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

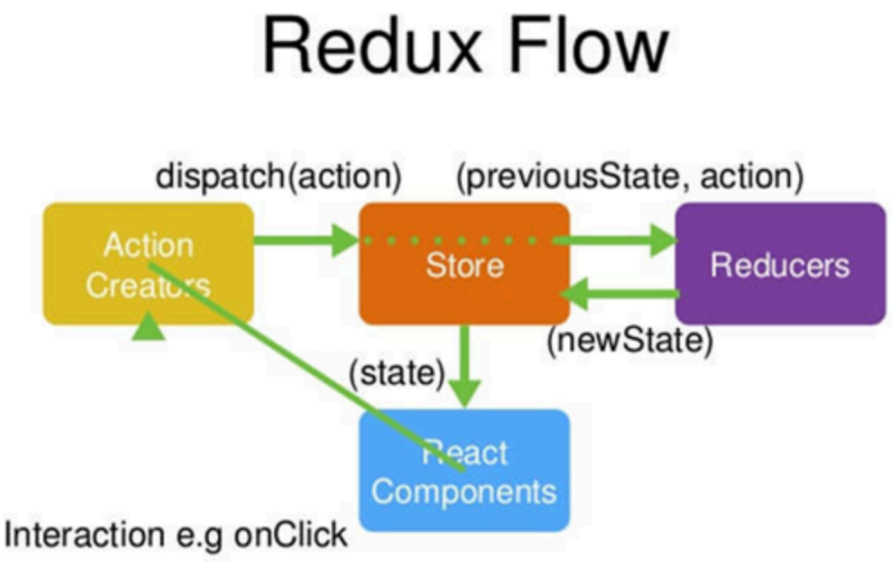
redux

资源

redux

react-redux

起步



redux快速上手

1.安装

```
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

```
import {
    createStore
} from 'redux';
// 创建reducer 状态修改具体执行者
function counter(state = 0, action) {
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        case 'DRCREMENT':
            return state - 1;
        default:
            return state;
    }
```

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

ReduxTest.js

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

class ReduxTest extends Component {
    render() {
        return (
            <div>
                <p>
                    {store.getState()}
                </p>
                <button onClick={() => store.dispatch({type:"DRCREMENT"})}>-1</button>
                <button onClick={() => store.dispatch({type: "INCREMENT" })}>+1</button>
            </div>
        );
    }
}

export default ReduxTest;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import ReduxTest from './components/ReduxTest';
import store from './store'
function render() {
    ReactDOM.render(<ReduxTest />,
document.querySelector('#root'));
}
render();
// 每次 state 更新时，打印日志
// 注意 subscribe() 返回一个函数用来注销监听器
// 订阅
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

```
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

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

```
import store from './store'
import { Provider } from 'react-redux';
function render() {
    ReactDOM.render((
        <Provider store = {store}>
            <ReduxTest />
        </Provider>
    ), document.querySelector('#root'));
}
render();
//订阅不需要了
// store.subscribe(render);
```

React Redux提供了`connect`将组件连接到store的功能

修改ReduxTest.js

```
import React, { Component } from 'react';
import { connect } from "react-redux";
const mapStateToProps = state => {
    return {
        num: state
    }
}
const mapDispatchToProps = dispatch => {
    return {
        increment: () => {
            dispatch({ type: 'INCREMENT' })
        },
        decrement: () => {
            dispatch({
                type: 'DRCREMENT'
            })
        }
    }
```

```
}
class ReduxTest extends Component {
    render() {
        return (
            <div>
                <p>{this.props.num}</p>
                <button onClick={() =>
this.props.decrement()}>-1</button>
                <button onClick={() =>
this.props.increment()}>+1</button>
            </div>
        );
    }
}

export default connect(mapStateToProps,
mapDispatchToProps)(ReduxTest);;
```

装饰器写法

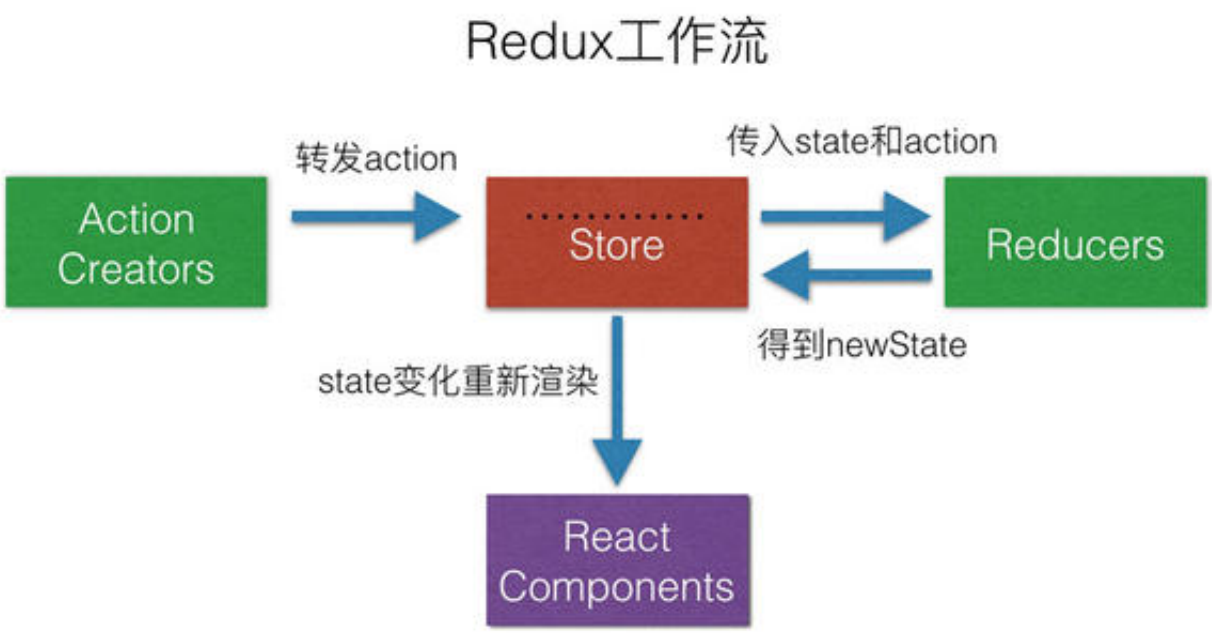
```
mport React, { Component } from 'react';
import { connect } from "react-redux";
const mapStateToProps = state => {
    return {
        num: state
    }
}
const mapDispatchToProps = dispatch => {
    return {
        increment: () => {
            dispatch({ type: 'INCREMENT' })
        },
        decrement: () => {
            dispatch({
```

```
                type: 'DRCREMENT'
            })
        }
    }
}
@connect(mapStateToProps, mapDispatchToProps)
class ReduxTest extends Component {
    render() {
        return (
            <div>
                <p>{this.props.num}</p>
                <button onClick={() =>
this.props.decrement()}>-1</button>
                <button onClick={() =>
this.props.increment()}>+1</button>
            </div>
        );
    }
}

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。

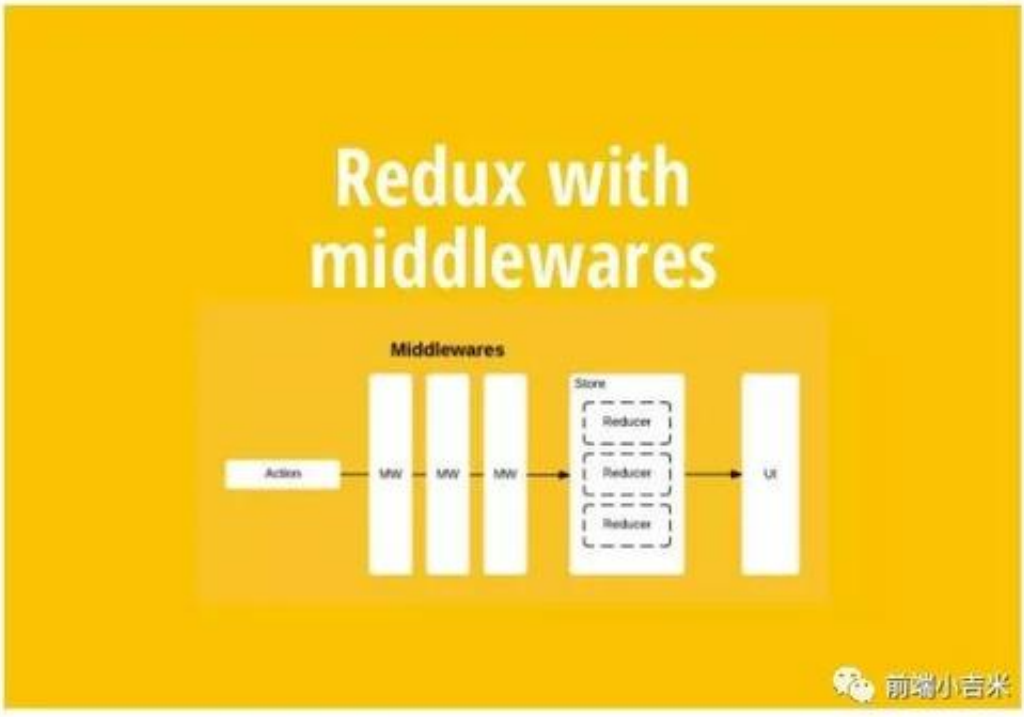
使用`connect()`前，需要先定义`mapStateToProps`这个函数来指定如何把当前的Redux store state映射到展示组件的props中。

redux中间件



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

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

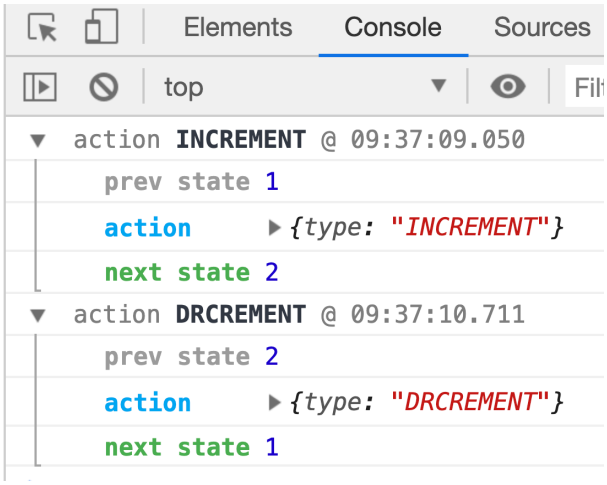
redux-logger的使用在store.js加入

```
import {
    createStore,
    applyMiddleware
} from 'redux';
import logger from 'redux-logger';
// 创建reducer
function counter(state = 0, action) {
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        case 'DRCREMENT':
            return state - 1;
        default:
            return state;
    }
}

export default createStore(counter,
applyMiddleware(logger));
```

效果：

1

-1 +1



redux-thunk 在store.js修改

```
import {
    createStore,
    applyMiddleware
} from 'redux';
import logger from 'redux-logger';
import thunk from 'redux-thunk';
// 创建reducer
function counter(state = 0, action) {
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        case 'DRCREMENT':
            return state - 1;
        default:
            return state;
    }
}
export default createStore(counter,
applyMiddleware(logger,thunk));
```

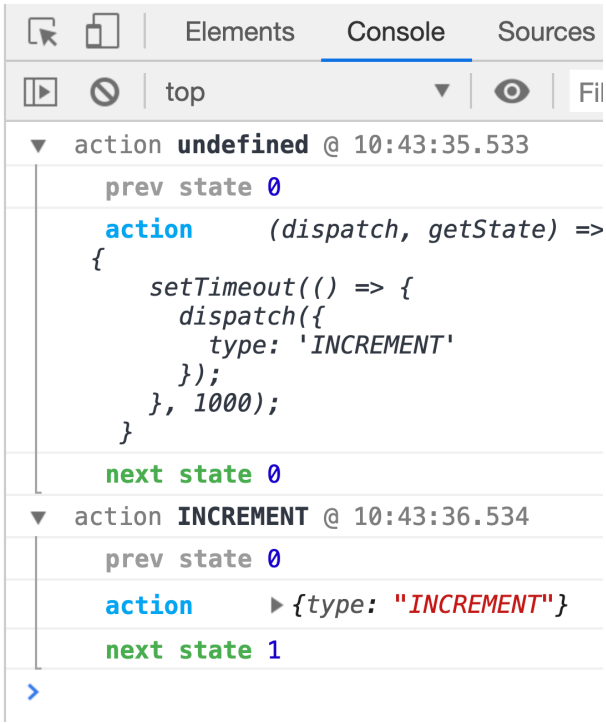
> 添加thunk的作用:action默认接收一个对象，执行下个任务，如果是个函数，则需要异步处理。

redux-thunk 在ReduxTest.js修改

```
import React, { Component } from 'react';
// import store from '../store';
import { connect } from "react-redux";
const mapStateToProps = state => {
    return {
        num: state
    }
}
const asyncAdd = () => {
    return (dispatch,getState)=>{
        setTimeout(() => {
            dispatch({type:'INCREMENT'})
        }, 1000);
    }
}
const mapDispatchToProps = (dispatch) => {
    return {
        increment: () => {
            dispatch({ type: 'INCREMENT' });
        },
        decrement: () => {
            dispatch({
                type: 'DRCREMENT'
            })
        },
        asyncIncrement: () => {
            //action的动作默认是对象，如果是返回函数则使用redux-thunk处理
            dispatch(asyncAdd());
        }

    }
}
@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:



重构项目

新建store/couter.reduce.js

```
// 创建reducer
const counter = (state = 0, action) => {
    switch (action.type) {
        case 'INCREMENT':
            return state + 1;
        case 'DRCREMENT':
            return state - 1;
        default:
            return state;
    }
}
export const mapStateToProps = state => {
    return {
        num: state
    }
}
const asyncAdd = () => {
    return (dispatch, getState) => {
        setTimeout(() => {
            dispatch({ type: 'INCREMENT' })
        }, 1000);
    }
}
export const mapDispatchToProps = (dispatch) => {
    return {
        increment: () => {
            // dispatch({ type: 'INCREMENT' })
            dispatch({ type: 'INCREMENT' });
        },
        decrement: () => {
            dispatch({
                type: 'DRCREMENT'
```

```
            })
        },
        asyncIncrement: () => {
            dispatch(asyncAdd());
        }

    }
}

export default counter;
```

新建store/index.js

```
import {
    createStore,
    applyMiddleware
} from 'redux';
import logger from 'redux-logger';
import thunk from 'redux-thunk';
import counter from './couter.reducer';

export default createStore(counter,
applyMiddleware(logger,thunk));
```

重构ReduxTest.js

```
import React, { Component } from 'react';
// import store from '../store';
import { connect } from "react-redux";
import {mapStateToProps,mapDispatchToProps} from
'../store/couter.reducer';

@connect(mapStateToProps, mapDispatchToProps)
class ReduxTest extends Component {
```

```
    render() {
        return (
            <div>
                <p>{this.props.num}</p>
                <button onClick={() => this.props.decrement()}>-1</button>
                <button onClick={() => this.props.increment()}>+1</button>
                <button onClick={() => this.props.asyncIncrement()}>async+1</button>
            </div>
        );
    }
}

export default ReduxTest;
```

合并reducer

使用`combineReducers`进行复合，实现状态的模块化

```
import {
    createStore,
    applyMiddleware,
    combineReducers
} from 'redux';
import logger from 'redux-logger';
import thunk from 'redux-thunk';
import counter from './couter.reducer';

export default createStore(
    combineReducers({
        counter
    }), applyMiddleware(logger,thunk));
```

counter.reducer.js

```
export const mapStateToProps = state => {
    return {
       //加上当前状态的key,来进行标识
        num: state.counter
    }
}
```

Mobx快速入门

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

```
npm i mobx mobx-react -S
```

新建store/mobx.js

```
import { observable,action,computed} from "mobx";

// 观察者
const appState = observable({
    num: 0
})
// 方法
appState.increment = action(()=>{
    appState.num+=1;
})
appState.decrement = action(()=>{
    appState.num-=1;
})
export default appState;
```

index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import {Provider} from 'react-redux';
import MobxTest from "./components/MobxTest";
ReactDOM.render((
    <div>
        <MobxTest appState = {appState}/>
    </div>
), document.querySelector('#root'));
```

MobxTest.js

```
import React, { Component } from 'react';
import { observer } from "mobx-react";
class MobxTest extends Component {
    render() {
        return (
            <div>
                {this.props.appState.num}
                <button onClick={() =>
this.props.appState.decrement()}>-1</button>
                <button onClick={() =>
this.props.appState.increment()}>+1</button>
            </div>
        );
    }
}

export default observer(MobxTest);
```

装饰器写法：

store/mobx.decorator.js

```
import { observable, action, computed } from "mobx";
// 常量改成类
class AppState {
    @observable num = 0;
    @action
    increment(){
        this.num +=1;
    }
    @action
    decrement() {
        this.num -= 1;
    }
}
const appState = new AppState();

export default appState;
```

MobxTest.decorator.js

```
import React, { Component } from 'react';
import { observer } from "mobx-react";
@observer
class MobxTest extends Component {
    render() {
        return (
            <div>
                {this.props.appState.num}
                <button onClick={() =>
this.props.appState.decrement()}>-1</button>
                <button onClick={() =>
this.props.appState.increment()}>+1</button>
            </div>
        );
    }
```

```
}

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

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

基本路由使用

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

```
    return (
        <h2>我是首页</h2>
    )
}
function Course() {
    return (
        <h2>我是课程</h2>
    )
}
function User() {
    return (
        <h2>我是首页</h2>
    )
}
class Basic_router extends Component {
    render() {
        return (
            <Router>
                <div>
                    {/* 定义路由页面 */}
                    <ul>
                        <li>
                            <Link to='/'>首页</Link>
                        </li>
                        <li>
                            <Link to='/course'>课
程</Link>
                        </li>
                        <li>
                            <Link to='/user'>用
户</Link>
                        </li>
                    </ul>

                    {/* 配置路由 */}
```

```
                    {/* 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 */}
                    <Route exact path='/' component={Home}></Route>
                    <Route path='/course' component={Course}></Route>
                    <Route path='/user' component={User}></Route>
                </div>

            </Router>
        );
    }
}

export default Basic_router;
```

路由URL参数(二级路由)

```
import React, { Component } from 'react';
import { BrowserRouter as Router, Route, Link } from "react-router-dom";
function Home() {
    return (
        <h2>我是首页</h2>
    )
}
function Course() {
    return (
        <div className='course'>
            <h2>我的课程</h2>
            {/*定义二级路由页面*/}
            <ul>
                <li>
```

```
                    <Link to='/course/vue'>Vue</Link>
                </li>
                <li>
                    <Link
to='/course/React'>React</Link>
                </li>
                <li>
                    <Link
to='/course/Angular'>Angular</Link>
                </li>
            </ul>
             {/*配置路由参数*/}
            <Route path='/course/:id' component=
{CourseChild}></Route>
            <Route exact path={match.path} render={()
=> <h3>请选择你的课程</h3>} />
        </div>

    )
}
function User() {
    return (
        <h2>我是首页</h2>
    )
}
//二级路由页面显示
function CourseChild({match,history,location}) {
    //match: 匹配路由信息对象
    //location: 本地信息对象
    //history: 历史信息对象
    console.log(location,match,history);

    return (
        <div>
            {match.params.id}
```

```
        </div>
    )
}
class Basic_router extends Component {
    render() {
        return (
            <Router>
                <div>
                    {/* 定义路由页面 */}
                    <ul>
                        <li>
                            <Link to='/'>首页</Link>
                        </li>
                        <li>
                            <Link to='/course'>课程</Link>
                        </li>
                        <li>
                            <Link to='/user'>用户</Link>
                        </li>
                    </ul>
                    {/* 配置路由 */}
                    <Route exact path='/' component={Home}></Route>
                    <Route path='/course' component={Course}></Route>
                    <Route path='/user' component={User}></Route>

                </div>

            </Router>
        );
    }
```

```
}

export default Basic_router;
```

上述的Course组件也可以这样修改

```
function Course({match}) {
    return (
        <div className='course'>
            <h2>我的课程</h2>
            <ul>
                <li>
                    <Link to={`${match.url}/vue`}>Vue</Link>
                </li>
                <li>
                    <Link to={`${match.url}/react`}>React</Link>
                </li>
                <li>
                    <Link to={`${match.url}/angular`}>Angular</Link>
                </li>
            </ul>
            <Route path='/course/:id' component={CourseChild}></Route>
            <Route exact path={match.path} render={() => <h3>请选择你的课程</h3>} />
        </div>

    )
}
```

不匹配(404)

```
// 404页面展示
function NoMatch() {
    return <div>404页面,网页找不到了</div>
}
class Basic_router extends Component {
    render() {
        return (
            <Router>
                <div>
                    {/* 定义路由页面 */}
                    <ul>
                        <li>
                            <Link to='/'>首页</Link>
                        </li>
                        <li>
                            <Link to='/course'>课程</Link>
                        </li>
                        <li>
                            <Link to='/user'>用户</Link>
                        </li>
                    </ul>
                    {/* 配置路由 */}
                    <Route exact path='/' component={Home}></Route>
                    <Route path='/course' component={Course}></Route>
                    <Route path='/user' component={User}></Route>
                    {/*添加不匹配路由配置*/}
                    <Route  component={NoMatch}></Route>
                </div>
```

```
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
   程</Link>
19                         </li>
20                         <li>
21                             <Link to='/user'>用
   户</Link>
```

```
                        </li>
                    </ul>
                    {/* 配置路由 */}
                    <Switch>
                        <Route exact path='/' component={Home}></Route>
                        <Route path='/course' component={Course}></Route>
                        <Route path='/user' component={User}></Route>
                         {/*添加不匹配路由配置*/}
                        <Route component={NoMatch}></Route>
                    </Switch>
                </div>

            </Router>
        );
    }
}
```

命令式导航

```
function Home({ location }) {
    console.log(location);
    return (
        <div>
            <h1>{location.state ? location.state.foo : ""}</h1>
            <h2>我是首页</h2>
        </div>
    )
}
function CourseChild({ match, history, location }) {
```

```
    return (
        <div>
            {match.params.id}课程
            <button onClick={history.goBack}>返回</button>
            <button onClick={() => { history.push('/') }}>跳转首页</button>
            <button onClick={()=>{
                history.push({
                    pathname:'/',
                    state:{
                        foo:'bar'
                    }
                })
            }}>跳转首页,并携带值</button>
        </div>
    )
}
```

重定向Redirect

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

function UserDeatil({match,location}) {
    return (
        <div>个人详情页面</div>
    )
}
function UserOrder(params) {
    return (
        <div>用户订单页面</div>
    )
```

```
}
function User() {
    return (
       <div>
           <h2>
               <Link to='/user/detail'>个人信息</Link>
           </h2>
            <h2>
                <Link to='/user/order'>个人订单</Link>
            </h2>
            <Switch>
                <Route path="/user/detail" component={UserDeatil}></Route>
                <Route path="/user/order" component={UserOrder}></Route>
                {/*重定向*/}
                <Redirect to='/user/detail'></Redirect>
            </Switch>
       </div>

    )
}

class Basic_router extends Component {
    render() {
        return (
            <Router>
                <div>
                    {/* 定义路由页面 */}
                    <ul>
                        <li>
                            <Link to='/user'>用户</Link>
                        </li>
                    </ul>
```

```
                    {/* 配置路由 */}
                    <Switch>
                        <Route path='/user' component={User}></Route>
                    </Switch>
                </div>

            </Router>
        );
    }
}

export default Basic_router;
```

路由守卫

定义可以验证的高阶组件

```
// 路由守卫：定义可以验证的高阶组件
function PrivateRoute({ component: Component, ...rest }) {
    return (
        <Route
            {...rest}
            render={props =>
                Auth.isAuth ? (
                    <Component {...props} />
                ) : (
                        <Redirect to={{
                            pathname: "/login",
                            state: { from: props.location }
                        }}
                        />
```

```
                    )
                }
            />
        )
}
```

认证类Auth

```
const Auth = {
    isAuth: false,
    login(cb) {
        this.isAuth = true;
        setTimeout(cb, 1000);
    },
    signout(cb) {
        this.isAuth = false;
        setTimeout(cb, 1000);
    }
}
```

定义登录组件

```
class Login extends Component {
    state = { isLogin: false };
    handlerlogin = () => {
        Auth.login(() => {
            this.setState({
                isLogin:true
            })
        })
    }
    render() {
        let { isLogin } = this.state;
```

```
        let { from } = this.props.location.state || { from: { pathname: '/' } }
        if (isLogin) return <Redirect to={from} />
        return (
            <div>
                <p>请先登录</p>
                <button onClick={this.handlerlogin}>登录</button>
            </div>
        );
    }
}
```

主路由组件中使用自定义路由和定义登录路由配置

```
<Switch>
    <Route exact path='/' component={Home}></Route>
    {/* <Route path='/course' component={Course}></Route> */}
    <PrivateRoute path='/course' component={Course}></PrivateRoute>
    <Route path='/user' component={User}></Route>
    <Route path='/login' component={Login}></Route>
    <Route component={NoMatch}></Route>
</Switch>
```

集成到redux中

1. 新建/store/user.reducer.js

```
const initState = {
  isLogin: false,//表示用户未登录
  userInfo: {}
}
```

```
function user(state = initState, action) {
  switch (action.type) {
    case 'login':
      return { isLogin: true }

    default:
      return initState
  }

}

export const mapStateToProps = state => {
  return {
    // 加上当前状态的key,来进行模块化的标识
    user: state.user
  }
}

const login = () => {
  return (dispatch) => {
    setTimeout(() => {
      dispatch({ type: 'login' })
    }, 1000);
  }
}
export const mapDispatchToProps = dispatch => {
  return {
    //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
    login: () => {
      dispatch(login())
    }
  }
}
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

```
....
import { connect } from "react-redux";
import { mapStateToProps } from
"./store/user.reducer";
// 高阶组件：定义验证功能的路由组件
@connect(mapStateToProps)
class PrivateRoute extends Component {
  render() {
    const Comp = this.props.component;
    return (
      <Route
        {...this.props}
        component={
          (props) =>
            this.props.user.isLogin ?
              (<Comp {...props} />) :
              (<Redirect to={{ pathname: '/login',
state: { from: props.location } }} />)
        }>
      </Route>
    )
  }
}
....
```

login.js组件修改

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

```
@connect(mapStateToProps,mapDispatchToProps)
 class Login extends Component {
  handleLogin = () => {
    // 异步处理
    this.props.login();
  }

  render() {
    let { isLogin } = this.props.user;
    let path =
this.props.location.state.from.pathname
    if (isLogin) {
      return <Redirect to={path}/>

    } else {
      return (
        <div>
          <p>请先登录</p>
          <Button onClick={this.handleLogin}>登
录</Button>
        </div>
      )
    }
  }
}
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。

```
export default function createStore(reducer,preloadedState,enchancer) {

     if (typeof preloadedState === 'function' && typeof enhancer === 'function' || typeof enhancer === 'function' && typeof arguments[3] === 'function') {
        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.');
    }
    //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
    if (typeof preloadedState === 'function' && typeof enhancer === 'undefined') {
        enhancer = preloadedState;
        preloadedState = undefined;
    }

    if (typeof enhancer !== 'undefined') {
        if (typeof enhancer !== 'function') {
            throw new Error('Expected the enhancer to be a function.');
        }
```

```
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
            currentState=currentReducer(currentState,action);
34          //通知其他的组件执行
35          currentListeners.forEach(l=>l());
36      }
37      //如果说其他的组件需要订阅状态变化时间的话，
38      function subscribe(listener) {
39          //将监听函数放入一个队列中
40          currentListeners.push(listener);
41          return function () {
42              currentListeners = currentListeners.filter(item=>item!==listener);
```

```
        }
    }
    //初始化的操作
    dispatch({type:'@@INIT'});
    return {
        getState,
        dispatch,
        subscribe
    }
}
```

- 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

```
export function applyMiddleware(...middlewares){
    return  (createStore) => {
        return function (...args) {
            //创建原始的store
            const store = createStore(...args);
            //获取原始的dispatch
            const _dispatch = store.dispatch;
            const middlewareAPI = {
                getState: store.getState,
                dispatch: (...args)=> {
                    return dispatch(...args)
                }
            };
            //调用第一层中间件
            const middlewareChain = middlewares.map(
(middleware)=> {
                //让每个中间件执行，传入一个对象
{getState,dispatch}
                return middleware(middlewareAPI);
```

```
            });
            //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
            _dispatch = compose(...middlewareChain)(store.dispatch);
             // return 一个被增强了dispatch的store
            return  {
                ...store,
                dispatch: _dispatch
             };
        };
    };
}
function compose(...fns){ //[add1,add2,add3] 都是函数
    if(fns.length === 0){
        return arg => arg;
    }
    if(fn2.length === 1){
        return fns[0]
    }
    return fns.reduce((f1,f2)=>(...args)=>f1(f2(...args)))
}
```

React-redux原理

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

```
 */
export default function connect(mapStateToProps,mapDispatchToProps) {
    return function (Com) {
        //在这个组件里实现仓库和组件的连接
        class Proxy extends Component{

 state=mapStateToProps(this.props.store.getState())
        componentDidMount() {
            //更新状态
            this.unsubscribe = this.props.store.subscribe(() => {

this.setState(mapStateToProps(this.props.store.getState()));
            });
        }
        componentWillUnmount = () => {
            this.unsubscribe();
        }
        render() {
            let actions={};
            //如果说mapDispatchToProps是一个函数，执行后得到属性对象
            if (typeof mapDispatchToProps === 'function') {
                actions = mapDispatchToProps(this.props.store.dispatch);
                //如果说mapDispatchToProps是一个对象的话，我们需要手工绑定
            } else {

actions=bindActionCreators(mapDispatchToProps,this.props.store.dispatch);
            }
```

```
            return <Com {...this.state} {...actions}/>
        }
    }
}
export default class Provider extends Component{
    //规定如果有人想使用这个组件，必须提供一个redux仓库属性
    static propTypes={
        store:PropTypes.object.isRequired
    }
    render() {
        let value={store:this.props.store};
        return (
            <StoreProvider value={value}>
                {this.props.children}
            </StoreProvider>
        )
    }
}
```

redux-thunk

```
const thunk = ({dispatch,getState})=>next=>action=>{
    if(typeof action=='function'){
        return action(dispatch,getState)
    }
    return next(action)
}
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

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

新建store/sagas.js

```
import { call, put, takeEvery } from "redux-saga/effects";

// 模拟登录的api 一般项目开发中会将此api放入service文件夹下
const api = {
    login(){
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                if(Math.random() > 0.5){
                    resolve({id:1,name:"Tom"})
                }else{
                    reject('用户名或密码错误')
                }
            }, 1000);
        })
```

```
15
16      }
17  }
18
19  // worker saga :将login action被dispacth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'
```

```
import mySaga from './sagas';

// 1.创建中间件
const mid = createSagaMiddleware();
// createSagaMiddleware是一个工厂函数，传入helloSaga参数之后会创建一个saga middleware
// 使用applyMiddleware将middleware连接到store
//2.应用中间件
const store =  createStore(
    combineReducers({
        user
    })
    , applyMiddleware(logger,mid));
//3.运行中间件
mid.run(mySaga)
export default store;
```

修改user.reducer.js

```
// 定义user的reducer
const initialState = {
    isLogin: false,//一开始表示没登录
}

export default (state = initialState, { type, payload }) => {
    switch (type) {
        case 'login':
            // return Object.assign({}, state, {
            //     isLogin: true
            // })
            return { ...state, ...{ isLogin: true} };
            // return {isLogin:true}
        default:
```

```
            return state
    }
}

export const mapStateToProps = state => {
    const {isLogin} = state.user;
    return {
        isLogin: isLogin
    }
}
export const mapDispatchToProps = (dispatch) => {
    return {
        login: () => {
            dispatch(asyncLogin());
        }

    }
}

// 异步方法 for redux-thunk
/* function asyncLogin() {
    return (dispatch) => {
        setTimeout(() => {
            dispatch({ type: 'login' })
        }, 1250);
    }
} */

// for redux-saga
function asyncLogin() {
    alert(1);
    return {type:'login_request'}
}

```

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以路由在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 数据流的深入融合**，支持 duck directory、model 的自动加载、code splitting 等等

快速上手

```
npm i yarn tyarn -g
# 以后所有的yarn 改成tyarn下载

# 全局安装umi,保证版本是2.0.0以上
yarn global add umi
```

脚手架

找个空地方新建空目录

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

然后通过`umi g`创建一些页面

```
umi g page index
```

执行命令`tree`，查看目录结构

```
└── pages
    ├── index.css
    ├── index.js
```

然后启动本地服务器

```
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：

```
└── pages
    ├── index.css
    ├── index.js
    └── users
        ├── $id.css
        ├── $id.js
```

路由配置如下：

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

修改`$id.js`

```

// 约定式路由
import styles from './$id.css';

export default function ({match}) {
    return (
        <div className={styles.normal}>
            <h1>user index {match.params.id}</h1>
        </div>
    );
}

```

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

```
umi g users/_layout
umi g users/index
```

生成如下目录结构

```
└── pages
    ├── index.css
    ├── index.js
    └── users
        ├── $id.css
        ├── $id.js
        ├── _layout.css
        ├── _layout.js
```

路由配置如下：

```
{
    path: '/users',
    exact: false,
    component: require('../users/_layout.js').default,
    routes: [
        {
            path: '/users',
            exact: true,
            component:
require('../users/index.js').default,
        },
        {
            path: '/users/:id',
            exact: true,
            component:
require('../users/$id.js').default,
        },
    ]
}
```

`users/_layout.js`

```

```

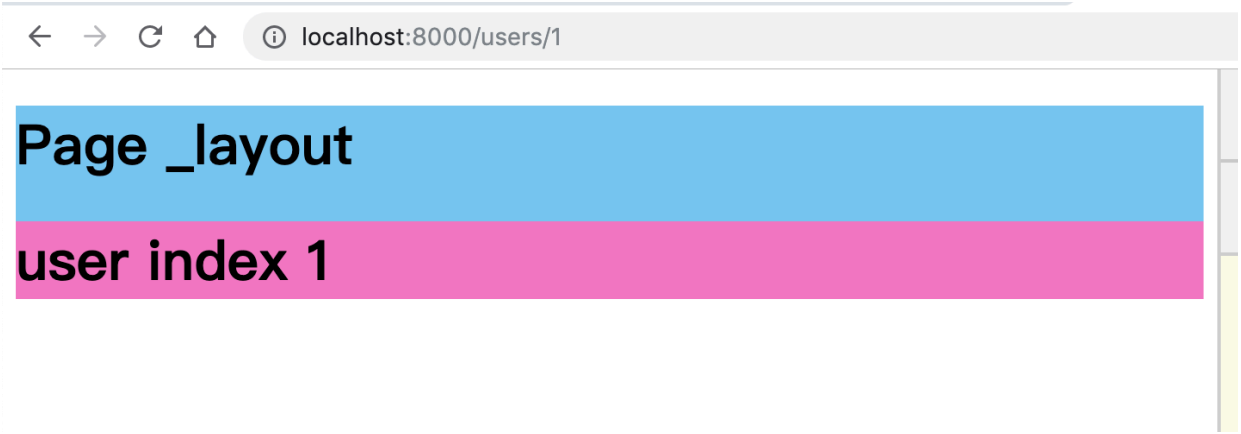
```
import styles from './_layout.css';

export default function(props) {
  return (
    <div className={styles.normal}>
      <h1>Page _layout</h1>
      <div>
        {props.children}
      </div>
    </div>
  );
}

```

users/index.js

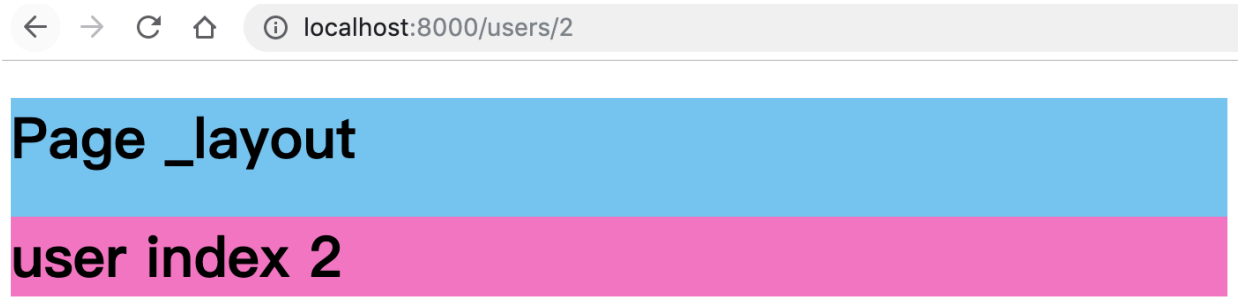
```

import Link from 'umi/link'
import styles from './index.css';

export default function() {
  return (
    <div className={styles.normal}>
      <h1>用户列表</h1>
      <Link to='/users/1'>用户1</Link>
      <Link to='/users/2'>用户2</Link>
    </div>
  );
}

```

访问 localhost:8000/users

点击用户1查看效果



点击用户2查看效果



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**

```
export default {
    // component是相对于根目录下/pages
    routes: [
        { path: '/', component: './index' },
        {
            path: '/users', component:
'./users/_layout',
            routes: [
                { path: '/users/', component:
'./users/index' },
                { path: '/users/:id', component:
'./users/$id' }
            ]
        },
    ],
};
```

404路由

约定`pages/404.js`为404页面，

路由配置中添加

```
export default {
    // component是相对于根目录下/pages
    routes: [
        { path: '/', component: './index' },
        {
            path: '/users', component:
'./users/_layout',
            routes: [
                { path: '/users/', component:
'./users/index' },
                { path: '/users/:id', component:
'./users/$id' }
```

```
            ]
        },
        {components:'./404.js'}
    ],
};

```

权限路由

`config/config.js`

```
export default {
    // component是相对于根目录下/pages
    routes: [
        { path: '/', component: './index' },
        //约定为大写Routes
        { path: '/about', component: './about', Routes: ['./routes/PrivateRoute.js'] },
        {
            path: '/users', component: './users/_layout',
            routes: [
                { path: '/users/', component: './users/index' },
                { path: '/users/:id', component: './users/$id' }
            ]
        },
        { path: '/login', component: './login' },
        {component:'./404.js'},

    ],
};
```

```
umi g page about #生成about页面
```

根目录下新建`routes/PrivateRoute.js`

```
import Redirect from 'umi/redirect';

export default (props) => {
    if(Math.random() > 0.5){
        return <Redirect to='/login'/>
    }
    return (
        <div>
            {props.children}
        </div>
    )
}

```

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

```
plugins: [
   ['umi-plugin-react', {
     antd: true,
   }],
 ],
```

page/login.js

```
import styles from './login.css';
import { Button } from "antd";
export default function() {
  return (
    <div className={styles.normal}>
      <h1>Page login</h1>
      <Button type='primary'>按钮</Button>
    </div>
  );
}

```

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.易学易用
  - 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
2.elm概念
  - 通过 reducers，effects 和 subscriptions 组织 model，简化 redux 和 redux-saga 引入的概念
3.插件机制
  - 比如 dva-loading 可以自动处理 loading 状态，不用一遍遍地写 showLoading 和 hideLoading
4.支持HMR
  - 基于babel-plugin-dva-hmr实现components、routes、和models的HMR
```

umi中使用dva

```
page g page goods //创建goods页面
```

config/config.js修改配置

```
export default {
    // component是相对于根目录下/pages
    routes: [
        { path: '/', component: './index' },
        { path: '/goods', component: './goods' }, #添加位置
        { path: '/about', component: './about', Routes: ['./routes/PrivateRoute.js'] },
        {
            path: '/users', component: './users/_layout',
            routes: [
                { path: '/users/', component: './users/index' },
```

```
                { path: '/users/:id', component: './users/$id' }
            ]
        },
        { path: '/login', component: './login' },
        { component: './404.js' },

    ],
    plugins: [
        ['umi-plugin-react', {
            antd: true,
            dva: true
        }],
    ],
};
```

配置models

创建models/goods.js

```
export default {
    namesapce: "goods", //model的命名空间，区分多个model
    state: [{ title: 'web架构课' }, { title: 'python架构课' }],//初始状态
    reducers:{
        addGood(state,action){
            return [...state,{title:action.payload.title}]
        }
    }, //更新状态
    effects: { //副作用 异步操作
    },
}
```

配置goods.js

```
import { Component } from 'react';
import styles from './goods.css';
import { connect } from "dva";
import { Card, Button } from "antd";

@connect(
  state => ({
    goodsList: state.goods  //获取指定命名空间的模型状态
  }),
  {
    addGood: title => ({
      type: 'goods/addGood', //action的type需要以命名空间为前缀+reducer名称
      payload: { title }
    }),
  }
)
export default class extends Component {
  render() {
    return (
      <div className={styles.normal}>
        <h1>Page goods</h1>
        <div>
          {
            this.props.goodsList.map(good => {
              return (
                <Card key={good.title}>
                  <div>{good.title}</div>
                </Card>
              )
            })
          }
```

```
        </div>
        <div>
          <Button onClick={() => this.props.addGood('商品' + new Date().getTime())}>
            添加商品
          </Button>
        </div>
      </div>
    );
  }
}
```

模拟Mock

创建mock/goods.js

```
let data = [ //初始状态
    {
        title: 'web架构课'
    },
    {
        title: 'python架构课'
    }
];

export default {
    "get /api/goods": function (req, res) {
        setTimeout(() => {
            res.json({ result: data });
        }, 1000);
    }
}
```

models/goods.js

```
import axios from 'axios'
function getGoods() {
    return axios.get('/api/goods')
}
export default {
    namesapce: "goods", //model的命名空间，区分多个model
    state: [],  //初始状态
    reducers:{
        addGood(state,action){
            return [...state,
{title:action.payload.title}]
        },
        initGoods(state,action){
            return action.payload
        }
    }, //更新状态
    effects: { //副作用 异步操作
        *getList(action, { call, put }) {
            const res = yield call(getGoods);
            // type的名字 不需要命名空间
            yield put({ type: 'initGoods', payload:
res.data.result })
        }
    },
}
```

goods.js修改

```
import { Component } from 'react';
import styles from './goods.css';
import { connect } from "dva";
import { Card, Button } from "antd";

@connect(
```

```
  state => ({
    goodsList: state.goods  //获取指定命名空间的模型状态
  }),
  {
    addGood: title => ({
      type: 'goods/addGood', //action的type需要以命名空间为前缀+reducer名称
      payload: { title }
    }),
    getList: () => ({
      type: 'goods/getList',
    })
  }
)
export default class extends Component {
  componentDidMount() {
      //调用
    this.props.getList()
  }
  render() {
    return (
      <div className={styles.normal}>
        {/**/}
      </div>
    );
  }
}
```

加载状态:利用内置的dva-loading实现

- 获取加载状态,goods.js

```
@connect(
  state => ({
    loading:state.loading
```

```
  }),
  {
    ...
  }
)
export default class extends Component {
    render(){
        if(this.props.loading.models.goods){
            return <div>加载中......</div>
        }
        ....
    }
}
```