项目实战01

课堂目标

1. 掌握生成器函数 - generator
2. 掌握redux异步方案 - redux-saga

资源

1. redux-saga：中文、英文
2. generator

知识点

Generator

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

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

```
function* helloWorldGenerator() {
  yield 'hello';
  yield 'world';
  return 'ending';
}

var hw = helloWorldGenerator();

//执行
console.log(hw.next());
console.log(hw.next());
console.log(hw.next());
console.log(hw.next());
```

4. `yield`表达式后面的表达式，只有当调用`next`方法、内部指针指向该语句时才会执行，因此等于为 JavaScript 提供了手动的“惰性求值”（Lazy Evaluation）的语法功能。

```
var a = 0;
function* fun() {
  let aa = yield (a = 1 + 1);
  return aa;
}

console.log("fun0", a);
let b = fun();
console.log("fun", b.next());//注释下这句试试，比较下前后a的值
console.log("fun1", a);
```

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

实现登录

LoginPage

```
import React, {Component} from "react";
import {Redirect} from "react-router-dom/";
import {connect} from "react-redux";
import {login} from "../action/user";

export default connect(
  ({user}) => ({isLogin: user.isLogin, loading:
user.loading, err: user.err}),
  {
    login
  }
)(
  class LoginPage extends Component {
    constructor(props) {
      super(props);
      this.state = {name: ""};
    }

    nameChange = event => {
      this.setState({name: event.target.value});
    };
    render() {
      const {isLogin, loading, location, login, err}
= this.props;
      const {redirect = "/"} = location.state || {};
      if (isLogin) {
        return <Redirect to={redirect} />;
```

```
      }
      const {name} = this.state;
      return (
        <div>
          <h3>LoginPage</h3>
          <input type="text" value={name} onChange=
{this.nameChange} />
          <p className="red">{err.msg}</p>
          <button onClick={() => login({name})}>
            {loading ? "loading..." : "login"}
          </button>
        </div>
      );
    }
  }
);
```

action/user.js

async 函数是什么？一句话，它就是 Generator 函数的语法糖。

async 函数的实现原理，就是将 Generator 函数和自动执行器，包装在一个函数里。

```
import {
  LOGIN_SUCCESS,
  LOGIN_FAILURE,
  LOGOUT_SUCCESS,
  REQUEST,
  LOGIN_SAGA
} from "./const";
import LoginService from "../service/login";
```

```
// export const login = () => ({
//   type: LOGIN_SUCCESS
// });

export function login(userInfo) {
  return async function(dispatch) {
    dispatch({type: REQUEST});
    const res1 = await loginPromise(dispatch,
userInfo);
    if (res1) {
      getMoreUserInfo(dispatch, res1);
    }
  };
}

// 嵌套
// export const login = userInfo => dispatch => {
//   dispatch({type: REQUEST});
//   LoginService.login(userInfo).then(
//     res => {
//       // dispatch({
//       //   type: LOGIN_SUCCESS,
//       //   payload: {...userInfo, ...res}
//       // });
//       getMoreUserInfo(dispatch, {...userInfo,
...res});
//       return res;
//     },
//     err => {
//       dispatch({type: LOGIN_FAILURE, payload:
err});
//     }
```

```
//   );
// };

export const loginPromise = (dispatch, userInfo) =>
{
  return LoginService.login(userInfo).then(
    res => {
      return res;
    },
    err => {
      dispatch({type: LOGIN_FAILURE, payload: err});
    }
  );
};

const getMoreUserInfo = (dispatch, userInfo) => {
  return
LoginService.getMoreUserInfo(userInfo).then(
    res => {
      dispatch({
        type: LOGIN_SUCCESS,
        payload: {...userInfo, ...res}
      });
      return res;
    },
    err => {
      dispatch({type: LOGIN_FAILURE, payload: err});
    }
  );
};

export const logout = () => ({
  type: LOGOUT_SUCCESS
```

```
});
```

store/loginReducer

```
const userInit = {
  isLogin: false,
  userInfo: {id: null, name: "", score: 0},
  loading: false,
  err: {msg: ""}
};

// 定义修改规则
export const loginReducer = (state = {...userInit},
{type, payload}) => {
  switch (type) {
    case "REQUEST":
      return {...state, loading: true};
    case "LOGIN_SUCCESS":
      return {...state, isLogin: true, loading:
false, userInfo: {...payload}};
    case "LOGIN_FAILURE":
      return {...state, ...userInit, ...payload};
    case "LOGOUT_SUCCESS":
      return {...state, isLogin: false, loading:
false};
    default:
      return state;
  }
};
```

UserPage

```
import React, {Component} from "react";
import {connect} from "react-redux";
import {logout} from "../action/user";

@connect(({user}) => ({user}), {logout})
class UserPage extends Component {
  render() {
    const {user, logout} = this.props;
    const {id, name, score} = user.userInfo;

    return (
      <div>
        <h3>UserPage</h3>
        <p>id: {id}</p>
        <p>name: {name}</p>
        <p>score: {score}</p>
        <button onClick={logout}>logout</button>
      </div>
    );
  }
}
export default UserPage;
```

redux-saga

- 概述：`redux-saga` 是一个用于管理应用程序 Side Effect（副作用，例如异步获取数据，访问浏览器缓存等）的 library，它的目标是让副作用管理更容易，执行更高效，测试更简单，在处理故障时更容易。

- 地址：https://github.com/redux-saga/redux-saga
- 安装：**yarn add redux-saga**
- 使用：用户登录

在 `redux-saga` 的世界里，Sagas 都用 Generator 函数实现。我们从 Generator 里 yield 纯 JavaScript 对象以表达 Saga 逻辑。 我们称呼那些对象为 *Effect*。

你可以使用 `redux-saga/effects` 包里提供的函数来创建 Effect。

effects

effect 是一个 javascript 对象，里面包含描述副作用的信息，可以通过 yield 传达给 sagaMiddleware 执行。

在 redux-saga 世界里，所有的 effect 都必须被 yield 才会执行，所以有人写了 eslint-plugin-redux-saga 来检查是否每个 Effect 都被 yield。并且原则上来说，所有的 yield 后面也只能跟effect，以保证代码的易测性。

put

作用和 redux 中的 dispatch 相同。

```
yield put({ type: LOGIN_SUCCESS);
```

call与fork：阻塞调用和无阻塞调用

redux-saga 可以用 fork 和 call 来调用子 saga ，其中 fork 是无阻塞型调用，call 是阻塞型调用，即call是有阻塞地调用 saga 或者返回 promise 的函数。

take

等待 redux dispatch 匹配某个 pattern 的 action 。

```
function* loginSaga(props) {
    // yield takeEvery("login", loginHandle);
    // 等同于
    const action = yield take(LOGIN_SAGA);
    yield fork(loginHandle, action);
}
```

takeEvery

`takeEvery` 可以让多个 `saga` 任务并行被 fork 执行。

```
import {fork, take} from "redux-saga/effects"

const takeEvery = (pattern, saga, ...args) =>
fork(function*() {
  while (true) {
    const action = yield take(pattern)
    yield fork(saga, ...args.concat(action))
  }
})
```

redux-saga 使用了 ES6 的 Generator 功能，让异步的流程更易于读取，写入和测试。*（如果你还不熟悉的话，这里有一些介绍性的链接）* 通过这样的方式，这些异步的流程看起来就像是标准同步的 Javascript 代码。（有点像 `async`/`await`，但 Generator 还有一些更棒而且我们也需要的功能）。

不同于 redux-thunk，**你不会再遇到回调地狱了**，你可以很容易地测试异步流程并保持你的 action 是干净的，因此我们可以说 **redux-saga更擅长解决复杂异步这样的场景，也更便于测试。**

saga的方式实现路由守卫

1. 创建一个./action/userSaga.js处理用户登录请求

   call： 调用异步操作

   put：状态更新

   takeEvery：做saga监听

action/loginSaga.js

```
// 调用异步操作 call、
// 状态更新 （dispatch） put
// 做监听 take

import {
  call,
  put,
  // takeEvery,
  take,
```

```
  fork
} from "redux-saga/effects";
import {
  LOGIN_SUCCESS,
  LOGIN_FAILURE,
  LOGOUT_SUCCESS,
  REQUEST,
  LOGIN_SAGA
} from "./const";
import LoginService from "../service/login";

// worker saga

function* loginHandle(action) {
  yield put({type: REQUEST});
  try {
    const res1 = yield call(LoginService.login,
action.payload);
    const res2 = yield
call(LoginService.getMoreUserInfo, res1);
    yield put({type: LOGIN_SUCCESS, payload:
{...res1, ...res2}});
  } catch (err) {
    yield put({type: LOGIN_FAILURE, payload: err});
  }
}

// watcher saga
function* loginSaga(params) {
  yield takeEvery(LOGIN_SAGA, loginHandle);

  // while (true) {
  //   const action = yield take(LOGIN_SAGA);
```

```
  //   // call是一个会阻塞的effect，即generator在调用结束之前不能执行的或处理任何其他事情
  //   yield call(loginHandle, action);
  //   console.log("call", action); //sy-log
  //   // fork是无阻塞型任务，任务会在后台启动，调用者也可以继续它自己的流程，而不用等待被fork的任务结束
  //   // yield fork(loginHandle, action);
  //   // console.log("fork", action); //sy-log
  // }
}

export default loginSaga;

const takeEvery = (pattern, saga, ...args) =>
  fork(function*() {
    while (true) {
      const action = yield take(pattern);
      yield fork(saga, ...args.concat(action));
    }
  });
```

store/index.js

注册redux-saga

```
import {createStore, combineReducers,
applyMiddleware} from "redux";
// import thunk from "redux-thunk";
import {loginReducer} from "./loginReducer";
import createSagaMiddleware from "redux-saga";
import loginSaga from "../action/loginSaga";
```

```
const sagaMiddleware = createSagaMiddleware();

const store = createStore(
  combineReducers({user: loginReducer}),
  // applyMiddleware(thunk)
  applyMiddleware(sagaMiddleware)
);

sagaMiddleware.run(loginSaga);

export default store;
```

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

当有多个saga的时候，rootSaga.js

```
import {all} from "redux-saga/effects";
import loginSaga from "./loginSaga";

export default function* rootSaga(params) {
  yield all([loginSaga()]);
}
```

store/index.js中引用改成rootSaga即可：

```
...
sagaMiddleware.run(rootSaga);
```

action/user.js

```
export const logout = () => ({
  type: LOGOUT_SUCCESS
});

// saga
export const login = userInfo => ({
  type: LOGIN_SAGA,
  payload: userInfo
});
```

手动搭建项目

管理数据redux

路由管理react-router-dom

异步操作thunk或者saga

Routes

```
import React from "react";
import {BrowserRouter as Router, Route, Link,
Switch} from "react-router-dom";
import HomePage from "../pages/HomePage";
import UserPage from "../pages/UserPage";
import _404Page from "../pages/_404Page";
import LoginPage from "../pages/LoginPage";
import PrivateRoute from "../pages/PrivateRoute";
import BottomNav from "../components/BottomNav";
```

```
export const routes = [
  {
    path: "/",
    exact: true,
    component: HomePage
  },
  {
    path: "/user",
    component: UserPage,
    auth: PrivateRoute
  },
  {
    path: "/login",
    component: LoginPage
  },
  {
    component: _404Page
  }
];

export default function Routes(props) {
  return (
    <Router>
      {/* <Link to="/">首页</Link>
      <Link to="/user">用户中心</Link>
      <Link to="/login">登录</Link> */}

      <BottomNav />

      <Switch>
        {routes.map(Route_ =>
          Route_.auth ? (
```

```
            <Route_.auth key={Route_.path + "route"}
{...Route_} />
          ) : (
            <Route key={Route_.path + "route"}
{...Route_} />
          )
        )}
        {/* <Route exact path="/" component=
{HomePage} />
        <Route path="/login" component={LoginPage}
/>
        <PrivateRoute path="/user" component=
{UserPage} />
        <Route component={_404Page} /> */}
      </Switch>
    </Router>
  );
}
```

模拟登录

service/login.js

```
// 模拟登录接口
const LoginService = {
  login(userInfo) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        if (userInfo.name === "小明") {
          resolve({id: 123, name: "omg原来是小明"});
        } else {
```

```
            reject({err: {msg: "用户名或密码错误"}});
          }
        }, 1000);
      });
    },
    // 获取更多信息
    getMoreUserInfo(userInfo) {
      return new Promise((resolve, reject) => {
        setTimeout(() => {
          if (userInfo.id === 123) {
            resolve({score: "100"});
          } else {
            reject({msg: "获取详细信息错误"});
          }
        }, 1000);
      });
    }
};
export default LoginService;
```

回顾

项目实战01

课堂目标

作业

丰富自己的项目，预习umi、dva、antd。

下节课内容

掌握umi、dva、antd，用框架写React项目。