React 进阶之路

1. JSX

1.1 JSX 语法规则

1. 定义虚拟DOM时，不写引号
2. 标签中混入**JS表达式**时用 { }
3. 样式类名指定用 className
4. 内联样式，用 style = {{ key: value }} 的形式写
5. 只有一个根标签
6. 标签必须闭合
7. 标签首字母
    - 若小写字母开头，则将标签转为 html 同名元素，若 html 中无该标签对应的同名元素，则报错
    - 若大写字母开头，react 会渲染对应的组件，若组件未定义，则报错

1.2 JSX 练习

```
    const data = ['AngularJS', 'Vue', 'React']
    const VDOM = (
      <div>
        <h1>前端框架</h1>
        <ul>
          {
            data.map((item, index) => {
              return <li key={index}>{item}</li>
            })
          }
        </ul>
      </div>
    )
    ReactDOM.render(VDOM,
document.getElementById('JSX'))
```

2. React 定义组件

2.1 函数式组件

```
// 组件名必须大写开头
    function Foo() {
      return <h2>函数式组件（适用于简单组件）</h2>
    }
    // 组件必须有结束标签
    ReactDOM.render(<Foo/>,
document.getElementById('react'))
```

2.2 类式组件

```
  // 类式组件必须继承自 React.Component
    class A extends React.Component {
      // 必须有个render函数，且有返回值，而且这个 render 函数
存在于 A 类式组件的实例对象上
      render() {
        console.log(this);  // A，原因：this指向A组件的实
例对象
        return <h2> 类式组件（适用于复杂组件）</h2 >
      }
    }
    // console.log(this); // undefined，原因：经过babel转
换后开启严格模式，this不能指向window
    ReactDOM.render(<A />,
document.getElementById("react"))
```

3. 组件实例三大属性

3.1 state

3.1.1 state 原始写法

```
// 1. 类式组件创建
    class State extends React.Component {
      // 构造器只在初始化的时候调用1次
      constructor(props) {

```

```
        super(props);
        // 初始化 state 状态
        this.state = {
          isHight: true
        }
        // 改变 change 的指向，让 change 指向 State 组件实例对象
        this.change = this.change.bind(this)
      }

      // render函数调用1+n次，第1次是初次渲染，n次是更新渲染
      render() {
        return <h2 onClick={this.change}>今天温度是{this.state.isHight ? '100度' : '20度'}</h2>
      }

      // change函数点击n次则调用n次
      change() {
        // 修改状态需要用到 setState, 更新是合并，只替换对应属性
        this.setState({ isHight: !this.state.isHight })
        // 不能直接修改状态
        // this.state.isHight = !this.state.isHight
      }
    }

    // 2. 将组件渲染到页面上
    ReactDOM.render(<State />, document.getElementById("react"))
```

3.1.2 state 简写方式

```
    // 1. 类式组件创建
    class State extends React.Component {

      // 直接使用赋值语句，代表在组件实例对象上添加属性
      // 直接初始化 state
      state = {
        isHight: true
      }

```

```
      render() {
        return <h2 onClick={this.change}>今天温度是{this.state.isHight ? '100度' : '20度'}</h2>
      }

      // 自定义方法--使用赋值语句 + 箭头函数
      change = () => {
        this.setState({ isHight: !this.state.isHight })
      }
    }

    // 2. 将组件渲染到页面上
    ReactDOM.render(<State />, document.getElementById("react"))
```

3.2 props

3.2.1 props 基本使用

```
    // 1. 创建组件
    class Props extends React.Component {
      render() {
        const { name, age, sex } = this.props
        return (
          <ul>
            <li>名字：{name}</li>
            <li>性别：{sex}</li>
            <li>年龄：{age}</li>
          </ul>
        )
      }
    }
    ReactDOM.render(<Props name='itchao' sex='男' age='22' />, document.getElementById("react1"))
```

3.2.2 props 批量传递（标签属性）

```
    // 1. 创建组件
    class Props extends React.Component {
      render() {
        const { name, age, sex } = this.props
```

```
        return (
          <ul>
            <li>名字：{name}</li>
            <li>性别：{sex}</li>
            <li>年龄：{age}</li>
          </ul>
        )
      }
    }
    const r1 = {
      name: 'itchao',
      sex: '男',
      age: '22'
    }

    // 2. 渲染组件到页面
    // 因为 react关键库 和 babel转化，所以 {...r1} 才能获取对象
    ReactDOM.render(<Props {...r1} />, document.getElementById("react1"))
```

3.2.3 props 进行限制

```
    // 1. 创建组件
    class Props extends React.Component {
      render() {
        const { name, age, sex } = this.props
        // 注意：props是只读属性，不能修改属性
        return (
          <ul>
            <li>名字：{name}</li>
            <li>性别：{sex}</li>
            <li>年龄：{age}</li>
          </ul>
        )
      }
    }
    // 对标签属性进行类型和必要性限制
    Props.propTypes = {
      name: PropTypes.string.isRequired,  // 限制类型为字符串类型 string，而且是必选值
```

```
      sex: PropTypes.string,  // 限制类型为字符串类型 string
      age: PropTypes.number,  // 限制类型为数字类型 number
      a: PropTypes.func  // 限制类型为函数类型 function
    }
    // 指定标签属性默认值
    Props.defaultProps = {
      sex: '男',
      age: 18
    }
    const r1 = {
      name: 'itchao',
      sex: '男',
      age: 22,
      a: function () {
        console.log('函数');
      }
    }

    // 2. 渲染组件到页面
    ReactDOM.render(<Props {...r1} />, document.getElementById("react1"))
```

3.2.4 props 简写方式

```
    // 1. 创建组件
    class Props extends React.Component {

      // 构造器是否接收props，是否传递给super，取决于：是否希望在构造器中通过this访问props
      // constructor(props) {
      //   super(props);
      //   console.log(this.props);
      // }

      // props简写方式：static XXX

      // 对标签属性进行类型和必要性限制
      static propTypes = {
        name: PropTypes.string.isRequired,  // 限制类型为字符串类型 string，而且是必选值
```

```
        sex: PropTypes.string,  // 限制类型为字符串类型 string
        age: PropTypes.number,  // 限制类型为数字类型 number
        a: PropTypes.func  // 限制类型为函数类型 function
      }

      // 指定标签属性默认值
      static defaultProps = {
        sex: '男',
        age: 18
      }

      render() {
        const { name, age, sex } = this.props
        // 注意：props是只读属性，不能修改属性
        return (
          <ul>
            <li>名字：{name}</li>
            <li>性别：{sex}</li>
            <li>年龄：{age}</li>
          </ul>
        )
      }
    }

    const r1 = {
      name: 'itchao',
      sex: '男',
      age: 22,
      a: function () {
        console.log('函数');
      }
    }

    // 2. 渲染组件到页面
    ReactDOM.render(<Props {...r1} />, document.getElementById("react1"))
```

3.2.5 函数式组件 props 使用

```
    function Foo(props) {
      // 函数接收 props 作为参数
      const { name, sex, age } = props;
      return (
        <ul>
          <li>名字：{name}</li>
          <li>性别：{sex}</li>
          <li>年龄：{age}</li>
        </ul>
      )
    }

    Foo.propTypes = {
      name: PropTypes.string.isRequired,
      sex: PropTypes.string,
      age: PropTypes.number
    }

    Foo.defaultProps = {
      sex: '男',
      age: 18
    }
    ReactDOM.render(<Foo name='itchao' sex='男'
age='22' />, document.getElementById('react'))
```

3.3 refs

3.3.1 字符串形式 ref

```
    class Refs extends React.Component {
      render() {
        return (
          <div>
            {/* ref 类似id，相当于打标识 */}
            <input ref='r1' type="text" placeholder="点
击获取提示信息" />
            <button onClick={this.click}>点击获取提示信息
</button>
```

```
            <input ref='r2' type="text" placeholder="失去焦点提示信息" onBlur={this.blur} />
          </div>
        )
      }

      click = () => {
        // this.refs.r1 获取ref为字符串r1的元素
        alert(this.refs.r1.value)
      }

      blur = () => {
        alert(this.refs.r2.value)
      }
    }

    ReactDOM.render(<Refs />, document.getElementById('react'))
```

3.3.2 回调形式 ref

```
    class Refs extends React.Component {
      render() {
        return (
          <div>
            {/*
            ref 类似id，相当于打标识
            回调形式 ref，回调函数传递的参数就是当前节点
            此处 this 指向当前组件实例对象，this.input1 相当于在当前组件实例对象上添加 input1 属性
            */}
            <input ref={c => this.input1 = c} type="text" placeholder="点击获取提示信息" />
            <button onClick={this.click}>点击获取提示信息</button>
            <input ref={c => this.input2 = c} type="text" placeholder="失去焦点提示信息" onBlur={this.blur} />
          </div>
        )
      }
```

```
16
17        click = () => {
18          alert(this.input1.value)
19        }
20
21        blur = () => {
22          alert(this.input2.value)
23        }
24      }
25
26      ReactDOM.render(<Refs />,
document.getElementById('react'))
```

3.3.3 回调形式 ref 调用次数问题

- 问题：若 ref 回调函数以内联函数方式定义，则更新过程中会被执行两次，第一次传入参数 null，第二次传入参数为当前 DOM 元素
- 原因：每次渲染时会创建一个新的函数实例，所以 React 会清空旧的 ref 并设置新的
- 解决方式：将 ref 回调函数定义成 class类 的绑定函数方式可避免该问题
- 注意：大多数情况下该问题无关紧要

```
1     class Refs extends React.Component {
2       render() {
3         return (
4           <div>
5             {/* 将 ref 回调函数定义成 class类 的绑定函数方式
*/}
6             <input ref={this.saveInput} placeholder="点
击获取提示信息" type="text" /><br /><br />
7             <button onClick={this.click}>点击获取提示信息
</button>
8           </div>
9         )
10      }
11
12      // 回调函数形式 ref
13      saveInput = c => this.input = c
14
15      // 点击函数
```

```
      click = () => {
        alert(this.input.value)
      }
    }

    ReactDOM.render(<Refs />,
document.getElementById('react'))
```

3.3.4 createRef

- React 官方推荐使用 ref 的方式

```
    // 1. 类式创建组件
    class Refs extends React.Component {
      render() {
        return (
          <div>
            <input ref={this.ref1} type="text"
placeholder="点击获取提示信息" />
            <button onClick={this.click}>点击获取提示信息
</button>
          </div>
        )
      }

      // React.createRef 调用后返回一个容器，该容器可以存储被
ref所标识的节点，使用一次则必须调用一次
      ref1 = React.createRef()

      // 点击获取提示信息
      // this.ref1.current，必须拿到 current 属性，才算是拿
到当前 DOM 节点
      click = () => {
        alert(this.ref1.current.value)
      }

    }

    // 2. 渲染组件到页面中
    ReactDOM.render(<Refs />,
document.getElementById('react'))
```

4. 事件处理

4.1 事件处理基本概念

1. 通过类似 onAbc 属性指定事件处理函数（注意大小写）

- React 使用自定义（合成）事件，而不使用原生 DOM 事件 —— 更好兼容性（原生 onclick -> React 中 onClick 等等）
- React 中事件通过事件委托方式处理（委托给组件最外层元素）—— 更高效

2. 通过 event.target 得到发生事件的 DOM 元素对象 —— 不要过渡使用 ref

5. React 收集表单数据

5.1 非受控组件

- 非受控组件：在表单中，所有输入控件，使用时才获取节点（现用现取）
- 缺点：多次使用 ref

```
   class Login extends React.Component {
      render() {
        return (
          <div>
            <form action="https://www.baidu.com" onSubmit={this.handleSubmit}>
              <label>用户名：</label><input ref={c => this.name = c} type="text" name='name' />
              <label>密码：</label><input ref={c => this.password = c} type="password" name='password' />
              <button>登录</button>
            </form>
          </div>
        )
      }

      // 非受控组件：在表单中，所有输入控件，使用时才获取节点（现用现取）
      // 缺点：多次使用 ref
```

```
      handleSubmit = event => {
        // 阻止表单提交
        event.preventDefault();
        alert(`你的用户名：${this.name.value}，你的密码：${this.password.value}`);
      }
    }

    ReactDOM.render(<Login />, document.getElementById('react'))
```

5.2 受控组件

- 受控组件：在表单中，所有输入控件，使用前就已经获取节点（先取后用）
- 优点：未使用 ref

```
    class Login extends React.Component {
      render() {
        return (
          <div>
            <form>
              用户名：<input onChange={this.handleUserName} type="text" name='userName' />  
              密码：<input onChange={this.handleUserPassword} type="password" name='userPassword' />
            </form>
          </div>
        )
      }

      // 受控组件：在表单中，所有输入控件，使用前就已经获取节点（先取后用）
      // 优点：未使用 ref
      state = {
        userName: '',
        userPassword: ''
      }

```

```
      handleUserName = event => {
        this.setState({ userName: event.target.value })
      }

      handleUserPassword = event => {
        this.setState({ userPassword:
event.target.value })
      }

    }

    ReactDOM.render(<Login />,
document.getElementById('react'))
```

6. 高阶函数-函数柯理化

6.1 高阶函数-函数柯理化

- 高阶函数：一个函数满足下面2个规范中的任何一个
    1. 若 A 函数，接收的参数是一个函数，则 A 函数是高阶函数
    2. 若 A 函数，调用的返回值仍是一个函数，则 A 函数是高阶函数
    3. 常见高阶函数：Promise、setTimeout、arr.map() 等等
- 函数的柯理化：函数调用的返回值是函数，且不断调用，实现多次接受参数最后统一处理的函数编码形式

```
function sum(a) {
  return b => {
    return c => {
      return a + b + c
    }
  }
}
```

- 受控组件使用高阶函数和函数柯理化

```
  class Login extends React.Component {
    render() {
      return (
        <div>
```

```
            <form>
              用户名：<input onChange={this.handleFormData('userName')} type="text" name='userName' />  
              密码：<input onChange={this.handleFormData('userPassword')} type="password" name='userPassword' />
            </form>
          </div>
        )
      }

      state = {
        userName: '',
        userPassword: ''
      }

      handleFormData = dataType => {
        return event => {
          // [dataType], 是为了把变量作为对象的对象属性
          this.setState({ [dataType]: event.target.value })
        }
      }
    }

    ReactDOM.render(<Login />, document.getElementById('react'))
```

7. React 生命周期函数（旧）

7.1 避遍 React 生命周期函数

- 名字：生命周期回调函数 - 生命周期钩子函数 - 生命周期函数 - 生命周期钩子
- 概念：React 内置好的回调函数，可以在适当的时候进行回调操作
- 分析：
    1. 组件从创建到销毁经历的一些特定阶段

2. React 组件中包含一系列钩子函数（生命周期回调函数），会在特定时刻被调用
3. 在定义组件时，在特定的生命周期回调函数中，做特定操作

- 例子如下：创建定时器和销毁定时器

```
    class Life extends React.Component {

      // 设置状态
      state = {
        opacity: 1
      }

      // 组件挂载完毕
      componentDidMount() {
        // 设置定时器
        this.timer = setInterval(() => {
          let { opacity } = this.state
          opacity -= 0.1
          if (opacity <= 0) opacity = 1
          this.setState({ opacity })
        }, 200)
      }

      // 组件将要卸载
      componentWillUnmount() {
        // 清除定时器
        clearInterval(this.timer)
      }

      // 组件初次渲染和状态更新后
      render() {
        return (
          <div>
            <h1 style={{ opacity: this.state.opacity }}>React 我一定能学会! </h1>
            <button onClick={this.click}>相信自己</button>
          </div>
        )
      }
```

```
34
35        click = () => {
36          // 卸载页面 DOM 节点
37
 ReactDOM.unmountComponentAtNode(document.getElementById('react'));
38        }
39      }
40
41      // 挂载页面 DOM 节点
42      ReactDOM.render(<Life />, document.getElementById('react'))
```

7.2 挂载阶段

挂载阶段调用构造函数如下：

1. constructor( )，构造器
2. componentWillMount( )，组件将要挂载
3. render( )，渲染组件
4. componentDidMount( )，组件完成挂载
5. componentWillUnmount( )，组件将要销毁

- 例子：

```
1     class Count extends React.Component {
2
3       // 构造器，首先调用
4       constructor(props) {
5         console.log('count---constructor');
6         super(props);
7         this.state = {
8           count: 0
9         }
10      }
11
12      // 组件将要挂载
13      componentWillMount() {
14        console.log('count---componentWillMount');
15      }
16
```

```
17        // 组件挂载完毕
18        componentDidMount() {
19          console.log('count---componentDidMount');
20        }
21
22        // 组件将要卸载
23        componentWillUnmount() {
24          console.log('count---componentWillUnmount');
25        }
26
27        // 点击事件
28        add = () => {
29          let count = this.state.count + 1
30          this.setState({ count })
31        }
32
33        // 卸载事件
34        uninstall = () => {
35
 ReactDOM.unmountComponentAtNode(document.getElementById('react'))
36        }
37
38        // 组件初次渲染和状态更新
39        render() {
40          console.log('count---render');
41          const { count } = this.state
42          return (
43            <div>
44              <h2>当前值为：{count}</h2>
45              <button onClick={this.add}>+1</button>
46              <button onClick={this.uninstall}>卸载组件</button>
47            </div>
48          )
49        }
50      }
51
52      ReactDOM.render(<Count />, document.getElementById('react'))
```

7.3 父组件 render 阶段

7.3.1 setState( )

- 前提是必须使用 setState( ) 实现状态更新后才回调如下函数

状态更新调用构造函数如下：

1. shouldComponentUpdate( )，控制组件是否更新的 ' 开关 '；需要返回一个布尔值，默认返回 true。
2. componentWillUpdate( )，组件将要更新
3. render( )，渲染组件
4. componentDidUpdate( )，组件更新完毕
5. componentWillUnmount( )，组件将要卸载

- 例子：

```
class Count extends React.Component {

  // 组件将要卸载
  componentWillUnmount() {
    console.log('count---componentWillUnmount');
  }

  // 控制组件是否更新的 '开关'
  shouldComponentUpdate() {
    console.log('count---shouldComponentUpdate');
    // 默认返回值为 true
    return true
  }

  // 组件将要更新
  componentWillUpdate() {
    console.log('count---componentWillUpdate');
  }

  // 组件更新完毕
  componentDidUpdate() {
    console.log('count---componentDidUpdate');
  }


```

```
26        // 点击事件
27        add = () => {
28          let count = this.state.count + 1
29          this.setState({ count })
30        }
31
32        // 卸载事件
33        uninstall = () => {
34
 ReactDOM.unmountComponentAtNode(document.getElementById('react'))
35        }
36
37        // 组件初次渲染和状态更新
38        render() {
39          console.log('count---render');
40          const { count } = this.state
41          return (
42            <div>
43              <h2>当前值为：{count}</h2>
44              <button onClick={this.add}>+1</button>
45              <button onClick={this.uninstall}>卸载组件</button>
46            </div>
47          )
48        }
49      }
50
51      ReactDOM.render(<Count />, document.getElementById('react'))
```

7.3.2 forceUpdate( )

强制更新调用构造函数如下：

1. componentWillUpdate( )，组件将要更新；需主动调用 forceUpdate( ) 函数，即可在不更新状态的情况下直接实现强制更新操作。
2. render( )，渲染组件
3. componentDidUpdate( )，组件更新完成
4. componentWillUnmount( )，组件将要卸载

- 例子:

```
    class Count extends React.Component {

      // 组件将要卸载
      componentWillUnmount() {
        console.log('count---componentWillUnmount');
      }

      // 组件将要更新
      componentWillUpdate() {
        console.log('count---componentWillUpdate');
      }

      // 组件更新完毕
      componentDidUpdate() {
        console.log('count---componentDidUpdate');
      }


      // 点击事件
      add = () => {
        let count = this.state.count + 1
        this.setState({ count })
      }

      // 卸载事件
      uninstall = () => {

 ReactDOM.unmountComponentAtNode(document.getElementById('react'))
      }

      // 强制更新事件
      force = () => {
        // forceUpdate(),强制更新：不改变状态也可直接实现更新操作
        this.forceUpdate()
      }

      // 组件初次渲染和状态更新
```

```
      render() {
        console.log('count---render');
        const { count } = this.state
        return (
          <div>
            <h2>当前值为：{count}</h2>
            <button onClick={this.add}>+1</button>
            <button onClick={this.uninstall}>卸载组件
</button>
            <button onClick={this.force}>强制更新
</button>
          </div>
        )
      }
    }

    ReactDOM.render(<Count />,
document.getElementById('react'))
```

7.3.3 父组件 render

父组件 render 调用构造函数如下：

1. componentWillReceiveProps( )，组件将要接收 props；可传入 props 参数，获取 state 内数据；但注意第一次接收 props 不回调该函数，后续更新传递 props 才回调该函数。
2. shouldComponentUpdate( )，控制组件是否更新的 ' 开关 '；需要返回一个布尔值，默认返回 true。
3. componentWillUpdate( )，组件将要更新
4. render( )，渲染组件
5. componentDidUpdate( )，组件更新完毕
6. componentWillUnmount( )，组件将要卸载

- 例子：

```
    class Parent extends React.Component {

      state = {
        name: '奥迪'
      }

```

```
      changeCar = () => {
        this.setState({ name: '兰博基尼' })
      }

      render() {
        return (
          <div>
            <h2>Parent 父组件</h2>
            <button onClick={this.changeCar}>买新车</button>
            <Children name={this.state.name} />
          </div>
        )
      }
    }

    class Children extends React.Component {

      componentWillReceiveProps() {
        console.log('Children---componentWillReceiveProps');
      }

      render() {
        return (
          <h2>当前汽车品牌：{this.props.name}</h2>
        )
      }
    }

    ReactDOM.render(<Parent />, document.getElementById('react'))
```

7.4 React 旧生命周期函数总结

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**
    - 常在该钩子函数中进行一些初始化操作，例如：开启定时器、发送网络请求、订阅消息 等等

2. 更新阶段：父组件 render 触发
    1. componentWillReceiveProps( )
    2. shouldComponentUpdate( ) -> setState( )
    3. componentWillUpdate( ) -> forceUpdate( )
    4. render( )，**必须调用**
    5. componentDidUpdate( )
3. 卸载组件阶段：由 ReactDOM.unmountComponentAtNode( ) 触发，该回调函数需要传入具体的 DOM 节点
    1. componentWillUnmount( )，**常用**
        - 常在该钩子函数中进行一些收尾操作，例如：关闭定时器、取消订阅消息 等等

8. React 生命周期函数（新）

8.1 React 新旧生命周期函数区别

8.1.1 即将废弃钩子函数

1. componentWillMount( )
2. componentWillReceiveProps( )
3. componentWillUpdate( )

- 注意：现在使用会出现警告，下一个大版本需要加上 UNSAFE_ 前缀才能使用，后续（React 18 ）可能被彻底废弃，不建议使用

8.1.2 新增钩子函数

1. getDerivedStateFromProps( )，从 props 中获取派生状态
    1. 必须是静态函数，static getDerivedStateFromProps( )
    2. 必须有返回值且返回值是 state 或者 null
    3. 可接受 props 参数
    4. 使用场景：若 state 在任何时候都取决于 props
    5. 缺点：派生状态会导致代码冗余，并使组件难以维护
    6. 注意：使用很少，了解即可
    - 例子：

```
      static getDerivedStateFromProps(props, state) {
        console.log('NewLife---getDerivedStateFromProps', props, state);
        return props
      }
```

2. getSnapshotBeforeUpdate( )，在更新前获取当前状态快照
    1. 必须有返回值且返回值是 snapshot(任何值都可作为快照值) 或者 null
    2. 在最近一次渲染输出（提交到 DOM 节点）之前调用
    3. 让组件在发生更改之前从 DOM 中捕获一些信息（例如：滚动位置）
    4. 此生命周期函数的任何返回值都将作为参数传递给 componentDidupdate( )
    5. 注意：使用很少，了解即可

8.2 React 新生命周期函数总结

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**
        - 常在该钩子函数中进行一些初始化操作，例如：开启定时器、发送网络请求、订阅消息 等等
2. 更新阶段：父组件 render 触发
    1. getDerivedStateFromProps( )
    2. shouldComponentUpdate( ) -> setState( )
    3. render( )，**必须调用**
    4. getSnapshotBeforeUpdate( )
    5. componentDidUpdate(preProps, preState, snapshotValue)，

        - 可接收三个参数，第一个参数为上一次的 props，第二个参数为上一次的 state，第三个参数为快照值 snapshotValue
    3. 卸载组件阶段：由 ReactDOM.unmountComponentAtNode( ) 触发，该回调函数需要传入具体的 DOM 节点
        1. componentWillUnmount( )，**常用**
            - 常在该钩子函数中进行一些收尾操作，例如：关闭定时器、取消订阅消息 等等

9. React 脚手架

9.1 邂逅 React 脚手架

1. xxx 脚手架：用来帮助程序员快速创建一个基于 xxx 库的模板项目
    1. 包含了所有需要的配置（语法检查、JSX 编译、devServer 等等）
    2. 下载好了所有相关依赖
    3. 可以直接运行处一个简单效果
2. react 提供了一个用于创建 react 项目的脚手架库：create-react-app
3. 项目整体技术架构为：react + webpack + es6 + eslint 等等
4. 使用脚手架开发项目的特点：模块化、组件化、工程化

9.2 创建项目并启动

第一步，全局安装：npm install -g create-react-app

第二步，切换到向创建项目的目录，使用命令：create-react-app hello-react（文件夹名）

第三步，切换目录，进入项目文件夹：cd hello-react（文件夹名）

第四步，启动项目：npm start

9.3 React 脚手架项目结构分析

public -- 静态资源文件夹

- favicon.icon -- 网站页签图标

- **index.html -- 主页面**
- logo192.png -- logo 图
- logo512.png -- logo 图
- manifest.json -- 应用加壳的配置文件
- robots.txt -- 爬虫协议文件

src -- 源码文件夹

- App.css -- APP 组件的样式
- **App.js -- App 组件**
- App.test.js -- 用于给 APP 做测试
- index.css -- 样式
- **index.js -- 入口文件**
- logo.svg -- logo 图
- reportWebVitals.js -- 页面性能分析文件（需要 web-vitals 库的支持）
- setupTests.js -- 组件单元测试的文件（需要 jest-dom 库的支持）

9.4 重构 React 脚手架项目结构（示例）

基本目录：

public -- 静态资源文件夹

- favicon.icon -- 网站页签图标
- **index.html -- 主页面**

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible"
content="IE=edge">
  <meta name="viewport" content="width=device-
width, initial-scale=1.0">
  <title>邂逅 React 脚手架</title>
</head>

<body>
  <div id="root"></div>
```

```
</body>

</html>
```

src -- 源码文件夹

- **components -- 存放组件的文件夹**
    - Hello -- 存放 Hello 组件的文件夹
        - index.css -- Hello 组件的样式

```
.hello {
  background-color: pink;
}
```

        - index.jsx -- Hello 组件的内容

```
// 从 react 核心库中单独导入 React.Component
import { Component } from 'react'
// 导入 index.css 文件
import './index.css'

// 创建并默认导出类 Hello 组件
export default class Hello extends Component {
  render() {
    return (
      <h2 className="hello">Hello, itchao! </h2>
    )
  }
}
```

- **App.js -- App 组件**

```
// 引入 React 核心库
import { Component } from 'react';
// 导入 Hello 组件
import Hello from './components/Hello'


```

```
// 创建并默认导出类 App 组件
export default class App extends Component {
  render() {
    return (
      <h2><Hello /></h2>
    )
  }
}
```

- **index.js -- 入口文件**

```
// 引入 React 核心库
import React from 'react';
// 引入 ReactDOM 虚拟DOM
import ReactDOM from 'react-dom';
// 引入 App 组件
import App from './App'

// 渲染组件
ReactDOM.render(<App />,
document.getElementById('root'))
```

注意：

- 组件后缀用 .jsx 或 .js 结尾都可以，推荐 .jsx，便于直接分辨该文件是组件文件
- 创建 components 文件夹，便于管理组件
- components 文件夹下再创建相关组件的文件夹（大写开头且使用驼峰命名法），便于独立管理相关组件
- 使用 index.css 和 index.jsx 取名的作用：便于书写路径，因为导入文件时会默认查找该文件夹下的 index 文件
- 可以直接使用组件名字定义组件文件，例如 Hello 组件，可用：hello.css 和 hello.jsx，但导入时书写路径不方便
- 在导入文件时， .jsx 和 .js 后缀名可省略，.css 后缀名不可省略

样式模块化：

- Hello 组件例子：
    - 目录结构：
        - Hello (文件夹)：

        - index.jsx
        - index.module.css
    - 具体代码：
        - index.jsx

```jsx
// 从 react 核心库中单独导入 React.Component
import { Component } from 'react'
// 导入 index.css 文件(关键)
import hello from './index.module.css'

// 创建并默认导出类 Hello 组件
export default class Hello extends Component {
  render() {
    return (
      // 关键：使用{}且用了导入的 hello.title 获取类名
      <h2 className={hello.title}>Hello，  itchao! </h2>
    )
  }
}
```

        - index.module.css

```css
.title {
  background-color: pink;
}
```

10. 功能界面的组件化编码流程（通用）

10.1 组件化编码流程

1. 拆分组件：拆分界面，抽取组件
2. 实现静态组件：使用组件实现静态页面效果
3. 实现动态组件

    3.1 动态显示初始化数据

11. React ajax

11.1 理解

11.1.1 前置说明

1. React 本身只关注于界面，不包含发送 ajax 请求代码
2. 前端应用需要通过 ajax 请求与后台进行交互（JSON 数据）
3. React 应用中需要集成第三方 ajax 库（或自己封装）

11.1.2 常见 ajax 请求库

1. jQuery：代码多，需要另外引入，不建议使用
2. axios：轻量级，建议使用
    - 封装 XMLHttpRequest 对象的 ajax
    - Promise 风格
    - 可以在浏览器端和 node 服务器端使用

11.2 axios

11.2.1 github 文档

- https://github.com/axios/axios

11.2.2 相关 API

1. get 请求

```
axios.get('/user?id=12345')
  .then( res => {
    console.log(res.data);
  })
  .catch( err => {
    console.log(err);
  });
```

```
8
9  axios.get('/user', {
10     params: {
11       id: 12345
12     }
13   })
14  .then( res => {
15     console.log(res.data);
16   })
17   .catch( err => {
18     console.log(err);
19   });
```

2. post 请求

```
1  axios.post('/user', {
2    firstName: 'Fred',
3    lastName: 'Flintstone'
4  })
5   .then( res => {
6     console.log(res.data);
7   })
8   .catch( err => {
9     console.log(err);
10  });
```

11.3 消息订阅-发布机制

1. 工具库：PubSubJS
2. 下载：npm install pubsub-js--save
3. 使用方式：
    1. `import PubSub from 'pubsub-js' // 引入`
    2. `PubSub.subscribe('itchao', function(data) { }) // 订阅`消息
    3. `PubSub.publish('itchao', data) // 发布消息`

11.4 扩展：Fetch

11.4.1 文档

1. https://github.github.io/fetch/
2. https://segmentfault.com/a/1190000003810652

11.4.2 特点

1. fetch：原生函数，不再使用 XMLHttpRequest 对象提交 ajax 请求
2. 兼容性存在问题，老版本浏览器可能不支持

11.4.3 相关 API

1. get 请求

```
fetch(url).then( res => {
    return res.json()
  }).then( data => {
    console.log(data)
  }).catch( err => {
    console.log(err)
  });
```

2. post 请求

```
  fetch(url, {
    method: "POST",
    body: JSON.stringify(data),
  }).then( data => {
    console.log(data)
  }).catch( err => {
    console.log(err)
  });
```

12. React 路由

12.1 相关概念理解

12.1.1 SPA（单页面富应用）的理解

1. 单页面 Web 应用（SPA）
2. 整个应用**只有一个完整页面**
3. 点击页面中的链接**不会刷新页面**，只会做页面的**局部更新**
4. 数据都需要通过 ajax 请求获取，并在前端异步展示

12.1.2 路由的理解

一. 什么是路由?

1. 一个路由就是一个映射关系（key: value）
2. key 为路径，value 可能是 function 或 component

二. 路由分类

1. 后端路由：
    1. 理解：value 是 function，用来处理客户端提交的请求
    2. 注册路由：router.get(path, function(req, res))
    3. 工作过程：当 node 接收到一个请求时，根据请求路径找到匹配的路由，调用路由中的函数来处理请求，返回响应数据
2. 前端路由：
    1. 浏览器端路由：value 是 component，用于展示页面内容
    2. 注册路由：`<Router path='/test' component={Test}>`
    3. 工作过程：当浏览器的 path 变为 /test 时，当前路由组件会变成 Test 组件

12.1.3 react-router-dom 的理解

1. react 的一个插件库
2. 专门用来实现一个 SPA 应用
3. 基于 react 的项目基本都会用到此库

12.2 React 路由基本使用

1. 明确好界面中的导航区、展示区
2. 导航区的a标签改为Link标签

    `<Link to='/abc'>Demo</Link>`

3. 展示区写Route标签进行路径匹配

   `<Route path='/abc' component={Demo}/>`

   - 如果出现报错可能需要写成如下格式：

```
<Routes>
  <Route path='/about' element={<About/}/>
  <Route path='/home' element={<Home/}/>
</Routes>
```

4. 的最外侧包裹了一个 或

12.3 路由组件与一般组件的区别

1. 写法不同：
   - 一般组件：
   - 路由组件：
2. 存放位置不同：
   - 一般组件：components 文件夹下
   - 路由组件：pages 文件夹下
3. 接收到的 props 不同：
   - 一般组件：写组件标签时传递什么，就接收到什么
   - 路由组件：接收到三个固定属性
     - history：
       - go: f go(n)
       - goBack: f goBack( )
       - goForward: f goForward( )
       - push: f push(path, state)
       - replace: f replace(path, state)
     - location:
       - pathname: '/about'
       - search: ' '
       - state: undefined
     - match:
       - params: { }
       - path: '/about'

- url: '/about'

12.4 NavLink 和封装 NavLink

1. NavLink 可以实现路由链接的高亮显示，通过 activeClassName 指定样式名（该属性现在好像不支持）
2. 标签体内容是一个特殊的标签属性
3. 通过 this.props.children 可以获取到标签体内容

12.5 Switch 基本使用

1. 通常情况下，path 和 component 是唯一对应的关系
2. Switch 可以提高路由匹配效率（单一匹配）

12.6 解决多级路径刷新页面样式丢失问题

1. public/index.html 中，引入样式时不写 ./ 写 /（常用）
2. public/index.html 中，引入样式时不写 ./ 写 %PUBLIC_URL&（常用，但只能在 React 中使用）
3. 使用 `<HashRouter></HashRouter>`

12.7 路由的严格匹配和模糊匹配

1. 默认使用模糊匹配（【输入的路径】必须包含【匹配的路径】，且顺序要一致）
2. 开启严格匹配: `<Route exact={true} path='/about' component={About} />`
3. 严格匹配不要随便开启，需要时再开，有时开启会导致无法继续匹配二级路由

12.8 Redirect 重定向的使用

1. 一般写在所有路由注册的最下方，当所有路由都无法匹配时，跳转到 Redirect 指定的路由
2. 示例:

```
    <Routes>
        <Route path='/about' element={<About />} />
        <Route path='/home' element={<Home />} />
        <Redirect to='/about' />
    </Routes>
```

3. 注意：好像现在不支持`<Redirct />`，需要后续再查下现在重定向怎么使用

12.9 嵌套路由

1. 注册子路由时要写上父路由的 path 值
2. 路由的匹配是按照注册路由的顺序进行

注意：

- 上述规则好像在新的路由里面不适用了，需要更改一下

嵌套路由新规则：（示例如下）

1. 父路由中注册路由：

```
    <Routes>
        <Route path='/about/*' element={<About />} />  // 关键点：path属性的末尾需要追加 /*
        <Route path='/home/*' element={<Home />} />
    </Routes>
```

2. 子路由中 NavLink 组件：

```
    <NavLink to='/home/news'>News</NavLink>  // 关键点：to属性中path需要写完整的嵌套路径
    <NavLink to='/home/message'>Message</NavLink>
```

3. 子路由中注册路由:

```
<Routes>
    <Route path='/news' element={<News />} />
// 关键点：path属性的只需要写子路由路径即可
    <Route path='/message' element={<Message />} />
</Routes>
```

12.10 向路由组件传递参数

1. params 参数 (V6 好像不适用)
    1. 路由链接（携带参数）：`<Link to='/demo/test/itchao/22'>详情</Link>`
    2. 注册路由（声明接收）：`<Route path='/demo/test/:name/:age' component={Test} />`
    3. 接收参数：`this.props.match.params`
2. search 参数 (V6 好像不适用)
    1. 路由链接（携带参数）：`<Link to='/demo/test?name=itchao&age=22'>详情</Link>`
    2. 注册路由（无需声明，正常注册即可）：`<Route path='/demo/test' component={Test} />`
    3. 接收参数：`this.props.location.search`
    4. 注意：获取到的 search 是 urlencoded 编码字符串，需要借助 querystring 解析
3. state 参数 (V6 好像不适用)
    1. 路由链接（携带参数）：`<Link to={{pathname: '/demo/test', state:{name: 'itchao', age: 22}}>详情</Link>`
    2. 注册路由（无需声明，正常注册即可）：`<Route path='/demo/test' component={Test} />`
    3. 接收参数：`this.props.location.state`
    4. 注意：刷新也可保留参数

12.11 编程式路由导航

1. 借助 this.props.history 对象上的 API 操作路由进行跳转、前进、后退
    - this.props.history.push( )
    - this.props.history.replace( )
    - this.props.history.goBack( )
    - this.props.history.goForward( )
    - this.props.history.go( )

12.12 withRouter

- withRouter 可以加工一般组件，让一般组件具备路由组件所特有的 API
- withRouter 的返回值是一个新组件
- 用法示例：`export default withRouter(Header)`

12.13 BrowserRouter 与 HashRouter 的区别

1. 底层原理不一样：
    - BrowserRouter 使用的是 H5 的 history API，不兼容 IE9 及以下版本
    - HashRouter 使用的是 URL 的哈希值
2. path 的表现形式不一样
    - BrowserRouter 的路径中没有 #，例如：localhost:3000/demo/test
    - HashRouter 的路径包含 #，例如：localhost:3000/#/demo/test
3. 刷新后对路由 state 参数的影响
    - BrowserRouter 没有任何影响，因为 state 保存在 history 对象中
    - HashRouter 刷新后会导致路由 state 参数丢失
4. 注意：HashRouter 可用于解决一些路径错误相关问题

13. React UI 组件库

13.1. 流行的开源React UI组件库

1. material-ui(国外)
    1. 官网: http://www.material-ui.com/#/
    2. github: https://github.com/callemall/material-ui
2. ant-design(国内蚂蚁金服)
    1. 官网: https://ant.design/index-cn
    2. Github: https://github.com/ant-design/ant-design/

14. redux

14.1 redux 理解

14.1.1 学习文档

1. 英文文档: https://redux.js.org/
2. 中文文档: http://www.redux.org.cn/
3. Github: https://github.com/reactjs/redux

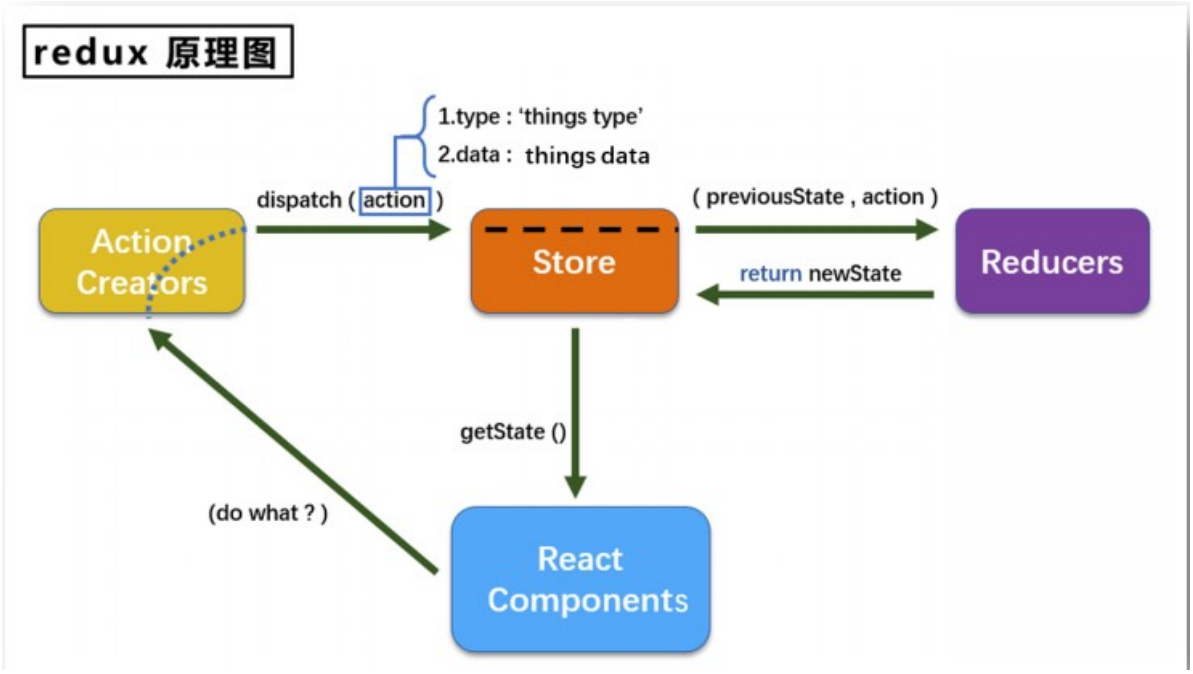
14.1.2 redux 是什么

1. redux 是一个专门用于做**状态管理**的JS库（不是 react 插件库）
2. 可以用于 react、angular、vue 等项目中，但在 react 中用的最多
3. 作用：集中式管理 react 应用中多个组件**共享状态**

14.1.3 redux 使用场景

1. 某个组件的状态，需要让其他组件拿到（共享）
2. 一个组件需要改变另一个组件的状态（通信）
3. 总体原则：尽量不用，除非必须用的情况才考虑使用

14.1.4 redux 工作流程



14.2 redux 三个核心概念

14.2.1 action

1. 动作对象
2. 包含2个属性
    - type：标识属性，值为字符串，唯一，必要属性
    - data：数据属性，值为任意类型，可选属性
3. 例子：{ type: 'ADD_STUDENT',data:{name: 'itchao',age:22} }

14.2.2 reducer

1. 初始化状态
2. 加工状态：根据旧的 state 和 action，产生新的 state 的**纯函数**

14.2.3 store

1. 将state、action、reducer联系在一起的对象
2. 如何得到此对象?
    1. import {createStore} from 'redux'
    2. import reducer from './reducers'
    3. const store = createStore(reducer)
3. 此对象的功能?
    1. getState(): 得到state

2. dispatch(action): 分发action, 触发reducer调用, 产生新的state
3. subscribe(listener): 注册监听, 当产生了新的state时, 自动调用

14.3 redux 核心 API

14.3.1 createstore( )

作用：创建包含指定 reducer 的 store 对象

14.3.2 store 对象

1. 作用: redux 库最核心的管理对象
2. 它内部维护着:
    1. state
    2. reducer
3. 核心方法:
    1. getState()
    2. dispatch(action)
    3. subscribe(listener)
4. 具体编码:
    1. store.getState()
    2. store.dispatch({type:'INCREMENT', number})
    3. store.subscribe(render)

14.3.3 applyMiddleware( )

作用：应用上基于 redux 的中间件(插件库)

14.3.4 combineReducers( )

作用：合并多个 reducer 函数

14.4 redux 异步编程

14.4.1 理解

1.redux默认是不能进行异步处理的,

2.某些时候应用中需要在redux中执行异步任务(ajax,定时器)

14.4.2 使用异步组件

- npm install --save redux-thunk

14.5 react-redux

14.5.1 理解

1.一个react插件库

2.专门用来简化react应用中使用redux

14.5.2 react-Redux将所有组件分成两大类

1.UI组件:

1. 只负责UI的呈现，不带有任何业务逻辑
2. 通过props接收数据(一般数据和函数)
3. 不使用任何Redux的API
4. 一般保存在components文件夹下

2.容器组件:

1. 负责管理数据和业务逻辑，不负责UI的呈现
2. 使用Redux的API
3. 一般保存在containers文件夹下

14.5.3 相关 API

1.Provider：让所有组件都可以得到state数据

```
<Provider store={store}>
  <App />
</Provider>
```

2. connect：用于包装 UI 组件生成容器组件

```
import { connect } from 'react-redux'
connect(
  mapStateToprops,
  mapDispatchToProps
)(Counter)
```

3. mapStateToprops：将外部的数据（即 state 对象）转换为 UI 组件的标签属性

```
const mapStateToprops = function (state) {
  return { value: state
         }
}
```

4. mapDispatchToProps：将分发 action 的函数转换为 UI 组件的标签属性

14.6 redux 调试工具

14.6.1 安装 chrome 浏览器插件

- Redux DevTools

14.6.2 下载工具依赖包

- npm install --save-dev redux-devtools-extension

15. 纯函数和高阶函数

15.1 纯函数

1. 一类特别的函数: 只要是同样的输入(实参)，必定得到同样的输出(返回)
2. 必须遵守以下一些约束
    1. 不得改写参数数据
    2. 不会产生任何副作用，例如网络请求，输入和输出设备
    3. 不能调用 Date.now()或者 Math.random()等不纯的方法
3. redux 的 reducer 函数必须是一个纯函数

15.2 高阶函数

1. 理解: 一类特别的函数
    1. 情况 1: 参数是函数
    2. 情况 2: 返回是函数
2. 常见的高阶函数:
    1. 定时器设置函数
    2. 数组的 forEach()/map()/filter()/reduce()/find()/bind()
    3. promise
    4. react-redux 中的 connect 函数
3. 作用: 能实现更加动态, 更加可扩展的功能