React 学习笔记

第一章 React 入门

1.1 React 简介

1. 英文官网: https://reactjs.org/
2. 中文官网: https://react.docschina.org/

1. React 是什么？

React 是用于构建用户界面的 JavaScript 库，是一个将数据渲染为 HTML 视图的开源 JavaScript 库。

1. 发送请求获取数据
2. 处理数据（过滤、整理格式等）
3. 操作 DOM 呈现

2. 谁开发的？

由 Facebook 开发，且开源。

1. 起初由 Facebook 的软件工程师 Jordan Walke 创建
2. 于 2011 年部署于 Facebook 的 newsfeed
3. 随后在 2012 年部署于 Instagram

4. 2013 年 5 月宣布开源
5. .....

近十年“陈酿”，React 正在被腾讯、阿里等一线大厂广泛使用。

3. 为什么要学？

1. 原生 JavaScript 操作 DOM 繁琐、效率低（ DOM-API 操作 UI ）。
2. 使用 JavaScript 直接操作 DOM，浏览器会进行大量的 重绘重排 。
3. 原生 JavaScript 没有 组件化 编码方案，代码复用率低。

4. React 的特点

1. 采用 组件化 模式、 声明式编码 ，提高开发效率及组件复用率。
2. 在 React Native 中可以使用 React 语法进行 移动端开发 。
3. 使用 虚拟 DOM + 优秀的 Diffing 算法 ，尽量减少与真实 DOM 的交互。

5. 学习 React 之前需要掌握的 JavaScript 基础知识

1. 判断 this 指向
2. class（类）
3. ES6 语法规范
4. npm 包管理器
5. 原型、原型链
6. 数组常用方法
7. 模块化

1.2 React 的基本使用

1.2.1 Hello React 案例

```
<body>
    <!-- 准备好一个“容器” -->
    <div id="test"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel" > /* 此处一定要写babel */
        //1.创建虚拟DOM
        const VDOM = <h1>Hello,React</h1> /* 此处一定不要写引号，因为不是字符串 */
        //2.渲染虚拟DOM到页面
        ReactDOM.render(VDOM,document.getElementById('test'))
    </script>
</body>
```

1.2.2 相关 js 库

1. react.js：React 核心库。
2. react-dom.js：提供操作 DOM 的 react 扩展库。
3. babel.min.js：解析 JSX 语法代码转为 JS 代码的库。

1.2.3 创建虚拟 DOM 的两种方式

1. 使用 JSX 创建虚拟 DOM

```
<body>
    <!-- 准备好一个“容器” -->
    <div id="test"></div>

    <!-- 引入react核心库 -->
```

```
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel" > /* 此处一定要写babel */
        //1.创建虚拟DOM
        const VDOM = <h1><span>Hello,React<span></h1> /* 此处一定不要写引号，因为不是字符串 */
        //2.渲染虚拟DOM到页面
        ReactDOM.render(VDOM,document.getElementById('test'))
    </script>
</body>
```

2. 使用 JS 创建虚拟 DOM

```
<body>
    <!-- 准备好一个“容器” -->
    <div id="test"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>

    <script type="text/javascript" >
        //1.创建虚拟DOM
        const VDOM = React.createElement('h1',{id:'title'},React.createElement('span',{},'Hello,React'))
        //2.渲染虚拟DOM到页面
        ReactDOM.render(VDOM,document.getElementById('test'))
    </script>
</body>
```

1.2.4 虚拟 DOM 与真实 DOM

```
<body>
    <!-- 准备好一个“容器” -->
    <div id="test"></div>
    <div id="demo"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel" > /* 此处一定要写babel */
        //1.创建虚拟DOM
        const VDOM = (  /* 此处一定不要写引号，因为不是字符串 */
            <h1 id="title">
                <span>Hello,React</span>
            </h1>
        )
        //2.渲染虚拟DOM到页面
        ReactDOM.render(VDOM,document.getElementById('test'))

        const TDOM = document.getElementById('demo')
        console.log('虚拟DOM',VDOM);
        console.log('真实DOM',TDOM);
        debugger;
        // console.log(typeof VDOM);
        // console.log(VDOM instanceof Object);
    </script>
</body>
```

关于虚拟 **DOM**：

1. 本质是 Object 类型的对象（一般对象）
2. 虚拟 DOM 比较“轻”，真实 DOM 比较“重”，因为虚拟 DOM 是 React 内部在用，无需真实 DOM 上那么多的属性。
3. 虚拟 DOM 最终会被 React 转化为 真实 DOM，呈现在页面上。

1.3 React JSX

1.3.1 JSX 语法规范

```
<body>
    <!-- 准备好一个“容器” -->
    <div id="test"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel" >
        const myId = 'aTgUiGu'
        const myData = 'HeLlo,rEaCt'

        //1.创建虚拟DOM
        const VDOM = (
            <div>
                <h2 className="title" id={myId.toLowerCase()}>
                    <span style={{color:'white',fontSize:'29px'}}>{myData.toLowerCase()}</span>
                </h2>
                <h2 className="title" id={myId.toUpperCase()}>
                    <span style={{color:'white',fontSize:'29px'}}>{myData.toLowerCase()}</span>
                </h2>
                <input type="text"/>
            </div>
        )
        //2.渲染虚拟DOM到页面
        ReactDOM.render(VDOM,document.getElementById('test'))
    </script>
</body>
```

jsx 语法规则：

1. 定义虚拟 DOM 时，不要写引号。
2. 标签中混入 JS 表达式时要用{}。
3. 样式的类名指定不要用`class`，要用`className`。
4. 内联样式，要用`style={{key:value}}`的形式去写。
5. 只有一个根标签
6. 标签必须闭合
7. 标签首字母
    1. 若小写字母开头，则将该标签转为 html 中同名元素，若 html 中无该标签对应的同名元素，则报错。
    2. 若大写字母开头，react 就去渲染对应的组件，若组件没有定义，则报错。

1.3.2 JSX 小练习

```
<body>
    <!-- 准备好一个“容器” -->
    <div id="test"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel" >
        //模拟一些数据
        const data = ['Angular','React','Vue']
        //1.创建虚拟DOM
        const VDOM = (
            <div>
                <h1>前端js框架列表</h1>
                <ul>
                    {
                        data.map((item,index)=>{
                            return <li key={index}>{item}</li>
                        })
```

```
                    }
                </ul>
            </div>
        )
        //2.渲染虚拟DOM到页面
        ReactDOM.render(VDOM,document.getElementById('test'))
    </script>
</body>
```

一定注意区分：【js 语句(代码)】与【js 表达式】

1. 表达式：一个表达式会产生一个值，可以放在任何一个需要值的地方。
    - 下面这些都是表达式：
        1. `a`
        2. `a+b`
        3. `demo(1)`
        4. `arr.map()`
        5. `function test () {}`
2. 语句(代码)：
    - 下面这些都是语句(代码)：
        1. `if(){}`
        2. `for(){}`
        3. `switch(){case:xxxx}`

1.4 模块与组件、模块化与组件化的理解

1.4.1 模块

1. 理解：向外提供特定功能的 js 程序, 一般就是一个 js 文件
2. 为什么要拆成模块：随着业务逻辑增加，代码越来越多且复杂。
3. 作用：复用 js, 简化js的编写, 提高 js 运行效率

1.4.2 组件

1. 理解：用来实现局部功能效果的代码和资源的集合(html/css/js/image 等等)
2. 为什么要用组件： 一个界面的功能更复杂
3. 作用：复用编码, 简化项目编码, 提高运行效率

1.4.3 模块化

当应用的 js 都以模块来编写的, 这个应用就是一个模块化的应用。

1.4.4 组件化

当应用是以多组件的方式实现, 这个应用就是一个组件化的应用。

第二章 React 面向组件编程

2.1 基本理解和使用

使用React开发者工具调试

注意：

1. 组件名必须首字母大写
2. 虚拟 DOM 元素只能有一个根元素
3. 虚拟 DOM 元素必须有结束标签

渲染类组件标签的基本流程

1. React 内部会创建组件实例对象
2. 调用 `render()` 得到虚拟 DOM, 并解析为真实 DOM
3. 插入到指定的页面元素内部

React 中定义组件

函数式组件

简单组件：无状态

```
<body>
    <!-- 准备好一个“容器” -->
    <div id="test"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel">
        //1.创建函数式组件
        function MyComponent(){
            console.log(this); //此处的this是undefined，因为babel编译后开启了严格模式
            return <h2>我是用函数定义的组件(适用于【简单组件】的定义)</h2>
        }
        //2.渲染组件到页面
        ReactDOM.render(<MyComponent/>, document.getElementById('test'))
    </script>
</body>
```

执行了 `ReactDOM.render(\<MyComponent/>.....)` 之后，发生了什么？

1. React 解析组件标签，找到了 `MyComponent` 组件。
2. 发现组件是使用函数定义的，随后调用该函数，将返回的虚拟 DOM 转为真实 DOM，随后呈现在页面中。

类式组件

复杂组件：有状态

```
<body>
    <!-- 准备好一个“容器” -->
    <div id="test"></div>

    <!-- 引入react核心库 -->
    <script type="text/javascript" src="../js/react.development.js"></script>
    <!-- 引入react-dom，用于支持react操作DOM -->
    <script type="text/javascript" src="../js/react-dom.development.js"></script>
    <!-- 引入babel，用于将jsx转为js -->
    <script type="text/javascript" src="../js/babel.min.js"></script>

    <script type="text/babel">
        //1.创建类式组件
        class MyComponent extends React.Component {
            render(){
                //render是放在哪里的？—— MyComponent的原型对象上，供实例使用。
                //render中的this是谁？—— MyComponent的实例对象 <-> MyComponent组件实例对象。
                console.log('render中的this:',this); //MyComponent的实例对象
                return <h2>我是用类定义的组件(适用于【复杂组件】的定义)</h2>
            }
        }
        //2.渲染组件到页面
        ReactDOM.render(<MyComponent/>, document.getElementById('test'))
    </script>
</body>
```

执行了 `ReactDOM.render(\<MyComponent/>.....)` 之后，发生了什么？

1. React 解析组件标签，找到了 `MyComponent` 组件。
2. 发现组件是使用类定义的，随后 new 出来该类的实例，并通过该实例调用到原型上的 `render` 方法。
3. 将 `render` 返回的虚拟 DOM 转为真实 DOM，随后呈现在页面中。

类的总结：

1. 类中的构造器不是必须要写的，要对实例进行一些初始化的操作，如添加指定属性时才写。
2. 如果 A 类继承了 B 类，且 A 类中写了构造器，那么 A 类构造器中的 `super` 是必须要调用的。
3. 类中所定义的方法，都放在了类的原型对象上，供实例去使用。

2.2 组件三大核心属性1: state

2.2.1 效果

需求: 定义一个展示天气信息的组件

1. 默认展示天气炎热 或 凉爽
2. 点击文字切换天气

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

    //构造器调用几次?  ----  1次
    constructor(props){
        console.log('constructor');
        super(props)
        //初始化状态
        this.state = {isHot:false, wind:'微风'}
        //解决changeWeather中this指向问题
        this.changeWeather = this.changeWeather.bind(this)
    }

    //render调用几次?  ----  1+n次 1是初始化的那次 n是状态更新的次数
    render(){
        console.log('render');
        //读取状态
        const {isHot,wind} = this.state
        return <h1 onClick={this.changeWeather}>今天天气很{isHot ? '炎热' : '凉爽'}, {wind}</h1>
    }

    //changeWeather调用几次?  ----  点几次调几次
    changeWeather(){
```

```
        //changeWeather放在哪里？ ---- Weather的原型对象上，供实例使用
        //由于changeWeather是作为onClick的回调，所以不是通过实例调用的，是直接调用
        //类中的方法默认开启了局部的严格模式，所以changeWeather中的this为undefined

        console.log('changeWeather');
        //获取原来的isHot值
        const isHot = this.state.isHot
        //严重注意：状态必须通过setState进行更新,且更新是一种合并，不是替换。
        this.setState({isHot: !isHot})

        //严重注意：状态(state)不可直接更改，下面这行就是直接更改！！！
        //this.state.isHot = !isHot //这是错误的写法
    }
}
//2.渲染组件到页面
ReactDOM.render(<Weather/>, document.getElementById('test'))
```

简写方式

```
//1.创建组件
class Weather extends React.Component{
    //初始化状态
    state = {isHot:false,wind:'微风'}

    render(){
        const {isHot,wind} = this.state
        return <h1 onClick={this.changeWeather}>今天天气很{isHot ? '炎热' : '凉爽'}，{wind}</h1>
    }

    //自定义方法----要用赋值语句的形式+箭头函数
    changeWeather = () => {
        const isHot = this.state.isHot
        this.setState({isHot: !isHot})
    }
}
//2.渲染组件到页面
ReactDOM.render(<Weather/>, document.getElementById('test'))
```

2.2.2 理解

1. `state` 是组件对象最重要的属性, 值是对象(可以包含多个 key-value 的组合)
2. 组件被称为"状态机", 通过更新组件的 `state` 来更新对应的页面显示(重新渲染组件)

2.2.3 强烈注意

1. 组件中 `render` 方法中的 `this` 为组件实例对象
2. 组件自定义的方法中 `this` 为 `undefined`，如何解决？
    1. 强制绑定 `this`：通过函数对象的 `bind()`
    2. 箭头函数
3. 状态数据，不能直接修改或更新，必须通过 `setState` 进行修改

2.3 组件三大核心属性2: props

2.3.1 效果

需求: 自定义用来显示一个人员信息的组件

1. 姓名必须指定，且为字符串类型；
2. 性别为字符串类型，如果性别没有指定，默认为男
3. 年龄为字符串类型，且为数字类型，默认值为18

```
//创建组件
class Person extends React.Component{
    render(){
        const {name,age,sex} = this.props
        return (
            <ul>
                <li>姓名: {name}</li>
                <li>性别: {sex}</li>
                <li>年龄: {age + 1}</li>
            </ul>
        )
```

```
    }
}
//渲染组件到页面
ReactDOM.render(<Person name="jerry" age={19}  sex="男"/>,
document.getElementById('test1'))
ReactDOM.render(<Person name="tom" age={18} sex="女"/>,
document.getElementById('test2'))

const p = {name:'老刘',age:18,sex:'女'}
ReactDOM.render(<Person {...p}/>,
document.getElementById('test3'))
```

对 **props** 进行限制

```
//创建组件
class Person extends React.Component{
    render(){
        const {name, age, sex} = this.props
        //props是只读的
        //this.props.name = 'jack' //此行代码会报错，因为props是只读的
        return (
            <ul>
                <li>姓名：{name}</li>
                <li>性别：{sex}</li>
                <li>年龄：{age + 1}</li>
            </ul>
        )
    }
}
//对标签属性进行类型、必要性的限制
Person.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
Person.defaultProps = {
    sex: '男',//sex默认值为男
    age: 18 //age默认值为18
}
//渲染组件到页面
```

```
ReactDOM.render(<Person name={100} speak={speak}/>,
document.getElementById('test1'))
ReactDOM.render(<Person name="tom" age={18} sex="女"/>,
document.getElementById('test2'))

const p = {name:'老刘',age:18,sex:'女'}
ReactDOM.render(<Person {...p}/>,
document.getElementById('test3'))

function speak(){
    console.log('我说话了');
}
```

props 的简写方式

```
//创建组件
class Person extends React.Component{

    constructor(props){
        //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
        super(props)
        console.log('constructor',this.props);
    }

    //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
    }

    //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
    }

    render(){
        const {name,age,sex} = this.props
        //props是只读的
        //this.props.name = 'jack' //此行代码会报错，因为props是只读的
```

```
        return (
            <ul>
                <li>姓名: {name}</li>
                <li>性别: {sex}</li>
                <li>年龄: {age + 1}</li>
            </ul>
        )
    }
}

//渲染组件到页面
ReactDOM.render(<Person name="jerry"/>,
document.getElementById('test1'))
```

函数组件使用

```
//创建组件
function Person (props){
    const {name,age,sex} = props
    return (
        <ul>
            <li>姓名: {name}</li>
            <li>性别: {sex}</li>
            <li>年龄: {age}</li>
        </ul>
    )
}
Person.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
}

//指定默认标签属性值
Person.defaultProps = {
    sex: '男',//sex默认值为男
    age: 18 //age默认值为18
}
//渲染组件到页面
ReactDOM.render(<Person name="jerry"/>,
document.getElementById('test1'))
```

2.3.2 理解

1. 每个组件对象都会有 `props` (properties 的简写)属性
2. 组件标签的所有属性都保存在 `props` 中

2.3.3 作用

1. 通过标签属性从组件外向组件内传递变化的数据
2. 注意: 组件内部不要修改 `props` 数据

2.3.4 编码操作

1. 内部读取某个属性值

```
this.props.name
```

2. 对 `props` 中的属性值进行类型限制和必要性限制
   - 第一种方式（React v15.5 开始已弃用）：

```
Person.propTypes = {
    name: React.PropTypes.string.isRequired,
    age: React.PropTypes.number
}
```

   - 第二种方式（新）：使用 `prop-types` 库进限制（需要引入 `prop-types` 库）

```
Person.propTypes = {
    name: PropTypes.string.isRequired,
    age: PropTypes.number.
}
```

3. 扩展属性: 将对象的所有属性通过props传递

```
<Person {...person}/>
```

4. 默认属性值：

```
Person.defaultProps = {
    age: 18,
    sex:'男'
}
```

5. 组件类的构造函数

```
constructor(props){
    super(props)
    console.log(props)//打印所有属性
}
```

2.4 组件三大核心属性3: refs与事件处理

2.4.1 效果

需求: 自定义组件, 功能说明如下:

1. 点击按钮, 提示第一个输入框中的值
2. 当第2个输入框失去焦点时, 提示这个输入框中的值

字符串形式的 ref

```
//创建组件
class Demo extends React.Component{
    //展示左侧输入框的数据
    showData = ()=>{
        const {input1} = this.refs
        alert(input1.value)
    }
    //展示右侧输入框的数据
    showData2 = ()=>{
        const {input2} = this.refs
        alert(input2.value)
    }
    render(){
        return(
            <div>
                <input ref="input1" type="text" placeholder="点击按钮提示数据"/> 
                <button onClick={this.showData}>点我提示左侧的数据</button> 
                <input ref="input2" onBlur={this.showData2} type="text" placeholder="失去焦点提示数据"/>
            </div>
        )
    }
```

```
}
//渲染组件到页面
ReactDOM.render(<Demo a="1"
b="2"/>,document.getElementById('test'))
```

回调函数形式的 ref

```
//创建组件
class Demo extends React.Component{
    //展示左侧输入框的数据
    showData = ()=>{
        const {input1} = this
        alert(input1.value)
    }
    //展示右侧输入框的数据
    showData2 = ()=>{
        const {input2} = this
        alert(input2.value)
    }
    render(){
        return(
            <div>
                <input ref={c => this.input1 = c } type="text"
placeholder="点击按钮提示数据"/> 
                <button onClick={this.showData}>点我提示左侧的数据
</button> 
                <input onBlur={this.showData2} ref={c =>
this.input2 = c } type="text" placeholder="失去焦点提示数据"/> 
            </div>
        )
    }
}
//渲染组件到页面
ReactDOM.render(<Demo a="1"
b="2"/>,document.getElementById('test'))
```

回调 ref 中回调执行次数的问题

```
//创建组件
class Demo extends React.Component{

    state = {isHot:false}
```

```
    showInfo = ()=>{
        const {input1} = this
        alert(input1.value)
    }

    changeWeather = ()=>{
        //获取原来的状态
        const {isHot} = this.state
        //更新状态
        this.setState({isHot:!isHot})
    }

    saveInput = (c)=>{
        this.input1 = c;
        console.log('@',c);
    }

    render(){
        const {isHot} = this.state
        return(
            <div>
                <h2>今天天气很{isHot ? '炎热':'凉爽'}</h2>
                {/*<input ref={(c)=>{this.input1 = c;console.log('@',c);}} type="text"/><br/><br/>*/}
                <input ref={this.saveInput} type="text"/><br/><br/>
                <button onClick={this.showInfo}>点我提示输入的数据</button>
                <button onClick={this.changeWeather}>点我切换天气</button>
            </div>
        )
    }
}
//渲染组件到页面
ReactDOM.render(<Demo/>, document.getElementById('test'))
```

createRef 的使用

```
//创建组件
class Demo extends React.Component{
    /*
```

```
    React.createRef调用后可以返回一个容器，该容器可以存储被ref所标识的节点,该容器是"专人专用"的
    */
    myRef = React.createRef()
    myRef2 = React.createRef()
    //展示左侧输入框的数据
    showData = ()=>{
        alert(this.myRef.current.value);
    }
    //展示右侧输入框的数据
    showData2 = ()=>{
        alert(this.myRef2.current.value);
    }
    render(){
        return(
            <div>
                <input ref={this.myRef} type="text" placeholder="点击按钮提示数据"/> 
                <button onClick={this.showData}>点我提示左侧的数据</button> 
                <input onBlur={this.showData2} ref={this.myRef2} type="text" placeholder="失去焦点提示数据"/> 
            </div>
        )
    }
}
//渲染组件到页面
ReactDOM.render(<Demo a="1" b="2"/>,
document.getElementById('test'))
```

2.4.2 理解

组件内的标签可以定义 ref 属性来标识自己

2.4.3 编码

1. 字符串形式的 ref

```
<input ref="input1"/>
```

1. 回调形式的 ref

```
<input ref={(c)=>{this.input1 = c}}
```

1. `createRef` 创建 ref 容器·

```
myRef = React.createRef()
<input ref={this.myRef}/>
```

2.4.4 事件处理

1. 通过 onXxx 属性指定事件处理函数(注意大小写)
    1. React 使用的是自定义(合成)事件, 而不是使用的原生 DOM 事件
    2. React 中的事件是通过事件委托方式处理的(委托给组件最外层的元素)
2. 通过 `event.target` 得到发生事件的 DOM 元素对象

```
//创建组件
class Demo extends React.Component{
    //创建ref容器
    myRef = React.createRef()
    myRef2 = React.createRef()

    //展示左侧输入框的数据
    showData = (event)=>{
        console.log(event.target);
        alert(this.myRef.current.value);
    }

    //展示右侧输入框的数据
    showData2 = (event)=>{
        alert(event.target.value);
    }

    render(){
        return(
            <div>
                <input ref={this.myRef} type="text" placeholder="点击按钮提示数据"/> 
                <button onClick={this.showData}>点我提示左侧的数据</button> 
                <input onBlur={this.showData2} type="text" placeholder="失去焦点提示数据"/> 
            </div>
        )
    }
}
```

```
//渲染组件到页面
ReactDOM.render(<Demo a="1" b="2"/>,
document.getElementById('test'))
```

2.5 收集表单数据

2.5.1 效果

需求: 定义一个包含表单的组件

输入用户名密码后, 点击登录提示输入信息

非受控组件

```
//创建组件
class Login extends React.Component{
    handleSubmit = (event)=>{
        event.preventDefault() //阻止表单提交
        const {username, password} = this
        alert(`你输入的用户名是: ${username.value},你输入的密码是: ${password.value}`)
    }
    render(){
        return(
            <form onSubmit={this.handleSubmit}>
                用户名: <input ref={c => this.username = c} type="text" name="username"/>
                密码: <input ref={c => this.password = c} type="password" name="password"/>
                <button>登录</button>
            </form>
        )
    }
}
//渲染组件
ReactDOM.render(<Login/>, document.getElementById('test'))
```

受控组件

```jsx
//创建组件
class Login extends React.Component{

    //初始化状态
    state = {
        username: '', //用户名
        password: '' //密码
    }

    //保存用户名到状态中
    saveUsername = (event) => {
        this.setState({username: event.target.value})
    }

    //保存密码到状态中
    savePassword = (event) => {
        this.setState({password: event.target.value})
    }

    //表单提交的回调
    handleSubmit = (event) => {
        event.preventDefault() //阻止表单提交
        const {username, password} = this.state
        alert(`你输入的用户名是: ${username},你输入的密码是: ${password}`)
    }

    render(){
        return(
            <form onSubmit={this.handleSubmit}>
                用户名: <input onChange={this.saveUsername} type="text" name="username"/>
                密码: <input onChange={this.savePassword} type="password" name="password"/>
                <button>登录</button>
            </form>
        )
    }
}
//渲染组件
ReactDOM.render(<Login/>, document.getElementById('test'))
```

2.5.2 理解

包含表单的组件分类

1. 受控组件
2. 非受控组件

2.6 高阶函数_函数柯里化

高阶函数：如果一个函数符合下面2个规范中的任何一个，那该函数就是高阶函数。

1. 若A函数，接收的参数是一个函数，那么A就可以称之为高阶函数。
2. 若A函数，调用的返回值依然是一个函数，那么A就可以称之为高阶函数。

常见的高阶函数有：Promise、setTimeout、arr.map()等等

函数的柯里化：通过函数调用继续返回函数的方式，实现多次接收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
        }
    }
}
const result = sum(1)(2)(3)
console.log(result)
```

```
//创建组件
class Login extends React.Component{
    //初始化状态
    state = {
        username:'', //用户名
        password:'' //密码
    }

    //保存表单数据到状态中
    saveFormData = (dataType)=>{
        return (event)=>{
            this.setState({[dataType]: event.target.value})
        }
    }
```

```
15
16      //表单提交的回调
17      handleSubmit = (event)=>{
18          event.preventDefault() //阻止表单提交
19          const {username, password} = this.state
20          alert(`你输入的用户名是: ${username},你输入的密码是:
    ${password}`)
21      }
22      render(){
23          return(
24              <form onSubmit={this.handleSubmit}>
25                  用户名: <input onChange=
    {this.saveFormData('username')} type="text" name="username"/>
26                  密码: <input onChange=
    {this.saveFormData('password')} type="password" name="password"/>
27                  <button>登录</button>
28              </form>
29          )
30      }
31  }
32  //渲染组件
33  ReactDOM.render(<Login/>, document.getElementById('test'))
```

不用函数柯里化的实现

```
1   //创建组件
2   class Login extends React.Component{
3       //初始化状态
4       state = {
5           username: '', //用户名
6           password: '' //密码
7       }
8
9       //保存表单数据到状态中
10      saveFormData = (dataType, event) => {
11          this.setState({[dataType]: event.target.value})
12      }
13
14      //表单提交的回调
15      handleSubmit = (event) => {
16          event.preventDefault() //阻止表单提交
17          const {username, password} = this.state
18          alert(`你输入的用户名是: ${username},你输入的密码是:
    ${password}`)
```

```
    }
    render() {
        return (
            <form onSubmit={this.handleSubmit}>
                用户名: <input onChange={event => this.saveFormData('username', event) } type="text" name="username"/>
                密码: <input onChange={event => this.saveFormData('password', event) } type="password" name="password"/>
                <button>登录</button>
            </form>
        )
    }
}
//渲染组件
ReactDOM.render(<Login/>, document.getElementById('test'))
```

2.7 组件的生命周期

2.7.1 效果

需求:定义组件实现以下功能:

1. 让指定的文本做显示 / 隐藏的渐变动画
2. 从完全可见，到彻底消失，耗时2S
3. 点击“不活了”按钮从界面中卸载组件

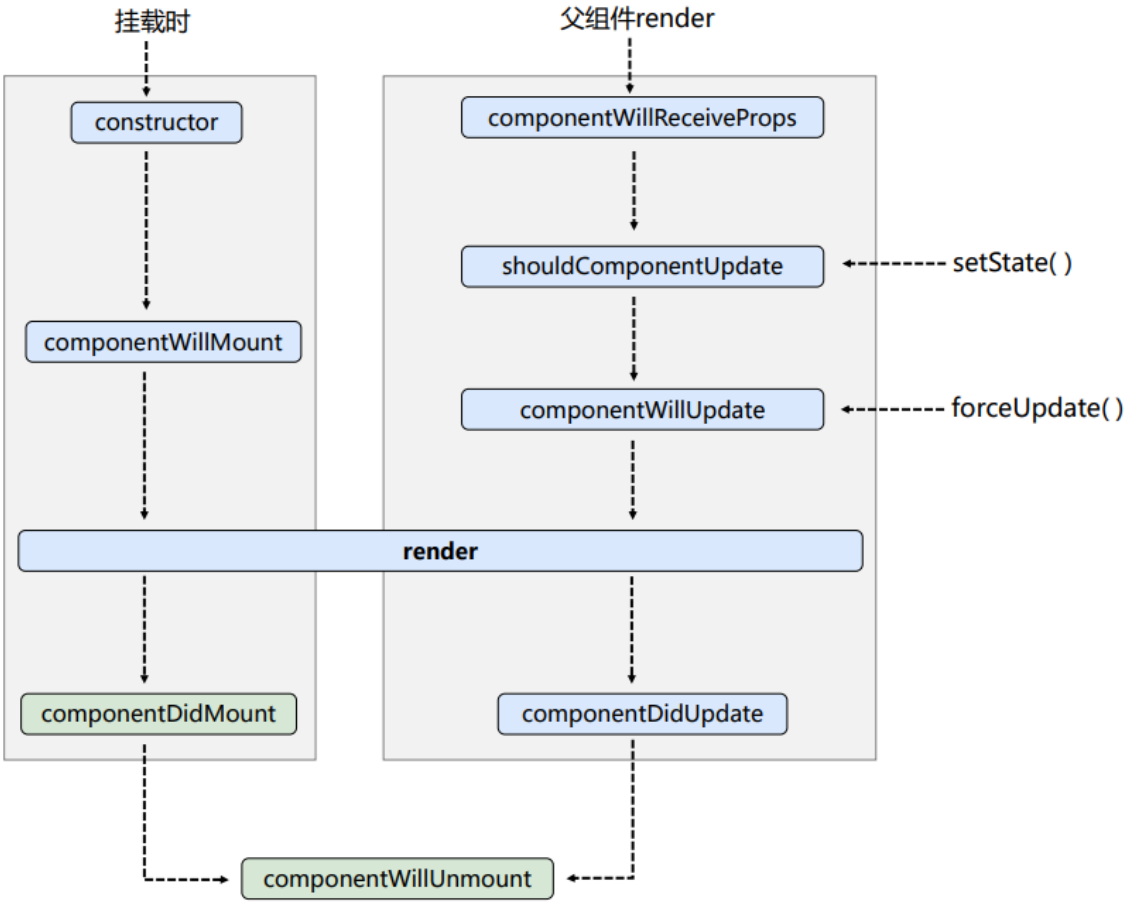
```
//创建组件
//生命周期回调函数 <=> 生命周期钩子函数 <=> 生命周期函数 <=> 生命周期钩子
class Life extends React.Component {

    state = {opacity: 1}

    death = () => {
        //卸载组件
 ReactDOM.unmountComponentAtNode(document.getElementById('test'))
    }

    //组件挂载完毕
```

```
    componentDidMount() {
        this.timer = setInterval(() => {
            //获取原状态
            let {opacity} = this.state
            //减小0.1
            opacity -= 0.1
            if(opacity <= 0) opacity = 1
            //设置新的透明度
            this.setState({opacity})
        }, 200);
    }

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

    //初始化渲染、状态更新之后
    render() {
        return (
            <div>
                <h2 style={{opacity:this.state.opacity}}>React学不会怎么办? </h2>
                <button onClick={this.death}>不活了</button>
            </div>
        )
    }
}
//渲染组件
ReactDOM.render(<Life/>, document.getElementById('test'))
```

2.7.2 理解

1. 组件从创建到死亡它会经历一些特定的阶段。
2. React组件中包含一系列勾子函数(生命周期回调函数), 会在特定的时刻调用。
3. 我们在定义组件时，会在特定的生命周期回调函数中，做特定的工作。

2.7.3 生命周期流程图(旧)



生命周期的三个阶段（旧）：

```
//创建组件
class Count extends React.Component{

    //构造器
    constructor(props){
        console.log('Count---constructor');
        super(props)
        //初始化状态
        this.state = {count:0}
    }

    //加1按钮的回调
    add = ()=>{
        //获取原状态
        const {count} = this.state
        //更新状态
        this.setState({count:count+1})
    }

    //卸载组件按钮的回调
    death = ()=>{
```

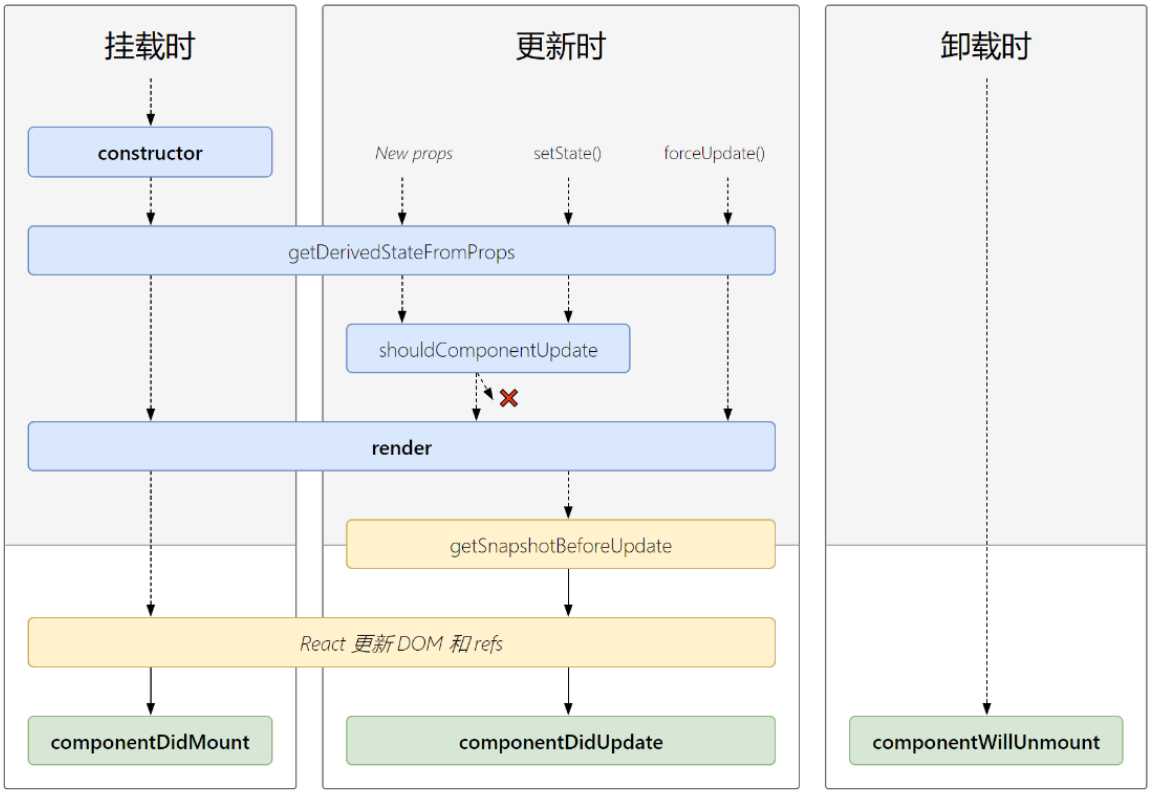
```
 ReactDOM.unmountComponentAtNode(document.getElementById('test')
)
    }

    //强制更新按钮的回调
    force = ()⇒{
        this.forceUpdate()
    }

    //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');
    }

    render(){
        console.log('Count---render');
        const {count} = this.state
        return(
```

```
            <div>
                <h2>当前求和为：{count}</h2>
                <button onClick={this.add}>点我+1</button>
                <button onClick={this.death}>卸载组件</button>
                <button onClick={this.force}>不更改任何状态中的数据，强制更新一下</button>
            </div>
        )
    }
}

//父组件A
class A extends React.Component{
    //初始化状态
    state = {carName: '奔驰'}

    changeCar = ()⇒{
        this.setState({carName: '奥拓'})
    }

    render(){
        return(
            <div>
                <div>我是A组件</div>
                <button onClick={this.changeCar}>换车</button>
                <B carName={this.state.carName}/>
            </div>
        )
    }
}

//子组件B
class B extends React.Component{
    //组件将要接收新的props的钩子
    componentWillReceiveProps(props){
        console.log('B---componentWillReceiveProps', props);
    }

    //控制组件更新的“阀门”
    shouldComponentUpdate(){
        console.log('B---shouldComponentUpdate');
        return true
    }
    //组件将要更新的钩子
    componentWillUpdate(){
```

```
109             console.log('B---componentWillUpdate');
110         }
111 
112         //组件更新完毕的钩子
113         componentDidUpdate(){
114             console.log('B---componentDidUpdate');
115         }
116 
117         render(){
118             console.log('B---render');
119             return(
120                 <div>我是B组件，接收到的车是:{this.props.carName}</div>
121             )
122         }
123     }
124 
125     //渲染组件
126     ReactDOM.render(<Count/>,document.getElementById('test'))
```

1. 初始化阶段: 由 `ReactDOM.render()` 触发---初次渲染
    1. `constructor()`
    2. `componentWillMount()`
    3. `render()`
    4. `componentDidMount()`
2. 更新阶段: 由组件内部 `this.setSate()` 或父组件重新 `render` 触发
    1. `shouldComponentUpdate()`
    2. `componentWillUpdate()`
    3. `render()`
    4. `componentDidUpdate()`
3. 卸载组件: 由 `ReactDOM.unmountComponentAtNode()` 触发
    1. `componentWillUnmount()`

2.7.4 生命周期流程图(新)



生命周期的三个阶段（新）：

```
//创建组件
class Count extends React.Component{
    //构造器
    constructor(props){
        console.log('Count---constructor');
        super(props)
        //初始化状态
        this.state = {count: 0}
    }

    //加1按钮的回调
    add = ()=>{
        //获取原状态
        const {count} = this.state
        //更新状态
        this.setState({count: count + 1})
    }

    //卸载组件按钮的回调
    death = ()=>{

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

    //强制更新按钮的回调
```

```
    force = ()=>{
        this.forceUpdate()
    }

    //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);
    }

    render(){
        console.log('Count---render');
        const {count} = this.state
        return(
            <div>
                <h2>当前求和为：{count}</h2>
```

```
                <button onClick={this.add}>点我+1</button>
                <button onClick={this.death}>卸载组件</button>
                <button onClick={this.force}>不更改任何状态中的数据，强制更新一下</button>
            </div>
        )
    }
}

//渲染组件
ReactDOM.render(<Count count={199}/>,
document.getElementById('test'))
```

1. 初始化阶段: 由 `ReactDOM.render()` 触发——初次渲染
    1. `constructor()`
    2. `getDerivedStateFromProps`
    3. `render()`
    4. `componentDidMount()`
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()`

2.7.5 重要的钩子

1. `render` ：初始化渲染或更新渲染调用
2. `componentDidMount` ：开启监听, 发送 ajax 请求
3. `componentWillUnmount` ：做一些收尾工作, 如: 清理定时器

2.7.6 即将废弃的钩子

1. `componentWillMount`
2. `componentWillReceiveProps`
3. `componentWillUpdate`

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，以后可能会被彻底废弃，不建议使用。

2.7.7 getSnapshotBeforeUpdate 的使用场景

```
class NewsList extends React.Component{

    state = {newsArr: []}

    componentDidMount(){
        setInterval(() => {
            //获取原状态
            const {newsArr} = this.state
            //模拟一条新闻
            const news = '新闻' + (newsArr.length + 1)
            //更新状态
            this.setState({newsArr: [news, ...newsArr]})
        }, 1000);
    }

    getSnapshotBeforeUpdate(){
        return this.refs.list.scrollHeight
    }

    componentDidUpdate(preProps, preState, height){
        this.refs.list.scrollTop += this.refs.list.scrollHeight - height
    }

    render(){
        return(
            <div className="list" ref="list">
                {
                    this.state.newsArr.map((n, index) => {
                        return <div key={index} className="news">{n}</div>
                    })
                }
```

```
            </div>
        )
    }
}
ReactDOM.render(<NewsList/>, document.getElementById('test'))
```

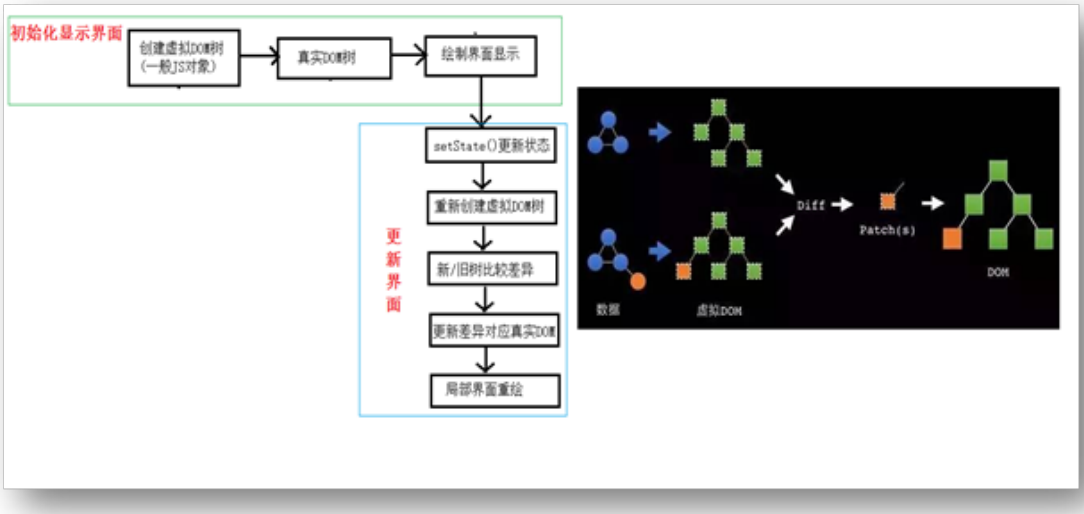
2.8 虚拟DOM 与 DOM Diffing 算法

2.8.1 效果

需求：验证虚拟 DOM Diffing 算法的存在

```
class Time extends React.Component {
    state = {date: new Date()}

    componentDidMount () {
        setInterval(() => {
            this.setState({
                date: new Date()
            })
        }, 1000)
    }

    render () {
        return (
            <div>
                <h1>hello</h1>
                <input type="text"/>
                <span>
                    现在是: {this.state.date.toTimeString()}
                    <input type="text"/>
                </span>
            </div>
        )
    }
}

ReactDOM.render(<Time/>, document.getElementById('test'))
```

2.8.2 基本原理图



2.8.3 key 的作用

经典面试题:

1. react/vue 中的 key 有什么作用？（key 的内部原理是什么？）
2. 为什么遍历列表时，key 最好不要用 index?

---

1. 虚拟**DOM** 中 **key** 的作用：
    1. 简单的说: key 是虚拟DOM对象的标识, 在更新显示时 key 起着极其重要的作用。
    2. 详细的说: 当状态中的数据发生变化时，react 会根据【新数据】生成【新的虚拟DOM】, 随后React进行【新虚拟DOM】与【旧虚拟DOM】的 diff 比较，比较规则如下：
        - 旧虚拟DOM中找到了与新虚拟DOM相同的 key：
            1. 若虚拟DOM中内容没变, 直接使用之前的真实DOM
            2. 若虚拟DOM中内容变了, 则生成新的真实DOM，随后替换掉页面中之前的真实DOM
        - 旧虚拟DOM中未找到与新虚拟DOM相同的 key
            1. 根据数据创建新的真实DOM，随后渲染到到页面
2. 用 **index** 作为 **key** 可能会引发的问题：
    1. 若对数据进行：逆序添加、逆序删除等破坏顺序操作:

- 会产生没有必要的真实DOM更新 ==> 界面效果没问题, 但效率低。

2. 如果结构中还包含输入类的DOM：
   - 会产生错误DOM更新 ==> 界面有问题。
3. 注意！如果不存在对数据的逆序添加、逆序删除等破坏顺序操作，
   - 仅用于渲染列表用于展示，使用 index 作为 key 是没有问题的。

3. 开发中如何选择 **key ?**:
   1. 最好使用每条数据的唯一标识作为 key, 比如 id、手机号、身份证号、学号等唯一值。
   2. 如果确定只是简单的展示数据，用 index 也是可以的。

慢动作回放----使用 **index** 索引值作为 **key**

```
初始数据:
        {id:1,name:'小张',age:18},
        {id:2,name:'小李',age:19},
初始的虚拟DOM:
        <li key=0>小张---18<input type="text"/></li>
        <li key=1>小李---19<input type="text"/></li>

更新后的数据:
        {id:3,name:'小王',age:20},
        {id:1,name:'小张',age:18},
        {id:2,name:'小李',age:19},
更新数据后的虚拟DOM:
        <li key=0>小王---20<input type="text"/></li>
        <li key=1>小张---18<input type="text"/></li>
        <li key=2>小李---19<input type="text"/></li>
```

慢动作回放----使用 **id** 唯一标识作为 **key**

```
初始数据:
        {id:1,name:'小张',age:18},
        {id:2,name:'小李',age:19},
初始的虚拟DOM:
        <li key=1>小张---18<input type="text"/></li>
        <li key=2>小李---19<input type="text"/></li>

更新后的数据:
        {id:3,name:'小王',age:20},
        {id:1,name:'小张',age:18},
```

```
        {id:2,name:'小李',age:19},
更新数据后的虚拟DOM:
        <li key=3>小王---20<input type="text"/></li>
        <li key=1>小张---18<input type="text"/></li>
        <li key=2>小李---19<input type="text"/></li>
```

```
    class Person extends React.Component{

        state = {
            persons:[
                {id: 1, name: '小张', age: 18},
                {id: 2, name: '小李', age: 19},
            ]
        }

        add = ()=>{
            const {persons} = this.state
            const p = {id: persons.length + 1, name: '小王', age: 20}
            this.setState({persons: [p, ...persons]})
        }

        render(){
            return (
                <div>
                    <h2>展示人员信息</h2>
                    <button onClick={this.add}>添加一个小王</button>
                    <h3>使用index（索引值）作为key</h3>
                    <ul>
                        {
                            this.state.persons.map((personObj, index)=>{
                                return <li key={index}>{personObj.name}---{personObj.age}<input type="text"/></li>
                            })
                        }
                    </ul>
                    <hr/>
                    <hr/>
                    <h3>使用id（数据的唯一标识）作为key</h3>
                    <ul>
                        {
                            this.state.persons.map((personObj)=>{
```

```
35                                  return <li key={personObj.id}>
{personObj.name}---{personObj.age}<input type="text"/></li>
36                              })
37                          }
38                      </ul>
39                  </div>
40              )
41          }
42      }
43
44      ReactDOM.render(<Person/>, document.getElementById('test'))
```

第三章 React 应用(基于 React 脚手架)

3.1 使用 create-react-app 创建 react 应用

3.1.1 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…）
    2. 下载好了所有相关的依赖
    3. 可以直接运行一个简单效果
2. react 提供了一个用于创建 react 项目的脚手架库: `create-react-app`
3. 项目的整体技术架构为: react + webpack + es6 + eslint
4. 使用脚手架开发的项目的特点: 模块化, 组件化, 工程化

3.1.2 创建项目并启动

- 第一步，全局安装：`npm i -g create-react-app`
- 第二步，切换到想创项目的目录，使用命令：`create-react-app hello-react`
- 第三步，进入项目文件夹：`cd hello-react`

- 第四步，启动项目：`npm start`

3.1.3 react 脚手架项目结构

```
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)
```

3.1.4 功能界面的组件化编码流程（通用）

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
    1. 动态显示初始化数据
        1. 数据类型
        2. 数据名称
        3. 保存在哪个组件?
    2. 交互(从绑定事件监听开始)

3.2 组件的组合使用-TodoList

功能: 组件化实现此功能

1. 显示所有 todo 列表
2. 输入文本, 点击按钮显示到列表的首位, 并清除输入的文本

todoList 案例相关知识点

1. 拆分组件、实现静态组件，注意：className、style 的写法
2. 动态初始化列表，如何确定将数据放在哪个组件的 state 中？
    - 某个组件使用：放在其自身的 state 中
    - 某些组件使用：放在他们共同的父组件 state 中（官方称此操作为：状态提升）
3. 关于父子之间通信：
    1. 【父组件】给【子组件】传递数据：通过 props 传递
    2. 【子组件】给【父组件】传递数据：通过 props 传递，要求父提前给子传递一个函数
4. 注意 `defaultChecked` 和 `checked` 的区别，类似的还有：`defaultValue` 和 `value`
5. 状态在哪里，操作状态的方法就在哪里

第四章 React ajax

4.1 理解

4.1.1 前置说明

1. React 本身只关注于界面, 并不包含发送 `ajax` 请求的代码
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)

4.1.2 常用的ajax请求库

1. `jQuery`: 比较重, 如果需要另外引入不建议使用
2. `axios`: 轻量级, 建议使用
    1. 封装 `XmlHttpRequest` 对象的 `ajax`
    2. `promise` 风格
    3. 可以用在浏览器端和 node 服务器端

4.2 axios

4.2.1 文档

https://github.com/axios/axios

4.2.2 相关API

1. GET 请求

```
axios.get('/user?ID=12345')
.then(function (response) {
    console.log(response.data);
})
.catch(function (error) {
    console.log(error);
});

axios.get('/user', {
    params: {ID: 12345}
})
.then(function (response) {
    console.log(response);
})
.catch(function (error) {
    console.log(error);
});
```

1. POST 请求

```
axios.post('/user', {
    firstName: 'Fred',
    lastName: 'Flintstone'
})
.then(function (response) {
    console.log(response);
})
.catch(function (error) {
    console.log(error);
});
```

4.3 案例—github用户搜索

4.3.1 效果

请求地址: https://api.github.com/search/users?q=xxxxxx

github 搜索案例相关知识点

1. 设计状态时要考虑全面，例如带有网络请求的组件，要考虑请求失败怎么办。
2. ES6 小知识点：解构赋值+重命名

```
 let obj = {a: {b: 1}}
const {a} = obj; //传统解构赋值
const {a: {b}} = obj; //连续解构赋值
const {a: {b: value}} = obj; //连续解构赋值+重命名
```

1. 消息订阅与发布机制
    1. 先订阅，再发布（理解：有一种隔空对话的感觉）
    2. 适用于任意组件间通信
    3. 要在组件的 `componentWillUnmoun` t中取消订阅
2. fetch 发送请求（关注分离的设计思想）

```
try {
    const response= await fetch(`/api1/search/users2?q=${keyWord}`)
    const data = await response.json()
    console.log(data);
} catch (error) {
    console.log('请求出错', error);
}
```

4.4 消息订阅-发布机制

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

4.5 扩展： Fetch

4.5.1 文档

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

4.5.2 特点

1. `fetch`: 原生函数，不再使用 `XmlHttpRequest` 对象提交 ajax 请求
2. 老版本浏览器可能不支持

4.5.3 相关API

1. GET 请求

```
fetch(url).then(function (response) {
    return response.json()
}).then(function (data) {
    console.log(data)
}).catch(function (e) {
    console.log(e)
});
```

1. POST 请求

```
fetch(url, {
    method: "POST",
    body: JSON.stringify(data)
}).then(function (data) {
    console.log(data)
}).catch(function (e) {
    console.log(e)
})
```

第五章 React路由

5.1 相关理解

5.1.1 SPA 的理解

1. 单页 Web 应用（single page web application，SPA）。
2. 整个应用只有 一个完整的页面 。
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, 并在前端异步展现。

5.1.2 路由的理解

1. 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

2. 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: `<Route path="/test" component={Test}>`
        3. 工作过程：当浏览器的 path 变为 /test 时, 当前路由组件就会变为 Test 组件

5.1.3 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。

5.2 react-router-dom 相关API

5.2.1 内置组件

1. `<BrowserRouter>`
2. `<HashRouter>`
3. `<Route>`
4. `<Redirect>`
5. `<Link>`
6. `<NavLink>`

7. `<Switch>`

5.2.2 其它

1. `history` 对象
2. `match` 对象
3. `withRouter` 函数

5.3 基本路由使用

5.3.1 效果

5.3.2 准备

1. 下载 `react-router-dom`：`npm install --save react-router-dom`
2. 引入 `bootstrap.css`：`<link rel="stylesheet" href="/css/bootstrap.css">`

---

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="/xxxxx">Demo</Link>
```

1. 展示区写 `Route 标签` 进行路径的匹配

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

1. `<App>` 的最外侧包裹了一个 `<BrowserRouter>` 或 `<HashRouter>`

5.4 路由组件与一般组件

1. 写法不同：

- 一般组件：`<Demo/>`
- 路由组件：`<Route path="/demo" component={Demo}/>`

2. 存放位置不同：
   - 一般组件：components
   - 路由组件：pages
3. 接收到的 props 不同：
   - 一般组件：写组件标签时传递了什么，就能收到什么
   - 路由组件：接收到三个固定的属性

```
history:
   go: ƒ go(n)
   goBack: ƒ goBack()
   goForward: ƒ goForward()
   push: ƒ push(path, state)
   replace: ƒ replace(path, state)

location:
   pathname: "/about"
   search: ""
   state: undefined

match:
   params: {}
   path: "/about"
   url: "/about"
```

5.5 NavLink 与封装 NavLink

1. `NavLink` 可以实现路由链接的高亮，通过 `activeClassName` 指定样式名
2. 标签体内容是一个特殊的标签属性
3. 通过 `this.props.children` 可以获取标签体内容

```
<NavLink activeClassName="atguigu" className="list-group-item"
to="/about">About</NavLink>
<NavLink activeClassName="atguigu" className="list-group-item"
to="/home">Home</NavLink>
```

封装 NavLink

```
import React, { Component } from 'react'
import {NavLink} from 'react-router-dom'

export default class MyNavLink extends Component {
    render() {
        return (
            <NavLink activeClassName="atguigu" className="list-group-item" {...this.props}/>
        )
    }
}
```

```
import MyNavLink from './components/MyNavLink'

<MyNavLink to="/about">About</MyNavLink>
<MyNavLink to="/home">Home</MyNavLink>
```

5.6 Switch 的使用

1. 通常情况下，path 和 component 是一一对应的关系。
2. Switch 可以提高路由匹配效率(单一匹配)。

```
<Switch>
    <Route path="/about" component={About}/>
    <Route path="/home" component={Home}/>
    <Route path="/home" component={Test}/>
</Switch>
```

5.7 解决多级路径刷新页面样式丢失的问题

1. public/index.html 中 引入样式时不写 `./` 写 `/` （常用）

```
<link rel="stylesheet" href="/css/bootstrap.css">
```

1. public/index.html 中 引入样式时不写 `./` 写 `%PUBLIC_URL%` （常用）

```
<link rel="stylesheet" href="%PUBLIC_URL%/css/bootstrap.css">
```

1. 使用 `HashRouter`

```
ReactDOM.render(
    <HashRouter>
        <App/>
    </HashRouter>,
    document.getElementById('root')
)
```

5.8 路由的严格匹配与模糊匹配

1. 默认使用的是模糊匹配（简单记：【输入的路径】必须包含要【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，需要再开，有些时候开启会导致无法继续匹配二级路由

5.9 Redirect 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. 具体编码：

```
<Switch>
    <Route path="/about" component={About}/>
    <Route path="/home" component={Home}/>
    <Redirect to="/about"/>
</Switch>
```

5.10 嵌套路由使用

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的

```
import React, { Component } from 'react'
import MyNavLink from '../../components/MyNavLink'
import {Route,Switch,Redirect} from 'react-router-dom'
import News from './News'
import Message from './Message'

export default class Home extends Component {
    render() {
        return (
            <div>
                <h3>我是Home的内容</h3>
                <div>
                    <ul className="nav nav-tabs">
                        <li>
                            <MyNavLink
to="/home/news">News</MyNavLink>
                        </li>
                        <li>
                            <MyNavLink
to="/home/message">Message</MyNavLink>
                        </li>
                    </ul>
                    {/* 注册路由 */}
                    <Switch>
                        <Route path="/home/news" component=
{News}/>
                        <Route path="/home/message" component=
{Message}/>
                        <Redirect to="/home/news"/>
                    </Switch>
                </div>
            </div>
        )
    }
}
```

5.11 向路由组件传递参数数据

1. **params 参数**
   - 路由链接(携带参数)：`<Link to='/demo/test/tom/18'}>详情</Link>`
   - 注册路由(声明接收)：`<Route path="/demo/test/:name/:age" component={Test}/>`
   - 接收参数：`this.props.match.params`

```
import React, { Component } from 'react'
import {Link,Route} from 'react-router-dom'
import Detail from './Detail'

export default class Message extends Component {
    state = {
        messageArr:[
            {id: '01', title: '消息1'},
            {id: '02', title: '消息2'},
            {id: '03', title: '消息3'}
        ]
    }
    render() {
        const {messageArr} = this.state
        return (
            <div>
                <ul>
                    {
                        messageArr.map((msgObj)=>{
                            return (
                                <li key={msgObj.id}>
                                    {/* 向路由组件传递params参数 */}
                                    <Link to=
{`/home/message/detail/${msgObj.id}/${msgObj.title}`}>
{msgObj.title}</Link>
                                </li>
                            )
                        })
                    }
                </ul>
                <hr/>
                {/* 声明接收params参数 */}
                <Route path="/home/message/detail/:id/:title"
component={Detail}/>
            </div>
        )
    }
```

```
}
```

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

const DetailData = [
    {id: '01', content: '你好，中国'},
    {id: '02', content: '你好，尚硅谷'},
    {id: '03', content: '你好，未来的自己'}
]
export default class Detail extends Component {
    render() {
        console.log(this.props);
        // 接收params参数
        const {id, title} = this.props.match.params
        const findResult = DetailData.find((detailObj) => {
            return detailObj.id === id
        })
        return (
            <ul>
                <li>ID:{id}</li>
                <li>TITLE:{title}</li>
                <li>CONTENT:{findResult.content}</li>
            </ul>
        )
    }
}
```

1. **search 参数**
   - 路由链接(携带参数)：`<Link to='/demo/test?name=tom&age=18'}>详情</Link>`
   - 注册路由(无需声明，正常注册即可)：`<Route path="/demo/test" component={Test}/>`
   - 接收参数：`this.props.location.search`
   - 备注：获取到的 search 是 `urlencoded` 编码字符串，需要借助 `querystring` 解析

```
import React, { Component } from 'react'
import {Link, Route} from 'react-router-dom'
import Detail from './Detail'

export default class Message extends Component {
    state = {
```

```
        messageArr:[
            {id: '01', title:'消息1'},
            {id: '02', title:'消息2'},
            {id: '03', title:'消息3'}
        ]
    }
    render() {
        const {messageArr} = this.state
        return (
            <div>
                <ul>
                    {
                        messageArr.map((msgObj)⇒{
                            return (
                                <li key={msgObj.id}>
                                    {/* 向路由组件传递search参数 */}
                                    <Link to=
{`/home/message/detail/?id=${msgObj.id}&title=${msgObj.title}`}>
{msgObj.title}</Link>
                                </li>
                            )
                        })
                    }
                </ul>
                <hr/>
                {/* search参数无需声明接收，正常注册路由即可 */}
                <Route path="/home/message/detail" component=
{Detail}/>
            </div>
        )
    }
}
```

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

const DetailData = [
    {id: '01', content: '你好，中国'},
    {id: '02', content: '你好，尚硅谷'},
    {id: '03', content: '你好，未来的自己'}
]
export default class Detail extends Component {
    render() {
        // 接收search参数
        const {search} = this.props.location
```

```
        const {id, title} = qs.parse(search.slice(1))

        const findResult = DetailData.find((detailObj)=>{
            return detailObj.id === id
        })
        return (
            <ul>
                <li>ID: {id}</li>
                <li>TITLE: {title}</li>
                <li>CONTENT: {findResult.content}</li>
            </ul>
        )
    }
}
```

1. **state 参数**
   - 路由链接(携带参数)：`<Link to={{pathname:'/demo/test',state:{name:'tom',age:18}}}>详情</Link>`
   - 注册路由(无需声明，正常注册即可)：`<Route path="/demo/test" component={Test}/>`
   - 接收参数：`this.props.location.state`
   - 备注：刷新也可以保留住参数

```
import React, { Component } from 'react'
import {Link, Route} from 'react-router-dom'
import Detail from './Detail'

export default class Message extends Component {
    state = {
        messageArr:[
            {id: '01', title: '消息1'},
            {id: '02', title: '消息2'},
            {id: '03', title: '消息3'}
        ]
    }
    render() {
        const {messageArr} = this.state
        return (
            <div>
                <ul>
                    {
                        messageArr.map((msgObj)=>{
```

```
                                return (
                                    <li key={msgObj.id}>
                                        {/* 向路由组件传递state参数 */}
                                        <Link to={{pathname: '/home/message/detail', state: {id: msgObj.id, title: msgObj.title}}}>{msgObj.title}</Link>
                                    </li>
                                )
                            })
                        }
                    </ul>
                    <hr/>
                    {/* state参数无需声明接收，正常注册路由即可 */}
                    <Route path="/home/message/detail" component={Detail}/>
                </div>
            )
        }
}
```

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

const DetailData = [
    {id: '01', content: '你好，中国'},
    {id: '02', content: '你好，尚硅谷'},
    {id: '03', content: '你好，未来的自己'}
]
export default class Detail extends Component {
    render() {
        // 接收state参数
        const {id, title} = this.props.location.state || {}

        const findResult = DetailData.find((detailObj)=>{
            return detailObj.id === id
        }) || {}
        return (
            <ul>
                <li>ID: {id}</li>
                <li>TITLE: {title}</li>
                <li>CONTENT: {findResult.content}</li>
            </ul>
        )
    }
}
```

5.12 编程式路由导航

借助 `this.prosp.history` 对象上的 API 对操作路由跳转、前进、后退

- `this.prosp.history.push()`
- `this.prosp.history.replace()`
- `this.prosp.history.goBack()`
- `this.prosp.history.goForward()`
- `this.prosp.history.go()`

```
import React, { Component } from 'react'
import {Link, Route} from 'react-router-dom'
import Detail from './Detail'

export default class Message extends Component {
    state = {
        messageArr:[
            {id: '01', title: '消息1'},
            {id: '02', title: '消息2'},
            {id: '03', title: '消息3'},
        ]
    }

    replaceShow = (id, title) => {
        //replace跳转+携带params参数

//this.props.history.replace(`/home/message/detail/${id}/${title}
`)

        //replace跳转+携带search参数
        // this.props.history.replace(`/home/message/detail?
id=${id}&title=${title}`)

        //replace跳转+携带state参数
        this.props.history.replace(`/home/message/detail`, {id,
title})
    }

    pushShow = (id, title) => {
        //push跳转+携带params参数
```

```
        //
this.props.history.push(`/home/message/detail/${id}/${title}`)

        //push跳转+携带search参数
        // this.props.history.push(`/home/message/detail?
id=${id}&title=${title}`)

        //push跳转+携带state参数
        this.props.history.push(`/home/message/detail`, {id,
title})

    }

    back = () => {
        this.props.history.goBack()
    }

    forward = () => {
        this.props.history.goForward()
    }

    go = () => {
        this.props.history.go(-2)
    }

    render() {
        const {messageArr} = this.state
        return (
            <div>
                <ul>
                    {
                        messageArr.map((msgObj) => {
                            return (
                                <li key={msgObj.id}>
                                    {/* 向路由组件传递state参数 */}
                                    <Link to={{pathname:
'/home/message/detail', state: {id: msgObj.id, title:
msgObj.title}}}>{msgObj.title}</Link>

                                     <button onClick={() =>
this.pushShow(msgObj.id, msgObj.title)}>push查看</button>
                                     <button onClick={() =>
this.replaceShow(msgObj.id, msgObj.title)}>replace查看</button>
                                </li>
                            )
```

```
                        })
                    }
                </ul>
                <hr/>
                {/* state参数无需声明接收，正常注册路由即可 */}
                <Route path="/home/message/detail" component={Detail}/>

                <button onClick={this.back}>回退</button> 
                <button onClick={this.forward}>前进</button> 
                <button onClick={this.go}>go</button>
            </div>
        )
    }
}
```

5.13 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
   1. BrowserRouter 没有任何影响，因为 state 保存在 history 对象中 。
   2. HashRouter 刷新后会导致路由 state 参数的丢失！！！
4. 备注：HashRouter 可以用于解决一些路径错误相关的问题。

5.14 withRouter 的使用

1. withRouter 可以加工一般组件，让一般组件具有路由组件特有的 API

2 withRouter 的返回值是一个新组件

```
import React, { Component } from 'react'
import {withRouter} from 'react-router-dom'

class Header extends Component {

    back = ()=>{
        this.props.history.goBack()
    }

    forward = ()=>{
        this.props.history.goForward()
    }

    go = ()=>{
        this.props.history.go(-2)
    }

    render() {
        console.log('Header组件收到的props是',this.props);
        return (
            <div className="page-header">
                <h2>React Router Demo</h2>
                <button onClick={this.back}>回退</button> 
                <button onClick={this.forward}>前进</button> 
                <button onClick={this.go}>go</button>
            </div>
        )
    }
}

export default withRouter(Header)
```

第六章 React UI 组件库

6.1 流行的开源 React UI组件库

6.1.1 material-ui(国外)

1. 官网: http://www.material-ui.com/#/
2. github: https://github.com/callemall/material-ui

6.1.2 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/

6.1.3 antd 的按需引入+自定主题

1. 安装依赖: `yarn add react-app-rewired customize-cra babel-plugin-import less less-loader`
2. 修改 package.json

```
"scripts": {
    "start": "react-app-rewired start",
    "build": "react-app-rewired build",
    "test": "react-app-rewired test",
    "eject": "react-scripts eject"
},
```

3. 根目录下创建 config-overrides.js

```
//配置具体的修改规则
const { override, fixBabelImports,addLessLoader} =
require('customize-cra');
module.exports = override(
    fixBabelImports('import', {
        libraryName: 'antd',
        libraryDirectory: 'es',
        style: true,
    }),
    addLessLoader({
        lessOptions:{
            javascriptEnabled: true,
            modifyVars: { '@primary-color': 'green' },
        }
    }),
);
```

1. 备注：不用在组件里亲自引入样式了，即：`import 'antd/dist/antd.css'` 应该删掉

第七章 redux

7.1 redux 理解

7.1.1 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

7.1.2 redux 是什么

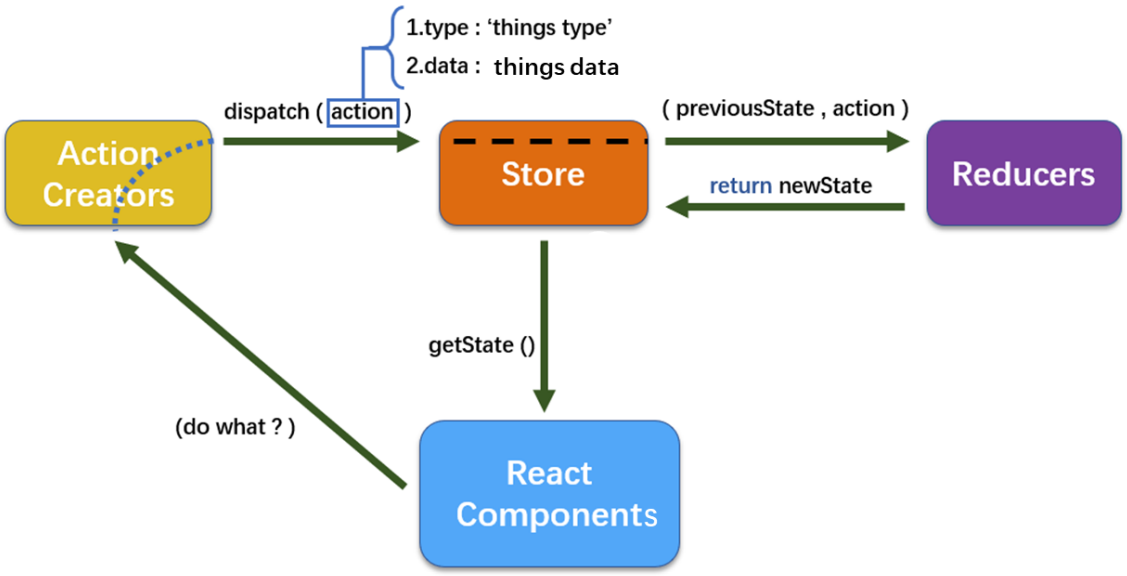
1. redux 是一个专门用于做 状态管理 的JS库(不是 react 插件库)。
2. 它可以用在 react, angular, vue 等项目中, 但基本与 react 配合使用。
3. 作用: 集中式管理 react 应用中多个组件 共享 的状态。

7.1.3 什么情况下需要使用 redux

1. 某个组件的状态，需要让其他组件可以随时拿到（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：能不用就不用, 如果不用比较吃力才考虑使用。

7.1.4 redux 工作流程

redux 原理图



7.2 redux 的三个核心概念

7.2.1 action

1. 动作的对象
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, 值类型任意, 可选属性
3. 例子：`{ type: 'ADD_STUDENT', data: {name: 'tom', age: 18} }`

```
/*
    该文件专门为Count组件生成action对象
*/

export const createIncrementAction = data => ({type:'increment', data})
export const createDecrementAction = data => ({type:'decrement', data})
```

7.2.2 reducer

1. `用于初始化状态、加工状态` 。
2. 加工时，根据旧的 state 和 action， 产生新的 state 的纯函数。

```
/*
    1.该文件是用于创建一个为Count组件服务的reducer，reducer的本质就是一个函数
    2.reducer函数会接到两个参数，分别为：之前的状态(preState)，动作对象(action)
*/

const initState = 0 //初始化状态
export default function countReducer(preState=initState, action){
    // console.log(preState);
    //从action对象中获取：type、data
    const {type, data} = action
    //根据type决定如何加工数据
    switch (type) {
        case 'increment': //如果是加
            return preState + data
        case 'decrement': //若果是减
            return preState - data
        default:
            return preState
    }
}
```

7.2.3 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

```
/*
    该文件专门用于暴露一个store对象，整个应用只有一个store对象
*/

//引入createStore，专门用于创建redux中最为核心的store对象
import {createStore} from 'redux'
//引入为Count组件服务的reducer
import countReducer from './count_reducer'
//暴露store
export default createStore(countReducer)
```

7.3 redux 的核心 API

7.3.1 createstore()

作用：创建包含指定 reducer 的 store 对象

7.3.2 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)`

7.3.3 applyMiddleware()

作用：应用上基于 redux 的中间件(插件库)

7.3.4 combineReducers()

作用：合并多个 `reducer` 函数

7.4 使用 redux 编写应用

7.5 redux 异步编程

7.5.1 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)

7.5.2 使用异步中间件

```
npm install --save redux-thunk
```

7.6 react-redux

7.6.1 理解

1. 一个 react 插件库
2. 专门用来简化 react 应用中使用 redux

7.6.2 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

7.6.3 相关 API

1. `Provider`：让所有组件都可以得到 state 数据

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

1. `connect`：用于包装 UI 组件生成容器组件

```
import { connect } from 'react-redux'

connect(
    mapStateToprops,
    mapDispatchToProps
)(Counter)
```

1. `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
    }
}
```

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

7.7 求和案例

1.求和案例_redux 精简版

1. 去除 Count组件 自身的状态
2. src下建立:
    - redux
        - store.js
        - count_reducer.js
3. store.js：
    1. 引入redux中的createStore函数，创建一个store
    2. createStore调用时要传入一个为其服务的reducer
    3. 记得暴露store对象
4. count_reducer.js：
    1. reducer 的本质是一个函数，接收：preState,action，返回加工后的状态
    2. reducer 有两个作用：初始化状态，加工状态
    3. reducer 被第一次调用时，是 store 自动触发的，
        - 传递的 preState 是 undefined,
        - 传递的 action 是: `{type: '@@REDUX/INIT_a.2.b.4}`
5. 在 index.js 中监测 store 中状态的改变，一旦发生改变重新渲染`<App/>`
    - 备注：redux 只负责管理状态，至于状态的改变驱动着页面的展示，要靠我们自己写。

2.求和案例_redux 完整版

新增文件：

1. count_action.js 专门用于创建 action 对象
2. constant.js 放置容易写错的 type 值

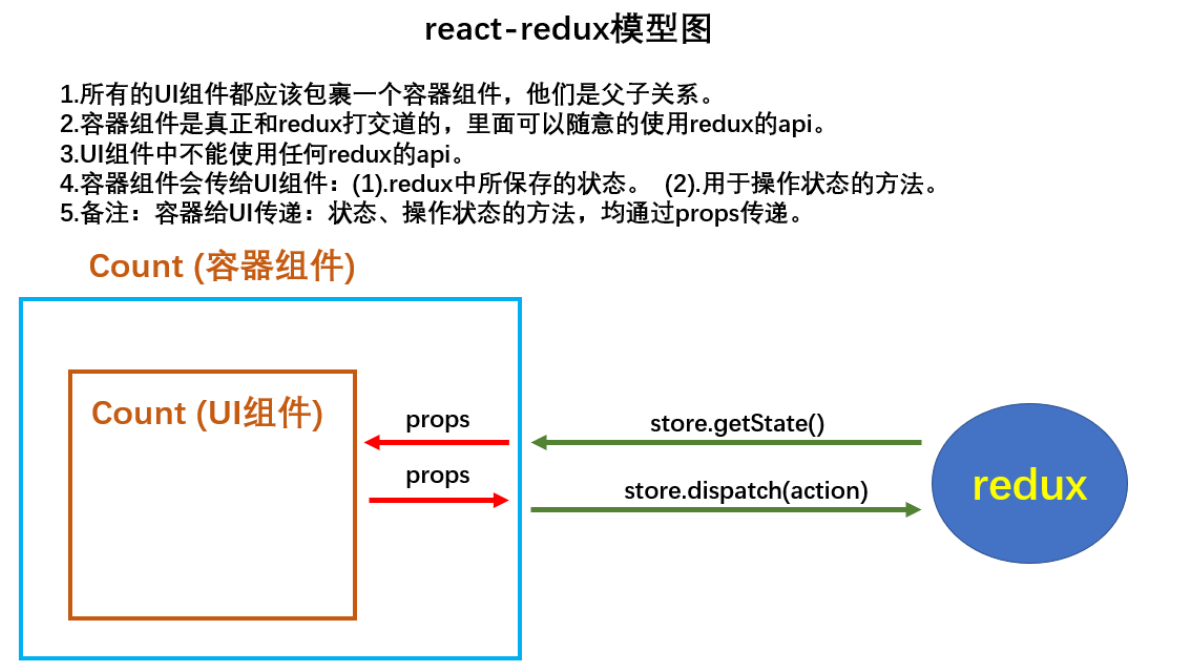
3.求和案例_redux 异步 action 版

1. 明确：延迟的动作不想交给组件自身，想交给 action
2. 何时需要异步 action：想要对状态进行操作，但是具体的数据靠异步任务返回。
3. 具体编码：
    1. `yarn add redux-thunk`，并配置在 store 中
    2. 创建 action 的函数不再返回一般对象，而是一个函数，该函数中写异步任务。
    3. 异步任务有结果后，分发一个同步的 action 去真正操作数据。
4. 备注：异步 action 不是必须要写的，完全可以自己等待异步任务的结果了再去分发同步 action。

```
/*
    该文件专门为Count组件生成action对象
*/
import {INCREMENT,DECREMENT} from './constant'

//同步action，就是指action的值为Object类型的一般对象
export const createIncrementAction = data => ({type:INCREMENT,data})
export const createDecrementAction = data => ({type:DECREMENT,data})

//异步action，就是指action的值为函数,异步action中一般都会调用同步action，异步action不是必须要用的。
export const createIncrementAsyncAction = (data, time) => {
    return (dispatch) => {
        setTimeout(() => {
            dispatch(createIncrementAction(data))
        }, time)
    }
}
```

```
/*
    该文件专门用于暴露一个store对象，整个应用只有一个store对象
*/

//引入createStore，专门用于创建redux中最为核心的store对象
import {createStore, applyMiddleware} from 'redux'
//引入为Count组件服务的reducer
import countReducer from './count_reducer'
//引入redux-thunk，用于支持异步action
import thunk from 'redux-thunk'
//暴露 store
export default createStore(countReducer,applyMiddleware(thunk))
```

4.求和案例_react-redux 基本使用



1. 明确两个概念：
    1. UI组件:不能使用任何 redux 的 api，只负责页面的呈现、交互等。
    2. 容器组件：负责和 redux 通信，将结果交给UI组件。
2. 如何创建一个容器组件————靠 react-redux 的 `connect` 函数
    - `connect(mapStateToProps,mapDispatchToProps)`(UI组件)
        - `mapStateToProps` :映射状态，返回值是一个对象
        - `mapDispatchToProps` :映射操作状态的方法，返回值是一个对象

3. 备注1：容器组件中的 store 是靠 props 传进去的，而不是在容器组件中直接引入

4. 备注2：`mapDispatchToProps` 也可以是一个对象

```
//引入Count的UI组件
import CountUI from '../../components/Count'
//引入action
import {
    createIncrementAction,
    createDecrementAction,
    createIncrementAsyncAction
} from '../../redux/count_action'

//引入connect用于连接UI组件与redux
import {connect} from 'react-redux'

/*
    1.mapStateToProps函数返回的是一个对象;
    2.返回的对象中的key就作为传递给UI组件props的key,value就作为传递给UI组件props的value
    3.mapStateToProps用于传递状态
*/
function mapStateToProps(state){
    return {count:state}
}

/*
    1.mapDispatchToProps函数返回的是一个对象;
    2.返回的对象中的key就作为传递给UI组件props的key,value就作为传递给UI组件props的value
    3.mapDispatchToProps用于传递操作状态的方法
*/
function mapDispatchToProps(dispatch){
    return {
        jia:number => dispatch(createIncrementAction(number)),
        jian:number => dispatch(createDecrementAction(number)),
        jiaAsync:(number,time) => dispatch(createIncrementAsyncAction(number,time)),
    }
}

//使用connect()()创建并暴露一个Count的容器组件
export default connect(mapStateToProps,mapDispatchToProps)(CountUI)
```

5.求和案例_react-redux 优化

1. 容器组件 和 UI组件 整合一个文件
2. 无需自己给容器组件传递 store，给 `<App/>` 包裹一个 `<Provider store={store}>` 即可。

```
import React from 'react'
import ReactDOM from 'react-dom'
import App from './App'
import store from './redux/store'
import {Provider} from 'react-redux'

ReactDOM.render(
    <Provider store={store}>
        <App/>
    </Provider>,
    document.getElementById('root')
)
```

1. 使用了 react-redux 后也不用再自己检测 redux 中状态的改变了，容器组件可以自动完成这个工作。
2. `mapDispatchToProps` 也可以简单的写成一个对象
3. 一个组件要和 redux “打交道”要经过哪几步？

    1. 定义好UI组件---不暴露
    2. 引入 connect 生成一个容器组件，并暴露，写法如下：

```
connect(
    state => ({key:value}), //映射状态
    {key:xxxxxAction} //映射操作状态的方法
)(UI组件)
```

    1. 在UI组件中通过 `this.props.xxxxxxx` 读取和操作状态

```
import React, { Component } from 'react'
//引入action
import {
    createIncrementAction,
    createDecrementAction,
    createIncrementAsyncAction
} from '../../redux/count_action'
```

```jsx
//引入connect用于连接UI组件与redux
import {connect} from 'react-redux'

//定义UI组件
class Count extends Component {

    state = {carName: '奔驰c63'}

    //加法
    increment = ()=>{
        const {value} = this.selectNumber
        this.props.jia(value*1)
    }
    //减法
    decrement = ()=>{
        const {value} = this.selectNumber
        this.props.jian(value*1)
    }
    //奇数再加
    incrementIfOdd = ()=>{
        const {value} = this.selectNumber
        if(this.props.count % 2 !== 0){
            this.props.jia(value*1)
        }
    }
    //异步加
    incrementAsync = ()=>{
        const {value} = this.selectNumber
        this.props.jiaAsync(value*1,500)
    }

    render() {
        //console.log('UI组件接收到的props是',this.props);
        return (
            <div>
                <h1>当前求和为: {this.props.count}</h1>
                <select ref={c => this.selectNumber = c}>
                    <option value="1">1</option>
                    <option value="2">2</option>
                    <option value="3">3</option>
                </select> 
                <button onClick={this.increment}>+</button> 
                <button onClick={this.decrement}>-</button> 
                <button onClick={this.incrementIfOdd}>当前求和为奇数再加</button> 
```

```
                <button onClick={this.incrementAsync}>异步加
</button> 
            </div>
        )
    }
}

//使用connect()()创建并暴露一个Count的容器组件
export default connect(
    state => ({count:state}),

    //mapDispatchToProps的一般写法
    /* dispatch => ({
        jia:number => dispatch(createIncrementAction(number)),
        jian:number => dispatch(createDecrementAction(number)),
        jiaAsync:(number,time) =>
dispatch(createIncrementAsyncAction(number,time)),
    }) */

    //mapDispatchToProps的简写
    {
        jia:createIncrementAction,
        jian:createDecrementAction,
        jiaAsync:createIncrementAsyncAction,
    }
)(Count)
```

6.求和案例_react-redux 数据共享版

1. 定义一个 Pserson组件，和 Count组件 通过 redux 共享数据。
2. 为 Person组件 编写：reducer、action，配置 constant 常量。
3. 重点：Person 的 reducer 和 Count 的 Reducer 要使用 `combineReducers` 进行合并，合并后的总状态是一个对象！！！
4. 交给 store 的是总 reducer，最后注意在组件中取出状态的时候，记得“取到位”。

```
import React, { Component } from 'react'
import {nanoid} from 'nanoid'
import {connect} from 'react-redux'
import {createAddPersonAction} from '../../redux/actions/person'

class Person extends Component {
```

```
7
8       addPerson = ()=>{
9           const name = this.nameNode.value
10          const age = this.ageNode.value
11          const personObj = {id:nanoid(),name,age}
12          this.props.jiaYiRen(personObj)
13          this.nameNode.value = ''
14          this.ageNode.value = ''
15      }
16
17      render() {
18          return (
19              <div>
20                  <h2>我是Person组件,上方组件求和为{this.props.he}</h2>
21                  <input ref={c=>this.nameNode = c} type="text"
    placeholder="输入名字"/>
22                  <input ref={c=>this.ageNode = c} type="text"
    placeholder="输入年龄"/>
23                  <button onClick={this.addPerson}>添加</button>
24                  <ul>
25                      {
26                          this.props.yiduiren.map((p)=>{
27                              return <li key={p.id}>{p.name}--
    {p.age}</li>
28                          })
29                      }
30                  </ul>
31              </div>
32          )
33      }
34  }
35
36  export default connect(
37      state => ({yiduiren:state.rens,he:state.he}),//映射状态
38      {jiaYiRen:createAddPersonAction}//映射操作状态的方法
39  )(Person)
```

```
1   /*
2       该文件专门用于暴露一个store对象，整个应用只有一个store对象
3   */
4
5   //引入createStore，专门用于创建redux中最为核心的store对象
6   import {createStore,applyMiddleware,combineReducers} from 'redux'
7   //引入为Count组件服务的reducer
8   import countReducer from './reducers/count'
```

```
//引入为Count组件服务的reducer
import personReducer from './reducers/person'
//引入redux-thunk，用于支持异步action
import thunk from 'redux-thunk'

//汇总所有的reducer变为一个总的reducer
const allReducer = combineReducers({
    he:countReducer,
    rens:personReducer
})

//暴露store
export default createStore(allReducer,applyMiddleware(thunk))
```

7.求和案例_react-redux 开发者工具的使用

1. `yarn add redux-devtools-extension`
2. store 中进行配置

```
import {composeWithDevTools} from 'redux-devtools-extension'

const store =
createStore(allReducer,composeWithDevTools(applyMiddleware(thunk))
)
```

```
/*
    该文件专门用于暴露一个store对象，整个应用只有一个store对象
*/

//引入createStore，专门用于创建redux中最为核心的store对象
import {createStore,applyMiddleware,combineReducers} from 'redux'
//引入为Count组件服务的reducer
import countReducer from './reducers/count'
//引入为Count组件服务的reducer
import personReducer from './reducers/person'
//引入redux-thunk，用于支持异步action
import thunk from 'redux-thunk'
//引入redux-devtools-extension
import {composeWithDevTools} from 'redux-devtools-extension'

//汇总所有的reducer变为一个总的reducer
const allReducer = combineReducers({
```

```
    he:countReducer,
    rens:personReducer
})

//暴露store
export default
createStore(allReducer,composeWithDevTools(applyMiddleware(thunk)
))
```

8.求和案例_react-redux 最终版

1. 所有变量名字要规范，尽量触发对象的简写形式。
2. reducers 文件夹中，编写 index.js 专门用于汇总并暴露所有的 reducer

```
/*
    该文件用于汇总所有的reducer为一个总的reducer
*/
//引入combineReducers，用于汇总多个reducer
import {combineReducers} from 'redux'
//引入为Count组件服务的reducer
import count from './count'
//引入为Person组件服务的reducer
import persons from './person'

//汇总所有的reducer变为一个总的reducer
export default  combineReducers({
    count,
    persons
})
```

7.8 使用 redux 调试工具

7.8.1 安装 chrome 浏览器插件

浏览器安装 redux devtools 插件

7.8.2 下载工具依赖包

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

7.9 纯函数和高阶函数

7.9.1 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

7.9.2 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

项目打包运行

使用 serve 包 —— `npm i serve -g`

- 以当前文件夹作为服务器主目录，直接运行 `serve`
- 以当前文件夹下的a文件夹作为服务器主目录，运行 `serve a`

打包项目 `yarn build` 或 `npm run build`

部署项目 `serve build`