

本节内容

课堂目标

redux

资源

起步

redux快速上手

如何更优雅的使用redux呢? react-redux

redux中间件

重构项目

合并reducer

Mobx快速入门

对比react和Mobx

react-router4.0

资源

快速入门

**基本路由使用**

**路由URL参数(二级路由)**

命令式导航

重定向Redirect

路由守卫

集成到redux中

redux原理

React-redux原理

redux-thunk

redux-saga完美方案

redux-thunk和redux-saga的区别

UmijS

特性

快速上手

脚手架

页面中跳转

路由

引入antd

Dvajs

## 本节内容

---

## 课堂目标

---

### redux

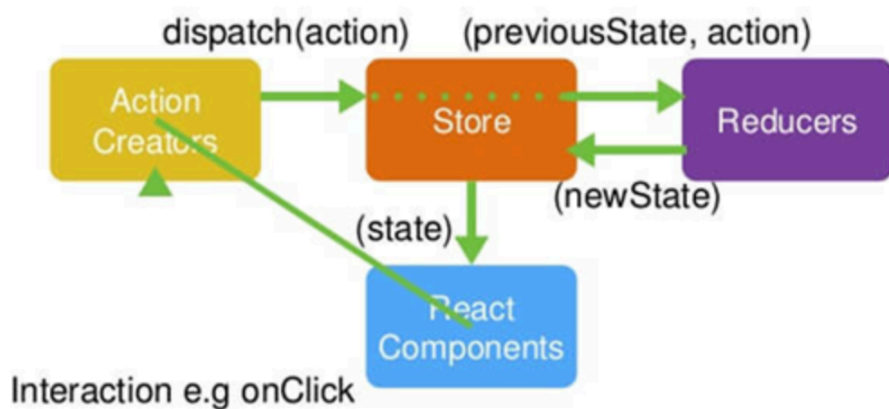
#### 资源

[redux](#)

[react-redux](#)

#### 起步

### Redux Flow



# redux快速上手

## 1.安装

```
1 npm i redux -S
```

## 2.redux中的角色

- Store
  - 维持应用的 state;
  - 提供 `getState()` 方法获取 state;
  - 提供 `dispatch(action)` 方法更新 state;
  - 通过 `subscribe(listener)` 注册监听器;
  - 通过 `subscribe(listener)` 返回的函数注销监听器。
- Reducer: 指定了应用状态的变化如何响应 [actions](#) 并发送到 store 的
- Action: 把数据从应用传到store的有效载荷

store.js

```
1 import {  
2   createStore  
3 } from 'redux';  
4 // 创建reducer 状态修改具体执行者  
5 function counter(state = 0, action) {  
6   switch (action.type) {  
7     case 'INCREMENT':  
8       return state + 1;  
9     case 'DRCREMENT':  
10      return state - 1;  
11     default:  
12      return state;  
13   }
```

```
14 }  
15 //创建store并导出  
16 export default createStore(counter);
```

## ReduxTest.js

```
1 import React, { Component } from 'react';  
2 import store from '../store';  
3  
4 class ReduxTest extends Component {  
5   render() {  
6     return (  
7       <div>  
8         <p>  
9           {store.getState()}  
10        </p>  
11        <button onClick={() => store.dispatch({  
type: "DRCREMENT"})}>-1</button>  
12        <button onClick={() => store.dispatch({  
type: "INCREMENT" })}>+1</button>  
13      </div>  
14    );  
15  }  
16 }  
17  
18 export default ReduxTest;
```

## index.js

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import ReduxTest from '../components/ReduxTest';
4 import store from '../store'
5 function render() {
6     ReactDOM.render(<ReduxTest />,
7     document.querySelector('#root'));
8 }
9 render();
10 // 每次 state 更新时, 打印日志
11 // 注意 subscribe() 返回一个函数用来注销监听器
12 // 订阅
13 store.subscribe(render)
```

Redux架构的设计核心：**严格的单向数据流**

问题:每次state更新, 都会重新render, 大型应用中会造成不必要的重复渲染。

## 如何更优雅的使用redux呢? react-redux

[react-redux](#)

```
1 npm i react-redux -S
```

具体步骤:

- React Redux提供了 `<Provider />`, 使得Redux store都应用到你的应用程序

修改index.js

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import ReduxTest from '../components/ReduxTest';
```

```

4 import store from './store'
5 import { Provider } from 'react-redux';
6 function render() {
7     ReactDOM.render((
8         <Provider store = {store}>
9             <ReduxTest />
10        </Provider>
11    ), document.querySelector('#root'));
12 }
13 render();
14 //订阅不需要了
15 // store.subscribe(render);

```

React Redux提供了 `connect` 将组件连接到store的功能

## 修改ReduxTest.js

```

1 import React, { Component } from 'react';
2 import { connect } from "react-redux";
3 const mapStateToProps = state => {
4     return {
5         num: state
6     }
7 }
8 const mapDispatchToProps = dispatch => {
9     return {
10         increment: () => {
11             dispatch({ type: 'INCREMENT' })
12         },
13         decrement: () => {
14             dispatch({
15                 type: 'DRCREMENT'
16             })
17         }
18     }

```

```

19 }
20 class ReduxTest extends Component {
21   render() {
22     return (
23       <div>
24         <p>{this.props.num}</p>
25         <button onClick={() =>
26 this.props.decrement()}>-1</button>
27         <button onClick={() =>
28 this.props.increment()}>+1</button>
29       </div>
30     );
31   }
32 }
33
34 export default connect(mapStateToProps,
  mapDispatchToProps)(ReduxTest);

```

## 装饰器写法

```

1 import React, { Component } from 'react';
2 import { connect } from "react-redux";
3 const mapStateToProps = state => {
4   return {
5     num: state
6   }
7 }
8 const mapDispatchToProps = dispatch => {
9   return {
10     increment: () => {
11       dispatch({ type: 'INCREMENT' })
12     },
13     decrement: () => {
14       dispatch({

```

```

15         type: 'DRCREMENT'
16     })
17 }
18 }
19 }
20 @connect(mapStateToProps, mapDispatchToProps)
21 class ReduxTest extends Component {
22     render() {
23         return (
24             <div>
25                 <p>{this.props.num}</p>
26                 <button onClick={() =>
27 this.props.decrement()}>-1</button>
28                 <button onClick={() =>
29 this.props.increment()}>+1</button>
30             </div>
31         );
32     }
33 }
34 export default ReduxTest;

```

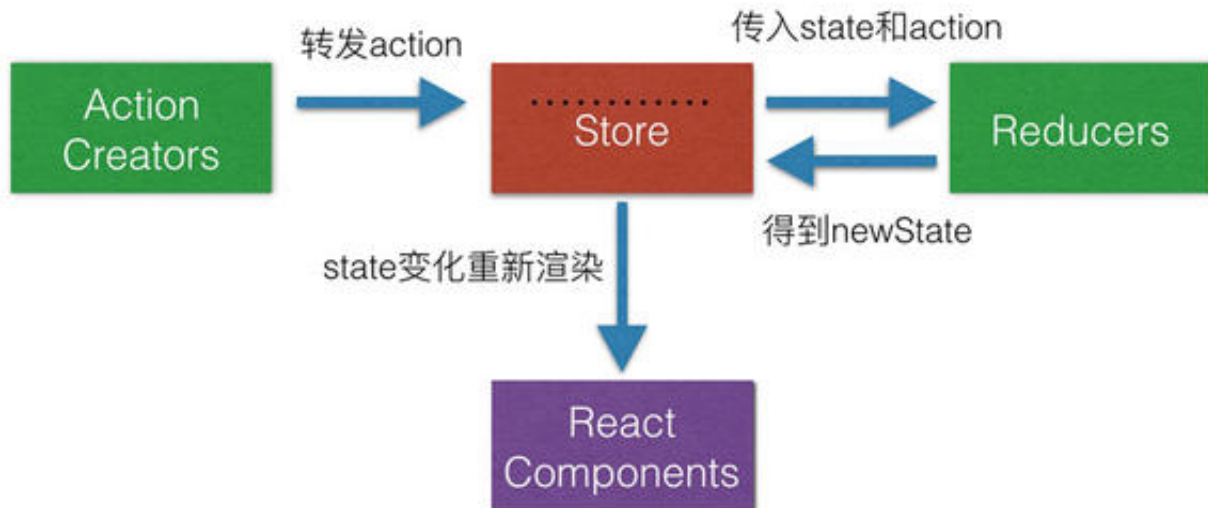
容器组件就是使用 `store.subscribe()` 从 Redux state 树中读取部分数据，并通过 props 来把这些数据提供给要渲染的组件。你可以手动来开发容器组件，但建议使用 **React Redux** 库的 `connect()` 方法来生成，这个方法做了性能优化来避免很多不必要的重复渲染。

使用 `connect()` 前，需要先定义 `mapStateToProps` 这个函数来指定如何把当前的 Redux store state 映射到展示组件的 props 中。

## redux中间件

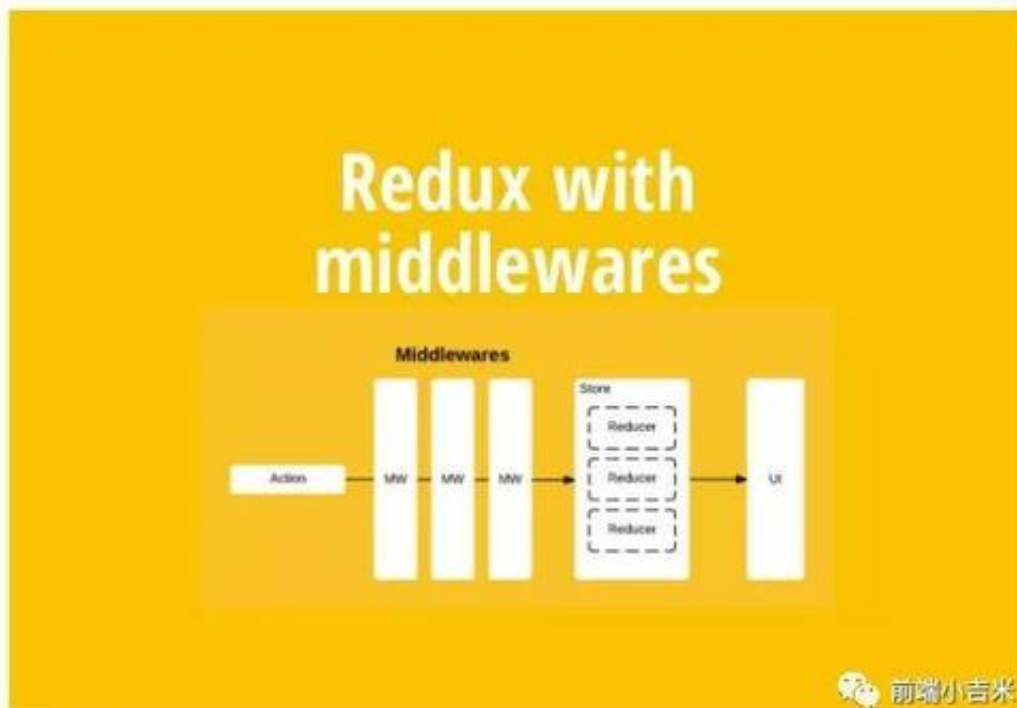


## Redux工作流



利用redux中间件机制可以在实际action响应前执行其它额外的业务逻辑。

特点：自由组合，自由插拔的插件机制



通常我们没有必要自己写中间件，介绍两款比较成熟的中间件

- redux-logger:处理日志记录的中间件

- Redux-thunk:处理异步action

```
1 | npm i redux-thunk redux-logger -S
```

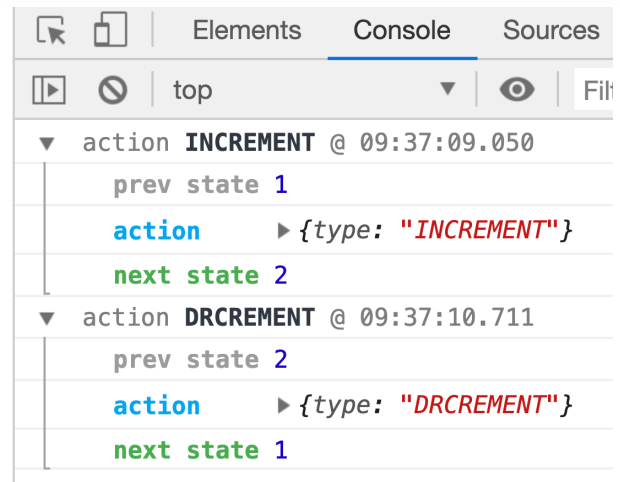
redux-logger的使用在store.js加入

```
1 | import {  
2 |     createStore,  
3 |     applyMiddleware  
4 | } from 'redux';  
5 | import logger from 'redux-logger';  
6 | // 创建reducer  
7 | function counter(state = 0, action) {  
8 |     switch (action.type) {  
9 |         case 'INCREMENT':  
10 |             return state + 1;  
11 |         case 'DRCREMENT':  
12 |             return state - 1;  
13 |         default:  
14 |             return state;  
15 |     }  
16 | }  
17 |  
18 | export default createStore(counter,  
    applyMiddleware(logger));
```

效果:

1

-1 +1



redux-thunk 在store.js修改

```
1 import {
2     createStore,
3     applyMiddleware
4 } from 'redux';
5 import logger from 'redux-logger';
6 import thunk from 'redux-thunk';
7 // 创建reducer
8 function counter(state = 0, action) {
9     switch (action.type) {
10         case 'INCREMENT':
11             return state + 1;
12         case 'DRCREMENT':
13             return state - 1;
14         default:
15             return state;
16     }
17 }
18 export default createStore(counter,
    applyMiddleware(logger,thunk));
```

添加thunk的作用:action默认接收一个对象，执行下个任务，如果是个函数，则需要异步处理。

redux-thunk 在ReduxTest.js修改

```
1 import React, { Component } from 'react';
2 // import store from '../store';
3 import { connect } from "react-redux";
4 const mapStateToProps = state => {
5     return {
6         num: state
7     }
8 }
9 const asyncAdd = () => {
10     return (dispatch,getState)=>{
11         setTimeout(() => {
12             dispatch({type:'INCREMENT'})
13         }, 1000);
14     }
15 }
16 const mapDispatchToProps = (dispatch) => {
17     return {
18         increment: () => {
19             dispatch({ type: 'INCREMENT' });
20         },
21         decrement: () => {
22             dispatch({
23                 type: 'DRCREMENT'
24             })
25         },
26         asyncIncrement: () => {
27             //action的动作默认是对象，如果是返回函数则使用
28             //redux-thunk处理
29             dispatch(asyncAdd());
30         }
31     }
32 }
33 @connect(mapStateToProps, mapDispatchToProps)
```

```

34 class ReduxTest extends Component {
35   render() {
36     return (
37       <div>
38         <p>{this.props.num}</p>
39         <button onClick={() =>
this.props.decrement()}>-1</button>
40         <button onClick={() =>
this.props.increment()}>+1</button>
41         <button onClick={() =>
this.props.asyncIncrement()}>async+1</button>
42       </div>
43     );
44   }
45 }
46
47 export default ReduxTest;

```

效果展示:

1

-1 +1 **async+1**

The screenshot shows the Redux DevTools interface with the Console tab selected. The state is initially 0. An action is dispatched with type 'INCREMENT', which is handled by a setTimeout function that dispatches another 'INCREMENT' action after 1000ms. The state then becomes 1.

Time	Action	Prev State	Next State
10:43:35.533	action undefined	0	0
10:43:36.534	action INCREMENT	0	1

重构项目

## 新建store/couter.reduce.js

```
1 // 创建reducer
2 const counter = (state = 0, action) => {
3   switch (action.type) {
4     case 'INCREMENT':
5       return state + 1;
6     case 'DRCREMENT':
7       return state - 1;
8     default:
9       return state;
10  }
11 }
12 export const mapStateToProps = state => {
13   return {
14     num: state
15   }
16 }
17 const asyncAdd = () => {
18   return (dispatch, getState) => {
19     setTimeout(() => {
20       dispatch({ type: 'INCREMENT' })
21     }, 1000);
22   }
23 }
24 export const mapDispatchToProps = (dispatch) => {
25   return {
26     increment: () => {
27       // dispatch({ type: 'INCREMENT' })
28       dispatch({ type: 'INCREMENT' });
29     },
30     decrement: () => {
31       dispatch({
32         type: 'DRCREMENT'
```

```

33         })
34     },
35     asyncIncrement: () => {
36         dispatch(asyncAdd());
37     }
38 }
39 }
40 }
41
42 export default counter;

```

### 新建store/index.js

```

1  import {
2      createStore,
3      applyMiddleware
4  } from 'redux';
5  import logger from 'redux-logger';
6  import thunk from 'redux-thunk';
7  import counter from './counter.reducer';
8
9  export default createStore(counter,
    applyMiddleware(logger,thunk));

```

### 重构ReduxTest.js

```

1  import React, { Component } from 'react';
2  // import store from '../store';
3  import { connect } from "react-redux";
4  import {mapStateToProps,mapDispatchToProps} from
    '../store/couter.reducer';
5
6  @connect(mapStateToProps, mapDispatchToProps)
7  class ReduxTest extends Component {

```

```
8     render() {
9         return (
10             <div>
11                 <p>{this.props.num}</p>
12                 <button onClick={() =>
this.props.decrement()}>-1</button>
13                 <button onClick={() =>
this.props.increment()}>+1</button>
14                 <button onClick={() =>
this.props.asyncIncrement()}>async+1</button>
15             </div>
16         );
17     }
18 }
19
20 export default ReduxTest;
```

## 合并reducer

使用 `combineReducers` 进行复合，实现状态的模块化



```

1 import {
2     createStore,
3     applyMiddleware,
4     combineReducers
5 } from 'redux';
6 import logger from 'redux-logger';
7 import thunk from 'redux-thunk';
8 import counter from './counter.reducer';
9
10 export default createStore(
11     combineReducers({
12         counter
13     }), applyMiddleware(logger,thunk));

```

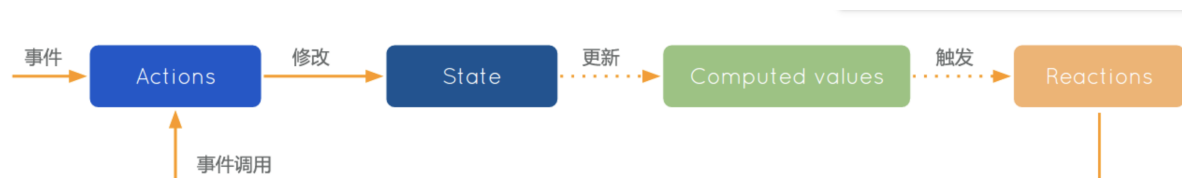
counter.reducer.js

```

1 export const mapStateToProps = state => {
2     return {
3         //加上当前状态的key,来进行标识
4         num: state.counter
5     }
6 }

```

## Mobx快速入门



事件调用 actions。  
Actions 是唯一可以修改  
state 的东西并且可能有其它  
副作用。

State 是可观察和最低限度定义的。  
不应包含冗余或推导数据。  
可以是图形，包含类、数组、引用、  
等等。

Computed values 是可以使用 pure  
function 从 state 中推导出的值。  
MobX 会自动更新它并在它不再使用时  
将其优化掉。

Reactions 很像 Computed values，  
会对 state 的变化做出反应。但它们不  
产生一个值，而是会产生一些副作用，  
像更新UI。

React 和 MobX 是一对强力组合。

React是一个消费者，将应用状态state渲染成组件树对其渲染。

Mobx是一个提供者，用于存储和更新状态state

下载

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

新建store/mobx.js

```
1 | import { observable, action, computed } from "mobx";
2 |
3 | // 观察者
4 | const appState = observable({
5 |   num: 0
6 | })
7 | // 方法
8 | appState.increment = action(()=>{
9 |   appState.num+=1;
10 | })
11 | appState.decrement = action(()=>{
12 |   appState.num-=1;
13 | })
14 | export default appState;
```

index.js

```

1 import React from 'react';
2 import ReactDOM from 'react-dom';
3 import {Provider} from 'react-redux';
4 import MobxTest from "../components/MobxTest";
5 ReactDOM.render((
6     <div>
7         <MobxTest appState = {appState}/>
8     </div>
9 ), document.querySelector('#root'));

```

## MobxTest.js

```

1 import React, { Component } from 'react';
2 import { observer } from "mobx-react";
3 class MobxTest extends Component {
4     render() {
5         return (
6             <div>
7                 {this.props.appState.num}
8                 <button onClick={() =>
9 this.props.appState.decrement()}>-1</button>
10                 <button onClick={() =>
11 this.props.appState.increment()}>+1</button>
12             </div>
13         );
14     }
15 }
16 export default observer(MobxTest);

```

装饰器写法:

store/mobx.decorator.js

```

1 import { observable, action, computed } from "mobx";
2 // 常量改成类
3 class AppState {
4     @observable num = 0;
5     @action
6     increment(){
7         this.num +=1;
8     }
9     @action
10    decrement() {
11        this.num -= 1;
12    }
13 }
14 const appState = new AppState();
15
16 export default appState;

```

## MobxTest.decorator.js

```

1 import React, { Component } from 'react';
2 import { observer } from "mobx-react";
3 @observer
4 class MobxTest extends Component {
5     render() {
6         return (
7             <div>
8                 {this.props.appState.num}
9                 <button onClick={() =>
10 this.props.appState.decrement()}>-1</button>
11                 <button onClick={() =>
12 this.props.appState.increment()}>+1</button>
13             </div>
14         );
15     }
16 }

```

```
14 }  
15  
16 export default MobxTest;
```

## 对比react和Mobx

- 学习难度
- 工作量
- 内存开销
- 状态管理的集中性
- 样板代码的必要性
- 结论：使用Mobx入门简单，构建应用迅速，但是当项目足够大的时候，还是redux,爱不释手，那还是开启严格模式，再加上一套状态管理的规范。爽的一p

## react-router4.0

### 资源

[react-router](#)

### 快速入门

#### 安装

```
1 npm install react-router-dom --save
```

### 基本路由使用

```
1 import React, { Component } from 'react';  
2 import { BrowserRouter as Router, Route, Link } from  
  "react-router-dom";  
3 function Home() {
```

```
4     return (
5         <h2>我是首页</h2>
6     )
7 }
8 function Course() {
9     return (
10         <h2>我是课程</h2>
11     )
12 }
13 function User() {
14     return (
15         <h2>我是首页</h2>
16     )
17 }
18 class Basic_router extends Component {
19     render() {
20         return (
21             <Router>
22                 <div>
23                     { /* 定义路由页面 */ }
24                     <ul>
25                         <li>
26                             <Link to="/">首页</Link>
27                         </li>
28                         <li>
29                             <Link to="/course">课
30 程</Link>
31                         </li>
32                         <li>
33                             <Link to="/user">用
34 户</Link>
35                         </li>
36                     </ul>
37                 </div>
38                 { /* 配置路由 */ }
```

```

37             /* 为什么要加exact 这是因为包含式
匹配,加上exact之后,表示确切匹配 */
38             <Route exact path='/' component=
{Home}></Route>
39             <Route path='/course' component=
{Course}></Route>
40             <Route path='/user' component=
{User}></Route>
41             </div>
42
43         </Router>
44     );
45 }
46 }
47
48 export default Basic_router;

```

## 路由URL参数(二级路由)

```

1  import React, { Component } from 'react';
2  import { BrowserRouter as Router, Route, Link } from
"react-router-dom";
3  function Home() {
4      return (
5          <h2>我是首页</h2>
6      )
7  }
8  function Course() {
9      return (
10         <div className='course'>
11             <h2>我的课程</h2>
12             /*定义二级路由页面*/
13             <ul>
14                 <li>

```

```
15         <Link to='/course/vue'>Vue</Link>
16     </li>
17     <li>
18         <Link
19 to='/course/React'>React</Link>
20     </li>
21     <li>
22         <Link
23 to='/course/Angular'>Angular</Link>
24     </li>
25 </ul>
26 { /*配置路由参数*/ }
27 <Route path='/course/:id' component=
28 {CourseChild}></Route>
29 <Route exact path={match.path} render={()
=> <h3>请选择你的课程</h3>} />
30 </div>
31 )
32 }
33 function User() {
34     return (
35         <h2>我是首页</h2>
36     )
37 }
38 //二级路由页面显示
39 function CourseChild({match,history,location}) {
40     //match: 匹配路由信息对象
41     //location: 本地信息对象
42     //history: 历史信息对象
43     console.log(location,match,history);
44
45     return (
46         <div>
47             {match.params.id}
```



```

46         </div>
47     )
48 }
49 class Basic_router extends Component {
50     render() {
51         return (
52             <Router>
53                 <div>
54                     { /* 定义路由页面 */ }
55                     <ul>
56                         <li>
57                             <Link to="/">首页</Link>
58                         </li>
59                         <li>
60                             <Link to="/course">课
程</Link>
61                         </li>
62                         <li>
63                             <Link to="/user">用
户</Link>
64                         </li>
65                     </ul>
66                     { /* 配置路由 */ }
67                     <Route exact path="/" component=
{Home}></Route>
68                     <Route path="/course" component=
{Course}></Route>
69                     <Route path="/user" component=
{User}></Route>
70
71                 </div>
72
73             </Router>
74         );
75     }

```

```
76 }  
77  
78 export default Basic_router;
```

上述的Course组件也可以这样修改

```
1 function Course({match}) {  
2     return (  
3         <div className='course'>  
4             <h2>我的课程</h2>  
5             <ul>  
6                 <li>  
7                     <Link to=  
8 {`${match.url}/vue`} >Vue</Link>  
9                     </li>  
10                    <li>  
11                        <Link to=  
12 {`${match.url}/react`} >React</Link>  
13                        </li>  
14                    <li>  
15                        <Link to=  
16 {`${match.url}/angular`} >Angular</Link>  
17                        </li>  
18                    </ul>  
19                    <Route path='/course/:id' component=  
20 {CourseChild}></Route>  
21                    <Route exact path={match.path} render={()  
=> <h3>请选择你的课程</h3>} />  
                </div>  
            )  
        }  
    }
```

不匹配(404)

```
1 // 404页面展示
2 function NoMatch() {
3     return <div>404页面,网页找不到了</div>
4 }
5 class Basic_router extends Component {
6     render() {
7         return (
8             <Router>
9                 <div>
10                     { /* 定义路由页面 */ }
11                     <ul>
12                         <li>
13                             <Link to="/">首页</Link>
14                         </li>
15                         <li>
16                             <Link to="/course">课
17                             程</Link>
18                         </li>
19                         <li>
20                             <Link to="/user">用
21                             户</Link>
22                         </li>
23                     </ul>
24                     { /* 配置路由 */ }
25                     <Route exact path="/" component=
26                     {Home}></Route>
27                     <Route path="/course" component=
28                     {Course}></Route>
29                     <Route path="/user" component=
30                     {User}></Route>
31                     { /* 添加不匹配路由配置 */ }
32                     <Route component={NoMatch}>
33                     </Route>
34                 </div>
35             </Router>
36         )
37     }
38 }
```

```

29
30         </Router>
31     );
32 }
33 }

```

此时会发现，每个页面都会匹配NoMatch组件，这时候该是Switch组件出厂了

修改以上代码如下

```

1  import React, { Component } from 'react';
2  import { BrowserRouter as Router, Route, Link, Switch }
   from "react-router-dom";
3  // 404页面展示
4  function NoMatch() {
5      return <div>404页面,网页找不到了</div>
6  }
7  class Basic_router extends Component {
8      render() {
9          return (
10             <Router>
11                 <div>
12                     { /* 定义路由页面 */ }
13                     <ul>
14                         <li>
15                             <Link to="/">首页</Link>
16                         </li>
17                         <li>
18                             <Link to="/course">课
19 程</Link>
20                         </li>
21                         <li>
22                             <Link to="/user">用
23 户</Link>

```

```

22         </li>
23     </ul>
24     { /* 配置路由 */ }
25     <Switch>
26         <Route exact path="/"
component={Home}></Route>
27         <Route path="/course"
component={Course}></Route>
28         <Route path="/user" component=
{User}></Route>
29         { /* 添加不匹配路由配置 */ }
30         <Route component={NoMatch}>
</Route>
31     </Switch>
32 </div>
33
34 </Router>
35 );
36 }
37 }

```

## 命令式导航

```

1 function Home({ location }) {
2     console.log(location);
3     return (
4         <div>
5             <h1>{location.state ? location.state.foo :
""}</h1>
6             <h2>我是首页</h2>
7         </div>
8     )
9 }
10 function CourseChild({ match, history, location }) {

```

```

11     return (
12         <div>
13             {match.params.id}课程
14             <button onClick={history.goBack}>返回
15         </button>
16             <button onClick={() => { history.push('/')
17 }}>跳转首页</button>
18             <button onClick={()=>{
19                 history.push({
20                     pathname: '/',
21                     state:{
22                         foo: 'bar'
23                     }
24                 })
25             }}>跳转首页,并携带值</button>
26         </div>
27     )
28 }

```

## 重定向Redirect

```

1  import React, { Component } from 'react';
2  import { BrowserRouter as Router, Route, Link, Switch,
3  Redirect} from "react-router-dom";
4
5  function UserDeatil({match,location}) {
6      return (
7          <div>个人详情页面</div>
8      )
9  }
10
11 function UserOrder(params) {
12     return (
13         <div>用户订单页面</div>
14     )
15 }

```

```

13 }
14 function User() {
15     return (
16         <div>
17             <h2>
18                 <Link to='/user/detail'>个人信息</Link>
19             </h2>
20             <h2>
21                 <Link to='/user/order'>个人订单</Link>
22             </h2>
23             <Switch>
24                 <Route path="/user/detail" component=
25 {UserDeatil}></Route>
26                 <Route path="/user/order" component=
27 {UserOrder}></Route>
28                 { /*重定向*/ }
29                 <Redirect to='/user/detail'></Redirect>
30             </Switch>
31         </div>
32     )
33 }
34 class Basic_router extends Component {
35     render() {
36         return (
37             <Router>
38                 <div>
39                     { /* 定义路由页面 */ }
40                     <ul>
41                         <li>
42                             <Link to='/user'>用
43                             户</Link>
44                         </li>
45                     </ul>

```

```

45         {/* 配置路由 */}
46         <Switch>
47             <Route path='/user' component=
{User}></Route>
48         </Switch>
49     </div>
50
51 </Router>
52 );
53 }
54 }
55
56 export default Basic_router;

```

## 路由守卫

定义可以验证的高阶组件

```

1  // 路由守卫：定义可以验证的高阶组件
2  function PrivateRoute({ component: Component, ...rest
}) {
3      return (
4          <Route
5              {...rest}
6              render={props =>
7                  Auth.isAuthenticated ? (
8                      <Component {...props} />
9                  ) : (
10                     <Redirect to={{
11                         pathname: "/login",
12                         state: { from:
props.location }
13                     }}
14                     />

```



```

15         )
16     }
17     />
18 )
19 }

```

## 认证类Auth

```

1  const Auth = {
2    isAuth: false,
3    login(cb) {
4      this.isAuth = true;
5      setTimeout(cb, 1000);
6    },
7    signout(cb) {
8      this.isAuth = false;
9      setTimeout(cb, 1000);
10   }
11 }

```

## 定义登录组件

```

1  class Login extends Component {
2    state = { isLogin: false };
3    handlerlogin = () => {
4      Auth.login(() => {
5        this.setState({
6          isLogin:true
7        })
8      })
9    }
10   render() {
11     let { isLogin } = this.state;

```

```

12         let { from } = this.props.location.state || {
13           from: { pathname: '/' } }
14         if (isLogin) return <Redirect to={from} />
15         return (
16           <div>
17             <p>请先登录</p>
18             <button onClick={this.handlerlogin}>登
19             录</button>
20           </div>
21         );
22       }
23     }

```

主路由组件中使用自定义路由和定义登录路由配置

```

1 <Switch>
2   <Route exact path="/" component={Home}></Route>
3   { /* <Route path="/course" component={Course}>
4     <PrivateRoute path="/course" component={Course}>
5     </PrivateRoute>
6     <Route path="/user" component={User}></Route>
7     <Route path="/login" component={Login}></Route>
8     <Route component={NoMatch}></Route>
9   }
10 </Switch>

```

## 集成到redux中

### 1. 新建/store/user.reducer.js

```

1 const initState = {
2   isLogin: false, //表示用户未登录
3   userInfo: {}
4 }

```

```
5 function user(state = initState, action) {
6   switch (action.type) {
7     case 'login':
8       return { isLogin: true }
9
10    default:
11      return initState
12  }
13
14 }
15
16 export const mapStateToProps = state => {
17   return {
18     // 加上当前状态的key,来进行模块化的标识
19     user: state.user
20   }
21 }
22
23 const login = () => {
24   return (dispatch) => {
25     setTimeout(() => {
26       dispatch({ type: 'login' })
27     }, 1000);
28   }
29 }
30 export const mapDispatchToProps = dispatch => {
31   return {
32     //action 默认接收一个对象, 执行下个任务, 如果是一个
    //函数, 则需要异步处理, react-thunk
33     login: () => {
34       dispatch(login())
35     }
36   }
37 }
38 export default user
```

## 新建store/index.js

```
1
2 // combineReducers 进行复合，实现状态的模块化
3 import { createStore, applyMiddleware,
  combineReducers } from "redux";
4 import logger from "redux-logger";
5 import thunk from "redux-thunk";
6 import user from './user.reducer'
7
8 // 创建store 有state和reducer的store
9 const store = createStore(combineReducers({
10   user
11 })), applyMiddleware(logger, thunk));
12 export default store;
```

## 在index.js

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

## App.js修改

```

1  ....
2  import { connect } from "react-redux";
3  import { mapStateToProps } from
   "./store/user.reducer";
4  // 高阶组件：定义验证功能的路由组件
5  @connect(mapStateToProps)
6  class PrivateRoute extends Component {
7    render() {
8      const Comp = this.props.component;
9      return (
10         <Route
11           {...this.props}
12           component={
13             (props) =>
14               this.props.user.isLogin ?
15                 (<Comp {...props} />) :
16                 (<Redirect to={{ pathname: '/login',
state: { from: props.location } }} />)
17             >>
18         </Route>
19       )
20     }
21   }
22  ....

```

## login.js组件修改

```

1  import React, { Component } from 'react'
2  import Auth from '../utils/auth';
3  import { Button } from "antd";
4  import { Redirect } from "react-router-dom";
5  import { connect } from "react-redux";
6  import { mapStateToProps, mapDispatchToProps } from
   '../store/user.reducer';

```

```

7  @connect(mapStateToProps, mapDispatchToProps)
8  class Login extends Component {
9    handleLogin = () => {
10      // 异步处理
11      this.props.login();
12    }
13
14    render() {
15      let { isLogin } = this.props.user;
16      let path =
17        this.props.location.state.from.pathname
18        if (isLogin) {
19          return <Redirect to={path}/>
20        } else {
21          return (
22            <div>
23              <p>请先登录</p>
24              <Button onClick={this.handleLogin}>登
25              录</Button>
26            </div>
27          )
28        }
29    }
30    export default Login

```

## redux原理

- `createStore` 是一个函数，接收三个参数 `reducer, preloadedState, enhancer`
  - `enhancer` 是一个高阶函数，用于增强create出来的store，他

的参数是 `createStore`，返回一个更强大的store生成函数。  
(功能类似于middleware)。

- 我们mobile仓库中的 `storeCreator` 其实就可以看成是一个 enhancer，在 `createStore` 的时候将saga揉入了进去只不过不是作为 `createStore` 的第三个参数完成，而是使用 `middleware` 完成。

```
1 export default function
  createStore(reducer,preloadedState,enhancer) {
2
3     if (typeof preloadedState === 'function' && typeof
enhancer === 'function' || typeof enhancer ===
'function' && typeof arguments[3] === 'function') {
4         throw new Error('It looks like you are passing
several store enhancers to ' + 'createStore(). This is
not supported. Instead, compose them ' + 'together to a
single function.');
```

5 }

6 //如果传递了第二个参数preloadedState，而且第二个参数不  
是一个function，则将preloadedState 保存在内部变量  
currentState中，也就是我们给State 的默认状态

```
7     if (typeof preloadedState === 'function' && typeof
enhancer === 'undefined') {
8         enhancer = preloadedState;
9         preloadedState = undefined;
10    }
11
12    if (typeof enhancer !== 'undefined') {
13        if (typeof enhancer !== 'function') {
14            throw new Error('Expected the enhancer to
be a function.');
```

15 }

```
16      // createStore 作为enhancer的参数, 返回一个被加强
    的createStore, 然后再将reducer, preloadedState传进去生成
    store
17      return enhancer(createStore)(reducer,
preloadedState);
18  }
19  //第一个参数reducer 是必须要传递的而且必须是一个函数,
    不然Redux会报错
20      if (typeof reducer !== 'function') {
21          throw new Error('Expected the reducer to be a
function.');
```

22 }

23 //仓库内部保存了一颗状态树。可以是任意类型

24 let currentState = preloadedState;

25 let currentListeners=[];

26 let currentReducer = reducer

27 function getState() {

28 return JSON.parse(JSON.stringify(state));

29 }

30 //组件可以派发动作给仓库

31 function dispatch(action) {

32 //调用reducer进行处理, 获取老的state,计算出新的
state

33

34 currentState=currentReducer(currentState,action);

35 //通知其他的组件执行

36 currentListeners.forEach(l=>l());

37 }

38 //如果说其他的组件需要订阅状态变化时间的话,

39 function subscribe(listener) {

40 //将监听函数放入一个队列中

41 currentListeners.push(listener);

42 return function () {

43 currentListeners =

44 currentListeners.filter(item=>item!==listener);



```

43     }
44   }
45   //初始化的操作
46   dispatch({type: '@@INIT'});
47   return {
48     getState,
49     dispatch,
50     subscribe
51   }
52 }

```

- applyMiddleware与enhancer关系

- 首先他们两个的功能一样，都是为了增强store
- applyMiddleware的结果，其实一个enhancer

```

1  export function applyMiddleware(...middlewares){
2    return (createStore) => {
3      return function (...args) {
4        //创建原始的store
5        const store = createStore(...args);
6        //获取原始的dispatch
7        const _dispatch = store.dispatch;
8        const middlewareAPI = {
9          getState: store.getState,
10         dispatch: (...args)=> {
11           return dispatch(...args)
12         }
13       };
14       //调用第一层中间件
15       const middlewareChain = middlewares.map(
16         (middleware)=> {
17           //让每个中间件执行，传入一个对象
18           {getState,dispatch}
19           return middleware(middlewareAPI);

```

```

18         });
19         //通过compose复合函数，先将当前中间件的事情做完，然后继续调用下一个中间件，并且将值（store.dispatch）传入，增强dispatch
20         _dispatch = compose(...middlewareChain)
        (store.dispatch);
21         // return 一个被增强了dispatch的store
22         return {
23             ...store,
24             dispatch: _dispatch
25         };
26     };
27 };
28 }
29 function compose(...fns){ //[add1,add2,add3] 都是函数
30     if(fns.length === 0){
31         return arg => arg;
32     }
33     if(fns.length === 1){
34         return fns[0]
35     }
36     return fns.reduce((f1,f2)=>(...args)=>
        f1(f2(...args)))
37 }

```

## React-redux原理

```

1 import React,{Component} from 'react';
2 import {bindActionCreators} from '../redux';
3 /**
4  * connect实现的是仓库和组件的连接
5  * mapStateToProps 是一个函数 把状态映射为一个属性对象
6  * mapDispatchToProps 也是一个函数 把dispatch方法映射为一个属性对象

```

```

7  */
8  export default function
   connect(mapStateToProps, mapDispatchToProps) {
9      return function (Com) {
10         //在这个组件里实现仓库和组件的连接
11         class Proxy extends Component{
12
13             state=mapStateToProps(this.props.store.getState())
14             componentDidMount() {
15                 //更新状态
16                 this.unsubscribe =
17                 this.props.store.subscribe(() => {
18
19                     this.setState(mapStateToProps(this.props.store.getState
20                     (())));
21                 });
22             }
23             componentWillUnmount = () => {
24                 this.unsubscribe();
25             }
26             render() {
27                 let actions={};
28                 //如果说mapDispatchToProps是一个函数，执行后
29                 //得到属性对象
30                 if (typeof mapDispatchToProps ===
31                 'function') {
32                     actions =
33                     mapDispatchToProps(this.props.store.dispatch);
34                 }
35                 //如果说mapDispatchToProps是一个对象的
36                 //话，我们需要手工绑定
37                 } else {
38                     actions=bindActionCreators(mapDispatchToProps,this.prop
39                     s.store.dispatch);
40                 }

```

```

31         return <Com {...this.state} {...actions}/>
32     }
33 }
34 }
35 export default class Provider extends Component{
36     //规定如果有人想使用这个组件，必须提供一个redux仓库属性
37     static propTypes={
38         store:PropTypes.object.isRequired
39     }
40     render() {
41         let value={store:this.props.store};
42         return (
43             <StoreProvider value={value}>
44                 {this.props.children}
45             </StoreProvider>
46         )
47     }
48 }

```

## redux-thunk

```

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完美方案

`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。

`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。

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。

不同于 `redux thunk`，你不会再遇到回调地狱了，你可以很容易地测试异步流程并保持你的 action 是干净的。

## 安装

```
1 npm install redux-saga --save
```

新建 `store/sagas.js`

```
1 import { call, put, takeEvery } from "redux-  
  saga/effects";  
2  
3 // 模拟登录的api 一般项目开发中会将此api放入service文件夹  
  下  
4 const api = {  
5   login(){  
6     return new Promise((resolve, reject) => {  
7       setTimeout(() => {  
8         if(Math.random() > 0.5){  
9           resolve({id:1,name:"Tom"})  
10          }else{  
11            reject('用户名或密码错误')  
12          }  
13        }, 1000);  
14      })  
15    }  
16  }  
17 }
```

```

15
16     }
17 }
18
19 // worker saga :将login action被dispatch时调用
20 function* login(action) {
21     try {
22         const result = yield call(api.login);
23         yield put({ type: 'login', result });
24     } catch (error) {
25         yield put({ type: 'loginError', message:
error.message });
26     }
27 }
28
29 // 类似监听器
30 function* mySaga() {
31     yield takeEvery('login_request',login);
32 }
33 export default mySaga;

```

为了跑起Saga,我们需要使用 `redux-saga` 中间件将Saga与Redux Store建立连接。

修改store/index.js

```

1 import {
2     createStore,
3     applyMiddleware,
4     combineReducers
5 } from 'redux';
6 import logger from 'redux-logger';
7 // 注册reducer
8 import user from './user.reducer';
9 import createSagaMiddleware from 'redux-saga'

```

```

10 import mySaga from './sagas';
11
12 // 1.创建中间件
13 const mid = createSagaMiddleware();
14 // createSagaMiddleware是一个工厂函数，传入helloSaga参数
    之后会创建一个saga middleware
15 // 使用applyMiddleware将middleware连接到store
16 //2.应用中间件
17 const store = createStore(
18     combineReducers({
19         user
20     })
21     , applyMiddleware(logger,mid));
22 //3.运行中间件
23 mid.run(mySaga)
24 export default store;

```

修改user.reducer.js

```

1 // 定义user的reducer
2 const initialState = {
3     isLogin: false, //一开始表示没登录
4 }
5
6 export default (state = initialState, { type, payload
    }) => {
7     switch (type) {
8         case 'login':
9             // return Object.assign({}, state, {
10                 //     isLogin: true
11                 // })
12             return { ...state, ...{ isLogin: true } };
13             // return {isLogin:true}
14         default:

```

```
15         return state
16     }
17 }
18
19 export const mapStateToProps = state => {
20     const {isLogin} = state.user;
21     return {
22         isLogin: isLogin
23     }
24 }
25 export const mapDispatchToProps = (dispatch) => {
26     return {
27         login: () => {
28             dispatch(asyncLogin());
29         }
30     }
31 }
32 }
33
34 // 异步方法 for redux-thunk
35 /* function asyncLogin() {
36     return (dispatch) => {
37         setTimeout(() => {
38             dispatch({ type: 'login' })
39         }, 1250);
40     }
41 } */
42
43 // for redux-saga
44 function asyncLogin() {
45     alert(1);
46     return {type: 'login_request'}
47 }
48
```



# redux-thunk和redux-saga的区别

- 1 thunk可以接受function类型的action,saga则是纯对象action解决方案
- 2 saga使用generator解决异步问题, 非常容易用同步方式编写异步代码

## UmiJS

它是一个可插拔的企业级的react应用框架。umi以路由为基础并配以完善的插件体系。覆盖从源码到构建产物的每个生命周期, 支持各种功能扩展和业务需求, 目前内外部加起来已有 50+ 的插件。

umi 是蚂蚁金服的底层前端框架, 已直接或间接地服务了 600+ 应用, 包括 java、node、H5 无线、离线 (Hybrid) 应用、纯前端 assets 应用、CMS 应用等。他已经很好地服务了我们的内部用户, 同时希望他也能服务好外部用户。

## 特性

- 📦 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 数据流的深度融合, 支持 duck directory、model 的自动加载、code splitting 等等

## 快速上手

```
1 npm i yarn tyarn -g
2 # 以后所有的yarn 改成tyarn下载
3
4 # 全局安装umi,保证版本是2.0.0以上
5 yarn global add umi
```

## 脚手架

找个空地方新建空目录

```
1 mkdir umi_app && cd umi_app
```

然后通过 `umi g` 创建一些页面

```
1 umi g page index
```

执行命令 `tree`，查看目录结构

```
1 └─ pages
2   └─ index.css
3   └─ index.js
```

然后启动本地服务器

```
1 umi dev
```

# Page index

## 页面中跳转

[文档参考, so easy.](#)

## 路由

umi会根据 `pages` 目录自动生成路由配置

### 基础路由

此操作在上面演示完成

### 动态路由

umi里约定，带 `$` 前缀的目录或文件为动态路由

目录结构如下：

```
1 | └─ pages
2 |   └─ index.css
3 |   └─ index.js
4 |   └─ users
5 |       └─ $id.css
6 |       └─ $id.js
```

路由配置如下：

```
1 {
2   path: '/users/:id',
3   exact: true,
4   component: require('../users/$id.js').default,
5 }
```

修改 `$id.js`

```
1
2 // 约定式路由
3 import styles from './$id.css';
4
5 export default function ({match}) {
6   return (
7     <div className={styles.normal}>
8       <h1>user index {match.params.id}</h1>
9     </div>
10  );
11 }
12
```

当访问 `localhost:8000/users/1` 和 `localhost:8000/user/2` 来查看效果

## 嵌套路由

umi里约定目录下有 `_layout.js` 时会生成嵌套路由，以 `_layout.js` 为该目录的layout

```
1 umi g users/_layout
2 umi g users/index
```

生成如下目录结构

```
1  └─ pages
2    └─ index.css
3    └─ index.js
4    └─ users
5        └─ $id.css
6        └─ $id.js
7        └─ _layout.css
8        └─ _layout.js
```

路由配置如下:

```
1  {
2    path: '/users',
3    exact: false,
4    component: require('../users/_layout.js').default,
5    routes: [
6      {
7        path: '/users',
8        exact: true,
9        component:
10         require('../users/index.js').default,
11      },
12      {
13        path: '/users/:id',
14        exact: true,
15        component:
16         require('../users/$id.js').default,
17      }
18    ]
19  }
```

users/\_layout.js

1

```

2 import styles from './_layout.css';
3
4 export default function(props) {
5   return (
6     <div className={styles.normal}>
7       <h1>Page _layout</h1>
8       <div>
9         {props.children}
10      </div>
11    </div>
12  );
13 }
14

```

users/index.js

```

1
2 import Link from 'umi/link'
3 import styles from './index.css';
4
5 export default function() {
6   return (
7     <div className={styles.normal}>
8       <h1>用户列表</h1>
9       <Link to='/users/1'>用户1</Link>
10      <Link to='/users/2'>用户2</Link>
11    </div>
12  );
13 }
14

```

访问 `localhost:8000/users`

Page \_layout

用户列表

[用户1](#)[用户2](#)

点击用户1查看效果

← → ↻ 🏠 ⓘ localhost:8000/users/1

Page \_layout

user index 1

点击用户2查看效果

← → ↻ 🏠 ⓘ localhost:8000/users/2

Page \_layout

user index 2

配置式路由

在根目录下创建 `config/config.js` 配置文件.此配置项存在时则不会对 `pages` 目录做约定式的解析

```
1 export default {
2   // component是相对于根目录下/pages
3   routes: [
4     { path: '/', component: './index' },
5     {
6       path: '/users', component:
7       './users/_layout',
8       routes: [
9         { path: '/users/', component:
10          './users/index' },
11         { path: '/users/:id', component:
12          './users/$id' }
13       ]
14     },
15   ],
16 };

```

## 404路由

约定 `pages/404.js` 为404页面,

路由配置中添加

```
1 export default {
2   // component是相对于根目录下/pages
3   routes: [
4     { path: '/', component: './index' },
5     {
6       path: '/users', component:
7       './users/_layout',
8       routes: [
9         { path: '/users/', component:
10          './users/index' },
11         { path: '/users/:id', component:
12          './users/$id' }
13       ]
14     },
15   ],
16 };

```



```
10         ]
11     },
12     {components: './404.js'}
13 ],
14 };
15
```

## 权限路由

config/config.js

```
1 export default {
2     // component是相对于根目录下/pages
3     routes: [
4         { path: '/', component: './index' },
5         //约定为大写Routes
6         { path: '/about', component: './about', Routes:
7           ['./routes/PrivateRoute.js'] },
8         {
9             path: '/users', component:
10            './users/_layout',
11            routes: [
12                { path: '/users/', component:
13                './users/index' },
14                { path: '/users/:id', component:
15                './users/$id' }
16            ]
17        },
18        { path: '/login', component: './login' },
19        {component: './404.js'},
20    ],
21 };

```

```
1 | umi g page about #生成about页面
```

根目录下新建 routes/PrivateRoute.js

```
1 | import Redirect from 'umi/redirect';
2 |
3 | export default (props) => {
4 |   if(Math.random() > 0.5){
5 |     return <Redirect to='/login'/>
6 |   }
7 |   return (
8 |     <div>
9 |       {props.children}
10 |     </div>
11 |   )
12 | }
13 |
```

## 引入antd

- 添加antd: `npm i antd -S`
- 添加umi-plugin-react: `npm i umi-plugin-react -D`
- 修改config/config.js

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

page/login.js

```
1 import styles from './login.css';
2 import { Button } from "antd";
3 export default function() {
4   return (
5     <div className={styles.normal}>
6       <h1>Page login</h1>
7       <Button type='primary'>按钮</Button>
8     </div>
9   );
10 }
11
```

效果展示：



## Dvajs

dva是一个基于redux和redux-saga的数据流方案，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

特点：

- 1 1. 易学易用
- 2 - 仅有 6 个 api, 对 redux 用户尤其友好, 配合 umi 使用后更是降低为 0 API
- 3 2. elm 概念
- 4 - 通过 reducers, effects 和 subscriptions 组织 model, 简化 redux 和 redux-saga 引入的概念
- 5 3. 插件机制
- 6 - 比如 dva-loading 可以自动处理 loading 状态, 不用一遍遍地写 showLoading 和 hideLoading
- 7 4. 支持HMR
- 8 - 基于babel-plugin-dva-hmr实现components、routes、和 models的HMR

## umi中使用dva

- 1 page g page goods //创建goods页面

## config/config.js修改配置

```
1 export default {
2   // component是相对于根目录下/pages
3   routes: [
4     { path: '/', component: './index' },
5     { path: '/goods', component: './goods' }, #添加
6     { path: '/about', component: './about', Routes:
7     ['./routes/PrivateRoute.js'] },
8     {
9       path: '/users', component:
10      './users/_layout',
11      routes: [
12        { path: '/users/', component:
13        './users/index' },
```

```

11         { path: '/users/:id', component:
    './users/$id' }
12     ]
13 },
14     { path: '/login', component: './login' },
15     { component: './404.js' },
16
17 ],
18 plugins: [
19     ['umi-plugin-react', {
20         antd: true,
21         dva: true
22     }],
23 ],
24 };

```

## 配置models

创建models/goods.js

```

1  export default {
2      namespace: "goods", //model的命名空间, 区分多个model
3      state: [{ title: 'web架构课' }, { title: 'python架
    构课' }], //初始状态
4      reducers: {
5          addGood(state, action) {
6              return [...state,
    {title: action.payload.title}]
7          }
8      }, //更新状态
9      effects: { //副作用 异步操作
10     },
11 }

```

## 配置goods.js

```
1 import { Component } from 'react';
2 import styles from './goods.css';
3 import { connect } from "dva";
4 import { Card, Button } from "antd";
5
6 @connect(
7   state => ({
8     goodsList: state.goods //获取指定命名空间的模型状态
9   }),
10  {
11    addGood: title => ({
12      type: 'goods/addGood', //action的type需要以命名空间为前缀+reducer名称
13      payload: { title }
14    }),
15  }
16 )
17 export default class extends Component {
18   render() {
19     return (
20       <div className={styles.normal}>
21         <h1>Page goods</h1>
22         <div>
23           {
24             this.props.goodsList.map(good => {
25               return (
26                 <Card key={good.title}>
27                   <div>{good.title}</div>
28                 </Card>
29               )
30             })
31           }

```

```

32         </div>
33         <div>
34             <Button onClick={() => this.props.addGood('商
品' + new Date().getTime())}>
35                 添加商品
36             </Button>
37         </div>
38     </div>
39 );
40 }
41 }

```

## 模拟Mock

创建mock/goods.js

```

1
2 let data = [ //初始状态
3     {
4         title: 'web架构课'
5     },
6     {
7         title: 'python架构课'
8     }
9 ];
10
11 export default {
12     "get /api/goods": function (req, res) {
13         setTimeout(() => {
14             res.json({ result: data });
15         }, 1000);
16     }
17 }

```

models/goods.js

```

1 import axios from 'axios'
2 function getGoods() {
3     return axios.get('/api/goods')
4 }
5 export default {
6     namespace: "goods", //model的命名空间, 区分多个model
7     state: [], //初始状态
8     reducers:{
9         addGood(state,action){
10             return [...state,
11 {title:action.payload.title}]
12         },
13         initGoods(state,action){
14             return action.payload
15         }, //更新状态
16     effects: { //副作用 异步操作
17         *getList(action, { call, put }) {
18             const res = yield call(getGoods);
19             // type的名字 不需要命名空间
20             yield put({ type: 'initGoods', payload:
21 res.data.result })
22         },
23     }

```

goods.js修改

```

1 import { Component } from 'react';
2 import styles from './goods.css';
3 import { connect } from "dva";
4 import { Card, Button } from "antd";
5
6 @connect(

```



```

7   state => ({
8     goodsList: state.goods  //获取指定命名空间的模型状态
9   }),
10  {
11    addGood: title => ({
12      type: 'goods/addGood', //action的type需要以命名空间为前缀+reducer名称
13      payload: { title }
14    }),
15    getList: () => ({
16      type: 'goods/getList',
17    })
18  }
19 )
20 export default class extends Component {
21   componentDidMount() {
22     //调用
23     this.props.getList()
24   }
25   render() {
26     return (
27       <div className={styles.normal}>
28         {/**/}
29       </div>
30     );
31   }
32 }

```

**加载状态:**利用内置的dva-loading实现

- 获取加载状态,goods.js

```

1  @connect(
2    state => ({
3      loading: state.loading

```

```
4    }),  
5    {  
6      ...  
7    }  
8  )  
9  export default class extends Component {  
10    render(){  
11      if(this.props.loading.models.goods){  
12        return <div>加载中.....</div>  
13      }  
14      ....  
15    }  
16  }
```