课堂目标

- 安装`create-react-app`脚手架
- 熟练React基础语法
- 掌握JSX语法
- 掌握setState
- 掌握React生命周期
- 掌握props传递参数
- 掌握React组件通信

资源

- react
- create-react-app

起步

上手

- `npm i -g create-react-app` 安装官方脚手架
- `react-react-app 01_react` 初始化
- react的api比较少,基本学习一次，就再也不用再看文档，它的核心是JS

文件结构

```
1
```

React&ReactDOM

React只做逻辑层,reactDOM去渲染真实的DOM

删除src下面所有代码,新建index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App'
ReactDOM.render(<App />,document.querySelector('#root'))
```

新建 App.js

```
import React from 'react'
class App extends React.Component {
    render() {
        return (
            <div>
               hello,小马哥
            </div>
        )
    }
}
export default App;
```

以上代码感觉都没有问题,但是发现了一个很有趣的标签语法,它既不是字符串也不是HTML

它被称为JSX,是一个JavaScript的语法扩展.我们建议在 React 中配合使用 JSX，JSX 可以很好地描述 UI 应该呈现出它应有交互的本质形式。JSX 可能会使人联想到模版语言，但它具有 JavaScript 的全部功能。

JSX

在JavaScript中直接写的标签,是一个JSX(JS+XML,由于HTML也是XML的一种,可以认为JS+HTML)元素,也是一个react元素,这个元素实际是一个对象,就是**虚拟DOM元素**

React 认为渲染逻辑本质上与其他 UI 逻辑内在耦合.比如，在 UI 中需要绑定处理事件、在某些时刻状态发生变化时需要通知到 UI，以及需要在 UI 中展示准备好的数据。

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

import logo from './favicon.ico'
const ele = <h2>hello,world</h2>
function formatName(user) {
    // 也可以是个表达式
    return user.firstName + '' + user.lastName;
}
const user = {
    firstName: '张',
    lastName: '三丰'
}

//jsx也可以是表达式
function getGreeting(user) {
    if (user) {
        return <h1>hello {formatName(user)}</h1>
    }
    return <h1>hello,小马哥</h1>
}
export default class App extends Component {

    render() {
        return (
            <div>
                {ele}
```

```
                {/* jsx运算 */}
                {2 + 1}
                <br />
                {/*jsx嵌入表达式*/}
                {formatName(user)}
                {/* 添加属性 */}
                <img src={logo} alt="" />
            </div>
            // getGreeting()
        )
    }
}
```

强烈建议大家阅读官网,针对`React.render()`内部的实现操作

元素渲染

> 元素是构成React应用的最小砖块,比如:
>
> `const ele = <h1>hello,world</h1>`

与浏览器的 DOM 元素不同，React 元素是创建开销极小的普通对象。React DOM 会负责更新 DOM 来与 React 元素保持一致

上节课的`ReactDOM.render()`其实就是在渲染DOM节点

更新已渲染的元素

React元素是**不可变对象**,一旦被创建,无妨更改它的子元素或者属性

计时器的例子

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>{new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element,
document.querySelector('#root'));
}

setInterval(tick, 1000);
```

大多数情况下,React应用只会调用一次`ReactDOM.render()`

React只需要更新它需要更新的部分

React DOM会将元素和它的子元素与它们之前的状态进行比较,并只会进行必要的更新来使DOM达到预期的状态

循环绑定元素

当数据从后端请求回来之后,在React中,一般都需要循环绑定元素

map绑定

在React中,循环绑定元素都是使用`map`方法,不能使用`forEach`是因为forEach没有返回值

```
let ul = (<ul>
  { arr.map((item, index)=>{
    return <li key={index}>{item}</li>
  }) }
</ul>);
```

结果会是一个 JSX 元素组成的数组，放入页面中，不会使用逗号分隔开。

> 循环绑定的 JSX 元素，**必须要有 key 属性**，来区分不同的元素，否则会报错。

过滤元素

同样通过`map`方法,只要把不符合条件的元素,返回为`null`即可,原因在于,`null`会被表示为空.如果使用`filter`,那么就没有办法对元素进行处理,只能过滤,还是需要使用`map`进行处理

```
let ul = (<ul>
  { arr.map((item, index)=>{
    return (
      item.price < 1000 ? null : <li key={index}>{item}</li>;
    )
  }) }
</ul>)
```

ref和refs

在React中,类似于Vue,可以通过`ref`标记元素,然后通过`this.refs`获取元素,只是Vue中使用的是`this.$refs`获取元素

字符串写法

通过设置一个字符串值来标记元素,然后通过这个字符串作为属性获取元素

```
class Input extends Component {
  handleChange = (e) => {
   console.log(this.refs.a.value)
  }
  render() {
    return (
      <div>
        <input type="text" ref="a" onChange=
{this.handleChange} />
      </div>
    )
  }
}
```

函数写法

函数作为一个ref的属性值,这个函数接受一个参数,就是真实的DOM元素

可以把这个元素挂载到实例上,方便后面的操作

```
class Input extends React.Component {

    componentDidMount() {
        console.log(this.a); //获取真实的DOM元素
    }
    render() {
        return (
            <div>
               <input type="text" ref={x=>this.a = x}/>
            </div>
        );
    }
}
```

组件&props

React创建组件有来两种方式

- 函数声明
- 类声明

React组件特点:

- 组件名称应该首字母大写,否则会报错
- 组件定义之后,可以像JSX元素一样使用
- 必须使用`render`函数才能将虚拟DOM渲染成真实的DOM
- 使用组件时,可以使用单标签,也可以使用双标签

函数组件

组件，从概念上类似于 JavaScript 函数。它接受任意的入参（即“props”），并返回用于描述页面展示内容的 React 元素。

- 函数声明的组件,必须返回一个JSX元素
- 可以通过属性给组件传递值,函数通过props参数属接收

定义组件最简单的方式就是编写JavaScript函数

```
function Welcome(props){
    return <h2>hello,{props.name}</h2>
}
```

我们称为该组件为"函数组件",因为它本质上就是一个函数

类声明组件

使用ES6的class的方式定义组件

类声明组件需要注意

- 在React中有一个属性Component,是一个基类,使用类声明组件时,**必须继承这个基类**
- 在类中,**必须有render函数**,`constructor`不是必须的
- 在`render`函数中,**需要return一个JSX元素**

```
class Welcome extends Component {
    render() {
        return (
            // 它会将 JSX 所接收的属性（attributes）转换为单个对象传递给组件，这个对象被称之为 “props”。
            <h2>Welcome,{this.props.name}</h2>
        );
    }
}
```

> 注意： 组件名称必须以大写字母开头。 React会将小写字母开头的组件称之为标签
>
> 例如,`<div />`代表HTML的div标签,而`<Welcome />`则代表一个组件,并且需在作用域内使用Welcome

两种方式的区别

真实项目中,都只使用class定义组件

- class定义的组件中有this,状态,生命周期
- function声明都没有

组合组件

组件可以输出的时候嵌入其他的组件。这就可以让我们用同一组件中来抽离出任意层次的细节(复用组件)。按钮，表单，对话框，甚至整个屏幕的内容：在 React 应用程序中，这些通常都会以组件的形式表示。

什么是复合组件:

- 将多个组件进行组合,例如调用两次相同的组件
- 结构非常复杂时需要将组件拆分成小组件
- 会存在父子关系的数据传递

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


class Welcome extends Component {
    render() {
        return (
            <div>
                <h3>Welcome,{this.props.name}</h3>
            </div>
        );
    }
}
class App extends Component {
    render() {
        return (
            <div>
                <Welcome name='张三'></Welcome>
                <Welcome name='李四'></Welcome>
                <Welcome name='王五'></Welcome>
            </div>
        )
    }
```

```
}
// 组合组件
ReactDOM.render(
    <App></App>
    ,
    document.querySelector('#root'));
```

可想而知,React开发其实就是组件化开发,因为React真的是组件化开发的鼻祖

提取组件

将组件拆分成更小的组件

```

import React, { Component } from 'react';
import ReactDOM from 'react-dom';


// 所有props是只读的,不能直接修改
class Avatar extends Component {
    render() {
        return (
            <img src={this.props.user.avatarUrl} alt={this.props.user.name} />
        );
    }
}

class UserInfo extends Component {
    constructor(props) {
        super(props);
        console.log(this.props);

    }
```

```
21
22      render() {
23          return (
24              <div className="userinfo">
25                  <Avatar user={this.props.user}>
</Avatar>
26                  <div className='username'>
27                      <h3>{this.props.user.name}</h3>
28                  </div>
29              </div>
30
31          );
32      }
33  }
34
35
36  class Comment extends Component {
37      constructor(props) {
38          super(props);
39      }
40      render() {
41          return (
42              <div className='comment'>
43                  <UserInfo user={this.props.user}>
</UserInfo>
44                  <div className="Comment-text">
45                      {this.props.user.text}
46                  </div>
47                  <div className="Comment-date">
48                      {this.props.user.date}
49                  </div>
50              </div>
51          );
52      }
53  }
```

```
class App extends Component {
    constructor() {
        super();
        this.user = {
            avatarUrl:
'https://hcdn1.apeland.cn/media/course/icon2.png',
            name: '张三',
            text: "hello,React component",
            date: new Date().toLocaleString()
        }
    }
    render() {
        return (
            <div>
                <Comment user={this.user}></Comment>
            </div>
        )
    }
}
// 组合组件
ReactDOM.render(
    <App></App>
    ,
    document.querySelector('#root'));


```

最初看上去，提取组件可能是一件繁重的工作，但是，在大型应用中，构建可复用组件库是完全值得的。 根据经验来看，如果 UI 中有一部分被多次使用（Button，Panel，Avatar），或者组件本身就足够复杂（App，Comment），那么它就是一个可复用组件的候选项。你说对吧!

父子组件通信

父传子

父组件通过行间属性传递数据到子组件,子组件通过实例上的props属性接收新的数据

React 的数据是单向数据流，只能一层一层往下传递。当组件的属性发生改变，那么当前的视图就会更新

子传父

通过给子组件传递一个函数，子组件调用父亲的函数将值作为参数传递给父组件，父组件更新值，刷新视图。

父组件中定义一个函数，通过属性传递给子组件。

> 这个传递的函数必须是一个箭头函数

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class ChildCom extends Component {
    constructor(props) {
        super(props);
        console.log(props);
        this.state = {
            val:''
        }
        this.handlerClick = this.handlerClick.bind(this);
        this.handlerChange = this.handlerChange.bind(this);
    }
    handlerChange(event){
        this.setState({
            val: event.target.value
        })
```

```
    }
    handlerClick(){
        if(this.state.val){
            this.props.addHandler(this.state.val);
            // 清空输入框
            this.setState({
                val: ''
            })
        }
    }
    render() {
        return (
            <div>
                <input type="text" value =
{this.state.val} onChange = {this.handlerChange}/>
                <button onClick={this.handlerClick}>添
加</button>
                {
                    this.props.menus.map((item,index)
=> {
                        return <p key = {index}>{item}
</p>
                    })
                }

            </div>
        );
    }
}


class App extends Component {
    constructor(props) {
        super(props);
        this.state = {
```

```
            menus: ['烤腰子', '辣炒鸡丁', '炸黄花鱼']
        }
    }
    // 一定要使用箭头函数
    addHandler = (val)=>{
        this.state.menus.push(val);
        this.setState({
           menus:this.state.menus
       })
    }

    render() {
        // 修改状态之后,会重新调用render
        return (
            <div>
                <ChildCom menus={this.state.menus}
addHandler = {this.addHandler}></ChildCom>
            </div>
        );
    }
}



ReactDOM.render(<App />,
document.querySelector('#root'));
```

Props的只读性

组件无论是使用函数声明还是通过 class 声明，都决不能修改自身的 props

```
//该函数不会尝试更改入参，且多次调用下相同的入参始终返回相同的结果。
function sum(a, b) {
    return a + b;
}
//它更改了自己的入参
function withdraw(account, amount) {
    account.total -= amount;
}
```

所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。

state

组件状态

组件中数据的来源

- 属性:是由外接传递过来的
- 状态:是自己的,只能通过`setState`来改变状态

> 只有类声明的组件中,才有状态

修改状态

除了`constructor`之外的其它地方,如果需要修改状态,都只能通过`this.setState`方法

这个方法传入的第一个参数,可以是一个对象,也可以是一个函数

- 是一个对象，这个对象中包含需要改变的属性，它会与原有的状态进行合并
- 是一个函数，接收第一个参数是 prevState，上一个状态对象，第二个参数是 props

这个方法的第二个参数，是一个回调函数，在状态改变之后执行。

> 如果下一个状态依赖于上一个状态，需要写成函数的方式

关于setState

- 在 react 组件的生命周期或事件的绑定中，setState 是异步的
- 在定时器或原生的事件中，setState 不一定是异步的

```
// state.count 当前为 0
componentDidMount(){
    this.setState({count: this.state.count + 1});
    console.log(this.state.count)
}
// 输出 0
```

在元素渲染章节中，我们只了解了一种更新 UI 界面的方法。通过调用 `ReactDOM.render()` 来修改我们想要渲染的元素

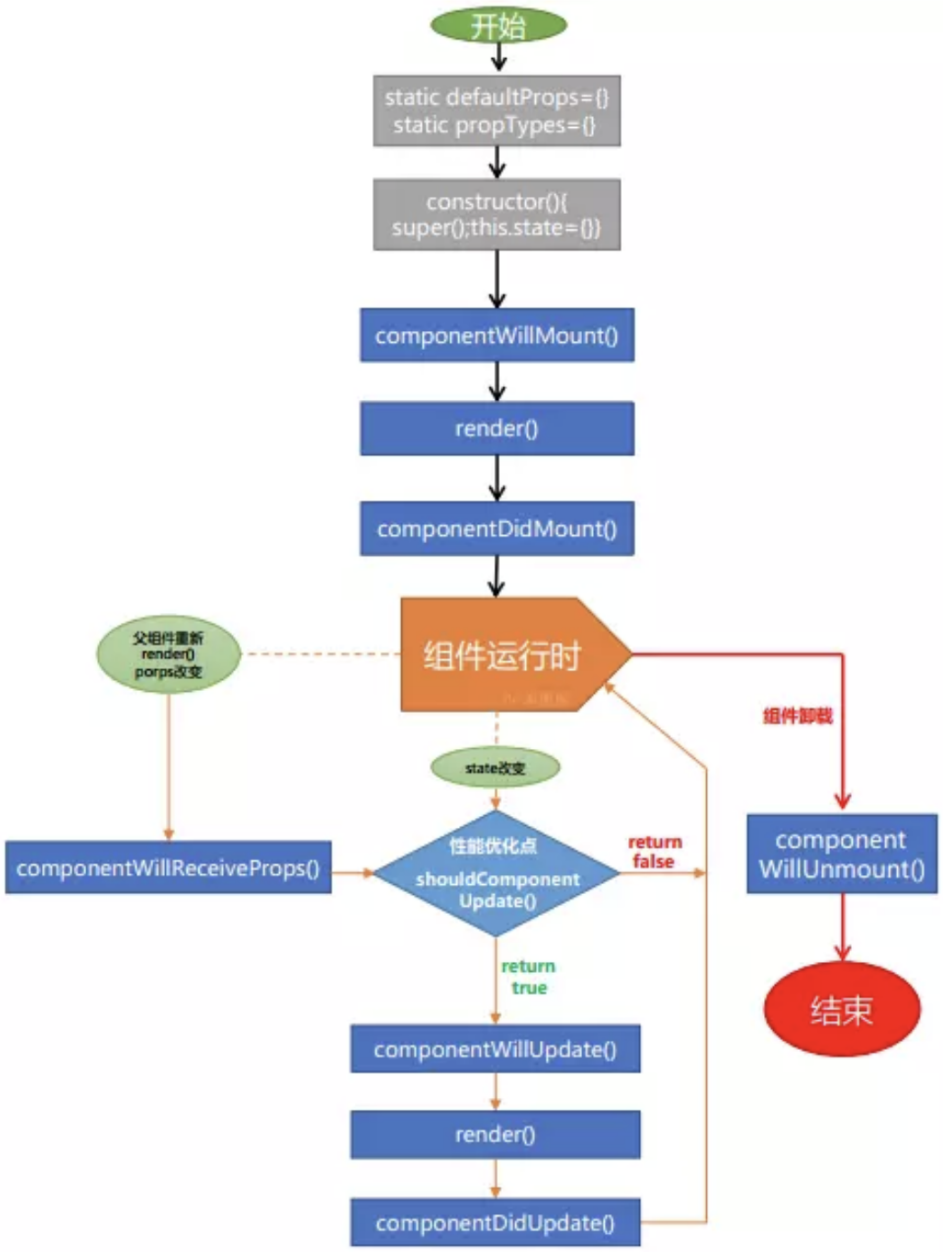
```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>{new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element,
document.querySelector('#root'));
}

setInterval(tick, 1000);
```

本节学习如何封装真正可复用的Clock组件

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

// 学习如何封装真正可复用的Clock组件。
class Clock extends Component {
    constructor(props) {
        super(props);
        this.state = {
            date: new Date().toLocaleString()
        }
    }
    componentDidMount() {
        this.timer = setInterval(() => {
            // 注意1 不能直接修改state
            // this.state.date = new Date(); //错误

            // 注意2： setState()是异步的
            this.setState({
                date: new Date().toLocaleString()
            })
        }, 1000);
    }
    componentWillUnmount() {
        clearInterval(this.timer);
    }
    render() {
        // 修改状态之后,会重新调用render
        return (
            <div>
                <h3>当前时间为:{this.state.date}</h3>
            </div>
        );
```

```
    }
}


ReactDOM.render(<Clock />,
document.querySelector('#root'));
```

生命周期



> 如图，React生命周期主要包括三个阶段：初始化阶段、运行中阶段和销毁阶段，在React不同的生命周期里，会依次触发不同的钩子函数，下面我们就来详细介绍一下React的生命周期函数

```
1 import React, { Component } from 'react';
2 import ReactDOM from 'react-dom';
3
```

```
class SubCounter extends Component {
    // 组件将要接收属性
    componentWillReceiveProps(newProps){
        console.log('9.子组件将要接收到新属性',newProps);
    }
    shouldComponentUpdate(newProps,newState){
        console.log('10.子组件是否需要更新')
        if(newProps.num % 3 === 0){
            return true;
        }else{
            return false;
        }

    }
    componentWillUpdate() {
        console.log('11、子组件将要更新');
    }

    componentDidUpdate() {
        console.log('13、子组件更新完成');
    }

    componentWillUnmount() {
        console.log('14、子组件将卸载');
    }
    render() {
        console.log('12.子组件挂载中')
        return (
            <div>
              <p>{this.props.num}</p>
            </div>
        );
    }
}
```

```
38
39
40  class Counter extends Component {
41      static defaultyProps = {
42          //1.加载默认属性
43          name:'小马哥',
44          age:18
45      }
46      constructor(props) {
47          super(props);
48          //2.记载默认状态
49          this.state = {
50              num: 0
51          }
52      }
53      componentWillMount() {
54          // 此时可以访问属性和状态，可以进行api调用，但没办法做DOM相关操作
55          console.log('3.父组件将要被挂载');
56
57      }
58      componentDidMount() {
59          // 组件已挂载，可进行状态更新操作。通常 都在此方法中发送请求
60          console.log('5.组件挂载完成');
61      }
62
63      shouldComponentUpdate(newProps, newState) {
64          // 组件是否需要更新，返回布尔值，优化点
65          console.log('6.父组件是否被更新');
66          // console.log(newProps, newState);
67          if (newState.num % 2 === 0) {
68              return true;
69          } else {
```

```
            // 此函数 会返回一个boolean值,返回true更新页面,返回false不更新页面
            return false;
        }
    }

    componentWillUpdate(){
        console.log('7.父组件将要更新');

    }
    componentDidUpdate(){
        console.log('8.父组件更新完成');

    }

    handlerClick = () => {
        // 可能,只是说可能,官网上都是这样说的.......会导致计数可能不准确,
        // this.setState({
        //     num: parseInt(this.props.increment) + this.state.num
        // })
        // 发现点击之后,得到的结果为0,这是因为setState()是异步的
        // console.log(this.state.num);

        // 要解决这个问题，可以让 setState() 接收一个函数而不是一个对象。
        // 这个函数用上一个 state 作为第一个参数，将此次更新被应用时的 props 做为第二个参数
        this.setState((state, props) => {
            return {
                num: state.num + parseInt(props.increment)
            }
```

```
        }, () => {
            console.log(this.state.num);
        })
    }
    render() {
        // 修改状态之后,会重新调用render
        console.log('4.render(父组件)渲染了');

        return (
            <div>
                <h3>当前数值:{this.state.num}</h3>
                <button onClick={this.handlerClick}>+1</button>
                <h3>我是子组件</h3>
                <SubCounter num = {this.state.num}></SubCounter>
            </div>
        );
    }
}



ReactDOM.render(<Counter increment='1' />, document.querySelector('#root'));
```

受控组件

受控组件，就是受状态控制的组件，需要与状态进行相应的绑定

- 受控组件必须要有一个 **onChange 事件**，否则不能使用
- 受控组件可以**赋予默认值**（实际上就是设置初始状态）

- **官方推荐使用受控组件**的写法

可以使用受控组件实现双向绑定。

非受控组件，则不是通过与状态进行绑定来实现的，而是**通过操作 DOM 来实现**。除非操作 DOM，否则没有办法设置默认值。

受控组件实现

- 设置初始状态，也就是设置默认值
- 将输入框的 value``` 值与相应状态进行绑定
- 使用 onChange`事件，对状态进行修改，从而反映到 value` 上

```
class Input extends Component {
  constructor() {
    super();
    this.state = {
      val: '' // 这个位置用来设置默认值
    }
  }
  handleChange = (e) => {
    let val = e.target.value
    this.setState({val});
  }
  render() {
    return (
      <div>
        // 让 value 与状态进行绑定，通过事件处理修改状态来达到修改值的效果
        <input type="text" value={this.state.val} onChange={this.handleChange} />
        {this.state.val}
```

```
      </div>
    )
  }
}
```

非受控组件的实现

- 通过 ref `标记一个元素，然后可以通过 this.refs.xx` 来获取这个元素
- 通过 onChange `事件监听到 value` 的变化，获取到这个数据
- 然后通过操作 DOM 将数据放到需要的地方

```
class Input extends Component {
  constructor() {
    super();
    this.state = {
      val: ''
    }
  }
  handleChange = (e) => {
    //=> 这里可以通过 e.target.value 获取
    let val = this.refs.a.value;
    this.setState({val});
  }
  render() {
    return (
      <div>
        <input type="text" onChange={this.handleChange} ref="a" />
        {this.state.val}
      </div>
    )
  }
```

```
}
```

> 实际上，上面实现的只是单向绑定，如果要实现双向绑定，需要使用两个 onChange 事件，可以不需要用到状态

下面来实现双向绑定：

```
class Input extends Component {
  handleChange = (e) => {
    //=> 这里可以通过 e.target.value 获取
    if (e.target === this.refs.a) {
      this.refs.b.value = e.target.value;
    } else {
      this.refs.a.value = e.target.value;
    }
  }
  render() {
    return (
      <div onChange={this.handleChange}>
        <input type="text" ref="a" />
        <input type="text" ref="b" />
      </div>
    )
  }
}
```