笔记五十一：react知识点整理

创建项目

创建项目指令

```javascript
npx create-react-app react-app
```

运行项目

```javascript
cd my-app
npm start
```

基础语法

JSX表达式

1、{}中间可以使用表达式 2、{}中间表达式中可以使用JSX对象 3、属性和html内容一样都是用{}来插入内容

```javascript
import React from "react";

class Child1 extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            title:"child1标题",
            isHot:false
        }
    }
    render() {
        const {title,isHot}=this.state
```

```
        let element = (
            <button>凉爽</button>
        )
        return (
            <div>
                {title}
                {isHot ? <button>炎热</button> : element}
            </div>
        )
    }
}

export default Child1
```

react 绑定class

1. 直接动态绑定，没有判断条件的（有判断条件这样写的话iconfont 不会被渲染出来 ）

javascript 复制代码

```
<i className={["iconfont"+" "+item.icon]} ></i>
```

2.有判断条件的

javascript 复制代码

```
<i className={["iconfont ",isRed ?item.icon :'' ].join('')} ></i>
```

3.使用ES6 模板字符串

javascript 复制代码

```
<i className={`iconfont ${isRed ?item.icon :'' }`} ></i>
```

react 绑定style

javascript 复制代码

```
  <div style={{display: this.state.show ? "block" : "none", color:"red"}}>此标签是否隐藏</div>
```

react 条件渲染

方案一：

javascript 复制代码

```
import React from "react";
import "./index.scss"

class Child1 extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            isHot:true,
        }
    }
    render() {
        const {isHot}=this.state
        let elementP=""
        if(isHot){
            elementP= <p>炎热</p>
        }else{
            elementP= <p>凉爽</p>
        }
        return (
            <div>
                {elementP}
            </div>
        )
    }
}

export default Child1
```

方案二：

javascript 复制代码

```
import React from "react";
import "./index.scss"

class Child1 extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            title:"child1标题",
            isHot:true,
        }
    }
    ifFun() {
        if(this.state.isHot){
            return <p>炎热1</p>
        }else{
            return <p>凉爽1</p>
        }
```

```
    }
    render() {
        const {isHot}=this.state
        return (
            <div>
                {this.ifFun()}
            </div>
        )
    }
}

export default Child1
```

方案三：

javascript 复制代码

```
import React from "react";
import "./index.scss"

class Child1 extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            isHot:true,
        }
    }
    render() {
        const {isHot}=this.state
        return (
            <div>
                {isHot ? <button>炎热</button> : <button>凉爽</button>}
            </div>
        )
    }
}

export default Child1
```

react 循环渲染

javascript 复制代码

```
import React from "react";
import "./index.scss"

class Child1 extends React.Component {
    constructor(props) {
```

```
        super(props)
        this.state = {
            list:[{
                title:"标题一"
            },{
              title:"标题二"
            }]
        }
    }
    render() {
        const {list}=this.state
        return (
            <div>
                {
                    list.map((item,index)=>{
                        return <div key={index}>{item.title}</div>
                    })
                }
            </div>
        )
    }
}

export default Child1
```

react 绑定事件

方法一：

javascript 复制代码

```
import React from "react";
import "./index.scss"

class Child1 extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
        }
    }
    clickHandle(){
        console.log("点击了")
    }
    render() {
        return (
            <div>
                <button onClick={()=>{this.clickHandle()}}>按钮</button>
            </div>
```

```
        )
    }
}

export default Child1
```

方法二：

javascript 复制代码

```
import React from "react";
import "./index.scss"

class Child1 extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
        }
    }
    clickHandle(){
        console.log("点击了")
    }
    render() {
        return (
            <div>
                <button onClick={this.clickHandle.bind(this)}>按钮</button>
            </div>
        )
    }
}

export default Child1
```

双向数据绑定

javascript 复制代码

```
import React from "react";
import "./index.scss"

class Child1 extends React.Component {
    constructor(props) {
        super(props)
        this.state = {
            value:"123"
        }
    }
    changHandle(e){
        this.setState({
```

```
            value:e.target.value
        },()=>{
            console.log(this.state.value)
        })
    }
    render() {
        const {value}=this.state
        return (
            <div>
                <input type="text" value={value} onChange={(e)=>{this.changHandle(e)}}/>
            </div>
        )
    }
}

export default Child1
```

state简写

javascript 复制代码

```
import React from "react";
import "./index.scss"

class Child1 extends React.Component {
    constructor(props) {
        super(props)
        // this.state = {
        //     title:"child1标题",
        //     isHot:true,
        //     claName:"box",
        //     show:true,
        //     list:[{
        //         title:"标题一"
        //     },{
        //       title:"标题二"
        //     }],
        //     value:"123"
        // }
    }
    state = {
        title:"child1标题",
        isHot:true,
        claName:"box",
        show:true,
        list:[{
            title:"标题一"
        },{
```

```
            title:"标题二"
        }],
        value:"123"}

    render() {
        const {title,isHot,claName,list,value}=this.state
        return (
            <div>
            </div>
        )
    }
}

export default Child1
```

父子组件传参

父组件传值给子组件 （props）

父组件：

javascript 复制代码

```
import logo from './logo.svg';
import './App.css';

//引入组件
import Child1 from "./components/child1";
import Child2 from "./components/child2";
function App() {
  return (
    <div className="App">
      <Child1 msg="这是父组件的消息"></Child1>
      <Child2></Child2>
    </div>
  );
}

export default App;
```

子组件：

```javascript 复制代码
import React from "react";
import "./index.scss"

class Child1 extends React.Component {
    state = { }
    clickHandle(){
        console.log(this.props.msg)
    }
    render() {
        const {}=this.state
        return (
            <div>
                <button onClick={()=>{this.clickHandle()}}>按钮</button>
            </div>
        )
    }
}

export default Child1
```

设置props默认值和数据类型限制

需要引入：import PropTypes from 'prop-types';

```javascript 复制代码
import React from "react";
import "./index.scss"
import PropTypes from 'prop-types';

class Child1 extends React.Component {
    state = { }
    render() {
        const {}=this.state
        return (
            <div>
                {this.props.name}
            </div>
        )
    }

}
//对标签属性进行类型、必要性的限制
Child1.propTypes = {
    name:PropTypes.string.isRequired, //限制name必传，且为字符串
    sex:PropTypes.string,//限制sex为字符串
    age:PropTypes.number,//限制age为数值
    speak:PropTypes.func,//限制speak为函数
```

```
}
//指定默认标签属性值
Child1.defaultProps = {
    sex:'男',//sex默认值为男
    age:18 //age默认值为18
}

export default Child1
```

简写方法：

javascript 复制代码

```
import React from "react";
import "./index.scss"
import PropTypes from 'prop-types';


class Child1 extends React.Component {
    state = { }
//对标签属性进行类型、必要性的限制
    static propTypes = {
        name:PropTypes.string.isRequired, //限制name必传，且为字符串
        sex:PropTypes.string,//限制sex为字符串
        age:PropTypes.number,//限制age为数值
        speak:PropTypes.func,//限制speak为函数
    }
//指定默认标签属性值
    static defaultProps = {
        sex:'男',//sex默认值为男
        age:18 //age默认值为18
    }
    clickHandle(){
        console.log(this.props.msg)
    }
    render() {
        const {}=this.state
        return (
            <div>
                <button onClick={()=>{this.clickHandle()}}>按钮</button>
                {this.props.msg}
            </div>
        )
    }
}
export default Child1
```

子组件传值给父组件，父组件调用子组件方法

方法一：使用ref 父组件

javascript 复制代码

```javascript
import React from "react";
import "./index.scss"
import Child3 from "../child3"

class Child1 extends React.Component {
    constructor(props) {
        super(props);
        this.child = React.createRef();
    }
    state = { }
    clickHandle(){
        console.log(this.child.current.state.msg)
        this.child.current.child3Fun()
    }
    render() {
        const {}=this.state
        return (
            <div>
                <Child3  ref={this.child}></Child3>
                <button onClick={()=>{this.clickHandle()}}>点我获取子组件的参数</button>
            </div>
        )
    }

}

export default Child1
```

子组件

javascript 复制代码

```javascript
import React from "react";
class Child3 extends React.Component {
    state = {msg:"这是child3中的数据" }
    child3Fun(){
        console.log("这是child3中的方法")
    }
    render() {
        const {msg}=this.state
        return (
            <div>
                {msg}
```

```
            </div>
        )
    }
}

export default Child3
```

子组件调用父组件的方法

父组件

javascript 复制代码

```
import React from "react";
import "./index.scss"
import Child3 from "../child3"

class Child1 extends React.Component {
    constructor(props) {
        super(props);
        this.child = React.createRef();
    }
    state = { }
    clickHandle(){
        console.log("这是父组件中的方法")
    }
    render() {
        const {}=this.state
        return (
            <div>
                <Child3  ref={this.child} clickHandle={this.clickHandle}></Child3>
            </div>
        )
    }
}


export default Child1
```

子组件

javascript 复制代码

```
import React from "react";

class Child3 extends React.Component {
    state = {msg:"这是child3中的数据" }
    child3Fun(){
```

```
        this.props.clickHandle()
    }
    render() {
        const {msg}=this.state
        return (
            <div>
                {msg}
                <button onClick={()=>{this.child3Fun()}}> 3按钮</button>
            </div>
        )
    }
}


export default Child3
```

生命周期

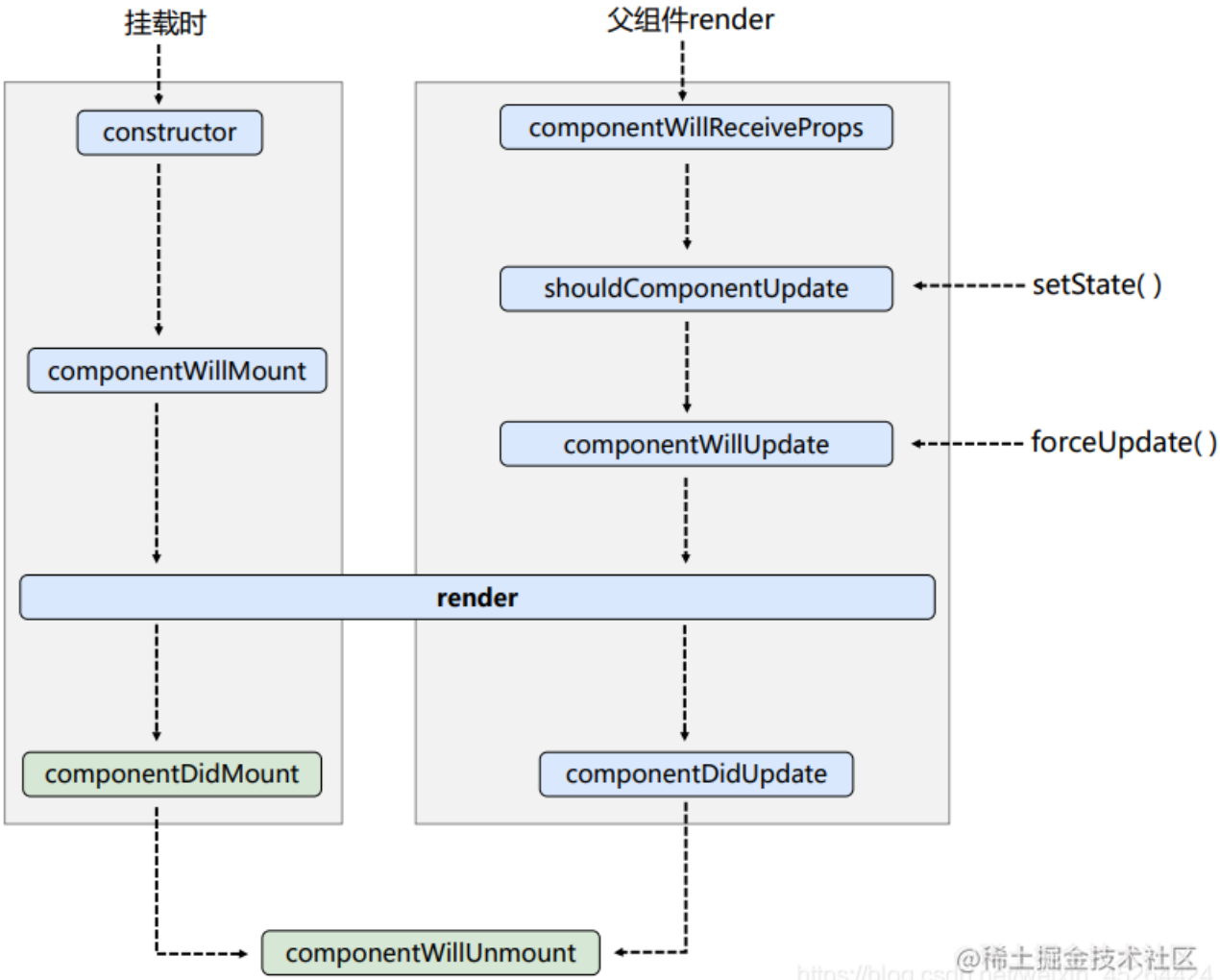
旧生命周期

javascript 复制代码

1. 初始化阶段: 由ReactDOM.render()触发---初次渲染
    1. constructor()
    2. componentWillMount()
    3. render()
    4. componentDidMount() =====> 常用一般在这个钩子中做一些初
2. 更新阶段: 由组件内部this.setSate()或父组件render触发
    1. shouldComponentUpdate()
    2. componentWillUpdate()
    3. render() =====> 必须使用的一个
    4. componentDidUpdate()
3. 卸载组件: 由ReactDOM.unmountComponentAtNode()触发
    1. componentWillUnmount() =====> 常用一般在这个钩子中做-

```
//组件将要挂载的钩子
componentWillMount(){
        console.log('Count---componentWillMount');
}

//组件挂载完毕的钩子
componentDidMount(){
        console.log('Count---componentDidMount');
}

//组件将要卸载的钩子
```

```
componentWillUnmount(){
        console.log('Count---componentWillUnmount');
}

//控制组件更新的“阀门”
shouldComponentUpdate(){
        console.log('Count---shouldComponentUpdate');
        return true
}

//组件将要更新的钩子
componentWillUpdate(){
        console.log('Count---componentWillUpdate');
}

//组件更新完毕的钩子
componentDidUpdate(){
        console.log('Count---componentDidUpdate');
}
//组件卸载完毕的钩子
componentWillUnmount(){
        console.log('Count---componentWillUnmount');
}
```



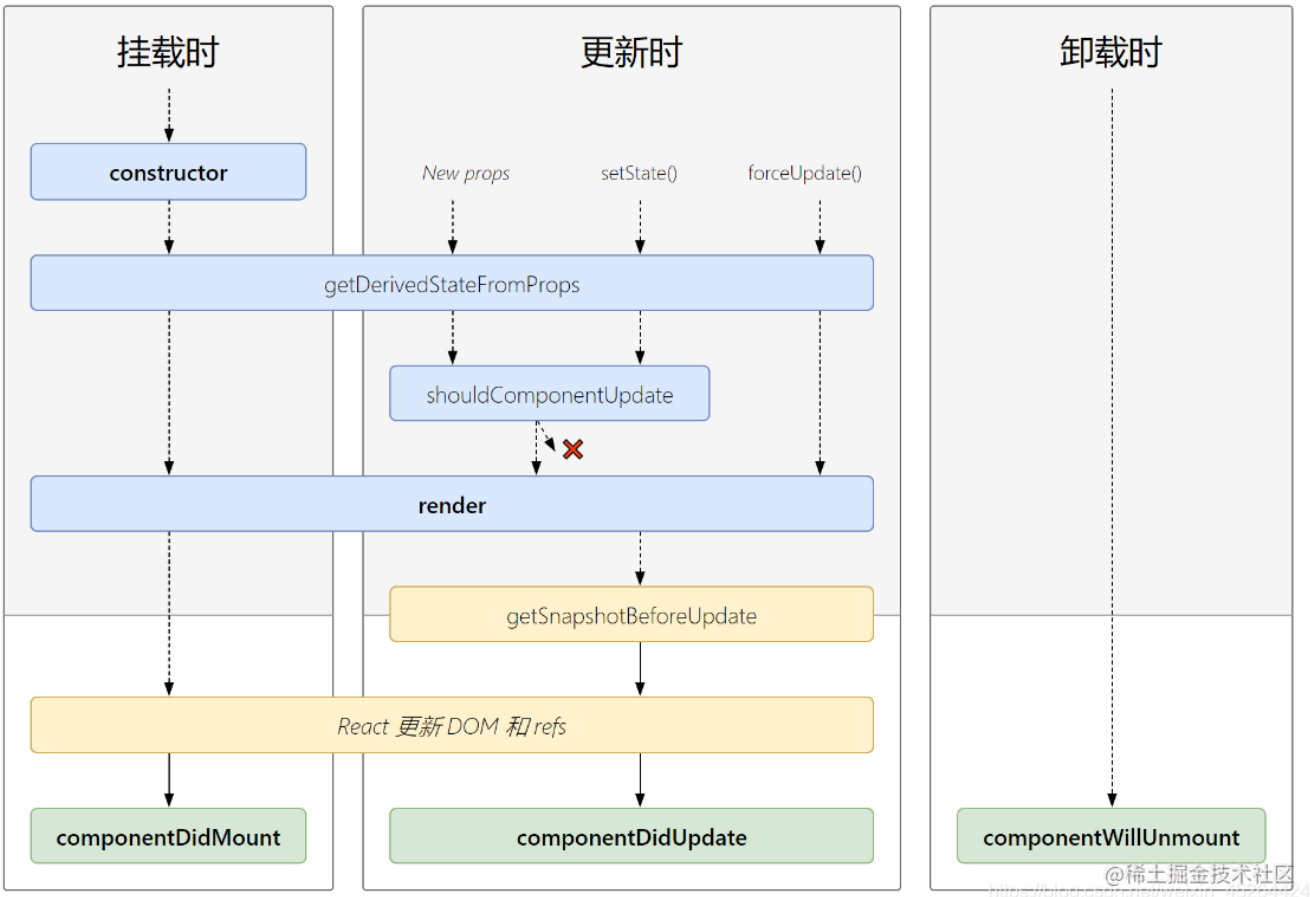
新生命周期

javascript 复制代码

1. 初始化阶段：由**ReactDOM.render()**触发---初次渲染
    1. **constructor()**
    2. getDerivedStateFromProps
    3. **render()**
    4. **componentDidMount()** =====> 常用一般在这个钩子中做一些初始化的事
2. 更新阶段：由组件内部**this.setSate()**或父组件重新render触发
    1. getDerivedStateFromProps
    2. **shouldComponentUpdate()**
    3. **render()**
    4. getSnapshotBeforeUpdate
    5. **componentDidUpdate()**
3. 卸载组件：由**ReactDOM.unmountComponentAtNode()**触发
    1. **componentWillUnmount()** =====> 常用一般在这个钩子中做一些收尾的

```
//若state的值在任何时候都取决于props，那么可以使用getDerivedStateFromProps
static getDerivedStateFromProps(props,state){
```

```
        console.log('getDerivedStateFromProps',props,state);
        return null
}

//在更新之前获取快照
getSnapshotBeforeUpdate(){
        console.log('getSnapshotBeforeUpdate');
        return 'atguigu'
}

//组件挂载完毕的钩子
componentDidMount(){
        console.log('Count---componentDidMount');
}

//组件将要卸载的钩子
componentWillUnmount(){
        console.log('Count---componentWillUnmount');
}

//控制组件更新的“阀门”
shouldComponentUpdate(){
        console.log('Count---shouldComponentUpdate');
        return true
}

//组件更新完毕的钩子
componentDidUpdate(preProps,preState,snapshotValue){
        console.log('Count---componentDidUpdate',preProps,preState,snapshotValue);
}
```



路由

安装路由插件

javascript 复制代码

```
npm install react-router-dom --save
```

ReactRouter三大组件： Router：所有路由组件的根组件（底层组件），包裹路由规则的最外层容器。 属性：basename->设置跟此路由根路径，router可以在1个组件中写多个。 Route:路由规则匹配组件，显示当前规则对应的组件 Link:路由跳转的组件

路由配置

App.js:

javascript 复制代码

```
import logo from './logo.svg';
import './App.css';
```

```
// 导入路由
import {
    BrowserRouter as Router,
    Switch,
    Route,
} from "react-router-dom";

//导入页面级组件
//登录
import Login from "./pages/Login";
//首页
import Index from "./pages/Index";
//数据管理
import System from "./pages/System";

function App() {
    return (
        <Router>
            <div> { /*路由配置*/}
                <Switch>
                    <Route path="/Login">
                        <Login/>
                    </Route>
                    <Route path="/Index">
                        <Index/>
                    </Route>
                    <Route path="/System">
                        <System/>
                    </Route>
                    <Route path="/">
                        <Login />
                    </Route>
                </Switch>
            </div>
        </Router>
    );
}

export default App;
```

子路由配置

某个页面下的js：

javascript 复制代码

```
import React from "react";
import "./index.scss"
```

```
import {Layout} from 'antd';
import {
    Switch,
    Route,
    Link,
    withRouter
} from "react-router-dom";


//引入组件
import HeaderBar from "../../components/System/HeaderBar";
import LeftNav from "../../components/System/LeftNav";
//引入页面级组件 路由
//市场主体专题
import MarketTopics from "./MarketTopics";
// 楼宇厂房载体专题
import FloorTopics from "./FloorTopics";
//产业项目专题
import IndustryTopics from "./IndustryTopics";

const {Header, Sider, Content} = Layout;
class System extends React.Component {
    constructor(props) {
        super(props)
        this.state = {}
    }

    render() {
        return (
            <Layout>
                <Header><HeaderBar></HeaderBar></Header>
                <Layout>
                    <Sider><LeftNav></LeftNav></Sider>
                    <Content>
                        {/*配置子路由*/}
                        <Switch>
                            <Route path={`/System/FloorTopics`}>
                                <FloorTopics/>
                            </Route>
                            <Route path={`/System/MarketTopics`}>
                                <MarketTopics/>
                            </Route>
                            <Route path={`/System/IndustryTopics`}>
                                <IndustryTopics/>
                            </Route>
                            {/*默认显示的子路由*/}
                            <Route path={"/System"}>
                                <MarketTopics/>
                            </Route>
```

```
                    </Switch>
                </Content>
            </Layout>
        </Layout>
    )
  }
}

export default withRouter(System)
```

路由传参(三种 推荐使用state传参)

方法一：params传参(刷新页面后参数不消失，参数会在地址栏显示)

javascript 复制代码

```
路由页面：<Route path='/demo/:id' component={Demo}></Route>  //注意要配置 /:id
路由跳转并传递参数：
    链接方式：<Link to={'/demo/'+'6'}>XX</Link>
        或：<Link to={{pathname:'/demo/'+'6'}}>XX</Link>

    js方式：this.props.history.push('/demo/'+'6')
        或：this.props.history.push({pathname:'/demo/'+'6'})
获取参数：this.props.match.params.id    //注意这里是match而非history
```

params传参(多个动态参数)

javascript 复制代码

```
state={
    id:88,
    name:'Jack',
}
路由页面：<Route path='/demo/:id/:name' component={Demo}></Route>
路由跳转并传递参数：
    链接方式：<Link to={{pathname:`/demo/${this.state.id}/${this.state.name}`}}>XX</Link>

    js方式：this.props.history.push({pathname:`/demo/${this.state.id}/${this.state.name}`})
获取参数：this.props.match.params     //结果 {id: "88", name: "Jack"}
```

方法二 query传参(刷新页面后参数消失)

javascript 复制代码

```
路由页面：<Route path='/demo' component={Demo}></Route>  //无需配置
路由跳转并传递参数：
    链接方式：<Link to={{pathname:'/demo',query:{id:22,name:'dahuang'}}}>XX</Link>
    js方式：this.props.history.push({pathname:'/demo',query:{id:22,name:'dahuang'}})
获取参数：  this.props.location.query.name
```

方法三 state传参(刷新页面后参数不消失，state传的参数是加密的，比query传参好用)

javascript 复制代码

```
路由页面：<Route path='/demo' component={Demo}></Route>  //无需配置
路由跳转并传递参数：
    链接方式：  <Link to={{pathname:'/demo',state:{id:12,name:'dahuang'}}}>XX</Link>
    js方式：this.props.history.push({pathname:'/demo',state:{id:12,name:'dahuang'}})
获取参数：  this.props.location.state.name
```

编程式导航

javascript 复制代码

```
1 push              props.history.push('/singer')

2 replace           props.history.replace({pathname:'/singer'})

3 go                props.history.go(-1)  //返回

4 goback            props.history.goBack()   //返回

5 goforward         props.history.goForward()  //前进
```

打包

javascript 复制代码

```
npm run build
```