

核心api

npm i create-react-app -g(<https://www.html.cn/create-react-app/docs/available-scripts/>)

创建: create-react-app projName

可选方案:npx create-react-app projName

JSX语法

什么是jsx?

jsx是语法糖

React使用jsx来替换常规的javascript

jsx是一个看起来很像xml的javaScript语法扩展

为什么需要JSX?

- jsx执行更快，因为它在编译为javascript代码后进行了优化
- 它是类型安全的，在编译过程中就能发现错误
- 使用jsx编写模板更加简单快速

原理: babel-loader会预编译jsx为React.createElement(...)

1.webpack+babel-loader编译时，替换jsx为React.createElement(...)

2.所有React.createElement(...)执行结束会得到一个js对象树，能够完整描述dom结构，称之为虚拟dom

3.React-dom.render(vdom,container)可以将vdom转换为dom追加至container中

- 插值表达式

```
1  /* 表达式:合法js表达式即可 */
2  <h1>{name}</h1>
3  /* 函数表达式 */
4  <p>{formatName({firstName:'tom',lastName:'zyl'})}</p>
5  /* jsx也是表达式 */
6  {<p>hello, jerry! </p>}
```

- 属性绑定

```
1  /* 属性 */
2  <img src={logo} style={{width:'100px'}} alt="logo"
3  className="img"/>
<label htmlFor="ff">ffff</label>
```

- 事件处理

```

1 <input
2   type="text"
3   value={this.state.name}
4   onChange={e => this.handleChange(e)}
5   />
6
7
8 handleChange = e => {
9   this.setState({
10    name: e.target.value
11  })
12}
13

```

注意: this的指向

组件: 两种形式

- 函数

```

1 function welcome1(props){
2   return(
3     <div>
4       welcome1,{props.name}-{props.age}
5     </div>
6   )
7 }

```

- 类

```

1 class welcome2 extends Component{
2   render(){
3     return(
4       <div>
5         welcome2,{this.props.name}-{this.props.age}
6       </div>
7     )
8   }
9 }

```

- 组件状态

- 状态声明

```

1 constructor(props){
2   super(props);
3   this.state={
4     date:new Date(),
5     counter:1
6   }
7 }

```

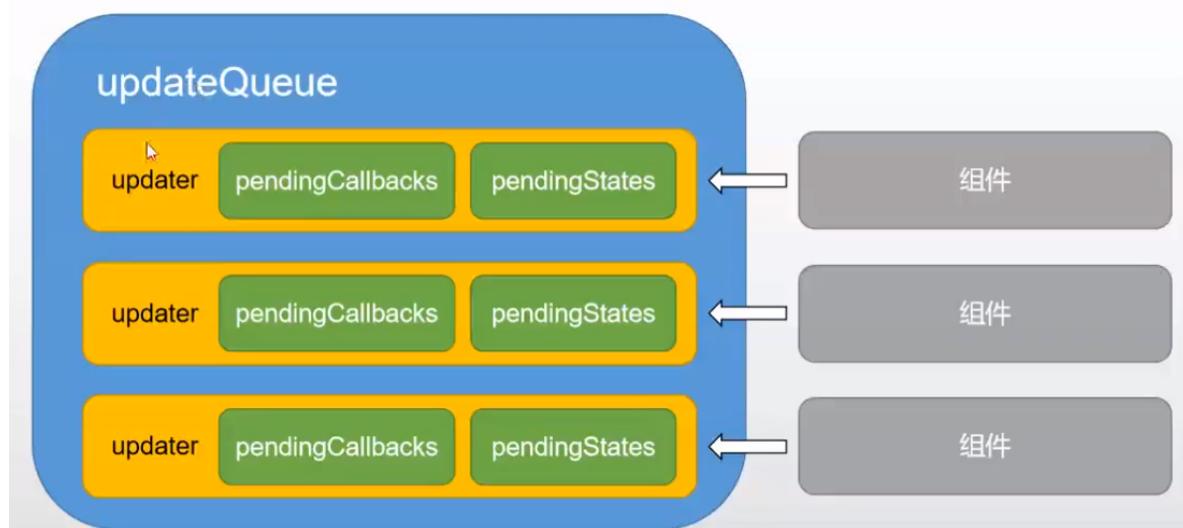
- 修改状态

```
1  this.setState({  
2      counter:this.state.counter+1  
3  })  
4  this.setState(state=>({  
5      counter:state.counter+1  
6  }))
```

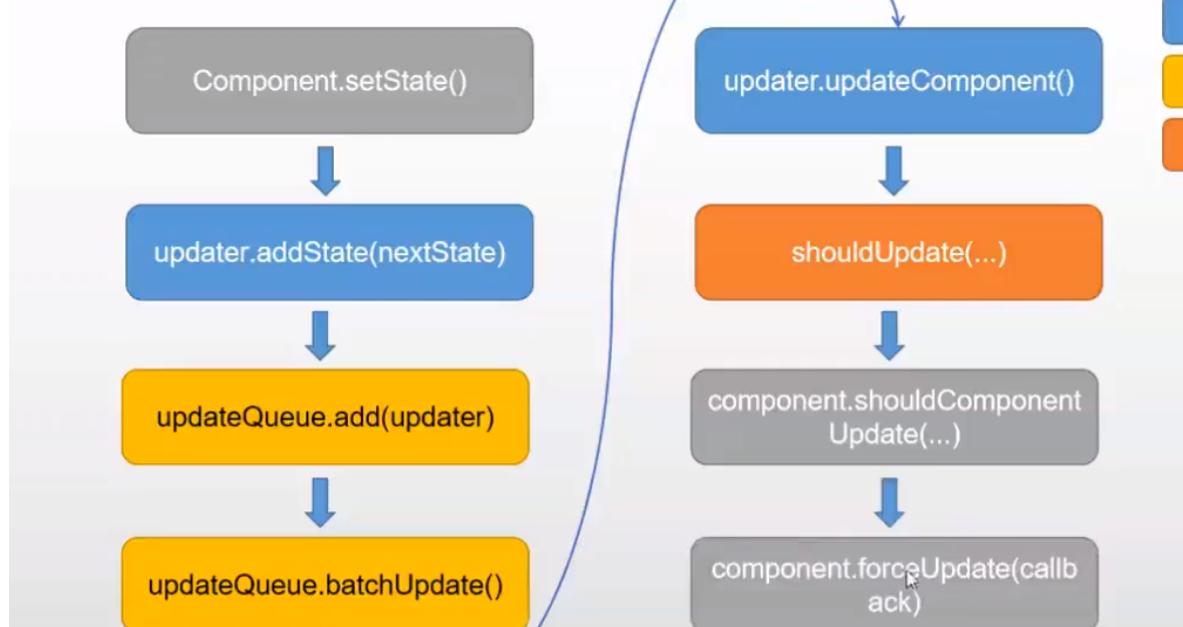
setState原理

setState是批量异步也可能同步执行的

setState工作原理1

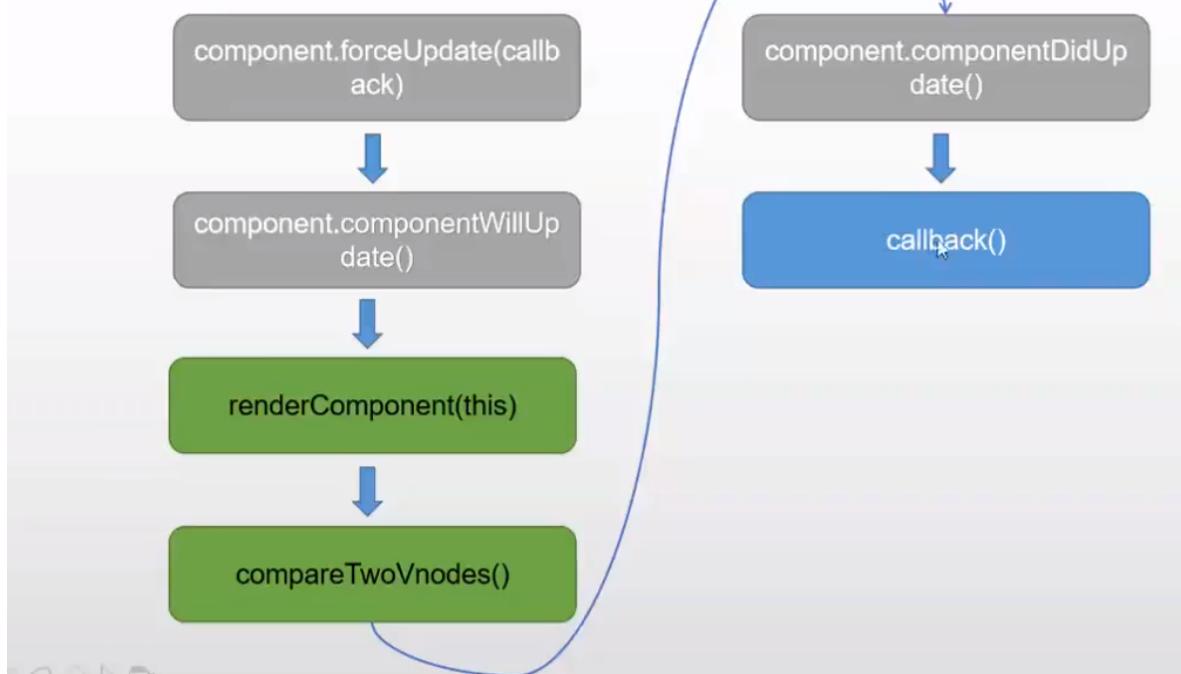


setState工作原理2





setState工作原理3



setState特性讨论

- 用setState更新状态而不能直接修改

```
1 | 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：

```
1 | componentDidMount() {
2 |   // 假如counter初始值为0，执行行行多次以后其结果是多少？
3 |   this.setState({counter: this.state.counter + 1});
4 |   this.setState({counter: this.state.counter + 2});
5 | }
```

- 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：

- 传递函数给setState方法，

```
1 | this.setState(nextState => ({ counter:
2 |   nextState.counter + 1})); // 1
3 | this.setState(nextState => ({ counter:
4 |   nextState.counter + 1})); // 2
5 | this.setState(nextState => ({ counter:
6 |   nextState.counter + 1})); // 3
```

- 使用定时器：

```
1 | setTimeout(() => {
2 |   this.addValue();
3 |   console.log(this.state.counter);
4 | }, 0);
```

- 原生事件中修改状态

```

1 | componentDidMount() {
2 |   document.body.addEventListener('click',
3 |     this.changeValue, false)
4 |   }
5 |   changeValue = () => {
6 |     this.setState({counter: this.state.counter+1})
7 |     console.log(this.state.counter)
8 |   }

```

总结： setState只有在合成事件和生命周期函数中是异步的，在原生事件和setTimeout中都是同步的，这里的异步其实是批量更新。

函数组件中的状态管理

函数组件通过hooks api维护状态

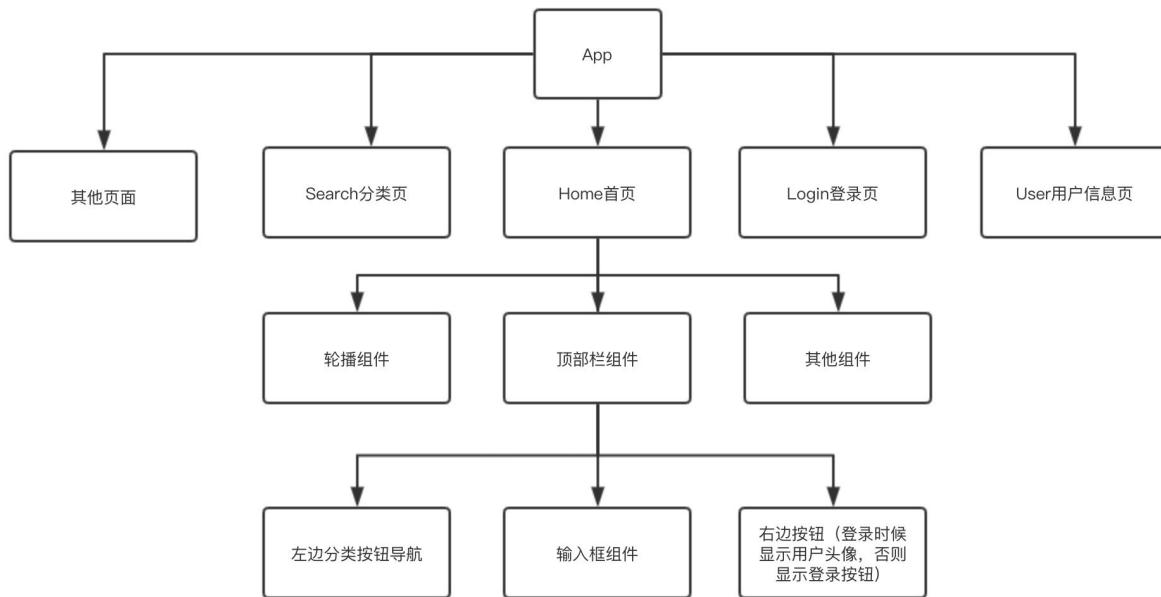
```

1 | import React, { useState, useEffect } from "react";
2 | export function FunctionComponent(props) {
3 |   const [date, setDate] = useState(new Date());
4 |   useEffect(() => {//副作用用
5 |     const timer = setInterval(() => {
6 |       setDate(new Date());
7 |     }, 1000);
8 |     return () => clearInterval(timer); //组件卸载的时候
9 |   }执行行行
10 |   );
11 |   return (
12 |     <div>
13 |       <h3>FunctionComponent</h3>
14 |       <p>{date.toLocaleTimeString()}</p>
15 |     </div>
16 |   );
17 | }

```

提示：如果你熟悉 React class 的生命周期函数，你可以把useEffect Hook 看做componentDidMount，componentDidUpdate 和componentWillUnmount 这三个函数的组合

组件跨层级通信-context



React中使用Context实现祖代组件向后代组件跨层级传值。Vue中的provide&inject来源于context。在Context模式下有两个角色:

- Provider:外层提供数据的组件
- Consumer:内层获取数据的组件

使用Context

创建Context => 获取Provider和Consumer => Provider提供值 => Consumer消费值

范例：模拟redux存放全局状态，在组件间共享

```

1 //App.js
2 import React from 'react';
3 import './App.css';
4 import Home from "./pages/Home";
5 import User from "./pages/User";
6 import { Provider } from "./AppContainer";
7
8 const store = {
9     home: {
10         imgs: [
11             {
12                 "src":
13                     "http://m.360buyimg.com/mobilecms/s700x280_jfs/t1/49973/2/8672/125419/5d679259
14                     Ecd46f8e7/0669f8801dff67e8.jpg!cr_1125x445_0_171!q70.jpg.dpg"
15             }
16         ],
17         user: {
18             isLoggedIn: true,
19             userName: "Rabbit"
20         }
21     }
22
23 function App() {
24     return (
25         <div className="App">
26             <header className="App-header">

```

```
27     <Provider value={store}>
28         <Home/>
29         <User/>
30     </Provider>
31   </header>
32 </div>
33 );
34 }
35
36 export default App;
37
```

```
1 //AppContext.js
2 import react,{createContext} from "react";
3
4 export const Context=createContext();
5 export const Provider=Context.Provider;
6 export const Consumer=Context.Consumer;
```

```
1 //pages/Home.js
2 import React from 'react'
3 import { Consumer } from "../AppContext";
4
5 export default function Home() {
6   return (
7     <div>
8       <Consumer>
9         {(ctx)=><HomeCmp {...ctx}>}
10      </Consumer>
11    </div>
12  )
13}
14
15
16 function HomeCmp(props){
17   const {home,user}=props;
18   const {isLogin,userName}=user;
19   return (
20     <div>
21       {isLogin?userName:'登录'}
22     </div>
23   )
24 }
```

```
1 import React from 'react'
2 import { Consumer } from '../AppContext';
3 import TabBar from "../components/TabBar";
4
5
6 export default function User(props) {
7   return (
8     <div>
9       <Consumer>
10         {ctx=><UserComp {...ctx}>}
```

```

11      </Consumer>
12      <TabBar/>
13  </div>
14  )
15 }
16
17 function UserComp(props){
18   const {user,home}=props;
19   const {isLogin,userName}=user;
20   return (
21     <div>
22       {isLogin?userName:'登录'}
23     </div>
24   )
25 }
```

```

1 //components/TabBar
2 import React from 'react'
3 import { Consumer } from '../AppContext';
4
5 export default function TabBar() {
6   return (
7     <div>
8       <Consumer>
9         {ctx=><TabBarComp {...ctx}/>}
10      </Consumer>
11    </div>
12  )
13 }
14
15 function TabBarComp(props){
16   const {user,home}=props;
17   const {isLogin,userName}=user;
18   return (
19     <div>
20       {isLogin?userName:'登录'}
21     </div>
22   )
23 }
```

在React的官方文档中， Context 被归类为高级部分(Advanced)，属于React的高级API，但官方并不建议在稳定版的App中使用Context。不过，这并非意味着我们不需要关注 Context。事实上，很多优秀的React组件都通过Context来完成自己的功能，比如react-redux的，就是通过 Context 提供一个全局态的 store，路由组件react-router通过 Context 管理路由状态等等。在React组件开发中，如果用好 Context，可以让你的组件变得强大，而且灵活。

高阶组件-HOC

为了提高组件复用率，可测试性，就要保证组件功能单一性；但是若要满足复杂需求就要扩展功能单一的组件，在React里就有了HOC (Higher-Order Components) 的概念，

定义：是一个函数，它接收一个组件并返回另一个组件。

基本使用

```

1 import React from "react";
2 function Child(props) {
3   return <div className="border">Child+{props.name}</div>;
4 }
5
6 //这里大写开头的Cmp是指function或者class组件
7 const foo = Cmp => props => {
8   return <Cmp {...props} />;
9 };
10 const Foo = foo(Child);
11 export default function HOCPage() {
12   return (
13     <div>
14       HocPage
15       <Foo name={"msg"} />
16       {/* {foo(Child)({
17         name: "msg"
18       })} */}
19     </div>
20   );
21 }
22

```

运用hoc改写前面的Context例子：

```

1 import React from "react";
2 import { Consumer } from "../AppContext";
3 const handleConsumer = Cmp => props => {
4   return <Consumer> {ctx => <Cmp {...ctx} {...props}></Cmp>} </Consumer>;
5 };
6 const HandleConsumer = handleConsumer(UserCmp);
7 export default function User(props) {
8   return (
9     <div>
10       <HandleConsumer />
11     </div>
12   );
13 }
14 function UserCmp(props) {
15   console.log("user", props);
16   return <div> User </div>;
17 }

```

链式调用

```

1 import React from "react";
2 function Child(props) {
3   return <div>Child</div>;
4 }
5 const foo = Cmp => props => {
6   return (
7     <div style={{ background: "red" }}>
8       <Cmp {...props} />
9     </div>
10   );
11 };
12 const foo2 = Cmp => props => {
13   return (

```

```

14     <div style={{ border: "solid 1px green" }}>
15         <Cmp {...props} />
16     </div>
17 );
18 };
19 const Foo = foo2(foo(child));
20 export default function HocPage() {
21     return (
22     <div>
23         HocPage
24         <Foo />
25     </div>
26 );
27 }

```

装饰器写法

高阶组件本身是对装饰器模式的应用，自然可以利用ES7中出现的装饰器语法来更优雅的书写代码。

cra项目配置装饰器方法：

- npm run eject (如果是直接down下来的代码，并且有改动，先commit本地代码)
- 配置package.json (如果不会配置，直接看提供的package.json代码)

```

1 "babel": {
2     "presets": [
3         "react-app"
4     ],
5     "plugins": [
6         [
7             "@babel/plugin-proposal-decorators",
8             {
9                 "legacy": true
10            }
11        ]
12    ]
13 }

```

- 安装装饰器插件 npm install @babel/plugin-proposal-decorators --save-dev
- 如果介意vscode的warning， vscode设置里加上
javascript.implicitProjectConfig.experimentalDecorators": true

CRA项目中默认不支持js代码使用装饰器语法，可修改后缀名为tsx则可以直接支持，cra版本高于2.1.0。

```

1 import React from "react";
2
3 const foo = Cmp => props => {
4     return (
5         <div style={{ background: "red" }}>
6             <Cmp {...props} />
7         </div>
8     );
9 };
10 const foo2 = Cmp => props => {
11     return (

```

```

12     <div style={{ border: "solid 1px green" }}>
13         <Cmp {...props} />
14     </div>
15 );
16 };
17
18 @foo2
19 @foo
20 function Child(props) {
21     return <div>Child</div>;
22 }
23 export default function HocPage() {
24     return (
25         <div>
26             HocPage
27             <Foo />
28         </div>
29     );
30 }

```

组件复合-Composition

复合组件给予你足够的敏捷去定义自定义组件的外观和行为，这种方式更明确和安全。如果组件间有公用的非UI逻辑，将它们抽取为JS模块导入使用而不是继承它。

基本使用

不具名

```

1 // /pages/Layout.js
2 import React, { Component } from "react";
3 export default class Layout extends Component {
4     componentDidMount() {
5         const { title = "商城" } = this.props;
6         document.title = title;
7     }
8     render() {
9         const { children, title = "商城" } = this.props;
10        return (
11            <div style={{ background: "yellow" }}>
12                <p>{title}</p>
13                {children.btns ? children.btns : children}
14                <TabBar />
15            </div>
16        );
17    }
18 }
19 function TabBar(props) {
20     return <div> TabBar</div>;
21 }
22

```

```

1 //pages/Home.js
2 import React from 'react'
3 import { Consumer } from '../AppContext';

```

```

4
5 export default function Home() {
6   return (
7     <div>
8       <Consumer>
9         {((ctx)=><HomeCmp {...ctx}>)}
10      </Consumer>
11    </div>
12  )
13}
14
15
16 function HomeCmp(props){
17   const {home,user}=props;
18   const { carsouel = [] } = home
19   const {isLogin,userName}=user;
20   return (
21     <Layout title="首页">
22       <div>
23         <div>{isLogin ? userName : '未登录'}</div>
24         {
25           carsouel.map((item, index) => {
26             return <img key={'img' + index} src={item.img} />
27           })
28         }
29       </div>
30     </Layout>
31   )
32 }
33

```

传个对象进去就是具名插槽

```

1 import React from "react";
2 import { Consumer } from "../AppContext";
3 import Layout from "./Layout";
4 const handleConsumer = Cmp => props => {
5   return <Consumer> {ctx => <Cmp {...ctx} {...props}></Cmp>} </Consumer>;
6 };
7 const HandleConsumer = handleConsumer(UserCmp);
8 export default function User(props) {
9   return (
10     <Layout title='用户中心'>
11       {/* <HandleConsumer /> */}
12       {{
13         btns:<button>下载</button>
14       }}
15     </Layout>
16   );
17 }
18 function UserCmp(props) {
19   const { home, user } = props
20   const { carsouel = [] } = home
21   const { isLogin, userName } = user
22   return (
23     <Layout title="用户中心">
24     {

```

```

25     {
26       btns: <button>下载</button>
27     }
28   }
29   /* <div>
30   <div>用户名: {isLogin ? userName : '未登录'}</div>
31   </div> */
32 </Layout>
33 )
34 }

```

实现一个简单的复合组件，如antd的card

```

1 import React, { Component } from 'react'
2 function Card(props) {
3   return <div xu="card">
4   {
5     props.children
6   }
7   </div>
8 }
9 function Formbutton(props) {
10  return <div className="Formbutton">
11    <button onClick={props.children.defaultBtns.searchClick}>默认查询</button>
12    <button onClick={props.children.defaultBtns.resetClick}>默认重置</button>
13  {
14    props.children.btns.map((item, index) => {
15      return <button key={'btn' + index} onClick={item.onClick}>
{item.title}
16      </button>
17      })
18    }
19    </div>
20  }
21  export default class CompositionPage extends Component {
22    render() {
23      return (
24        <div>
25          <Card>
26            <p>我是内容</p>
27          </Card>
28          CompositionPage
29          <Card>
30            <p>我是内容2</p>
31          </Card>
32          <Formbutton>
33          {{
34            /* btns: (
35              <>
36                <button onClick={() => console.log('enn')}>查询</button>
37                <button onClick={() => console.log('enn2')}>查询2</button>
38              </>
39            ) */
40            defaultBtns: {
41              searchClick: () => console.log('默认查询'),
42              resetClick: () => console.log('默认重置')
43            },

```

```

44     btns: [
45     {
46       title: '查询',
47       onClick: () => console.log('查询')
48     }, {
49       title: '重置',
50       onClick: () => console.log('重置')
51     }
52   ]
53 }
54 </Formbutton>
55 </div>
56 )
57 }
58 }

```

HOOKS

Hook (<https://reactjs.org/docs/hooks-overview.html>) 是React16.8一个新增项，它可以让你在不编写 class 的情况下使用 state 以及其他 React 特性。

Hooks的特点：

- 使你在无需修改组件结构的情况下复用状态逻辑
- 可将组件中相互关联的部分拆分成更小的函数，复杂组件将变得更容易理解
- 更简洁、更易理解的代码

状态钩子State Hook

- 创建HookPage.js

```

1 import React, { useState, useEffect } from "react";
2 export default function HookPage() {
3   // const [date, setDate] = useState(new Date());
4   const [counter, setCounter] = useState(0);
5   return (
6     <div>
7       <h1>HookPage</h1>
8       <p>{useClock().toLocaleTimeString()}</p>
9       <p onClick={() => setCounter(counter + 1)}>{counter}</p>
10    </div>
11  );
12 }
13 //自定义 Hook 是一个函数，其名称以“use”开头，函数内部可以调用其他的 Hook。
14 function useClock() {
15   const [date, setDate] = useState(new Date());
16   useEffect(() => {
17     console.log("useEffect");
18     const timer = setInterval(() => {
19       setDate(new Date());
20     }, 1000);
21     return () => clearInterval(timer);
22   }, []);
23   return date;
24 }

```

更新函数类似setState，但它不会整合新旧状态

- 声明多个状态变量

```
1 import React, { useState, useEffect } from "react";
2 import FruitList from "../components/FruitList";
3 // import FruitAdd from "../components/FruitAdd";
4 export default function HookPage() {
5     // const [date, setDate] = useState(new Date());
6     const [counter, setCounter] = useState(0);
7     const [fruits, setFruits] = useState(["apple", "banana"]);
8     return (
9         <div>
10             <h1>HookPage</h1>
11             <p>{useClock().toLocaleTimeString()}</p>
12             <p onclick={() => setCounter(counter + 1)}>{counter}</p>
13             <FruitList fruits={fruits} setFruits={setFruits} />
14         </div>
15     );
16 }
//自定义 Hook 是一个函数，其名称以 “use” 开头，函数内部可以调用其他的 Hook。
17 function useClock() {
18     const [date, setDate] = useState(new Date());
19     useEffect(() => {
20         console.log("useEffect");
21         const timer = setInterval(() => {
22             setDate(new Date());
23         }, 1000);
24         return () => clearInterval(timer);
25     }, []);
26     return date;
27 }
28
29
```

```
1 import React from "react";
2 export default function FruitList({ fruits, setFruits }) {
3     const delFruit = delIndex => {
4         const tem = [...fruits];
5         tem.splice(delIndex, 1);
6         setFruits(tem);
7     };
8     return (
9         <ul>
10             {fruits.map((item, index) => (
11                 <li key={item} onClick={() => delFruit(index)}>
12                     {item}
13                     </li>
14                 )));
15         </ul>
16     );
17 }
```

```
1 import React, { useState } from "react";
2 export default function FruitAdd({ fruits, addFruit }) {
3   const [name, setName] = useState("");
4   return (
5     <div>
6       <input value={name} onChange={event => setName(event.target.value)} />
7       <button onClick={() => addFruit(name)}>add</button>
8     </div>
9   );
10 }
```

副作用钩子 Effect Hook总结:

useEffect 给函数组件增加了执行副作用操作的能力。副作用 (Side Effect) 是指一个 function 做了和本身运算返回值无关的事，比如：修改了全局变量、修改了传入的参数、甚至是 console.log()，所以 ajax 操作，修改 dom 都是算作副作用。React 保证了每次运行 effect 的同时，DOM 都已经更新完毕。

- 异步数据获取，更新HooksTest.js

```
1 import { useEffect } from "react";
2 useEffect(()=>{
3   setTimeout(() => {
4     setFruits(['香蕉', '西瓜'])
5   }, 1000);
6 })
```

测试会发现副作用操作会被频繁调用

- 设置依赖

```
1 // 设置空数组意为没有依赖，则副作用操作仅执行一次
2 useEffect(()=>{...}, [])
```

如果副作用操作对某状态有依赖，务必添加依赖选项

```
1 import React, { useState } from "react";
2 export default function FruitAdd({ fruits, addFruit }) {
3   const [name, setName] = useState("");
4   return (
5     <div>
6       <input value={name} onChange={event => setName(event.target.value)} />
7       <button onClick={() => addFruit(name)}>add</button>
8     </div>
9   );
10 }
```

- 清除工作：有一些副作用是需要清除的，清除工作非常重要的，可以防止引起内存泄露

```

1  useEffect(() => {
2    const timer = setInterval(() => {
3      console.log('msg');
4    }, 1000);
5    return function(){
6      clearInterval(timer);
7    }
8  }, []);

```

组件卸载后会执行返回的清理函数

useReducer

reducer 就是一个纯函数，接收旧的 state 和 action，返回新的 state。

useReducer是useState的可选项，常用于组件有复杂状态逻辑时，类似于redux中reducer概念。

- 水果列表状态维护

```

1  import React, { useEffect, useReducer } from "react";
2  import FruitList from "../components/FruitList";
3  import FruitAdd from "../components/FruitAdd";
4  function fruitReducer(state = [], action) {
5    switch (action.type) {
6      case "replace":
7      case "init":
8        return [...action.payload];
9      case "add":
10        return [...state, action.payload];
11      default:
12        return state;
13    }
14  }
15  export default function UseReducerPage() {
16    const [fruits, dispatch] = useReducer(fruitReducer, []);
17    useEffect(() => {
18      setTimeout(() => {
19        dispatch({ type: "init", payload: ["apple", "banana"] });
20      }, 1000);
21      return () => {};
22    }, []);
23    return (
24      <div>
25        <h1>UseReducerPage</h1>
26        <FruitAdd
27          fruits={fruits}
28          addFruit={name => dispatch({ type: "add", payload: name })}
29        />
30        <FruitList
31          fruits={fruits}
32          setFruits={newList => dispatch({ type: "init", payload: newList
})}>
33        />
34      </div>
35    );
36  }

```

useContext

useContext用于在快速在函数组件中导入上下文。

```
1 import React, { useContext } from "react";
2 import { Context } from "../AppContext";
3 export default function UseContextPage() {
4   const ctx = useContext(Context);
5   const { name } = ctx.user;
6   return (
7     <div>
8       <h1>UseContextPage</h1>
9       <p>{name}</p>
10    </div>
11  );
12}
```

hook规则

- 只在最顶层使用 Hook，不要在循环，条件或嵌套函数中调用 Hook。

```
1 //下面这些用法都是错误的
2 if (counter % 2) {
3   const [counter, setCounter] = useState(0);
4 }
5 if (counter % 2) {
6   useEffect(() => {
7     setCounter(100);
8   });
9 }
```

如果我们想要有条件地执行一个 effect，可以将判断放到 Hook 的内部:

```
1 useEffect(() => {
2   //把条件判断放在effect内部
3   if (counter % 2) {
4     setCounter(100);
5   }
6 });
```

我们在单个组件中可以使用多个state hook或者effect hook，那么 React 怎么知道哪个 state 对应哪个useState？答案是 React 靠的是 Hook 调用的顺序。因为我们的示例中，Hook 的调用顺序在每次渲染中都是相同的，所以它能够正常工作。只要 Hook 的调用顺序在多次渲染之间保持一致，React 就能正确地将内部 state 和对应的 Hook 进行关联。

只在 React 函数中调用 Hook。不要在普通的 JavaScript 函数中调用 Hook：要在 React 的函数组件中调用 Hook；在自定义 Hook 中调用其他 Hook。

生命周期

React v16.0前的生命周期

大部分团队不见得会跟进升到16版本，所以16前的生命周期还是很有必要掌握的，何况16也是基于之前的修改

第一个是组件初始化(initialization)阶段

也就是以下代码中类的构造方法(constructor()),Test类继承了react Component这个基类，也就继承这个react的基类，才能有render(),生命周期等方法可以使用，这也说明为什么函数组件不能使用这些方法的原因

super(props)用来调用基类的构造方法(constructor()),也将父组件的props注入给子组件，供子组件读取(组件中props只读不可变，state可变)。而constructor()用来做一些组件的初始化工作，如定义this.state的初始内容。

```
1 import React,{Component} from 'react'  
2  
3 class Test extends Component{  
4     constructor(props){  
5         super(props);  
6     }  
7 }
```

第二个是组件的挂载(Mounting)阶段

此阶段分为componentWillMount,render,componentDidMount三个时期

- componentWillMount

在组件挂载到DOM前调用，且只会被调用一次，在这边调用this.setState不会引起组件重新渲染，也可以把写在这边的内容提前到constructor()中，所以项目中很少使用

- render

根据组件的props和state(无两者的重传递和重赋值，论值是否发生变化，都可以引起组件重新render),return一个React元素(描述组件，即UI)，不负责组件实际渲染工作，之后由React自身根据此元素去渲染出页面DOM。render是纯函数 (Pure function：函数的返回结果只依赖于它的参数；参数执行过程里面没有副作用),不能在里面执行this.setState,会有改变组件状态的副作用。

- componentDidMount

组件挂载到DOM后调用，且只会被调用一次

第三个组件的更新(update)阶段

在讲述此阶段前需要先明确下react组件更新机制。setState引起的state更新或父组件重新render引起的props更新，更新后的state和props相对以前无论是否有变化，都将引起子组件的更新render。

造成组件更新有两类(三种)情况：

- 1.父组件重新render

父组件重新render引起子组件重新render的情况有两种

a.直接使用，每当父组件重新render导致的重传props，子组件将直接跟着重新渲染，无论props是否有变化。可通过shouldComponentUpdate方法优化。

```

1 class Child extends Component{
2     shouldComponentUpdate(nextProps){//应该使用这个方法，否则无论props是否有
3         变化都将会导致组件跟着重新渲染
4         if(nextProps.someThings === this.props.someThings){
5             return false
6         }
7         render(){
8             return <div>{this.props.someThings}</div>
9         }
10    }

```

b.在componentWillReceiveProps方法中，将props转换成自己的state

```

1 class Child extends Component{
2     constructor(props){
3         super(props);
4         this.state={
5             someThings:props.someThings
6         }
7     }
8     componentWillReceiveProps(nextProps){//父组件重传props时会调用这个方法
9         this.setState({
10             someThings:nextProps.someThings
11         })
12     }
13     render(){
14         return<div>{this.state.someThings}</div>
15     }
16 }

```

根据官网的描述

在该函数(componentWillReceiveProps)中调用this.setState()将不会引起第二次渲染

是因为componentWillReceiveProps中判断props是否变化了，若变化了，this.setState将引起state变化，从而引起render，此时没必要再做第二次因重传props引起的render了，不然重复做一样的渲染了。

- 2.组件本身调用setState,无论state有没有变化。可通过shouldComponentUpdate方法优化

```

1 class Child extends Component{
2     constructor(props){
3         super(props);
4         this.state={
5             someThings: 1
6         }
7     }
8     shouldComponentUpdate(nextProps){//应该使用这个方法，否则无论state是否有
9         变化都将会导致组件重新渲染
10        if(nextProps.someThings === this.state.someThings){
11            return false
12        }
13
14        handleClick={()=>{//虽然调用了setState,但state并无变化
15            const preSomeThings = this.state.someThings;

```

```

16     this.setState({
17       someThings: preSomeThings
18     })
19   }
20   render(){
21     return <div onClick={this.handleClick}>{this.state.someThings}
22   }
23 }

```

此阶段分为

componentWillReceiveProps,shouldComponentUpdate,componentWillUpdate,render,componentDidUpdate

- componentWillReceiveProps(nextProps)

此方法只调用于props引起的组件更新过程中，参数nextProps是父组件传给当前组件的新props。但父组件render方法的调用不能保证重传给当前组件的props是有变化的，所以在此方法中根据nextProps和this.props来查明重传的props是否改变，以及如果改变了要执行啥，比如根据新的props调用this.setState触发当前组件的重新render

- shouldComponentUpdate(nextProps,nextState)

此方法通过比较nextProps,nextState及当前组件的this.props,this.state,返回true时当前组件将继续执行更新过程，返回false则当前组件更新停止，以此用来减少组件的不必要渲染，优化组件性能

ps:这边可以看出，就算componentWillReceiveProps()中执行了this.setState,更新了state，但在render前（如shouldComponentUpdate, componentWillUpdate),this.state依然指向更新前的state，不然nextState及当前组件的this.state的对比就一直是true了

- componentWillUpdate(nextProps,nextState)

此方法在调用render方法前执行，在这边可执行一些组件更新发生前的工作，一般较少用

- render

- componentDidUpdate(prevProps,prevState)

此方法在组件更新后被调用，可以操作组件更新的Dom,prevProps和prevState这两个参数指的组件更新前的props和state

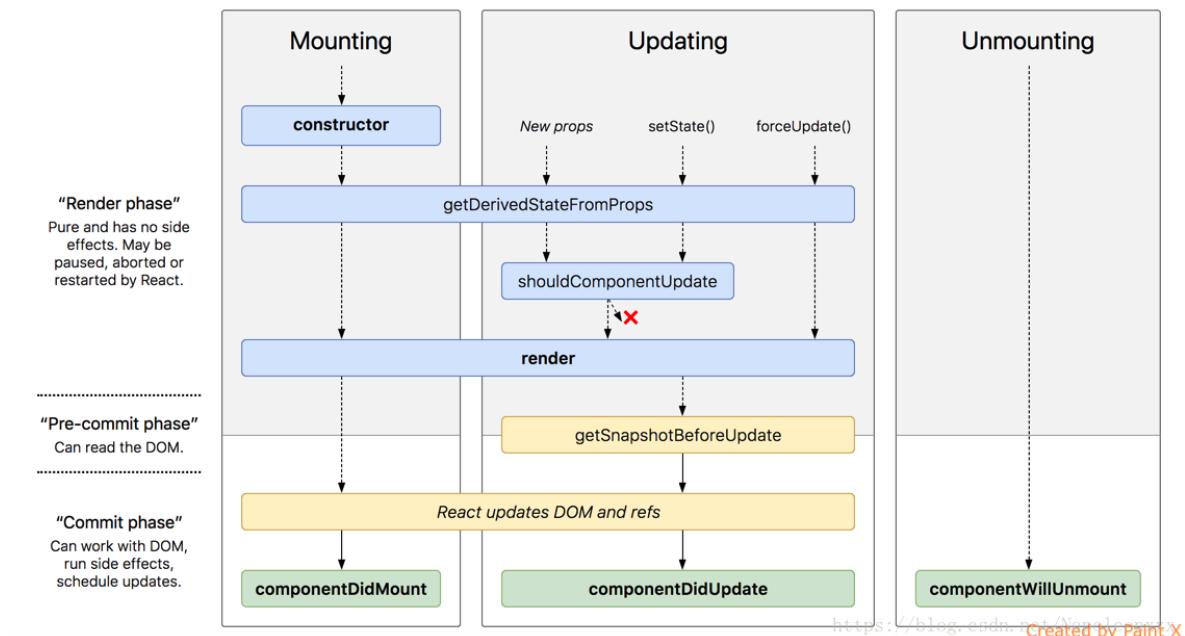
卸载阶段

此阶段只有一个生命周期的方法：componentWillUnmount

- componentWillUnmount

此方法在组件被卸载前调用，可以在这里执行一些清理工作，比如清除组件中使用的定时器，清除componentDidMount中手动创建的Dom元素等，以避免引起内存泄漏

Reactv16.4的生命周期



变更缘由

原来 (Reactv16.0前) 的生命周期在Reactv16推出的Fiber之后就不适合了，因为如果要开启async rendering,在render函数之前的所有函数，都可能被执行多次。

原来(React v16.0前)的生命周期有哪些是在render前执行的呢？

- `componentWillMount`
- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`

如果开发者开了async rendering,而且又在以上这些render前执行的生命周期方法做AJAX请求的话，那AJAX将被无谓地多次调用。。。明显不是我们期望的结果。而且在`componentWillMount`里发起AJAX，不管多快得到结果也赶不上首次render,而且`componentWillMount`在服务器端渲染也会被调用到(当然，也许这是预期的结果)，这样的IO操作放在`componentDidMount`里更合适

禁止不能用比劝导开发者不要这样用的效果更好，所以除了`shouldComponentUpdate`,其他在render之前只做了无副作用的操作，而且能做的操作局限在根据props和state决定新的state

React v16.0刚推出的时候，是增加了一个`componentDidCatch`生命周期函数，这只是一个增量式修改，完全不影响原有生命周期函数；但是，到了React v16.3,发生了大改动，引入了两个新的生命周期函数。

首先，async render不是那种服务端渲染，比如发异步请求到后台返回newState甚至新的html，这里的async render还是限制在React作为一个View框架的View层本身。

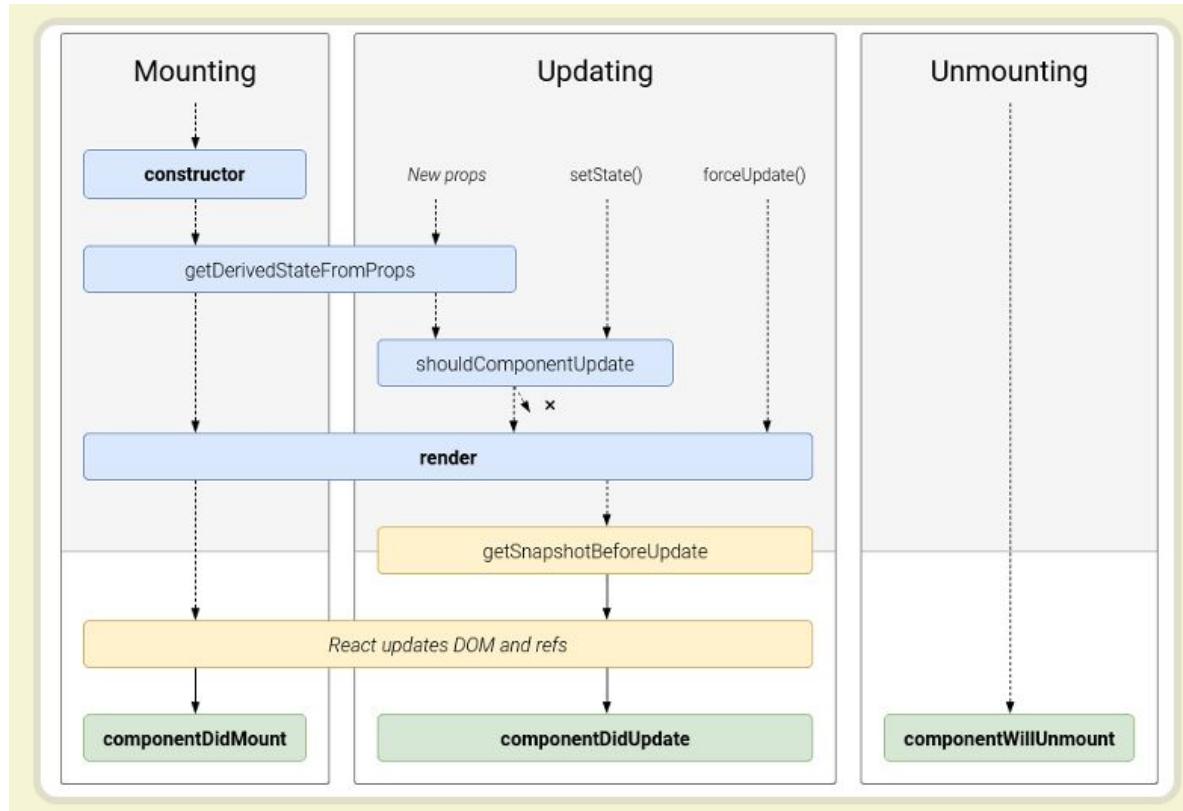
通过进一步观察可以发现，预废弃的三个生命周期函数都发生在虚拟dom的构建期间，也就是render之前。在将来的React 17中，在dom真正render之前，React中的调度机制可能会不定期的去查看有没有更高优先级的任务，如果有，就打断当前的周期执行函数(哪怕已经执行了一半)，等高优先级任务完成，再回来重新执行之前被打断的周期函数。这种新机制对现存周期函数的影响就是它们的调用时机变得复杂而不可预测，这也就是为什么“UNSAFE”。

新引入了两个新的生命周期函数：`getDerivedStateFromProps`, `getSnapshotBeforeUpdate`

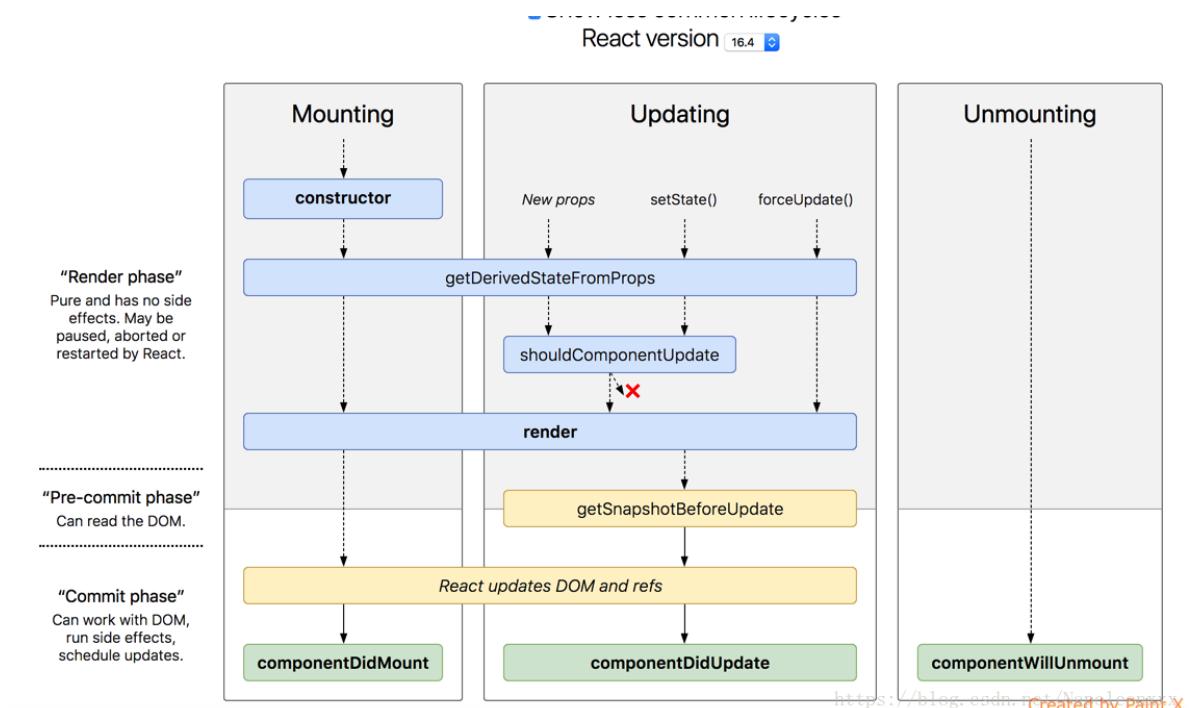
`getDerivedStateFromProps`

getDerivedStateFromProps本来（React v16.3中）是只在创建和更新（父组件引发部分），也就是不是有父组件引发的，那么getDerivedStateFromProps也不会被调用，如自身setState引发或者forceUpdate引发。getDerivedStateFromProps应该是个纯函数，没有副作用，不能通过this。在getDerivedStateFromProps中，在条件限制下(if/else)调用setState。如果不设任何条件setState，这个函数超高的调用频率，不停的setState，会导致频繁的重绘，既有可能产生性能问题，同时也容易产生bug。

Reactv16.3的生命周期图



React v16.3，这样理解起来有点乱，在React v16.4中改正了这一点，让getDerivedStateFromProps无论是Mounting还是Updating，也无论是因为什么引起的Updating，全部都会被调用，具体可以看React v16.4的生命周期图。



static getDerivedStateFromProps(props,state)在组件创建时和更新时的render之前调用，它应该返回一个对象来更新状态，或者返回null来不更新任何内容。getDerivedStateFromProps生命周期是为了替代componentWillReceiveProps存在的，所以需要使用componentWillReceiveProps的时候，可以考虑使用getDerivedStateFromProps来进行替代了。

两者的参数是不相同的，而getDerivedStateFromProps是一个静态函数，也就是这个函数不能通过this访问到class的属性，也并不推荐直接访问属性，而是应该通过参数提供的nextProps以及prevState来进行判断，根据新传入的props来映射到state

需要注意的是，如果props传入的内容不需要影响到你的state,那么就需要返回一个null,这个返回值是必须的，尽量将其写到函数末尾。

在看一个例子，这个例子是一个颜色选择器，这个组件能选择相应的颜色并显示，同时它能根据传入prop值显示颜色。

```
1 class ColorPicker extends React.Component {
2     state = {
3         color: '#000000'
4     }
5     static getDerivedStateFromProps (props, state) {
6         if (props.color !== state.color) {
7             return {
8                 color: props.color
9             }
10        }
11        return null
12    }
13    ... // 选择颜色方法
14    render () {
15        .... // 显示颜色和选择颜色操作
16    }
17 }
18 }
```

现在我们可以这个颜色选择器来选择颜色，同时我们能传入一个颜色值并显示。但是这个组件有一个bug，如果我们传入一个颜色值后，再使用组件内部的选择颜色方法，我们会发现颜色不会变化，一直是传入的颜色值。

这是使用这个生命周期的一个常见 bug。为什么会发生这个 bug 呢？在开头有说到，在 React 16.4^ 的版本中 `setState` 和 `forceUpdate` 也会触发这个生命周期，所以内部 state 变化后，又会走 `getDerivedStateFromProps` 方法，并把 state 值更新为传入的 prop。

接下里我们来修复这个bug。

```
1 class ColorPicker extends React.Component {
2     state = {
3         color: '#000000',
4         prevPropColor: ''
5     }
6     static getDerivedStateFromProps (props, state) {
7         if (props.color !== state.prevPropColor) {
8             return {
9                 color: props.color
10                prevPropColor: props.color
11            }
12        }
13        return null
14    }
15 }
```

```

14     }
15     ... // 选择颜色方法
16     render () {
17       .... // 显示颜色和选择颜色操作
18     }
19   }
20

```

通过保存一个之前 prop 值，我们就可以在只有 prop 变化时才去修改 state。这样就解决上述的问题。

这里小结下 `getDerivedStateFromProps` 方法使用的注意点：

- 在使用此生命周期时，要注意把传入的 prop 值和之前传入的 prop 进行比较。
- 因为这个生命周期是静态方法，同时要保持它是纯函数，不要产生副作用

使用派生状态遇到的常见bug

“受控”和“不受控制”的术语通常指的是表单输入，但他们还可以描述任何组件数据的位置。作为props传递进组件的数据可以被认为是**受控的**（因为父组件控制数据）。只存在于内部状态的数据可以被认为是**不受控制的**（因为父类不能直接更改它）。

派生状态最常见的错误是混合这两个；当一个派生状态值也通过 `setState` 被更新时，数据就没有单一的真实来源。

当这些约束被改变时，就会出现问题。这通常有两种形式。让我们来看看这两种情况。

反模式：无条件得使用props对state赋值

一个常见的误解是，当props“改变”时，`getDerivedStateFromProps` 和 `componentWillReceiveProps` 才会被调用。事实上，只要父组件重新渲染，这些生命周期函数就会被调用，不管这些props是否与以前“不同”。正因为如此，使用任何一个去 无条件地覆盖state都是不安全的。**这样做会导致状态更新丢失。**

```

1 class EmailInput extends Component {
2   state = { email: this.props.email };
3
4   render() {
5     return <input onChange={this.handleChange} value={this.state.email} />;
6   }
7
8   handleChange = event => {
9     this.setState({ email: event.target.value });
10  };
11
12  componentWillReceiveProps(nextProps) {
13    // This will erase any local state updates!
14    // Do not do this.
15    this.setState({ email: nextProps.email });
16  }
17}

```

这个组件可能看起来不错。state被初始化为由props指定的值，并且在输入 `<input>` 时实时更新 state。但是，如果我们的组件的父类重新渲染，我们在 `<input>` 输入的任何东西都将丢失！即使我们在重置之前比较 `nextProps.email !== this.state.email`，也一样。

在这个简单的例子中，为了解决这个问题，必须通过添加 `shouldComponentUpdate`，并判断只有 `prop email` 发生改变时才重新渲染。然而在实际情况中，组件通常接受多个 `prop`；任何一个 `prop` 发生改变都会导致重新运行和不正确的重置。而且对于 `Function` 和 `object` 类型的 `prop`，`shouldComponentUpdate` 很难判断是否发生了实质性的变化。`shouldComponentUpdate` 最好作为性能优化使用，而不是为了确保派生状态的正确性。

反模式：当 `props` 改变时清除 state

继续上边的例子，我们判断只有当 `props.email` 发生改变时才去执行更新，以此来避免状态被清除：

```
1 class EmailInput extends Component {
2   state = {
3     email: this.props.email
4   };
5
6   componentWillReceiveProps(nextProps) {
7     // Any time props.email changes, update state.
8     if (nextProps.email !== this.props.email) {
9       this.setState({
10       email: nextProps.email
11     });
12   }
13 }
14
15 // ...
16 }
```

现在出现了一个微妙的问题。想象一个使用上述输入组件的密码管理器应用程序。当使用相同的电子邮件在两个帐户的详细信息之间导航时，输入将无法重置。这是因为传递给组件的属性值对于两个帐户都是相同的！这对用户来说是一个惊喜，因为一个账户的未保存的变更似乎会影响到其他的帐户，这些帐户碰巧共享相同的电子邮件

这种设计从根本上来说是有缺陷的，但这却是一个极易犯的错误。幸运的是，有两种替代方案可以更好地工作。两种方案的关键在于——对于任何数据，您都需要确保只有一个组件作为实际的来源，并避免在其他组件中复制它。现在来看一下这两种方案。

首选方案

推荐：完全受控组件

避免上面提到的问题的一种方法是彻底从组件中删除状态。如果“邮件地址”只是作为属性存在，那么我们就不必担心与状态的冲突。我们甚至可以把 `EmailInput` 转换成轻量的函数组件：

```
1 function EmailInput(props) {
2   return <input onChange={props.onChange} value={props.email} />;
3 }
```

这种方法简化了组件的实现，但是如果我们仍然想要储存一个中间值（draft value），那么父表单组件现在就只能手动完成这件事。

推荐：有“key”的完全非受控组件

另一种方案是，让组件完全拥有中间的 `email` 状态（draft `email` state）。在这个示例中，我们的组件仍然接收一个属性用来设置 `email` 的初始值，但是却无法接收这个属性之后的变化：

```

1 class EmailInput extends Component {
2   state = { email: this.props.defaultEmail };
3
4   handleChange = event => {
5     this.setState({ email: event.target.value });
6   };
7
8   render() {
9     return <input onChange={this.handleChange} value={this.state.email} />;
10  }
11}

```

为了在移动到另一项（如密码管理器场景）时可以重新赋值，我们可以使用“key”这个React的特殊属性。当一个“key”发生变化时，React将创建一个新的组件实例，而不是更新当前的一个实例。“key”通常用于动态列表，但在这里也很有用。在我们的例子中，我们可以使用用户ID在新用户被选中时重新创建“EmailInput”：

```

1 <EmailInput
2   defaultEmail={this.props.user.email}
3   key={this.props.user.id}
4 />

```

每当ID改变时，`EmailInput` 将被重新创建，它的状态将被重置为最新的 `defaultEmail` 值。使用这种方法，您不需要向每个输入项添加 `key`。把 `key` 放在整个表单上可能更有意义。每次改变时，表单中的所有组件都将用一个新初始化的状态重新创建。

getSnapshotBeforeUpdate

`getSnapshotBeforeUpdate()`被调用于`render`之后，它的要点是在于触发时，DOM还没有更新，可以读取但无法使用DOM的时候。它使您的组件可以在可能更改之前从DOM捕获一些信息(例如滚动位置)。此生命周期返回的任何值都将作为参数传递给`componentDidUpdate()`.

```

1 class ScrollingList extends React.Component {
2   constructor(props) {
3     super(props);
4     this.listRef = React.createRef();
5   }
6
7   getSnapshotBeforeUpdate(prevProps, prevState) {
8     // 我们是否在 list 中添加新的 items ?
9     // 捕获滚动位置以便我们稍后调整滚动位置。
10    if (prevProps.list.length < this.props.list.length) {
11      const list = this.listRef.current;
12      return list.scrollHeight - list.scrollTop;
13    }
14    return null;
15  }
16
17  componentDidUpdate(prevProps, prevState, snapshot) {
18    // 如果我们 snapshot 有值，说明我们刚刚添加了新的 items,
19    // 调整滚动位置使得这些新 items 不会将旧的 items 推出视图。
20    // (这里的 snapshot 是 getSnapshotBeforeUpdate 的返回值)
21    if (snapshot !== null) {
22      const list = this.listRef.current;
23      list.scrollTop = list.scrollHeight - snapshot;

```

```

24     }
25   }
26
27   render() {
28     return (
29       <div ref={this.listRef}>{/* ...contents... */}</div>
30     );
31   }
32 }

```

在上述示例中，重点是从 `getSnapshotBeforeUpdate` 读取 `scrollHeight` 属性，因为“render”阶段生命周期（如 `render`）和“commit”阶段生命周期（如 `getSnapshotBeforeUpdate` 和 `componentDidUpdate`）之间可能存在延迟

React组件化

怎么使css模块化？

1. 创建xx.module.css

2. 组件中写法

```

1 import style from "./xx.module.css";
2
3 <div className={style.box}></div>

```

使用第三方组件

- 安装: `npm i antd -S`
- 配置按需加载

安装react-app-rewired取代react-scripts,可扩展webpack的配置，类似vue.config.js

由于新的 react-app-rewired@2.x (<https://github.com/timarney/react-app-rewired#alternatives>) 版本的关系，你还需要安装 customize-cra (<https://github.com/arackaf/customize-cra>)。babel-plugin-import (<https://github.com/ant-design/babel-plugin-import>)是一个用于按需加载组件代码和样式的 babel 插件（原理 (<https://ant.design/docs/react/getting-started-cn#E6%8C%89%E9%9C%80%E5%8A%A0%E8%BD%BD>)）。

`npm i react-app-rewired customize-cra babel-plugin-import -D`

```

1 const { override, fixBabelImports } = require('customize-cra');
2
3 //override返回一个函数，该函数返回对象将作为webpack的配置对象
4 module.exports = override(
5   fixBabelImports('import', {
6     libraryName: 'antd',
7     libraryDirectory: 'es',
8     style: 'css',
9   })
10 );
11
12
13 //修改package.json
14 "scripts": {
15   "start": "react-app-rewired start",

```

```
16     "build": "react-app-rewired build",
17     "test": "react-app-rewired test",
18     "eject": "react-app-rewired eject"
19 },
```

- 使用组件

```
1 | import {Button} from "antd";
```

支持装饰器配置

```
npm install -D @babel/plugin-proposal-decorators
```

```
1 | //配置完成后记得重启下
2 | const { addDecoratorsLegacy } = require("customize-cra");
3 | module.exports = override(
4 |   ...,
5 |   addDecoratorsLegacy() //配置装饰器
6 | );
```

容器组件vs展示组件

基本原则：容器组件负责数据获取，展示组件负责根据props显示信息

优势：更小，更专注，重用性高，高可用，易于测试，性能更好

```
1 | import React, { Component, PureComponent } from "react";
2 |
3 | // 容器组件
4 | export default class CommentList extends Component {
5 |   constructor(props) {
6 |     super(props);
7 |     this.state = {
8 |       comments: []
9 |     };
10 |   }
11 |   componentDidMount() {
12 |     setInterval(() => {
13 |       // immutable.js
14 |       this.setState({
15 |         comments: [
16 |           { body: "react is very good", author: "facebook" },
17 |           { body: "vue is very good", author: "youyuxi" }
18 |         ]
19 |       });
20 |     }, 1000);
21 |   }
22 |   render() {
23 |     return (
24 |       <div>
25 |         {this.state.comments.map((c, i) => (
26 |           <Comment key={i} {...c} />
27 |         )));
28 |       </div>
29 |     );
30 |   }
31 | }
```

```
31 }
32 // 展示组件
33 function Comment({body, author}) {
34 console.log("render comment");
35
36     return (
37         <div>
38             <p>{body}</p>
39             <p> --- {author}</p>
40         </div>
41     );
42 }
```

PureComponent

定制了shouldComponentUpdate后的component

```
1 class Comp extends React.PureComponent{}
```

由于比较方式是浅比较，注意传值方式，值类型或者地址不变的且权限属性变化的引用类型才能享受该特性

React.memo

React 16.6.0使用React.memo让函数式的组件也有PureComponent的功能

```
1 const Joke = React.memo(()=>{
2     return (
3         <div>{this.props.value || 'loading...'}</div>
4     )
5 })
```

高阶组件

高阶组件HOC(Higher-Order Components)是React中重要组件逻辑的高阶技术，它不是react的api，而是一种组件增强模式。高阶函数是一个函数，它返回另外一个组件，**产生的组件可以对被包装组件属性进行包装，也可以重写部分生命周期**

```
1 const withKaiba=(Component)=>{
2     const NewComponent=(props)=>{
3         return <Component name="高阶组件" {...props}/>
4     }
5     return NewComponent;
6 }
```

上面的withKaiba组件，其实是代理了Component，只是多传递了一个name参数

链式调用

```
1 import React, { Component } from 'react';
2 import { Button } from "antd"
3
4 //创建一个函数接收一个组件返回另一个组件
5 function withKaiKeBa(Comp) {
6     return (props) => (<Comp {...props} name="高阶组件"></Comp>)
```

```

7 }
8
9 //功能: 日志记录
10 function withLog(Comp) {
11     class NewComponent extends React.Component{
12         render(){
13             return <Comp {...props}></Comp>
14         }
15     }
16     componentDidMount(){
17         console.log("didMount",this.props);
18     }
19 }
20
21 function App() {
22     return (
23         <div className="App">
24             <Button type="primary">Button</Button>
25         </div>
26     );
27 }
28
29 export default withKaiKeBa(withLog(App))

```

装饰器写法

ES7中有一个优秀的语法-装饰器，可使代码更简洁

安装:

```
1 | npm i @babel/plugin-proposal-decorators -D
```

配置:

```

1 const { addDecoratorsLegacy } = require('customize-cra');
2 module.exports=override(
3     ...,
4     addDecoratorsLegacy()
5 )

```

应用:

```

1 import React, { Component } from 'react';
2 import { Button } from "antd"
3
4 //创建一个函数接收一个组件返回另一个组件
5 function withKaiKeBa(Comp) {
6     return (props) => (<Comp {...props} name="高阶组件"></Comp>)
7 }
8
9 //功能: 日志记录
10 function withLog(Comp) {
11     class NewComponent extends React.Component{
12         render(){
13             return <Comp {...props}></Comp>
14         }
15     }
16 }
17
18 export default withKaiKeBa(withLog(App))

```

```

15    }
16    componentDidMount(){
17      console.log("didMount",this.props);
18    }
19  }
20
21 @withKaiba
22 @withLog
23 function App() {
24   return (
25     <div className="App">
26       <Button type="primary">Button</Button>
27     </div>
28   );
29 }
30
31 export default App;

```

组件复合-Composition

复合组件使我们以更敏捷的方式定义组件的外观和行为，比起继承的方式它更加明确和安全

```

1  function Dialog(props) {
2    const color = props.color || 'red';
3    return (
4      <div style={{ border: `4px solid ${color}` }}>
5        {props.children}
6        <div>{props.foo("这个内容是foo传过来的")}</div>
7        <div>
8          {props.footer}
9        </div>
10       </div>
11     )
12   }
13
14
15
16 //welcomeDialog通过复合提供内容
17 function welcomeDialog() {
18   const footer = <button onClick={() => alert('...')}>sure</button>
19   return (
20     <Dialog color="blue" footer={footer} foo={(c) => <p>{c}</p>}>
21       <h1>欢迎光临</h1>
22       <p>感谢使用react</p>
23     </Dialog>
24   )
25 }

```

通过属性通信

```

1 <Dialog color="blue">
2   <div style={{ border: `4px solid ${color}` }}>

```

传递任意表达式均可：

```
1 <Dialog color="blue" footer={<button onClick={() =>
  alert('...')}>sure</button>} foo={(c) => <p>{c}</p>}>
```

children也是如此

```
1 <Dialog>
2   {(value)=><p>{value}</p>}
3 </Dialog>
```

编辑Children

```
1 | React.Children.map(props.children, child=>child.type==='p')
```

类似的还有React.Children.forEach,React.Children.toArray等

传递进来的child如果是vdom, 注意不能修改

```
1 | React.Children.map(props.children, child=>child.props.name='mvvm') //错误
2 | React.Children.map(props.children, child=>React.cloneElement(child,
  {name: 'mvvm'}))
```

组件跨层级通信-Context

vuejs的provide&inject模式的来源-context

这种模式下有两个角色：

- Provider:内层提供数据的组件
- Consumer: 内层获取数据的组件

使用:

```
1 //创建上下文
2 const Context = React.createContext();
3
4 const store = {
5   token: 'lily'
6 }
7 export default class ContextTest extends Component {
8   render() {
9     return (
10       <Context.Provider value={store}>
11         <div>
12           <Context.Consumer>
13             {value => <p>{store.token}</p>}
14           </Context.Consumer>
15         </div>
16       </Context.Provider>
17     )
18   }
19 }
```

Hook

升级react, react-dom至16.8以上

```
1 | npm i react react-dom -S
```

状态钩子State Hook

函数组件可以使用状态

```
1 import React,{useState} from 'react'
2
3 function FruitList({ fruits,setFruit }) {
4     return (
5         fruits.map(f => {
6             return( <li onClick={()=>setFruit(f)}>{f}</li> )
7         })
8     )
9 }
10
11 function FruitAdd(props){
12     const [pname,setPname]=useState('')
13     const onAddFruit=(e)=>{
14         if(e.key==='Enter'){
15             props.onAddFruit(pname);
16             setPname("");
17         }
18     }
19     return (
20         <div>
21             <input type="text"
22                 value={pname}
23                 onChange={e=>setPname(e.target.value)}
24                 onKeyDown={onAddFruit}
25             />
26         </div>
27     )
28 }
29
30 export default function HookTest() {
31     //useState参数是状态的初始值
32     //返回一个数组，第一个元素是状态变量，第二个元素是状态变更函数
33     const [fruit, setFruit] = useState('苹果');
34     const [fruits, setFruits] = useState(['苹果','香蕉'])
35     return (
36         <div>
37             <p>{fruit === '' ? '请选择喜爱的水果' : `你选择的是${fruit}`}</p>
38             <FruitList fruits={fruits} setFruit={setFruit}></FruitList>
39             <FruitAdd onAddFruit={pname=>setFruits([...fruits,pname])}>
40             </FruitAdd>
41         </div>
42     )
43 }
```

副作用钩子Effect Hook

函数组件执行副作用操作

- 基本使用

```
1 import React, {useEffect} from 'react'
2
3 //使用useEffect操作副作用
4 //请务必设置依赖选项，如果没有则设置空数组表示仅执行一次
5 useEffect(() => {
6     setTimeout(() => {
7         setFruits(['苹果', '香蕉'])
8     }, 1000)
9 }, []);
```

- 设置依赖

```
1 useEffect(()=>{...},[])
```

- 清除工作:有一些副作用是需要清除的，防止内存泄漏

```
1 useEffect(() => {
2     const timer = setInterval(() => {
3         console.log('应用启动了')
4     }, 1000);
5     return function () {
6         clearInterval(timer);
7     }
8 }, []);
```

useReducer

useState的可选项，常用于组件有复杂状态逻辑时，类似于redux中reducer概念

```
1 import {useReducer} from "react"
2
3 //将状态移至全局
4 function fruitReducer(state, action) {
5     switch (action.type) {
6         case 'init':
7             return action.payload;
8         case 'add':
9             return [...state, action.payload]
10        default:
11            return state
12    }
13 }
14
15 export default function HookTest() {
16     //useState参数是状态的初始值
17     //返回一个数组，第一个元素是状态变量，第二个元素是状态变更函数
18     const [fruit, setFruit] = useState('苹果');
19     // const [fruits, setFruits] = useState([]);
20
21     //参数一是相关的reducer，参数二是初始值,
22     const [fruits, dispatch] = useReducer(fruitReducer, [])
23     //使用useEffect操作副作用
24     //请务必设置依赖选项，如果没有则设置空数组表示仅执行一次
25     useEffect(() => {
26         setTimeout(() => {
```

```

27         // setFruits(['苹果', '香蕉'])
28         dispatch({ type: 'init', payload: ['苹果', '香蕉'] })
29     }, 1000)
30 }, []);
31
32 return (
33     <div>
34         <p>{fruit === '' ? '请选择喜爱的水果' : `你选择的是
35 ${fruit}`}</p>
36         <FruitAdd onAddFruit=
37             {pname=>dispatch({type: 'add', payload:pname})}></FruitAdd>
38     </div>
39 )
40 }

```

useContext

useContext用于快速在组件中导入上下文

```

1 import {useContext} from "react"
2
3 const Context = React.createContext();
4
5 export default function HookTest() {
6     //useState参数是状态的初始值
7     //返回一个数组，第一个元素是状态变量，第二个元素是状态变更函数
8     const [fruit, setFruit] = useState('苹果');
9     // const [fruits, setFruits] = useState([]);
10
11     //参数一是相关的reducer，参数二是初始值,
12     const [fruits, dispatch] = useReducer(fruitReducer, [])
13     //使用useEffect操作副作用
14     //请务必设置依赖选项，如果没有则设置空数组表示仅执行一次
15     useEffect(() => {
16         setTimeout(() => {
17             // setFruits(['苹果', '香蕉'])
18             dispatch({ type: 'init', payload: ['苹果', '香蕉'] })
19         }, 1000)
20     }, []);
21
22     return (
23         <Context.Provider value={{ fruits, dispatch }}>
24             <div>
25                 <p>{fruit === '' ? '请选择喜爱的水果' : `你选择的是
26 ${fruit}`}</p>
27                 <FruitAdd></FruitAdd>
28             </div>
29         </Context.Provider>
30     )
31
32     function FruitAdd(props) {
33         const [pname, setPname] = useState('');
34         const { dispatch } = useContext(Context)
35         const onAddFruit = (e) => {
36             if (e.key === 'Enter') {
37                 // props.onAddFruit(pname);
38             }
39         }
40     }
41 }

```

```

38         dispatch({ type: 'add', payload: pname })
39         setPname("");
40     }
41   }
42   return (
43     <div>
44       <input type="text"
45         value={pname}
46         onChange={e => setPname(e.target.value)}
47         onKeyDown={onAddFruit}
48       />
49     </div>
50   )
51 }

```

Hook相关拓展

1. 基于useReducer的方式实现异步action

```

1  function fruitReducer(state, action) {
2    switch (action.type) {
3      case 'init':
4
5        return {...state, list:action.payload};
6      case 'add':
7        return {...state, list:[...state, action.payload]};
8      case "loading_start":
9        return {...state, loading:true};
10     case "loading_end":
11        return {...state, loading:false}
12     default:
13       return state
14   }
15 }
16
17
18 //判断对象是否是Promise
19 function isPromise(obj){
20   return(
21     !!obj&&
22     (typeof obj === "object" || typeof obj ==="function") &&
23     typeof obj.then === "function"
24   )
25 }
26
27 //mock一个异步方法
28 async function asyncFetch(p){
29   return new Promise(resolve=>{
30     setTimeout(()=>{
31       reslove(p);
32     },1000)
33   })
34 }
35
36 //对dispatch函数进行封装，使其支持处理异步action
37 function wrapDispatch(dispatch){
38   return function(action){

```

```

39     if(isPromise(action.payload)){
40         dispatch({type:'loading_start'});
41         action.payload.then(v=>{
42             dispatch({type:action.type,payload:v});
43             dispatch({type:"loading_end"});
44         })
45     }else{
46         dispatch(action)
47     }
48 }
49
50
51 export default function HookTest() {
52     //useState参数是状态的初始值
53     //返回一个数组，第一个元素是状态变量，第二个元素是状态变更函数
54     const [fruit, setFruit] = useState('苹果');
55     const [{list:fruits,loading},originDispatch] = useReducer(fruitReducer,
56     {
57         list:[],
58         loading:false
59     })
60     //使用useEffect操作副作用
61     //请务必设置依赖选项，如果没有则设置空数组表示仅执行一次
62     useEffect(() => {
63         dispatch({ type: 'init', payload:asyncFetch(['苹果', '香蕉']) })
64     }, []);
65
66     return (
67         <Context.Provider value={{ fruits, dispatch }}>
68             <div>
69                 <p>{fruit === '' ? '请选择喜爱的水果' : `你选择的是${fruit}`}</p>
70                 {loading}<br>
71                     <div>数据加载中....</div>
72                 ):(<FruitList fruits={fruits} setFruit={setFruit}>
73             </FruitList>)
74         </div>
75     </Context.Provider>
76 )
77 }

```

2.Hook规则，自定义Hook

3.一推第三方hook实现

组件设计与实现

表单组件实现

- KFormTest.js

```

1 import React,{Component} from "react"
2
3 export default class KFromTest extends Component{
4     render(){
5         return(
6             <div>
7                 <Input type="text"/>
8                 <Input type="password"/>
9                 <Button>登录</Button>
10            </div>
11        )
12    }
13 }

```

- 扩展现有表单KFormTest.js

```

1 //扩展组件：扩展现有表单，提供控件包裹、事件处理、表单校验
2 function kFromCreate(Comp) {
3     return class extends Component {
4         constructor(props) {
5             super(props);
6             //数据
7             this.options = {};
8             //数据
9             this.state = {};
10        }
11
12        handleChange = e => {
13            //数据设置和校验
14            const { name, value } = e.target;
15            this.setState({
16                [name]: value
17            }, () => {
18                //单字段校验
19                this.validateFeild(name);
20            })
21
22        }
23
24
25        //焦点处理、错误获取等
26        handleFocus=e=>{
27            const field=e.target.name;
28            this.setState({
29                [field+"Focus"]:true
30            })
31        }
32
33        isFieldTouched=field=>{
34            return !!this.state[field+"Focus"]
35        }
36
37        getFieldError=field=>{
38            return this.state[field+"Message"]
39        }
40

```

```

41     validateFeild = (field) => {
42         const rules = this.options[field].rules;
43         //some里面任何一项不通过就返回true跳出，取反表示校验失败
44         const isValid = !rules.some(rule => {
45             if (rule.required) {
46                 if (!this.state[field]) {
47                     //校验失败
48                     this.setState({
49                         [field + 'Message']: rule.message
50                     })
51                     return true;
52                 }
53             }
54         })
55         return false;
56     )
57     if (isValid) {
58         this.setState({
59             [field + 'Message']: ''
60         })
61     }
62
63     return isValid;
64
65 }
66
67 validateFeilds = (cb) => {
68     const results = Object.keys(this.options).map(field =>
this.validateFeild(field));
69     const result = results.every(v => v === true);
70     cb(result, this.state);
71 }
72
73 //包装函数：接收字段名和校验选项返回一个高阶组件
74 getFieldDec = (field, option) => {
75     this.options[field] = option;//选项告诉我们如何校验
76     return InputComp => (
77         <div>
78         {
79             React.cloneElement(InputComp, {
80                 name: field,
81                 value: this.state[field] || '',
82                 onChange: this.handleChange,//执行校验设置状态
等
83                 onFocus:this.handleFocus//焦点处理
84             })
85         }
86         </div>
87     )
88 }
89 render() {
90     return <Comp {...this.props} isFieldTouched=
{this.isFieldTouched} getFieldValue={this.getFieldValue} getFieldDec=
{this.getFieldDec} validateFeilds={this.validateFeilds}></Comp>
91     }
92 }
93 }
94

```

- 应用高阶组件

```
1  @kFromCreate
2  class KForm extends Component {
3      onSubmit = () => {
4          const { validateFeilds } = this.props;
5          validateFeilds((isValid, values) => {
6              if (isValid) {
7                  alert('登陆啦');
8              } else {
9                  alert('校验失败');
10             }
11         })
12     }
13     render() {
14         const { getFieldDec, isFieldTouched, getFieldError } =
15             this.props;
16         const
17             usernameError=isFieldTouched("username")&&getFieldError("username");
18             const
19             passwordError=isFieldTouched("password")&&getFieldError("password");
20             console.log(usernameError);
21             return (
22                 <div>
23                     <FormItem
24                         validateStatus="error"
25                         help={usernameError||''}
26                     >
27                         {getFieldDec('username', {
28                             rules: [{ required: true, message: 'Please
29             input your username' }]
30                     })(<KInput prefix={<Icon type="user"/>} type="text"
31             />)}
32                     </FormItem>
33                     <FormItem
34                         validateStatus="error"
35                         help={passwordError||''}
36                     >
37                         {getFieldDec('password', {
38                             rules: [{ required: true, message: 'Please
39             input your password' }]
40                     })(<KInput type="password" prefix={<Icon
41                         type="lock"/>}>)}
42                     </FormItem>
43                     <button onClick={this.onSubmit}>登录</button>
44                 </div>
45             )
46         }
47     }
48 }
```

- FormItem(错误信息)

```
1  class FormItem extends Component{
```

```

3   render(){
4     return(
5       <div>
6         {this.props.children}
7         {this.props.help&&(
8           <p style=
9             {{color:this.props.validateStatus==='error'? 'red':'green'}}>
10            {this.props.help}
11          </p>
12        )}
13      </div>
14    )
15  }
16

```

- Input(前缀图标)

```

1 function KInput(props){
2   const {prefix,...rest}=props;
3   return(
4     <div>
5       {prefix}
6       <input {...rest}/>
7     </div>
8   )
9 }

```

弹窗类组件设计与实现

设计思路

弹窗类组件的要求弹窗内容在A处声明，却在B处展示。react中相当于弹窗内容看起来被render到一个组件里面去，实际改变的是网页上另一处的DOM结构，这个显然不符合正常逻辑。但是通过使用框架提供的特定API创建组件实例并指定挂载目标仍可完成任务。

```

1 // 常见用法如下：dialog在当前组件声明，但是却在body中另一个div中显示
2 <div class="foo">
3   <div> ... </div>
4   {
5     needDialog &&
6     <Dialog>
7       <header>Any Header</header>
8       <section>Any content</section>
9     </Dialog>
10   }
11 </div>

```

具体实现

方案1：Portal

传送门，react v16之后出现的portal可以实现内容传送功能。

范例：Dialog组件

```

1 import React,{Component} from "react";
2 import { createPortal } from "react-dom";
3
4 export default class Dialog extends Component{
5   constructor(props){
6     super(props);
7     const doc=window.document;
8     this.node=doc.createElement('div');
9     doc.body.appendChild(this.node)
10 }
11 componentWillUnmount(){
12   window.document.body.removeChild(this.node)
13 }
14
15 render(){
16   const {hideDialog}=this.props;
17   return createPortal(
18     <div className='dialog'>
19       {this.props.children}
20       {typeof hideDialog==='function' &&(
21         <button onClick={hideDialog}>关闭弹窗</button>
22       )}
23     </div>
24   ,this.node)
25 }
26 }
27
28 import React from 'react'
29
30 export default function Dialog(props) {
31   useEffect(() => {
32     const doc=window.document;
33     this.node=doc.createElement('div');
34     doc.body.appendChild(this.node)
35     return () => {
36       window.document.body.removeChild(this.node)
37     }
38   }, [])
39   const {hideDialog}=props;
40   return createPortal(
41     <div className='dialog'>
42       {this.props.children}
43       {typeof hideDialog==='function' &&(
44         <button onClick={hideDialog}>关闭弹窗</button>
45       )}
46     </div>
47   ,this.node)
48 }
49

```

```

1 //Dialog/index.css
2 .dialog {
3   position: absolute;
4   top: 0;
5   right: 0;
6   bottom: 0;

```

```
7 |     left: 0;
8 |     line-height: 30px;
9 |     width: 400px;
10|     height: 300px;
11|     transform: translate(50%, 50%);
12|     border: solid 1px gray;
13|     text-align: center;
14| }
15| }
```

方案2: unstable_renderSubtreeIntoContainer

在v16之前，实现“传送门”，要用到react中两个秘而不宣的React API

```
1 | export class Dialog2 extends React.Component {
2 |   render() {
3 |     return null;
4 |   }
5 |   componentDidMount() {
6 |     const doc = window.document;
7 |     this.node = doc.createElement("div");
8 |     doc.body.appendChild(this.node);
9 |     this.createPortal(this.props);
10|   }
11|   componentDidUpdate() {
12|     this.createPortal(this.props);
13|   }
14|   componentWillUnmount() {
15|     unmountComponentAtNode(this.node);
16|     window.document.body.removeChild(this.node);
17|   }
18|   createPortal(props) {
19|     unstable_renderSubtreeIntoContainer(
20|       this, //当前组件
21|       <div className="dialog">{props.children}</div>, // 塞进传送门的JSX
22|       this.node // 传送门另一端的DOM node
23|     );
24|   }
25| }
```

Reducer

什么是reducer(<https://cn.redux.js.org/docs/basics/Reducers.html>)

reducer就是一个纯函数，接收旧的 state 和 action，返回新的 state。

```
1 | (previousState, action) => newState
```

之所以将这样的函数称之为 reducer，是因为这种函数与被传入 [Array.prototype.reduce\(reducer, initialValue\)](#) 里的回调函数属于相同的类型。保持 reducer 纯净非常重要。永远不要在 reducer里做这些操作：

- 修改传入参数

- 执行有副作用的操作，如api请求和路由跳转；
- 调用非纯函数，如Date.now() 或 Math.random()

什么是reduce

此例来自https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce

```

1 const array1 = [1, 2, 3, 4];
2 const reducer = (accumulator, currentValue) => accumulator + currentValue;
3 // 1 + 2 + 3 + 4
4 console.log(array1.reduce(reducer));
5 // expected output: 10
6 // 5 + 1 + 2 + 3 + 4
7 console.log(array1.reduce(reducer, 5));
8 // expected output: 15

```

思考：有如下函数，聚合成一个函数，并把第一个函数的返回值传递给下一个函数，如何处理。

```

1 function f1(arg) {
2   console.log("f1", arg);
3   return arg;
4 }
5 function f2(arg) {
6   console.log("f2", arg);
7   return arg;
8 }
9 function f3(arg) {
10  console.log("f3", arg);
11  return arg;
12 }

```

方法：

```

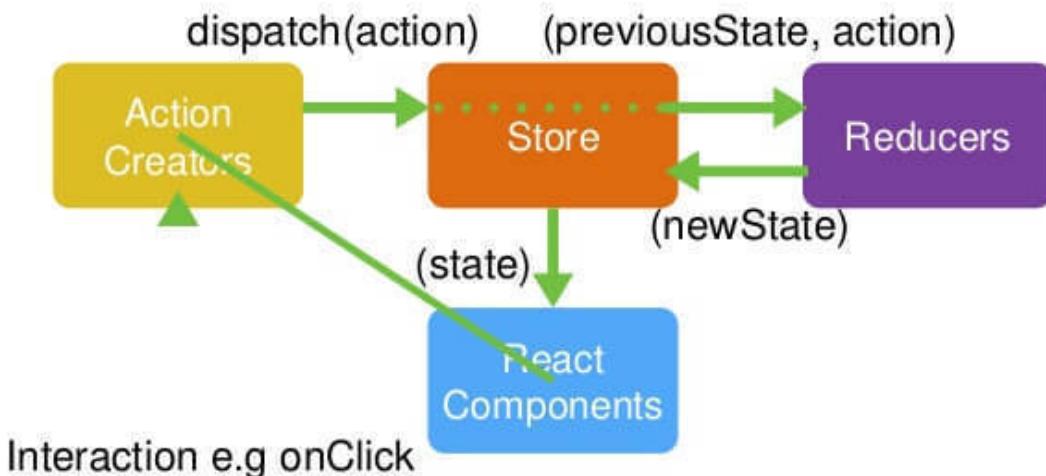
1 function compose(...funcs) {
2   if (funcs.length === 0) {
3     return arg => arg
4   }
5   if (funcs.length === 1) {
6     return funcs[0]
7   }
8   return funcs.reduce((a, b) => (...args) => a(b(...args)))
9 }
10 console.log(compose(f1, f2, f3)("omg"));

```

Redux上手

Redux是JavaScript应用的状态容器。它保证程序行为一致性且易于测试。

Redux Flow



React + Redux

@nikgraf

安装redux

npm install redux --save

redux上手

redux用一个累加器举例

1. 需要一个store来存储数据
2. store里的reducer初始化state并定义state修改规则
3. 通过dispatch一个action来提交对数据的修改
4. action提交到reducer函数里，根据传入的action的type，返回新的state

创建store,src/store/ReduxStore.js

```
1 import {createStore} from 'redux'
2 const counterReducer = (state = 0, action) => {
3   switch (action.type) {
4     case 'add':
5       return state + 1
6     case 'minus':
7       return state - 1
8     default:
9       return state
10   }
11 }
12 const store = createStore(counterReducer)
13 export default store
```

创建ReduxPage

```

1 import React, { Component } from "react";
2 import store from "../store/ReduxStore";
3 export default class ReduxPage extends Component {
4     componentDidMount() {
5         store.subscribe(() => {
6             console.log("subscribe");
7             this.forceUpdate(); //this.setState({}); //如果使用forceUpdate，需要手动调用setState
8         });
9     }
10    add = () => {
11        store.dispatch({ type: "add" });
12    };
13    minus = () => {
14        store.dispatch({ type: "minus" });
15    };
16    stayStatic = () => {
17        store.dispatch({ type: "others" });
18    };
19    render() {
20        console.log("store", store);
21        return (
22            <div>
23                <h1>ReduxPage</h1>
24                <p>{store.getState()}</p>
25                <button onClick={this.add}>add</button>
26                <button onClick={this.minus}>minus</button>
27                <button onClick={this.stayStatic}>static</button>
28            </div>
29        );
30    }
31 }
32

```

如果点击按钮不能更新，因为没有订阅(subscribe)状态变更

还可以在src/index.js的render里订阅状态变更

```

1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import './index.css';
4 import App from './App';
5 import store from "./store/ReduxStore";
6
7 const render=()=>{
8     ReactDOM.render(
9         <App />,
10        document.getElementById('root')
11    );
12 }
13
14 render();
15 store.subscribe(render);

```

检查点

1. createStore 创建store
2. reducer 初始化、修改状态函数

3. getState 获取状态值
4. dispatch 提交更新
5. subscribe 变更订阅

Redux拓展

核心实现

- 存储状态state
- 获取状态getState
- 更新状态dispatch
- 变更订阅subscribe

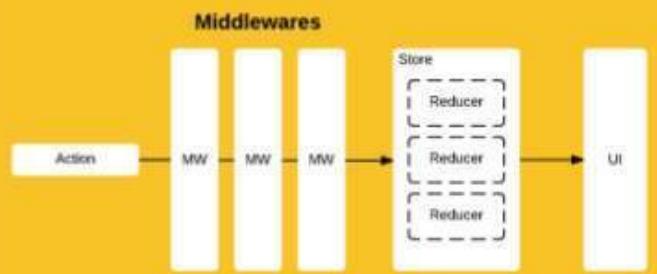
```
1 //kRedux.js
2 const createStore = (reducer, enhancer) => {
3   if (enhancer) {
4     return enhancer(createStore)(reducer);
5   }
6   let currentState;
7   let listeners = [];
8   //保存状态
9   function getState() {
10     return currentState;
11   }
12
13   function subscribe(listener) {
14     listeners.push(listener);
15   }
16   function dispatch(action) {
17     currentState = reducer(currentState, action);
18     listeners.forEach(v => v());
19     return action
20   }
21   dispatch({type: '@@initData'})
22   return { getState, subscribe, dispatch };
23 };
24
25 export default createStore;
```

异步

Redux只是个纯粹的状态管理器，默认只支持同步，实现异步任务比如延迟，网络请求，需要中间件的支持，比如我们试用最简单的redux-thunk和redux-logger。中间件就是一个函数，对 store.dispatch 方法进行改造，在发出 Action 和执行 Reducer 这两步之间，添加了其他功能。

```
1 | npm install redux-thunk redux-logger --save
```

Redux with middlewares



应用中间件, store.js

```
1 import { createStore, applyMiddleware } from "redux";
2 import logger from "redux-logger";
3 import thunk from "redux-thunk";
4 import counterReducer from './counterReducer'
5 const store = createStore(counterReducer, applyMiddleware(logger, thunk));
```

使用异步操作时的变化, ReduxPage.js

```
1 const mapDispatchToProps = {
2   add: () => {
3     return { type: "add" };
4   },
5   minus: () => {
6     return { type: "minus" };
7   },
8   asyAdd: () => dispatch => {
9     setTimeout(() => {
10       // 异步结束后, 手动执行dispatch
11       dispatch({ type: "add" });
12     }, 1000);
13   },
14 }
```

中间件实现

核心任务是实现函数序列执行

```
1 function compose(...fn) {
2   if (fn.length === 0) {
```

```

3     return (...args) => {
4   }
5   if (fn.length === 1) {
6     return fn(...args);
7   }
8   return fn.reduce((f1, f2) => (...args) => f1(f2(...args)));
9 }
10
11 export const applyMiddleware = (...middlewares) => {
12   // 返回强化后的函数
13   return createStore => (...args) => {
14     const store = createStore(...args);
15     let dispatch = store.dispatch;
16
17     const midApi = {
18       getState: store.getState(),
19       dispatch: (...args) => dispatch(...args)
20     };
21     // 使中间件可以获取状态值、派发action
22     const middlewareChain = middlewares.map(middleware =>
23       middleware(midApi));
24     // compose可以将middlewareChain函数数组合并成一个函数
25     dispatch = compose(...middlewareChain)(store.dispatch);
26     return {
27       ...store,
28       dispatch
29     };
30   };

```

redux-logger原理

```

1 function logger() {
2   return dispatch => action => {
3     // 中间件任务
4     console.log(action.type + "执行了!");
5     return dispatch(action);
6   };
7 }
8 const store = createStore(counterReducer, applyMiddleware(logger));

```

redux-thunk原理

thunk增加了处理函数型action的能力

```

1 function thunk({ getState }) {
2   return dispatch => action => {
3     if (typeof action === "function") {
4       return action(dispatch, getState);
5     } else {
6       return dispatch(action);
7     }
8   };
9 }
10 const store = createStore(counterReducer, applyMiddleware(thunk, logger));

```

react-redux的实现原理

```
1 import React,{useContext,useState} from "react";
2
3 const Context=React.createContext();
4
5 //返回dispatch方法
6 function bindActionCreator(actionCreator,dispatch){
7     return (...args)=>dispatch(actionCreator(...args))
8 }
9
10 function bindActionCreators(actionCreators,dispatch){
11     let obj={};
12     Object.keys(actionCreators).map(actionCreator=>{
13
14         obj[actionCreator]=bindActionCreator(actionCreators[actionCreator],dispatch)
15     })
16     return obj;
17 }
18
19 export const connect=(mapStateToProps=state=>state,mapDispatchToProps={}=>(Comp)=>props=>{
20     const store = useContext(Context)
21     const getMoreProps=()=>{
22         const stateProps=mapStateToProps(store.getState());
23         const dispatchProps=
24             bindActionCreators(mapDispatchToProps,store.dispatch)//给每个函数绑定dispatch
25             return {
26                 ...stateProps,
27                 ...dispatchProps
28             }
29     }
30     const [moreProps, setmoreProps] = useState(getMoreProps())
31     useEffect(() => {
32         store.subscribe(()=>{
33             setmoreProps({
34                 ...moreProps,
35                 ...getMoreProps(),
36             })
37         }, [])
38     }
39     return <Comp {...props} {...moreProps} dispatch={store.dispatch}/>
40 }
41
42 export function Provider({store,children}){
43     return(
44         <Context.Provider value={store}>
45             {children}
46             </Context.Provider>
47     )
48 }
```

使用redux

1. 安装: npm i redux react-redux -S

2. 创建store实例

```
1 import { createStore } from "redux"
2
3 function fruitReducer(state=initial,action){}
4 const store=createStore(fruitReducer);
5 export default store;
6
```

3.注册该实例Provider

```
1 import {Provider} from "react-redux"
2 import store from "./store"
3 import ReduxTest from "./ReduxTest"
4
5 <Provider store={store}><ReduxTest/></Provider>
```

4.组件中使用状态connect

```
1 import {connect} from "react-redux"
2
3 //connect返回一个高阶组件，可以把redux 状态作为属性注入到包装组件
4 export default connect(state=>({
5   loading:s
6 }))
```

5.代码优化

1.提取action creator

```
1 export const init=(payload)=>({
2   type:'init',
3   payload
4 })
5
6 export const add=(payload)=>({
7   type:'add',
8   payload
9 })
10
11 export const loadingStart=()=>({
12   type:'loading_start'
13 })
14
15 export const loadingEnd=()=>({
16   type:'loading_end'
17 })
```

2.映射为dispatch函数到属性

```
1 import { init, loadingStart, loadingEnd } from "./store"
2
3
4 function HookTest({fruits, loading, dispatch, loadingStart, loadingEnd, init})
{
5     ...
6 }
7
8 export default connect(state=>({
9     fruits:state.list,
10    loading:state.loading
11 }),{init,loadingStart,loadingEnd})(HookTest);
```

6.异步

1.安装redux-thunk:npm i redux-thunk -S

2.应用中间件， store.js

```
1 import { createStore, applyMiddleware } from "redux";
2 import logger from "redux-logger"
3 import thunk from "redux-thunk"
4
5 const store=createStore(fruitReducer,applyMiddleware(logger,thunk));
```

3.定义异步操作

```
1 export const asyncFetch=(payload)=>{
2     return dispatch=>{
3         dispatch({
4             type:'loading_start'
5 });
6         setTimeout(()=>{
7             dispatch({type:'loading_end'});
8             dispatch({type:'init',payload:['草莓','香蕉']})
9         },1000)
10    }
11 }
```

4.使用

```
1 import { asyncFetch } from './store'
2
3 const mapDispatchToProps={
4     asyncFetch
5 }
6
7 export default connect(asyncFetch)(HookTest);
```

7.模块化

```
1 //把action和Reducer移至fruit.redux.js
2
3 //store/index.js
4 import { createStore, applyMiddleware, combineReducers } from "redux";
```

```

5 import logger from "redux-logger"
6 import thunk from "redux-thunk"
7 import fruitReducer from './fruit.redux'
8
9 const
10 store=createStore(combineReducers(fruitReducer),applyMiddleware(logger,thunk));
11
12 //ReduxTest.js
13 import { asyncFetch } from './store/fruit.redux'
14 const mapStateToProps=state=>({
15     loading:state.fruit.loading,
16     fruits:state.fruit.list
17 })

```

react-router

1. 安装: npm i react-router-dom -S

2. 设定路由器

```
1 | import { BrowserRouter } from "react-router-dom";
```

3. 导航

```
1 | <Link to="/">水果列表</Link>
2 | <Link to="/add">添加水果</Link>
```

4. 路由

```

1 | {/*跟路由要添加exact, render可以实现条件渲染*/}
2 | <Route
3 |   exact
4 |   path='/'
5 |   render={props=>
6 |     loading?<div>数据加载中...</div>:<FruitList fruits=
7 |     {fruits}/>
8 |   }
9 |
10 |   / >

```

5. 传参

```
1 | <Link to={`/detail/${f}`}>{f}</Link>
2 | <Route path="/detail/:fruit" component={Detail}/>
```

6. 嵌套

Route组件嵌套在其他页面组件中就产生了嵌套关系

7. 404

```

1  /*添加Switch表示仅匹配一个*/
2  <Switch>
3      <Route exact path="/" render={props=><Redirect to="/list">
4          </Redirect>}/>
5      <Route component={()=><h3>页面不存在</h3>}/>

```

8. 路由守卫

创建高阶组件包装Route使其具有权限判断功能

```

1  function PrivateRoute({component:Component,isLogin,...rest}){
2      return(
3          <Route
4              {...rest}
5              render={
6                  props=>isLogin?(<Component {...props}></Component>):
7                      (<Redirect
8                          to={{pathname:'login',state:
9                              {redirect:props.location.pathname}}}></Redirect>)
10                     }
11                 )
12     }

```

使用

```

1  <PrivateRoute path="/add" component={FruitAdd}/>

```

react-router原理

react-router包含3个库，react-router、react-router-dom和react-router-native。react-router提供最基本的路路由功能，实际使用的时候我们不会直接安装react-router，而是根据应用运行的环境选择安装react-router-dom（在浏览器器中使用）或react-router-native（在rn中使用）。react-router-dom和react-router-native都依赖react-router，所以在安装时，react-router也会自动安装，创建web应用Route渲染内容的三种方式

Route渲染优先级：children>component>render。三者能接收到同样的[route props]，包括match，location and history，但是当不匹配的时候，children的match为null。这三种方式互斥，你只能用一种，它们的不同之处可以参考下文：

children: func

有时候，不管location是否匹配，你都需要渲染一些内容，这时候你可以用children。除了了不管location是否匹配都会被渲染之外，其它工作方法与render完全一样

```

1  import React, { Component } from "react";
2  import ReactDOM from "react-dom";
3  import { BrowserRouter as Router, Link, Route } from "react-router-dom";
4  function ListItemLink({ to, name, ...rest }) {
5      return (
6          <Route
7              path={to}
8              children={({ match }) => (
9                  <li className={match ? "active" : ""}>

```

```

10         <Link to={to} {...rest}>
11             {name}
12         </Link>
13     </Ti>
14   )
15   /
16 );
17 }
18 export default class RouteChildren extends Component {
19   render() {
20     return (
21       <div>
22         <h3>RouteChildren</h3>
23         <Router>
24           <ul>
25             <ListItemLink to="/somewhere" name="链接1" />
26             <ListItemLink to="/somewhere-else" name="链接2" />
27           </ul>
28         </Router>
29       </div>
30     );
31   }
32 }
33

```

render: func

但是当你用render的时候，你调用的只是个函数。但是它和component一样，能访问到所有的[route props]。

```

1 import React from "react";
2 import ReactDOM from "react-dom";
3 import { BrowserRouter as Router, Route } from "react-router-dom";
4 // 方方便便的内联渲染
5 ReactDOM.render(
6   <Router>
7     <Route path="/home" render={() => <div>Home</div>} />
8   </Router>,
9   node
10 );
11 // wrapping/composing
12 // 把route参数传递给你的组件
13 function FadingRoute({ component: Component, ...rest }) {
14   return (
15     <Route
16       {...rest}
17       render={routeProps => (
18         <FadeIn>
19           <Component {...routeProps} />
20         </FadeIn>
21       )}
22     />
23   );
24 }
25 ReactDOM.render(
26   <Router>
27     <FadingRoute path="/cool" component={Something} />

```

```
28     </Router>,
29     node
30   );
31 
```

component: component

只在当location匹配的时候渲染。

当你用 component 的时候，Router会用你指定的组件和React.createElement创建一个新的[React element]。这意味着当你提供的是一个内联函数的时候，每次render都会创建一个新的组件。这会导致不再更新已经现有组件，而是直接卸载然后再去挂载一个新的组件。因此，当用到内联函数的内联渲染时，请使用render或children。

```
1 import React, { Component } from "react";
2 import ReactDOM from "react-dom";
3 import { BrowserRouter, Route } from "react-router-dom";
4 class Foo extends Component {
5   componentDidMount() {
6     console.log("Foo componentDidMount");
7   }
8   componentWillUnmount() {
9     console.log("Foo componentWillUnmount");
10  }
11  render() {
12    const { counter } = this.props;
13    return <div>Foo: {counter}</div>;
14  }
15}
16 export default class RouterPage extends Component {
17  constructor(prop) {
18    super(prop);
19    this.state = { counter: 1 };
20  }
21  render() {
22    const { counter } = this.state;
23    return (
24      <div>
25        <button
26          onClick={() =>
27            this.setState({
28              counter: counter + 1
29            })
30          }
31        >
32          {counter}
33        </button>
34        <BrowserRouter>
35          {/* 渲染component会调用用
36 React.createElement, 如果使用用下面面这种匿名函数的形式,
37 每次都会生生成一个新得匿名函数, 导致生生成的组件的
38 type总不相同, 会产生生重复的卸载和挂载。
39 所以请正确使用用Route中的component和render。
40 */}
41          {/* <Route component={() => <Foo
42 counter={this.state.counter} />} /> */}
43        </BrowserRouter>
44      </div>
45    );
46  }
47}
```

```

44     /* 以下才是正确使用用 */
45     <Route render={() => <Foo counter={this.state.counter} />} />
46   </BrowserRouter>
47   </div>
48 );
49 }
50 }
51

```

实现BrowserRouter

BrowserRouter: 历史记录管理对象history初始化及向下传递, location变更监听

创建测试页面MyRouterPage.js,

```

1 import React, { Component } from "react";
2 import { BrowserRouter, Link, Route } from "../my-react-router-dom";
3 import HomePage from "./Home";
4 import UserPage from "./User";
5 export default class MyRouterPage extends Component {
6   render() {
7     return (
8       <div>
9         <h3>MyRouterPage</h3>
10        <BrowserRouter>
11          <Link to="/">首页</Link>
12          <Link to="/user">用户中心</Link>
13          <Route path="/" exact component={HomePage} />
14          <Route path="/user" component={UserPage} />
15        </BrowserRouter>
16      </div>
17    );
18  }
19 }
20

```

my-react-router-dom.js。BrowserRouter:history初始化及向下传递, location变更监听

```

1 import { createBrowserHistory } from "history";
2 import React, { Component, useContext } from "react";
3 const Context = React.createContext();
4
5 {
6   /* <BrowserRouter>
7   <Link to="/">首页页</Link>
8   <Link to="/user">用用户中心心</Link>
9   <Route path="/" exact component={HomePage} />
10  <Route path="/user" component={UserPage} />
11  </BrowserRouter> */
12 }
13 export class BrowserRouter extends Component {
14   constructor(props) {
15     super(props);
16     this.history = createBrowserHistory(this.props);
17     this.state = {
18       location: this.history.location

```

```

19    };
20    this.unlisten=this.history.listen((location)=>{
21      this.setState({
22        location:location
23      })
24    })
25  }
26
27  componentWillUnmount(){
28    if(this.listen){
29      this.unlisten();
30    }
31  }
32  render() {
33    return (
34      <Context.Provider value={{ history:
35        this.history,location:this.state.location }}>
36        {this.props.children}
37      </Context.Provider>
38    );
39  }
40  export function Route(props){
41    const {path,component:Comp}=props;
42
43    const {location} = useContext(Context);
44    const match=path===location.pathname
45    return (<div>
46      {match?<Comp/>:null}
47    </div>)
48  }
49  export class Link extends Component {
50    handleClick = (event,history )=> {
51      event.preventDefault();
52      history.push(this.props.to)
53    };
54    render() {
55      const { children, to } = this.props;
56      return (
57        <Context.Consumer>
58          {ctx=>(
59            <a href="#" onClick=
60              {(event)=>this.handleClick(event,ctx.history)}>
61              {children}
62            </a>
63          )}
64        </Context.Consumer>
65      );
66    }
67  }

```

<https://github.com/ReactTraining/react-router>

- Router: history 初始化及向下传递, location 变更监听

```

1 | import React from "react";

```

```
2 import { createBrowserHistory as createHistory } from "history";
3
4 export const RouterContext = React.createContext();
5 export default class BrowserRouter extends React.Component {
6
7     static computeRootMatch(pathname) {
8         return { path: "/", url: "/", params: {}, isExact: pathname === "/" };
9     }
10
11     constructor(props) {
12         super(props);
13
14         this.state = {
15             location: props.history.location
16         };
17
18         // This is a bit of a hack. we have to start listening for location
19         // changes here in the constructor in case there are any
20         <Redirect>
21             // on the initial render. If there are, they will replace/push when
22             // they mount and since cdm fires in children before parents, we
23             // may
24             // get a new location before the <Router> is mounted.
25             this._isMounted = false;
26             this._pendingLocation = null;
27
28             if (!props.staticContext) {
29                 this.unlisten = props.history.listen(location => {
30                     if (this._isMounted) {
31                         this.setState({ location });
32                     } else {
33                         this._pendingLocation = location;
34                     }
35                 });
36             }
37
38             componentDidMount() {
39                 this._isMounted = true;
40
41                 if (this._pendingLocation) {
42                     this.setState({ location: this._pendingLocation });
43                 }
44
45             componentWillUnmount() {
46                 if (this.unlisten) this.unlisten();
47             }
48
49             render() {
50                 return (
51                     <RouterContext.Provider
52                         children={this.props.children || null}
53                         value={{
54                             history: this.props.history,
55                             location: this.state.location,
56                             match: Router.computeRootMatch(this.state.location.pathname),
57                         }
58                     >
```

```

57         staticContext: this.props.staticContext
58     }
59     />
60   );
61 }
62 }
```

- Route:路由配置，匹配检测，内容渲染

```

1 import React from "react";
2 import { isValidElementType } from "react-is";
3 import PropTypes from "prop-types";
4 import invariant from "tiny-invariant";
5 import warning from "tiny-warning";
6
7 import RouterContext from "./RouterContext";
8 import matchPath from "./matchPath";
9
10 function isEmptyChildren(children) {
11   return React.Children.count(children) === 0;
12 }
13
14 /**
15 * The public API for matching a single path and rendering.
16 */
17 class Route extends React.Component {
18   render() {
19     return (
20       <RouterContext.Consumer>
21         {context => {
22
23           const location = this.props.location || context.location;
24           const match = this.props.computedMatch
25             ? this.props.computedMatch // <Switch> already computed the
match for us
26             : this.props.path
27             ? matchPath(location.pathname, this.props)
28             : context.match;
29
30           const props = { ...context, location, match };
31
32           let { children, component, render } = this.props;
33
34           // Preact uses an empty array as children by
35           // default, so use null if that's the case.
36           if (Array.isArray(children) && children.length === 0) {
37             children = null;
38           }
39
40           return (
41             <RouterContext.Provider value={props}>
42               {props.match
43                 ? children
44                   ? typeof children === "function"
45                     ? __DEV__
46                     ? evalChildrenDev(children, props,
47                         this.props.path)
```

```

47         : children(props)
48         : children
49         : component
50         ? React.createElement(component, props)
51         : render
52         ? render(props)
53         : null
54         :null}
55     </RouterContext.Provider>
56   );
57 }
58 </RouterContext.Consumer>
59 );
60 }
61 }
62 }
63
64 export default Route;

```

依赖: matchPath.js

```

1 import pathToRegexp from "path-to-regexp";
2
3 const cache = {};
4 const cacheLimit = 10000;
5 let cacheCount = 0;
6
7 function compilePath(path, options) {
8   const cacheKey =
` ${options.end}${options.strict}${options.sensitive}`;
9   const pathCache = cache[cacheKey] || (cache[cacheKey] = {});
10
11   if (pathCache[path]) return pathCache[path];
12
13   const keys = [];
14   const regexp = pathToRegexp(path, keys, options);
15   const result = { regexp, keys };
16
17   if (cacheCount < cacheLimit) {
18     pathCache[path] = result;
19     cacheCount++;
20   }
21
22   return result;
23 }
24
25 /**
26 * Public API for matching a URL pathname to a path.
27 */
28 function matchPath(pathname, options = {}) {
29   if (typeof options === "string" || Array.isArray(options)) {
30     options = { path: options };
31   }
32
33   const { path, exact = false, strict = false, sensitive = false } =
options;
34

```

```

35 const paths = [].concat(path);
36
37 return paths.reduce((matched, path) => {
38   if (!path) return null;
39   if (matched) return matched;
40
41   const { regexp, keys } = compilePath(path, {
42     end: exact,
43     strict,
44     sensitive
45   });
46   const match = regexp.exec(pathname);
47
48   if (!match) return null;
49
50   const [url, ...values] = match;
51   const isExact = pathname === url;
52
53   if (exact && !isExact) return null;
54
55   return {
56     path, // the path used to match
57     url: path === "/" && url === "" ? "/" : url, // the matched
portion of the URL
58     isExact, // whether or not we matched exactly
59     params: keys.reduce((memo, key, index) => {
60       memo[key.name] = values[index];
61       return memo;
62     }, {})
63   };
64 }, null);
65 }
66
67 export default matchPath;

```

- Link.js:跳转连接，处理点击事件

```

1 import React from "react";
2 import { __RouterContext as RouterContext } from "react-router";
3 import { createLocation } from "./utils/locationutils";
4
5 /**
6  * The public API for rendering a history-aware <a>.
7  */
8 class Link extends React.Component{
9   handleClick(event, history){
10     event.preventDefault();
11     history.push(this.props.to);
12   }
13   render(){
14     const {to, ...rest} = this.props;
15     return (
16       <RouterContext.Consumer>
17         {context => {
18           const location = typeof to === 'string'
19             ? createLocation(to, null, null, context.location):to;
20           const href = location ? history.createHref(location) : "";
21         }
22       )
23     );
24   }
25 }
26
27 export default Link;

```

```

21         return (
22             <a
23                 {...rest}
24                 onClick=
25 {event=>this.handleClick(event,context.history)}
26                     href={href}
27                     >
28                         {this.props.children}
29                     </a>
30                 )
31             );
32         </RouterContext.Consumer>
33     );
34 }
35 }
36
37 export default Link;

```

redux原理

<https://zhuanlan.zhihu.com/p/50247513>

```

1  export default function createStore(reducer, enhancer) {
2      if (typeof enhancer !=='undefined') {
3          if (typeof enhancer !=='function') {
4              throw new Error('Expected the enhancer to be a function.')
5          }
6          //函数柯里化, enhancer提供增强版(中间件扩展)的store
7          return enhancer(createStore)(reducer)
8      }
9
10     //store内部私有变量(外部无法直接访问)
11     let currentState ={}
12     let currentListeners = []
13
14     //获取最新state
15     function getState() {
16         return currentState
17     }
18
19     //用于订阅state的更新
20     function subscribe(listener) {
21         currentListeners.push(listener)
22     }
23
24     function dispatch(action) {
25         currentState=reducer(currentState,action)
26         //通知所有之前通过subscribe订阅state更新的回调listener
27         const listeners = currentListeners
28         for(let i =0; i < listeners.length; i++) {
29             const listener = listeners[i]listener()
30         }
31         return action
32     }
33     dispatch({ type:'@@redux/INIT'})
34     return {

```

```
35     dispatch,
36     subscribe,
37     getState
38   }
39
40 }
41
42
43 export default function applyMiddleware (...middlewares) {
44   return (next) => (reducer, initialState) => {
45     var store = next(reducer, initialState);
46     var dispatch = store.dispatch;
47     var chain = [];
48     var middlewareAPI = {
49       getState: store.getState,
50       dispatch:(action)=>dispatch(action)
51     };
52     chain = middlewares.map(middleware =>middleware(middlewareAPI));
53     dispatch = compose(...chain, store.dispatch);
54     return {
55       ...store,
56       dispatch
57     };
58   };
59 }
60
61 export default function compose(...funcs) {
62   if (funcs.length ===0) {
63     return arg => arg
64   }
65   if (funcs.length ===1) {
66     return funcs[0]
67   }
68   return funcs.reduce((a, b) => (...args) => a(b(...args)))
69 }
70
71 function bindActionCreator (actionCreator, dispatch) {
72   return (...args) => dispatch(actionCreator(...args))
73 }
74
75 export default function bindActionCreators (actionCreators, dispatch) {
76   if (typeof actionCreators ==='function') {
77     return bindActionCreator(actionCreators, dispatch)
78   }
79   if (typeof actionCreators !=='object'|| actionCreators ===null) {
80     throw new Error(`bindActionCreators expected an object or a
function, instead received ${actionCreators === null ? 'null' : typeof
actionCreators}. ` + `Did you write "import ActionCreators from" instead of
"import * as ActionCreators from"?`)
81   }
82   const keys = Object.keys(actionCreators)
83   const boundActionCreators ={}
84   for (let i =0; i < keys.length; i++) {
85     const key = keys[i]
86     const actionCreator = actionCreators[key]
87     if (typeof actionCreator ==='function') {
88       boundActionCreators[key] = bindActionCreator(actionCreator,
dispatch)
```

```

89         } else {
90             warning(`bindActionCreators expected a function actionCreator
91             for key '${key}', instead received type '${typeof actionCreator}'.`)
92         }
93     return boundActionCreators
94 }
95

```

react-redux原理

```

1 import {Component} from "react";
2 import React from "react";
3 import {PropTypes} from 'prop-types'
4
5 const connect = (mapStateToProps, mapDispatchToProps) => (wrappedComponent
6 => {
7     class Connect extends Component {
8         static contextTypes={
9             store:PropTypes.object
10        }
11        constructor(props,context) {
12            super(props,context)
13            this.state = {
14                props:{}
15            }
16        }
17        componentDidMount(){
18            const {store} =this.context
19            store.subscribe(() => {
20                this.update()
21            })
22        }
23        update(){
24            const {store} =this.context
25            const stateProps=mapStateToProps(store.getState())
26            const
27            dispatchProps=bindActionCreators(mapStateToProps,store.dispatch)
28            this.setState({
29                props:{
30                    ...this.state.props,
31                    ...stateProps,
32                    ...dispatchProps
33                }
34            })
35            render() {
36                return <wrappedComponent {...this.state}>
37            }
38        }
39
40        return Connect
41    }
42 }
43
44 export default class Provider extends Component {

```

```

45     static childContextTypes = {
46       store: PropTypes.object
47     }
48
49     getChildContext(){
50       return{ store: this.props.store}
51     }
52     constructor(props,context) {
53       super(props,context)
54       this.store=props.store
55     }
56
57     render() {
58       return this.props.children
59     }
60   }
61

```

redux-thunk原理

```

1 const thunk={({dispatch,getState})=>next=>action=>{
2   if(typeof action==='function'){
3     return action(dispatch,getState)
4   }
5   return next(action)
6 }
7 export default thunk;

```

redux-saga

redux-saga和redux-thunk的区别?

redux-saga利用了ES6的generator,派发action依然是对象,易管理,执行,测试和失败处理

redux-thunk派发的是函数

使用

- 安装 npm i redux-saga -S
- 创建清单saga.js

```

1 import {call,put,takeEvery} from 'redux-saga/effects'
2
3 //模拟登陆
4 const UserService={
5   login(username){
6     return new Promise((resolve,reject)=>{
7       setTimeout(()=>{
8         if(username==='zy1'){
9           resolve({id:1,name:'zy1',age:18})
10        }else{
11          reject('用户名或密码错误')
12        }
13      },1000)
14    })
15  }
16}

```

```

17
18
19 //work Saga
20 function* login(action){
21     try {
22         yield put({type:'requestLogin'});
23         const result=yield call(userservice.login,action.username);
24         yield put({type:'loginsuccess',result})
25     } catch (error) {
26         yield put({type:'loginFailure',error})
27     }
28 }
29
30 function* mySaga(){
31     yield takeEvery('login',login);
32 }
33 export default mySaga;

```

- 注册

```

1 import createSagaMiddleware from 'redux-saga';
2 import mysaga from 'saga'
3
4 //1.创建saga中间件并注册
5 const sagaMiddleware= createSagaMiddleware();
6 const store=createStore({
7     combineReducers({user})
8     applyMiddleware(logger,sagaMiddleware)
9 })
10
11 //2.中间件运行saga
12 sagaMiddleware.run(mysaga)
13 export default store

```

2.generator

- 基本概念：流程控制语句

```

1 function* g(){
2     yield 'a';
3     return 'b';
4 }
5 const gen=g();
6 gen.next();//{value:'a',done:false}
7 gen.next();//{value:'b',done:true}

```

- 传参

```

1 function* g(x){
2     let y=yield x;
3     yield y;
4 }
5 const gen=g(1);
6 gen.next();//{value:1,done:false}
7 gen.next();//{value:2,done:true}

```

- 异步

```

1 function* g(a){
2     let b=yield asyncFetch(a);
3     yield asyncFetch(b);
4 }
5
6 function asyncFetch(x){
7     return new Promise((reslove)=>{
8         reslove(x*x)
9     })
10}
11 const gen=g(2);
12 gen.next().value
13 .then(r=>gen.next(r).value)
14 .then(r=>console.log(r)

```

umi+dva

- 安装: npm i umi -D
- 自动生成路由:umi g page index
- 启动服务:umi dev
- 动态路由:以\$开头的文件或目录

```

1 export default function({match}) {
2     return (
3         <div>
4             <h1>users/{match.params.id}</h1>
5         </div>
6     );
7 }

```

- 嵌套路由:目录下创建_layout

```

1 //创建父组件 umi g page users/_layout
2 export default function({match,children}) {
3     return (
4         <div>
5             <h1>users layout</h1>
6             {children}
7         </div>
8     );
9 }

```

- 页面跳转

```

1 import styles from './index.css';
2 import Link from "umi/link";
3 import router from "umi/router";
4
5 export default function() {
6     const users=[{id:1,name:'zy1'},{id:2,name:'yw'}]
7     return (
8         <div className={styles.normal}>

```

```

9   <h1>Page index</h1>
10  <ul>
11    {
12      users.map(user=>(
13        <li key={user.id} onclick=
14          {()=>router.push(`/users/${user.id}`)}>{user.name}</li>
15          <li key={user.id}><Link to={`/users/${user.id}`}>
16            {user.name}</li>
17          )
18        )
19      );

```

- 配置路由：业务复杂后仍需配置路由

```

1 export default{
2   routes:[
3     {
4       path: '/',
5       component: './index'
6     },
7     {
8       path: '/users',
9       component: './users/_layout',
10      routes:[
11        {path: '/users/' ,component: './users/index'},
12        {path: '/users/:id' ,component: './users/$id'}
13      ]
14    },
15  ]
16}
17

```

- 404页面：添加不带path的路由配置项:{component:'./notFound'}

```

1  {
2    component: './notFound'
3  },

```

- 权限路由

```

1  {
2    path: '/',
3    component: './about',
4    Routes:[
5      './routes/PrivateRoute.js' //相对于根目录
6    ]
7  },

```

- 引入antd

- 添加antd: npm i antd -S

- 添加umi-plugin-react:npm i umi-plugin-react -D

win10有权限错误，通过管理员权限打开vscode

- 修改config/config.js

```

1  plugins:[
2      ['umi-plugin-react',{
3          antd:true
4      }]
5 ],

```

- 数据流管理dva

- 配置

```

1  plugins:[
2      ['umi-plugin-react',{
3          antd:true,
4          dva:true
5      }]
6 ],

```

- 创建src/models/goods.js文件

```

1 export default{
2     namespace:'goods',//model
3     state:[{title:'web全栈'},{title:'java架构师'}],
4     effects:{},//异步操作
5     reducers:{}//更新状态
6 }
7 }

```

- 使用状态

```

1 import {connect} from 'dva'
2 @connect({
3     state=>{
4         goodsList:state.goods//获取指定命名空间的模型状态
5     }
6 })
7
8 class Goods extends Component{}

```

React源码解析

顶层目录

当克隆React仓库 (<https://github.com/facebook/react>) 之后，你们将看到一些顶层目录：

- 源代码在/packages/，每个包的src子目录是你最需要花费精力的地方。
- fixtures (<https://github.com/facebook/react/tree/master/fixtures>) 包含一些给贡献者准备的小型React测试项目。
- build是React的输出目录。源码仓库中并没有这个目录，但是它会在你克隆React并且第一次构建它之后出现

React核心api

React(<https://github.com/facebook/react/blob/master/packages/react/src/React.js>)

```
1 const Children = {
2   map,
3   forEach,
4   count,
5   toArray,
6   only,
7 };
8
9 export {
10   Children,
11   createMutableSource,
12   createRef,
13   Component,
14   PureComponent,
15   createContext,
16   forwardRef,
17   lazy,
18   memo,
19   useCallback,
20   useContext,
21   useEffect,
22   useImperativeHandle,
23   useDebugValue,
24   useLayoutEffect,
25   useMemo,
26   useMutableSource,
27   useReducer,
28   useRef,
29   useState,
30   REACT_FRAGMENT_TYPE as Fragment,
31   REACT_PROFILER_TYPE as Profiler,
32   REACT_STRICT_MODE_TYPE as StrictMode,
33   REACT_DEBUG_TRACING_MODE_TYPE as unstable_DebugTracingMode,
34   REACT_SUSPENSE_TYPE as Suspense,
35   createElement,
36   cloneElement,
37   isValidElement,
38   ReactVersion as version,
39   ReactSharedInternals as
40   __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED,
41   // Deprecated behind disableCreateFactory
42   createFactory,
43   // Concurrent Mode
44   useTransition,
45   useDeferredValue,
46   REACT_SUSPENSE_LIST_TYPE as suspenseList,
47   REACT_LEGACY_HIDDEN_TYPE as unstable_LegacyHidden,
48   withSuspenseConfig as unstable_withSuspenseConfig,
49   // enableBlocksAPI
50   block,
51   // enableDeprecatedFlareAPI
52   useResponder as DEPRECATED_useResponder,
53   createEventResponder as DEPRECATED_createResponder,
```

```
53 // enableFundamentalAPI
54 createFundamental as unstable_createFundamental,
55 // enableScopeAPI
56 createScope as unstable_createScope,
57 useOpaqueIdentifier as unstable_useOpaqueIdentifier,
58 };
```

核心精简后：

```
1 const React = {
2   createElement,
3   Component
4 }
```

react-dom (<https://github.com/facebook/react/blob/master/packages/react-dom/src/client/ReactDOM.js>) 主要是render逻辑

最核心的api：React.createElement：创建虚拟DOM React.Component：实现自定义组件

ReactDOM.render：渲染真实DOM

ReactDOM(<https://zh-hans.reactjs.org/docs/react-dom.html>)

render()

```
1 ReactDOM.render(element, container[, callback])
```

在提供的 container 里渲染一个 React 元素，并返回对该组件的引用（或者针对无状态组件返回 null）。

当首次调用时，容器节点里的所有 DOM 元素都会被替换，后续的调用则会使用 React 的 DOM 差分算法 (DOM diffing algorithm) 进行高效的更新。

如果提供了可选的回调函数，该回调将在组件被渲染或更新之后被执行。

注意：使用 ReactDOM.render() 对服务端渲染容器器进行hydrate 操作的方式已经被废弃，并且会在 React 17 被 移除。作为替代，请使用 hydrate()<https://zh-hans.reactjs.org/docs/react-dom.html#hydrate>。

jsx

- 什么是JSX 语法糖 React 使用 JSX 来替代常规的 JavaScript。JSX 是一个看起来很像 XML 的 JavaScript 语法扩展。
- 为什么需要JSX 开发效率：使用 JSX 编写模板简单快速。执行效率：JSX编译为 JavaScript 代码后进行了优化，执行更快。类型安全：在编译过程中就能发现错误。
- 原理理：babel-loader会预编译JSX为 React.createElement(...)
- 与vue的异同：react中虚拟dom+jsx的设计一开始就有，vue则是演进过程中才出现的jsx本来就是js扩展，转义过程简单直接的多；vue把template编译为render函数的过程需要复杂的编译器转换字符串-ast-javascript字符串

jsx预处理前：

REACT 编辑器

显示JSX

```
class App extends React.Component {  
    render() {  
        return (  
            <div>  
                Hello {this.props.name}, I am {2 + 2} years old  
            </div>  
        )  
    }  
}  
  
ReactDOM.render(  
    <App name="React" />,  
    mountNode  
)
```

JSX预处理后:

REACT 编辑器

显示JSX

```
class App extends React.Component {  
    render() {  
        return React.createElement(  
            "div",  
            null,  
            "Hello ",  
            this.props.name,  
            ", I am ",  
            2 + 2,  
            " years old"  
        )  
    }  
}  
  
ReactDOM.render(React.createElement(App, { name: "React" }),  
    mountNode)
```

使用自定义组件的情况:

```
1 import React, { Component } from "react";  
2 import ReactDOM from "react-dom";  
3 import "./index.css";  
4 function FuncCmp(props) {  
    return <div>name: {props.name}</div>;  
}  
5  
6 class ClassCmp extends Component {  
7     render() {  
8         return <div>name: {this.props.name}</div>;  
9     }  
}
```

```

10    }
11 }
12 const jsx = (
13   <div>
14     <p>我是内容</p>
15     <FuncCmp name="我是function组件" />
16     <ClassCmp name="我是class组件" />
17   </div>
18 );
19
20 console.log.jsx;
21
22 const render = () => {
23   ReactDOM.render.jsx, document.getElementById("root"));
24 };
25
26 render();

```

build之后

```

1 function FuncCmp(props) {
2   return React.createElement("div", null, "name: ", props.name);
3 }
4 class ClassCmp extends React.Component {
5   render() {
6     return React.createElement("div", null, "name: ", this.props.name);
7   }
8 }
9 let jsx = React.createElement(
10   "div",
11   null,
12   " ",
13   React.createElement("div", { className: "border" }, "我是内容"),
14   " ",
15   React.createElement(FuncCmp, { name: "我是function组件" }),
16   " ",
17   React.createElement(ClassCmp, { name: "我是class组件" }),
18   " "
19 );
20 ReactDOM.render.jsx, document.getElementById("root"));

```

实现三大接口：

React.createElement, React.Component, ReactDOM.render

CreateElement

将传入的节点定义转换为vdom。

src/index.js

```

1 // import React, { Component } from "react";
2 // import ReactDOM from "react-dom";
3 import React from "./kreact/";
4 import ReactDOM from "./kreact/ReactDOM";
5 import "./index.css";
6 import App from "./App";

```

```

7 import store from "./store/ReduxStore";
8
9 function FuncCmp(props) {
10   return <div>name: {props.name}</div>;
11 }
12 class ClassCmp extends React.Component {
13   constructor(props) {
14     super(props);
15     this.state = { counter: 0 };
16   }
17   clickHandle = () => {
18     this.setState({
19       counter: this.state.counter+1
20     })
21   };
22   render() {
23     const { counter } = this.state;
24     return (
25       <div>
26         name: {this.props.name}
27         <p>counter: {counter}</p>
28         {[0, 1, 2].map(item => {
29           return <FuncCmp name={"我是function组件" + item} key={item} />;
30         })}
31         <button onClick={this.clickHandle}>点击</button>
32       </div>
33     );
34   }
35 }
36
37 const jsx = (
38   <div>
39     <p className="border">我是内容</p>
40     <FuncCmp name="我是function组件" />
41     <ClassCmp name="我是class组件" />
42   </div>
43 );
44
45 console.log.jsx;
46
47 const render = () => {
48   ReactDOM.render.jsx, document.getElementById("root"));
49 };
50
51 render();
52 store.subscribe(render);
53

```

- 创建./src/kreact/index.js, 它需要包含createElement方法

```

1 import { diff } from "./diff";
2
3 //jsx转换成vdom
4 const createElement = (type, props, ...children) => {
5   console.log(type, props, children);
6   props.children = children;
7   let vtype;

```

```

8  if (typeof type === "string") {
9    //原生标签
10   vtype = 1;
11 }
12
13 if (typeof type === "function") {
14   vtype = type.isClassComp ? 2 : 3;
15 }
16
17 return {
18   type,
19   vtype,
20   props
21 };
22 };
23
24 class Component {
25   static isClassComp = {};
26   constructor(props) {
27     this.props = props;
28     this.state={};
29     this.$cache={};//可以存储父节点
30   }
31
32   setState=(nextState,callback)=>{
33     //真实的setState是批量处理
34     //暂时是假的
35     this.state={
36       ...this.state,
37       ...nextState,
38     }
39     this.forceUpdate();
40   }
41
42   forceUpdate=()=>{
43     //vnode=>node=>container
44     const {$cache:cache}=this;
45     const newVnode=this.render();
46     const newNode=diff(cache,newVnode);//newVnode=>node,最终更新到container
47     console.log(newVnode,"newVnode")
48     //每次更新vnode、node
49     this.$cache={
50       ...cache,
51       vnode:newVnode,
52       node:newNode
53     }
54   }
55 }
56
57 const React = {
58   Component,
59   createElement
60 };
61 export default React;
62

```

- 修改index.js实际引入入kreact，测试

```
1 | import React from './kreact/';
```

createElement被调用时会传入标签类型type，标签属性props及若干子元素children index.js中从未使用过React类或者其任何接口，为何需要导入它？JSX编译后实际调用的是React.createElement方法，所以只要出现JSX的文件中都需要导入React类

ReactDOM.render

- 创建react-dom.js
- 需要实现一个render函数，能够将vdom渲染出来，这里先打印vdom结构

```
1 | import { initVnode } from './virtual-dom';
2 | function render(vnode, container) {
3 |   const node = initVnode(vnode, container);
4 |   container.appendChild(node)
5 | }
6 |
7 |
8 | export default {
9 |   render,
10| }
```

实现Component

要实现class组件，需要添加Component类，Component.js

```
1 | class Component {
2 |   static isClassComp = {};
3 |   constructor(props) {
4 |     this.props = props;
5 |     this.state = {};
6 |     this.$cache = {} // 可以存储父节点
7 |   }
8 |
9 |   setState = (nextState, callback) => {
10 |     // 真实的setState是批量处理
11 |     // 暂时是假的
12 |     this.state = {
13 |       ...this.state,
14 |       ...nextState,
15 |     }
16 |     this.forceUpdate();
17 |   }
18 |
19 |   forceUpdate = () => {
20 |     // vnode=>node=>container
21 |     const {$cache: cache} = this;
22 |     const newVnode = this.render();
23 |     const newNode = diff(cache, newVnode); // newVnode=>node，最终更新到container
24 |     console.log(newVnode, "newVnode")
25 |     // 每次更新vnode、node
26 |     this.$cache = {
27 |       ...cache,
28 |       vnode: newVnode,
29 |       node: newNode
30 |     }
31 |   }
32 | }
```

```
30  }
31  }
32 }
```

浅层封装，`setState`(<https://github.com/facebook/react/blob/master/packages/react/src/ReactBaseClasses.js>) 现在只是一个占位符

组件类型判断

传递给`createElement`的组件有三种组件类型，1: dom组件，2. class组件，3. 函数组件，使用`vtype`属性标识

转换vdom为真实dom

`./virtual-dom.js`

```
1 //当前把函数vnode做成node
2 export const initVnode = (vnode, container) => {
3   let node;
4   const { vtype } = vnode;
5   if (!vtype) {
6     node = initTextNode(vnode, container);
7   }
8   if (vtype === 1) {
9     node = initHtmlNode(vnode, container);
10 } else if (vtype === 3) {
11   node = initFunNode(vnode, container);
12 }else if(vtype==2){
13   node=initClassNode(vnode,container)
14 }
15
16   return node;
17 };
18
19 const initHtmlNode = (vnode, container) => {
20   const { type, props } = vnode;
21   const node = document.createElement(type);
22   const { children,...rest } = props;
23   children.map(child => {
24     if(Array.isArray(child)){
25       child.map(item=>{
26         node.appendChild(initVnode(item, node));
27       })
28     }else{
29
30       node.appendChild(initVnode(child, node));
31     }
32   );
33
34   object.keys(rest).map(key=>{
35     if(key==='className'){
36       node.setAttribute('class',rest[key])
37     }
38     if(key.slice(0,2)==='on'){
39       node.addEventListener('click',rest[key])
40     }
41   )
42   return node;
```

```

43  };
44
45  const initFunNode = (vnode, container) => {
46    const { type, props } = vnode;
47    const v vnode = type(props); //这里拿到的是string/div/function,所以得再执行一次
48    return initVnode(v vnode, container);
49  };
50
51  const initTextNode = (vnode, container) => {
52    const node = document.createTextNode(vnode);
53    return node;
54  };
55
56  const initClassNode=(vnode,container)=>{
57    const { type, props } = vnode;
58    const comp = new type(props);
59    const v vnode=comp.render();
60    const node=initVnode(v vnode,container);
61    let cache={
62      parentNode:container,
63      vnode:v vnode,//当前的虚拟dom节点用于diff
64      node//真实的dom节点，可以用于最后的替换
65    }
66    comp.$cache=cache;
67    return node;
68  }
69

```

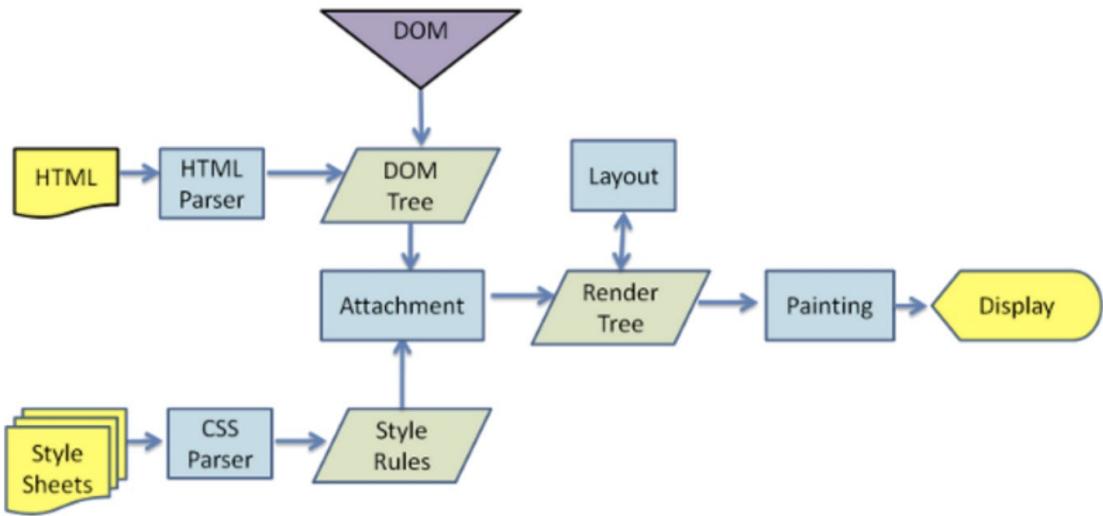
总结:

1. webpack+babel编译时，替换JSX为React.createElement(type,props,...children)
2. 所有React.createElement()执行结束后得到一个JS对象即vdom，它能够完整描述dom结构
3. ReactDOM.render(vdom, container)可以将vdom转换为dom并追加到container中
4. 实际上，转换过程需要经过一个diff过程，比对出实际更新补丁操作dom

虚拟DOM

常见问题：react virtual dom是什么？说一下diff算法？what？用JavaScript 对象表示 DOM 信息和结构，当状态变更的时候，重新渲染这个 JavaScript 的对象结构。这个 JavaScript 对象称为virtual dom；

传统dom渲染流程



```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta name="viewport" content="width=device-width, initial-scale=1.0">
6      <meta http-equiv="X-UA-Compatible" content="ie=edge">
7      <title>Document</title>
8  </head>
9  <body>
10     <script>
11         let div=document.createElement('div');
12         let ret='';
13         for(let key in div){
14             ret+=key+""
15         }
16         console.log(ret)
17     </script>
18  </body>
19  </html>

```

```

not available

alignTitleLangTranslatedDirDataSetHiddenTabIndexAccessKeyDraggablespellCheckAutoCapitalizeContentEditableInputMode_index.html:16
offsetParentOffsetTopOffsetLeftOffsetWidthOffsetHeightStyleInnerTextOuterTextOnCopyOnCutOnPasteOnAbortOnBlurOnCancelOnPlayOnCanPlayThroughOnChangeOnCloseOnContextMenuOnContextMenuOnChangeOnDbClickOnDragAndDropOnDragStartOnDragEndOnDragLeaveOnDragOverOnDragEndOnDropOnDropDurationOnChangeOnEmptiedOnEndOnErrorOrOnFocusOnInputOnPutOnValidationKeyDownOnKeyPressOnKeyUpOnLoadOnLoadedMetadataOnLoadStartOnMouseDownOnMouseEnterOnMouseLeaveOnMouseMoveOnMouseUpOnMouseOverOnMouseUpOnMouseWheelOnPauseOnPlayOnPlayingOnProgressOnRateOnChangeOnResetOnResizeOnScrollOnSeekedOnSeekingOnSelectOnStalledOnUpdateOnSuspensionOnUpdateOnToggleOnVolumeOnChangeOnWaitingOnWheelOnAuxClickOnGotPointerCaptureOnLostPointerCaptureOnPointerDownOnPointeOnRemoveOnPointerUpOnPointerCancelOnPointerOverOnPointerEnterOnPointerLeaveOnSelectStartOnSelectionChangeOnOnceClickOnFocusBlurNameSpaceURIprefixLocalNameTagNameIdClassNameStyleListSlotAttributeShadowRootAssignedSlotInInnerHTMLOuterHTMLScrollTopScrollLeftScrollWidthScrollHeightClientLeftClientWidthClientHeightAttributeStyleMapOnBeforeCopyOnBeforeCutOnBeforePasteOnSearchOnPreviousElementSiblingNextElementSiblingChildElementFirstChildLastElementChildChildElementCountOnFullScreenChangeOnFullScreenErrorOnWebKitFullScreenChangeOnWebKitFullScreenErrorSetPointerCaptureOnReleasePointerCaptureHasPointerCaptureHasAttributesGetAttributeNamesGetAttributeSetAttributeGetAttributeNSSetAttributeNodeSetAttributeNodeNSRemoveAttributeMoveAttributeNSHasAttributeHasAttributeNStoggleAttributeGetAttributeNodeGetAttributeNodeNSSetAttributeNodeSetAttributeNodeNSRemoveAttributeNodeDocloseSetAttributeMatchToSelectorAttachShadowElementsByTagNameGetElementsByTagNameByClassNameInsertAdjacentElementInsertAdjacentTextInsertAdjacentHTMLRequestPointerLockGetClientRectRectGetBoundingClientRectOnViewScrollOnScrollOnScrollByOnScrollIntoViewIfNeedAnimateComputedStyleMapBeforeAfterReplaceWithRemoveAppendQuerySelectorOrQueryAllRequestFullScreenOnWebKitRequestFullScreenOnRequestFullScreeenCreateShadowRootElementGetDestinationInsertionPointsELEMENT_NODEATTRIBUTE_NODETEXT_NODEDATA_SECTION_NODEENTITY_REFERENCE_NODEPROCESSING_INSTRUCTION_NODECOMMENT_NODEDOCUMENT_NODEDOCUMENT_TYPE_NODEDOCUMENT_FRAGMENT_NODENOTATION_NODEDOCUMENT_POSITION_DISCONNECTEDDOCUMENT_POSITION_PRECEDINGDOCUMENT_POSITION_FOLLOWINGDOCUMENT_POSITION_CONTAINSDOCUMENT_POSITION_CONTAINED_BYDOCUMENT_POSITION_IMPLEMENTATION_SPECIFICnodeTypeOpenDeNamebaseURIIsConnectedToOwnerDocumentparentNodeDeparentElementChildNodesFirstChildLastChildPreviousSiblingNextSiblingNodeValueTextContentHasChildNodesGetRootNodeNormalizeCloneNodeIsEqualNodeIsSameNodeCompareDocumentPositionContainsLookupPrefixLookupNamespaceURIIsDefaultNamespaceInsertBeforeAppendChildReplaceChildRemoveChildAddEventListenerRemoveEventListennerDispatchEvent

```

why? DOM操作很慢，轻微的操作都可能导致页面重新排版，非常耗性能。相对于DOM对象，js对象处理起来更快，而且更简单。通过diff算法对比新旧vdom之间的差异，可以批量的、最小化的执行dom操作，从而提高性能。where? react中用JSX语法描述视图，通过babel-loader转译后它们变为React.createElement(...)形式，该函数将生成vdom来描述真实dom。将来如果状态变化，vdom将作出相应变化，再通过diff算法对比新老vdom区别从而做出最终dom操作。

reconciliation协调 (<https://zh-hans.reactjs.org/docs/reconciliation.html>)

设计动力

在某一时间节点调用 React 的 render() 方法，会创建一棵由 React 元素组成的树。在下一次 state 或 props 更更新时，相同的 render() 方法会返回一棵不同的树。React 需要基于这两棵树之间的差别来判断如何有效率的更新 UI 以保证当前 UI 与最新的树保持同步。

这个算法问题有一些通用的解决方案，即生成将一棵树转换成另一棵树的最小操作数。然而，即使在最前沿的算法中(http://grfia.dlsi.ua.es/ml/algorithms/references/editsurvey_bille.pdf)，该算法的复杂程度为 $O(n^3)$ ，其中 n 是树中元素的数量。

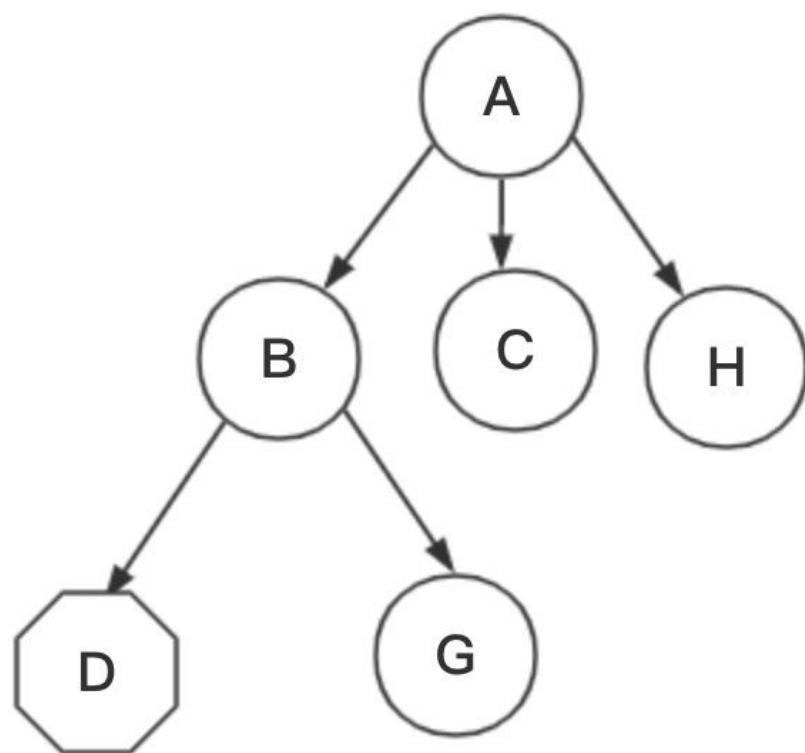
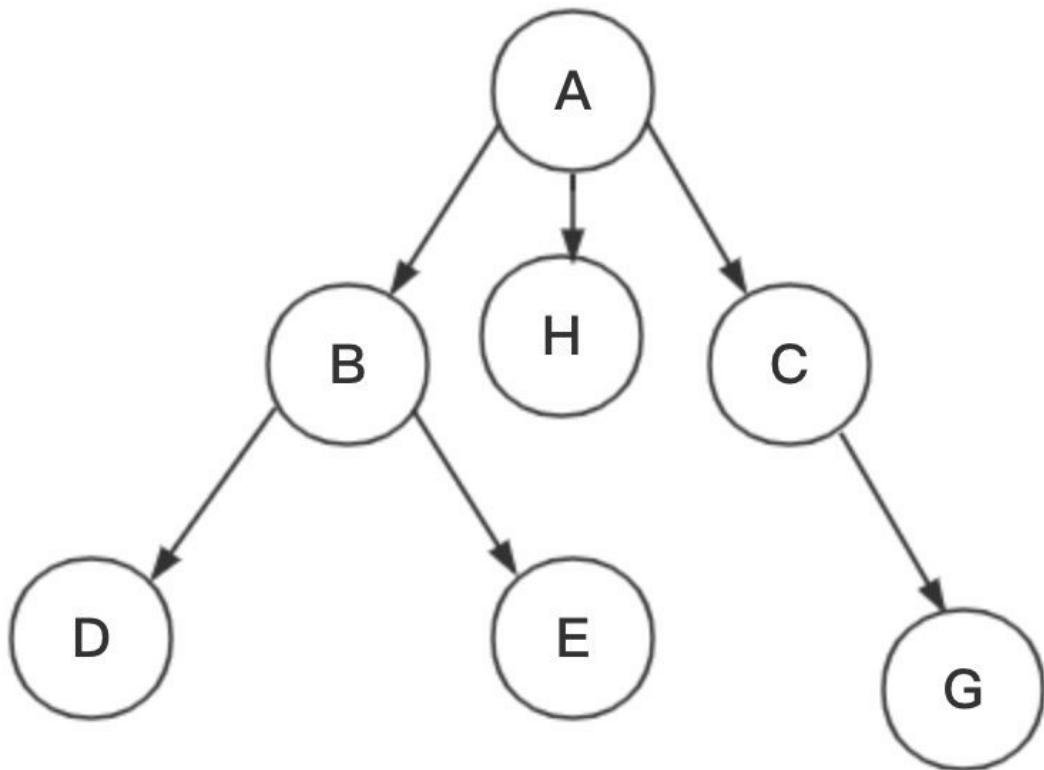
如果在 React 中使用用了该算法，那么展示 1000 个元素所需要执行行行的计算量量将在十亿的量级范围。这个开销实在是太过高昂。于是 React 在以下两个假设的基础之上提出了一套 $O(n)$ 的启发式算法：

1. 两个不同类型的元素会产生出不同的树；
2. 开发者可以通过 key prop 来暗示哪些子元素在不同的渲染下能保持稳定；

在实践中，我们发现以上假设在几乎所有实用的场景下都成立。

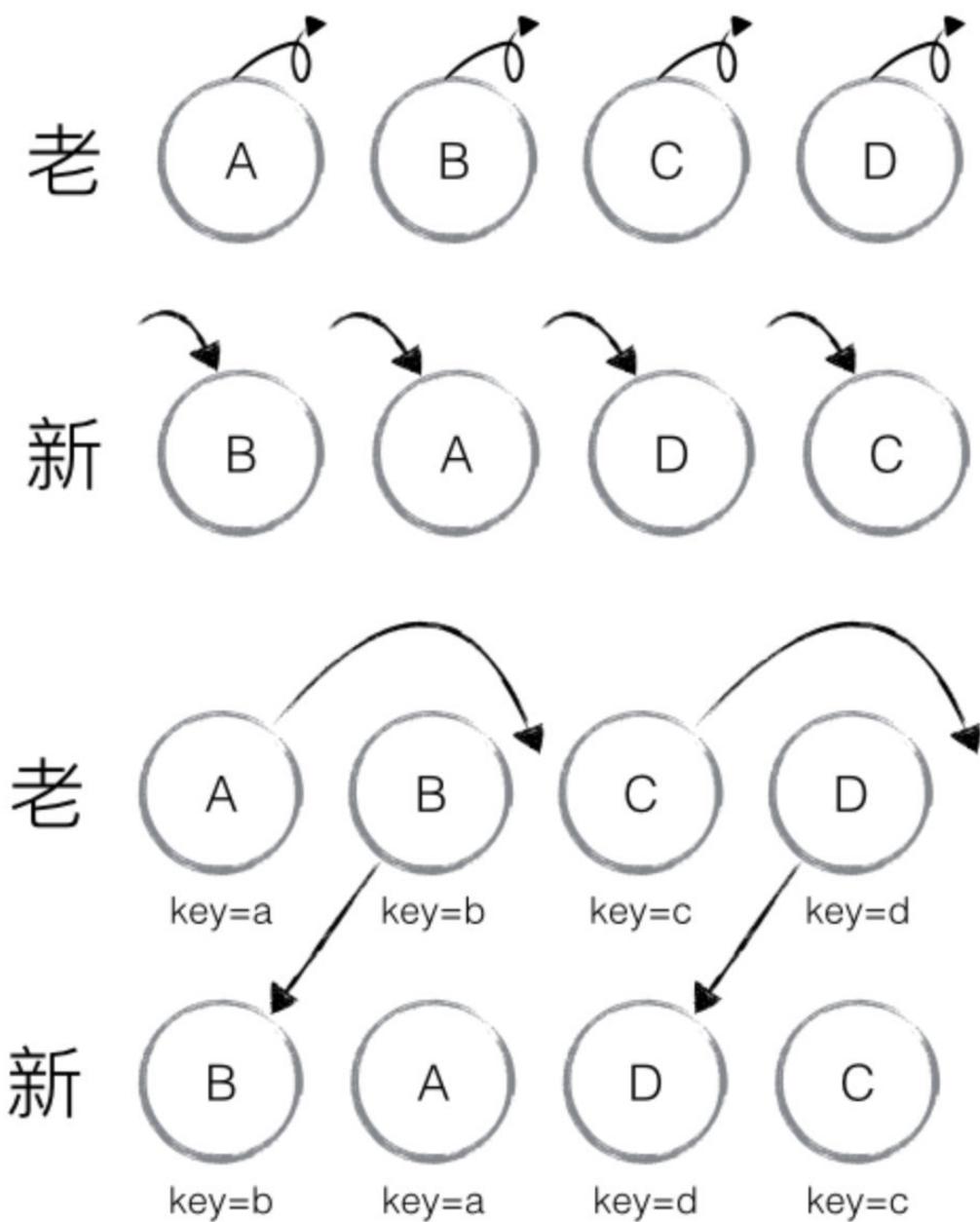
diff 算法

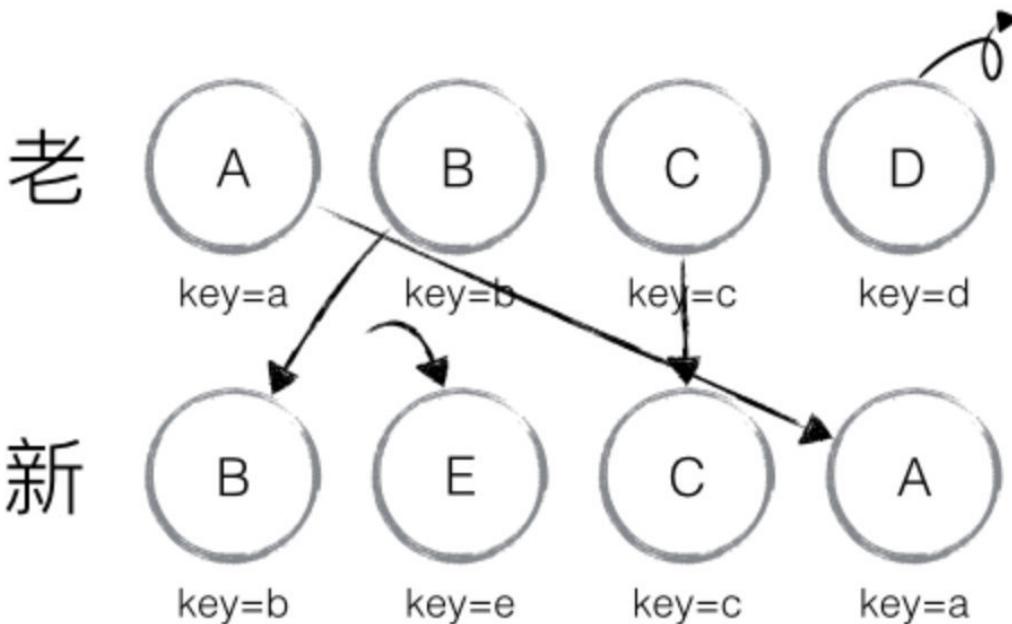
算法复杂度 $O(n)$



diff策略

- 同级比较，Web UI 中 DOM 节点跨层级的移动操作特别少，可以忽略不计。
- 拥有不同类的两个组件将会生成不同的树形结构。例如：div->p, CompA->CompB
- 开发者可以通过 key prop 来暗示哪些子元素在不同的渲染下能保持稳定；





在实践中也证明这三个前提策略略是合理且准确的，它保证了整体界面构建的性能。

diff过程

比对两个虚拟dom时会有三种操作：删除、替换和更新

删除：newVnode不存在时 替换：vnode和newVnode类型不同或key不同时 更新：有相同类型和key但vnode和newVnode不同时

```

1  export function compareTwoVnodes(vnode, newVnode, node, parentContext) {
2    let newNode = node;
3    if (newVnode == null) {
4      // remove
5      destroyVnode(vnode, node); //在vdom中删除
6      node.parentNode.removeChild(node); //dom中删除
7    } else if (vnode.type !== newVnode.type || vnode.key !== newVnode.key) {
8      // replace
9      destroyVnode(vnode, node); //在vdom中删除
10     newNode = initVnode(newVnode, parentContext);
11     node.parentNode.replaceChild(newNode, node);
12   } else if (vnode !== newVnode || parentContext) {
13     // same type and same key -> update
14     newNode = updateVnode(vnode, newVnode, node, parentContext);
15   }
16   return newNode;
17 }
```

更新操作

根据组件类型执行不同更新操作

```

1  function updateVnode(vnode, newVnode, node, parentContext) {
2    let { vtype } = vnode;
3    //更新class类型组件
4    if (vtype === VCOMPONENT) {
5      return updateVcomponent(vnode, newVnode, node, parentContext);
```

```

6 }
7 //更更新函数类型组件
8 if (vtype === VSTATELESS) {
9   return updateVstateless(vnode, newVnode, node, parentContext);
10 }
11 // ignore VCOMMENT and other vtypes
12 if (vtype !== VELEMENT) {
13   return node;
14 }
15 // 更更新元素
16 let oldHTML = vnode.props[HTML_KEY] && vnode.props[HTML_KEY].__html;
17 if (oldHTML != null) {
18   // 设置了innerHTML时先更更新当前元素在初始化innerHTML
19   updateVelem(vnode, newVnode, node, parentContext);
20   initVchildren(newVnode, node, parentContext);
21 } else {
22   // 正常更更新: 先更更新子元素, 在更更新当前元素
23   updateVchildren(vnode, newVnode, node, parentContext);
24   updateVelem(vnode, newVnode, node, parentContext);
25 }
26 return node;
27 }
28

```

patch过程

虚拟dom比对最终要转换为对应patch操作

属性更新

```

1 function updateVelem(velem, newVelem, node) {
2   let isCustomComponent =
3     velem.type.indexOf("-") >= 0 || velem.props.is != null;
4   _.patchProps(node, velem.props, newVelem.props, isCustomComponent);
5   return node;
6 }

```

子元素更新

```

1 function updateVchildren(vnode, newVnode, node, parentContext) {
2   // 更更新children, 产出三个patch数组
3   let patches = {
4     removes: [],
5     updates: [],
6     creates: []
7   };
8   diffVchildren(patches, vnode, newVnode, node, parentContext);
9   _.flatEach(patches.removes, applyDestroy);
10  _.flatEach(patches.updates, applyUpdate);
11  _.flatEach(patches.creates, applyCreate);
12 }

```

权衡

请谨记协调算法是一个实现细节。React 可以在每个 action 之后对整个应用进行重新渲染，得到的最终结果也会是一样的。在此情境下，重新渲染表示在所有组件内调用 render 方法，这不代表 React 会卸载或装载它们。React 只会基于以上提到的规则来决定如何进行差异的合并。由于 React 依赖探索的算法，因此当以下假设没有得到满足，性能会有所损耗。

1. 该算法不会尝试匹配不同组件类型的子树。如果你发现你在两种不同类型的组件中切换，但输出非常相似的内容，建议把它们改成同一类型。在实践中，我们没有遇到这类问题。
2. Key 应该具有稳定，可预测，以及列表内唯一的特质。不稳定的 key (比如通过 Math.random() 生成的) 会导致许多组件实例和 DOM 节点被不必要的重新创建，这可能导致性能下降和子组件中的状态丢失。优化点：减少 reflow

setState (<https://zh-hans.reactjs.org/docs/faq-state.html#what-does-setstate-do>)

setState() 会对一个组件的 state 对象安排一次更新。当 state 改变了，该组件就会重新渲染。class 组件的特点，就是拥有特殊状态并且可以通过 setState 更新状态并重新渲染视图，是 React 中最重要的 api。

问题：

1.setState 异步

```
1 // 批量
2 this.setState({ counter: this.state.counter + 1 });
3 this.setState({ counter: this.state.counter + 2 });
4 console.log("counter", this.state);
5 // 回调
6 this.setState({ counter: this.state.counter + 1 }, () => {});
7 this.setState(nextState => {
8   console.log("next", nextState);
9 });
10 // 异步
11 this.setState({ counter: this.state.counter + 1 });
12 console.log("counter", this.state); //0
13 // 不同步
14 setTimeout(() => {
15   setState({ foo: "bar" });
16 }, 1000);
17 // 原生事件
18 dom.addEventListener("click", () => {
19   setState({ foo: "bar" });
20 });
```

setState 并没有直接操作去渲染，而是执行了了一个异步的updater队列 我们使用一个类来专门管理理，./kreact/Component.js

```
1 export let updateQueue = {
2   updaters: [],
3   isPending: false,
4   add(updater) {
5     _.addItem(this.updaters, updater);
6   },
7   batchUpdate() {
8     if (this.isPending) {
9       return;
10    }
```

```
11 |     this.isPending = true;
12 |     /*
13 | each Updater.update may add new Updater to updateQueue
14 | clear them with a loop
15 | event bubbles from bottom-level to top-level
16 | reverse the Updater order can merge some props and state and reduce the
17 | refresh times
18 | see Updater.update method below to know why
19 | */
20 |     let { updaters } = this;
21 |     let Updater;
22 |     while ((Updater = updaters.pop())) {
23 |         Updater.updateComponent();
24 |     }
25 |     this.isPending = false;
26 | }
27 | };
28 | function Updater(instance) {
29 |     this.instance = instance;
30 |     this.pendingStates = [];
31 |     this.pendingCallbacks = [];
32 |     this.isPending = false;
33 |     this.nextProps = this.nextContext = null;
34 |     this.clearCallbacks = this.clearCallbacks.bind(this);
35 | }
36 | Updater.prototype = {
37 |     emitUpdate(nextProps, nextContext) {
38 |         this.nextProps = nextProps;
39 |         this.nextContext = nextContext;
40 |         // receive nextProps!! should update immediately
41 |         nextProps || !updateQueue.isPending
42 |             ? this.updateComponent()
43 |             : updateQueue.add(this);
44 |     },
45 |     updateComponent() {
46 |         let { instance, pendingStates, nextProps, nextContext } = this;
47 |         if (nextProps || pendingStates.length > 0) {
48 |             nextProps = nextProps || instance.props;
49 |             nextContext = nextContext || instance.context;
50 |             this.nextProps = this.nextContext = null;
51 |             // merge the nextProps and nextState and update by one time
52 |             shouldUpdate(
53 |                 instance,
54 |                 nextProps,
55 |                 this.getState(),
56 |                 nextContext,
57 |                 this.clearCallbacks
58 |             );
59 |         }
60 |     },
61 |     addState(nextState) {
62 |         if (nextState) {
63 |             _.addItem(this.pendingStates, nextState);
64 |             if (!this.isPending) {
65 |                 this.emitUpdate();
66 |             }
67 |         }
68 |     },
69 | }
```

```

69  replaceState(nextState) {
70    let { pendingStates } = this;
71    pendingStates.pop();
72    // push special params to point out should replace state
73    _.addItem(pendingStates, [nextState]);
74  },
75  getState() {
76    let { instance, pendingStates } = this;
77    let { state, props } = instance;
78    if (pendingStates.length) {
79      state = _.extend({}, state);
80      pendingStates.forEach(nextState => {
81        let isReplace = _.isArr(nextState);
82        if (isReplace) {
83          nextState = nextState[0];
84        }
85        if (_.isFn(nextState)) {
86          nextState = nextState.call(instance, state, props);
87        }
88        // replace state
89        if (isReplace) {
90          state = _.extend({}, nextState);
91        } else {
92          _.extend(state, nextState);
93        }
94      });
95      pendingStates.length = 0;
96    }
97    return state;
98  },
99  clearCallbacks() {
100  let { pendingCallbacks, instance } = this;
101  if (pendingCallbacks.length > 0) {
102    this.pendingCallbacks = [];
103    pendingCallbacks.forEach(callback => callback.call(instance));
104  }
105  },
106  addCallback(callback) {
107    if (_.isFn(callback)) {
108      _.addItem(this.pendingCallbacks, callback);
109    }
110  }
111};
112

```

2. 为什么 React 不同步地更新 this.state? 在开始重新渲染之前, React 会有意地进行“等待”, 直到所有在组件的事件处理函数内调用的setState() 完成之后。这样可以通过避免不必要的重新渲染来提升性能。
3. 为什么setState是异步的? 这里的异步指的是多个state会合成到一起进行批量更新。
4. 为什么 setState只有在React合成事件和生命周期函数中是异步的, 在原生事件和setTimeout、setInterval、addEventListener中都是同步的?

setState 什么时候是异步的?

目前，在事件处理函数内部的 `setState` 是异步的。

例如，如果 `Parent` 和 `Child` 在同一个 `click` 事件中都调用了 `setState`，这样就可以确保 `Child` 不会被重新渲染两次。取而代之的是，React 会将该 state “冲洗” 到浏览器事件结束的时候，再统一地进行更新。这种机制可以在大型应用中得到很好的性能提升。

这只是一个实现的细节，所以请不要直接依赖于这种机制。在以后的版本当中，React 会在更多的情况下静默地使用 `state` 的批更新机制。

原生事件绑定不会通过合成事件的方式处理，自然也不会进入更新事务的处理流程。

`setState` 总结：

1. `setState()` 执行时，`updater` 会将 `partialState` 添加到它维护的 `pendingStates` 中，等到
2. `updateComponent` 负责合并 `pendingStates` 中所有 `state` 变成一个 `state`
3. `forceUpdate` 执行新旧 `vdom` 比对 - `diff` 以及实际更新操作

flow

flow 是一个针对 JavaScript 代码的静态类型检测器 (<https://zh-hans.reactjs.org/docs/static-type-checking.html>)。Flow 由 Facebook 开发，经常与 React 一起使用。flow 通过特殊的类型语法为变量、函数、以及 React 组件提供注解，帮助你尽早地发现错误。你可以阅读 introduction to Flow (<https://flow.org/en/docs/getting-started/>) 来了解它的基础知识。React 将 flow 引入源码，用于类型检查。在许可证头部的注释中，标记为 `@flow` 注释的文件是已经过类型检查的。

fiber

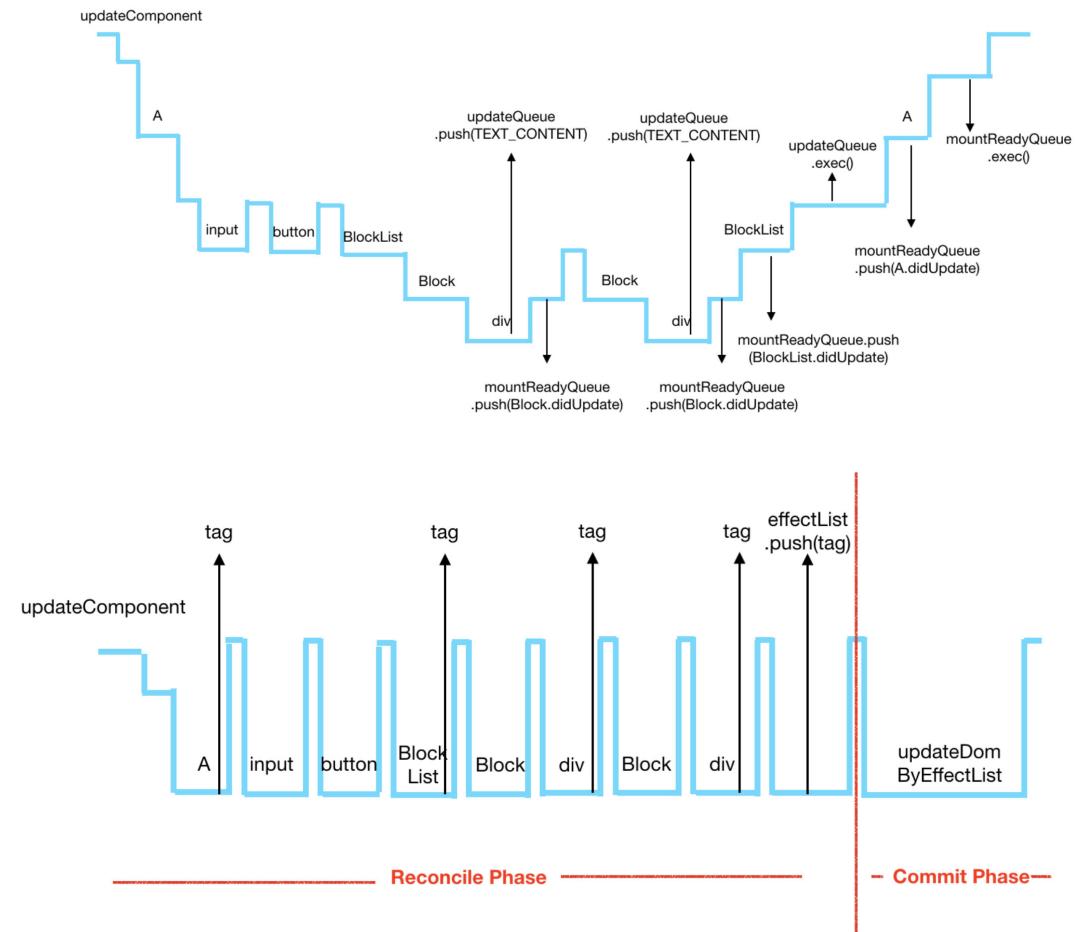
为什么需要fiber

React 的 killer feature：virtual dom

1. 为什么需要 fiber 对于大型项目，组件树会很大，这个时候递归遍历的成本就会很高，会造成主线程被持续占用，结果就是主线程上的布局、动画等周期性任务就无法立即得到处理，造成视觉上的卡顿，影响用户体验。
2. 任务分解的意义 解决上面的问题
3. 增量渲染（把渲染任务拆分成块，匀到多帧）
4. 更新时能够暂停，终止，复用渲染任务
5. 给不同类型的更新赋予优先级

6. 并发方面新的基础能力

7. 更流畅



什么是fiber

fiber是指组件上将要完成或者已经完成的任务，每个组件可以一个或者多个。

fiber数据结构

查看react/packages/react-reconciler/src/ReactFiber.js -Fiber **tag: WorkTag** 标记fiber的类型，类似我们之前写react核心api时候的vtype WorkTag地址: /react/packages/shared/ReactWorkTags.js **key: null | string** 当前子节点的唯一标识，ReactElement里面的key。 **elementType: any**, element.type的值，保存的是协调期间当前child的身份。也就是我们调用 createElement 的第一个参数。 **type: any** 当前fiber关联的function或者class。 **alternate: Fiber | null** fiber的一个合成版本。每一个更新过的fiber都会有一个alternate。 fiber的alternate是使用一个叫做 cloneFiber 的函数懒创建的(created lazily)。意思就是，如果这个fiber的alternate存在，那么 cloneFiber 不会去创建一个新的对象，而是复用这个fiber的alternate，从而最大程度地减少任务分配(allocations)。

stateNode: any, 当前fiber的本地状态。 **child: Fiber | null**, 第一个子节点，单向链表结构 **sibling: Fiber | null**, 下一个兄弟节点，单向链表结构。兄弟节点的return指向的是同一个父节点。 **index** 下标 **return: Fiber | null** 是指程序(program)在处理完当前fiber之后应当返回的fiber，它也可被认为是一个parent fiber。

pendingProps 与 memoizedProps

概念上来说，props是指函数的参数。一个fiber的pendingProps 被赋值在这个fiber执行(execution)的开始，memoizedProps 则是在执行完成时被赋值。当新传入的 pendingProps 等于 memoizedProps 的时候，这表示这个fiber 的上一个输出(output)可以被复用，从而避免了不必要的任务。

updateQueue: UpdateQueue | null,

用于state更新和回调的队列。 **memoizedState: any** 上一次渲染时候的state。

fiber reconciler

“fiber” reconciler 是一个新尝试，致力于解决 stack reconciler 中固有的问题，同时解决一些历史遗留问题。Fiber 从 React 16 开始变成了默认的 reconciler。它的主要目标是：

- 能够把可中断的任务切片处理。
- 能够调整优先级，重置并复用任务。
- 能够在父元素与子元素之间交错处理，以支持 React 中的布局。
- 能够在 render() 中返回多个元素。
- 更好地支持错误边界。

你可以在这里 (<https://github.com/acdlite/react-fiber-architecture>) 和这里 (<https://medium.com/react-in-depth/inside-fiber-in-depth-overview-of-the-new-reconciliation-algorithm-in-react-e1c04700ef6e>) ，深入了解 React Fiber 架构。

源代码在 packages/react-reconciler (<https://github.com/facebook/react/tree/master/packages/react-reconciler>) 目录下。

Hooks

1. Hooks是什么？为了拥抱正能量 函数式
2. Hooks带来的变革，让函数组件有了状态，可以替代class
3. 类似链表的实现原理

```
1 import React, { useState, useEffect } from "react";
2 function FunComp(props) {
3     const [data, setData] = useState("initialstate");
4     function handleChange(e) {
5         setData(e.target.value);
6     }
7
8     useEffect(() => {
9         //subscribeToSomething()
10        return () => {
11            //unSubscribeToSomething()
12        };
13    });
14    return <input value={data} onChange={handleChange} />;
15 }
16
```

```
1 function FunctionalComponent () {
2     const [state1, setState1] = useState(1)
3     const [state2, setState2] = useState(2)
4     const [state3, setState3] = useState(3)
5 }
6 hook1 => Fiber.memoizedState
7 state1 === hook1.memoizedState
8 hook1.next => hook2
9 state2 === hook2.memoizedState
10 hook2.next => hook3
11 state3 === hook2.memoizedState
```

事件系统

React 实现一个合成事件，这与渲染器无关，它适用于 ReactDOM 和 React Native。

树形组件设计与实现

设计思路

递归：自己调用自己 如计算 $f(n)=f(n-1)*n$; $n>0$, $f(1)=1$

```
1 function foo(n) {
2     return n === 1 ? 1 : n * foo(n - 1)
3 }
```

react中实现递归组件更加纯粹，就是组件递归渲染即可。假设我们的节点组件是TreeNode，它的render中只要发现当前节点拥有子子节点就要继续渲染自己。节点的打开状态可以通过给组件一个open状态来维护。

实现

//TreePage.js

```
1 import React, { Component } from "react";
2 import TreeNode from "../../components/TreeNode";
3 //数据源
4 const treeData = {
5     key: 0, //标识唯一性
6     title: "全国", //节点名称显示
7     children: [
8         //子节点数组
9         {
10             key: 6,
11             title: "北北方区域",
12             children: [
13                 {
14                     key: 1,
15                     title: "黑黑龙江省",
16                     children: [
17                         {
18                             key: 6,
19                             title: "哈尔滨"
20                         }
21                     ]
22                 ]
23             ]
24         }
25     ]
26 }
```

```

22     },
23     {
24         key: 2,
25         title: "北北京"
26     }
27 ]
28 },
29 {
30     key: 3,
31     title: "南方方区域",
32     children: [
33     {
34         key: 4,
35         title: "上海海"
36     },
37     {
38         key: 5,
39         title: "深圳"
40     }
41 ]
42 }
43 ];
44 };
45 export default class TreePage extends Component {
46     render() {
47         return (
48             <div>
49                 <h1>TreePage</h1>
50                 <TreeNode data={treeData} />
51             </div>
52         );
53     }
54 }
55

```

TreeNode.js

```

1 import React, { Component } from "react";
2 import classNames from "classnames"; //先安装下
3 // npm install classnames
4 export default class TreeNode extends Component {
5     constructor(props) {
6         super(props);
7         this.state = {
8             expanded: false
9         };
10    }
11    handleExpanded = () => {
12        this.setState({
13            expanded: !this.state.expanded
14        });
15    };
16    render() {
17        const { title, children } = this.props.data;
18        const { expanded } = this.state;
19        const hasChildren = children && children.length > 0;
20        return (

```

```
21      <div>
22        <div className="nodeInner" onClick={this.handleExpanded}>
23          {hasChildren && (
24            <i
25              className={classnames("tri", expanded ? "tri-open" : "tri-
close")}>
26            </i>
27          )} {title}</span>
28        </div>
29        {expanded && hasChildren && (
30          <div className="children">
31            {children.map(item => {
32              return <TreeNode key={item.key} data={item} />;
33            })
34          </div>
35        )} {</div>}
36      );
37    );
38  );
39 }
40 }
41 import React, { Component } from "react";
42 import classnames from "classnames"; //先安装下
43 // npm install classnames
44 export default class TreeNode extends Component {
45   constructor(props) {
46     super(props);
47     this.state = {
48       expanded: false
49     };
50   }
51   handleExpanded = () => {
52     this.setState({
53       expanded: !this.state.expanded
54     });
55   };
56   render() {
57     const { title, children } = this.props.data;
58     const { expanded } = this.state;
59     const hasChildren = children && children.length > 0;
60     return (
61       <div>
62         <div className="nodeInner" onClick={this.handleExpanded}>
63           {hasChildren && (
64             <i
65               className={classnames("tri", expanded ? "tri-open" : "tri-
close")}>
66             </i>
67           )} {title}</span>
68         </div>
69         {expanded && hasChildren && (
70           <div className="children">
71             {children.map(item => {
72               return <TreeNode key={item.key} data={item} />;
73             })
74           </div>
75         )} {</div>}
76       );
77     }
78   }
79 }
```

```
77     </div>
78   );
79 }
80 }
81 }
```

```
1 /*树组件css*/
2 /* 树组件css */
3 .nodeInner {
4   cursor: pointer;
5 }
6 .children {
7   margin-left: 20px;
8 }
9 .tri {
10  width: 20px;
11  height: 20px;
12  margin-right: 2px;
13  padding-right: 4px;
14 }
15 .tri-close:after,
16 .tri-open:after {
17   content: "";
18   display: inline-block;
19   width: 0;
20   height: 0;
21   border-top: 6px solid transparent;
22   border-left: 8px solid black;
23   border-bottom: 6px solid transparent;
24 }
25 .tri-open:after {
26   transform: rotate(90deg);
27 }
28 }
```

常见组件技术

核心：减少不必要的渲染，只渲染需要被渲染的组件

定制组件的shouldComponentUpdate钩子

范例：通过shouldComponentUpdate优化组件

```
1 import React, { Component } from "react";
2 export default class CommentListPage extends Component {
3   constructor(props) {
4     super(props);
5     this.state = {
6       commentList: []
7     };
8   }
9   componentDidMount() {
10     this.timer = setInterval(() => {
11       this.setState({
12         commentList: [
13           {
```

```

14         id: 0,
15         author: "小小明",
16         body: "这是小小明写的文文章"
17     },
18     {
19         id: 1,
20         author: "小小红",
21         body: "这是小小红写的文文章"
22     }
23   ]
24 );
25 }, 1000);
26 }
27 render() {
28   const { commentList } = this.state;
29   return (
30     <div>
31       <h1>CommentListPage</h1>
32       {commentList.map(item => {
33         return <Comment key={item.id} data={item} />;
34       })}
35     </div>
36   );
37 }
38 }
39 class Comment extends Component {
40   shouldComponentUpdate(nextProps, nextState) {
41     const { author, body } = this.props.data;
42     const { author: newAuthor, body: newBody } = nextProps.data;
43     if (author === newAuthor && body === newBody) {
44       return false; //如果不执行行行这里里里，将会多次
45       render;
46     }
47     return true;
48   }
49   render() {
50     const { author, body } = this.props.data;
51     return (
52       <div className="border">
53         <p>{author}</p>
54         <p>{body}</p>
55       </div>
56     );
57   }
58 }
59 
```

PureComponent

定制了shouldComponentUpdate后的Component

```

1 import React, { Component, PureComponent } from "react";
2 export default class PureComponentPage extends PureComponent {
3   constructor(props) {
4     super(props);
5     this.state = {
6       counter: 0
7     }
8   }
9   shouldComponentUpdate(nextProps, nextState) {
10     const { author, body } = this.props.data;
11     const { author: newAuthor, body: newBody } = nextProps.data;
12     if (author === newAuthor && body === newBody) {
13       return false; //如果不执行行行这里里里，将会多次
14       render;
15     }
16     return true;
17   }
18   render() {
19     const { author, body } = this.props.data;
20     return (
21       <div className="border">
22         <p>{author}</p>
23         <p>{body}</p>
24       </div>
25     );
26   }
27 }
28 
```

```
7      // obj: {
8      // num: 2,
9      // },
10     };
11   }
12   setCounter = () => {
13     this.setState({
14       counter: 100
15       // obj: {
16       // num: 200,
17       // },
18     });
19   };
20   render() {
21     const { counter, obj } = this.state;
22     console.log("render");
23     return (
24       <div>
25         <h1>PuerComponentPage</h1>
26         <div onClick={this.setCounter}>
27           counter:
28           {counter}
29         </div>
30       </div>
31     );
32   }
33 }
34 }
```

缺点是必须要用class形式，而且要注意是浅比较, 路径: /react/packages/shared/shallowEqual.js

```

/** 
 * Performs equality by iterating through keys on an object and returning false
 * when any key has values which are not strictly equal between the arguments.
 * Returns true when the values of all keys are strictly equal.
 */
function shallowEqual(objA: mixed, objB: mixed): boolean {
  if (Object.is(objA, objB)) {
    return true;
  }
  if (
    typeof objA !== 'object' ||
    objA === null ||
    typeof objB !== 'object' ||
    objB === null
  ) {
    return false;
  }
  const keysA = Object.keys(objA);
  const keysB = Object.keys(objB);
  if (keysA.length !== keysB.length) {
    return false;
  }
  for (let i = 0; i < keysA.length; i++) {
    if (
      !hasOwnProperty.call(objB, keysA[i]) ||
      !Object.is(objA[keysA[i]], objB[keysA[i]])
    ) {
      return false;
    }
  }
  return true;
}

```

React.memo

React.memo(...) 是React v16.6引进来的新属性，是一个高阶组件。它的作用和 React.PureComponent 类似，是用来控制函数组件的重新渲染的。 React.memo(...) 其实就是函数组件的 React.PureComponent。

```

1 import React, { Component, memo } from "react";
2 export default class ReactMemoPage extends Component {
3   constructor(props) {
4     super(props);
5     this.state = {
6       date: new Date(),
7       counter: 0
8     };
9   }
10  componentDidMount() {
11    this.timer = setInterval(() => {
12      this.setState({
13        date: new Date()
14        //counter: this.state.counter + 1,
15      });
16    }, 1000);
}

```

```

17 }
18 componentWillUnmount() {
19   clearInterval(this.timer);
20 }
21 render() {
22   const { counter, date } = this.state;
23   console.log("render", counter);
24   return (
25     <div>
26       <h1>ReactMemoPage</h1>
27       <p>{date.toLocaleTimeString()}</p>
28       <MemoCounter counter={counter} />
29     </div>
30   );
31 }
32 }
33 const MemoCounter = memo(props => {
34   console.log("MemoCounter");
35   return <div>{props.counter}</div>;
36 });
37

```

useMemo

把“创建”函数和依赖项数组作为参数传入 useMemo，它仅会在某个依赖项改变时才重新计算 memoized 值。这种优化有助于避免在每次渲染时都进行高开销的计算。

```

1 import React, { useState, useMemo } from "react";
2 export default function ReactMemoPage(props) {
3   const [counter, setCounter] = useState(0);
4   const [val, setValue] = useState("");
5   const expensive = useMemo(() => {
6     console.log("compute");
7     return counter * 100;
8   }, [counter]);
9   // const expensive = () => {
10   //   console.log("compute");
11   //   return counter * 100;
12   // };
13   return (
14     <div>
15       <p>{expensive}</p>
16       <div>
17         <button onClick={() => setCounter(counter + 1)}>counter</button>
18         <input value={val} onChange={event => setValue(event.target.value)} />
19       </div>
20     </div>
21   );
22 }
23

```

useCallback

把内联回调函数及依赖项数组作为参数传入 useCallback，它将返回该回调函数的 memoized 版本，该回调函数仅在某个依赖项改变时才会更新。当你把回调函数传递给经过优化的并使用引用相等性去避免非必要渲染（例如shouldComponentUpdate）的子组件时，它将非常有用。

```
1 import React, { useState, useCallback } from "react";
2 export default function ReactMemoPage() {
3   const [val, setVal] = useState("");
4   const [counter, setCounter] = useState(0);
5   const addClick = useCallback(() => {
6     setCounter(counter + 1);
7   }, [counter]);
8   // const addClick = () => {
9   //   setCounter(counter + 1);
10  // };
11  console.log("----");
12  return (
13    <div>
14      <input value={val} onChange={event => setVal(event.target.value)} />
15      <button onClick={addClick}>{counter}</button>
16    </div>
17  );
18}
19
```

useCallback(fn, deps) 相当于 useMemo(() => fn,deps)。

注意

依赖项数组不会作为参数传给“创建”函数。虽然从概念上来说它表现为：所有“创建”函数中引用的值都应该出现在依赖项数组中。未来编译器会更加智能，届时自动创建数组将成为可能。

Generator

Generator 函数是 ES6 提供的一种异步编程解决方案，语法行为与传统函数完全不同，详细参考文章。

1. function关键字与函数名之间有一个*;
2. 函数体内部使用yield表达式，定义不同的内部状态。
3. yield表达式只能在 Generator 函数里使用，在其地方会报错。

```
1 function* helloworldGenerator() {
2   yield 'hello';
3   yield 'world';
4   return 'ending';
5 }
6 var hw = helloworldGenerator();
7 //执行
8 console.log(hw.next());
9 console.log(hw.next());
10 console.log(hw.next());
11 console.log(hw.next());
```

由于 Generator 函数返回的遍历器对象，只有调用 next 方法才会遍历下一个内部状态，所以其实提供了了一种可以暂停执行的函数。yield 表达式就是暂停标志。

redux-saga

- 概述：redux-saga 是一个用于管理应用程序 Side Effect（副作用，例如异步获取数据，访问浏览器缓存等）的 library，它的目标是让副作用管理更容易，执行更高效，测试更简单，在处理故障时更容易。
- 地址：<https://github.com/redux-saga/redux-saga>
- 安装：npm install --save redux-saga
- 使用：用户登录

src/App

```
1 import React, { Component } from "react";
2 import { BrowserRouter, Link, Route, Switch } from "react-router-dom";
3 import HomePage from "./pages/HomePage";
4 import UserPage from "./pages/UserPage";
5 import LoginPage from "./pages/LoginPage";
6 import PrivateRoute from "./pages/PrivatePage";
7 import { connect } from "react-redux";
8 class App extends Component {
9   render() {
10     const { userName } = this.props;
11     return (
12       <div className="App">
13         <BrowserRouter>
14           <Link to="/">首首页页</Link>
15           <Link to="/user">个人中心心</Link>
16           <Link to={userName ? "/user" : "/login"}>{userName || "登录"}</Link>
17         <Switch>
18           <Route path="/" exact component={HomePage} />
19           {/* <Route path="/user" component={UserPage} /> */}
20           <Route path="/login" component={LoginPage} />
21           <PrivateRoute path="/user" component={UserPage} />
22         </Switch>
23       </BrowserRouter>
24     </div>
25   );
26 }
27 }
28 }
29 export default connect(state => ({
30   userName: state.userName
31 }))(App);
32 }
```

创建store/index.js

```
1 import { createStore, applyMiddleware } from "redux";
2 import thunk from "redux-thunk";
3 const loginInfo = {
4   isLogin: false,
5   loading: false,
6   userName: ""
7 };
8 function LoginReducer(state = { ...loginInfo }, action) {
```

```

9  console.log("action", action.payload);
10 switch (action.type) {
11   case "loginRequest":
12     return { ...state, ...loginInfo, loading: true };
13   case "loginSuccess":
14     return { ...state, isLoggedIn: true, loading: false,
15             ...action.payload };
16   case "loginFailure":
17     return { ...state, ...loginInfo, ...action.payload };
18   default:
19     return state;
20 }
21 const store = createStore(loginReducer, applyMiddleware(thunk));
22 export default store;
23

```

登录页面pages/LoginPage.js

```

1 import React, { Component } from "react";
2 import { Route, Redirect } from "react-router-dom";
3 import { connect } from "react-redux";
4 class LoginPage extends Component {
5   constructor(props) {
6     super(props);
7     this.state = { userName: "" };
8   }
9   render() {
10     const { isLogin, location, login, loading, err } = this.props;
11     const { redirect = "/" } = location.state || {};
12     if (isLogin) {
13       return <Redirect to={redirect} />;
14     }
15     return (
16       <div>
17         <h3>LoginPage</h3>
18         <input
19           value={this.state.userName}
20           onChange={event =>
21             this.setState({
22               userName: event.target.value
23             })
24           }
25         />
26         <button onClick={() => login(this.state.userName)}>
27           {loading ? "loading..." : "login"}
28         </button>
29         <p>{err}</p>
30       </div>
31     );
32   }
33 }
34 export default connect(
35   state => ({
36     isLogin: state.isLoggedIn,
37     loading: state.loading,
38     err: state.error
39   })
40 );

```

```

39 },
40 {
41   login: userName => dispatch => {
42     dispatch({ type: "loginRequest" });
43     setTimeout(() => {
44       dispatch({ type: "loginsuccess", payload: { userName } });
45     }, 1000);
46   }
47 }
48 )(LoginPage);
49

```

路由守卫/pages/PrivatePage.js:

```

1 import React, { Component } from "react";
2 import { Route, Redirect } from "react-router-dom";
3 import { connect } from "react-redux";
4 class PrivateRoute extends Component {
5   render() {
6     const { isLogin, path, component } = this.props;
7     if (isLogin) {
8       return <Route path={path} component={component} />;
9     }
10    return (
11      <Redirect
12        to={{
13          pathname: "/login",
14          state: {
15            redirect: path
16          }
17        }}
18      />
19    );
20  }
21 }
22 export default connect(
23   state => ({
24     isLogin: state.isLogin
25   }),
26   {}
27 )(PrivateRoute);
28

```

用saga的方式实现:

1.创建一个./store/mySaga.js处理用户登录请求 call: 调用用异步操作 put: 状态更新 takeEvery: 做saga监听

```

1 import { call, put, takeEvery } from "redux-saga/effects";
2 // 模拟登录接口
3 const UserService = {
4   login(userName) {
5     return new Promise((resolve, reject) => {
6       setTimeout(() => {
7         if (userName === "小小明") {
8           resolve({ userName: "小小明" });

```

```

9     } else {
10       reject({ err: "用户名或密码错误" });
11     }
12   }, 1000);
13 );
14 }
15 };
16 //worker saga
17 function* loginHandle(action) {
18   try {
19     yield put({ type: "loginRequest" });
20     //登录
21     const res = yield call(UserService.login, action.payload.userName);
22     console.log("res", res);
23     yield put({ type: "loginSuccess", payload: { ...res } });
24   } catch (err) {
25     yield put({ type: "loginFailure", payload: { ...err } });
26   }
27 }
28 //watcher saga
29 function* mySaga(props) {
30   yield takeEvery("login", loginHandle);
31 }
32 export default mySaga;
33

```

2.注册redux-saga, ./store/index.js

```

1 import { createStore, applyMiddleware } from "redux";
2 import thunk from "redux-thunk";
3 const loginInfo = {
4   isLogin: false,
5   loading: false,
6   userName: ""
7 };
8 function loginReducer(state = { ...loginInfo }, action) {
9   console.log("action", action.payload);
10  switch (action.type) {
11    case "loginRequest":
12      return { ...state, ...loginInfo, loading: true };
13    case "loginSuccess":
14      return { ...state, isLogin: true, loading: false, ...action.payload };
15    case "loginFailure":
16      return { ...state, ...loginInfo, ...action.payload };
17    default:
18      return state;
19  }
20 }
21 const store = createStore(loginReducer, applyMiddleware(thunk));
22 export default store;
23

```

3.测试, LoginPage.js

```

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

```

```

2 import { Route, Redirect } from "react-router-dom";
3 import { connect } from "react-redux";
4 class LoginPage extends Component {
5   constructor(props) {
6     super(props);
7     this.state = { userName: "" };
8   }
9   render() {
10   const { isLogin, location, login, loading, err } = this.props;
11   const { redirect = "/" } = location.state || {};
12   if (isLogin) {
13     return <Redirect to={redirect} />;
14   }
15   return (
16     <div>
17       <h3>LoginPage</h3>
18       <input
19         value={this.state.userName}
20         onChange={event =>
21           this.setState({
22             userName: event.target.value
23           })
24         }
25       />
26       <button onClick={() => login(this.state.userName)}>
27         {loading ? "loading..." : "login"}
28       </button>
29       <p>{err}</p>
30     </div>
31   );
32 }
33 }
34 export default connect(
35   state => ({
36     isLogin: state.isLogin,
37     loading: state.loading,
38     err: state.err
39   }),
40   {
41     login: userName => dispatch => {
42       dispatch({ type: "loginRequest" });
43       setTimeout(() => {
44         dispatch({ type: "loginSuccess", payload: { userName } });
45       }, 1000);
46     }
47   }
48 )(LoginPage);
49

```

redux-saga基于generator实现，使用前搞清楚generator 相当重要

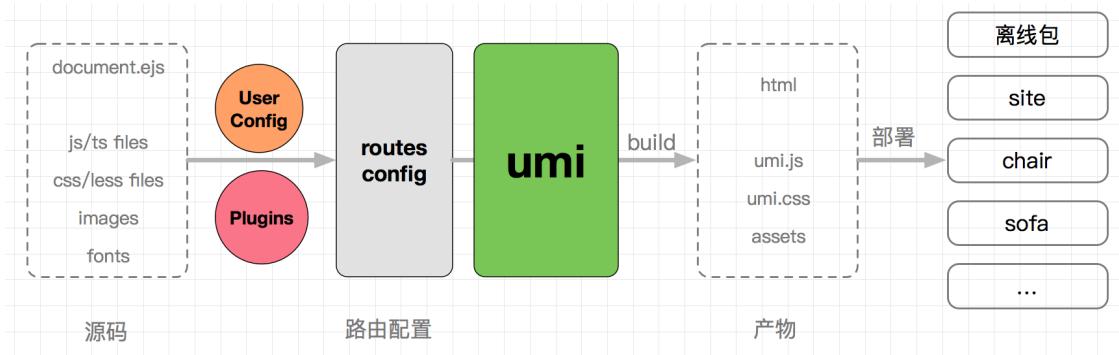
redux-saga 使用了 ES6 的 Generator 功能，让异步的流程更易于读取，写入和测试。（如果你还不熟悉的话，这里有一些介绍性的链接）通过这样的方式，这些异步的流程看起来就像是标准同步的 Javascript 代码。（有点像async / await，但 Generator 还有一些更棒而且我们也需要的功能）。不同于 redux thunk，你不会再遇到回调地狱了了，你可以很容易地测试异步流程并保持你的 action 是干净的，因此我们可以说redux-saga更擅长解决复杂异步这样的场景，也更便于测试。

umi

umi，中文可发音为乌米，是一个可插拔的企业级 react 应用框架。

why umi

- **开箱即用**，内置 react、react-router 等
- **类 next.js 且功能完备的路由约定**，同时支持配置的路由方式
- **完善的插件体系**，覆盖从源码到构建产物的每个生命周期
- **高性能**，通过插件支持 PWA、以路由为单元的 code splitting 等
- **支持静态页面导出**，适配各种环境，比如中台业务、无线业务、[egg](#)、支付宝钱包、云凤蝶等
- **开发启动快**，支持一键开启 [dll](#) 等
- **一键兼容到 IE9**，基于 [umi-plugin-polyfills](#)
- **完善的 TypeScript 支持**，包括 d.ts 定义和 umi test
- **与 dva 数据流的深入融合**，支持 duck directory、model 的自动加载、code splitting 等等



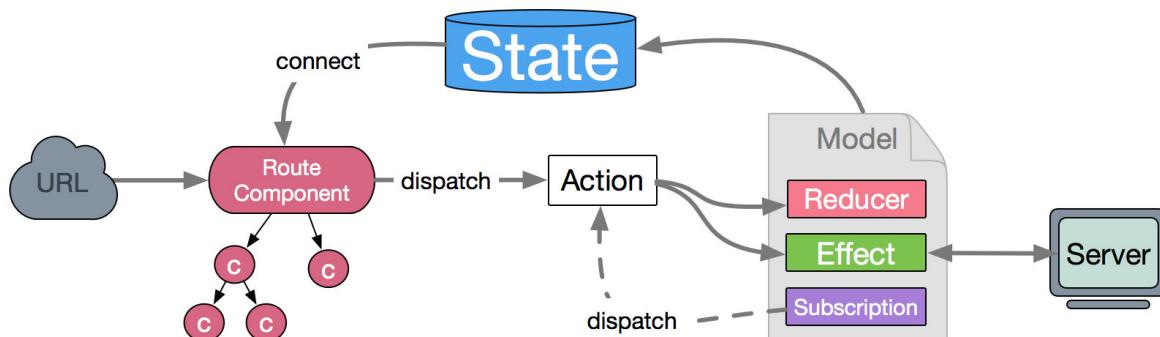
umi和dva、roadhog是什么关系？

简单来说，

- roadhog 是基于 webpack 的封装工具，目的是简化 webpack 的配置
- umi 可以简单地理解为 roadhog + 路由，思路类似 next.js/nuxt.js，辅以一套插件机制，目的是通过框架的方式简化 React 开发
- dva 目前是纯粹的数据流，和 umi 以及 roadhog 之间并没有相互的依赖关系，可以分开使用也可以一起使用，个人觉得 [umi + dva 是比较搭的](#)

dva

dva 首先是一个基于 [redux](#) 和 [redux-saga](#) 的数据流方案，然后为了简化开发体验，dva 还额外内置了 [react-router](#) 和 [fetch](#)，所以也可以理解为一个轻量级的应用框架。



dva+umi的约定

- src源码
 - pages页面
 - components组件
 - layout布局
- model
- config配置
- mock数据模拟
- test测试等

```
.
├── dist/                                // 默认的 build 输出目录
└── mock/                               // mock 文件所在目录，基于 express
├── config/
  └── config.js                          // umi 配置，同 .umirc.js，二选一
└── src/
  ├── layouts/index.js                 // 全局布局
  ├── pages/                            // 页面目录，里面的文件即路由
  │   ├── .umi/                           // dev 临时目录，需添加到 .gitignore
  │   ├── .umi-production/              // build 临时目录，会自动删除
  │   ├── document.ejs                  // HTML 模板
  │   ├── 404.js                         // 404 页面
  │   ├── page1.js                      // 页面 1，任意命名，导出 react 组件
  │   ├── page1.test.js                 // 用例文件，umi test 会匹配所有 .test.js 和 .e2e.js 结尾
  │   └── page2.js                      // 页面 2，任意命名
  ├── global.css                        // 约定的全局样式文件，自动引入，也可以用 global.less
  ├── global.js                         // 可以在这里加入 polyfill
  ├── app.js                            // 运行时配置文件
  ├── .umirc.js                         // umi 配置，同 config/config.js，二选一
  ├── .env                               // 环境变量
  └── package.json
```

安装(<https://pro.ant.design/docs/getting-started-cn#%E5%A1%E8%A8%A3%85>)

环境要求：node版本>=8.10

```
1 antd-pro 安装:
2 新建立一个空文件夹: mkdir lesson8-umi
3 进入文件夹: cd lesson8-umi
4 创建: yarn create umi
5 选择ant-design-pro
6 选择Javascript
7 安装依赖: yarn
8 启动: yarn start或者umi dev
```

其他例子：如umi-antd-mobile等

umi基本使用

建立pages下面的单页面about:

```
1 | umi g page about
```

建立文件夹channel(默认是css):

```
1 | umi g page channel/index --less
```

import router from 'umi/router' 跳转 router.push('/user/2')

起服务看效果

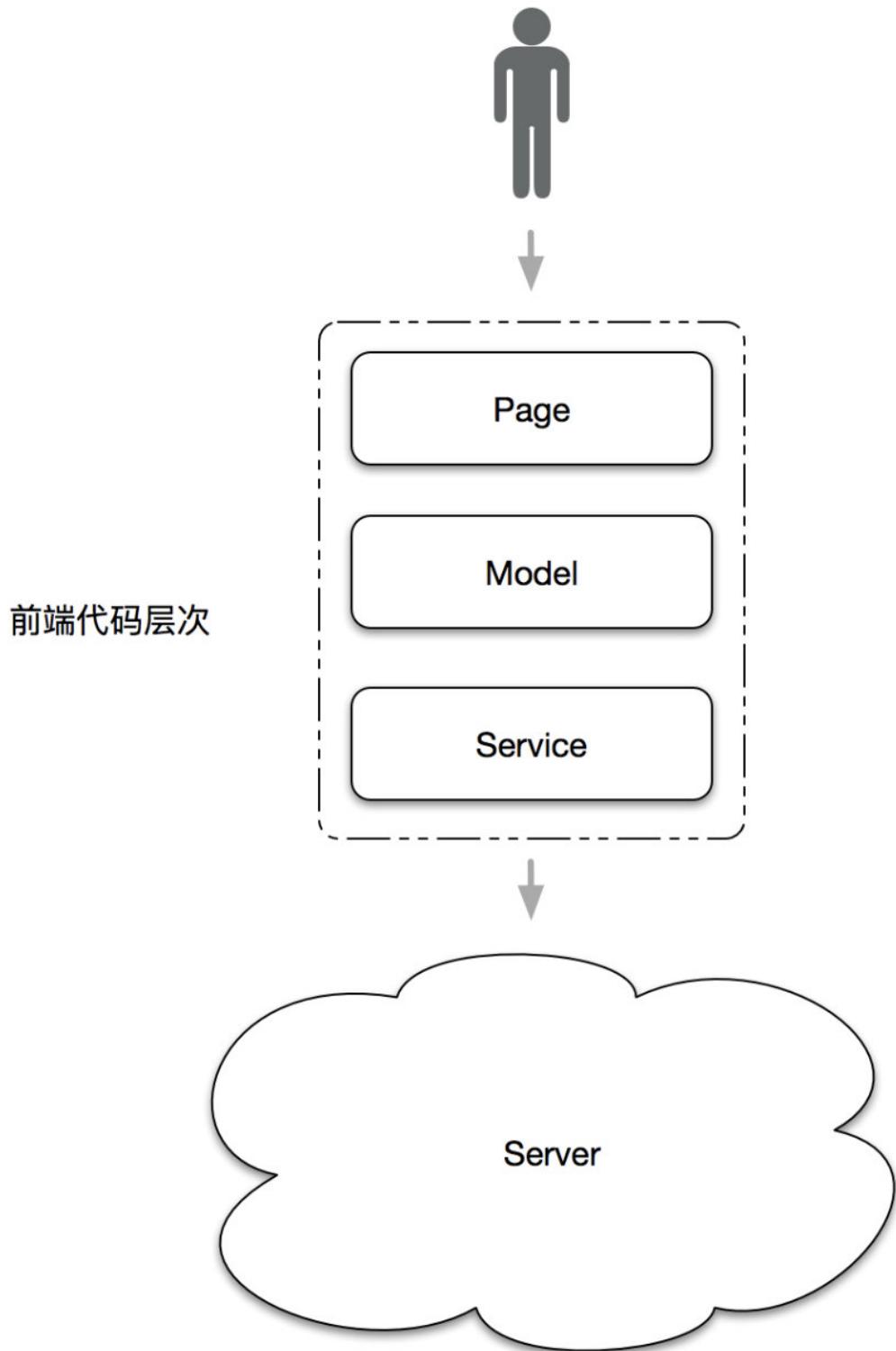
```
1 | umi dev
```

访问index: <http://localhost:8000/>

访问about: <http://localhost:8000/about>

理解dva

软件分层：回顾react，为了让数据流更易于维护，我们分成了store，reducer，action等模块，各司其职，软件开发也是一样



1. Page 负责与用户直接打交道：渲染页面、接受用户的操作输入，侧重于展示型交互性逻辑。
2. Model 负责处理业务逻辑，为 Page 做数据、状态的读写、变换、暂存等
3. Service 负责与 HTTP 接口对接，进行纯粹的数据读写

DVA 是基于 redux、redux-saga 和 react-router 的轻量级前端框架及最佳实践沉淀，核心api如下：

1. model
 - state 状态
 - action
 - dispatch
 - reducer

- effect 副作用，处理异步

2. subscriptions 订阅

3. router 路由

注意

1. namespace : model 的命名空间，只能用字符串。一个大型应用可能包含多个 model，通过 namespace 区分
2. reducers : 用于修改 state，由 action 触发。reducer 是一个纯函数，它接受当前的 state 及一个action 对象。action 对象里面可以包含数据体 (payload) 作为入参，需要返回一个新的 state。
3. effects : 用于处理异步操作（例如：与服务端交互）和业务逻辑，也是由 action 触发。但是，它不可以修改 state，要通过触发 action 调用 reducer 实现对 state 的间接操作。
4. action : 是 reducers 及 effects 的触发器，一般是一个对象，形如 { type: 'add', payload: todo }，通过 type 属性可以匹配到具体某个 reducer 或者effect，payload 属性则是数据体，用于传送给reducer 或 effect。

实例：

实现如下图

姓名	年龄	住址
名字0	0	城市0
名字1	1	城市1
名字2	2	城市2
名字3	3	城市3
名字4	4	城市4
名字5	5	城市5

使用状态：state + connect

- 创建页面channel.js: umi g page channel/index --less

```

1 import React, { Component } from 'react';
2 import styles from './index.less';
3 import { PageHeaderWrapper } from '@ant-design/pro-layout';
4 import { Card, Form, Input, Button, Table }
5 from 'antd';
6 import { connect } from 'dva';
7 const columns = [
8 {

```

```
9  title: '姓名',
10 dataIndex: 'name',
11 key: 'name',
12 },
13 {
14 title: '年齡',
15 dataIndex: 'age',
16 key: 'age',
17 },
18 {
19 title: '住址',
20 dataIndex: 'city',
21 key: 'city',
22 },
23 ];
24 class Channel extends Component {
25 componentDidMount() {
26   this.props.getChannelData();
27 }
28   search = () => {
29     const { getFieldValue } = this.props.form;
30     const name = getFieldValue('name');
31     this.props.getChannelDataBySearch({ name
32 });
33 };
34 render() {
35   const { form, data } = this.props;
36   const { getFieldDecorator } = form;
37   return (
38     <div className={styles.channel}>
39       <PageHeaderWrapper>
40         <Card className={styles.formCard}>
41           <Form>
42             <Form.Item label="姓名">
43               {getFieldDecorator('name')}(<Input />)
44             </Form.Item>
45             <Form.Item label="城市">
46               {getFieldDecorator('city')}(<Input />)
47             </Form.Item>
48           </Form>
49           <Button type="primary" onClick=
50             {this.search}>
51             提交
52           </Button>
53           <Button>重置</Button>
54         </Card>
55         <Card>
56           <Table dataSource={data} columns=
57             {columns} rowKey={record => record.id} />
58         </Card>
59       </PageHeaderWrapper>
60     </div>
61   );
62 }
63 }
64 export default connect(({ channel }) => ({
65   ...channel }), {
66   getChannelData: () => ({ type:
```

```
67 'channel/getChannelData' }),
68 getChannelDataBySearch: search => ({ type:
69   'channel/getChannelDataBySearch', payload:
70   search }),
71 })(Form.create()(channel));
```

- 更新模型src/models/channel.js

```
1 import { getChannelData, getChannelDataBySearch } from
2   '@/services/channel.js';
3 const model = {
4   namespace: 'channel',
5   state: {
6     data: [],
7   },
8   effects: {
9     *getChannelData({ payload }, { call, put }) {
10       const response = yield
11       call(getChannelData, payload);
12       yield put({
13         type: 'channelData',
14         payload: response,
15       });
16     },
17     *getChannelDataBySearch({ payload }, {
18       call, put }) {
19       const response = yield
20       call(getChannelDataBySearch, payload);
21       console.log('has', response, payload);
22       yield put({
23         type: 'channelData',
24         payload: response,
25       });
26     },
27   },
28   reducers: {
29     channelData(state, { payload }) {
30       return { ...state, data:
31         [...payload.data] };
32     },
33   },
34 };
35 export default model;
```

- 添加服务: src/service/channel.js

```
1 import request from '@/utils/request';
2 export async function getChannelData(params)
3 {
4   return request('/api/getChannelData', {
5     method: 'get',
6   });
7 }
8 export async function
9   getChannelDataBySearch(params) {
```

```
10 | return
11 | request('/api/getChannelDataBySearch', {
12 |   method: 'post',
13 |   data: params,
14 | });
15 | }
```

数据mock：模拟数据接口

mock目录和src平级，新建mock/channel.js

```
1 | const channelTableData = [];
2 | for (let i = 0; i < 10; i++) {
3 |   channelTableData.push({
4 |     id: i,
5 |     name: '名字' + i,
6 |     age: i,
7 |     city: '城市' + i,
8 |   });
9 | }
10 | function searchChannelData(name) {
11 |   const res = [];
12 |   for (let i = 0; i < 10; i++) {
13 |     if (channelTableData[i].name.indexOf(name)
14 |       > -1) {
15 |       res.push(channelTableData[i]);
16 |     }
17 |   }
18 |   return res;
19 | }
20 | export default {
21 |   // 支持值为 Object 和 Array
22 |   'GET /api/getChannelData': {//查询表单数据
23 |     data: [...channelTableData],
24 |   },
25 |   'POST /api/getChannelDataBySearch': (req,
26 |     res) => {//搜索
27 |     res.send({
28 |       status: 'ok',
29 |       data: searchChannelData(req.body.name),
30 |     });
31 |   },
32 | }
```

移动端cra项目简介

所用技术：react、redux、react-redux、react-router-dom 等等
项目安装：npm install
项目启动：npm start
mock: cd mock-server npm i npm start

移动端适配(<https://github.com/kaola-fed/blog/issues/133>)

React16

虚拟dom setState React15的问题

1. react diff的时间，可能会让浏览器卡顿
2. 60fps 一帧大概在16.6ms

fiber

一种全新的描述虚拟dom的数据结构，让diff的过程，可以被中断。计算diff的时候，一旦又优先级更高的任务，比如动画，用户交互加入后，会终端diff，把js的主线程控制权换回去，进行浏览器的渲染，下次空闲的时候，再继续计算。

```
{  
  type:"div", function,  
  props:{  
    id  
    children: [ {type,children.....} ]  
  }  
}
```

树微观改造成了链表，之前只能用props.children递归来遍历，不好中断，利用链表的特性，就像打麻将一样，出牌顺序固定，

```
{ type:div or function, child:'第一个子元素' sibling: 下一个兄弟元素 return 父元素 }
```

这种递归，children会嵌套N层，中间的状态太多，很难中断

元素=》子元素 子元素没了，找下一个兄弟元素 下一个兄弟元素没了，返回父元素 遍历走向基本是单项的，任何节点终止，都可以随时继续

requestIdleCallback

利用浏览器的空闲时间，执行函数window.requestIdleCallback(计算函数，) 浏览器流畅基本都是60FPS，1s渲染60次 1帧16.6ms，一帧的时间内，有渲染页面，回流重绘等，还会有空闲时间

```
function 计算函数(idleDeadline){ idleDeadline 仕idleCallback函数提供给你的  
idleDeadline.timeRemaining() 获取当前帧还剩下的时间
```

```
}
```

```
1 | function createElement(type, props, ...children) {  
2 |   delete props.__source;  
3 |   delete props.__self;  
4 |   return {  
5 |     type,  
6 |     props: {  
7 |       ...props,  
8 |       children: children.map(child => {  
9 |         return typeof child === "object" ? child :  
10 |           createTextElement(child);  
11 |       })  
12 |     };  
13 |   }  
14 | }  
15 | function createTextElement(text) {  
16 |   return {  
17 |     type: "TEXT",  
18 |     props: {  
19 |       nodevalue: text,
```

```
20     children: []
21   }
22 };
23 }
24
25
26
27 function render(vdom, container) {
28   // container.innerHTML=`<pre>${JSON.stringify(vdom,null,2)}</pre>` 
29   wipRoot = {
30     dom: container,
31     props: {
32       child: [vdom]
33     },
34     base: currentRoot
35   };
36
37   //vdom需要重新的构建
38   nextUnitOfWork = wipRoot;
39 }
40
41 function createDom(vdom) {
42   const dom = (vdom.type = "TEXT"
43     ? document.createTextNode("")
44     : document.createElement(vdom.type));
45   updateDom(dom, {}, vdom.props);
46   return dom;
47 }
48 //设置dom属性
49 function updateDom(dom, prevProps, nextProps) {
50   //删除
51   Object.keys(prevProps)
52     .filter(name => name !== "children")
53     .filter(name => !(name in nextProps))
54     .forEach(name => {
55       if (name.slice(0, 2) === "on") {
56         dom.removeEventListener(
57           name.slice(2).toLowerCase(),
58           prevProps[name],
59           false
60         );
61       } else {
62         dom[name] = "";
63       }
64     });
65 //新增
66 Object.keys(nextProps)
67   .filter(name => name !== "children")
68   .forEach(name => {
69     if (name.slice(0, 2) === "on") {
70       dom.addEventListener(
71         name.slice(2).toLowerCase(),
72         nextProps[name],
73         false
74       );
75     } else {
76       dom[name] = nextProps[name];
77     }
78   });
79 }
```

```
78     });
79 }
80 function commitRoot(){
81     commitWork(wipFiber.child);
82     currentRoot=wipFiber;
83     wipFiber=null;
84 }
85 function commitWork(fiber) {
86     if(!fiber){
87         return
88     }
89
90     //这次是找effectTag,之前这个diff工作已经实现
91     let domParenFiber=fiber.return
92     while(!domParenFiber.dom){
93         domParenFiber=domParenFiber.parent;
94     }
95     const domParent=domParenFiber.dom;
96     if(fiber.effectTag==='PLACEMENT'&&fiber.effectTag!==null){
97         domParent.appendChild(fiber.dom)
98
99     }else if(fiber.effectTag==='UPDATE'&&fiber.effectTag!==null){
100        updateDom(fiber.dom,fiber.base.props,fiber.props)
101    }
102
103    //提交修改
104    commitWork(fiber.child)
105    commitWork(fiber.sibling)
106 }
107
108 //开始任务=null
109 //调度
110 //fiber任务变成一个链表，我们记住当前要做的任务即可
111 //通过当前任务，指向下一个
112 //woking in progress;
113 let wipRoot = null; //当前工作fiber的根节点
114 let nextUnitwork = null;
115 let currentRoot = null;
116 function workLoop(deadline) {
117     while (nextUnitwork && deadline.timeRemaining() > 1) {
118         //获取下一个任务
119         nextUnitwork = performUnitOfWork(nextUnitwork);
120     }
121
122     if (!nextUnitwork && wipRoot) {
123         //执行我们的修改，提交修改，比如dom修改的diff结果
124         comitRoot();
125         return;
126     }
127
128     window.requestIdleCallback(workLoop);
129 }
130 window.requestIdleCallback(workLoop);
131 //下一个单元任务
132 //把fiber记录下来,performUnitOfWork就可以直接启动
133 function performUnitOfWork(fiber) {
134     //根据当前任务，计算diff返回下一个任务(子兄父)
135     //更新vdom的逻辑
```

```
136
137     const isFuncComponent = fiber.type instanceof Function;
138     if (isFuncComponent) {
139         //更新函数组件
140         updateFuncComponent(fiber);
141     } else {
142         //更新dom标签 hostComponent
143         updateHostComponet(fiber);
144     }
145
146     if (fiber.child) {
147         return fiber.child;
148     }
149     // child->slibing->slibing
150     let newFiber = fiber;
151     while (newFiber) {
152         if (newFiber.sibling) {
153             return newFiber.sibling;
154         }
155         //如果没有兄弟元素,返回父元素重新查找
156         newFiber = newFiber.parent;
157     }
158 }
159 let wipFiber = null;
160 let hookIndex = null;
161 //更新函数型组件
162 function updateFuncComponent(fiber) {
163     // wipFiber=fiber;
164     wipFiber = fiber;
165     hookIndex = 0;
166     wipFiber.hooks = [];//存储hooks的地方
167     const children = [fiber.type(fiber.props)];
168     reconcileChildren(fiber, children);
169 }
170
171 function updateHostComponet(fiber) {
172     if (!fiber.dom) {
173         fiber.dom = createDom(fiber);
174     }
175
176     reconcileChildren(fiber, fiber.props.children);
177 }
178
179 // fiber{
180 //   child,return,sibling
181 //   efectTag:副作用的类型UPDATE ,REMOVE ,PLACEMENT
182 // }
183
184 function reconcileChildren(wipFiber,elements) {
185     //子元素的遍历
186     //其实和vdom的diff核心逻辑是一致的
187     // 找出最小的修改, 操作最少的dom
188     //由于我们把props.children改成了链表, 我们现在简化一下, 直接挨个对比
189     let index=0;
190     let oldFiber=wipFiber.base&&wipFiber.base.child;
191     let prevsibling=null;
192     while(index<element.length||oldFiber!=null){
193         let element=element[index];
```

```
194     let newFiber=null;
195     //对比新老元素
196     const sameType=oldFiber&&element&&oldFiber.type==element.type
197     if(sameType){
198         newFiber={
199             type:oldFiber.type,
200             props:element.props,
201             dom:oldFiber.dom,
202             parent:wipFiber,
203             base:oldFiber,
204             effectTag:'UPDATE'
205         }
206     }
207
208     if(element&&!sameType){
209         newFiber={
210             type:element.type,
211             props:element.props,
212             dom:null,
213             parent:wipFiber,
214             base:null,
215             effectTag:'PLACEMENT'
216         }
217     }
218
219     if(oldFiber){
220         oldFiber.oldFiber.sibling;
221     }
222     if(index==0){
223         wipFiber.child=newFiber
224     }else{
225         prevSibling.sibling=newFiber
226     }
227
228     prevSibling=newFiber;//单纯为了缓存
229     index++;
230 }
231 }
232
233 function useState(init) {
234     //hooks就是一个数组，每一个数据记录这当前的数据
235     //如果修改了，更新wipRoot，启动workLoop，自动performUnitOfWork
236     const oldHook =
237         wipFiber.base && wipFiber.base.hooks &&
238         wipFiber.base.hooks[hookIndex];
239     const hook = {
240         state: oldHook ? oldHook.state : init,
241         queue: []
242     };
243
244     const actions = oldHook ? oldHook.queue : [];
245     actions.forEach(action => {
246         hook.state = action;
247     });
248
249     //所谓的异步修改
250     const setState = action => {
251         hook.queue.push(action);
252     };
253 }
```

```
251 //通过nextUnitwork启动任务
252 wipRoot = {
253   dom: currentRoot.dom,
254   props: currentRoot.props,
255   base: currentRoot
256 };
257 nextUnitwork = wipRoot; //相当于启动了任务
258 };
259
260 wipFiber.hooks.push(hook);
261 hookIndex++;
262 return [hook.state, setState];
263 }
264
265 class Component{
266   constructor(props){
267     this.props=props;
268   }
269 }
270
271 function useComponent(Component){
272   return function (props){
273     const compont=new Component(props);
274     let [state,setState]=useState(compont.state);
275     compont.props=props;
276     compont.setState=setState;
277     compont.state=state;
278     return compont.render();
279   }
280 }
281
282
283 export default {
284   Component,
285   createElement,
286   render,
287   useState
288 };
289
290 // fiber构建
291 // jsx=>createElement=>vdom
292 // 子元素=>props.children
293
```