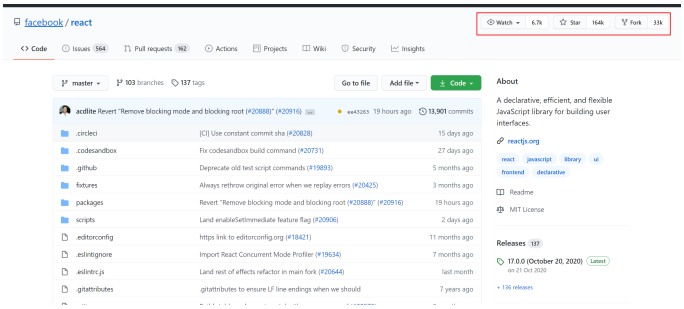
React基础知识分享

React是Facebook推出的前端框架，目前主流的前端框架之一，国内许多互联网公司都使用React作为前端主要技术栈。

React具有声明式编码、组件化编码、高效的Diffing算法等特点。

React的核心思想就是 封装组件，各个组件维护自己的状态和UI，当状态改变时，自动重新渲染整个组件。基于这种方式就实现了不再需要像JavaScript代码找到某个Dom元素，然后操作Dom来更改UI。



官网: https://reactjs.org/

github: https://github.com/facebook/react/

React主要的概念

React组件

- ##### 类组件

```
react import React from 'react' export default class TestIndex extends React.Component{ comstructor(props){ super(props); this.state = {} } render(){ return <div>类组件</div> } }
```

- 函数组件（无状态组件）

```
react import React from 'react' export defalut FuncIndex = (props) => { return <div>{props.name}</div> }
```

JSX

JSX就是JavaScript和XML结合的一种格式。React使用JSX，方便的利用HTML语法创建虚拟DOM，当遇到{}会当成JavaScript来解析，简化了工作。

Virtual DOM

关于virtual Dom,在React的设计中，每个React组件都是用Virtual Dom渲染的，可以理解为JavaScript实现的内存DOM抽象。React在Virtual Dom实现了Diff算法，通过Diff算法，能高效找到变更的Dom节点，刷新到实际的Dom上。

响应式设计和数据的绑定

React不推荐直接操作DOM元素，而是通过数据进行驱动，改变界面的效果。React会根据数据的变化，自动完成界面的改变。因此，在使用React开发时，无需关注DOM相关的操作，只需要关注数据的操作就可以。

```
jsx constructor(props){ super(props) //调用父类的构造函数，固定写法 this.state={ inputValue:'1111' , // } } onInputChange=(e)=>{ this.setState({ inputValue: e.target.value }) }
```

```
jsx <input value={this.state.testValue} onChange={this.onInputChange}></input>
```

React生命周期

在组件创建、组件属性更新、组件被销毁的过程中，总是伴随着各种各样的函数执行，这些在组件特定时期，被触发执行的函数，统称为组件的生命周期函数。

1. **初始化阶段（Mounting）：** 初始化，在构造函数中初始化props和state数据；
2. **加载阶段（Mounting）：** 在组件初始化时执行，有一个显著的特点：创建阶段生命周期函数在组件的一辈子中只执行一次；
3. **更新阶段（Updating）：** 属性和状态改变时执行，根据组件的state和props的改变，有选择性的触发0次或多次；
4. **卸载阶段（Unmounting）：** 在组件对象销毁时执行，一辈子只执行一次；

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

export default class OldReactComponent extends Component { constructor(props) { super(props) // getDefaultProps：接收初始props // getInitialState：初始化state } state = {

} componentWillMount() { // 组件挂载前触发

} render() { return (

## Old React.Component

) } componentDidMount() { // 组件挂载后触发

} componentWillReceivePorps(nextProps) { // 接收到新的props时触发

} shouldComponentUpdate(nextProps, nextState) { // 组件Props或者state改变时触发，true：更新，false：不更新 //常用与性能优化，较少不必要的render return true } componentWillUpdate(nextProps, nextState) { // 组件更新前触发

} componentDidUpdate() { // 组件更新后触发

} componentWillUnmount() { // 组件卸载时触发

} } ```
```

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

export default class NewReactComponent extends Component { constructor(props) { super(props) // getDefaultProps：接收初始props // getInitialState：初始化state } state = {

```
}
static getDerivedStateFromProps(props, state) {
// 组件每次被rerender的时候，包括在组件构建之后(虚拟dom之后，实际dom挂载之前)，每次获取新的props或state之后；;每次接收   新的props之后都会返回
    return state
}
componentDidCatch(error, info) { // 获取到javascript错误

}
render() {
    return (
        <h2>New React.Component</h2>
)
}
componentDidMount() { // 挂载后

}
shouldComponentUpdate(nextProps, nextState) { // 组件Props或者state改变时触发，true：更新，false：不更新
    return true
}
getSnapshotBeforeUpdate(prevProps, prevState) { // 组件更新前触发

}
componentDidUpdate() { // 组件更新后触发

}
```
```

```
componentWillUnmount() { // 组件卸载时触发

}
```

} ```

新旧总结：

1. React16新的生命周期弃用了`componentWillMount`、`componentWillReceivePorps`，`componentWillUpdate`；
2. 新增了`getDerivedStateFromProps`、`getSnapshotBeforeUpdate`来代替弃用的三个钩子函数（`componentWillMount`、`componentWillReceivePorps`，`componentWillUpdate`）；
3. React16并没有删除这三个钩子函数，但是不能和新增的钩子函数（`getDerivedStateFromProps`、`getSnapshotBeforeUpdate`）混用，`React17`将会删除`componentWillMount`、`componentWillReceivePorps`，`componentWillUpdate`；
4. 新增了对错误的处理（componentDidCatch）

对比Vue

都是组件化的解决方案：

- 数据渲染上的区别：

  `Vue`采用的是响应式数据渲染，你修改了响应式数据对应的视图也进行渲染，相当于更新通知时`Vue`帮你做了，开发者只需要关注数据的修改。

  `React`则是`setState`手动更新数据，通过`setState`更新`state`的值来达到重新`render`视图的效果，开发者只需要关注什么时候进行`state`的更改和重新`render`。

- API上的区别：

  `Vue`使用`template`的解决方案，React使用`JSX`的解决方案，各有优劣，`Vue`使用`template`在处理上优于`React`，`React`使用`JSX`的可开拓性优于`Vue`；

  `Vue`与`React`在生命周期（钩子）上的使用不同：

  ```react import React from "react"; class ReactPage extends React.Component { constructor(props) { super(props); this.state = { flag: true, msg: '展示数据' } } render() { const { flag,msg } = this.props; return ({ flag ?

  {msg}
  :
  test
  }

  );

  } } ```

  ```vue

  {{msg}}

  test

  ```

React开发需要了解的知识：

单向数据流

React组件之间的数据通过Props属性传递的，如果在`子组件试图去改变Props`的值

```
handelClick=()=>{
this.props.list = []; }
```

会报以下的错误

```
TypeError: Cannot assign to read only property 'list' of object '#<Object>'
```

意思是`list`是只读的，props属性是单向的数据流，要想改变数据，就通过改变父组件`state`的值

JSX代码注释

`jsx <div> {/* JSX代码多行注释，建议使用这种写法 */} {...} <div>`

`jsx <div> { //JSX代码单行注释 } {...} <div>`

JSX的样式

JSX直接加`class`在浏览器控制台会报警告，因为`class`是JavaScript的保留关键字，防止冲突，React的解决方案是使用`className`代替。

`jsx {/* bad */} <input class="input" value={this.state.inputValue} onChange={this.inputChange.bind(this)} />`

`jsx {/* good */} <input className="input" value={this.state.inputValue} onChange={this.inputChange.bind(this)} />`

如何获取setState后的值？

`jsx handelChange=()=>{ this.setState({ testValue: '111111' }) } console.log( this.state.testValue )`

在实际的开发中，会碰到这样的问题，我想要拿到setState后数据进行操作，但是我setState后`console.log`发现打印出来的值还是原来的。

这其中的原因便是，React中的`setState`是一个`异步`函数，涉及到虚拟DOM的渲染，可以简单的理解为setState是异步的，还没有等虚拟DOM渲染完，就执行了`console.log`，因为打印的值还是原来的。我们可以通过`setState`的回调函数，来对改变后值进行操作；

`jsx handelChange=()=>{ this.setState({ testValue: '111111' }, ()=>{ console.log( this.state.testValue ) }) }`

`补充：setState的两种写法： (1). setState(stateChange, [callback])------对象式的setState 1.stateChange为状态改变对象(该对象可以体现出状态的更改) 2.callback是可选的回调函数，它在状态更新完毕、界面也更新后(render调用后)才被调用 this.setState({ count: 1 },()=>{console.log(this.state.count)}); (2). setState(updater, [callback])------函数式的setState 1.updater为返回stateChange对象的函数。 2.updater可以接收到state和props。 4.callback是可选的回调函数，它在状态更新、界面也更新后(render调用后)才被调用。 this.setState((state,props)=>{ count: state.count+1 },()=>{console.log(this.state.count)}) 小总结：1.对象式的setState是函数式的setState的简写方式(语法糖) 2.使用原则： (1).如果新状态不依赖于原状态 ===> 使用对象方式 (2).如果新状态依赖于原状态 ===> 使用函数方式 (3).如果需要在setState()执行后获取最新的状态数据，要在第二个callback函数中读取`

为什么要加key?

如果每一个组件是一个数组或迭代器的话，那么必须有一个唯一的key prop。它是用来标识当前项的唯一性的。如果使用index，它相当于一个随机键，无论有没有相同的项，更新都会渲染。如果key相同，react会抛警告，且只会渲染第一个key。

```jsx ZZ

```
<ul ref={(ul) => {this.ul=ul}}>
    {
        this.state.list.map((item,index)=>{
            return (
                <li
                    key={item.id}
                    className='ditem'
                    // dangerouslySetInnerHTML={{__html:item}}
                    >
                    {item}

)
        })
    }

```

```

#### findDOMNode与Refs的用法

1. findDomNode，当组件加载到页面上后，可以通过`react-dom`提供的`findDomNode()`来拿到组件对应的（真实）DOM元素。

   `jsx import { findDOMNode } from 'react-dom'; ... //组件内 componentDidMoud() { const el = finDOMNode(this); }`

2. Refs是React的一种特殊的属性，它能够绑定组件的实例。

   ref如果绑定在原生HTML元素上，它拿到的就是DOM元素，如果绑定在自定义组件上面，它拿到的就是组件的实例；

   编码：

   `{/* 1.字符串形式的ref */} <input ref="input1"/>`
```

{/* 2.回调的ref 推荐使用*/} <input ref={(c)=>{this.input1 = c}}/>

{/* 3.createRef创建ref容器 */} myRef = React.createRef() <input ref={this.myRef}/>

class组件使用Form.create()如何获取组件实例的引用？

通过 wrappedComponentRef绑定可获取Form.create的表单组件

<QxlbForm wrappedComponentRef={(form) => this.nhQxlbForm = form}/>

``` import React from "react"; class QxlbInitForm extends React.Component { constructor(props) { super(props); this.state = {} } render() { const { form } = this.props; return (

子页面
); } }

const QxlbForm = Form.create()(QxlbInitForm); export default QxlbForm; ```

### 组件之间通信

1. 父组件向子组件通信

   父组件通过props向子组件传递数据

2. 子组件向父组件通信

   使用回调函数的方式，父组件将一个函数作为 props 传递给子组件，子组件调用该回调函数，便可以向父组件通信。

   react import React from "react"; const Sub = (props) => { onhandelChange = () = > { this.props.onParentChange('通信数据'); } return( <div> <button onClick = { this.onChange() }>点击我</button> </div> ) } export default Son;

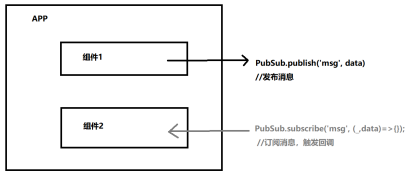
   ``` import React,{ Component } from "react"; import Son from "./SonComponent.js";

   export default class App extends Component{ onhandelChangeP = (msg) =>{ console.log(msg); } render(){ return(

   ) } } ```

3. 非嵌套组件间通信

   消息发布与订阅模式




工具包：PubSubJS

下载：cnpm install pubsub-js --save

使用：

1) import PubSub from 'pubsub-js' //引入

2) PubSub.subscribe('msg', (_,data)=>{}); //在需要接受数据的组件添加订阅

3) PubSub.publish('msg', data) //发布数据的组件添加消息发布

```react import React, { Component } from 'react'; import PropTypes from 'prop-types'; import { Input, Button , Card} from 'antd' import PubSub from 'pubsub-js'

export default class PubPage extends Component { constructor(props) { super(props); this.state = { inputVal : '' } }

```
handelSened = () => {
    const inputVal = this.state.inputVal;
    console.log(inputVal)
    // 发布
    PubSub.publish('msg',inputVal)
}
inputChange = (e) => {
    this.setState({
        inputVal: e.target.value
    })
}

render() {
    return (

            <Card title="发布者" extra={More} style={{ width: 300 }}>
                <Input value={this.state.inputVal} onChange={this.inputChange}/>

                <Button type='primary' size='small' onClick={this.handelSened}>发送</Button>
            </Card>

    );
}
```

} ```

``` import React, { Component } from 'react'; import PropTypes from 'prop-types'; import { Input, Button , Card} from 'antd' import PubSub from 'pubsub-js'

export default class subPage extends Component { constructor(props) { super(props); this.state = { list: [] } }

```
componentDidMount() {
    // 订阅
    PubSub.subscribe('msg',(_,data)=>{
        const {list} = this.state;
        this.setState({
            list: [ data,...list ]
        })
    })
}

render() {
    return (
        <div>
            <Card title="订阅者" extra={<a href="#">More</a>} style={{ width: 300 }}>
                {this.state.list.map(item=>{
                    return(
                        <p>{item}</p>
                    )
                })}

            </Card>
        </div>
    );
}
```

} ```

其他：

3.集中式管理： redux、dva等等 4.conText: 生产者-消费者模式