问。Relay 通过仅提取已更改的数据而为您缓存数据并优化数据提取。

⬆ 返回顶部

React Native

188. React Native 和 React 有什么区别?

React是一个 JavaScript 库，支持前端 Web 和在服务器上运行，用于构建用户界面和 Web 应用程序。

React Native是一个移动端框架，可编译为本机应用程序组件，允许您使用 JavaScript 构建本机移动应用程序（iOS，Android和Windows），允许您使用 React 构建组件。

⬆ 返回顶部

189. 如何测试 React Native 应用程序?

React Native 只能在 iOS 和 Android 等移动模拟器中进行测试。您可以使用 expo app（https://expo.io）在移动设备上运行该应用程序。如果使用 QR 代码进行同步，则您的移动设备和计算机应位于同一个无线网络中。

⬆ 返回顶部

190. 如何在 React Native 查看日志?

您可以使用 `console.log`， `console.warn` 等。从 React Native v0.29 开始，您只需运行以下命令即可在控制台中查看日志：

```
$ react-native log-ios
$ react-native log-android
```

⬆ 返回顶部

191. 怎么调试 React Native 应用?

按照以下步骤调试 React Native 应用程序：

1. 在 iOS 模拟器中运行您的应用程序。
2. 按 `Command + D`，然后在网页中打开 `http://localhost:8081/debugger-ui`。
3. 启用 *Pause On Caught Exceptions* 以获得更好的调试体验。
4. 按 `Command + Option + I` 打开 Chrome Developer 工具，或通过 `View` -> `Developer` -> `Developer Tools` 打开它。
5. 您现在应该能够像平常那样进行调试。

⬆ 返回顶部

React supported libraries & Integration

192. 什么是 Reselect 以及它是如何工作的?

Reselect 是一个**选择器库**（用于 Redux），它使用 *memoization* 概念。它最初编写用于计算类似 Redux 的应用程序状态的派生数据，但它不能绑定到任何体系结构或库。

Reselect 保留最后一次调用的最后输入/输出的副本，并仅在其中一个输入发生更改时重新计算结果。如果连续两次提供相同的输入，则 Reselect 将返回缓存的输出。它的 memoization 和缓存是完全可定制的。

⬆ 返回顶部

193. 什么是 Flow?

Flow 是一个静态类型检查器，旨在查找 JavaScript 中的类型错误。与传统类型系统相比，Flow 类型可以表达更细粒度的区别。例如，与大多数类型系统不同，Flow 能帮助你捕获涉及 `null` 的错误。

⬆ 返回顶部

194. Flow 和 PropTypes 有什么区别?

Flow 是一个静态分析工具（静态检查器），它使用该语言的超集，允许你在所有代码中添加类型注释，并在编译时捕获整个类的错误。PropTypes 是一个基本类型检查器（运行时检查器），已经被添加到 React 中。除了检查传递给给定组件的属性类型外，它不能检查其他任何内容。如果你希望对整个项目进行更灵活的类型检查，那么 Flow/TypeScript 是更合适的选择。

⬆ 返回顶部

195. 在 React 中如何使用 Font Awesome 图标?

接下来的步骤将在 React 中引入 Font Awesome：

1. 安装 `font-awesome`：

```
$ npm install --save font-awesome
```

2. 在 `index.js` 文件中导入 `font-awesome`：

```
import 'font-awesome/css/font-awesome.min.css'
```

3. 在 `className` 中添加 Font Awesome 类:

```
render() {
  return <div><i className={'fa fa-spinner'} /></div>
}
```

⬆ 返回顶部

196. 什么 是 React 开发者工具?

React Developer Tools 允许您检查组件层次结构，包括组件属性和状态。它既可以作为浏览器扩展（用于 Chrome 和 Firefox），也可以作为独立的应用程序（用于其他环境，包括 Safari、IE 和 React Native）。

可用于不同浏览器或环境的官方扩展。

1. **Chrome插件**
2. **Firefox插件**
3. **独立应用**（Safari，React Native 等）

⬆ 返回顶部

197. 在 Chrome 中为什么 DevTools 没有加载本地文件?

如果您在浏览器中打开了本地 HTML 文件（`file://...`），则必须先打开 *Chrome Extensions* 并选中“允许访问文件URL”。

⬆ 返回顶部

198. 如何在 React 中使用 Polymer?

1. 创建 Polymer 元素：

```
<link rel='import' href='../../bower_components/polymer/polymer.html' />
Polymer({
  is: 'calender-element',
  ready: function() {
    this.textContent = 'I am a calender'
  }
})
```

2. 通过在 HTML 文档中导入 Polymer 组件，来创建该组件对应的标签。例如，在 React 应用程序的 `index.html` 文件中导入。

```
<link rel='import' href='./src/polymer-components/calender-element.html'>
```

3. 在 JSX 文件中使用该元素：

```
import React from 'react'

class MyComponent extends React.Component {
  render() {
    return (
      <calender-element />
    )
  }
}

export default MyComponent
```

⬆ 返回顶部

199. 与 Vue.js 相比，React 有哪些优势?

与 Vue.js 相比，React 具有以下优势：

1. 在大型应用程序开发中提供更大的灵活性。
2. 更容易测试。
3. 更适合创建移动端应用程序。
4. 提供更多的信息和解决方案。

⬆ 返回顶部

200. React 和 Angular 有什么区别?

React	Angular
React 是一个库，只有View层	Angular是一个框架，具有完整的 MVC 功能
React 可以处理服务器端的渲染	AngularJS 仅在客户端呈现，但 Angular 2 及更高版本可以在服务器端渲染
React 在 JS 中使用看起来像 HTML 的 JSX，这可能令人困惑	Angular 遵循 HTML 的模板方法，这使得代码更短且易于理解
React Native 是一种 React 类型，它用于构建移动应用程序，它更快，更稳定	Ionic，Angular 的移动 app 相对原生 app 来说不太稳定和慢
在 Reac t中，数据只以单一方向传递，因此调试很容易	在 Angular 中，数据以两种方式传递，即它在子节点和父节点之间具有双向数据绑定，因此调试通常很困难

⬆ 返回顶部

201. 为什么 React 选项卡不会显示在 DevTools 中?

当页面加载时， *React DevTools* 设置一个名为 `__REACT_DEVTOOLS_GLOBAL_HOOK__` 的全局变量，然后 React 在初始化期间与该钩子通信。如果网站没有使用 React，或者如果 React 无法与 DevTools 通信，那么它将不会显示该选项卡。

⬆ 返回顶部

202. 什么是 Styled Components?

styled-components 是一个用于样式化 React 应用程序的 JavaScript 库。 它删除了样式和组件之间的映射，并允许您在 js 中编写 CSS。

⬆ 返回顶部

203. 举一个 Styled Components 的例子?

让我们创建具有特定样式的 `<Title>` 和 `<Wrapper>` 组件。

```
import React from 'react'
import styled from 'styled-components'

// Create a <Title> component that renders an <h1> which is centered, red and
sized at 1.5em
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`

// Create a <Wrapper> component that renders a <section> with some padding and a
papayawhip background
const Wrapper = styled.section`
  padding: 4em;
```

```
  background: papayawhip;
`
```

`Title` 和 `Wrapper` 变量现在是可以像任何其他 react 组件一样渲染。

```
<Wrapper>
  <Title>{'Lets start first styled component!'}</Title>
</Wrapper>
```

⬆ 返回顶部

204. 什么是 Relay?

Relay 是一个 JavaScript 框架，用于使用 React 视图层为 Web 应用程序提供数据层和客户端与服务器之间的通信。

⬆ 返回顶部

205. 如何在 `create-react-app` 中使用 TypeScript?

当您创建一个新项目带有 `--scripts-version` 选项值为 `react-scripts-ts` 时便可将 TypeScript 引入。

生成的项目结构如下所示：

```
my-app/
├─ .gitignore
├─ images.d.ts
├─ node_modules/
├─ public/
├─ src/
|  └─ ...
├─ package.json
├─ tsconfig.json
├─ tsconfig.prod.json
├─ tsconfig.test.json
└─ tslint.json
```

⬆ 返回顶部

Miscellaneous

206. Reselect 库的主要功能有哪些?

1. 选择器可以计算派生数据，允许 Redux 存储最小可能状态。
2. 选择器是有效的。除非其参数之一发生更改，否则不会重新计算选择器。
3. 选择器是可组合的。它们可以用作其他选择器的输入。

⬆ 返回顶部

207. 举一个 Reselect 用法的例子?

让我们通过使用 Reselect 来简化计算不同数量的装运订单：

```
import { createSelector } from 'reselect'

const shopItemsSelector = state => state.shop.items
const taxPercentSelector = state => state.shop.taxPercent

const subtotalSelector = createSelector(
  shopItemsSelector,
  items => items.reduce((acc, item) => acc + item.value, 0)
)

const taxSelector = createSelector(
  subtotalSelector,
  taxPercentSelector,
  (subtotal, taxPercent) => subtotal * (taxPercent / 100)
)

export const totalSelector = createSelector(
  subtotalSelector,
  taxSelector,
  (subtotal, tax) => ({ total: subtotal + tax })
)

let exampleState = {
  shop: {
    taxPercent: 8,
    items: [
      { name: 'apple', value: 1.20 },
      { name: 'orange', value: 0.95 },
    ]
  }
}

console.log(subtotalSelector(exampleState)) // 2.15
console.log(taxSelector(exampleState))      // 0.172
console.log(totalSelector(exampleState))    // { total: 2.322 }
```

⬆ 返回顶部

208. Redux 中的 Action 是什么?

Actions 是纯 JavaScript 对象或信息的有效负载，可将数据从您的应用程序发送到您的 Store。 它们是 Store 唯一的数据来源。 Action 必须具有指示正在执行的操作类型的 type 属性。

例如，表示添加新待办事项的示例操作：

```
{
  type: ADD_TODO,
  text: 'Add todo item'
}
```

⬆ 返回顶部

209. 在 React 中 statics 对象是否能与 ES6 类一起使用?

不行，`statics` 仅适用于 `React.createClass()`：

```
someComponent= React.createClass({
  statics: {
    someMethod: function() {
      // ..
    }
  }
})
```

但是你可以在 ES6+ 的类中编写静态代码，如下所示：

```
class Component extends React.Component {
  static propTypes = {
    // ...
  }

  static someMethod() {
    // ...
  }
}
```

⬆ 返回顶部

210. Redux 只能与 React 一起使用么?

Redux 可以用做任何 UI 层的数据存储。最常见的应用场景是 React 和 React Native，但也有一些 bindings 可用于 AngularJS，Angular 2，Vue，Mithril 等项目。Redux 只提供了一种订阅机制，任何其他代码都可以使用它。

⬆ 返回顶部

211. 您是否需要使用特定的构建工具来使用 Redux ?

Redux 最初是用 ES6 编写的，用 Webpack 和 Babel 编译成 ES5。 无论您的 JavaScript 构建过程如何，您都应该能够使用它。Redux 还提供了一个 UMD 版本，可以直接使用而无需任何构建过程。

⬆ 返回顶部

212. Redux Form 的 `initialValues` 如何从状态更新?

你需要添加 `enableReinitialize: true` 设置。

```
const InitializeFromStateForm = reduxForm({
  form: 'initializeFromState',
  enableReinitialize : true
})(UserEdit)
```

如果你的 `initialValues` 属性得到更新，你的表单也会更新。

⬆ 返回顶部

213. React 是如何为一个属性声明不同的类型?

你可以使用 `PropTypes` 中的 `oneOfType()` 方法。

例如，如下所示 size 的属性值可以是 `string` 或 `number` 类型。

```
Component.PropTypes = {
  size: PropTypes.oneOfType([
    PropTypes.string,
    PropTypes.number
  ])
}
```

⬆ 返回顶部

214. 我可以导入一个 SVG 文件作为 React 组件么?

你可以直接将 SVG 作为组件导入，而不是将其作为文件加载。此功能仅在 `react-scripts@2.0.0` 及更高版本中可用。

```
```jsx
import { ReactComponent as Logo } from './logo.svg'

const App = () => (
  <div>
    {/* Logo is an actual react component */}
    <Logo />
  </div>
)
```

[⬆ 返回顶部](#目录)
```

215. 为什么不建议使用内联引用回调或函数?

如果将 ref 回调定义为内联函数，则在更新期间它将会被调用两次。首先使用 null 值，然后再使用 DOM 元素。这是因为每次渲染的时候都会创建一个新的函数实例，因此 React 必须清除旧的 ref 并设置新的 ref。

```
class UserForm extends Component {
  handleSubmit = () => {
    console.log("Input Value is: ", this.input.value)
  }

  render () {
   return (
     <form onSubmit={this.handleSubmit}>
       <input
         type='text'
         ref={(input) => this.input = input} /> // Access DOM input in handle
submit
       <button type='submit'>Submit</button>
     </form>
   )
  }
 }
```

但我们期望的是当组件挂载时，ref 回调只会被调用一次。一个快速修复的方法是使用 ES7 类属性语法定义函数。

```
class UserForm extends Component {
 handleSubmit = () => {
   console.log("Input Value is: ", this.input.value)
 }

 setSearchInput = (input) => {
   this.input = input
 }

 render () {
   return (
     <form onSubmit={this.handleSubmit}>
       <input
         type='text'
         ref={this.setSearchInput} /> // Access DOM input in handle submit
       <button type='submit'>Submit</button>
     </form>
   )
 }
}
```

⬆ 返回顶部

216. 在 React 中什么是渲染劫持?

渲染劫持的概念是控制一个组件将从另一个组件输出什么的能力。实际上，这意味着你可以通过将组件包装成高阶组件来装饰组件。通过包装，你可以注入额外的属性或产生其他变化，这可能会导致渲染逻辑的更改。实际上它不支持劫持，但通过使用 HOC，你可以使组件以不同的方式工作。

⬆ 返回顶部

217. 什么是 HOC 工厂实现?

在 React 中实现 HOC 有两种主要方式。 1.属性代理（PP）和 2.继承倒置（II）。他们遵循不同的方法来操纵 *WrappedComponent* 。

属性代理 在这种方法中，HOC 的 render 方法返回 WrappedComponent 类型的 React 元素。我们通过 HOC 收到 props，因此定义为**属性代理**。

```
function ppHOC(WrappedComponent) {
 return class PP extends React.Component {
   render() {
     return <WrappedComponent {...this.props}/>
   }
 }
}
```

继承倒置 在这种方法中，返回的 HOC 类（Enhancer）扩展了 WrappedComponent 。它被称为继承反转，因为它不是扩展一些 Enhancer 类的 WrappedComponent，而是由 Enhancer 被动扩展。通过这种方式，它们之间的关系似乎是**逆的**。

```
function iiHOC(WrappedComponent) {
 return class Enhancer extends WrappedComponent {
   render() {
     return super.render()
   }
 }
}
```

⬆ 返回顶部

218. 如何传递数字给 React 组件?

传递数字时你应该使用 `{}` ，而传递字符串时还需要使用引号：

```
    React.render(<User age={30} department={"IT"} />,
document.getElementById('container'));
```

⬆ 返回顶部

219. 我需要将所有状态保存到 Redux 中吗？我应该使用 react 的内部状态吗?

这取决于开发者的决定。即开发人员的工作是确定应用程序的哪种状态，以及每个状态应该存在的位置，有些用户喜欢将每一个数据保存在 Redux 中，以维护其应用程序的完全可序列化和受控。其他人更喜欢在组件的内部状态内保持非关键或UI状态，例如“此下拉列表当前是否打开”。

以下是确定应将哪种数据放入Redux的主要规则：

1. 应用程序的其他部分是否关心此数据?
2. 您是否需要能够基于此原始数据创建更多派生数据?
3. 是否使用相同的数据来驱动多个组件?
4. 能够将此状态恢复到给定时间点（即时间旅行调试）是否对您有价值?
5. 您是否要缓存数据（即，如果已经存在，则使用处于状态的状态而不是重新请求它）？

220. 在 React 中 registerServiceWorker 的用途是什么?

默认情况下，React 会为你创建一个没有任何配置的 service worker。Service worker 是一个 Web API，它帮助你缓存资源和其他文件，以便当用户离线或在弱网络时，他/她仍然可以在屏幕上看到结果，因此，它可以帮助你建立更好的用户体验，这是你目前应该了解的关于 Service worker 的内容。

```
    import React from 'react';
    import ReactDOM from 'react-dom';
    import App from './App';
    import registerServiceWorker from './registerServiceWorker';

    ReactDOM.render(<App />, document.getElementById('root'));
    registerServiceWorker();
```

⬆ 返回顶部

221. React memo 函数是什么?

当类组件的输入属性相同时，可以使用 **pureComponent** 或 **shouldComponentUpdate** 来避免组件的渲染。现在，你可以通过把函数组件包装在 **React.memo** 中来实现相同的功能。

```
const MyComponent = React.memo(function MyComponent(props) {
 /* only rerenders if props change */
});
```

⬆ 返回顶部

222. React lazy 函数是什么?

使用 React.lazy 函数允许你将动态导入的组件作为常规组件进行渲染。当组件开始渲染时，它会自动加载包含 OtherComponent 的包。它必须返回一个 Promise，该 Promise 解析后为一个带有默认导出 React 组件的模块。

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
 return (
   <div>
     <OtherComponent />
   </div>
 );
}
```

注意： React.lazy 和 Suspense 还不能用于服务端渲染。如果要在服务端渲染的应用程序中进行代码拆分，我们仍然建议使用 React Loadable。

⬆ 返回顶部

223. 如何使用 setState 防止不必要的更新?

你可以把状态的当前值与已有的值进行比较，并决定是否重新渲染页面。如果没有更改，你需要返回 `null` 以阻止渲染，否则返回最新的状态值。例如，用户配置信息组件将按以下方式实现条件渲染：

```
getUserProfile = user => {
  const latestAddress = user.address;
  this.setState(state => {
    if (state.address === latestAddress) {
      return null;
    } else {
      return { title: latestAddress };
    }
  });
};
```

⬆ 返回顶部

224. 如何在 React 16 版本中渲染数组、字符串和数值?

Arrays: 与旧版本不同的是，在 React 16 中你不需要确保 **render** 方法必须返回单个元素。通过返回数组，你可以返回多个没有包装元素的同级元素。例如，让我们看看下面的开发人员列表：

```
const ReactJSDevs = () => {
  return [
    <li key="1">John</li>,
    <li key="2">Jackie</li>,
    <li key="3">Jordan</li>
  ];
}
```

你还可以将此数组项合并到另一个数组组件中：

```
const JSDevs = () => {
  return (
    <ul>
      <li>Brad</li>
      <li>Brodge</li>
      <ReactJSDevs/>
      <li>Brandon</li>
    </ul>
  );
}
```

Strings and Numbers: 在 render 方法中，你也可以返回字符串和数值类型：

```
// String
render() {
 return 'Welcome to ReactJS questions';
}
// Number
render() {
 return 2018;
}
```

⬆ 返回顶部

225. 如何在 React 类中使用类字段声明语法?

使用类字段声明可以使 React 类组件更加简洁。你可以在不使用构造函数的情况下初始化本地状态，并通过使用箭头函数声明类方法，而无需额外对它们进行绑定。让我们以一个 counter 示例来演示类字段声明，即不使用构造函数初始化状态且不进行方法绑定：

```
class Counter extends Component {
  state = { value: 0 };

  handleIncrement = () => {
    this.setState(prevState => ({
      value: prevState.value + 1
    }));
  };

  handleDecrement = () => {
    this.setState(prevState => ({
      value: prevState.value - 1
    }));
```

```
  };

  render() {
    return (
      <div>
        {this.state.value}

        <button onClick={this.handleIncrement}>+</button>
        <button onClick={this.handleDecrement}>-</button>
      </div>
    )
  }
}
```

⬆ 返回顶部

226. 什么是 hooks?

Hooks 是一个新的草案，它允许你在不编写类的情况下使用状态和其他 React 特性。让我们来看一个 useState 钩子示例：

```
import { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

阅读资源：

1. 掘金 - 30分钟精通React Hooks

⬆ 返回顶部

227. Hooks 需要遵循什么规则?

为了使用 hooks，你需要遵守两个规则：

1. 仅在顶层的 React 函数调用 hooks。也就是说，你不能在循环、条件或内嵌函数中调用 hooks。这将确保每次组件渲染时都以相同的顺序调用 hooks，并且它会在多个 useState 和 useEffect 调用之间保留 hooks 的状态。
2. 仅在 React 函数中调用 hooks。例如，你不能在常规的 JavaScript 函数中调用 hooks。

⬆ 返回顶部

228. 如何确保钩子遵循正确的使用规则?

React 团队发布了一个名为**eslint-plugin-react-hooks**的 ESLint 插件，它实施了这两个规则。您可以使用以下命令将此插件添加到项目中，

```
npm install eslint-plugin-react-hooks@next
```

并在您的 ESLint 配置文件中应用以下配置：

```
// Your ESLint configuration
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error"
  }
}
```

注意： 此插件在 Create React App 已经默认配置。

⬆ 返回顶部

229. Flux 和 Redux 之间有什么区别?

以下是 Flux 和 Redux 之间的主要区别

Flux	Redux
状态是可变的	状态是不可变的
Store 包含状态和更改逻辑	存储和更改逻辑是分开的
存在多个 Store	仅存在一个 Store
所有的 Store 都是断开连接的	带有分层 reducers 的 Store
它有一个单独的 dispatcher	没有 dispatcher 的概念
React 组件监测 Store	容器组件使用连接函数

⬆ 返回顶部

230. React Router V4 有什么好处?

以下是 React Router V4 模块的主要优点：

1. 在React Router v4（版本4）中，API完全与组件有关。路由器可以显示为单个组件（），它包装特定的子路由器组件（）。
2. 您无需手动设置历史记录。路由器模块将通过使用组件包装路由来处理历史记录。
3. 通过仅添加特定路由器模块（Web，core 或 native）来减少应用大小。

⬆ 返回顶部

231. 您能描述一下 componentDidCatch 生命周期方法签名吗?

在后代层级的组件抛出错误后，将调用**componentDidCatch**生命周期方法。该方法接收两个参数：

1. error: - 抛出的错误对象
2. info: - 具有 componentStack 键的对象，包含有关哪个组件引发错误的信息。

方法结构如下：

```
componentDidCatch(error, info)
```

⬆ 返回顶部

232. 在哪些情况下，错误边界不会捕获错误?

以下是错误边界不起作用的情况：

1. 在事件处理器内。
2. **setTimeout** 或 **requestAnimationFrame** 回调中的异步代码。
3. 在服务端渲染期间。
4. 错误边界代码本身中引发错误时。

⬆ 返回顶部

233. 为什么事件处理器不需要错误边界?

错误边界不会捕获事件处理程序中的错误。与 render 方法或生命周期方法不同，在渲染期间事件处理器不会被执行或调用。

如果仍然需要在事件处理程序中捕获错误，请使用下面的常规 JavaScript `try/catch` 语句：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
  }

  handleClick = () => {
    try {
      // Do something that could throw
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <div onClick={this.handleClick}>Click Me</div>
  }
}
```

上面的代码使用普通的 JavaScript try/catch 块而不是错误边界来捕获错误。

⬆ 返回顶部

234. try catch 与错误边界有什么区别?

Try catch 块使用命令式代码，而错误边界则是使用在屏幕上呈现声明性代码。

例如，以下是使用声明式代码的 try/catch 块：

```
try {
  showButton();
} catch (error) {
  // ...
}
```

而错误边界包装的声明式代码如下：

```
<ErrorBoundary>
  <MyComponent />
</ErrorBoundary>
```

因此，如果在组件树深处某个位置组件的 **componentDidUpdate** 方法中，发生了由 **setState** 引发的错误，它仍然会正确地冒泡到最近的错误边界。

⬆ 返回顶部

235. React 16 中未捕获的错误的行为是什么?

在 React 16 中，未被任何错误边界捕获的错误将导致整个 React 组件树的卸载。这一决定背后的原因是，与其显示已损坏的界面，不如完全移除它。例如，对于支付应用程序来说，显示错误的金额比什么都不提供更糟糕。

⬆ 返回顶部

236. 放置错误边界的正确位置是什么?

错误边界使用的粒度由开发人员根据项目需要决定。你可以遵循这些方法中的任何一种：

1. 可以包装顶层路由组件以显示整个应用程序中常见的错误消息。
2. 你还可以将单个组件包装在错误边界中，以防止它们奔溃时影响到应用程序的其余部分。

⬆ 返回顶部

237. 从错误边界跟踪组件堆栈有什么好处?

除了错误消息和 JavaScript 堆栈，React 16 将使用错误边界的概念显示带有文件名和行号的组件堆栈。例如，BuggyCounter 组件显示组件堆栈信息：

⬆ 返回顶部

238. 在定义类组件时，什么是必须的方法?

在类组件中 render() 方法是唯一需要的方法。也就是说，对于类组件，除了 render() 方法之外的所有方法都是可选的。

⬆ 返回顶部

239. render 方法可能返回的类型是什么?

以下列表是 render 方法返回的类型：

1. **React elements:** 用于告诉 React 如何渲染 DOM 节点。它包括 HTML 元素，如 `<div />` 和用户定义的元素。
2. **Arrays and fragments:** 以数组的形式返回多个元素和包装多个元素的片段。
3. **Portals:** 将子元素渲染到不同的 DOM 子树中。
4. **String and numbers:** 在 DOM 中将字符串和数字都作为文本节点进行呈现。
5. **Booleans or null:** 不会渲染任何内容，但这些类型用于有条件地渲染内容。

⬆ 返回顶部

240. 构造函数的主要目的是什么?

使用构造函数主要有两个目的：

1. 通过将对象分配给 this.state 来初始化本地状态。
2. 用于为组件实例绑定事件处理方法。

例如，下面的代码涵盖了上述两种情况：

```
constructor(props) {
  super(props);
  // Don't call this.setState() here!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

⬆ 返回顶部

241. 是否必须为 React 组件定义构造函数?

不，这不是强制的。也就是说，如果你不需要初始化状态且不需要绑定方法，则你不需要为 React 组件实现一个构造函数。

⬆ 返回顶部

242. 什么是默认属性?

defaultProps 被定义为组件类上的属性，用于设置组件类默认的属性值。它只适用于 undefined 的属性，而不适用于 null 属性。例如，让我们为按钮组件创建默认的 color 属性：

```
class MyButton extends React.Component {
  // ...
}

MyButton.defaultProps = {
  color: 'red'
};
```

如果未设置 props.color，则会使用默认值 `red` 。也就是说，每当你试图访问 color 属性时，它都使用默认值。

```
render() {
   return <MyButton /> ; // props.color will be set to red
 }
```

注意： 如果你提供的是 null 值，它会仍然保留 null 值。

⬆ 返回顶部

243. 为什么不能在 componentWillUnmount 中调用 setState() 方法?

不应在 componentWillUnmount() 中调用 setState()，因为一旦卸载了组件实例，就永远不会再次装载它。

⬆ 返回顶部

244. getDerivedStateFromError 的目的是什么?

在子代组件抛出异常后会调用此生命周期方法。它以抛出的异常对象作为参数，并返回一个值用于更新状态。该生命周期方法的签名如下：

```
static getDerivedStateFromError(error)
```

让我们举一个包含上述生命周期方法的错误边界示例，来说明 getDerivedStateFromError 的目的：

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

⬆ 返回顶部

245. 当组件重新渲染时顺序执行的方法有哪些?

更新可能由属性或状态的更改引起。在重新渲染组件时，会按以下顺序调用下列方法。

1. static getDerivedStateFromProps()
2. shouldComponentUpdate()
3. render()
4. getSnapshotBeforeUpdate()
5. componentDidUpdate()

⬆ 返回顶部

246. 错误处理期间调用哪些方法?

在渲染期间，生命周期方法内或任何子组件的构造函数中出现错误时，将会调用以下方法：

1. static getDerivedStateFromError()
2. componentDidCatch()

⬆ 返回顶部

247. displayName 类属性的用途是什么?

displayName 被用于调试信息。通常，你不需要显式设置它，因为它是从定义组件的函数或类的名称推断出来的。如果出于调试目的或在创建高阶组件时显示不同的名称，可能需要显式设置它。

例如，若要简化调试，请选择一个显示名称，以表明它是 withSubscription HOC 的结果。

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName =
`WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}
```

⬆ 返回顶部

248. 支持 React 应用程序的浏览器有哪一些?

React 支持所有流行的浏览器，包括 Internet Explorer 9 和更高版本，但旧版本的浏览器（如IE 9 和 IE 10）需要一些 polyfill。如果你使用 **es5-shim and es5-sham** polyfill，那么它甚至支持不支持 ES5 方法的旧浏览器。

⬆ 返回顶部

249. unmountComponentAtNode 方法的目的是什么?

此方法可从 react-dom 包中获得，它从 DOM 中移除已装载的 React 组件，并清除其事件处理程序和状态。如果容器中没有装载任何组件，则调用此函数将不起任何作用。如果组件已卸载，则返回 true；如果没有要卸载的组件，则返回 false。该方法的签名如下：

```
ReactDOM.unmountComponentAtNode(container)
```

⬆ 返回顶部

250. 什么是代码拆分?

Code-Splitting 是 Webpack 和 Browserify 等打包工具所支持的一项功能，它可以创建多个 bundles，并可以在运行时动态加载。React 项目支持通过 dynamic import() 特性进行代码拆分。例如，在下面的代码片段中，它将使 moduleA.js 及其所有唯一依赖项作为单独的块，仅当用户点击 'Load' 按钮后才加载。

moduleA.js

```
const moduleA = 'Hello';

export { moduleA };
```

App.js

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

class App extends Component {
  handleClick = () => {
    import('./moduleA')
      .then(({ moduleA }) => {
        // Use moduleA
      })
      .catch(err => {
        // Handle failure
      });
  };

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Load</button>
      </div>
    );
  }
}

export default App;
```

⬆ 返回顶部

251. 严格模式有什么好处?

在下面的情况下， 将有所帮助：

1. 使用 **unsafe lifecycle methods** 标识组件。
2. 有关 **legacy string ref** API 用法发出警告。
3. 检测无法预测的 **side effects**。
4. 检测 **legacy context** API。
5. 有关已弃用的 findDOMNode 用法的警告。

⬆ 返回顶部

252. 什么是 Keyed Fragments ?

使用显式 <React.Fragment> 语法声明的片段可能具有 key 。一般用例是将集合映射到片段数组，如下所示，

```jsx
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Without the `key`, React will fire a key warning
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

注意： 键是唯一可以传递给 Fragment 的属性。将来，可能会支持其他属性，例如事件处理程序。

⬆ 返回顶部

253. React 支持所有的 HTML 属性么？

从 React 16 开始，完全支持标准或自定义 DOM 属性。由于 React 组件通常同时使用自定义和与 DOM 相关的属性，因此 React 与 DOM API 一样都使用 camelCase 约定。让我们对标准 HTML 属性采取一些措施：

```jsx
<div tabIndex="-1" />      // Just like node.tabIndex DOM API
<div className="Button" /> // Just like node.className DOM API
<input readOnly={true} />  // Just like node.readOnly DOM API
```

除了特殊情况外，这些属性的工作方式与相应的 HTML 属性类似。它还支持所有 SVG 属性。

⬆ 返回顶部

254. HOC 有哪些限制?

除了它的好处之外，高阶组件还有一些注意事项。 以下列出的几个注意事项:

1. **不要在渲染方法中使用HOC：** 建议不要将 HOC 应用于组件的 render 方法中的组件。

   ```jsx
   render() {
     // A new version of EnhancedComponent is created on every render
     // EnhancedComponent1 !== EnhancedComponent2
     const EnhancedComponent = enhance(MyComponent);
     // That causes the entire subtree to unmount/remount each time!
     return <EnhancedComponent />;
   }
   ```

   上述代码通过重新装载，将导致该组件及其所有子组件状态丢失，会影响到性能。正确的做法应该是在组件定义之外应用 HOC ，以便仅生成一次生成的组件

2. **静态方法必须复制：** 将 HOC 应用于组件时，新组件不具有原始组件的任何静态方法

```
// Define a static method
WrappedComponent.staticMethod = function() {/*...*/}
// Now apply a HOC
const EnhancedComponent = enhance(WrappedComponent);

// The enhanced component has no static method
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

您可以通过在返回之前将方法复制到输入组件上来解决此问题

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

3. **Refs 不会被往下传递** 对于HOC，您需要将所有属性传递给包装组件，但这对于 refs 不起作用。这是因为 ref 并不是一个类似于 key 的属性。在这种情况下，您需要使用 React.forwardRef API。

⬆ 返回顶部

255. 如何在 DevTools 中调试 forwardRefs?

React.forwardRef接受渲染函数作为参数，DevTools 使用此函数来确定为 ref 转发组件显示的内容。例如，如果您没有使用 displayName 属性命名 render 函数，那么它将在 DevTools 中显示为“ForwardRef”，

```
const WrappedComponent = React.forwardRef((props, ref) => {
  return <LogProps {...props} forwardedRef={ref} />;
});
```

但如果你命名 render 函数，那么它将显示为 **“ForwardRef(myFunction)”**

```
const WrappedComponent = React.forwardRef(
  function myFunction(props, ref) {
    return <LogProps {...props} forwardedRef={ref} />;
  }
);
```

作为替代方案，您还可以为 forwardRef 函数设置 displayName 属性，

```
function logProps(Component) {
  class LogProps extends React.Component {
    // ...
  }

  function forwardRef(props, ref) {
    return <LogProps {...props} forwardedRef={ref} />;
  }

  // Give this component a more helpful display name in DevTools.
```

```
    // e.g. "ForwardRef(logProps(MyComponent))"
    const name = Component.displayName || Component.name;
    forwardRef.displayName = `logProps(${name})`;

    return React.forwardRef(forwardRef);
  }
```

⬆ 返回顶部

256. 什么时候组件的 props 属性默认为 true?

如果没有传递属性值，则默认为 true。此行为可用，以便与 HTML 的行为匹配。例如，下面的表达式是等价的：

```
<MyInput autocomplete />

<MyInput autocomplete={true} />
```

注意： 不建议使用此方法，因为它可能与 ES6 对象 shorthand 混淆（例如，{name}，它是{ name:name } 的缩写）

⬆ 返回顶部

257. 什么是 NextJS 及其主要特征?

Next.js 是一个流行的轻量级框架，用于使用 React 构建静态和服务端渲染应用程序。它还提供样式和路由解决方案。以下是 NextJS 提供的主要功能：

1. 默认服务端渲染
2. 自动代码拆分以加快页面加载速度
3. 简单的客户端路由 (基于页面)
4. 基于 Webpack 的开发环境支持 (HMR)
5. 能够使用 Express 或任何其他 Node.js HTTP 服务器
6. 可自定义你自己的 Babel 和 Webpack 配置

⬆ 返回顶部

258. 如何将事件处理程序传递给组件?

可以将事件处理程序和其他函数作为属性传递给子组件。它可以在子组件中使用，如下所示：

```
<button onClick={this.handleClick}>
```

⬆ 返回顶部

259. 在渲染方法中使用箭头函数好么?

是的，你可以用。它通常是向回调函数传递参数的最简单方法。但在使用时需要优化性能。

```jsx
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>;
  }
}
```

注意： 组件每次渲染时，在 render 方法中的箭头函数都会创建一个新的函数，这可能会影响性能。

⬆ 返回顶部

260. 如何防止函数被多次调用?

如果你使用一个事件处理程序，如 **onClick or onScroll** 并希望防止回调被过快地触发，那么你可以限制回调的执行速度。

这可以通过以下可能的方式实现：

1. **Throttling:** 基于时间的频率进行更改。例如，它可以使用 lodash 的 _.throttle 函数。
2. **Debouncing:** 在一段时间不活动后发布更改。例如，可以使用 lodash 的 _.debounce 函数。
3. **RequestAnimationFrame throttling:** 基于 requestAnimationFrame 的更改。例如，可以使用 raf-schd。

> 注意：_.debounce， _.throttle 和 raf-schd 都提供了一个 cancel 方法来取消延迟回调。所以需要调用 componentWillUnmount，或者对代码进行检查来保证在延迟函数有效期间内组件始终挂载。

⬆ 返回顶部

261. JSX 如何防止注入攻击?

React DOM 会在渲染 JSX 中嵌入的任何值之前对其进行转义。因此，它确保你永远不能注入任何未在应用程序中显式写入的内容。

```jsx
const name = response.potentiallyMaliciousInput;
const element = <h1>{name}</h1>;
```

这样可以防止应用程序中的XSS（跨站点脚本）攻击。

⬆ 返回顶部

262. 如何更新已渲染的元素?

通过将新创建的元素传递给 ReactDOM 的 render 方法，可以实现 UI 更新。例如，让我们举一个滴答时钟的例子，它通过多次调用 render 方法来更新时间：

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}

setInterval(tick, 1000);
```

⬆ 返回顶部

263. 你怎么说 props 是只读的?

当你将组件声明为函数或类时，它决不能修改自己的属性。让我们来实现一个 capital 的函数：

```
function capital(amount, interest) {
   return amount + interest;
}
```

上面的函数称为“纯”函数，因为它不会尝试更改输入，并总是为相同的输入返回相同的结果。因此，React 有一条规则，即“所有 React 组件的行为都必须像纯函数一样”。

⬆ 返回顶部

264. 你认为状态更新是如何合并的?

当你在组件中调用 setState() 方法时，React 会将提供的对象合并到当前状态。例如，让我们以一个使用帖子和评论详细信息的作为状态变量的 Facebook 用户为例：

```
  constructor(props) {
    super(props);
    this.state = {
      posts: [],
      comments: []
    };
  }
```

现在，你可以独立调用 setState() 方法，单独更新状态变量：

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

如上面的代码段所示，`this.setState({comments})` 只会更新 comments 变量，而不会修改或替换 posts 变量。

⬆ 返回顶部

265. 如何将参数传递给事件处理程序?

在迭代或循环期间，向事件处理程序传递额外的参数是很常见的。这可以通过箭头函数或绑定方法实现。让我们以网格中更新的用户详细信息为例：

```
<button onClick={(e) => this.updateUser(userId, e)}>Update User details</button>
<button onClick={this.updateUser.bind(this, userId)}>Update User details</button>
```

在这两种方法中，合成参数 e 作为第二个参数传递。你需要在箭头函数中显式传递它，并使用 bind 方法自动转发它。

⬆ 返回顶部

266. 如何防止组件渲染?

你可以基于特定的条件通过返回 null 值来阻止组件的渲染。这样它就可以有条件地渲染组件。

```
function Greeting(props) {
  if (!props.loggedIn) {
    return null;
  }

  return (
    <div className="greeting">
      welcome, {props.name}
    </div>
  );
}
```

```
class User extends React.Component {
  constructor(props) {
    super(props);
    this.state = {loggedIn: false, name: 'John'};
  }
```

```
render() {
 return (
     <div>
       //Prevent component render if it is not loggedIn
       <Greeting loggedIn={this.state.loggedIn} />
       <UserDetails name={this.state.name}>
     </div>
 );
}
```

在上面的示例中，greeting 组件通过应用条件并返回空值跳过其渲染部分。

⬆ 返回顶部

267. 安全地使用索引作为键的条件是什么？

有三个条件可以确保，使用索引作为键是安全的：

1. 列表项是静态的，它们不会被计算，也不会更改。
2. 列表中的列表项没有 ids 属性。
3. 列表不会被重新排序或筛选。

⬆ 返回顶部

268. keys 是否需要全局唯一？

数组中使用的键在其同级中应该是唯一的，但它们不需要是全局唯一的。也就是说，你可以在两个不同的数组中使用相同的键。例如，下面的 book 组件在不同的组件中使用相同的数组：

```
function Book(props) {
  const index = (
    <ul>
      {props.pages.map((page) =>
        <li key={page.id}>
          {page.title}
        </li>
      )}
    </ul>
  );
  const content = props.pages.map((page) =>
    <div key={page.id}>
      <h3>{page.title}</h3>
      <p>{page.content}</p>
      <p>{page.pageNumber}</p>
    </div>
  );
  return (
    <div>
      {index}
      <hr />
      {content}
    </div>
  );
}
```

⬆ 返回顶部

269. 用于表单处理的流行选择是什么?

Formik 是一个用于 React 的表单库，它提供验证、跟踪访问字段和处理表单提交等解决方案。具体来说，你可以按以下方式对它们进行分类：

1. 获取表单状态输入和输出的值。
2. 表单验证和错误消息。
3. 处理表单提交。

它用于创建一个具有最小 API 的可伸缩、性能良好的表单助手，以解决令人讨厌的问题。

⬆ 返回顶部

270. formik 相对于其他 redux 表单库有什么优势?

下面是建议使用 formik 而不是 redux 表单库的主要原因：

1. 表单状态本质上是短期的和局部的，因此不需要在 redux（或任何类型的flux库）中跟踪它。
2. 每次按一个键，Redux-Form 都会多次调用整个顶级 Redux Reducer。这样就增加了大型应用程序的输入延迟。
3. 经过 gzip 压缩过的 Redux-Form 为 22.5 kB，而 Formik 只有 12.7 kB。

⬆ 返回顶部

271. 为什么不需要使用继承?

在 React 中，建议使用组合而不是继承来重用组件之间的代码。Props 和 composition 都为你提供了以一种明确和安全的方式自定义组件外观和行为所需的灵活性。但是，如果你希望在组件之间复用非 UI 功能，建议将其提取到单独的 JavaScript 模块中。之后的组件导入它并使用该函数、对象或类，而不需扩展它。

⬆ 返回顶部

272. 我可以在 React 应用程序中可以使用 web components 么?

是的，你可以在 React 应用程序中使用 Web Components。尽管许多开发人员不会使用这种组合方式，但如果你使用的是使用 Web Components 编写的第三方 UI 组件，则可能需要这种组合。例如，让我们使用 Vaadin 提供的 Web Components 日期选择器组件：

```
import React, { Component } from 'react';
import './App.css';
import '@vaadin/vaadin-date-picker';

class App extends Component {
  render() {
    return (
      <div className="App">
        <vaadin-date-picker label="When were you born?"></vaadin-date-picker>
      </div>
    );
  }
}

export default App;
```

⬆ 返回顶部

273. 什么是动态导入?

动态导入语法是 ECMAScript 提案，目前不属于语言标准的一部分。它有望在不久的将来被采纳。在你的应用程序中，你可以使用动态导入来实现代码拆分。让我们举一个加法的例子：

1. **Normal Import**

```
import { add } from './math';
console.log(add(10, 20));
```

2. **Dynamic Import**

```
import("./math").then(math => {
  console.log(math.add(10, 20));
});
```

⬆ 返回顶部

274. 什么是 loadable 组件?

如果你想要在服务端渲染的应用程序中实现代码拆分，建议使用 Loadable 组件，因为 React.lazy 和 Suspense 还不可用于服务器端渲染。Loadable 允许你将动态导入的组件作为常规的组件进行渲染。让我们举一个例子：

```
import loadable from '@loadable/component'

const OtherComponent = loadable(() => import('./OtherComponent'))

function MyComponent() {
  return (
    <div>
      <OtherComponent />
    </div>
  )
}
```

现在，其他组件将以单独的包进行加载。

⬆ 返回顶部

275. 什么是 suspense 组件?

如果父组件在渲染时包含 dynamic import 的模块尚未加载完成，在此加载过程中，你必须使用一个 loading 指示器显示后备内容。这可以使用 **Suspense** 组件来实现。例如，下面的代码使用 Suspense 组件：

```jsx
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

正如上面的代码中所展示的，懒加载的组件被包装在 Suspense 组件中。

⬆ 返回顶部

276. 什么是基于路由的代码拆分?

进行代码拆分的最佳位置之一是路由。整个页面将立即重新渲染，因此用户不太可能同时与页面中的其他元素进行交互。因此，用户体验不会受到干扰。让我们以基于路由的网站为例，使用像 React Router 和 React.lazy 这样的库：

```jsx
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import React, { Suspense, lazy } from 'react';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
);
```

在上面的代码中，代码拆分将发生在每个路由层级。

⬆ 返回顶部

277. 举例说明如何使用 context?

Context 旨在共享可被视为全局的数据，用于 React 组件树。例如，在下面的代码中，允许手动通过一个 theme 属性来设置按钮组件的样式。

```jsx
//Lets create a context with a default theme value "luna"
const ThemeContext = React.createContext('luna');
// Create App component where it uses provider to pass theme value in the tree
class App extends React.Component {
  render() {
    return (
```

```
      <ThemeContext.Provider value="nova">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
// A middle component where you don't need to pass theme prop anymore
function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}
// Lets read theme value in the button component to use
class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

⬆ 返回顶部

278. 在 context 中默认值的目的是什么?

当在组件树中的组件没有匹配到在其上方的 Provider 时，才会使用 defaultValue 参数。这有助于在不包装组件的情况下单独测试组件。下面的代码段提供了默认的主题值 Luna。

```
const defaultTheme = "Luna";
const MyContext = React.createContext(defaultTheme);
```

⬆ 返回顶部

279. 你是怎么使用 contextType?

ContextType 用于消费 context 对象。ContextType 属性可以通过两种方式使用：

1. **contextType as property of class:** 可以为类的 contextType 属性分配通过 React.createContext() 创建的 context 对象。之后，你可以在任何生命周期方法和 render 函数中使用 `this.context` 引用该上下文类型最近的当前值。

让我们在 MyClass 上按如下方式设置 contextType 属性：

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* perform a side-effect at mount using the value of MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
```

```
    /* ... */
  }
  render() {
    let value = this.context;
    /* render something based on the value of MyContext */
  }
}
MyClass.contextType = MyContext;
```

2. **Static field** 你可以使用静态类属性来初始化 contextType 属性：

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* render something based on the value */
  }
}
```

⬆ 返回顶部

280. 什么是 consumer?

Consumer 是一个订阅上下文更改的 React 组件。它需要一个函数作为子元素，该函数接收当前上下文的值作为参数，并返回一个 React 元素。传递给函数 value 参数的参数值将等于在组件树中当前组件最近的 Provider 元素的 value 属性值。举个简单的例子：

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

⬆ 返回顶部

281. 在使用 context 时，如何解决性能方面的问题?

Context 使用引用标识来确定何时重新渲染，当 Provider 的父元素重新渲染时，会有一些问题即可能会在 Consumers 中触发无任何意图的渲染。 例如，下面的代码将在每次 Provider 重新渲染时，重新渲染所有的 Consumers，这是因为渲染 Provider 时，始终会为 value 属性创建一个新的对象：

```
class App extends React.Component {
  render() {
    return (
      <Provider value={{something: 'something'}}>
        <Toolbar />
      </Provider>
    );
  }
}
```

可以通过把 value 的值提升到父状态中来解决这个问题：

```
class App extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.state = {
      value: {something: 'something'},
    };
  }

  render() {
    return (
      <Provider value={this.state.value}>
        <Toolbar />
      </Provider>
    );
  }
}
```

⬆ 返回顶部

282. 在 HOCs 中 forward ref 的目的是什么?

因为 ref 不是一个属性，所以 Refs 不会被传递。就像 **key** 一样，React 会以不同的方式处理它。如果你将 ref 添加到 HOC，则该 ref 将引用最外层的容器组件，而不是包装的组件。在这种情况下，你可以使用 Forward Ref API。例如，你可以使用 React.forwardRef API 显式地将 refs 转发的内部的 FancyButton 组件。

以下的 HOC 会记录所有的 props 变化：

```
function logProps(Component) {
  class LogProps extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('old props:', prevProps);
      console.log('new props:', this.props);
    }

    render() {
      const {forwardedRef, ...rest} = this.props;

      // Assign the custom prop "forwardedRef" as a ref
      return <Component ref={forwardedRef} {...rest} />;
    }
  }

  return React.forwardRef((props, ref) => {
    return <LogProps {...props} forwardedRef={ref} />;
  });
}
```

让我们使用这个 HOC 来记录所有传递到我们 “fancy button” 组件的属性：

```
class FancyButton extends React.Component {
  focus() {
    // ...
  }

  // ...
}
export default logProps(FancyButton);
```

现在让我们创建一个 ref 并将其传递给 FancyButton 组件。在这种情况下，你可以聚焦到 button 元素上。

```
import FancyButton from './FancyButton';

const ref = React.createRef();
ref.current.focus();
<FancyButton
  label="Click Me"
  handleClick={handleClick}
  ref={ref}
/>;
```

⬆ 返回顶部

283. ref 参数对于所有函数或类组件是否可用?

常规函数或类组件不会接收到 ref 参数，并且 ref 在 props 中也不可用。只有在使用 React.forwardRef 定义组件时，才存在第二个 ref 参数。

⬆ 返回顶部

284. 在组件库中当使用 forward refs 时，你需要额外的注意?

当你开始在组件库中使用 forwardRef 时，你应该将其视为一个破坏性的更改，并为库发布一个新的主要版本。这是因为你的库可能具有不同的行为，如已分配了哪些引用，以及导出哪些类型。这些更改可能会破坏依赖于旧行为的应用程序和其他库。

⬆ 返回顶部

285. 如何在没有 ES6 的情况下创建 React 类组件

如果你不使用 ES6，那么你可能需要使用 create-react-class 模块。对于默认属性，你需要在传递对象上定义 getDefaultProps() 函数。而对于初始状态，必须提供返回初始状态的单独 getInitialState 方法。

```
var Greeting = createReactClass({
  getDefaultProps: function() {
      return {
        name: 'Jhohn'
      };
    },
  getInitialState: function() {
      return {message: this.props.message};
    },
  handleClick: function() {
```

```
      console.log(this.state.message);
    },
    render: function() {
      return <h1>Hello, {this.props.name}</h1>;
    }
  });
```

注意： 如果使用 createReactClass，则所有方法都会自动绑定。也就是说，你不需要在事件处理程序的构造函数中使用 .bind(this)。

⬆ 返回顶部

286. 是否可以在没有 JSX 的情况下使用 React?

是的，使用 React 不强制使用 JSX。实际上，当你不想在构建环境中配置编译环境时，这是很方便的。每个 JSX 元素只是调用 React.createElement(component, props, ...children) 的语法糖。例如，让我们来看一下使用 JSX 的 greeting 示例：

```
class Greeting extends React.Component {
  render() {
    return <div>Hello {this.props.message}</div>;
  }
}

ReactDOM.render(
  <Greeting message="World" />,
  document.getElementById('root')
);
```

你可以在没有 JSX 的情况下编写相同的功能，如下所示：

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

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

⬆ 返回顶部

287. 什么是差异算法?

React 需要使用算法来了解如何有效地更新 UI 以匹配最新的树。差异算法将生成将一棵树转换为另一棵树的最小操作次数。然而，算法具有 O(n3) 的复杂度，其中 n 是树中元素的数量。在这种情况下，对于显示 1000 个元素将需要大约 10 亿个比较。这太昂贵了。相反，React 基于两个假设实现了一个复杂度为 O(n) 的算法：

1. 两种不同类型的元素会产生不同的树结构。
2. 开发者可以通过一个 key 属性，标识哪些子元素可以在不同渲染中保持稳定。

⬆ 返回顶部

288. 差异算法涵盖了哪些规则?

在区分两棵树时，React 首先比较两个根元素。根据根元素的类型，行为会有所不同。它在重构算法中涵盖了以下规则：

1. **不同类型的元素：** 每当根元素具有不同的类型时，React 将移除旧树并从头开始构建新树。例如，元素 到 ，或从

   到 的不同类型的元素引导完全重建。

2. 相同类型的DOM元素： 当比较两个相同类型的 React DOM 元素时，React 查看两者的属性，保持相同的底层 DOM 节点，并仅更新已更改的属性。让我们以相同的 DOM 元素为例，除了 className 属性，

```
<div className="show" title="ReactJS" />

<div className="hide" title="ReactJS" />
```

3. 相同类型的组件元素：

   当组件更新时，实例保持不变，以便在渲染之间保持状态。React 更新底层组件实例的 props 以匹配新元素，并在底层实例上调用 componentWillReceiveProps() 和 componentWillUpdate()。之后，调用 render() 方法，diff 算法对前一个结果和新结果进行递归。

4. 递归子节点： 当对 DOM 节点的子节点进行递归时，React 会同时迭代两个子节点列表，并在出现差异时生成变异。例如，在子节点末尾添加元素时，在这两个树之间进行转换效果很好。

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

5. 处理 Key：

React支持 key 属性。当子节点有 key 时，React 使用 key 将原始树中的子节点与后续树中的子节点相匹配。例如，添加 key 可以使树有效地转换，

```jsx
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

⬆ 返回顶部

289. 你什么时候需要使用 refs?

这里是 refs 的一些使用场景：

1. 管理聚焦、文本选择或媒体播放。
2. 触发命令式动画。
3. 与第三方 DOM 库集成。

⬆ 返回顶部

290. 对于渲染属性来说是否必须将 prop 属性命名为 render?

即使模式名为 `render props`，你也不必使用名为 render 的属性名来使用此模式。也就是说，组件用于知道即将渲染内容的任何函数属性，在技术上都是一个 `render props`。让我们举一个名为 children 渲染属性的示例：

```jsx
<Mouse children={mouse => (
  <p>The mouse position is {mouse.x}, {mouse.y}</p>
)}/>
```

实际上，以上的 children 属性不一定需要在 JSX 元素的 `attributes` 列表中命名。反之，你可以将它直接放在元素内部：

```jsx
<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

当使用上述的技术，需要在 propTypes 中明确声明 children 必须为函数类型：

```jsx
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

⬆ 返回顶部

291. 在 Pure Component 中使用渲染属性会有什么问题?

如果在渲染方法中创建函数，则会否定纯组件的用途。因为浅属性比较对于新属性总是返回 false，在这种情况下，每次渲染都将为渲染属性生成一个新值。你可以通过将渲染函数定义为实例方法来解决这个问题。

⬆ 返回顶部

292. 如何使用渲染属性创建 HOC?

可以使用带有渲染属性的常规组件实现大多数高阶组件（HOC）。例如，如果希望使用 withMouse HOC 而不是 `<Mouse>` 组件，则你可以使用带有渲染属性的常规 `<Mouse>` 组件轻松创建一个 HOC 组件。

```
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}/>
      );
    }
  }
}
```

⬆ 返回顶部

293. 什么是 windowing 技术?

Windowing 是一种技术，它在任何给定时间只呈现一小部分行，并且可以显著减少重新呈现组件所需的时间以及创建的 DOM 节点的数量。如果应用程序呈现长的数据列表，则建议使用此技术。react-window 和 react-virtualized 都是常用的 windowing 库，它提供了几个可重用的组件，用于显示列表、网格和表格数据。

⬆ 返回顶部

294. 你如何在 JSX 中打印 falsy 值?

Falsy 值比如 false，null，undefined 是有效的子元素，但它们不会呈现任何内容。如果仍要显示它们，则需要将其转换为字符串。我们来举一个如何转换为字符串的例子：

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

⬆ 返回顶部

295. portals 的典型使用场景是什么?

当父组件拥有 `overflow: hidden` 或含有影响堆叠上下文的属性（z-index、position、opacity 等样式），且需要脱离它的容器进行展示时，React portal 就非常有用。例如，对话框、全局消息通知、悬停卡和工具提示。

⬆ 返回顶部

296. 如何设置非受控组件的默认值?

在 React 中，表单元素的属性值将覆盖其 DOM 中的值。对于非受控组件，你可能希望能够指定其初始值，但不会控制后续的更新。要处理这种情形，你可以指定一个 **defaultValue** 属性来取代 **value** 属性。

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        User Name:
        <input
          defaultValue="John"
          type="text"
          ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
```

这同样适用于 `select` 和 `textArea` 输入框。但对于 `checkbox` 和 `radio` 控件，需要使用 **defaultChecked**。

⬆ 返回顶部

297. 你最喜欢的 React 技术栈是什么?

尽管技术栈因开发人员而异，但最流行的技术栈用于 React boilerplate 项目代码中。它主要使用 redux 和 redux saga 进行状态管理和具有副作用的异步操作，使用 react-router 进行路由管理，使用 styled-components 库开发 React 组件，使用 axios 调用 REST api，以及其他支持的技术栈，如 webpack、reseselect、esnext、babel 等。

你可以克隆 https://github.com/react-boilerplate/react-boilerplate 并开始开发任何新的 React 项目。

⬆ 返回顶部

298. Real DOM 和 Virtual DOM 有什么区别?

以下是Real DOM和Virtual DOM之间的主要区别：

Real DOM	Virtual DOM
更新速度慢	更新速度快
DOM 操作非常昂贵	DOM 操作非常简单
可以直接更新 HTML	你不能直接更新 HTML
造成太多内存浪费	更少的内存消耗
如果元素更新了，创建新的 DOM 节点	如果元素更新，则更新 JSX 元素

⬆ 返回顶部

299. 如何为 React 应用程序添加 bootstrap?

Bootstrap 可以通过三种可能的方式添加到 React 应用程序中：

1. 使用 Bootstrap CDN: 这是添加 bootstrap 最简单的方式。在 head 标签中添加 bootstrap 相应的 CSS 和 JS 资源。

2. 把 Bootstrap 作为依赖项： 如果你使用的是构建工具或模块绑定器（如Webpack），那么这是向 React 应用程序添加 bootstrap 的首选选项。

```
npm install bootstrap
```

3. 使用 React Bootstrap 包: 在这种情况下，你可以将 Bootstrap 添加到我们的 React 应用程序中，方法是使用一个以 React 组件形式对 Bootstrap 组件进行包装后包。下面的包在此类别中很流行：

    1. react-bootstrap
    2. reactstrap

⬆ 返回顶部

300. 你能否列出使用 React 作为前端框架的顶级网站或应用程序?

以下是使用 React 作为前端框架的前 10 个网站：

1. Facebook
2. Uber
3. Instagram
4. WhatsApp
5. Khan Academy
6. Airbnb
7. Dropbox
8. Flipboard
9. Netflix
10. PayPal

⬆ 返回顶部

301. 是否建议在 React 中使用 CSS In JS 技术?

React 对如何定义样式没有任何意见，但如果你是初学者，那么好的起点是像往常一样在单独的 *.css 文件中定义样式，并使用类名引用它们。此功能不是 React 的一部分，而是来自第三方库。但是如果你想尝试不同的方法（JS中的CSS），那么 styled-components 库是一个不错的选择。

⬆ 返回顶部

302. 我需要用 hooks 重写所有类组件吗?

不需要。但你可以在某些组件（或新组件）中尝试使用 hooks，而无需重写任何已存在的代码。因为在 ReactJS 中目前没有移除 classes 的计划。

⬆ 返回顶部

303. 如何使用 React Hooks 获取数据?

名为 useEffect 的 effect hook 可用于使用 axios 从 API 中获取数据，并使用 useState 钩子提供的更新函数设置组件本地状态中的数据。让我们举一个例子，它从 API 中获取 react 文章列表。

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [data, setData] = useState({ hits: [] });
```

```
  useEffect(async () => {
    const result = await axios(
      'http://hn.algolia.com/api/v1/search?query=react',
    );

    setData(result.data);
  }, []);

  return (
    <ul>
      {data.hits.map(item => (
        <li key={item.objectID}>
          <a href={item.url}>{item.title}</a>
        </li>
      ))}
    </ul>
  );
}

export default App;
```

记住，我们为 effect hook 提供了一个空数组作为第二个参数，以避免在组件更新时再次激活它，它只会在组件挂载时被执行。比如，示例中仅在组件挂载时获取数据。

⬆ 返回顶部

304. Hooks 是否涵盖了类的所有用例?

Hooks 并没有涵盖类的所有用例，但是有计划很快添加它们。目前，还没有与不常见的 getSnapshotBeforeUpdate 和componentDidCatch 生命周期等效的钩子。

⬆ 返回顶部