

React全家桶

一. React介绍

- 1.React起源与发展
- 2.React与传统MVC的关系
- 3.React的特性
- 4.虚拟DOM

二. create-react-app

三、编写第一个react应用程序

四. JSX语法与组件

- 1. JSX语法
- 2.Class组件
- 3. 函数式组件
- 4. 组件的样式
- 5. 事件处理
 - 5.1、绑定事件
 - 5.2、事件handler的写法
 - 5.3、Event 对象
- 6.Ref的应用
 - 6.1给标签设置ref="username"
 - 6.2 给组件设置ref="username"
 - 6.3 新的写法

五、组件的数据挂载方式

- 1、状态(state)
 - (1) 定义state
 - (2) setState
- 2、属性(props)
- 3、属性vs状态
- 4、渲染数据

六、表单中的受控组件与非受控组件

- 1、非受控组件
 - 默认值
- 2、受控组件

七. 组件通信的方式

- 1. 父子组件通信方式
- 2. 非父子组件通信方式
 - (1) 状态提升(中间人模式)
 - (2) 发布订阅模式实现
 - (3) context状态树传参

八. React生命周期

- 1. 初始化阶段
- 2. 运行中阶段
- 3. 销毁阶段

老生命周期的问题

新生命周期的替代

react中性能优化的方案

- 1. shouldComponentUpdate
- 2. PureComponent

九. React Hooks

使用hooks理由

useState(保存组件状态)

useEffect(处理副作用)和useLayoutEffect (同步执行副作用)

useEffect和useLayoutEffect有什么区别?

useCallback(记忆函数)

useMemo 记忆组件

useRef(保存引用值)

useReducer和useContext(减少组件层级)

自定义hooks

十. React 路由

1. 什么是路由?

2. 路由安装

3. 路由使用

(1) 路由方法导入

(2) 定义路由以及重定向

(3) 嵌套路由

(4) 路由跳转方式

(5) 路由传参

(6) 路由拦截

(7) withRouter的应用与原理

4.项目注意

(1) 反向代理

(2) css module

十一. Flux与Redux

1. redux介绍及设计和使用的三大原则

2. redux工作流

3. 与react绑定后使用redux实现案例

4. redux原理解析

5. reducer 扩展

6. redux中间件

7. Redux DevTools Extension

十二. react-redux

1. 介绍

2. 容器组件与UI组件

3. Provider与connect

4. HOC与context通信在react-redux底层中的应用

5. 高阶组件构建与应用

6. Redux 持久化

十三. UI组件库

1. ant-design (PC端)

2. antd-mobile (移动端)

十四. Immutable

1.Immutable.js介绍

2. 深拷贝与浅拷贝的关系

3. Immutable优化性能的方式

4. Immutable中常用类型 (Map, List)

5. Immutable+Redux的开发方式

6. 缺点

十五. Mobx

1. Mobx介绍

2. Mobx与redux的区别

3. Mobx的使用

4. mobx-react的使用

5. 支持装饰器

十六. TS

1-typescript

2-安装

3-声明

4-变量声明

5-定义普通函数

 接口描述形状

 传参

 类型断言as

6-定义普通类

7-定义类组件

- 8-定义函数式组件
 - useRef
 - useContext
 - useReducer
- 9-父子通信
- 10-路由
 - 编程式导航
 - 动态路由
- 11-redux
- 十七. styled-components
 - 基本
 - 透传props
 - 基于props做样式判断
 - 样式化任意组件(一定要写className)
 - 扩展样式
 - 加动画
- 十八.单元测试
 - 挂载组件
 - 测试组件渲染出来的 HTML
 - 模拟用户交互
- 十九. redux-saga
 - 代码实现
- 二十. React补充
 - 1. Portal
 - 1、用法
 - 2、在protoal中的事件冒泡
 - 2.Lazy 和 Suspense
 - 1、React.lazy 定义
 - 2、如何使用React.lazy
 - 3. forwardRef
 - 未使用forwardRef
 - 使用forwardRef
 - 4. Functional Component缓存
 - 为啥起memo这个名字?
 - 作用
 - 与PureComponent 区别
 - 用法
- 二十一. React扩展
 - 1. GraphQL
 - (1) 介绍与hello
 - (2) 参数类型与传递
 - (3) mutation
 - (4) 结合数据库
 - (5) 客户端访问
 - (6) 结合React
 - 2. dva
 - dva 应用的最简结构
 - 数据流图
 - 3. umi
 - 安装脚手架
 - 目录
 - 路由
 - mock功能
 - 反向代理
 - antd
 - dva集成

React全家桶

作者: kerwin

版本: QF1.0

版权: 千锋HTML5大前端教研院

公众号: 大前端私房菜

一. React介绍

1.React起源与发展

React 起源于 Facebook 的内部项目，因为该公司对市场上所有 JavaScript MVC 框架，都不满意，就决定自己写一套，用来架设Instagram 的网站。做出来以后，发现这套东西很好用，就在2013年5月开源了。

2.React与传统MVC的关系

轻量级的视图层库! *A JavaScript library for building user interfaces*

React不是一个完整的MVC框架，最多可以认为是MVC中的V (View)，甚至React并不非常认可MVC开发模式；React 构建页面 UI 的库。可以简单地理解为，React 将界面分成了各个独立的小块，每一个块就是组件，这些组件之间可以组合、嵌套，就成了我们的页面。

3.React的特性

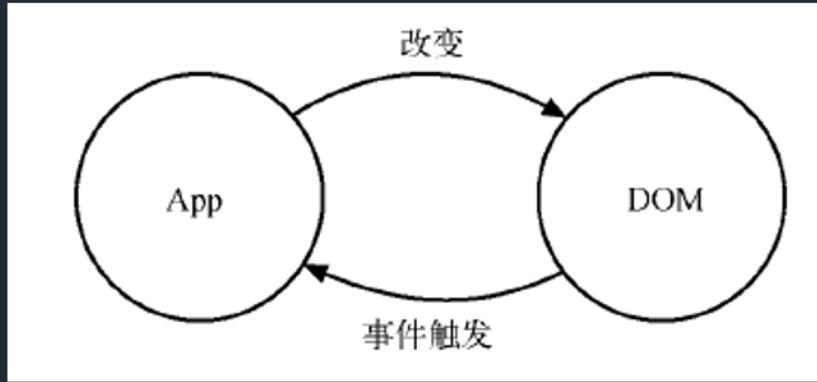
特点:

- 1.声明式设计 -React采用声明范式，可以轻松描述应用。
- 2.高效 -React通过对DOM的模拟(虚拟dom)，最大限度地减少与DOM的交互。
- 3.灵活 -React可以与已知的库或框架很好地配合。
- 4.JSX - JSX 是 JavaScript 语法的扩展。
- 5.组件 - 通过 React 构建组件，使得代码更加容易得到复用，能够很好的应用在大项目的开发中。
- 6.单向响应的数据流 - React 实现了单向响应的数据流，从而减少了重复代码，这也是它为什么比传统数据绑定更简单。

4.虚拟DOM

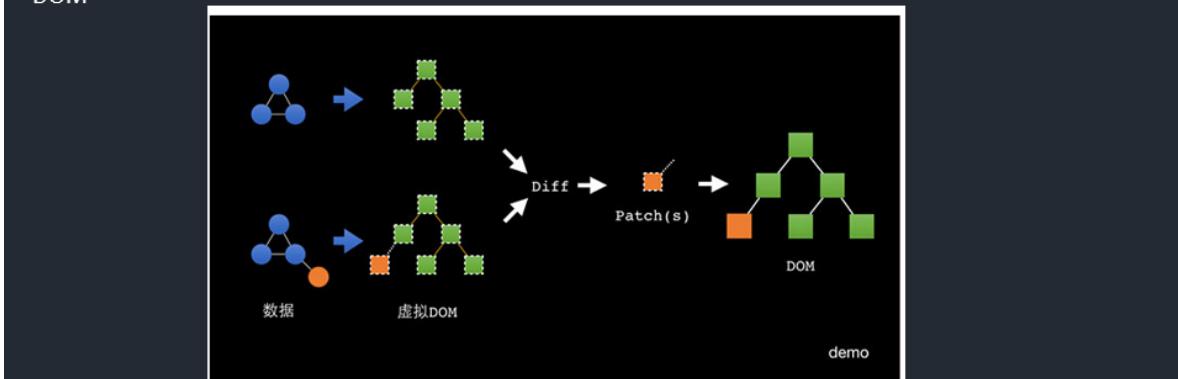
传统dom更新:

真实页面对应一个 DOM 树。在传统页面的开发模式中，每次需要更新页面时，都要手动操作 DOM 来进行更新



虚拟dom:

DOM 操作非常昂贵。我们都知道在前端开发中，性能消耗最大的就是 DOM 操作，而且这部分代码会让整体项目的代码变得难以维护。React 把真实 DOM 树转换成 JavaScript 对象树，也就是 Virtual DOM



二. create-react-app

全局安装create-react-app

```
$ npm install -g create-react-app
```

创建一个项目

```
$ create-react-app your-app 注意命名方式  
Creating a new React app in /dir/your-app.  
Installing packages. This might take a couple of minutes. 安装过程较慢,  
Installing react, react-dom, and react-scripts...
```

如果不想全局安装，可以直接使用npx

```
$ npx create-react-app myapp 也可以实现相同的效果
```

这需要等待一段时间，这个过程实际上会安装三个东西

- react: react的顶级库
- react-dom: 因为react有很多的运行环境，比如app端的react-native, 我们要在web上运行就使用react-dom
- react-scripts: 包含运行和打包react应用程序的所有脚本及配置

出现下面的界面，表示创建项目成功:

```
Success! Created your-app at /dir/your-app  
Inside that directory, you can run several commands:  
  
npm start  
  Starts the development server.  
  
npm run build  
  Bundles the app into static files for production.
```

```
npm test  
Starts the test runner.  
  
npm run eject  
Removes this tool and copies build dependencies, configuration files  
and scripts into the app directory. If you do this, you can't go back!
```

We suggest that you begin by typing:

```
cd your-app  
npm start
```

Happy hacking!

根据上面的提示，通过 `cd your-app` 命令进入目录并运行 `npm start` 即可运行项目。

生成项目的目录结构如下：

── README.md	使用方法的文档
── node_modules	所有的依赖安装的目录
── package-lock.json	锁定安装时的包的版本号，保证团队的依赖能保证一致。
── package.json	
── public	静态公共目录
── src	开发用的源代码目录

常见问题：

- npm安装失败
 - 切换为npm镜像为淘宝镜像
 - 使用yarn，如果本来使用yarn还要失败，还得把yarn的源切换到国内
 - 如果还没有办法解决，请删除node_modules及package-lock.json然后重新执行 `npm install` 命令
 - 再不能解决就删除node_modules及package-lock.json的同时清除npm缓存 `npm cache clean --force` 之后再执行 `npm install` 命令

三、编写第一个react应用程序

react开发需要引入多个依赖文件：react.js、react-dom.js，分别又有开发版本和生产版本，create-react-app里已经帮我们把这些东西都安装好了。把通过CRA创建的工程目录下的src目录清空，然后在里面重新创建一个index.js。写入以下代码：

```
// 从 react 的包当中引入了 React。只要你要写 React.js 组件就必须引入React，因为react里有一种语法叫JSX，稍后会讲到JSX，要写JSX，就必须引入React
import React from 'react'
// ReactDOM 可以帮助我们把 React 组件渲染到页面上去，没有其它的作用了。它是从 react-dom 中引入的，而不是从 react 引入。
import ReactDOM from 'react-dom'

// ReactDOM里有一个render方法，功能就是把组件渲染并且构造 DOM 树，然后插入到页面上某个特定的元素上
ReactDOM.render(
// 这里就比较奇怪了，它并不是一个字符串，看起来像是纯 HTML 代码写在 JavaScript 代码里面。语法错误吗？这并不是合法的 JavaScript 代码，“在 JavaScript 写的标签的”语法叫 JSX-JavaScript XML。
<h1>欢迎进入React的世界</h1>,
// 渲染到哪里
document.getElementById('root')
)
```

注意：

<React.StrictMode> 目前有助于：

识别不安全的生命周期

关于使用过时字符串 ref API 的警告

检测意外的副作用

检测过时的 context API

四. JSX语法与组件

1. JSX语法

JSX 将 HTML 语法直接加入到 JavaScript 代码中，再通过翻译器转换到纯 JavaScript 后由浏览器执行。在实际开发中，JSX 在产品打包阶段都已经编译成纯 JavaScript，不会带来任何副作用，反而会让代码更加直观并易于维护。编译过程由Babel 的 JSX 编译器实现。

<https://reactjs.org/docs/hello-world.html>

原理是什么呢？

要明白JSX的原理，需要先明白如何用 JavaScript 对象来表现一个 DOM 元素的结构？

看下面的DOM结构

```
<div class='app' id='appRoot'>
  <h1 class='title'>欢迎进入React的世界</h1>
  <p>
    React.js 是一个帮助你构建页面 UI 的库
  </p>
</div>
```

上面这个 HTML 所有的信息我们都可以用 JavaScript 对象来表示：

```
{  
  tag: 'div',  
  attrs: { className: 'app', id: 'appRoot' },  
  children: [  
    {  
      tag: 'h1',  
      attrs: { className: 'title' },  
      children: ['欢迎进入React的世界']  
    },  
    {  
      tag: 'p',  
      attrs: null,  
      children: ['React.js 是一个构建页面 UI 的库']  
    }  
  ]  
}
```

但是用 JavaScript 写起来太长了，结构看起来又不清晰，用 HTML 的方式写起来就方便很多了。

于是 React.js 就把 JavaScript 的语法扩展了一下，让 JavaScript 语言能够支持这种直接在 JavaScript 代码里面编写类似 HTML 标签结构的语法，这样写起来就方便很多了。编译的过程会把类似 HTML 的 JSX 结构转换成 JavaScript 的对象结构。

下面代码：

```
import React from 'react'  
import ReactDOM from 'react-dom'  
  
class App extends React.Component {  
  render () {  
    return (  
      <div className='app' id='appRoot'>  
        <h1 className='title'>欢迎进入React的世界</h1>  
        <p>  
          React.js 是一个构建页面 UI 的库  
        </p>  
      </div>  
    )  
  }  
}  
  
ReactDOM.render(  
  <App />,  
  document.getElementById('root')  
)
```

编译之后将得到这样的代码：

```
import React from 'react'  
import ReactDOM from 'react-dom'  
  
class App extends React.Component {  
  render () {  
    return (  
      React.createElement(  
        "div",  
        {
```

```

        className: 'app',
        id: 'appRoot'
    },
    React.createElement(
        "h1",
        { className: 'title' },
        "欢迎进入React的世界"
    ),
    React.createElement(
        "p",
        null,
        "React.js 是一个构建页面 UI 的库"
    )
)
)
}
}

ReactDOM.render(
    React.createElement(App),
    document.getElementById('root')
)

```

`React.createElement` 会构建一个 JavaScript 对象来描述你 HTML 结构的信息，包括标签名、属性、还有子元素等，语法为

```

React.createElement(
    type,
    [props],
    [...children]
)

```

所谓的 JSX 其实就是 JavaScript 对象，所以使用 React 和 JSX 的时候一定要经过编译的过程：

JSX — 使用 react 构造组件， bable 进行编译 —> JavaScript 对象 — `ReactDOM.render()` —> DOM 元素 —> 插入页面

2. Class 组件

ES6 的加入让 JavaScript 直接支持使用 class 来定义一个类， react 创建组件的方式就是使用的类的继承，`ES6 class` 是目前官方推荐的使用方式，它使用了 ES6 标准语法来构建，看以下代码：

```

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

class App extends React.Component {
    render () {
        return (
            <h1>欢迎进入React的世界</h1>
        )
    }
}
ReactDOM.render(
    <App />,
    document.getElementById('root')
)

```

- es6 class 组件其实就是一个构造器,每次使用组件都相当于在实例化组件, 像这样:

```

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

class App extends React.Component {
  render () {
    return (
      <h1>欢迎进入{this.props.name}的世界</h1>
    )
  }
}

const app = new App({
  name: 'react'
}).render()

ReactDOM.render(
  app,
  document.getElementById('root')
)

```

3. 函数式组件

```

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

const App = (props) => <h1>欢迎进入React的世界</h1>

ReactDOM.render(
  // React组件的调用方式
  <App />,
  document.getElementById('root')
)

```

这样一个完整的函数式组件就定义好了。但要注意! 注意! 注意! 组件名必须**大写**, 否则报错。

4. 组件的样式

- 行内样式

想给虚拟dom添加行内样式, 需要使用表达式传入样式对象的方式来实现:

```
// 注意这里的两个括号, 第一个表示我们在要JSX里插入JS了, 第二个是对象的括号
<p style={{color:'red', fontSize:'14px'}}>Hello world</p>
```

行内样式需要写入一个样式对象, 而这个样式对象的位置可以放在很多地方, 例如 `render` 函数里、组件原型上、外链js文件中

- 使用 `class`

React推荐我们使用行内样式, 因为React觉得每一个组件都是一个独立的整体

其实我们大多数情况下还是大量的在为元素添加类名，但是需要注意的是，`class`需要写成`className`（因为毕竟是在写类js代码，会收到js规则的现在，而`class`是关键字）

```
<p className="hello">Hello world</p>
```

注意：

`class` ==> `className` , `for` ==> `htmlFor(label)`

5. 事件处理

5.1、绑定事件

采用`on+事件名`的方式来绑定一个事件，注意，这里和原生的事件是有区别的，原生的事件全是小写`onClick`, React里的事件是驼峰`onClick`，**React的事件并不是原生事件，而是合成事件。**

5.2、事件handler的写法

- 直接在`render`里写行内的箭头函数(不推荐)
- 在组件内使用箭头函数定义一个方法(推荐)
- 直接在组件内定义一个非箭头函数的方法，然后在`render`里直接使用`onClick={this.handleClick.bind(this)}`(不推荐)
- 直接在组件内定义一个非箭头函数的方法，然后在`constructor`里`bind(this)`(推荐)

5.3、Event 对象

和普通浏览器一样，事件`handler`会被自动传入一个`event`对象，这个对象和普通的浏览器`event`对象所包含的方法和属性都基本一致。不同的是 React中的`event`对象并不是浏览器提供的，而是它自己内部所构建的。它同样具有`event.stopPropagation`、`event.preventDefault`这种常用的方法

6.Ref的应用

6.1给标签设置`ref="username"`

通过这个获取`this.refs.username`，`ref`可以获取到应用的真实dom

6.2 给组件设置`ref="username"`

通过这个获取`this.refs.username`，`ref`可以获取到组件对象

6.3 新的写法

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

访问`this.myRef.current`

五、组件的数据挂载方式

1、状态(state)

状态就是组件描述某种显示情况的数据，由组件自己设置和更改，也就是说由组件自己维护，使用状态的目的就是为了在不同的状态下使组件的显示不同(自己管理)

(1) 定义state

第一种方式

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

class App extends Component {
  state = {
    name: 'React',
    isLiked: false
  }
  render () {
    return (
      <div>
        <h1>欢迎来到{this.state.name}的世界</h1>
        <button>
          {
            this.state.isLiked ? '♥取消' : '❤收藏'
          }
        </button>
      </div>
    )
  }
}
ReactDOM.render(
  <App/>,
  document.getElementById('root')
)
```

另一种方式

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

class App extends Component {
  constructor() {
    super()
    this.state = {
      name: 'React',
      isLiked: false
    }
  }
  render () {
    return (
      <div>
        <h1>欢迎来到{this.state.name}的世界</h1>
        <button>
          {
            this.state.isLiked ? '♥取消' : '❤收藏'
          }
        </button>
      </div>
    )
  }
}
```

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

`this.state`是纯js对象,在vue中, `data`属性是利用`Object.defineProperty`处理过的, 更改`data`的数据的时候会触发数据的`getter`和`setter`, 但是React中没有做这样的处理, 如果直接更改的话, react是无法得知的, 所以, 需要使用特殊的更改状态的方法`setState`。

(2) `setState`

`isLiked`存放在实例的`state`对象当中, 组件的`render`函数内, 会根据组件的`state`中的`isLiked`不同显示“取消”或“收藏”内容。下面给`button`加上了点击的事件监听。

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

class App extends Component {
    constructor() {
        super()
        this.state = {
            name: 'React',
            isLiked: false
        }
    }
    handleBtnClick = () => {
        this.setState({
            isLiked: !this.state.isLiked
        })
    }
    render () {
        return (
            <div>
                <h1>欢迎来到{this.state.name}的世界</h1>
                <button onClick={this.handleBtnClick}>
                    {
                        this.state.isLiked ? '❤取消' : '❤收藏'
                    }
                </button>
            </div>
        )
    }
}
ReactDOM.render(
    <App/>,
    document.getElementById('root')
)
```

`setState`有两个参数

第一个参数可以是对象, 也可以是方法return一个对象, 我们把这个参数叫做`updater`

- 参数是对象

```
this.setState({  
  isLiked: !this.state.isLiked  
})
```

- 参数是方法

```
this.setState((prevState, props) => {  
  return {  
    isLiked: !prevState.isLiked  
  }  
})
```

注意的是这个方法接收两个参数，第一个是上一次的state，第二个是props

`setState` 是异步的，所以想要获取到最新的state，没有办法获取，就有了第二个参数，这是一个可选的回调函数

```
this.setState((prevState, props) => {  
  return {  
    isLiked: !prevState.isLiked  
  }  
}, () => {  
  console.log('回调里的', this.state.isLiked)  
}  
console.log('setState外部的', this.state.isLiked)
```

2、属性(props)

`props` 是正常是外部传入的，组件内部也可以通过一些方式来初始化的设置，属性不能被组件自己更改，但是你可以通过父组件主动重新渲染的方式来传入新的 `props`

属性是描述性质、特点的，组件自己不能随意更改。

之前的组件代码里面有 `props` 的简单使用，总的来说，在使用一个组件的时候，可以把参数放在标签的属性当中，所有的属性都会作为组件 `props` 对象的键值。通过箭头函数创建的组件，需要通过函数的参数来接收 `props`：

(1) 在组件上通过`key=value` 写属性，通过`this.props`获取属性，这样组件的可复用性提高了。

(2) 注意在传参数时候，如果写成`isShow="true"` 那么这是一个字符串 如果写成`isShow={true}` 这个是布尔值

(3) `{...对象}` 展开赋值

(4) 默认属性值

```
*. defaultProps = {  
}  
static defaultProps = {  
  myname: "默认的myname",  
  myshow:true  
}
```

(5) `prop-types` 属性验证

```

import propTypes from "prop-types";
*.propTypes={
    name:propTypes.string,
    age:propTypes.number
}

static propTypes={
    myname:propTypes.string,
    myshow:propTypes.bool
}

```

3、属性vs状态

相似点：都是纯js对象，都会触发render更新，都具有确定性（状态/属性相同，结果相同）

不同点：

1. 属性能从父组件获取，状态不能
2. 属性可以由父组件修改，状态不能
3. 属性能在内部设置默认值，状态也可以，设置方式不一样
4. 属性不在组件内部修改，状态要在组件内部修改
5. 属性能设置子组件初始值，状态不可以
6. 属性可以修改子组件的值，状态不可以

`state` 的主要作用是用于组件保存、控制、修改自己的可变状态。`state` 在组件内部初始化，可以被组件自身修改，而外部不能访问也不能修改。你可以认为 `state` 是一个局部的、只能被组件自身控制的数据源。`state` 中状态可以通过 `this.setState` 方法进行更新，`setState` 会导致组件的重新渲染。

`props` 的主要作用是让使用该组件的父组件可以传入参数来配置该组件。它是外部传进来的配置参数，组件内部无法控制也无法修改。除非外部组件主动传入新的 `props`，否则组件的 `props` 永远保持不变。

没有 `state` 的组件叫无状态组件（stateless component），设置了 `state` 的叫做有状态组件（stateful component）。因为状态会带来管理的复杂性，我们尽量多地写无状态组件，尽量少地写有状态的组件。这样会降低代码维护的难度，也会在一定程度上增强组件的可复用性。

4、渲染数据

- 条件渲染

```

{
  condition ? '渲染列表的代码' : '空空如也'
}

```

- 列表渲染

```

// 数据
const people = [
  {
    id: 1,
    name: 'Leo',
    age: 35
  },
  {
    id: 2,
    name: 'XiaoMing',
    age: 16
  }
]

```

```

}]
// 渲染列表
{
  people.map(person => {
    return (
      <dl key={person.id}>
        <dt>{person.name}</dt>
        <dd>age: {person.age}</dd>
      </dl>
    )
  })
}

```

React的高效依赖于所谓的 Virtual-DOM，尽量不碰 DOM。对于列表元素来说会有一个问题：元素可能会在一个列表中改变位置。要实现这个操作，只需要交换一下 DOM 位置就行了，但是React并不知道其实我们只是改变了元素的位置，所以它会重新渲染后面两个元素（再执行 Virtual-DOM），这样会大大增加 DOM 操作。但如果给每个元素加上唯一的标识，React 就可以知道这两个元素只是交换了位置，这个标识就是 `key`，这个 `key` 必须是每个元素唯一的标识

- `dangerouslySetInnerHTML`

对于富文本创建的内容，后台拿到的数据是这样的：

```
content = "<p>React.js是一个构建UI的库</p>"
```

出于安全的原因，React当中所有表达式的内容会被转义，如果直接输入，标签会被当成文本。这时候就需要使用 `dangerouslySetInnerHTML` 属性，它允许我们动态设置 `innerHTML`

```

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

class App extends Component {
  constructor() {
    super()
    this.state = {
      content : "<p>React.js是一个构建UI的库</p>"
    }
  }
  render () {
    return (
      <div
        // 注意这里是两个下划线 __html
        dangerouslySetInnerHTML={{__html: this.state.content}}
      />
    )
  }
}
ReactDOM.render(
  <App/>,
  document.getElementById('root')
)

```

六、表单中的受控组件与非受控组件

1、非受控组件

React要编写一个非受控组件，可以 [使用 ref](#) 来从 DOM 节点中获取表单数据，就是非受控组件。

例如，下面的代码使用非受控组件接受一个表单的值：

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

因为非受控组件将真实数据储存在 DOM 节点中，所以在使用非受控组件时，有时候反而更容易同时集成 React 和非 React 代码。如果你不介意代码美观性，并且希望快速编写代码，使用非受控组件往往可以减少你的代码量。否则，你应该使用受控组件。

默认值

在 React 渲染生命周期时，表单元素上的 `value` 将会覆盖 DOM 节点中的值，在非受控组件中，你经常希望 React 能赋予组件一个初始值，但是不去控制后续的更新。在这种情况下，你可以指定一个 `defaultValue` 属性，而不是 `value`。

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input
          defaultValue="Bob"
          type="text"
          ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}
```

同样，`<input type="checkbox">` 和 `<input type="radio">` 支持 `defaultChecked`，`<select>` 和 `<textarea>` 支持 `defaultValue`。

2. 受控组件

在 HTML 中，表单元素（如

 和
<select>）通

```
class NameForm extends React.Component {  
  
  constructor(props) {  
  
    super(props);  
  
    this.state = {value: ''};  
  
    this.handleChange = this.handleChange.bind(this);  
  
    this.handleSubmit = this.handleSubmit.bind(this);  
  
  }  
  
  handleChange(event) {  
  
    this.setState({value: event.target.value});  
  
  }  
  
  handleSubmit(event) {  
  
    alert('提交的名字：' + this.state.value);  
  
    event.preventDefault();  
  
  }  
  
  render() {  
  
    return (  
      <form onSubmit={this.handleSubmit}>  
        <label>  
          名字：  
          <input type="text" value={this.state.value} onChange={this.handleChange}>  
        </label>  
        <input type="submit" value="提交" />  
      </form>  
    );  
  }  
}
```

```
    );
}

}
```

由于在表单元素上设置了 `value` 属性，因此显示的值将始终为 `this.state.value`，这使得 React 的 state 成为唯一数据源。由于 `handlechange` 在每次按键时都会执行并更新 React 的 state，因此显示的值将随着用户输入而更新。

对于受控组件来说，输入的值始终由 React 的 state 驱动。你也可以将 `value` 传递给其他 UI 元素，或者通过其他事件处理函数重置，但这意味着你需要编写更多的代码。

注意: 另一种说法（广义范围的说法），React 组件的数据渲染是否被调用者传递的 `props` 完全控制，控制则为受控组件，否则非受控组件。

七. 组件通信的方式

1. 父子组件通信方式

(1) 传递数据(父传子)与传递方法(子传父)

(2) ref 标记(父组件拿到子组件的引用，从而调用子组件的方法)

在父组件中清除子组件的 input 输入框的 value 值。`this.refs.form.reset()`

2. 非父子组件通信方式

(1) 状态提升(中间人模式)

React 中的状态提升概括来说，就是将多个组件需要共享的状态提升到它们最近的父组件上，在父组件上改变这个状态然后通过 `props` 分发给子组件。

(2) 发布订阅模式实现

(3) context 状态树传参

a. 先定义全局 context 对象

```
import React from 'react'
const GlobalContext = React.createContext()
export default GlobalContext
```

b. 根组件引入 `GlobalContext`，并使用 `GlobalContext.Provider`（生产者）

```
//重新包装根组件 class App {}
```

```
<GlobalContext.Provider
  value={{
    name: "kerwin",
    age: 100,
    content: this.state.content,
    show: this.show.bind(this),
    hide: this.hide.bind(this)
  }}>
```

```
>
<之前的根组件></之前的根组件>
</GlobalContext.Provider>
```

c. 任意组件引入`GlobalContext`并调用`context`, 使用`GlobalContext.Consumer` (消费者)

```
<GlobalContext.Consumer>
{
  context => {
    this.myshow = context.show; //可以在当前组件任意函数触发
    this.myhide = context.hide; //可以在当前组件任意函数触发
    return (
      <div>
        {context.name}-{context.age}-{context.content}
      </div>
    )
  }
}
</GlobalContext.Consumer>
```

注意: `GlobalContext.Consumer`内必须是回调函数, 通过`context`方法改变根组件状态

context优缺点:

优点: 跨组件访问数据

缺点: react组件树种某个上级组件`shouldComponetUpdate`返回false,当`context`更新时, 不会引起下级组件更新

八. React生命周期

1. 初始化阶段

- `componentWillMount`: `render`之前最后一次修改状态的机会
- `render`: 只能访问`this.props`和`this.state`, 不允许修改状态和DOM输出
- `componentDidMount`: 成功`render`并渲染完成真实DOM之后触发, 可以修改DOM

2. 运行中阶段

- `componentWillReceiveProps`: 父组件修改属性触发
- `shouldComponentUpdate`: 返回false会阻止`render`调用
- `componentWillUpdate`: 不能修改属性和状态
- `render`: 只能访问`this.props`和`this.state`, 不允许修改状态和DOM输出
- `componentDidUpdate`: 可以修改DOM

3. 销毁阶段

- `componentWillUnmount`: 在删除组件之前进行清理操作，比如计时器和事件监听器

老生命周期的问题

- (1) `componentWillMount`,在ssr中这个方法将会被多次调用，所以会重复触发多遍，同时在这里如果绑定事件，将无法解绑，导致内存泄漏，变得不够安全高效逐步废弃。
- (2) `componentWillReceiveProps` 外部组件多次频繁更新传入多次不同的 props，会导致不必要的异步请求
- (3) `componentWillUpdate`, 更新前记录 DOM 状态，可能会做一些处理，与`componentDidUpdate`相隔时间如果过长，会导致状态不太信

新生命周期的替代

- (1) `getDerivedStateFromProps` 第一次的初始化组件以及后续的更新过程中(包括自身状态更新以及父传子)，返回一个对象作为新的state，返回null则说明不需要在这里更新state

```
//老的生命周期的写法
componentDidMount() {
  if(this.props.value!==undefined){
    this.setState({
      current:this.props.value
    })
  }
}

componentWillReceiveProps(nextProps){
  if(nextProps.value !==undefined){
    this.setState({
      current:nextProps.value
    })
  }
}

// 新的生命周期写法
static getDerivedStateFromProps(nextProps) {
  if(nextProps.value!==undefined){
    return {
      current:nextProps.value
    }
  }
  return null
}
```

- (2) `getSnapshotBeforeUpdate` 取代了 `componentWillUpdate`,触发时间为update发生的时候，在**render之后dom渲染之前**返回一个值，作为`componentDidUpdate`的第三个参数。

```
//新的数据不断插入数据前面，导致我正在看的数据向下走，如何保持可视区依旧是我之前看的数据呢？
getSnapshotBeforeUpdate(){
  return this.refs.wrapper.scrollHeight
}

componentDidUpdate(prevProps, prevState, preHeight) {
```

```

    //if(preHeight === 200) return ;
    this.refs.wrapper.scrollTop +=this.refs.wrapper.scrollHeight-preHeight
}

<div style={{height:"200px",overflow:"auto"}} ref="wrapper">
  <ul>
    .....
  </ul>
</div>

```

react中性能优化的方案

1. shouldComponentUpdate

控制组件自身或者子组件是否需要更新，尤其在子组件非常多的情况下，需要进行优化。

2. PureComponent

PureComponent会帮你比较新props 跟旧的props，新的state和老的state（值相等,或者对象含有相同的属性、且属性值相等），决定shouldcomponentUpdate 返回true 或者 false，从而决定要不要呼叫 render function。

注意：

如果你的 state 或 props 『永远都会变』，那 PureComponent 并不会比较快，因为 shallowEqual 也需要花时间。

九. React Hooks

使用hooks理由

1. 高阶组件为了复用，导致代码层级复杂
2. 生命周期的复杂
3. 写成functional组件,无状态组件，因为需要状态，又改成了class,成本高

useState(保存组件状态)

```
const [state, setState] = useState(initialState)
```

useEffect(处理副作用)和useLayoutEffect (同步执行副作用)

Function Component 不存在生命周期，所以不要把 Class Component 的生命周期概念搬过来试图对号入座。

```

useEffect(() => {
  //effect
  return () => {
    //cleanup
  };
}, [依赖的状态;空数组,表示不依赖])

```

不要对 Dependencies 撒谎, 如果你明明使用了某个变量，却没有申明在依赖中，你等于向 React 撒了谎，后果就是，当依赖的变量改变时，useEffect 也不会再次执行, eslint会报警告

Preview页面改造成函数式组件，在路径上从id=1切换到id=2也会自动重新加载，比class组件方便

```
let id = props.match.params.myid
useEffect(()=>{
  axios.get(`/articles/${id}`).then(res => {
    setttitle(res.data.title)
    setcontent(res.data.content)
    setcategory(res.data.category)
  })
},[id])
```

useEffect和useLayoutEffect有什么区别？

简单来说就是调用时机不同，`useLayoutEffect` 和原来 `componentDidMount & componentDidUpdate` 一致，在 react 完成 DOM 更新后马上同步调用的代码，会阻塞页面渲染。而 `useEffect` 是会在整个页面渲染完才会调用的代码。

官方建议优先使用 `useEffect`

However, we recommend starting with `useEffect` first and only trying `useLayoutEffect` if that causes a problem.

在实际使用时如果想避免 **页面抖动**（在 `useEffect` 里修改 DOM 很有可能出现）的话，可以把需要操作 DOM 的代码放在 `useLayoutEffect` 里。在这里做点 dom 操作，这些 dom 修改会和 react 做出的更改一起被一次性渲染到屏幕上，只有一次回流、重绘的代价。

useCallback(记忆函数)

防止因为组件重新渲染，导致方法被重新创建，起到缓存作用 只有第二个参数变化了，才重新声明一次

```
var handleClick = useCallback(()=>{
  console.log(name)
},[name])
<button onclick={()=>handleClick()}>hello</button>

//只有name改变后，这个函数才会重新声明一次，
//如果传入空数组，那么就是第一次创建后就被缓存，如果name后期改变了，拿到的还是老的name。
//如果不传第二个参数，每次都会重新声明一次，拿到的就是最新的name.
```

useMemo 记忆组件

`useCallback` 的功能完全可以由 `useMemo` 所取代，如果你想通过使用 `useMemo` 返回一个记忆函数也是完全可以的。

```
useCallback(fn, inputs) is equivalent to useMemo(() => fn, inputs).
```

唯一的区别是：`useCallback` 不会执行第一个参数函数，而是将它返回给你，而 `useMemo` 会执行第一个函数并且将函数执行结果返回给你。所以在前面的例子中，可以返回 `handleClick` 来达到存储函数的目的。

所以 `useCallback` 常用记忆事件函数，生成记忆后的事件函数并传递给子组件使用。而 `useMemo` 更适合经过函数计算得到一个确定的值，比如记忆组件。

useRef(保存引用值)

```
const myswiper = useRef(null);
<Swiper ref={myswiper}/>
```

useReducer和useContext(减少组件层级)

```
import React from 'react'
var GlobalContext= React.createContext()
// 注意此时的reducer 返回值是一个对象 {isShow:false,list:[]}

function App(props){
    let [state,dispatch] = useReducer(reducer,{isShow:true,list:[{}]})
    return <GlobalContext.Provider value={{ dispatch
}}>
    <div>
        {
            state.isShow?
            <div>我是选项卡</div>
            :null
        }
        {props.children}
    </div>
</GlobalContext.Provider>
}

function Detail(){
    var {dispatch} = useContext(GlobalContext)
    useEffect(() => {
        //隐藏
        dispatch({
            type:"Hide",
            payload:false
        })
        return () => {
            //显示
            dispatch({
                type:"Show",
                payload:true
            })
        };
    }, [])
    return <div>
        detail
    </div>
}

}
```

自定义hooks

当我们想在两个函数之间共享逻辑时，我们会把它提取到第三个函数中。

必须以“use”开头吗？必须如此。这个约定非常重要。不遵循的话，由于无法判断某个函数是否包含对其内部 Hook 的调用，React 将无法自动检查你的 Hook 是否违反了 Hook 的规则

十. React 路由

1. 什么是路由？

路由是根据不同的 url 地址展示不同的内容或页面。

2. 路由安装

<https://reacttraining.com/react-router/web/guides/quick-start>

```
npm install react-router-dom@5
```

3. 路由使用

(1) 路由方法导入

```
import React from "react";
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link
} from "react-router-dom";
```

(2) 定义路由以及重定向

```
<HashRouter>
  <Switch>
    <Route path="/films" component={Films}/>
    <Route path="/cinemas" component={Cinemas}/>
    <Route path="/center" component={Center}/>

    <Redirect from="/" to="/films" />
    {/* <Redirect from="/" to="/films" exact/>
      <Route path="/" component={NotFound}/> */}
  </Switch>
</HashRouter>
```

注意： a.

- b. exact 精确匹配 (Redirect 即使使用了exact, 外面还要嵌套Switch 来用)
- c. Warning: Hash history cannot PUSH the same path; a new entry will not be added to the history stack,这个警告只有在hash 模式会出现。

(3) 嵌套路由

```
<Switch>
  <Route path="/films/nowplaying" component={Nowplaying}/>
  <Route path="/films/comingsoon" component={Comingsoon}/>

  <Redirect from="/films" to="/films/nowplaying"/>
</Switch>
```

(4) 路由跳转方式

a. 声明式导航

```
<NavLink to="/films" activeClassName="active">films</NavLink>
<NavLink to="/cinemas" activeClassName="active">cinemas</NavLink>
<NavLink to="/center" activeClassName="active">center</NavLink>
```

b. 编程式导航

```
this.props.history.push('/center')
```

(5) 路由传参

```
(1)
this.props.history.push({ pathname : '/user' ,query : { day: 'Friday'} })
this.props.location.query.day
(2)
this.props.history.push({ pathname:'/user',state:{day : 'Friday' } })
this.props.location.state.day
```

(6) 路由拦截

```
<Route path="/center" render={()=>isAuth() ?<Center/>:<Login/>} />
```

(7) withRouter的应用与原理

You can get access to the `history` object's properties and the closest `<Route>`'s `match` via the `withRouter` higher-order component. `withRouter` will pass updated `match`, `location`, and `history` props to the wrapped component whenever it renders.

```
import { withRouter } from "react-router";
withRouter(MyComponent);
withRouter(connect(...)(MyComponent))
```

4.项目注意

(1) 反向代理

<https://facebook.github.io/create-react-app/docs/proxying-api-requests-in-development>

```
npm install http-proxy-middleware --save
```

```

const { createProxyMiddleware } = require('http-proxy-middleware');

module.exports = function(app) {
  app.use(
    '/api',
    createProxyMiddleware({
      target: 'http://localhost:5000',
      changeOrigin: true,
    })
  );
};

```

(2) css module

<https://facebook.github.io/create-react-app/docs/adding-a-css-modules-stylesheet>

```

全局
:global(.active){

}

```

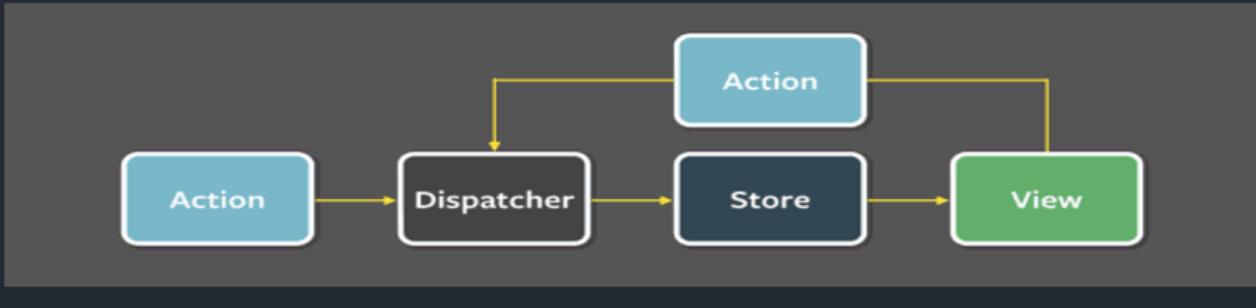
十一. Flux与Redux

Flux 是一种架构思想，专门解决软件的结构问题。它跟MVC架构是同一类东西，但是更加简单和清晰。Flux存在多种实现(至少15种)

<https://github.com/voronianski/flux-comparison>

Facebook Flux是用来构建客户端Web应用的应用架构。它利用**单向数据流**的方式来组合React中的视图组件。它更像一个模式而不是一个正式的框架，开发者不需要太多的新代码就可以快速的上手Flux。

1. 用户访问 View
2. View 发出用户的 Action
3. Dispatcher 收到 Action，要求 Store 进行相应的更新
4. Store 更新后，发出一个"change"事件
5. View 收到"change"事件后，更新页面

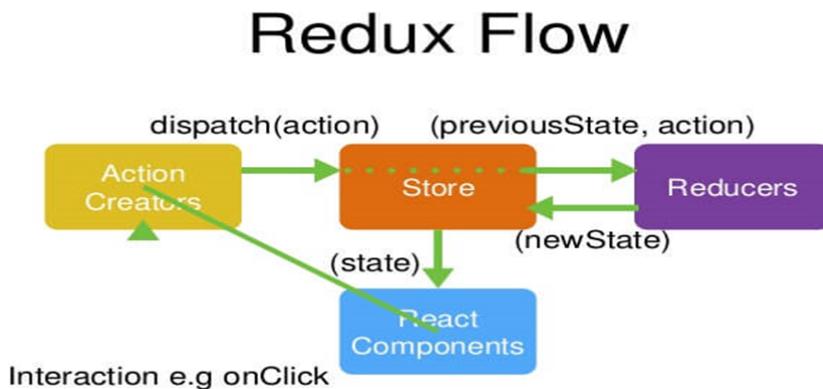


Redux最主要是用作应用状态的管理。简言之，Redux用一个单独的常量状态树（state对象）保存这一整个应用的状态，这个对象不能直接被改变。当一些数据变化了，一个新的对象就会被创建（使用actions和reducers），这样就可以进行数据追踪，实现时光旅行。

1. redux介绍及设计和使用的三大原则

- state 以单一对象存储在 store 对象中
- state 只读 (每次都返回一个新的对象)
- 使用纯函数 reducer 执行 state 更新

2. redux工作流



3. 与react绑定后使用redux实现案例

```
import {createStore} from "redux";
const reducer = (state="defaultState",data={})=>{
  let {type,payload} = data;
  switch(type){
    case "changetitle":
      return payload;
    default:
      return state;
  }
}
const store = createStore(reducer);
export default store;
```

```
store.dispatch({
  type:"changetitle",
  payload:res.data.data.film.name
})
componentDidMount() {
  store.subscribe(()=>{
    this.setState({
      title:store.getState()
    })
  })
}
```

4. redux原理解析

store 是通过 `createStore` 创建出来的，所以他的结构

```
export const createStore = 
function(reducer, initialState) {
  ...
  return {
    dispatch, // 用于action的分发，改变store里面的state (currentState = reducer(currentState,action))，并在内部遍历subscribe注册的监听器
    subscribe, // 注册listener，store里面state发生改变后，执行该listener
    getState, // 取store里面的state
    ...
  }
}
```

```

function createStore(reducer){
  var list = [];
  var state = reducer();
  function subscribe(callback){
    list.push(callback);
  }
  function dispatch(data){
    state =
    reducer(state,data);
    for(var i in list){
      list[i]();
    }
  }
  function getState(){
    return state;
  }
  return {
    subscribe,
    dispatch,
    getState
  }
}

```

5. reducer 扩展

如果如果不同的action所处理的属性之间没有联系，我们可以把 Reducer 函数拆分。不同的函数负责处理不同属性，最终把它们合并成一个大的 Reducer 即可。

```

import {combineReducers} from "redux";
const reducer = combineReducers({
  a: functionA,
  b: functionB,
  c: functionC
})

```

访问：

```

(state)=>{
  return {
    kerwinstate:state.a (不同的命名空间)
  }
}

```

6. redux中间件

在redux里，action仅仅是携带了数据的普通js对象。action creator返回的值是这个action类型的对象。然后通过store.dispatch()进行分发。同步的情况下一切都很完美，但是reducer无法处理异步的情况。

那么我们就需要在action和reducer中间架起一座桥梁来处理异步。这就是middleware。

i. 中间件的由来与原理、机制

```

export default function thunkMiddleware({ dispatch, getState }) {
  return next => action =>
    typeof action === 'function' ?
      action(dispatch, getState) :
      next(action);
}

```

这段代码的意思是，中间件这个桥梁接受到的参数action，如果不是function则和过去一样直接执行next方法(下一步处理)，相当于中间件没有做任何事。如果action是function，则先执行action，action的处理结束之后，再在action的内部调用dispatch。

ii. 常用异步中间件：

a. redux-thunk (store.dispatch参数可以是一个function)

```

import thunk from 'redux-thunk';
import {applyMiddleware} from "redux";
const store = createStore(fetchReducer, applyMiddleware(thunk));

const getComingSoon = ()=>{
  //进行异步请求
  return (dispatch,store)=>{
    }
}

```

b. redux-promise (store.dispatch参数可以是一个promise对象)

```

import promiseMiddleware from 'redux-promise';
const store = createStore(fetchReducer, applyMiddleware(thunk,promiseMiddleware));

const getComingSoon = ()=>{
  //进行异步请求
  return axios.get(`****`).then(res=>{
    return {
      type:"cominglist",
      info:res.data.data
    }
  })
}

```

7. Redux DevTools Extension

<https://github.com/zalmoxisus/redux-devtools-extension>

```

import { createStore, compose} from 'redux'
import reducer from './reducer'

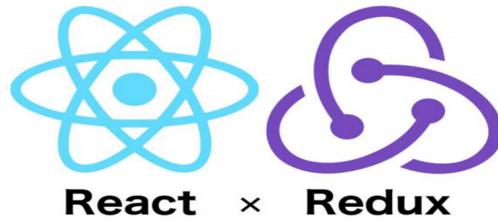
const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;
const store = createStore(reducer, /* preloadedState, */ composeEnhancers())

export default store

```

十二. react-redux

1. 介绍



<https://github.com/reactjs/react-redux>

2. 容器组件与UI组件

(1) UI组件

- 只负责 UI 的呈现，不带有任何业务逻辑
- 没有状态（即不使用this.state这个变量）
- 所有数据都由参数（this.props）提供
- 不使用任何 Redux 的 API

(2) 容器组件

- 负责管理数据和业务逻辑，不负责 UI 的呈现
- 带有内部状态
- 使用 Redux 的 API

3. Provider与connect

(1) React-Redux 提供Provider组件，可以让容器组件拿到state。

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

import { Provider } from 'react-redux'
import store from './store'

import App from './App'

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

(2) React-Redux 提供connect方法，用于从 UI 组件生成容器组件。connect的意思，就是将这两种组件连起来。

```
import { connect } from 'react-redux'
import { increment, decrement, reset } from './actionCreators'

// const Counter = ...
```

```

const mapStateToProps = (state /*, ownProps*/) => {
  return {
    counter: state.counter
  }
}
const mapDispatchToProps = { increment, decrement, reset }
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Counter)

```

4. HOC与context通信在react-redux底层中的应用

- (1) connect 是HOC， 高阶组件
- (2) Provider组件，可以让容器组件拿到state， 使用了context

5. 高阶组件构建与应用

HOC不仅仅是一个方法，确切说应该是一个组件工厂，获取低阶组件，生成高阶组件。

- (1)代码复用，代码模块化
- (2)增删改props
- (3)渲染劫持

```

// child.js

//高阶函数
function Control(wrappedComponent) {
  return class MyControl extends React.Component {
    render(){
      if(!this.props.data) {
        return <div>loading...</div>
      }
      return <wrappedComponent {...props} />
    }
  }
}

class MyComponent extends React.Component {
  render(){
    return <div>{this.props.data}</div>
  }
}
export default Control(MyComponent); //高阶组件

```

```

//Parent.js
import MyControlComponent from "./Child"
<MyControlComponent data={this.state.value}/>

//在父级传入data是null的时候，这一块儿就只会显示loading...
//不会显示组件的具体内容，如果data不为null，就显示真实组件信息。

```

6. Redux 持久化

```

import {persistStore, persistReducer} from 'redux-persist';
import storage from 'redux-persist/lib/storage';
import autoMergeLevel2 from 'redux-persist/lib/stateReconciler/autoMergeLevel2';

const persistConfig = {
  key: 'kerwin',
  storage: storage,
  //localStorage: import storage from 'redux-persist/lib/storage'
  //sessionStorage: import storageSession from 'redux-persist/lib/storage/session'
  stateReconciler: autoMergeLevel2
  //控制在本地存储中，新老状态怎么合并，覆盖？或者合并？
};

//改造reducer
const myPersistReducer = persistReducer(persistConfig, reducer)

//改造store
export const persistor = persistStore(store)

//改造根组件
import {persistor} from './store'
import {PersistGate} from 'redux-persist/lib/integration/react';

<PersistGate loading={null} persistor={persistor}>
  ...
</PersistGate>

```

十三. UI组件库

Ant Design 是一个致力于提升『用户』和『设计者』使用体验的设计语言；旨在统一中台项目的前端 UI 设计，屏蔽不必要的设计差异和实现成本，解放设计和前端的研发资源；包含很多设计原则和配套的组件库。

1. ant-design (PC端)

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

<https://ant-design.gitee.io/index-cn> (镜像库，快)

2. antd-mobile (移动端)

<https://mobile.ant.design>

十四. Immutable

```

var obj = { /* 一个复杂结构的对象 */ };
doSomething(obj);
// 上面的函数之行完后，此时的 obj 还是最初的那个 obj 吗？

// deepCopy?

```

1.Immutable.js介绍

<https://github.com/immutable-js/immutable-js>

每次修改一个 Immutable 对象时都会创建一个新的不可变的对象，在新对象上操作并不会影响到原对象的数据。

这个库的实现是深拷贝还是浅拷贝？

2. 深拷贝与浅拷贝的关系

- (1) var arr = {} ; arr2 = arr ;
- (2) Object.assign() 只是一级属性复制，比浅拷贝多拷贝了一层而已。
- (3) const obj1 = JSON.parse(JSON.stringify(obj)); 数组，对象都好用的方法(缺点：不能有undefined)

3. Immutable优化性能的方式

Immutable 实现的原理是 Persistent Data Structure (持久化数据结构)，也就是使用旧数据创建新数据时，要保证旧数据同时可用且不变。同时为了避免 deepCopy 把所有节点都复制一遍带来的性能损耗，Immutable 使用了 Structural Sharing (结构共享)，即如果对象树中一个节点发生变化，只修改这个节点和受它影响的父节点，其它节点则进行共享。

https://upload-images.jianshu.io/upload_images/2165169-cebb05bca02f1772

4. Immutable中常用类型 (Map, List)

- (1) Map

```
const { Map } = require('immutable');
const map1 = Map({ a: 1, b: 2, c: 3 });
const map2 = map1.set('b', 50);
map1.get('b') + " vs. " + map2.get('b'); // 2 vs. 50
```

```
import { Map } from 'immutable';
let a = Map({
  select: 'users',
  filter: Map({ name: 'Cam' })
})
let b = a.set('select', 'people');

a === b; // false
a.get('filter') === b.get('filter'); // true
```

延深：如果上述select 属性给一个组件用，因为此值改变了，shouldComponentUpdate 应该返回true，而filter 属性给另一个组件用，通过判断并无变化，shouldComponentUpdate 应该返回false，此组件就避免了重复进行diff对比

- (2) List

```

const { List } = require('immutable');
const list1 = List([ 1, 2 ]);
const list2 = list1.push(3, 4, 5);
const list3 = list2.unshift(0);
const list4 = list1.concat(list2, list3);
assert.equal(list1.size, 2);
assert.equal(list2.size, 5);
assert.equal(list3.size, 6);
assert.equal(list4.size, 13);
assert.equal(list4.get(0), 1);
//push, set, unshift or splice 都可以直接用，返回一个新的immutable对象

```

(3) merge , concat

```

const { Map, List } = require('immutable');
const map1 = Map({ a: 1, b: 2, c: 3, d: 4 });
const map2 = Map({ c: 10, a: 20, t: 30 });
const obj = { d: 100, o: 200, g: 300 };
const map3 = map1.merge(map2, obj);
// Map { a: 20, b: 2, c: 10, d: 100, t: 30, o: 200, g: 300 }
const list1 = List([ 1, 2, 3 ]);
const list2 = List([ 4, 5, 6 ]);
const array = [ 7, 8, 9 ];
const list3 = list1.concat(list2, array);
// List [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]

```

(4) toJS

```

const { Map, List } = require('immutable');
const deep = Map({ a: 1, b: 2, c: List([ 3, 4, 5 ]) });
console.log(deep.toObject()); // { a: 1, b: 2, c: List [ 3, 4, 5 ] }
console.log(deep.toArray()); // [ 1, 2, List [ 3, 4, 5 ] ]
console.log(deep.toJS()); // { a: 1, b: 2, c: [ 3, 4, 5 ] }
JSON.stringify(deep); // '{"a":1,"b":2,"c":[3,4,5]}'

```

(5)fromJS

```

const { fromJS } = require('immutable');
const nested = fromJS({ a: { b: { c: [ 3, 4, 5 ] } } });
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ] } } }

const nested2 = nested.mergeDeep({ a: { b: { d: 6 } } });
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 6 } } }

console.log(nested2.getIn([ 'a', 'b', 'd' ])); // 6
//如果取一级属性 直接通过get方法，如果取多级属性  getIn(["a","b","c"]))

//setIn 设置新的值
const nested3 = nested2.setIn([ 'a', 'b', 'd' ], "kerwin");
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: "kerwin" } } }

//updateIn 回调函数更新

```

```

const nested3 = nested2.updateIn([ 'a', 'b', 'd' ], value => value + 1);
console.log(nested3);
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 7 } } }

const nested4 = nested3.updateIn([ 'a', 'b', 'c' ], list => list.push(6));
// Map { a: Map { b: Map { c: List [ 3, 4, 5, 6 ], d: 7 } } }

```

5. Immutable+Redux的开发方式

```

//reducer.js
const initialState = fromJS({
  category:"",
  material:""
})
const reducer = (prevstate = initialState,action={})=>{
  let {type,payload} = action
  switch(type){
    case GET_HOME:
      var newstate = prevstate.set("category",fromJS(payload.category))
      var newstate2 = newstate.set("material",fromJS(payload.material))
      return newstate2;
    default:
      return prevstate
  }
}

//home.js
const mapStateToProps = (state)=>{
  return {
    category:state.homeReducer.getIn(["category"]) || Map({}),
    material:state.homeReducer.getIn(["material"]) || Map({})
  }
}

this.props.category.get("相关属性")
this.props.category.toJS() //或者转成普通对象

```

6. 缺点

容易跟原生混淆

文档与调试不方便

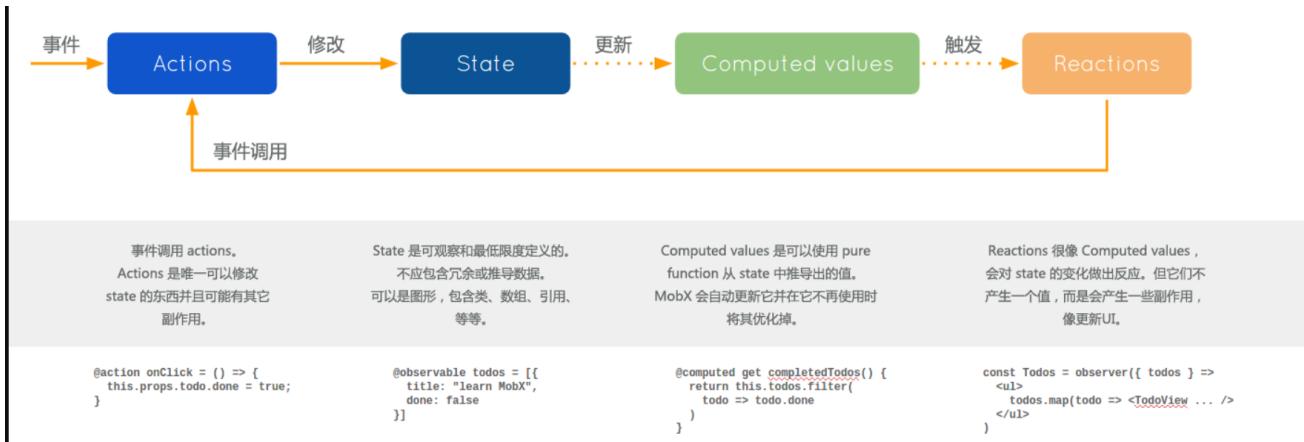
十五. Mobx

<https://cn.mobx.js.org/>

1. Mobx介绍

- (1) Mobx是一个功能强大，上手非常容易的状态管理工具。
- (2) Mobx背后的哲学很简单：任何源自应用状态的东西都应该自动地获得。
- (3) Mobx利用getter和setter来收集组件的数据依赖关系，从而在数据发生变化的时候精确知道哪些组件需要重绘，在界面的规模变大的时候，往往会有许多细粒度更新。

(vue类似)



2. Mobx与redux的区别

- Mobx写法上更偏向于OOP
 - 对一份数据直接进行修改操作，不需要始终返回一个新的数据
 - 并非单一store,可以多store。
 - Redux默认以JavaScript原生对象形式存储数据，而Mobx使用可观察对象.

优点:

- a. 学习成本小
- b. 面向对象编程, 而且对 TS 友好

缺点:

- a. 过于自由: Mobx提供的约定及模版代码很少, 代码编写很自由, 如果不做一些约定, 比较容易导致团队代码风格不统一,
- b. 相关的中间件很少, 逻辑层业务整合是问题。

3. Mobx的使用

(1) observable 和 autorun

```
import { observable, autorun } from 'mobx';

const value = observable.box(0);
const number = observable.box(100);
autorun(() => {
  console.log(value.get());
});
value.set(1);
value.set(2);
number.set(101);
//0,1,2。 // autorun 使用到才能被执行
//只能是同步，异步需要处理

//观察对象，通过map
const map = observable.map({ key: "value" });
//map.set("key", "new value");
//map.get("key")
```

```
// 观察对象，不通过map
const map = observable({ key: "value"});
// map.key map.key="xiaoming"

// 观察数组
const list = observable([1, 2, 4]);
list[2] = 3;
```

(2) action, runInAction和严格模式

```
import {observable, action, configure, runInAction} from 'mobx';
configure({enforceActions: 'always'})
// 严格模式，必须写action,
// 如果是never，可以不写action,
// 最好设置always，防止任意地方修改值，降低不确定性。
class Store {
  @observable number = 0;
  @observable name = "kerwin";
  @action add = () => {
    this.number++;
  }
  // action 只能影响正在运行的函数，而无法影响当前函数调用的异步操作
  @action load = async () => {
    const data = await getData();
    runInAction(() => {
      this.name = data.name;
    });
  }
  // runInAction 解决异步问题
}
const newStore = new Store();
newStore.add();

// 如果在组件监听
componentDidMount() {
  autorun(()=>{
    console.log(newStore.number);
  })
}
```

4. mobx-react的使用

(1) react 组件里使用 @observer

observer 函数/装饰器可以用来将 React 组件转变成响应式组件。

(2) 可观察的局部组件状态

@observable 装饰器在React组件上引入可观察属性。而不需要通过 React 的冗长和强制性的 setState 机制来管理。

```
import {observer} from "mobx-react"
import {observable} from "mobx"
```

```

@observer class Timer extends React.Component {
  @observable secondsPassed = 0

  componentWillMount() {
    setInterval(() => {
      this.secondsPassed++
    }, 1000)
  } //如果是严格模式需要加上 @action 和 runInAction

  //一个新的生命周期钩子函数 componentWillMountReact
  //当组件因为它观察的数据发生了改变，它会安排重新渲染,
  //这个时候 componentWillMountReact 会被触发
  componentWillMountReact() {
    console.log("I will re-render, since the todo has changed!");
  }

  render() {
    return (<span>Seconds passed: { this.secondsPassed } </span> )
  }
}

ReactDOM.render(<Timer />, document.body)

```

(3) Provider 组件

它使用了 React 的上下文(context)机制，可以用来向下传递 stores。要连接到这些 stores，需要传递一个 stores 名称的列表给 inject，这使得 stores 可以作为组件的 props 使用。this.props

```

class Store {
  @observable number = 0;
  @action add = () => {
    this.number++;
  }
}
export default new Store() //导出Store实例

```

```

@inject("kerwinstore")
@observer //需要转换为响应式组件
class Child extends Component{
  render(){
    return <div>
      Child --{this.props.kerwinstore.number}
    </div>
  }
}

@inject("kerwinstore")
class Middle extends Component{
  render(){
    return <div>
      Middle-<button onClick={()=>{
        this.props.kerwinstore.add();
      }}>test</button>
    </div>
  }
}

```

```
<Child/>
</div>
}
}
//通过provider传store进去
<Provider store={store}>
    <Middle/>
</Provider>
```

5. 支持装饰器

```
npm i @babel/core @babel/plugin-proposal-decorators @babel/preset-env
```

创建 .babelrc

```
{
  "presets": [
    "@babel/preset-env"
  ],
  "plugins": [
    [
      "@babel/plugin-proposal-decorators",
      {
        "legacy": true
      }
    ]
  ]
}
```

创建config-overrides.js

```
const path = require('path')
const { override, addDecoratorsLegacy } = require('customize-cra')

function resolve(dir) {
  return path.join(__dirname, dir)
}

const customize = () => (config, env) => {
  config.resolve.alias['@'] = resolve('src')
  if (env === 'production') {
    config.externals = {
      'react': 'React',
      'react-dom': 'ReactDOM'
    }
  }

  return config
};

module.exports = override(addDecoratorsLegacy(), customize())
```

安装依赖

```
npm i customize-cra react-app-rewired
```

修改package.json

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

▲ 扩展 (1)
TypeScript (1)

Javascript › Implicit Project Config: Experimental Decorators

对不属于任何工程的 JavaScript 文件启用或禁用 experimentalDecorators 设置。
覆盖此设置。要求工作区使用高于 2.3.1 版本的 TypeScript。

十六. TS

1-typescript

[文档地址](#)

1. TypeScript 的定位是静态类型语言，在写代码阶段就能检查错误，而非运行阶段
2. 类型系统是最好的文档，增加了代码的可读性和可维护性。
3. 有一定的学习成本，需要理解接口（Interfaces）、泛型（Generics）、类（Classes）等
4. ts最后被编译成js

2-安装

```
create-react-app my-app --template typescript
```

```
You are running `create-react-app` 4.0.3, which is behind the latest release (5.0.0).  
We no longer support global installation of Create React App.  
Please remove any global installs with one of the following commands:  
- npm uninstall -g create-react-app  
- yarn global remove create-react-app
```

3-声明

1. 可以在当前文件加上`declare const $: any;`
2. 安装 `npm i @types/jquery` `@types`是`npm`的一个分支，用来存放`*.d.ts`文件

```
npm i --save react-router-dom
npm i --save @types/react-router-dom //编译器需要通过这个声明文件，进行类型检查工作
```

4-变量声明

```
// string(原生的构造函数) vs string (ts中的类型)
var myname:string = "字符"
var mybool:boolean = false
var mynumber:number = 100
var mylist:Array<string> = ["111","222","3333"]

var myname2:string | number | boolean = 100
var myname3:string | number = "kerwin"
var mylist2:Array<string| number> = [1,2,"kerwin"]
var mylist3:(string| number)[] = [1,2,"kerwin"]
```

5-定义普通函数

接口描述形状

```
interface SearchFunc {
  (source: string, subString: string): boolean;
}

//对于函数类型的类型检查来说，函数的参数名不需要与接口里定义的名字相匹配。
let mySearch: SearchFunc;
mySearch = function(src: string, sub: string): boolean {
  let result = src.search(sub);
  return result > -1;
}
```

传参

```
function Test(list:String[],text?:String,...args:String[]):void{
  console.log(list,text,args)
}

Test(["1111","2222"])
//list:["1111","2222"] text: undefined args: []

Test(["0","1"],"a","b","c")

//list:["0","1"] text: "a" args: ["b","c"]
```

类型断言as

```
function Test( mytext:string|number ){

  console.log((mytext as string).length) //对
  console.log((mytext as any).length) //对
  console.log((mytext as string[]).length) //错，原声明没有这个类型，无法断言
}
```

6-定义普通类

```
interface MyInter {
  name:String, //必选属性
  readonly country:String,//只读属性
  getName():void //定义方法
}
```

```

class MyObj implements MyInter{
    name="kerwin"
    country="China"
    private age = 100 //私有属性， 不能在接口定义
    getName(){
        //...
    }
    private getAge(){
        //...
    } //私有方法， 不能在接口定义
}

```

7-定义类组件

```

interface PropInter {
    name: string | number;
    firstName?: string;//可选属性
    lastName?: string;//可选属性
    // [propName: string]: any 任意属性
}
interface StateInter {
    count: number
}
//根组件 ， 第一个参数可以传any
class HelloClass extends React.Component<PropInter, StateInter> {
    state: State = {
        count: 0,
    }; //setState时候也才会检查
    static defaultProps = { // 属性默认值
        name: "default name"
        firstName: "",
        lastName: ""
    };
}

```

8-定义函数式组件

```

//根组件
const App:React.FC = (props)=>{
    console.log(props)
    const [name, setname] = useState<string>("kerwin")
    return <div>
        app
    </div>
}
//子组件接受属性 -1
interface iprops {
    count:number
}
const Child:React.FC<iprops> = (props)=>{
    return <div>
        child-{props.count}
    </div>
}
//子组件接受属性 -2
const Child = (props:iprops)=>{

```

```
    return <div>
      child-{props.count}
    </div>
}
```

useRef

```
const mytext = useRef<HTMLInputElement>(null)

<input type="text" ref={mytext}/>

useEffect(() => {
  console.log(mytext.current && mytext.current.value)
}, [])
```

useContext

```
interface IContext{
  call:string
}

const GlobalContext = React.createContext<IContext>({
  call:"" //定义初始值,按照接口规则
})

<GlobalContext.Provider value={{ 
  call:"电话"
}}>
  ....
</GlobalContext.Provider>

const {call} = useContext(GlobalContext)
```

useReducer

```
interface IPrevState{
  count:number
}
interface IAction{
  type:string,
  payload:any
}
function reducer (prevState:IPrevState,action:IAction){
  ....
  return prevState
}
const [state,dispatch]= useReducer(reducer,{
  count:1
})

dispatch({
  type:"Action1",
  payload:[]
})
```

```

//父组件调用
<Child key={index} item={item} index={index} cb={(index)=>{
    var newList= [...list]
    newList.splice(index,1)
    setList(newList)
}}/>

//子组件
interface ItemType{
    item:string,
    index:number, //定义接口
    cb:(param:number)=>void //定义接口
}

const Child = (props:ItemType)=>{
    let {index,item,cb} = props
    return <div>{item}
        <button onClick={()=>cb(index)}>del-{index}</button>
    </div>
}

```

10-路由

编程式导航

```

// 使用编程式导航，需要引入接口配置
import { RouteComponentProps } from "react-router-dom";

interface IProps {自己定义的接口}

type HomeProps = IProps & RouteComponentProps; //两个接口属性都支持

interface IState {}

class Home extends React.Component<HomeProps, IState> {
    private handleSubmit = async () => {
        //code for API calls
        this.props.history.push("/home");
    };

    public render(): any {
        return <div>Hello</div>;
    }
}

```

动态路由

```

interface IParams{
    id:string
}
// RouteComponentProps是一个泛型接口
class Detail extends Component< RouteComponentProps<IParams> >{
    componentDidMount() {
        console.log(this.props.match.params.id)
    }
}

```

```
render(){
    return <div>
        detail
    </div>
}
}
```

11-redux

```
import {createStore} from 'redux'
interface ActionInter{
    type:String,
    payload:any
}
const reducer = (prevState={},action:ActionInter)=>{
    return action.payload
}
const store = createStore(reducer, //enhancer)
export default store
```

十七. styled-components

它是通过JavaScript改变CSS编写方式的解决方案之一，从根本上解决常规CSS编写的一些弊端。

通过JavaScript来为CSS赋能，我们能达到常规CSS所不好处理的逻辑复杂、函数方法、复用、避免干扰。样式书写将直接依附在SX上面，HTML、CSS、JS三者再次内聚。**all in js的思想**

基本

```
const StyleApp = styled.div`  
background:yellow;  
border:1px solid black;  
ul{  
    li{  
        color:red;  
    }  
}  
  
&:hover{  
    background:pink  
} //pc 测试  
  
/*  
<StyleApp>  
    <ul>  
        <li>1111</li>  
        <li>22222</li>  
    </ul>  
</StyleApp>  
*/
```

透传props

```
const styledInput = styled.input`  
  color: red;  
  background: yellow;  
  border: none;  
  border-radius: 3px;  
  
<styledInput type="text" placeholder="okok"/>
```

基于props做样式判断

```
const StyledButton = styled.button`  
  background:${props=>props.bg || 'blue'}  
  
/*<StyledButton>click</StyledButton>  
<StyledButton bg="red">click</StyledButton>*/
```

样式化任意组件(一定要写className)

```
const child = (props)=><div className={props.className}>child</div>  
  
const Styledchild = styled(child)`  
  background:red;  
  
<Styledchild/>
```

扩展样式

```
const MyButton = styled.button`  
  background:yellow;  
  
const BigButton = styled(MyButton)`  
  height:100px;  
  width:100px;
```

加动画

```
import styled,{keyframes} from 'styled-components'  
const rotate360 = keyframes`  
  from {  
    transform: rotate(0deg);  
  }  
  to {  
    transform: rotate(360deg);  
  }  
;  
  
const Rotate = styled.div`  
  width:100px;  
  height:100px;  
  background:yellow;  
  animation:${rotate360} 1s linear infinite;
```

十八. 单元测试

挂载组件

```
import Enzyme,{mount} from 'enzyme';
import Adapter from '@wojtekmaj/enzyme-adapter-react-17'
//在使用Enzyme 前需要先适配React对应的版本
Enzyme.configure({ adapter: new Adapter() })

it('挂载拿到状态', () => {
  const app = mount(<App />);
  expect(app.state().name).toEqual('kerwin');
  expect(app.state().age).toEqual(100);
})

/*
.text(): 返回当前组件的文本内容

.html(): 返回当前组件的HTML代码形式

.props(): 返回根组件的所有属性

.prop(key): 返回根组件的指定属性

.state([key]): 返回根组件的状态

.setState(nextState): 设置根组件的状态

.setProps(nextProps): 设置根组件的属性

*/

```

测试组件渲染出来的 HTML

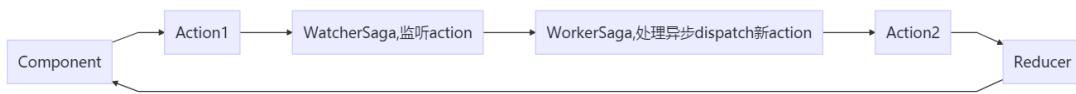
```
it('组件渲染出来的 HTML', () => {
  const app = mount(<App />);
  expect(app.find('#myid').text()).toEqual('kerwin');
})
```

模拟用户交互

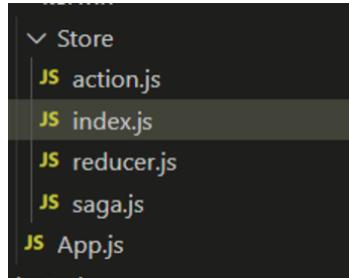
```
it('模拟用户交互', () => {
  const app = mount(<App />);
  app.find('#mybtn').simulate('click')
  expect(app.state().name).toEqual('xiaoming');
})
```

十九. redux-saga

在saga中，全局监听器和接收器使用Generator函数和saga自身的一些辅助函数实现对整个流程的管控



代码实现



```
//index.js
import {createStore, applyMiddleware} from 'redux'
import createSagaMiddleware from 'redux-saga';
import {reducer} from './reducer'
import mySagas from './saga'
const sagaMiddleware = createSagaMiddleware(); //创建中间件
const store = createStore(reducer, {list: []}, applyMiddleware(sagaMiddleware))

//注意运行的时机是在store创建好了之后
sagaMiddleware.run(mySagas);

export default store
```

```
//saga.js
import {takeEvery, put} from 'redux-saga/effects'
import {changeList} from './action'
function *mySagas(){
    //监听GET_LIST
    //在每个 `监听GET_LIST` action 被 dispatch 时调用 getList
    yield takeEvery("GET_LIST", getList);
    //yield takeEvery("DELETE_LIST", deleteList);
}

function *getList(){
    //异步处理
    let res = yield new Promise(resolve=>{
        setTimeout(()=>{
            resolve(["1111", "2222", "3333"])
        }, 2000)
    })
    yield put(changeList(res)) //发出新的action
}
export default mySagas
```

```
//action.js
export const changeList = (value)=>{
    return {
        type: "CHANGE_LIST",
        payload: value
}
```

```
        }
    }

export const getSaAction = ()=>{
    //GET_LIST 被saga监听
    return {
        type:"GET_LIST"
    }
}
```

```
//reducer.js
export const reducer = (prevState,action)=>{
    let {type,payload} = action;
    switch(type){
        case "CHANGE_LIST":
            let newstate = {...prevState}
            newstate.list = [...newstate.list,...payload]
            return newstate
        default :
            return prevState
    }
}
```

```
//App.js
class App extends Component {
    componentDidMount() {
        store.subscribe(()=>{
            console.log(store.getState())
        })
    }

    handleClick = ()=>{
        store.dispatch(getSaAction())
    }

    render() {
        return <div>
            <button onClick={this.handleClick}>获取异步</button>
        </div>
    }
}
```

二十. React补充

1. Portal

Portals 提供了一个最好的在父组件包含的DOM结构层级外的DOM节点渲染组件的方法。

```
ReactDOM.createPortal(child,container);
```

第一个参数child是可渲染的react子项，比如元素，字符串或者片段等。第二个参数container是一个DOM元素。

1、用法

普通的组件，子组件的元素将挂载到父组件的DOM节点中。

```
render() {
  // React 挂载一个div节点，并将子元素渲染在节点中
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

有时需要将元素渲染到DOM中的不同位置上去，这是就用到的portal的方法。

```
render(){
  // 此时React不再创建div节点，而是将子元素渲染到Dom节点上。domNode，是一个有效的任意位置的dom节点。
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  )
}
```

一个典型的用法就是当父组件的dom元素有 `overflow:hidden` 或者 `z-index` 样式，而你又需要显示的子元素超出父元素的盒子。举例来说，如对话框，悬浮框，和小提示。

2、在protal中的事件冒泡

虽然通过portal渲染的元素在父组件的盒子之外，但是渲染的dom节点仍在React的元素树上，在那个dom元素上的点击事件仍然能在dom树中监听到。

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

const getDiv = () => {
  const div = document.createElement('div');
  document.body.appendChild(div);
  return div;
};

const withPortal = (WrappedComponent) => {
  class AddPortal extends Component {
    constructor(props) {
      super(props);
      this.el = getDiv();
    }

    componentWillUnmount() {
      document.body.removeChild(this.el);
    }

    render(props) {
      return ReactDOM.createPortal(<WrappedComponent {...props} />, this.el);
    }
  }
  return AddPortal;
}
```

```

};

class Modal extends Component {
  render() {
    return (
      <div>
        <div>amodal content</div>
        <button type="button">Click</button>
      </div>
    );
  }
}

const PortalModal = withPortal(Modal);

class Page extends Component {
  constructor(props) {
    super(props);
    this.state = { clicks: 0 };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      clicks: state.clicks + 1
    }));
  }

  render() {
    return (
      <div onClick={this.handleClick}>
        <h3>pppppppp</h3>
        <h3>num: {this.state.clicks}</h3>
        <PortalModal />
      </div>
    );
  }
}

export default Page;

```

2.Lazy 和 Suspense

1、React.lazy 定义

`React.lazy` 函数能让你像渲染常规组件一样处理动态引入（的组件）。

什么意思呢？其实就是懒加载。

(1) 为什么代码要分割

当你的程序越来越大，代码量越来越多。一个页面上堆积了很多功能，也许有些功能很可能都用不到，但是一样下载加载到页面上，所以这里面肯定有优化空间。就如图片懒加载的理论。

(2) 实现原理

当 Webpack 解析到该语法时，它会自动地开始进行代码分割(Code Splitting)，分割成一个文件，当使用到这个文件的时候会这段代码才会被异步加载。

(3) 解决方案

在 `React.lazy` 和常用的三方包 `react-loadable`，都是使用了这个原理，然后配合 webpack 进行代码打包拆分达到异步加载，这样首屏渲染的速度将大大的提高。

由于 `React.lazy` 不支持服务端渲染，所以这时候 `react-loadable` 就是不错的选择。

2、如何使用 `React.lazy`

下面示例代码使用 create-react-app 脚手架搭建：

```
//otherComponent.js 文件内容
import React from 'react'
const OtherComponent = ()=>{
  return (
    <div>
      我已加载
    </div>
  )
}
export default OtherComponent

// App.js 文件内容
import React from 'react';
import './App.css';

//使用React.lazy导入OtherComponent组件
const OtherComponent = React.lazy(() => import('./OtherComponent'));
function App() {
  return (
    <div className="App">
      <OtherComponent/>
    </div>
  );
}
export default App;
```

这是最简单的 `React.lazy`，但是这样页面会报错。这个报错提示我们，在 React 使用了 `lazy` 之后，会存在一个加载中的空档期，React 不知道在这个空档期中该显示什么内容，所以需要我们指定。接下来就要使用到 `Suspense`。

Error: A React component suspended while rendering, but no fallback UI was specified.

x

Add a <Suspense fallback=...> component higher in the tree to provide a loading indicator or placeholder to display.

in Unknown (at App.js:22)
in header (at App.js:9)
in div (at App.js:8)
in App (at src/index.js:6)

► 19 stack frames were collapsed.

Module.../src/index.js
E:/code/lazydemo/src/index.js:6

```
3 | import './index.css';
4 | import App from './App';
5 |
> 6 | ReactDOM.render(<App />, document.getElementById('root'));
7 |
```

[View compiled](#)

(1) Suspense

如果在 `App` 渲染完成后，包含 `otherComponent` 的模块还没有被加载完成，我们可以使用加载指示器为此组件做优雅降级。这里我们使用 `Suspense` 组件来解决。

这里将 `App` 组件改一改

```
import React, { Suspense, Component } from 'react';
import './App.css';

// 使用React.lazy导入otherComponent组件
const otherComponent = React.lazy(() => import('./otherComponent'));

export default class App extends Component {
  state = {
    visible: false
  }
  render() {
    return (
      <div className="App">
        <button onClick={() => {
          this.setState({ visible: true })
        }}>
```

```

        加载OtherComponent组件
    </button>
    <Suspense fallback={<div>Loading...</div>}>
    {
        this.state.visible
        ?
        <OtherComponent />
        :
        null
    }
    </Suspense>
</div>
)
}
}

```

我们指定了空档期使用Loading展示在界面上面，等 otherComponent 组件异步加载完毕，把 otherComponent 组件的内容替换掉Loading上。

The screenshot shows the Network tab in the Chrome DevTools developer console. The title bar says "加载OtherComponent组件". The Network tab is selected, showing a list of resources. The "All" filter is applied. The resource list includes:

- websocket
- localhost
- bundle.js
- 0.chunk.js
- main.chunk.js
- info?t=1562685433132
- wrs_env.js
- manifest.json
- favicon.ico

The screenshot shows the Elements tab in the Chrome DevTools developer console. The title bar says "加载OtherComponent组件" and below it, "我已加载". The Elements tab is selected, showing the rendered HTML structure of the page. A red box highlights the script tag at the bottom of the head section:

```

<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link rel="shortcut icon" href="/favicon.ico">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <meta name="theme-color" content="#000000">
    <link rel="manifest" href="/manifest.json">
    <title>React App</title>
    <style type="text/css">...</style>
    <style type="text/css">...</style>
    <script charset="utf-8" src="/static/js/2.chunk.js"></script> == $0
  </head>
  <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
    <div id="root">...</div>
    <script src="/static/js/bundle.js"></script>
    <script src="/static/js/0.chunk.js"></script>
    <script src="/static/js/main.chunk.js"></script>
  </body>
</html>

```

为了演示我把chrome网络调到 lower-end mobile，不然看不到loading出现。

可以从上面图片看出，当点击加载的时候，页面的head会插入`这段代码，发出一个get请求，页面开始显示loading，去请求2.chunk.js`文件。

请求结束返回内容就是 otherComponent 组件的内容，只是文件名称和文件内容经过webpack处理过。

注意：Suspense 使用的时候，fallback一定是存在且有内容的，否则会报错。

3. forwardRef

引用传递 (Ref forwading) 是一种通过组件向子组件自动传递引用ref 的技术。对于应用者的大多数组件来说没什么作用。但是对于有些重复使用的组件，可能有用。例如某些input组件，需要控制其focus，本来是可以使用ref来控制，但是因为该input已被包裹在组件中，这时就需要使用Ref forward来透过组件获得该input的引用。可以透传多层

未使用forwardRef

```
//子组件
class Child extends Component{
  componentDidMount() {
    this.props.callback(this.refs.myinput)
  }

  render(){
    return <div>
      <input type="text" ref="myinput"/>
    </div>
  }
}

//父组件
class App extends Component {
  render() {
    return (
      <div>
        <Child callback={(el)=>{
          el.focus()
        }}/>
      </div>
    )
  }
}
```

使用forwardRef

```
//子组件
const Child = forwardRef((props, ref)=>{
  return <div>
    <input type="text" ref={ref}/>
  </div>
})

//父组件
class App extends Component {
```

```

myref = createRef()

componentDidMount() {
    this.myref.current.focus()
}

render() {
    return (
        <div>
            <child ref={this.myref}>/</child>
        </div>
    )
}
}

```

4. Functional Component缓存

为啥起memo这个名字？

在计算机领域，记忆化是一种主要用来提升计算机程序速度的优化技术方案。它将开销较大的函数调用的返回结果存储起来，当同样的输入再次发生时，则返回缓存好的数据，以此提升运算效率。

作用

组件仅在它的props发生改变的时候进行重新渲染。通常来说，在组件树中React组件，只要有变化就会走一遍渲染流程。但是React.memo()，我们可以仅仅让某些组件进行渲染。

与PureComponent区别

PureComponent 只能用于class组件，memo 用于functional组件

用法

```

import {memo} from 'react'

const Child = memo(()=>{
    return <div>
        <input type="text" />
    </div>
})

```

或者

```

const Child = ()=>{
    return <div>
        <input type="text" />
    </div>
}
const MemoChild = memo(Child)

```

二十一. React扩展

1. GraphQL

(1) 介绍与hello

```
query {
```

```

        user (id : "1") {
            name
            gender
            employee (first: 20) {
                name
                email
            }
            father {
                telephone
            }
            son {
                school
            }
        }
    }
}

```

2. 获取多个资源，只用一个请求
3. 描述所有可能类型的系统。便于维护，根据需求平滑演进，添加或者隐藏字段。

- restful一个接口只能返回一个资源，graphql一次可以获取多个资源。
- restful用不同的url来区分资源，graphql用类型区分资源

```

var express = require('express');
var graphqlHTTP = require('express-graphql');
var { buildSchema } = require('graphql');

// Construct a schema, using GraphQL schema language
var schema = buildSchema(`

  type Query {
    hello: String
  }
`);

// The root provides a resolver function for each API endpoint
var root = {
  hello: () => {
    return 'Hello world!';
  },
};

var app = express();
app.use('/graphql', graphqlHTTP({
  schema: schema,
  rootValue: root,
  graphiql: true,
}));
app.listen(4000);

```

构建schema, 这里定义查询的语句和类型

定义查询所对应的resolver, 也就是查询对应的处理器

(2) 参数类型与传递

- 基本类型：String, Int, Float, Boolean和ID。可以在shema声明的时候直接使用。
- [类型]代表数组，例如：[Int]代表整型数组

(3) mutation
(4) 结合数据库

```

        var mongoose =
        require("mongoose")
    )
    // 链接数据库-a

        mongoose.connect("mongodb://localhost:27017/maizuo",{
        useNewUrlParser: true,useUnifiedTopology: true })

        var FilmModel =
        mongoose.model("film",new
        mongoose.Schema({
        name:String,
        poster:String,
        price:Number
    }))

}

}

```

(5) 客户端访问

```

var username = 3;
var query = `query Account($username: Int!) {
  account(username: $username)
}`;

fetch('/graphql', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
  },
  body: JSON.stringify({
    query,
    variables: { username },
  })
})
.then(r => r.json())
.then(data => console.log('data returned:', data));

```

Redux-saga

dva-cli is deprecated, please use create-umi instead, checkout <https://umijs.org/guide/create-umi-app.html> for detail.

dva 应用的最简结构

(6) 结合React

2. dva

<https://dvajs.com/guide>

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 = React-Router + Redux +

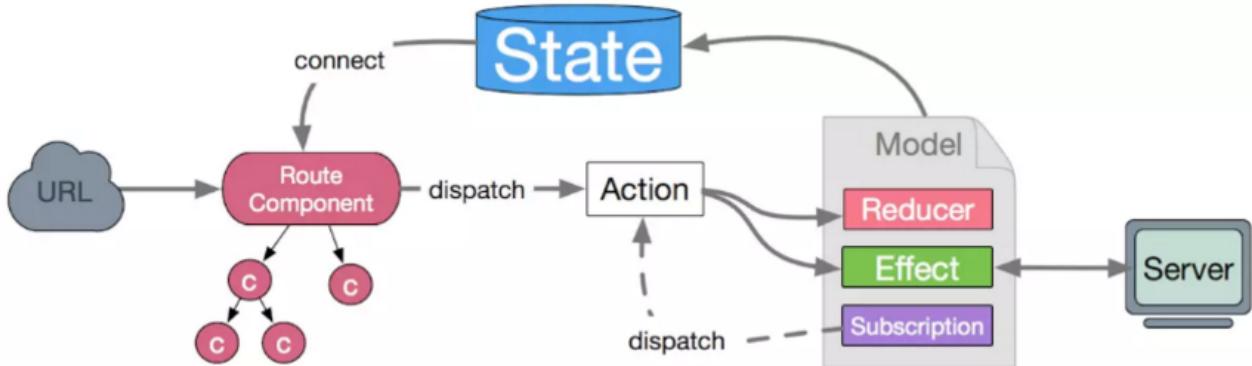
```

import dva from 'dva';
const App = () => <div>Hello dva</div>

// 创建应用
const app = dva();
// 注册视图
app.router(() => <App />);
// 启动应用
app.start('#root');

```

数据流图



3. umi

地址

umi，中文可发音为乌米，是一个可插拔的企业级 react 应用框架。umi 以路由为基础的，支持类 next.js 的约定式路由，以及各种进阶的路由功能，并以此进行功能扩展，比如支持路由级的按需加载。umi 在约定式路由的功能层面会更像 nuxt.js 一些。

开箱即用，省去了搭框架的时间

安装脚手架

```

$ mkdir myapp && cd myapp //空目录
$ npx @umijs/create-umi-app

```

目录

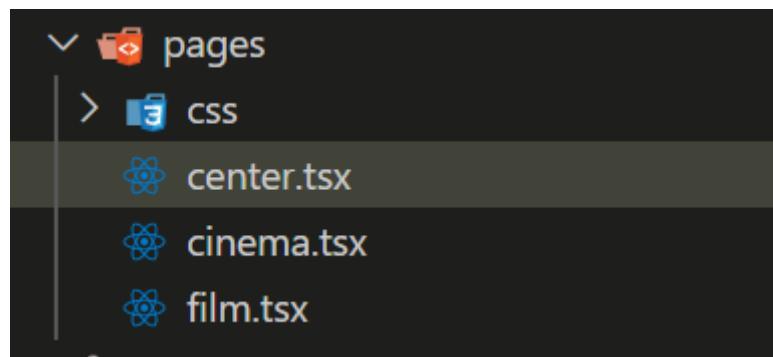
一个基础的 Umi 项目大致是这样的,

```
.  
├── package.json  
├── .umirc.ts  
├── .env  
├── dist  
├── mock  
├── public  
└── src  
    ├── .umi  
    ├── layouts/index.tsx  
    ├── pages  
    │   ├── index.less  
    │   └── index.tsx  
    └── app.ts
```

路由

umi 会根据 `pages` 目录自动生成路由配置。需要注释 `umirc.js`, `routes` 相关, 否则自动配置不生效

(1) 基础路由



(2) 重定向

```
//新建pages/index.js

import React from 'react';
import {Redirect} from 'umi'
export default () => {
  return (
    <Redirect to="/film"/>
  );
}

// 在film中的_layout.js

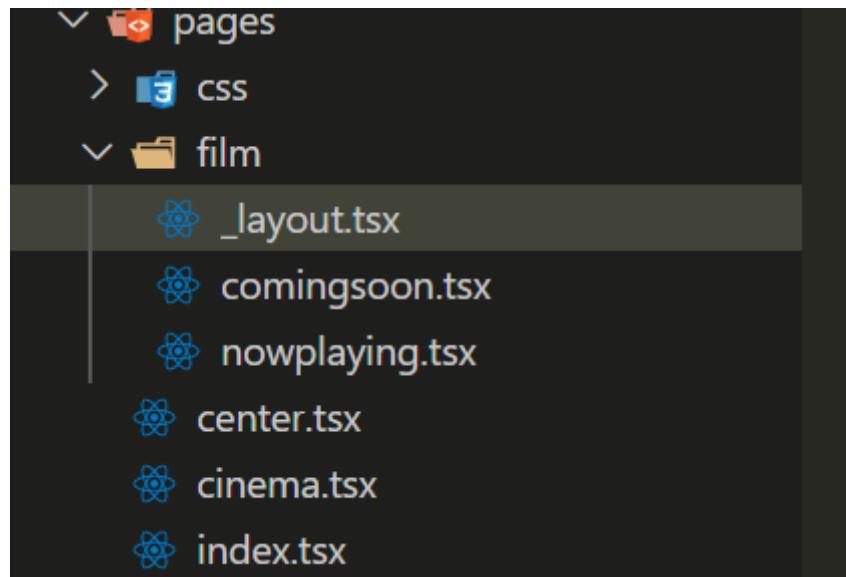
import {Redirect} from 'umi'

export default function Film(props) {
  if(props.location.pathname === '/film' || props.location.pathname === '/film/'){
    // ...
  }
}
```

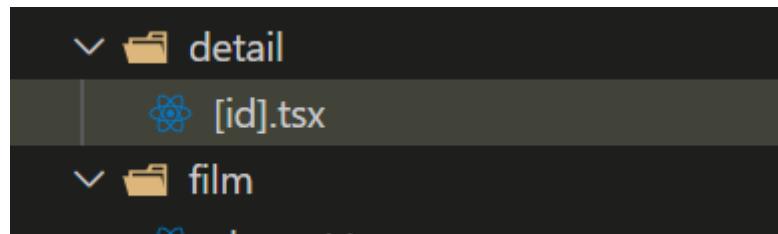
```
        return <Redirect to="/film/nowplaying" />
    }
    return (
        <div>
            {props.children}
        </div>
    )
}
```

(3) 嵌套路由

有时候不好用，重启一下



(4) 动态路由



(5) 路由拦截

```
// center.tsx
import React from 'react';
const Center = () => {
    return (
        <div>
            <h1>center</h1>
        </div>
    );
}

Center.wrappers = ['@/wrappers/auth']
export default Center

//auth.tsx
```

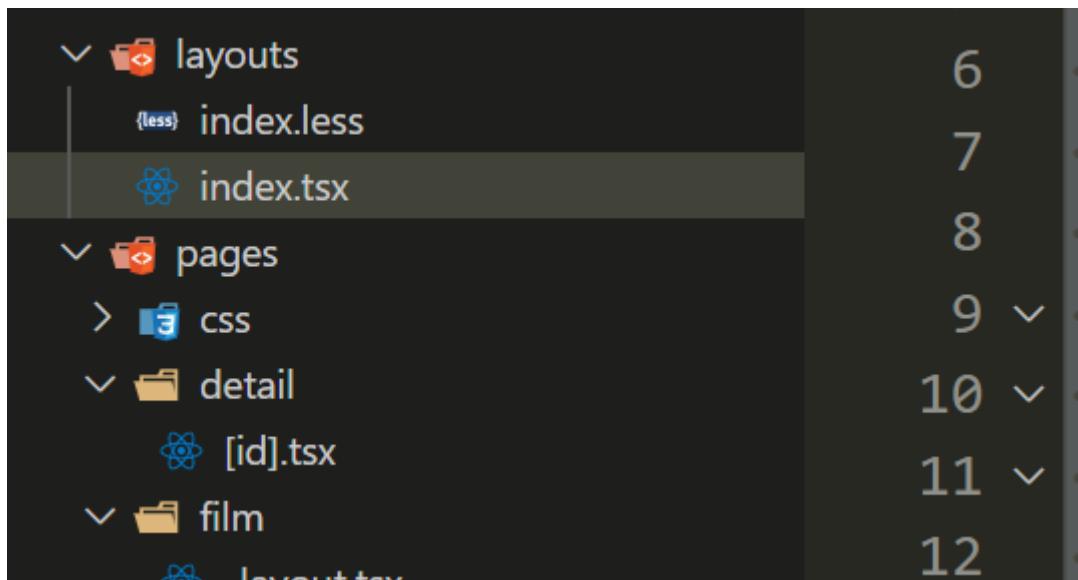
```
import React from 'react'
import { Redirect } from 'umi'
export default (props:any) => {
  const isLogin = localStorage.getItem("token")
  if (isLogin) {
    return <div>{ props.children }</div>;
  } else {
    return <Redirect to="/Login" />;
  }
}
```

(6) hash模式

```
// 在.umirc.js

export default {
  history:{ type: 'hash' }
}
```

(7) 声明式导航



```
import React from 'react';
import {NavLink} from 'umi'
import style from './index.less'
export default function(props:any) {
  if (props.location.pathname.includes('/detail')) {
    return <div>{ props.children }</div>
  }
  return (
    <div>
      <ul>
        <li>
          <NavLink to="/film" activeClassName={style.active}>film</NavLink>
        </li>
        <li>
          <NavLink to="/cinema" activeClassName={style.active}>cinema</NavLink>
        </li>
        <li>
          <NavLink to="/center" activeClassName={style.active}>center</NavLink>
        </li>
      </ul>
    </div>
  )
}
```

```
</ul>
  { props.children }
</div>
);
}
```

(8) 编程式导航

```
import { history } from 'umi';

history.push(`/detail/${item}`)
```

mock功能

umi 里约定 mock 文件夹下的文件或者 page(s) 文件夹下的 _mock 文件即 mock 文件，文件导出接口定义，支持基于 require 动态分析的实时刷新，支持 ES6 语法，以及友好的出错提示

```
// mock/api.js
export default {
  // 支持值为 Object 和 Array
  'GET /api/users': { users: [1, 2] },

  // GET POST 可省略
  '/api/users/1': { id: 1 },

  // 支持自定义函数，API 参考 express@4
  'POST /api/users/create': (req, res) => { res.end('OK'); },
}
```

反向代理

```
// 在.umirc.js
proxy: {
  '/ajax': {
    target: 'https://m.maoyan.com',
    // pathRewrite: { '^/api': '' },
    changeOrigin: true
  }
},
```

antd

```
// .umirc.ts
antd: {
  // 自定义配置
}

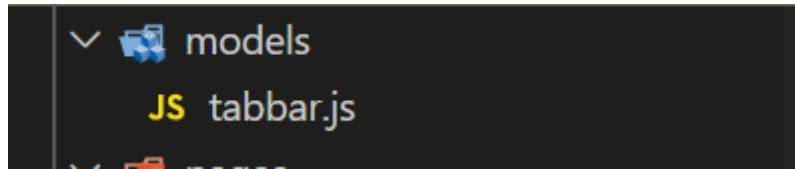
// 组件页面中使用
import { Button } from 'antd-mobile'
<Button type="primary">add</Button>
```

dva集成

- 按目录约定注册 model，无需手动 app.model
- 文件名即 namespace，可以省去 model 导出的 namespace key
- 无需手写 router.js，交给 umi 处理，支持 model 和 component 的按需加载
- 内置 query-string 处理，无需再手动解码和编码
- 内置 dva-loading 和 dva-immer，其中 dva-immer 需通过配置开启(简化 reducer 编写)

```
// .umirc.ts

dva:{
  //自定义配置
}
```



(1) 同步

```
// models/kerwin.js
export default{
  //命名空间
  namespace: 'kerwin',
  state: {
    isShow: true,
    list: []
  },
  //处理state一一同步
  reducers: {
    //reducer简写， type类型是show的时候自动处理
    show(state, {payload}) {
      return {...state, ...payload}
    },
    hide(state, {payload}) {
      return {...state, ...payload}
    }
  },
  // yield表示后面的方法执行完以后 call表示调用一个api接口
  // put表示一个派发
  effects: {
    *showEffect(payload, {put}) {
      yield put({
        type: 'show',
        payload: {
          isShow: true
        }
      })
    },
    *hideEffect(payload, {put}) {
      yield put({
        type: 'hide',
        payload: {
          isShow: false
        }
      })
    }
  }
}
```

```
        }
    }
}
```

```
//根组件
import {connect} from 'dva';

function BasicLayout(props) {

  return (
    <div>
      {
        props.isShow?
          ...
          :null
      }
      {props.children}
    </div>
  );
}

//state.kerwin 命名空间
export default connect(state=>state.kerwin)(BasicLayout);
```

```
//detail.js
import { connect,useDispatch } from 'dva';
function Detail(props) {
  const dispatch = useDispatch()

  useEffect(() => {
    dispatch({
      type:"kerwin/hideEffect" //命名空间kerwin
    })
    return () => {
      dispatch({
        type:"kerwin/showEffect"//命名空间kerwin
      })
    };
  }, [])
  return <div>
    Detail
  </div>
}

export default connect(state=>state.kerwin)(Detail)
```

(2) 异步

`loading`命名空间，是`dva-loading`插件创建的，每次切换路由都会自动控制内部的属性`true or false`

State	Action	State	Diff
Tree	Chart	Raw	
▶ <code>router</code> (<code>pin</code>): { <code>location</code> : [...], <code>action</code> : "POP" }			
<code>@@dva</code> (<code>pin</code>): 0			
▶ <code>kerwin</code> (<code>pin</code>): { <code>isShow</code> : true, <code>list</code> : [...] }			
▶ <code>loading</code> (<code>pin</code>): { <code>global</code> : false, <code>models</code> : {...}, <code>effects</code> : {...} }			

```
// models/kerwin.js
import {getNowplaying} from '../util/getNowplaying'; //封装的fetch调用接口

export default{
  ...
  reducers:{
    ...
    changeList(state,{payload}){
      return {...state,...payload}
    }
  },
  // 异步
  // yield表示后面的方法执行完以后 call表示调用一个api接口
  effects:{
    ...
    *getListEffect(payload,{put,call}){
      let res =yield call(getNowplaying,"test-by-kerwin")
      yield put({
        type:"changeList",
        payload:{
          list:res
        }
      })
    }
  }
}
```

```
// /util/getNowplaying
import {fetch} from 'dva' //dva内置的fetch
export async function getNowplaying(value){
  console.log(value) //value 是call的第二个参数
  var res = await fetch("/ajax/comingList?
ci=65&token=&limit=10&optimus_uuid=43388C403C4911EABDC9998C784A573A4F64A16AA5A34184BADE80
7E506D749E&optimus_risk_level=71&optimus_code=10")
  var result = await res.json()
  return result.coming
}
```

```
// nowplaying.js
import React,{useEffect} from 'react'
import { connect,useDispatch } from 'dva';

function Nowplaying(props) {
  let {list,loading} = props
  let dispatch = useDispatch()
  useEffect(() => {
    if(list.length==0){
      dispatch({
        type:"kerwin/getListEffect" //命名空间kerwin
      })
    }
  }, [list])

  return (
    <div>
      nowplaying--{loading.global?'正在加载数据...':''}
      {
        遍历list
      }
    </div>
  )
}
export default connect(({kerwin,loading})=>({
  ...kerwin,
  loading
})(Nowplaying)
```