# 父子组件传值

1. 父组件<Child content={value}/>
2. 子组件可以用this.props.content获取父组件传过来的值。方法也是同样的方法传递。但是父组件需要将传递的方法的作用域绑定到父组件。

3）子组件如果想要和父组件沟通，需要子组件调用父组件传递过来的方法。

# 2、react的特性



1. 可与其他框架并存，指的是react组件只挂载到一个id上，其他的dom元素可以采用其他的组件（vue, jquery）
2. 单向数据流：指的是数据指向一个方向：父组件可以随时修改子组件的数据，子组件要修改父组件的数据必须调用父组件的方法进行修改。
3. 函数式编程：每个组件由多个函数组成。

# 3、渲染render（）

1. 当组件初次创建的时候render函数会执行一次
2. 当state数据发生变化的时候，render函数会被重新执行
3. 当props数据发生变更的时候，render函数会被重新执行

# 4、ref的使用

1. 在jsx中对应的html标签上增加ref属性其为一个函数，函数的参数就是ref挂载的标签对应的真实的dom对象

<button ref={(button) => {}}>

ref对应的函数会在组件初次渲染的时候执行。所以可以在这个函数中将button对象赋值给组件变量。

1. 如果将ref属性写在自定义的组件上，那个函数的参数就是组件对象的实例。

# 5、this.setState()

这个函数式异步执行的。

this.setState(obj, fn)

this.setState(fn,fn)

# 6、生命周期函数

如果在某个组件中绑定了window事件，那么在离开（移除）此组件的时候也需要把window事件移除，否则会影响其他的组件。

# 7、react的ajax请求的地址：

demo.json

header.json

list.json

ajax请求需要放在componentDidMount中

Ant Design组件库：

<https://ant.design/index-cn>

npm install antd --save

# 8、前端路由

npm install react-router-dom --save

1）index.js中

import {BrowserRouter} from "react-router-dom"

...

ReactDOM.render(

<BrowserRouter>

<App />

</BrowserRouter>,

document.getElementById('root')

);

2）App或者其他任何组件中

import {Routes, Route, Link} from "react-router-dom"

<Routes>

<Route path="/button" element={<ButtonRouter/>}/>

<Route path="/list" element = {<ListRouter/>}/>

</Routes>

1. link用法

import {Link} from "react-router-dom"

render() {

return (

<Link to="/list">

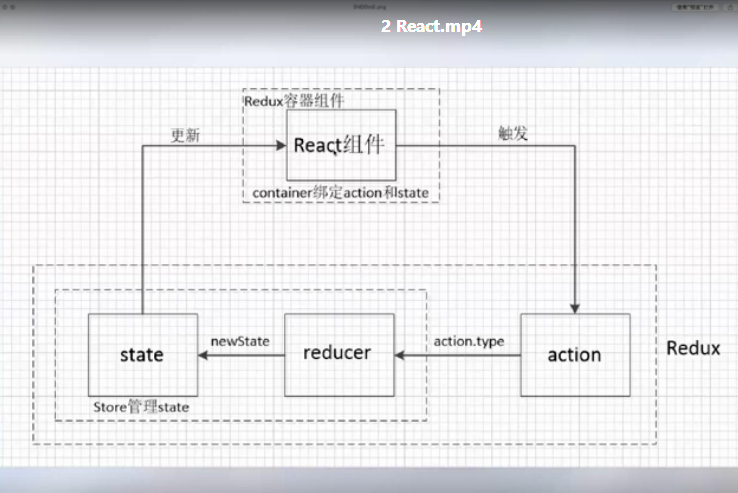
<Button size="large" type="primary">按钮</Button>

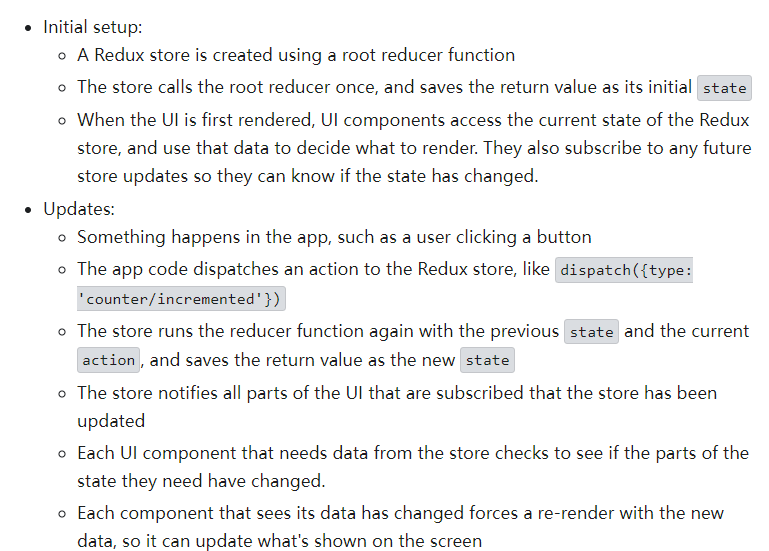
</Link>

)

}

# 9、redux和react-redux学习





应用中所有的 state 都以一个对象树的形式储存在一个单一的 store 中。 惟一改变 state 的办法是触发 action，一个描述发生什么的对象。 为了描述 action 如何改变 state 树，你需要编写 reducers。

1. 安装： npm install redux react-redux -S

react-redux是连接react视图和redux的中转

2）建立三个文件夹,并在每个文件夹中创建对应的js文件

a、contains

b、actions

**Action** 是把数据从应用（译者注：这里之所以不叫 view 是因为这些数据有可能是服务器响应，用户输入或其它非 view 的数据 ）传到 store 的有效载荷。它是 store 数据的**唯一**来源。

Action 本质上是 JavaScript 普通对象。我们约定，action 内必须使用一个字符串类型的 type 字段来表示将要执行的动作。除了 type 字段外，action 对象的结构完全由你自己决定。**我们应该尽量减少在 action 中传递的数据**。

**Action 创建函数** 就是生成 action 的方法。“action” 和 “action 创建函数” 这两个概念很容易混在一起，使用时最好注意区分。

在 Redux 中的 action 创建函数只是简单的返回一个 action:

function addTodo(text) {

return {

type: ADD\_TODO,

text

}

}

c、reducers/indexjs

**Reducers** 指定了应用状态的变化如何响应 [actions](https://www.redux.org.cn/docs/basics/Actions.html) 并发送到 store 的，记住 actions 只是描述了有事情发生了这一事实，并没有描述应用如何更新 state。

/\*\*

\* 这是一个 reducer，形式为 (state, action) => state 的纯函数。

\* 描述了 action 如何把 state 转变成下一个 state。

\*

\* state 的形式取决于你，可以是基本类型、数组、对象、

\* 甚至是 Immutable.js 生成的数据结构。惟一的要点是

\* 当 state 变化时需要返回全新的对象，而不是修改传入的参数。

\*

\* 下面例子使用 `switch` 语句和字符串来做判断，但你可以写帮助类(helper)

\* 根据不同的约定（如方法映射）来判断，只要适用你的项目即可。

\*/

function counter(state = 0, action) {

switch (action.type) {

case 'INCREMENT':

return state + 1;

case 'DECREMENT':

return state - 1;

default:

return state;

}

}

聚合reducer

import {combineReducers} from "redux"

import user from "./user.js"

export default combineReducers({

user

})

d、store

创建Redux store来存放应用的状态

import userReducer from "../reducer";

import {createStore} from "redux";

export default function userStore(initState) {

return createStore(userReducer, initState)

}

3）异步redux

npm install redux-thunk

a.在store中

// applyMiddleware用来注册中间件

//compose用来聚合中间件和扩展程序，

import {createStore, applyMiddleware,compose} from “redux”

// thunkMiddleware中间件用来拦截actions

import thunkMiddleware from “redux-thunk”

import {createStore} from "redux";

export default function userStore(initState) {

return createStore(userReducer, initState,)

}

b.在actions中

const loginRequest = () => {

return new Promise((resolve) => {

window.setTimeout(() => { //模拟登录完以后执行的动作（ajax请求）

resolve(update({ //update是一个同步的action

isLogin: true,

nickName: “”

}))

}, 3000)

})

}

export const login = (payload) => {

return async (dispatch) => {

const res = await *loginRequest*();

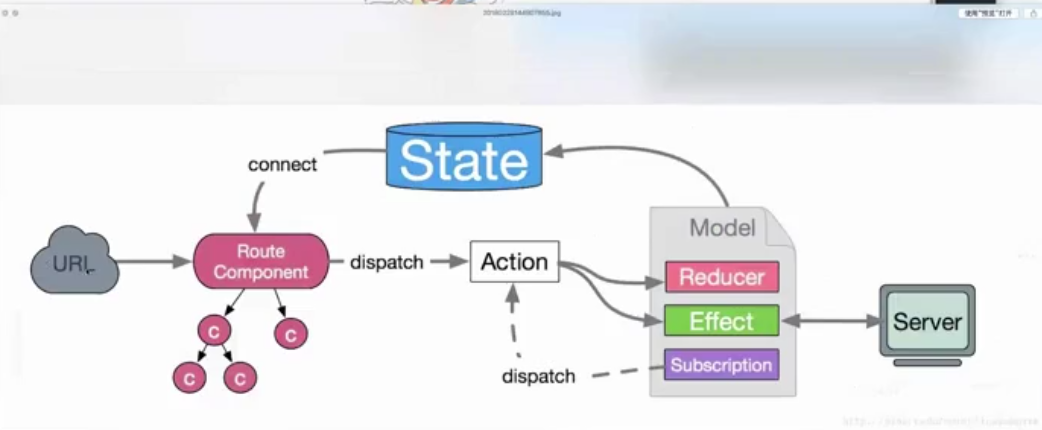
dispatch(res)

}

}

# 10、dav学习

## 1）数据的处理过程



1. hook
2. useEffect:副作用（数据获取，设置订阅以及手动更改 React 组件中的 DOM ）函数：有时候，我们只想**在 React 更新 DOM 之后运行一些额外的代码。**比如发送网络请求，手动变更 DOM，记录日志，这些都是常见的无需清除的操作。

a、执行时机：useEffect 会在浏览器绘制后延迟执行，但会保证在任何新的渲染前执行。在开始新的更新前，React 总会先清除上一轮渲染的 effect。

b、默认情况下useEffect会在每次渲染之后/和每次更新之后都会执行。如何控制？useEffect(()=> {}, []),useEffect(()=> {}, [count])

c、每次我们重新渲染，都会生成新的 effect，替换掉之前的。

d、useEffect的effect函数中返回一个清除函数，用于清除订阅，每次重新渲染更新的时候都会调用。

useEffect(() => {

function handleStatusChange(status) {

setIsOnline(status.isOnline);

}

ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

// Specify how to clean up after this effect:

return function cleanup() { //

ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);

};

});

e、

就像[你可以使用多个 state 的 Hook](https://react.docschina.org/docs/hooks-state.html" \l "tip-using-multiple-state-variables) 一样，你也可以使用多个 effect。这会将不相关逻辑分离到不同的 effect 中：**Hook 允许我们按照代码的用途分离他们，** 而不是像生命周期函数那样。React 将按照 effect 声明的顺序依次调用组件中的每一个 effect。

## 2）hook使用规则

a. Only Call Hooks at the Top Level

b. Only Call Hooks from React Functions

## 3）自定义HOOK逻辑

与 React 组件不同的是，自定义 Hook 不需要具有特殊的标识。我们可以自由的决定它的参数是什么，以及它应该返回什么（如果需要的话）。换句话说，它就像一个正常的函数。但是它的名字应该始终以 use 开头，这样可以一眼看出其符合 [Hook 的规则](https://zh-hans.reactjs.org/docs/hooks-rules.html)。

## 4）useContext(MyContext)

只是让你能够读取 context 的值以及订阅 context 的变化。你仍然需要在上层组件树中使用 <MyContext.Provider> 来为下层组件提供 context。在子组件（通过调用祖先组件中修改值的方法）和祖先组件中对context的修改会导致从祖先组件到子组件的重新渲染。

联想：用useReducer和useContext实现redux的功能：

在最外围的组件中myContext = React.createContext();

const [state, dispatch] = useReducer(reducer, initState)

<myContext.provider value={{state, dispatch}}>

<ChildComponent1/>

<ChildComponent2/>

</myContext.provider>

其value值等于state 和 dispatch

## 5）useReducer：可以替代useState

const initialState = {count: 0};

function reducer(state, action) {

switch (action.type) {

case 'increment':

return {count: state.count + 1};

case 'decrement':

return {count: state.count - 1};

default:

throw new Error();

}

}

function Counter() {

const [state, dispatch] = useReducer(reducer, initialState); //这里的initState会覆盖reducer中设置的默认值

return (

<>

Count: {state.count}

<button onClick={() => dispatch({type: 'decrement'})}>-</button>

<button onClick={() => dispatch({type: 'increment'})}>+</button>

</>

);

}

## 6）useRef:

函数组件中的实例属性，和类中的实例属性几乎完全一致

只要实例不销毁，实例属性保持不变。b、可以作为dom元素的引用

例如:const inputRef = useRef()

//上面的可以写成const inputRef = React.createRef()

//但是是有区别的用createRef()生成的属性由于不是实例属性，所以每次重新渲染的时候就会重新生产一个新的变量。

const inputFocus = () => {

inputRef.current.focus()

}

<input ref={inputRef}/>

<button onClick={inputFoucs}>点击获取焦点</button>

## 7）useMemo

:保存了渲染的结果

import React, {useMemo, useState} from "react";

let ch;

function Child1({a}) {

return (

<div>

{console.log("Child1重新渲染")}

{a}

</div>

)

}

function UseMemoApp() {

const [a, setA] = useState(0);

const [b, setB] = useState(0);

const child1 = useMemo(()=> (<div>

{console.log("useMemo,这是一个复杂的计算child1")}

<Child1 a= {b}/>

</div>), [a]) //依赖于a的变化当a发生变化的时候才会重新渲染

console.log("ch和child1相等", ch === child1) //第一次渲染和只改变a的渲染不会相等

ch = child1;

const child2 = (<div>

{console.log("没有使用useMemo，child2")}

<Child1 a ={b}/>

</div>) //只要组件中的值改变就会重新渲染，也就是改变a,b的值都会重新渲染

return (

<div>

{child1}

{child2}

<p><button onClick={() => (setA(a + 1))}>改变a</button></p>

<p><button onClick={() => (setB(b + 1))}>改变b</button></p>

</div>

)

}

export default UseMemoApp;

## 8）useCallback:

避免函数重新生成：因为hook用在函数组件中，每次渲染的时候，函数组件中的定义的函数就会重新生成。

# 11、mobx学习

1）a、要让项目支持装饰器写法b、安装mobx c、安装mobx-react

2）创建一个store文件

import {observable} from ‘mobx’

class AppStore {

@observable input = “” //让input可被跟踪

}

export default new AppStore();

3）如何在组件中使用mobx仓库的数据

在最外层包裹一个Provider组件，这个Provider有mobx-react提供，还需要给这个组件提供一个store数据，属性值就是store的实例

1. 使用仓库的组件，需要通过inject去注入Provider上的store

@inject(“store”)

然后在组件中就可以通过this.props...使用store中的数据

可以使用this.props.state....直接赋值(但是不推荐直接赋值)

1. 第四步只是拿到了数据，但是如果我修改数据，并不能体现在UI上。

需要监听仓库数据的变化，需要使用mobx-react中的observer来对组件类做一个装饰

@inject(“store”)

@observer

class App extends Component{}

1. 如果需要在组件中修改store中的值，那么需要在store中

import {observable,action} from ‘mobx’

class AppStore {

@observable input = “” //让input可被跟踪

@action setInputValue(value){

this.input = value

}

}

在组件中this.props.store.setInputValue(“newValue”)

1. 如果需要获取计算值：比如store中有个数组，我们需要在组件中获取数组的长度

则：需要在store中

import {observable,action, computed} from ‘mobx’

class AppStore {

@observable arr= [] //让arr被跟踪

@action arrAdd(value){

this.arr.push(value)

}

@computed get arrElementNum() {

return this.arr.length;

}

}

//在组件中

<div>{this.props.state.arrElementNum}</div>

8）性能优化Observer autorun