React入门

课堂目标

1. create-react-app使用
2. 掌握组件使用
3. 掌握JSX语法
4. 掌握setState
5. 理解事件处理、组件生命周期
6. 掌握组件通信各种方式

资源

1. react
2. create-react-app

起步

1. 安装官方脚手架：`npm install -g create-react-app`
2. 创建项目：`create-react-app lesson1`
3. 启动项目：`npm start`
4. 暴露配置项：`npm run eject`

文件结构

文件结构一览

```
├── README.md                    文档

├── public                       静态资源
|   ├── favicon.ico
|   ├── index.html
|   └── manifest.json
└── src                          源码
    ├── App.css
    ├── App.js                   根组件
    ├── App.test.js
    ├── index.css                全局样式
    ├── index.js                 入口文件
    ├── logo.svg
    └── serviceWorker.js         pwa支持
```

```
├── package.json                    npm 依赖
```

env.js用来处理.env文件中配置的环境变量

```
// node运行环境：development、production、
test等
const NODE_ENV = process.env.NODE_ENV;

// 要扫描的文件名数组
var dotenvFiles = [
  `${paths.dotenv}.${NODE_ENV}.local`, //
.env.development.local
  `${paths.dotenv}.${NODE_ENV}`,       //
.env.development
  NODE_ENV !== 'test' &&
`${paths.dotenv}.local`, // .env.local
  paths.dotenv, // .env
].filter(Boolean);

// 从.env*文件加载环境变量
dotenvFiles.forEach(dotenvFile => {
  if (fs.existsSync(dotenvFile)) {
    require('dotenv-expand')(
      require('dotenv').config({
        path: dotenvFile,
```

```
      })
    );
  }
});
```

实践一下，修改一下默认端口号，创建.env文件

```
PORT=8080
```

webpack.config.js 是webpack配置文件，开头的常量声明可以看出cra能够支持ts、sass及css模块化

```
// Check if TypeScript is setup
const useTypeScript =
fs.existsSync(paths.appTsConfig);

// style files regexes
const cssRegex = /\.css$/;
const cssModuleRegex = /\.module\.css$/;
const sassRegex = /\.(scss|sass)$/;
const sassModuleRegex = /\.module\.
(scss|sass)$/;
```

React和ReactDom

删除src下面所有代码，新建index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

// 这里怎么没有出现React字眼?
// JSX => React.createElement(...)
ReactDOM.render(<h1>Hello React</h1>,
document.querySelector('#root'));
```

> React负责逻辑控制，数据 -> VDOM
>
> ReactDom渲染实际DOM，VDOM -> DOM
>
> React使用JSX来描述UI

> 入口文件定义，webpack.config.js
>
> ```
> entry: [
> // WebpackDevServer客户端，它实现开发时热更新功能
> isEnvDevelopment &&
> require.resolve('react-dev-utils/webpackHotDevClient'),
> // 应用程序入口：src/index
> paths.appIndexJs,
> ].filter(Boolean),
> ```

JSX

JSX是一种JavaScript的语法扩展，其格式比较像模版语言，但事实上完全是在JavaScript内部实现的。

JSX可以很好地描述UI，能够有效提高开发效率，体验JSX

> JSX实质就是React.createElement的调用，最终的结果是React“元素”（JavaScript对象）。
>
> ```
> const jsx = <h2>react study</h2>;
> ReactDOM.render(jsx,
> document.getElementById('root'));
> ```

使用JSX

表达式{}的使用，index.js

```
const name = "react study";
const jsx = <h2>{name}</h2>;
```

函数也是合法表达式，index.js

```
const user = { firstName: "tom", lastName:
"jerry" };
function formatName(user) {
  return user.firstName + " " +
user.lastName;
}
const jsx = <h2>{formatName(user)}</h2>;
```

jsx是js对象，也是合法表达式，index.js

```
const greet = <p>hello, Jerry</p>
const jsx = <h2>{greet}</h2>;
```

条件语句可以基于上面结论实现，index.js

```
const showTitle = true;
const title = name ? <h2>{name}</h2> :
null;
const jsx = (
  <div>
    {/* 条件语句 */}
    {title}
  </div>
);
```

数组会被作为一组子元素对待，数组中存放一组jsx可用于显示列表数据

```
const arr = [1,2,3].map(num => <li key=
{num}>{num}</li>)
const jsx = (
  <div>
    {/* 数组 */}
    <ul>{arr}</ul>
  </div>
);
```

属性的使用

```
import logo from "./logo.svg";

const jsx = (
  <div>
    {/* 属性：静态值用双引号，动态值用花括号；class、for等要特殊处理。 */}
    <img src={logo} style={{ width: 100 }} className="img" />
  </div>
);
```

> css模块化，创建index.module.css，index.js
>
> ```
> import style from "./index.module.css";
> <img className={style.img} />
> ```
>
> 更多css modules规则参考

组件

组件是抽象的独立功能模块，react应用程序由组件构建而成。

组件的两种形式

组件有两种形式：**function组件和class组件**。

class组件

class组件通常**拥有状态和生命周期，继承于Component，实现render方法**，创建pages/Home.js

提取前面jsx相关代码至pages/Home.js

```
import React, { Component } from "react";
import logo from "../logo.svg";
import style from "../index.module.css";

export default class Home extends Component
{
  render() {
    const name = "react study";
    const user = { firstName: "tom",
lastName: "jerry" };
    function formatName(user) {
      return user.firstName + " " +
user.lastName;
    }
    const greet = <p>hello, Jerry</p>;
    const arr = [1, 2, 3].map(num => <li
key={num}>{num}</li>);
```

```
    return (
      <div>
        {/* 条件语句 */}
        {name ? <h2>{name}</h2> : null}
        {/* 函数也是表达式 */}
        {formatName(user)}
        {/* jsx也是表达式 */}
        {greet}
        {/* 数组 */}
        <ul>{arr}</ul>
        {/* 属性 */}
        <img src={logo} className=
{style.img} alt="" />
      </div>
    );
  }
}
```

创建并指定src/App.js为根组件

```
import React, { Component } from
'react';
import Home from './pages/Home';

class App extends Component {
render() {
```

```
return (
<div>
  <Home></Home>
</div>
);
}
}

export default App;
```

index.js中使用App组件

```
import App from "./App";

ReactDOM.render(<App />,
document.getElementById("root"));
```

function组件

函数组件通常**无状态**，仅**关注内容展示**，返回渲染结果即可。

改造App.js

```
import React from "react";
import User from "./pages/User";

function App() {
  return (
    <div>
      <User />
    </div>
  );
}

export default App;
```

> 从React16.8开始引入了hooks，函数组件也能够拥有状态，后面组件状态管理部分讨论

组件状态管理

如果组件中数据会变化，并影响页面内容，则组件需要拥有状态（state）并维护状态。

类组件中的状态管理

class组件使用state和setState维护状态

创建一个Clock

```
import React, { Component } from "react";

export default class Home extends
React.Component {
  constructor(props) {
    super(props);
    // 使用state属性维护状态，在构造函数中初始化
状态
    this.state = { date: new Date() };
  }
  componentDidMount() {
    // 组件挂载时启动定时器每秒更新状态
    this.timerID = setInterval(() => {
      // 使用setState方法更新状态
      this.setState({
        date: new Date()
      });
    }, 1000);
  }
  componentWillUnmount() {
    // 组件卸载时停止定时器
    clearInterval(this.timerID);
  }
  render() {
```

```
    return <div>
{this.state.date.toLocaleTimeString()}
</div>;
  }
}
```

拓展：**setState特性讨论**

- 用setState更新状态而不能直接修改

```
this.state.counter += 1; //错误的
```

- setState是批量执行的，因此对同一个状态执行多次只起一次作用，多个状态更新可以放在同一个setState中进行：

```
componentDidMount() {
    // 假如couter初始值为0，执行三次以后其结果是多少?
    this.setState({counter: this.state.counter + 1});
    this.setState({counter: this.state.counter + 1});
    this.setState({counter: this.state.counter + 1});
}
```

- setState通常是异步的，因此如果要获取到最新状态值有以下三种方式：

  1. 传递函数给setState方法，

```
this.setState((nextState, props) => ({
 counter: state.counter + 1}));// 1
this.setState((nextState, props) => ({
 counter: state.counter + 1}));// 2
this.setState((nextState, props) => ({
 counter: state.counter + 1}));// 3
```

  2. 使用定时器：

```
setTimeout(() => {
  console.log(this.state.counter);
}, 0);
```

  3. 原生事件中修改状态

```
componentDidMount(){

 document.body.addEventListener('click
', this.changeValue, false)
}

changeValue = () => {
    this.setState({counter:
this.state.counter+1})
    console.log(this.state.counter)
}
```

函数组件中的状态管理

函数组件通过hooks api维护状态

```
import React, { useState, useEffect } from
"react";

export default function User() {
  const [date, setDate] = useState(new
Date());
  useEffect(() => {
    const timeId = setInterval(() => {
      setDate(new Date());
    }, 1000);
    return () => clearInterval(timeId);
  });

  return <div>{date.toLocaleTimeString()}
</div>;
}
```

> hooks api后面课程会继续深入讲解

事件处理

React中使用onXX写法来监听事件。

范例：用户输入事件，创建Search.js

```
import React, { Component } from "react";

export default class Search extends
Component {
  constructor(props) {
    super(props);
    this.state = { name: "" };
    // this.change =
this.change.bind(this);
  }
  btn = () => {
    //使用箭头函数，不需要指定回调函数this，且便
于传递参数
    console.log("btn");
  };
  change = e => {
    let value = e.target.value;
    this.setState({
      name: value,
    });
    console.log("name", this.state.name);
  };
  render() {
    const { name } = this.state;
    return (
      <div>
```

```
        <button onClick={this.btn}>按钮
</button>
        <input
          type="text"
          placeholder="请输入"
          name={name}
          onChange={this.change}
        />
      </div>
    );
  }
}
```

事件回调函数注意绑定this指向，常见三种方法：

1. 构造函数中绑定并覆盖：this.change = this.change.bind(this)
2. 方法定义为箭头函数：change = ()=>{}
3. 事件中定义为箭头函数：onChange={()=>this.change()}

react里遵循单项数据流，没有双向绑定，输入框要设置value和onChange，称为受控组件

组件通信

Props属性传递

Props属性传递可用于父子组件相互通信

```
// index.js
ReactDOM.render(<App title="开课吧真不错" />,
document.querySelector('#root'));

// App.js
<h2>{this.props.title}</h2>
```

如果父组件传递的是函数，则可以把子组件信息传入父组件，这个常称为状态提升，StateMgt.js

```
// StateMgt
<Clock change={this.onChange}/>

// Clock
this.timerID = setInterval(() => {
    this.setState({
        date: new Date()
    }, ()=>{
        // 每次状态更新就通知父组件
        this.props.change(this.state.date);
    });
}, 1000);
```

context

跨层级组件之间通信

> 主要用于组件库开发中，后面组件化内容中详细介绍

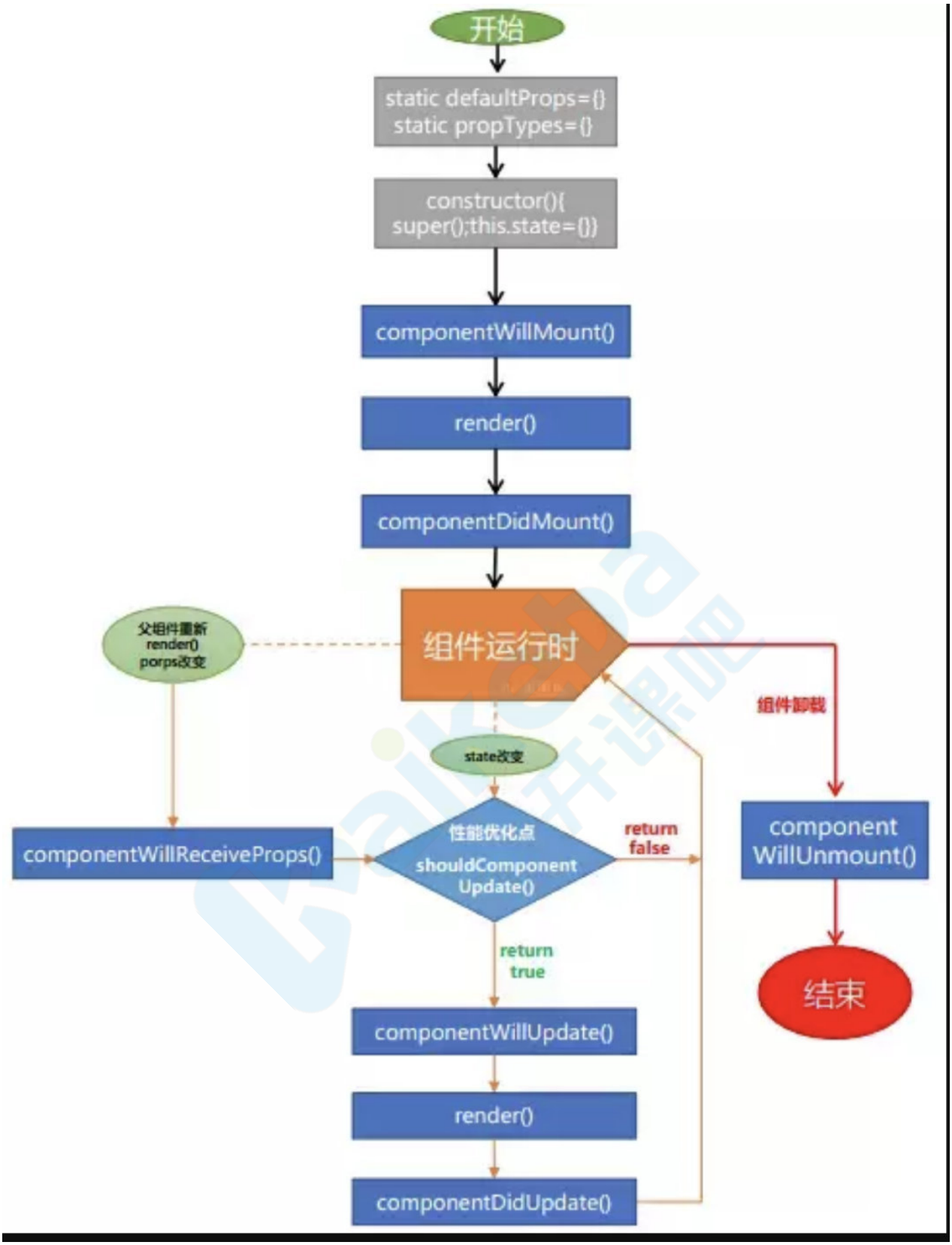
redux

类似vuex，无明显关系的组件间通信

> 后面全家桶部分详细介绍

生命周期

React V16.3之前的生命周期



范例：验证生命周期，创建Lifecycle.js

```
import React, { Component } from "react";

export default class LifeCycle extends
Component {
  constructor(props) {
    super(props);
    this.state = { counter: 0 };
    console.log("constructor");
  }
  componentWillMount() {//组件挂载之前
    const { counter } = this.state;
    console.log("componentWillMount",
counter);
  }
  componentDidMount() {//组件挂载之后
    const { counter } = this.state;
    console.log("componentDidMount",
counter);
  }
  componentWillUpdate() {//组件更新之前
    const { counter } = this.state;
    console.log("componentWillUpdate",
counter);
  }
  componentDidUpdate() {//组件更新之后
    const { counter } = this.state;
```

```
    console.log("componentDidUpdate",
counter);
  }
  componentWillUnmount() {//组件卸载之前
    const { counter } = this.state;
    console.log("componentWillUnmount",
counter);
  }
  shouldComponentUpdate(nextProps,
nextState) {//组件是否render
    const { counter } = this.state;
    console.log("shouldComponentUpdate",
counter, nextState.counter);
    return counter != 5;
  }
  setShow = () => {
    this.setState({
      counter: this.state.counter + 1,
    });
  };
  render() {
    const { counter } = this.state;
    return (
      <div>
        <h1>LifeCycle</h1>
        <button onClick={this.setShow}>改变
</button>
```

```
        {!!(counter % 2) && (
          <>
            <h2>{counter}</h2>
            <Foo />
          </>
        )}
      </div>
    );
  }
}

class Foo extends Component {
  componentWillUnmount() {
    console.log("Foo
componentWillUnmount");
  }
  render() {
    return <h2>我是Foo组件</h2>;
  }
}
```

> 组件生命周期在[React v16.x之后的变化]

后续展望

回顾