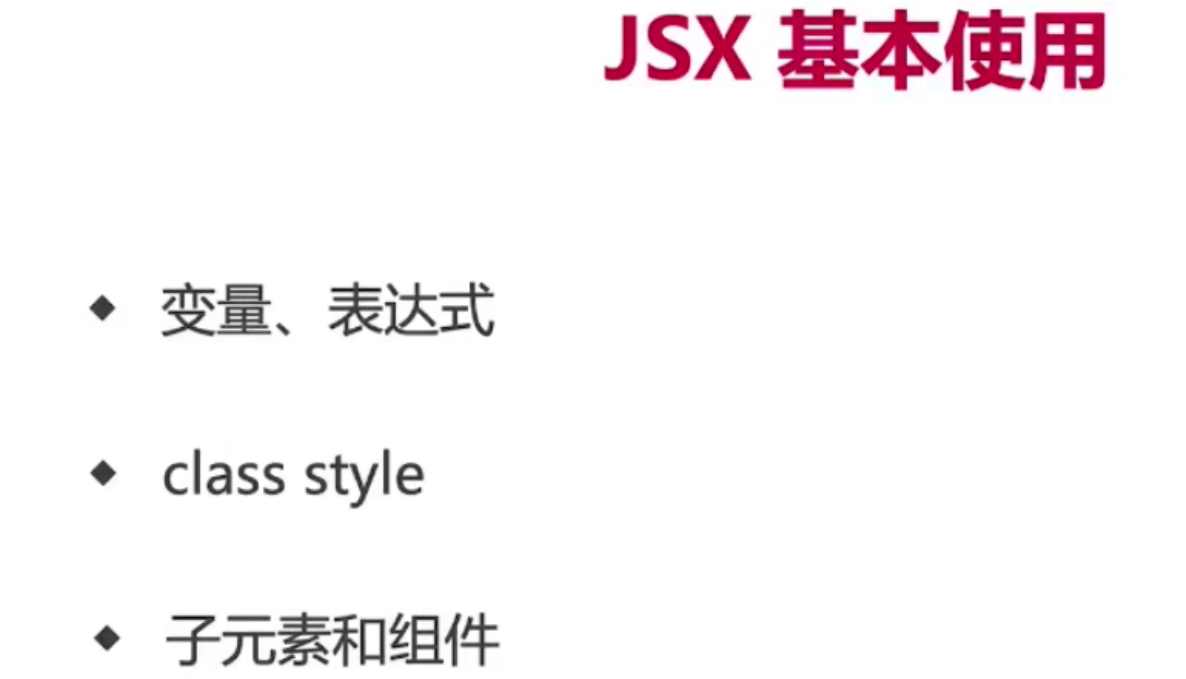
react基本使用

JSX




```
import React from 'react'
import './style.css'
import List from '../List'

class JSXBaseDemo extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            name: '双越',
            imgUrl: 'https://img1.mukewang.com/5a9fc8070001a82402060220-140-
140.jpg',
            flag: true
        }
    }
    render() {
        // // 获取变量 插值
        // const pElem = <p>{this.state.name}</p>
        // return pElem

        // // 表达式
        // const exprElem = <p>{this.state.flag ? 'yes' : 'no'}</p>
        // return exprElem

        // // 子元素
        // const imgElem = <div>
        //     <p>我的头像</p>
        //     <img src="xxxx.png"/>
        //     <img src={this.state.imgUrl}/>
```

```
        // </div>
        // return imgElem

        // // class
        // const classElem = <p className="title">设置 css class</p>
        // return classElem

        // // style
        // const styleData = { fontSize: '30px',  color: 'blue' }
        // const styleElem = <p style={styleData}>设置 style</p>
        // // 内联写法，注意 {{ 和 }}
        // // const styleElem = <p style={{ fontSize: '30px',  color: 'blue' }}>
设置 style</p>
        // return styleElem

        // 原生 html
        const rawHtml = '<span>富文本内容<i>斜体</i><b>加粗</b></span>'
        const rawHtmlData = {
            __html: rawHtml // 注意，必须是这种格式
        }
        const rawHtmlElem = <div>
            <p dangerouslySetInnerHTML={rawHtmlData}></p>
            <p>{rawHtml}</p>
        </div>
        return rawHtmlElem

        // // 加载组件
        // const componentElem = <div>
        //     <p>JSX 中加载一个组件</p>
        //     <hr/>
        //     <List/>
        // </div>
        // return componentElem
    }
}

export default JSXBaseDemo
```

条件判断

条件判断

- if else
- 三元表达式
- 逻辑运算符 && ||

```
import React from 'react'
import './style.css'

class ConditionDemo extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            theme: 'black'
        }
    }
    render() {
        const blackBtn = <button className="btn-black">black btn</button>
        const whiteBtn = <button className="btn-white">white btn</button>

        // // if else
        // if (this.state.theme === 'black') {
        //     return blackBtn
        // } else {
        //     return whiteBtn
        // }

        // // 三元运算符
        // return <div>
        //     { this.state.theme === 'black' ? blackBtn : whiteBtn }
        // </div>

        // &&
        return <div>
            { this.state.theme === 'black' && blackBtn }
        </div>
    }
}

export default ConditionDemo
```

渲染列表

渲染列表

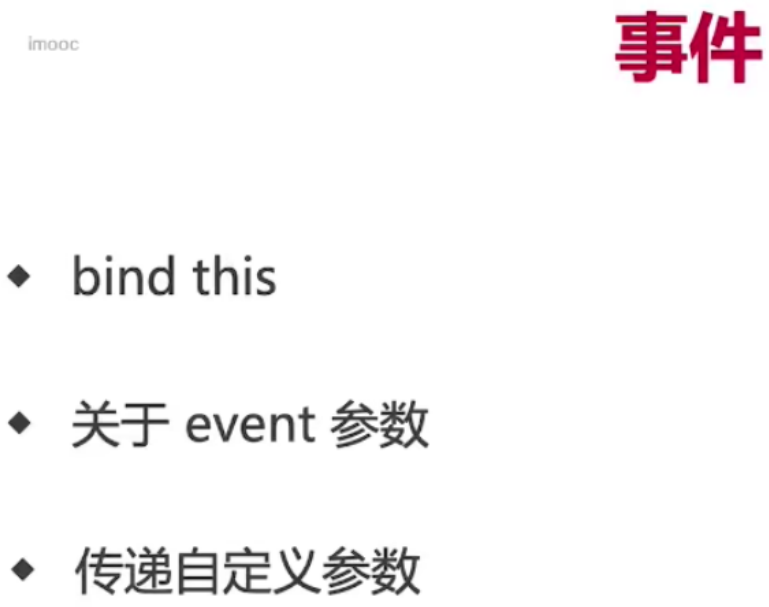
- map
- key

```
import React from 'react'

class ListDemo extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            list: [
                {
                    id: 'id-1',
                    title: '标题1'
                },
                {
                    id: 'id-2',
                    title: '标题2'
                },
                {
                    id: 'id-3',
                    title: '标题3'
                }
            ]
        }
    }
    render() {
        return <ul>
            { /* vue v-for */
                this.state.list.map(
                    (item, index) => {
                        // 这里的 key 和 Vue 的 key 类似，必填，不能是 index 或
random
                        return <li key={item.id}>
                            index {index}; id {item.id}; title {item.title}
                        </li>
                    }
                )
            }
        </ul>
    }
}

export default ListDemo
```

事件




```
import React from 'react'

class EventDemo extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            name: 'zhangsan',
            list: [
                {
                    id: 'id-1',
                    title: '标题1'
                },
                {
                    id: 'id-2',
                    title: '标题2'
                },
                {
                    id: 'id-3',
                    title: '标题3'
                }
            ]
        }

        // 修改方法的 this 指向
        this.clickHandler1 = this.clickHandler1.bind(this)
    }
    render() {
        // // this - 使用 bind
        // return <p onClick={this.clickHandler1}>
        //     {this.state.name}
        // </p>

        // // this - 使用静态方法
        // return <p onClick={this.clickHandler2}>
```

```
        //     clickHandler2 {this.state.name}
        // </p>

        // // event
        // return <a href="https://imooc.com/" onClick={this.clickHandler3}>
        //     click me
        // </a>

        // 传递参数 - 用 bind(this, a, b)
        return <ul>{this.state.list.map((item, index) => {
            return <li key={item.id} onClick={this.clickHandler4.bind(this,
item.id, item.title)}>
                index {index}; title {item.title}
            </li>
        })}</ul>
    }
    clickHandler1() {
        // console.log('this....', this) // this 默认是 undefined
        this.setState({
            name: 'lisi'
        })
    }
    // 静态方法，this 指向当前实例
    clickHandler2 = () => {
        this.setState({
            name: 'lisi'
        })
    }
    // 获取 event
    clickHandler3 = (event) => {
        event.preventDefault() // 阻止默认行为
        event.stopPropagation() // 阻止冒泡
        console.log('target', event.target) // 指向当前元素，即当前元素触发
        console.log('current target', event.currentTarget) // 指向当前元素，假
象！！！

        // 注意，event 其实是 React 封装的。可以看 __proto__.constructor 是
SyntheticEvent 组合事件
        console.log('event', event) // 不是原生的 Event ，原生的 MouseEvent
        console.log('event.__proto__.constructor', event.__proto__.constructor)

        // 原生 event 如下。其 __proto__.constructor 是 MouseEvent
        console.log('nativeEvent', event.nativeEvent)
        console.log('nativeEvent target', event.nativeEvent.target)  // 指向当前元
素，即当前元素触发
        console.log('nativeEvent current target',
event.nativeEvent.currentTarget) // 指向 document ！！！

        // 1. event 是 SyntheticEvent ，模拟出来 DOM 事件所有能力
        // 2. event.nativeEvent 是原生事件对象
        // 3. 所有的事件，都被挂载到 document 上
        // 4. 和 DOM 事件不一样，和 Vue 事件也不一样
    }
    // 传递参数
    clickHandler4(id, title, event) {
        console.log(id, title)
        console.log('event', event) // 最后追加一个参数，即可接收 event
    }
```

```
}

export default EventDemo
```

表单

```
import React from 'react'

class FormDemo extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            name: '双越',
            info: '个人信息',
            city: 'beijing',
            flag: true,
            gender: 'male'
        }
    }
    render() {

        // // 受控组件（非受控组件，后面再讲）
        // return <div>
        //     <p>{this.state.name}</p>
        //     <label htmlFor="inputName">姓名：</label> {/* 用 htmlFor 代替 for
*/}
        //     <input id="inputName" value={this.state.name} onChange=
{this.onInputChange}/>
        // </div>

        // textarea - 使用 value
        return <div>
            <textarea value={this.state.info} onChange={this.onTextareaChange}/>
            <p>{this.state.info}</p>
        </div>

        // // select - 使用 value
        // return <div>
        //     <select value={this.state.city} onChange={this.onSelectChange}>
        //         <option value="beijing">北京</option>
        //         <option value="shanghai">上海</option>
        //         <option value="shenzhen">深圳</option>
        //     </select>
        //     <p>{this.state.city}</p>
        // </div>

        // // checkbox
        // return <div>
        //     <input type="checkbox" checked={this.state.flag} onChange=
{this.onCheckboxChange}/>
        //     <p>{this.state.flag.toString()}</p>
        // </div>

        // // radio
        // return <div>
```

```
        //     male <input type="radio" name="gender" value="male" checked=
{this.state.gender === 'male'} onChange={this.onRadioChange}/>
        //     female <input type="radio" name="gender" value="female" checked=
{this.state.gender === 'female'} onChange={this.onRadioChange}/>
        //     <p>{this.state.gender}</p>
        // </div>

        // 非受控组件 - 后面再讲
    }
    onInputChange = (e) => {
        this.setState({
            name: e.target.value
        })
    }
    onTextareaChange = (e) => {
        this.setState({
            info: e.target.value
        })
    }
    onSelectChange = (e) => {
        this.setState({
            city: e.target.value
        })
    }
    onCheckboxChange = () => {
        this.setState({
            flag: !this.state.flag
        })
    }
    onRadioChange = (e) => {
        this.setState({
            gender: e.target.value
        })
    }
}

export default FormDemo
```

组件使用

组件使用

- props 传递数据
- props 传递函数
- props 类型检查

```
/**
 * @description 演示 props 和事件
 * @author 双越老师
 */

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

class Input extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            title: ''
        }
    }
    render() {
        return <div>
            <input value={this.state.title} onChange={this.onTitleChange}/>
            <button onClick={this.onSubmit}>提交</button>
        </div>
    }
    onTitleChange = (e) => {
        this.setState({
            title: e.target.value
        })
    }
    onSubmit = () => {
        const { submitTitle } = this.props
        submitTitle(this.state.title) // 'abc'

        this.setState({
            title: ''
        })
    }
}
// props 类型检查
```

```
Input.propTypes = {
    submitTitle: PropTypes.func.isRequired
}

class List extends React.Component {
    constructor(props) {
        super(props)
    }
    render() {
        const { list } = this.props

        return <ul>{list.map((item, index) => {
            return <li key={item.id}>
                <span>{item.title}</span>
            </li>
        })}</ul>
    }
}
// props 类型检查
List.propTypes = {
    list: PropTypes.arrayOf(PropTypes.object).isRequired
}

class Footer extends React.Component {
    constructor(props) {
        super(props)
    }
    render() {
        return <p>
            {this.props.text}
            {this.props.length}
        </p>
    }
    componentDidUpdate() {
        console.log('footer did update')
    }
    shouldComponentUpdate(nextProps, nextState) {
        if (nextProps.text !== this.props.text
            || nextProps.length !== this.props.length) {
            return true // 可以渲染
        }
        return false // 不重复渲染
    }

    // React 默认：父组件有更新，子组件则无条件也更新！！！
    // 性能优化对于 React 更加重要!
    // SCU 一定要每次都用吗？—— 需要的时候才优化
}

class TodoListDemo extends React.Component {
    constructor(props) {
        super(props)
        // 状态（数据）提升
        this.state = {
            list: [
                {
                    id: 'id-1',
                    title: '标题1'
```

```
                },
                {
                    id: 'id-2',
                    title: '标题2'
                },
                {
                    id: 'id-3',
                    title: '标题3'
                }
            ],
            footerInfo: '底部文字'
        }
    }
    render() {
        return <div>
            <Input submitTitle={this.onSubmitTitle}/>
            <List list={this.state.list}/>
            <Footer text={this.state.footerInfo} length=
{this.state.list.length}/>
        </div>
    }
    onSubmitTitle = (title) => {
        this.setState({
            list: this.state.list.concat({
                id: `id-${Date.now()}`,
                title
            })
        })
    }
}

export default TodoListDemo
```

☆☆☆setState

- 不可变值
- 可能是异步更新
- 可能会被合并

不可变值

可以直接使用state上的值，不能直接更改state上的值，必须通过setState函数更改

```
// // 不可变值（函数式编程，纯函数） - 数组
// const list5Copy = this.state.list5.slice()
// list5Copy.splice(2, 0, 'a') // 中间插入/删除
// this.setState({
//     list1: this.state.list1.concat(100), // 追加
//     list2: [...this.state.list2, 100], // 追加
//     list3: this.state.list3.slice(0, 3), // 截取
//     list4: this.state.list4.filter(item => item > 100), // 筛选
//     list5: list5Copy // 其他操作
// })
// // 注意，不能直接对 this.state.list 进行 push pop splice 等，这样违反不可变值

// // 不可变值 - 对象
// this.setState({
//     obj1: Object.assign({}, this.state.obj1, {a: 100}),
//     obj2: {...this.state.obj2, a: 100}
// })
// // 注意，不能直接对 this.state.obj 进行属性设置，这样违反不可变值
```

可能是异步更新，也可能是同步更新

同步更新：

在定时器setTimeout中

在自定义Dom事件中

异步更新：

setState函数传入第二个参数（函数），在此传入的函数中可获取到更新后的state

可能被合并

setState传入对象，会被合并

setState传入函数，不会被合并

```
import React from 'react'

// 函数组件（后面会讲），默认没有 state
class StateDemo extends React.Component {
    constructor(props) {
        super(props)

        // 第一，state 要在构造函数中定义
        this.state = {
            count: 0
        }
    }
    render() {
        return <div>
            <p>{this.state.count}</p>
            <button onClick={this.increase}>累加</button>
        </div>
    }
    increase = () => {
        // // 第二，不要直接修改 state ，使用不可变值 ----------------------------
```

```
        // // this.state.count++ // 错误
        // this.setState({
        //     count: this.state.count + 1 // SCU
        // })
        // 操作数组、对象的的常用形式

        // 第三，setState 可能是异步更新（有可能是同步更新） ----------------------------

        // this.setState({
        //     count: this.state.count + 1
        // }, () => {
        //     // 联想 Vue $nextTick - DOM
        //     console.log('count by callback', this.state.count) // 回调函数中可以拿到最新的 state
        // })
        // console.log('count', this.state.count) // 异步的，拿不到最新值

        // // setTimeout 中 setState 是同步的
        // setTimeout(() => {
        //     this.setState({
        //         count: this.state.count + 1
        //     })
        //     console.log('count in setTimeout', this.state.count)
        // }, 0)

        // 自己定义的 DOM 事件，setState 是同步的。示例在 componentDidMount 中

        // 第四，state 异步更新的话，更新前会被合并 ----------------------------

        // // 传入对象，会被合并（类似 Object.assign ）。执行结果只一次 +1
        // this.setState({
        //     count: this.state.count + 1
        // })
        // this.setState({
        //     count: this.state.count + 1
        // })
        // this.setState({
        //     count: this.state.count + 1
        // })

        // 传入函数，不会被合并。执行结果是 +3
        this.setState((prevState, props) => {
            return {
                count: prevState.count + 1
            }
        })
        this.setState((prevState, props) => {
            return {
                count: prevState.count + 1
            }
        })
        this.setState((prevState, props) => {
            return {
                count: prevState.count + 1
            }
        })
    }
```

```
    // bodyClickHandler = () => {
    //     this.setState({
    //         count: this.state.count + 1
    //     })
    //     console.log('count in body event', this.state.count)
    // }
    // componentDidMount() {
    //     // 自己定义的 DOM 事件，setState 是同步的
    //     document.body.addEventListener('click', this.bodyClickHandler)
    // }
    // componentWillUnmount() {
    //     // 及时销毁自定义 DOM 事件
    //     document.body.removeEventListener('click', this.bodyClickHandler)
    //     // clearTimeout
    // }
}

export default StateDemo

// ------------------------- 我是分割线 -----------------------------

// // 不可变值（函数式编程，纯函数） - 数组
// const list5Copy = this.state.list5.slice()
// list5Copy.splice(2, 0, 'a') // 中间插入/删除
// this.setState({
//     list1: this.state.list1.concat(100), // 追加
//     list2: [...this.state.list2, 100], // 追加
//     list3: this.state.list3.slice(0, 3), // 截取
//     list4: this.state.list4.filter(item => item > 100), // 筛选
//     list5: list5Copy // 其他操作
// })
// // 注意，不能直接对 this.state.list 进行 push pop splice 等，这样违反不可变值

// // 不可变值 - 对象
// this.setState({
//     obj1: Object.assign({}, this.state.obj1, {a: 100}),
//     obj2: {...this.state.obj2, a: 100}
// })
// // 注意，不能直接对 this.state.obj 进行属性设置，这样违反不可变值
```

生命周期

组件生命周期

- 单组件生命周期
- 父子组件生命周期，和 Vue 的一样