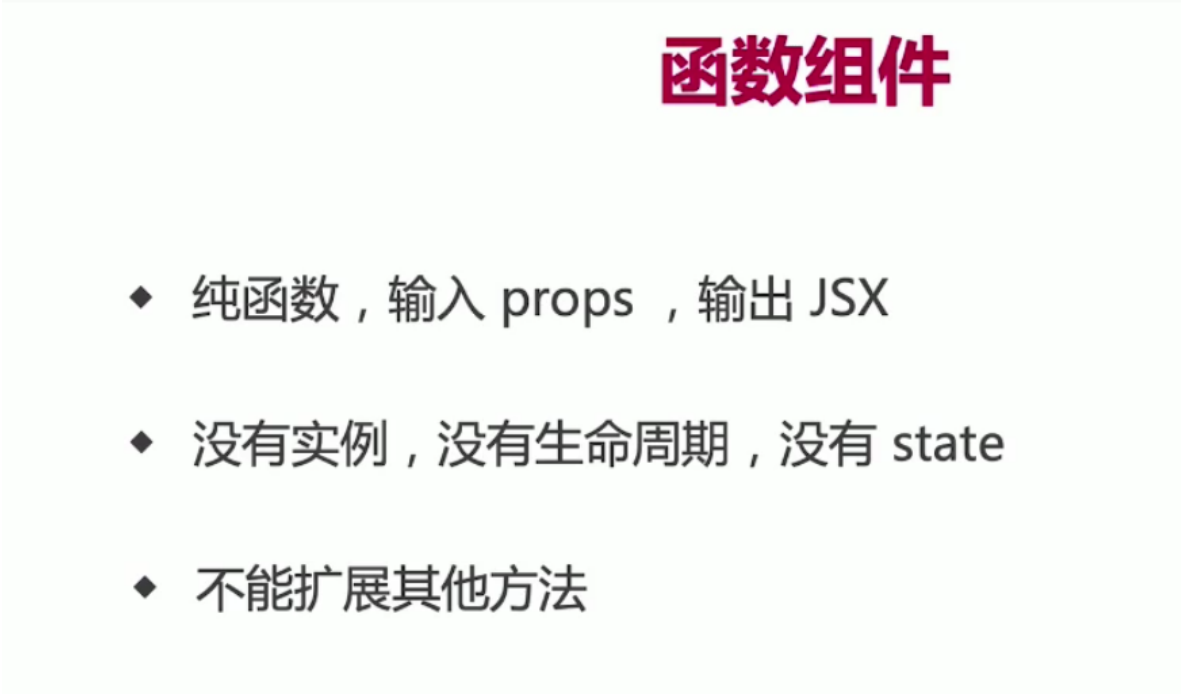
react高级使用

函数组件



非受控组件

非受控组件 – 使用场景

- 必须手动操作 DOM 元素，setState 实现不了
- 文件上传 <input type=file>
- 某些富文本编辑器，需要传入 DOM 元素

受控组件 vs 非受控组件

- 优先使用受控组件，符合 React 设计原则
- 必须操作 DOM 时，再使用非受控组件

```
import React from 'react'

class App extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            name: '双越',
            flag: true,
        }
        this.nameInputRef = React.createRef() // 创建 ref
        this.fileInputRef = React.createRef()
    }
    render() {
        // // input defaultValue
        // return <div>
        //     {/* 使用 defaultValue 而不是 value ，使用 ref */}
        //     <input defaultValue={this.state.name} ref={this.nameInputRef}/>
        //     {/* state 并不会随着改变 */}
        //     <span>state.name: {this.state.name}</span>
        //     <br/>
        //     <button onClick={this.alertName}>alert name</button>
```

```
        // </div>

        // // checkbox defaultChecked
        // return <div>
        //     <input
        //         type="checkbox"
        //         defaultChecked={this.state.flag}
        //     />
        // </div>

        // file
        return <div>
            <input type="file" ref={this.fileInputRef}/>
            <button onClick={this.alertFile}>alert file</button>
        </div>

    }
    alertName = () => {
        const elem = this.nameInputRef.current // 通过 ref 获取 DOM 节点
        alert(elem.value) // 不是 this.state.name
    }
    alertFile = () => {
        const elem = this.fileInputRef.current // 通过 ref 获取 DOM 节点
        alert(elem.files[0].name)
    }
}

export default App
```

Portals（传送门）


```
import React from 'react'
import ReactDOM from 'react-dom'
import './style.css'

class App extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
        }
    }
    render() {
        // // 正常渲染
        // return <div className="modal">
        //     {this.props.children} {/* vue slot */}
        // </div>

        // 使用 Portals 渲染到 body 上。
        // fixed 元素要放在 body 上，有更好的浏览器兼容性。
        return ReactDOM.createPortal(
            <div className="modal">{this.props.children}</div>,
            document.body // DOM 节点
        )
    }
}

export default App
```

context（上下文）

使用场景：多层组件嵌套，共享少量数据，如主题、语言

可行方式：

1. props传递：多层传递，繁杂易出错
2. redux或其他数据管理第三方工具：大材小用
3. 最佳方案：context传递

```
import React from 'react'
```

```
// 创建 Context 填入默认值（任何一个 js 变量）
const ThemeContext = React.createContext('light')

// 底层组件 - 函数是组件
function ThemeLink (props) {
    // const theme = this.context // 会报错。函数式组件没有实例，即没有 this

    // 函数式组件可以使用 Consumer
    return <ThemeContext.Consumer>
        { value => <p>link's theme is {value}</p> }
    </ThemeContext.Consumer>
}

// 底层组件 - class 组件
class ThemedButton extends React.Component {
    // 指定 contextType 读取当前的 theme context。
    // static contextType = ThemeContext // 也可以用 ThemedButton.contextType =
ThemeContext
    render() {
        const theme = this.context // React 会往上找到最近的 theme Provider，然后使
用它的值。
        return <div>
            <p>button's theme is {theme}</p>
        </div>
    }
}
ThemedButton.contextType = ThemeContext // 指定 contextType 读取当前的 theme
context。

// 中间的组件再也不必指明往下传递 theme 了。
function Toolbar(props) {
    return (
        <div>
            <ThemedButton />
            <ThemeLink />
        </div>
    )
}

class App extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            theme: 'light'
        }
    }
    render() {
        return <ThemeContext.Provider value={this.state.theme}>
            <Toolbar />
            <hr/>
            <button onClick={this.changeTheme}>change theme</button>
        </ThemeContext.Provider>
    }
    changeTheme = () => {
        this.setState({
            theme: this.state.theme === 'light' ? 'dark' : 'light'
        })
```

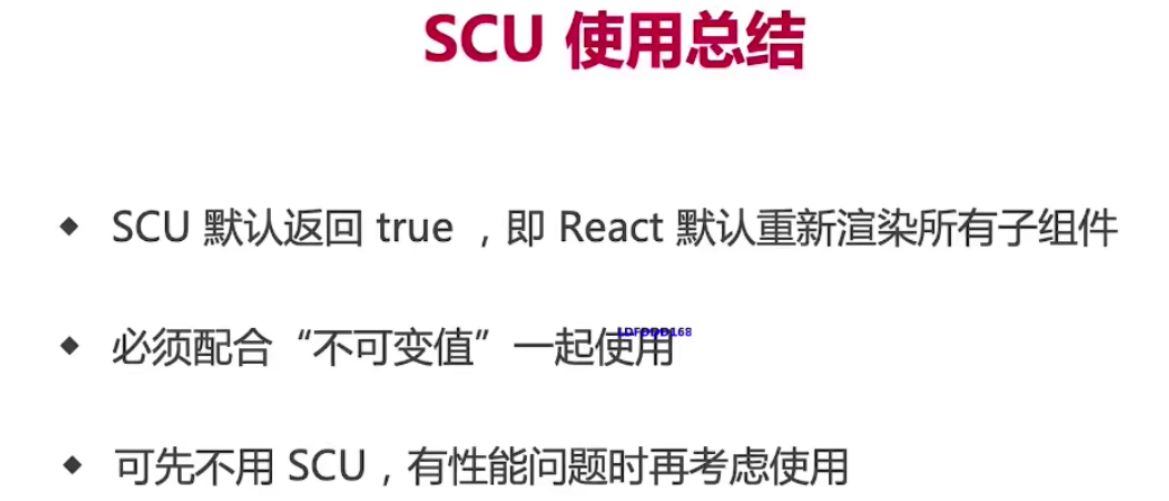
```
    }
}

export default App
```

异步组件

- import
- React.lazy
- React.Suspense

```
import React from 'react'

const ContextDemo = React.lazy(() => import('./ContextDemo'))

class App extends React.Component {
    constructor(props) {
        super(props)
    }
    render() {
        return <div>
            <p>引入一个动态组件</p>
            <hr />
            <React.Suspense fallback={<div>Loading...</div>}>
                <ContextDemo/>
            </React.Suspense>
        </div>

        // 1. 强制刷新，可看到 loading （看不到就限制一下 chrome 网速）
        // 2. 看 network 的 js 加载
    }
}

export default App
```

性能优化



PureComponent 和 memo

- PureComponent，SCU 中实现了浅比较
- memo，函数组件中的 PureComponent
- 浅比较已使用大部分情况（尽量不要做深度比较）

LDFDDD168

React.memo

```
function MyComponent(props) {
  /* 使用 props 渲染 */
}
function areEqual(prevProps, nextProps) {
  /*
  如果把 nextProps 传入 render 方法的返回结果与
  将 prevProps 传入 render 方法的返回结果一致则返回 true,
  否则返回 false
  */
}
export default React.memo(MyComponent, areEqual);
```

immutable.js

- 彻底拥抱“不可变质”
- 基于共享数据（不是深拷贝），速度好
- 有一定学习和迁移成本，按需使用

imooc

```
import React from 'react'
```

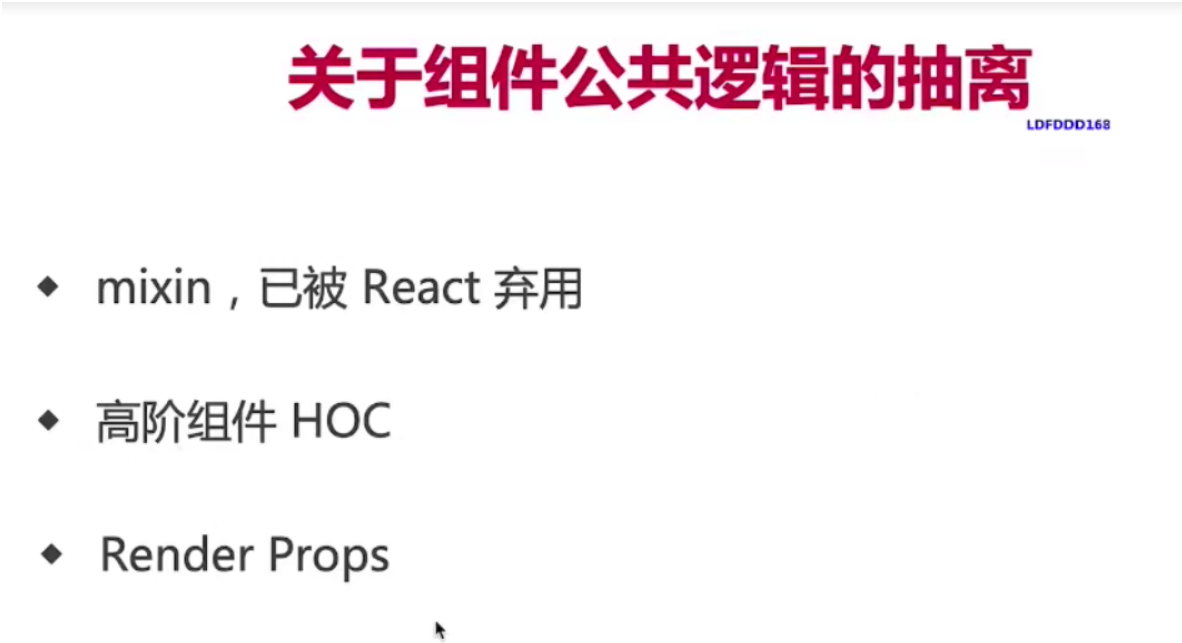
```
class App extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            count: 0
        }
    }
    render() {
        return <div>
            <span>{this.state.count}</span>
            <button onClick={this.onIncrease}>increase</button>
        </div>
    }
    onIncrease = () => {
        this.setState({
            count: this.state.count + 1
        })
    }
    // 演示 shouldComponentUpdate 的基本使用
    shouldComponentUpdate(nextProps, nextState) {
        if (nextState.count !== this.state.count) {
            return true // 可以渲染
        }
        return false // 不重复渲染
    }
}

export default App
```

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

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)

        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>
    }

    // 增加 shouldComponentUpdate
    shouldComponentUpdate(nextProps, nextState) {
        // _.isEqual 做对象或者数组的深度比较（一次性递归到底）
        if (_.isEqual(nextProps.list, this.props.list)) {
            // 相等，则不重复渲染
            return false
        }
        return true // 不相等，则渲染
    }
}
// props 类型检查
List.propTypes = {
    list: PropTypes.arrayOf(PropTypes.object).isRequired
}

class TodoListDemo 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 <div>
            <Input submitTitle={this.onSubmitTitle}/>
            <List list={this.state.list}/>
        </div>
    }
    onSubmitTitle = (title) => {
        // 正确的用法
        this.setState({
            list: this.state.list.concat({
                id: `id-${Date.now()}`,
                title
            })
        })

        // // 为了演示 SCU ，故意写的错误用法
        // this.state.list.push({
        //     id: `id-${Date.now()}`,
        //     title
        // })
        // this.setState({
        //     list: this.state.list
        // })
    }
}

export default TodoListDemo
```

高阶组件（HOC）



高阶组件 基本用法

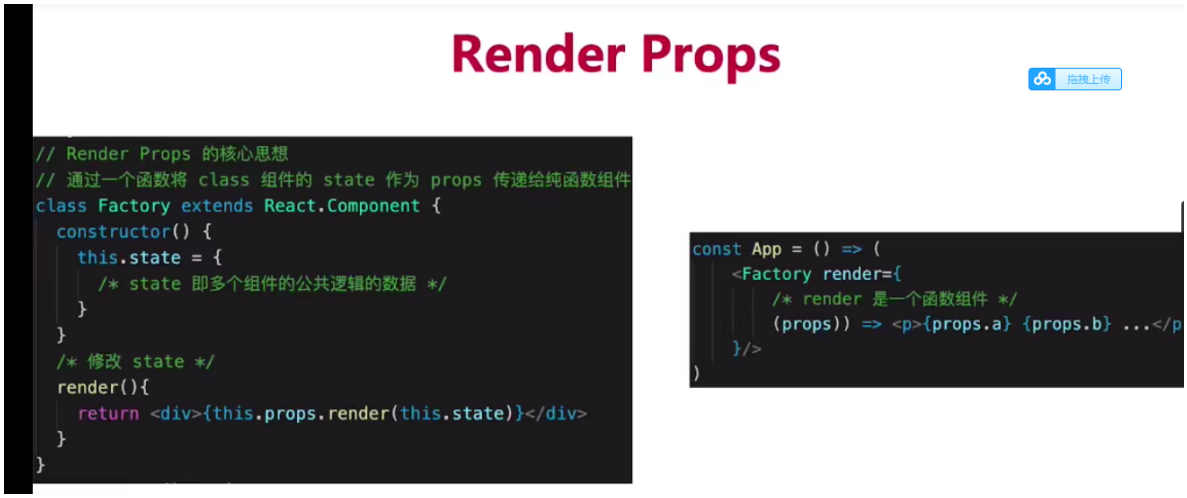
imooc

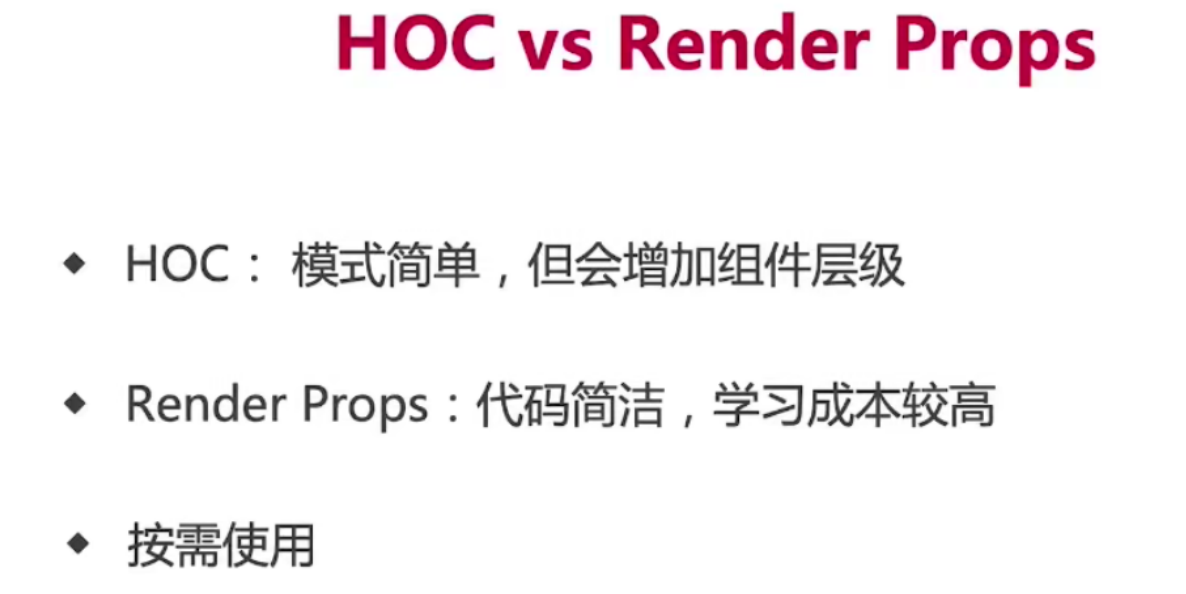
```
// 高阶组件不是一种功能，而是一种模式
const HOCFactory = (Component) => {
  class HOC extends React.Component {
    // 在此定义多个组件的公共逻辑
    render(){
      return <Component {...this.props} /> // 返回拼装的结果
    }
  }
  return HOC
}
const EnhancedComponent1 = HOCFactory(WrappedComponent1)
const EnhancedComponent2 = HOCFactory(WrappedComponent2)
```

```
import React from 'react'

// 高阶组件
const withMouse = (Component) => {
    class withMouseComponent extends React.Component {
        constructor(props) {
            super(props)
            this.state = { x: 0, y: 0 }
        }

        handleMouseMove = (event) => {
            this.setState({
                x: event.clientX,
                y: event.clientY
            })
        }

        render() {
            return (
                <div style={{ height: '500px' }} onMouseMove=
{this.handleMouseMove}>
                    {/* 1. 透传所有 props 2. 增加 mouse 属性 */}
                    <Component {...this.props} mouse={this.state}/>
                </div>
            )
        }
    }
    return withMouseComponent
}

const App = (props) => {
    const a = props.a
    const { x, y } = props.mouse // 接收 mouse 属性
    return (
        <div style={{ height: '500px' }}>
            <h1>The mouse position is ({x}, {y})</h1>
            <p>{a}</p>
```

```
        </div>
    )
}

export default withMouse(App) // 返回高阶函数
```

Render Props






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

class Mouse extends React.Component {
    constructor(props) {
        super(props)
        this.state = { x: 0, y: 0 }
    }

    handleMouseMove = (event) => {
      this.setState({
        x: event.clientX,
        y: event.clientY
      })
```

```
    }

    render() {
      return (
        <div style={{ height: '500px' }} onMouseMove={this.handleMouseMove}>
            {/* 将当前 state 作为 props ，传递给 render （render 是一个函数组件） */}
            {this.props.render(this.state)}
        </div>
      )
    }
}
Mouse.propTypes = {
    render: PropTypes.func.isRequired // 必须接收一个 render 属性，而且是函数
}

const App = (props) => (
    <div style={{ height: '500px' }}>
        <p>{props.a}</p>
        <Mouse render={
            /* render 是一个函数组件 */
            ({ x, y }) => <h1>The mouse position is ({x}, {y})</h1>
        }/>

    </div>
)

/**
 * 即，定义了 Mouse 组件，只有获取 x y 的能力。
 * 至于 Mouse 组件如何渲染，App 说了算，通过 render prop 的方式告诉 Mouse 。
 */

export default App
```