**React React Narive 面试题**

**React 部分**

**1.调用 setState 之后发生了什么？**

在代码中调用setState函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发所谓的调和过程（Reconciliation）。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个UI界面。在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

**2.React 中 Element 与 Component 的区别是？**

简单而言，React Element 是描述屏幕上所见内容的数据结构，是对于 UI 的对象表述。典型的 React Element 就是利用 JSX 构建的声明式代码片然后被转化为createElement的调用组合。而 React Component 则是可以接收参数输入并且返回某个 React Element 的函数或者类。更多介绍可以参考React Elements vs React Components。

**3.在什么情况下你会优先选择使用 Class Component 而不是 Functional Component？**

在组件需要包含内部状态或者使用到生命周期函数的时候使用 Class Component ，否则使用函数式组件。

**4.React 中 refs 的作用是什么？**

注意，根据React最新文档，下面这种用法已经被弃用了，统一改为回调函数模式

this.refs.textInput

Refs 是 React 提供给我们的安全访问 DOM 元素或者某个组件实例的句柄。我们可以为元素添加ref属性然后在回调函数中接受该元素在 DOM 树中的句柄，该值会作为回调函数的第一个参数返回：

*class CustomForm extends Component {*

*handleSubmit = () => {*

*console.log("Input Value: ", this.input.value)*

*}*

*render () {*

*return (*

*<form onSubmit={this.handleSubmit}>*

*<input*

*type='text'*

*ref={(input) => this.input = input} />*

*<button type='submit'>Submit</button>*

*</form>*

*)*

*}*

*}*

上述代码中的input域包含了一个ref属性，该属性声明的回调函数会接收input对应的 DOM 元素，我们将其绑定到this指针以便在其他的类函数中使用。另外值得一提的是，refs 并不是类组件的专属，函数式组件同样能够利用闭包暂存其值：

*function CustomForm ({handleSubmit}) {*

*let inputElement*

*return (*

*<form onSubmit={() => handleSubmit(inputElement.value)}>*

*<input*

*type='text'*

*ref={(input) => inputElement = input} />*

*<button type='submit'>Submit</button>*

*</form>*

*)*

*}*

**5.React 中 keys 的作用是什么？**

*Keys 是 React 用于追踪哪些列表中元素被修改、被添加或者被移除的辅助标识。*

*render () {*

*return (*

*<ul>*

*{this.state.todoItems.map(({task, uid}) => {*

*return <li key={uid}>{task}</li>*

*})}*

*</ul>*

*)*

*}*

*在开发过程中，我们需要保证某个元素的 key 在其同级元素中具有唯一性。在 React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染。此外，React 还需要借助 Key 值来判断元素与本地状态的关联关系，因此我们绝不可忽视转换函数中 Key 的重要性。*

**6.如果你创建了类似于下面的Twitter元素，那么它相关的类定义是啥样子的？**

<Twitter username='tylermcginnis33'>

{(user) => user === null

? <Loading />

: <Badge info={user} />}

</Twitter>

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

import fetchUser from 'twitter'

// fetchUser take in a username returns a promise

// which will resolve with that username's data.

class Twitter extends Component {

// finish this

}

如果你还不熟悉回调渲染模式（Render Callback Pattern），这个代码可能看起来有点怪。这种模式中，组件会接收某个函数作为其子组件，然后在渲染函数中以props.children进行调用：

*import React, { Component, PropTypes } from 'react'*

*import fetchUser from 'twitter'*

*class Twitter extends Component {*

*state = {*

*user: null,*

*}*

*static propTypes = {*

*username: PropTypes.string.isRequired,*

*}*

*componentDidMount () {*

*fetchUser(this.props.username)*

*.then((user) => this.setState({user}))*

*}*

*render () {*

*return this.props.children(this.state.user)*

*}*

*}*

这种模式的优势在于将父组件与子组件解耦和，父组件可以直接访问子组件的内部状态而不需要再通过Props传递，这样父组件能够更为方便地控制子组件展示的UI界面。将原本展示的Badge替换为Profile，我们可以轻易地修改下回调函数即可：

*<Twitter username='tylermcginnis33'>*

*{(user) => user === null*

*? <Loading />*

*: <Profile info={user} />}*

*</Twitter>*

**7.Controlled Component 与 Uncontrolled Component 之间的区别是什么？**

React 的核心组成之一就是能够维持内部状态的自治组件，不过当我们引入原生的HTML表单元素时（input,select,textarea 等），我们是否应该将所有的数据托管到 React 组件中还是将其仍然保留在 DOM 元素中呢？这个问题的答案就是受控组件与非受控组件的定义分割。受控组件（Controlled Component）代指那些交由 React 控制并且所有的表单数据统一存放的组件。譬如下面这段代码中username变量值并没有存放到DOM元素中，而是存放在组件状态数据中。任何时候我们需要改变username变量值时，我们应当调用setState函数进行修改。

*class ControlledForm extends Component {*

*state = {*

*username: ''*

*}*

*updateUsername = (e) => {*

*this.setState({*

*username: e.target.value,*

*})*

*}*

*handleSubmit = () => {}*

*render () {*

*return (*

*<form onSubmit={this.handleSubmit}>*

*<input*

*type='text'*

*value={this.state.username}*

*onChange={this.updateUsername} />*

*<button type='submit'>Submit</button>*

*</form>*

*)*

*}*

*}*

而非受控组件（Uncontrolled Component）则是由DOM存放表单数据，并非存放在 React 组件中。我们可以使用 refs 来操控DOM元素：

*class UnControlledForm extends Component {*

*handleSubmit = () => {*

*console.log("Input Value: ", this.input.value)*

*}*

*render () {*

*return (*

*<form onSubmit={this.handleSubmit}>*

*<input*

*type='text'*

*ref={(input) => this.input = input} />*

*<button type='submit'>Submit</button>*

*</form>*

*)*

*}*

*}*

竟然非受控组件看上去更好实现，我们可以直接从 DOM 中抓取数据，而不需要添加额外的代码。不过实际开发中我们并不提倡使用非受控组件，因为实际情况下我们需要更多的考虑表单验证、选择性的开启或者关闭按钮点击、强制输入格式等功能支持，而此时我们将数据托管到 React 中有助于我们更好地以声明式的方式完成这些功能。引入 React 或者其他 MVVM 框架最初的原因就是为了将我们从繁重的直接操作 DOM 中解放出来。

**8.在生命周期中的哪一步你应该发起 AJAX 请求？**

我们应当将AJAX 请求放到 componentDidMount 函数中执行，主要原因有下：

React 下一代调和算法 Fiber 会通过开始或停止渲染的方式优化应用性能，其会影响到 componentWillMount 的触发次数。对于 componentWillMount 这个生命周期函数的调用次数会变得不确定，React 可能会多次频繁调用 componentWillMount。如果我们将 AJAX 请求放到 componentWillMount 函数中，那么显而易见其会被触发多次，自然也就不是好的选择。

如果我们将 AJAX 请求放置在生命周期的其他函数中，我们并不能保证请求仅在组件挂载完毕后才会要求响应。如果我们的数据请求在组件挂载之前就完成，并且调用了setState函数将数据添加到组件状态中，对于未挂载的组件则会报错。而在 componentDidMount 函数中进行 AJAX 请求则能有效避免这个问题。

**9.shouldComponentUpdate 的作用是啥以及为何它这么重要？**

shouldComponentUpdate 允许我们手动地判断是否要进行组件更新，根据组件的应用场景设置函数的合理返回值能够帮我们避免不必要的更新。

如何告诉 React 它应该编译生产环境版本？

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

**10.为什么我们需要使用 React 提供的 Children API 而不是 JavaScript 的 map？**

props.children并不一定是数组类型，譬如下面这个元素：

*<Parent>*

*<h1>Welcome.</h1>*

*</Parent>*

如果我们使用props.children.map函数来遍历时会受到异常提示，因为在这种情况下props.children是对象（object）而不是数组（array）。React 当且仅当超过一个子元素的情况下会将props.children设置为数组，就像下面这个代码片：

*<Parent>*

*<h1>Welcome.</h1>*

*<h2>props.children will now be an array</h2>*

*</Parent>*

这也就是我们优先选择使用React.Children.map函数的原因，其已经将props.children不同类型的情况考虑在内了。

**11.概述下 React 中的事件处理逻辑**

为了解决跨浏览器兼容性问题，React 会将浏览器原生事件（Browser Native Event）封装为合成事件（SyntheticEvent）传入设置的事件处理器中。这里的合成事件提供了与原生事件相同的接口，不过它们屏蔽了底层浏览器的细节差异，保证了行为的一致性。另外有意思的是，React 并没有直接将事件附着到子元素上，而是以单一事件监听器的方式将所有的事件发送到顶层进行处理。这样 React 在更新 DOM 的时候就不需要考虑如何去处理附着在 DOM 上的事件监听器，最终达到优化性能的目的。

**12.createElement 与 cloneElement 的区别是什么？**

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

**13.传入 setState 函数的第二个参数的作用是什么？**

该函数会在setState函数调用完成并且组件开始重渲染的时候被调用，我们可以用该函数来监听渲染是否完成：

*this.setState(*

*{ username: 'tylermcginnis33' },*

*() => console.log('setState has finished and the component has re-rendered.')*

*)*

**14 setState 同步更新**

为了提高性能React将setState设置为批次更新，即是异步操作函数，并不能以顺序控制流的方式设置某些事件，我们也不能依赖于this.state来计算未来状态。典型的譬如我们希望在从服务端抓取数据并且渲染到界面之后，再隐藏加载进度条或者外部加载提示：

*componentDidMount() {*

*fetch('https://example.com')*

*.then((res) => res.json())*

*.then(*

*(something) => {*

*this.setState({ something });*

*StatusBar.setNetworkActivityIndicatorVisible(false);*

*}*

*);*

*}*

因为setState函数并不会阻塞等待状态更新完毕，因此setNetworkActivityIndicatorVisible有可能先于数据渲染完毕就执行。我们可以选择在componentWillUpdate与componentDidUpdate这两个生命周期的回调函数中执行setNetworkActivityIndicatorVisible，但是会让代码变得破碎，可读性也不好。实际上在项目开发中我们更频繁遇见此类问题的场景是以某个变量控制元素可见性：

this.setState({showForm : !this.state.showForm});

我们预期的效果是每次事件触发后改变表单的可见性，但是在大型应用程序中如果事件的触发速度快于setState的更新速度，那么我们的值计算完全就是错的。本节就是讨论两种方式来保证setState的同步更新。

*完成回调*

*setState函数的第二个参数允许传入回调函数，在状态更新完毕后进行调用，譬如：*

*this.setState({*

*load: !this.state.load,*

*count: this.state.count + 1*

*}, () => {*

*console.log(this.state.count);*

*console.log('加载完成')*

*});*

这里的回调函数用法相信大家很熟悉，就是JavaScript异步编程相关知识，我们可以引入*Promise来封装setState:*

*setStateAsync(state) {*

*return new Promise((resolve) => {*

*this.setState(state, resolve)*

*});*

*}*

setStateAsync 返回的是Promise对象，在调用时我们可以使用Async/Await语法来优化代码风格：

*async componentDidMount() {*

*StatusBar.setNetworkActivityIndicatorVisible(true)*

*const res = await fetch('https://api.ipify.org?format=json')*

*const {ip} = await res.json()*

*await this.setStateAsync({ipAddress: ip})*

*StatusBar.setNetworkActivityIndicatorVisible(false)*

*}*

*这里我们就可以保证在setState渲染完毕之后调用外部状态栏将网络请求状态修改为已结*束，整个组件的完整定义为：

*class AwesomeProject extends Component {*

*state = {}*

*setStateAsync(state) {*

*...*

*}*

*async componentDidMount() {*

*...*

*}*

*render() {*

*return (*

*<View style={styles.container}>*

*<Text style={styles.welcome}>*

*My IP is {this.state.ipAddress || 'Unknown'}*

*</Text>*

*</View>*

*);*

*}*

*}*

传入状态计算函数

除了使用回调函数的方式监听状态更新结果之外，React还允许我们传入某个状态计算函数而不是对象来作为第一个参数。状态计算函数能够为我们提供可信赖的组件的State与Props值，即会自动地将我们的状态更新操作添加到队列中并等待前面的更新完毕后传入最新的状态值：

*this.setState(function(prevState, props){*

*return {showForm: !prevState.showForm}*

*});*

这里我们以简单的计数器为例，我们希望用户点击按钮之后将计数值连加两次，基本的组件为：

*class Counter extends React.Component{*

*constructor(props){*

*super(props);*

*this.state = {count : 0}*

*this.incrementCount = this.incrementCount.bind(this)*

*}*

*incrementCount(){*

*...*

*}*

*render(){*

*return <div>*

*<button onClick={this.incrementCount}>Increment</button>*

*<div>{this.state.count}</div>*

*</div>*

*}*

*}*

直观的写法我们可以连续调用两次setState函数，这边的用法可能看起来有点怪异，不过更多的是为了说明异步更新带来的数据不可预测问题。

*incrementCount(){*

*this.setState({count : this.state.count + 1})*

*this.setState({count : this.state.count + 1})*

*}*

上述代码的效果是每次点击之后计数值只会加1，实际上第二个setState并没有等待第一个setState执行完毕就开始执行了，因此其依赖的当前计数值完全是错的。我们当然可以使用上文提及的setStateAsync来进行同步控制，不过这里我们使用状态计算函数来保证同步性：

*incrementCount(){*

*this.setState((prevState, props) => ({*

*count: prevState.count + 1*

*}));*

*this.setState((prevState, props) => ({*

*count: prevState.count + 1*

*}));*

*}*

这里的第二个setState传入的prevState值就是第一个setState执行完毕之后的计数值，也顺利保证了连续自增两次。