



A SIMPLE TODO LIST FUNCTION IMPLEMENTATION BASED ON REACT

ReactJs + Redux + Redux-saga + Jest

Abstract

My Todo project is similar to a memo, users can add tasks to do. When the user completes the task, he can click the check box to complete the task. If the task is no longer needed, you can click the delete button to delete the corresponding task. At the same time, the user can see the number of remaining unfinished tasks in the todo interface.

Ella Gao

273853065@qq.com

Catalog

1	<i>Project Description</i>	2
2	<i>Manual</i>	2
3	<i>function realization</i>	4
3.1	Project Catalog	4
3.2	Local Operation	7
3.3	Implementation	8
3.3.1	Project construction	8
3.3.2	Technical Principle	9
3.3.3	My Todo redux-sagas Business realization	13
4	<i>Unit Test</i>	20
5	<i>References</i>	24

1 Project Description

My Todo project is similar to a memo, users can add tasks to do. When the user completes the task, he can click the check box to complete the task. If the task is no longer needed, you can click the delete button to delete the corresponding task. At the same time, the user can see the number of remaining unfinished tasks in the todo interface.

2 Manual

- 1) Start the project locally (please refer to the function realization for details), enter the URL <http://localhost:3000> in the browser, you can see the main interface of the project, which consists of three parts, which are the project title, the number of tasks prompt and Task input box, as shown in Figure 1;

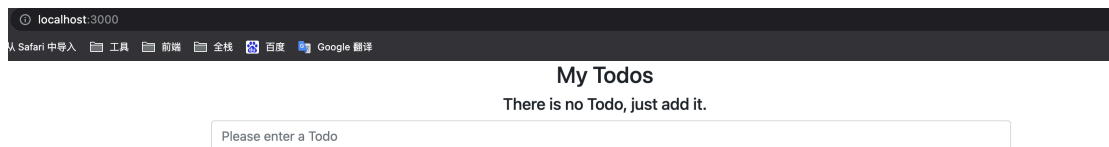


Figure 1

- 2) To add a task, enter the task name in the task input box, here is named "Event 1", as shown in Figure 2;

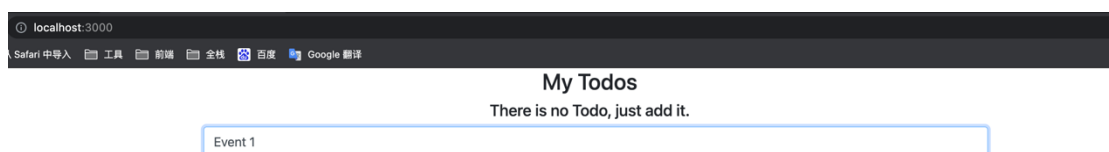


Figure 2

Click Enter to add a task, as shown in Figure 3, you can see that “Event 1” is successfully added, and a task with a check box will be displayed below the input box. The user can edit or delete the added task. Click the check box to indicate that the task has been completed, and the number of tasks will be updated synchronously according to user operations.

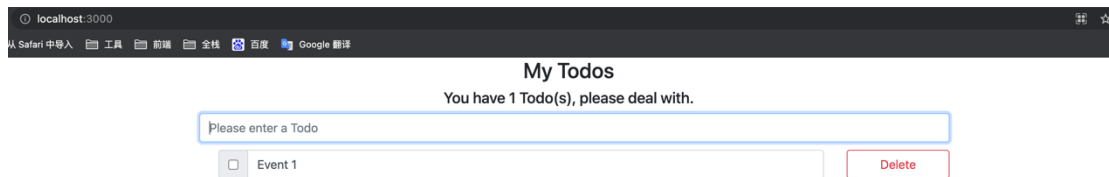


Figure 3

3) Edit task, select an already added task and edit the task box to realize the update task list function, as shown in Figure 4.

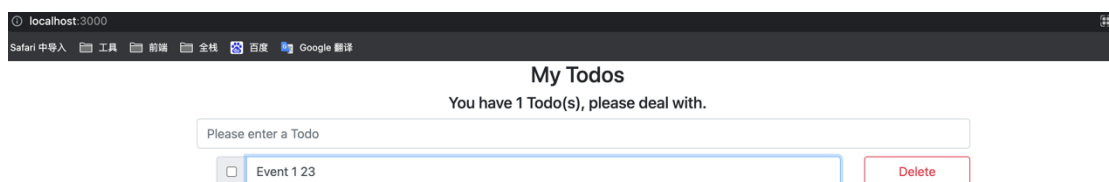


Figure 4

Click the check box of the task to complete the project, and the number of tasks will be updated simultaneously, as shown in Figure 5.

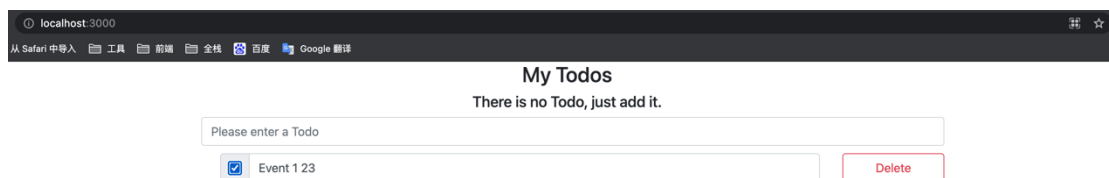


Figure 5

4) To delete a task, click the delete button of any task to delete the task, as

shown in Figure 6 and Figure 7.

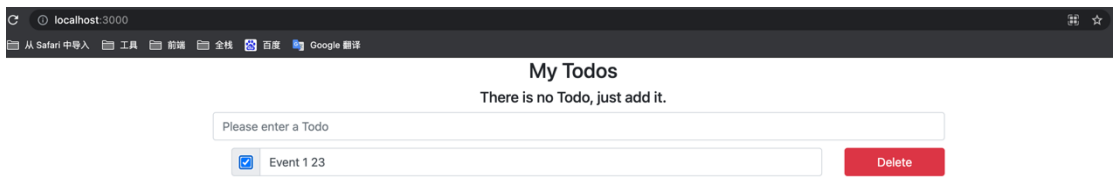


Figure 6

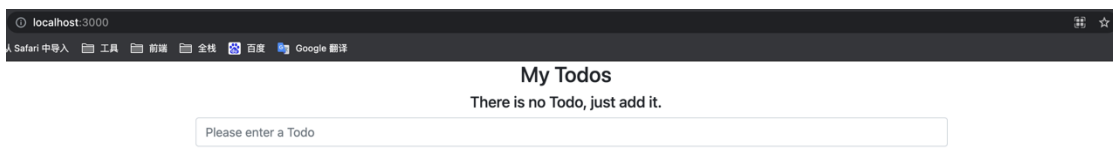


Figure 7

3 function realization

3.1 Project Catalog

The project directory structure is shown in Figure 8. The package.json file mainly configures the dependencies in the project, as well as the configuration of startup, build and test projects.

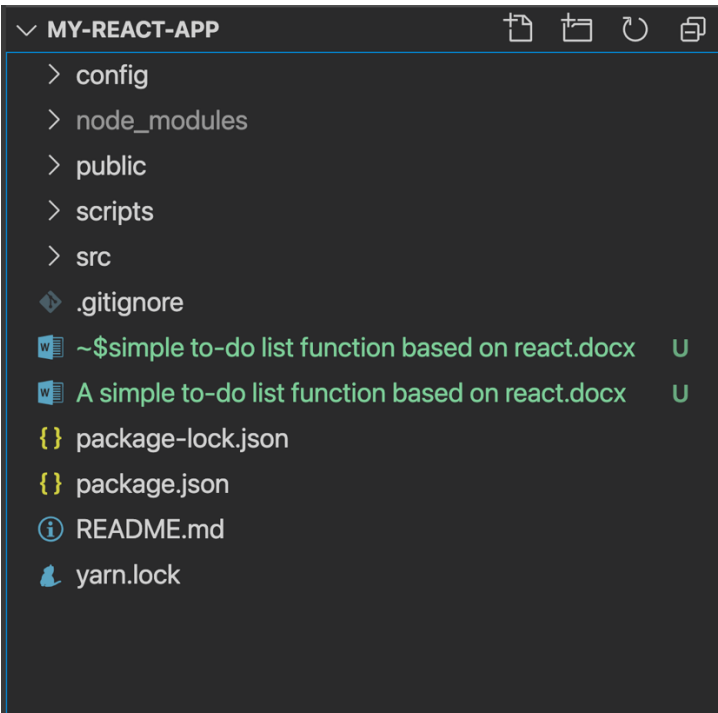


Figure 8

The config folder is a configuration folder, which contains webpack configuration, unit test jest configuration, environment information configuration and other files.

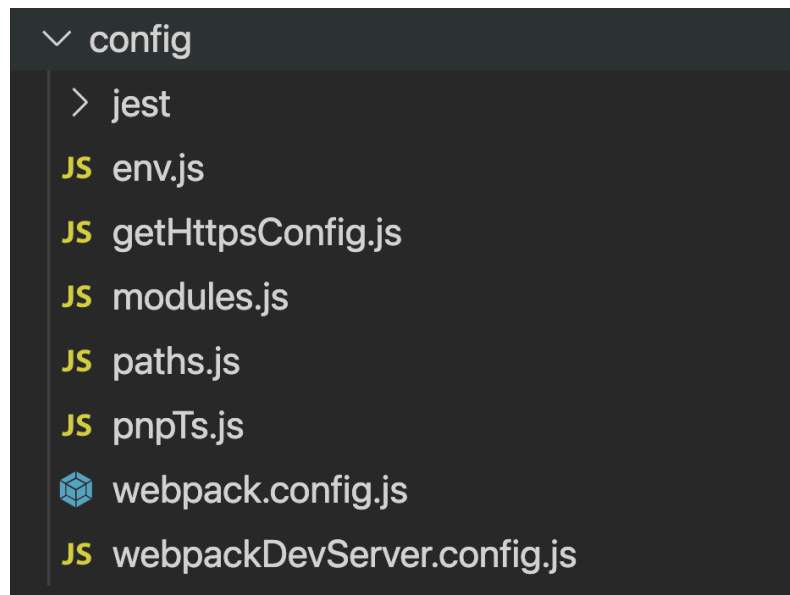


Figure 9

node_modules stores the dependent packages used in the project, please refer to the package.json file for details.

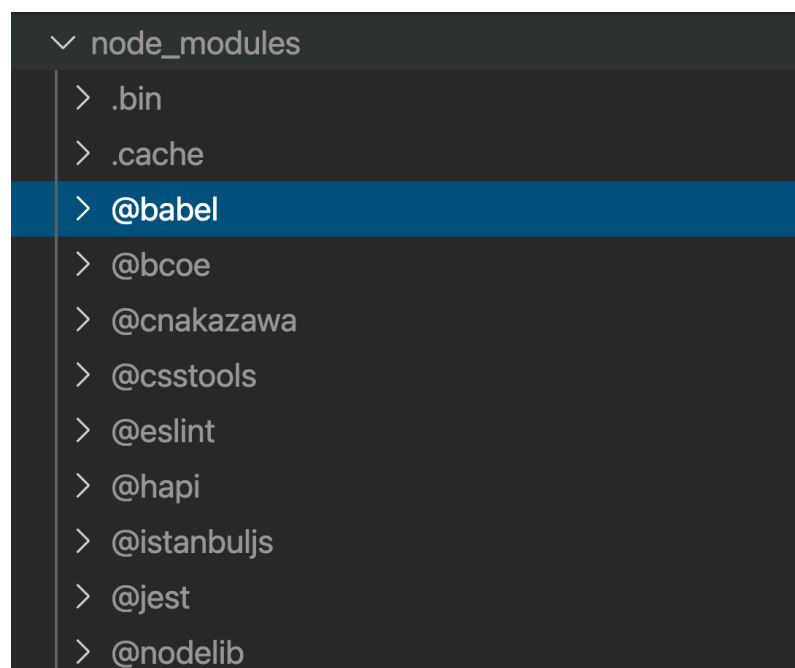


Figure 10

Public stores public files, where index.html is the entrance of react app.

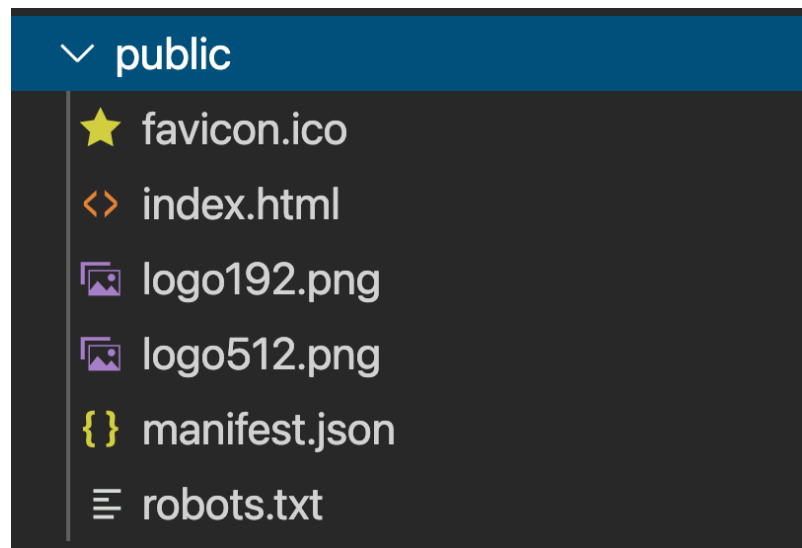


Figure 11

The files in the scripts folder correspond to the scripts in the package.json file and are mainly used to start, build and test the project.

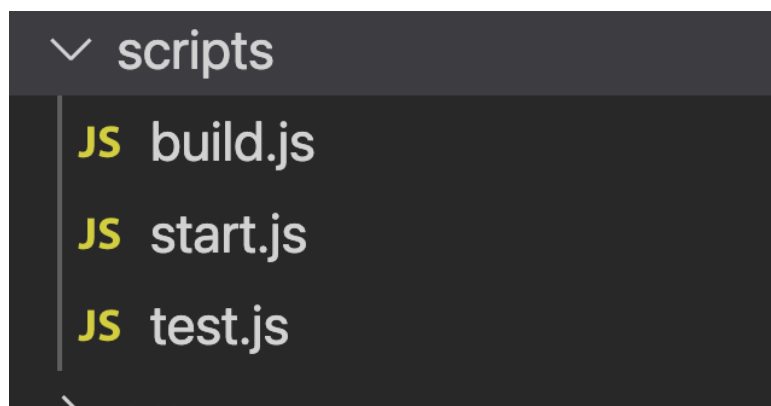


Figure 12

The src file is mainly the business code of the project. The project uses react, redux, redux-saga. The project folder is divided into actions, common, components, reducers and sagas. It also provides the unit test file of the App, App.test.js.

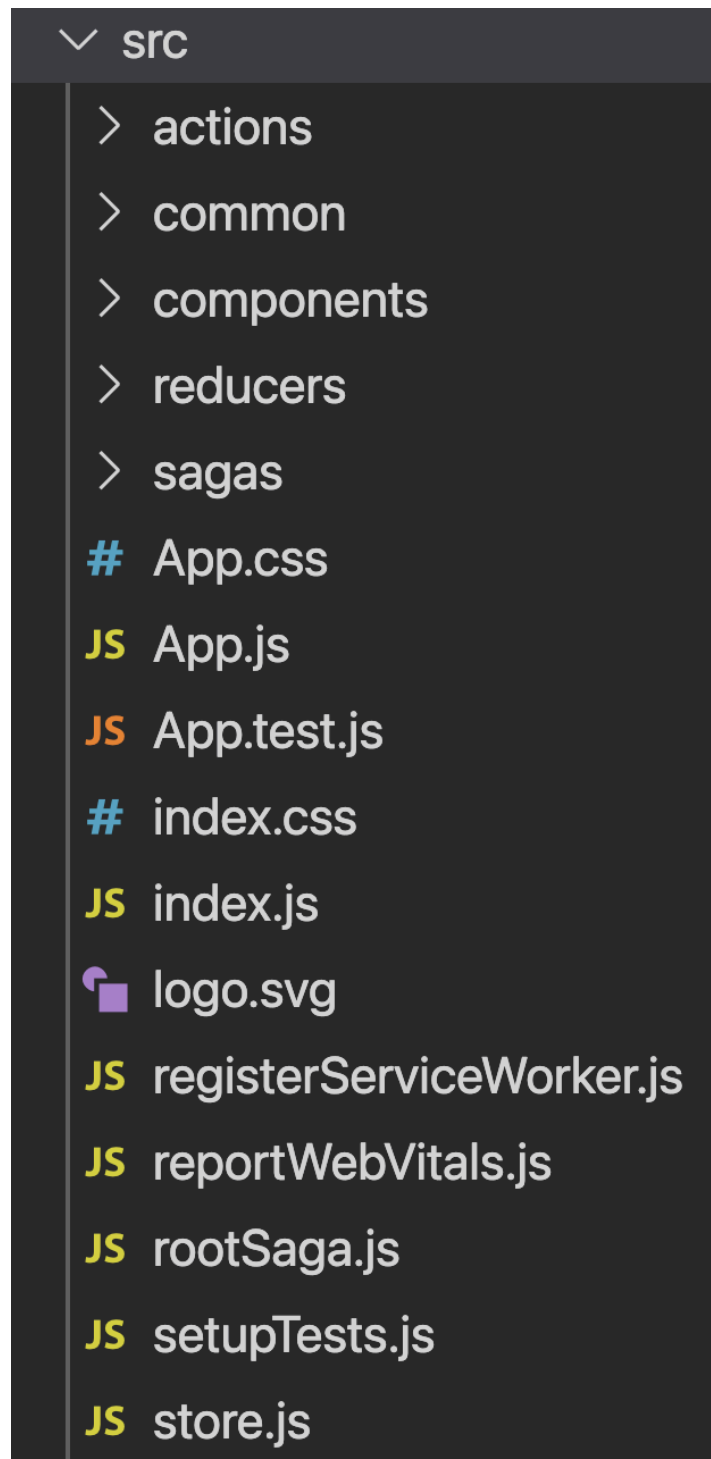


Figure 13

3.2 Local Operation

I uploaded the project to github, the project URL is:

<https://github.com/273853065/my-react-app>

The code branch is master.

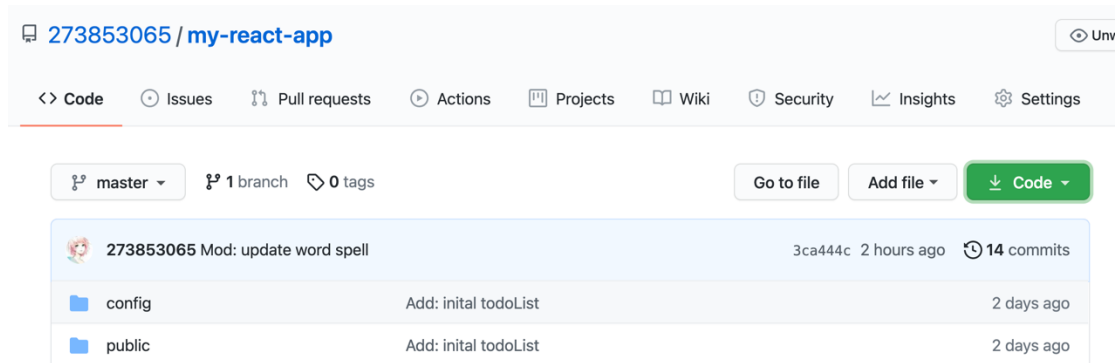


Figure 14

Clone the code first:

```
git clone git@github.com:273853065/my-react-app.git
```

Download the corresponding dependency package and execute it locally:

```
npm install
```

Start up:

```
yarn start
```

Startup project, <http://localhost:3000/>

3.3 Implementation

3.3.1 Project construction

The project was created using Create React App. This tool encapsulates most of the build components in node_modules, such as webpack. In order to better understand the project as a whole, I executed the following commands in the project:

```
npm run eject
```

Note: This is a one-way operation. Once ejected, you cannot return!

If you are not satisfied with the build tools and configuration choices, you can

always pop up. This command will remove a single build dependency from your project.

Instead, it will copy all configuration files and transitive dependencies (webpack, Babel, ESLint, etc.) to your project as dependencies in package.json. Technically speaking, for front-end applications that generate static packages, the distinction between dependencies and development dependencies is very arbitrary.

In addition, it used to cause problems with some hosting platforms that did not have development dependencies installed (so it was impossible to build the project on the server or test it before deployment). You can freely rearrange your dependencies in package.json as needed.

3.3.2 Technical Principle

1) The data flow in redux roughly is:

UI ———> *action (plain)* ———> *reducer* ———> *state* ———> *UI*

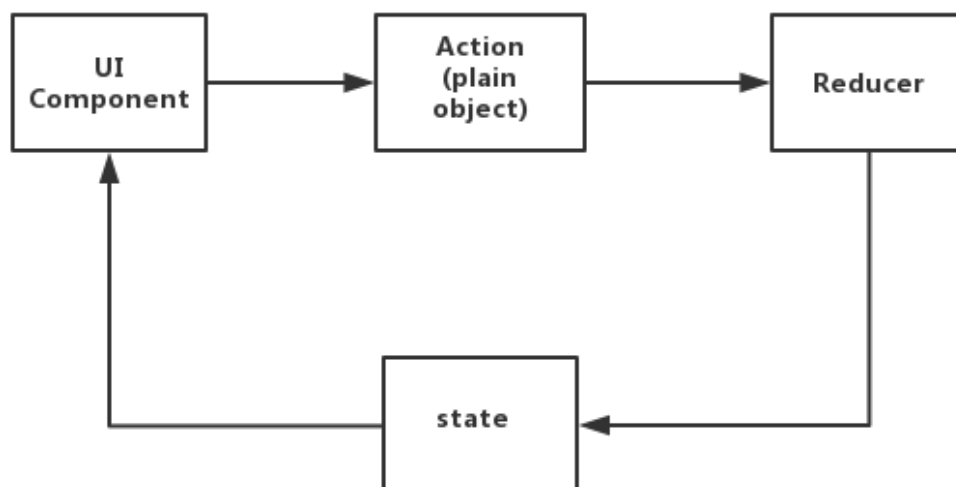


Figure 15

Redux follows the rules of functional programming. In the above data flow, action is a plain object and reducer is a pure function. For operations that are synchronized and have no side effects, the above data flow can manage data. So as to control the purpose of updating the view layer.

If there is a side-effect function, then we need to process the side-effect function first, and then generate the original js object. How to deal with side-effect operations? In redux, choose to use middleware to handle side-effects between issuing action and reducer processing function.

The data flow after redux adds middleware to handle side effects is roughly as follows:

UI—>action(side function)—>middleware—>action(plain)—>reducer—>state—>UI

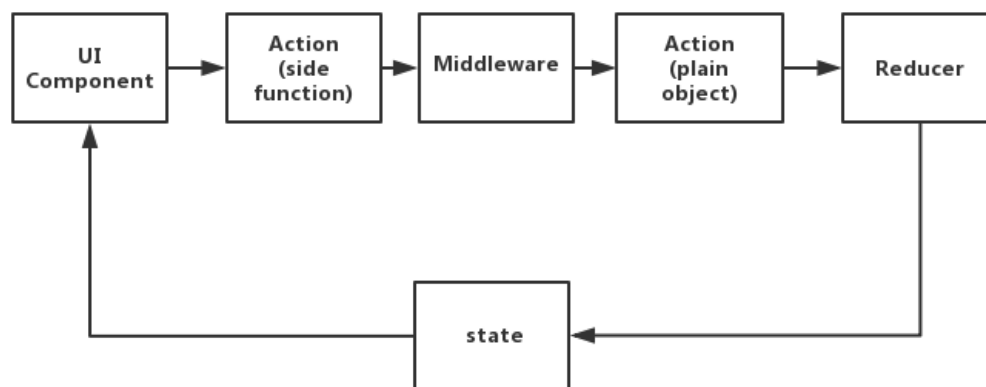


Figure 16

Add middleware processing between the side-effect action and the original action. From the figure, we can also see that the role of middleware is:

Convert the asynchronous operation to generate the original action, so that the reducer function can process the corresponding action, thereby changing

the state and updating the UI.

2) redux-thunk

In redux, thunk is a middleware given by the author of redux, implemented as follows:

```
function createThunkMiddleware(extraArgument) {  
  return ({ dispatch, getState }) => next => action => {  
    if (typeof action === 'function') {  
      return action(dispatch, getState, extraArgument);  
    }  
    return next(action);  
  };  
}
```

```
const thunk = createThunkMiddleware();  
thunk.withExtraArgument = createThunkMiddleware;  
export default thunk;
```

Determine the type of action. If the action is a function, call this function. The calling steps are:

```
action(dispatch, getState, extraArgument);
```

It is found that the actual parameters are dispatch and getState, so when the action is defined as a thunk function, the general parameters are dispatch and getState.

3) redux-thunk Shortcomings

The shortcomings of redux-thunk are also very obvious. The thunk only executes the function and does not care what is in the body of the function. That is to say, thunk allows redux to accept functions as actions, but the internals of the functions can be varied. For example, the following is an action corresponding to an asynchronous operation to obtain a list of products:

```
export default ()=>(dispatch)=>{

  fetch('/api/goodList',{ //fetch returns a promise

    method: 'get',

    dataType: 'json',

  }).then(function(json){

    var json=JSON.parse(json);

    if(json.msg==200){

      dispatch({type:'init',data:json.data});

    }

  },function(error){

    console.log(error);

  });

};
```

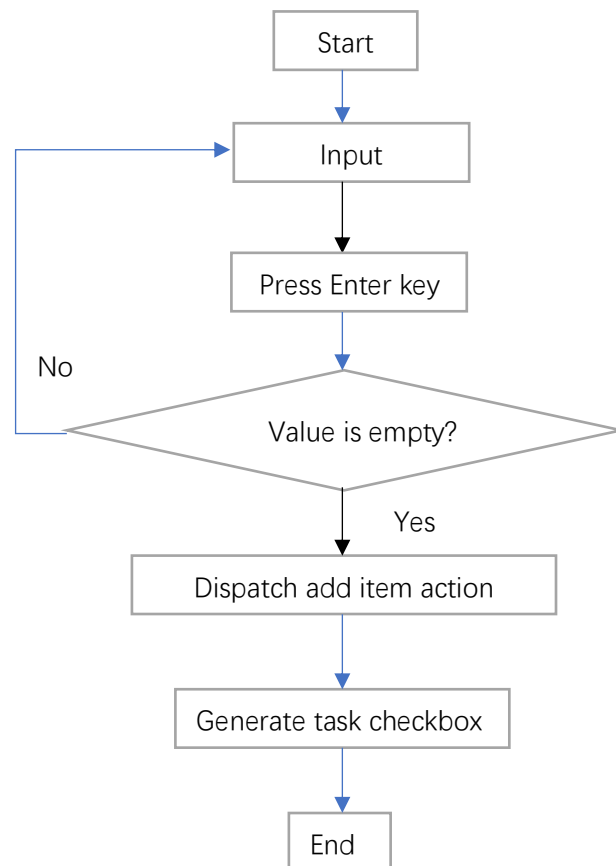
From this action with side effects, we can see that the inside of the function is extremely complicated. If you need to define an action for each asynchronous operation in this way, obviously the action is not easy to maintain.

Reasons why actions are not easy to maintain:

- The form of action is not uniform
- Asynchronous operations are too scattered, scattered in various actions

3.3.3 My Todo redux-sagas Business realization

Take adding a task list as an example, the business flowchart is as follows:



Add task list functions include:

- 1) Enter the task name in the new input box, press the Enter button to dispatch the new task action, and clear the value of the new task input box
- 2) Execute new task event, generate task check box
- 3) Enter the task name in the new input box, press the Enter button to

dispatch the new task action, and clear the value of the new task input box

3.3.3.1 Create the input.js file to realize the new task input box

Use the connect method to establish the association between state and action

Path: /src/components/Input.js

```
class Input extends Component {  
  constructor(props) {  
    super(props)  
    this.state = {  
      value: ''  
    }  
  }  
  
  addTodo(value) {  
    this.props.addItem(value)  
    this.setState({ value: '' })  
  }  
  
  render() {  
    return (  
      <div>  
        <div className="col-sm-12 mb10 pr0">
```

```

    <input
      id="todo_input"

      className="form-control"

      placeholder="Please enter a Todo"

      value={this.state.value}

      onChange={(e) => this.setState({ value: e.target.value })}

      onKeyDown={e => {
        if (e.key === "Enter") {
          let title = e.target.value;

          if (title.length > 0) {
            this.addToDo(title);
          }
        }
      }}
    />
  </div>
</div>

)
}

}

function mapStateToProps(state) {

```



```

    return {
      text: ""
    }
  }
}

function mapDispatchToProps(dispatch) {
  return {
    addItem: bindActionCreators(addItem, dispatch)
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(Input)

```

3.3.3.2 Execute new task event, generate task check box

Create an inputSag.js file

Path: /src/sagas/inputSaga.js

```

export const delay = ms => new Promise(resolve => setTimeout(resolve, ms))

export function* addItem(value) {
  try {
    return yield call(delay, 500)
  } catch (err) {
    yield put({type: actionTypes.ERROR})
  }
}

```

```

export function* addItemFlow() {
  while (true) {
    let request = yield take(actionTypes.ADD_ITEM)

    let response = yield call(addItem, request.value)

    let tempList = yield select(state => state.getTodoList.list)

    let list = []

    list = list.concat(tempList)

    const tempObj = {}

    tempObj.title = request.value

    tempObj.id = list.length

    tempObj.finished = false

    list.push(tempObj)

    yield put({
      type: actionTypes.UPDATE_DATA,
      data: list
    })
  }
}

```

Use redux-saga middleware in redux

Create a new store.js file

Create store, associate reducers and saga middleware

Path: /src/store.js

```

const sagaMiddleware = createSagaMiddleware()

const store = createStore(
  reducers,
  applyMiddleware(sagaMiddleware)
)

sagaMiddleware.run(rootSaga)

export default store

```

Use Provider to establish root component and store association

Path: /src/index.js

```

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

```

Create actions

Path: /src/common/actionTypes.js

```

const actionTypes = {
  ADD_ITEM: 'ADD_ITEM'
}

```

Path: /src/actions/index.js

```

export {actionTypes}

import { actionTypes } from '../common/actionTypes'

```

```
export function addItem(value) {  
  
  return {  
  
    type: actionTypes.ADD_ITEM,  
  
    value  
  
  }  
}
```

Get the list of tasks that have been added

Path: /src/reducers/list.js

```
const initialState = {  
  
  list: []  
  
}  
  
function getTodoList(state = initialState, action) {  
  
  switch (action.type) {  
  
    case actionTypes.UPDATE_DATA:  
  
      return {  
  
        ...state,  
  
        list: action.data  
  
      }  
  
    default:  
  
      return state  
  
  }  
}
```

```
export default getTodoList
```

Multiple reducers need to use combineReducers

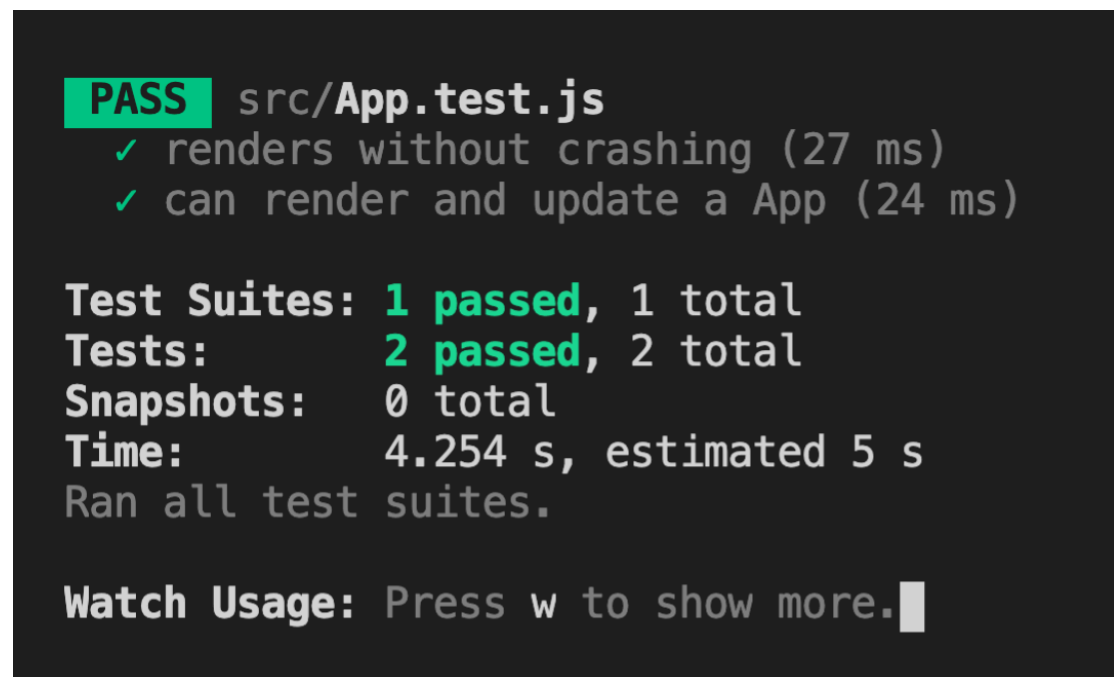
Path: /src/reducers/index.js

```
const reducer = combineReducers({  
  getTodoCount,  
  getTodoList  
})  
  
export default reducer
```

4 Unit Test

The project unit test needs to be entered on the command line:

```
yarn test
```



```
PASS src/App.test.js  
  ✓ renders without crashing (27 ms)  
  ✓ can render and update a App (24 ms)  
  
Test Suites: 1 passed, 1 total  
Tests:      2 passed, 2 total  
Snapshots:  0 total  
Time:       4.254 s, estimated 5 s  
Ran all test suites.  
  
Watch Usage: Press w to show more.
```

Figure 17

Output unit test report, input command:

```
yarn my-test
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	36.18	17.07	43.33	35.66	
src	17.02	0	11.11	17.02	
App.js	100	100	100	100	
index.js	0	100	100	0	10-16
...eWorker.js	0	0	0	0	11-105
...bVitals.js	0	0	0	0	1-8
rootSaga.js	100	100	100	100	
store.js	100	100	100	100	
src/actions	25	100	25	25	
index.js	25	100	25	25	11-25
src/common	100	100	100	100	
...onTypes.js	100	100	100	100	
src/components	67.74	30	60	70	
Header.js	83.33	25	66.67	100	8-12
Input.js	100	50	100	100	31-33
List.js	30.77	0	30	30.77	10-18,27-51
src/reducers	77.78	66.67	100	77.78	
header.js	75	66.67	100	75	10
index.js	100	100	100	100	
list.js	75	66.67	100	75	10
src/sagas	28.33	100	56.25	25	
inputSaga.js	42.11	100	80	37.5	10,18-26
listSaga.js	21.95	100	45.45	19.44	...33,41-47,58-71
Test Suites: 1 passed, 1 total					
Tests: 2 passed, 2 total					
Snapshots: 0 total					
Time: 6.58 s					
Ran all test suites.					

Figure 18

Add the ***.test.js** file to the module that needs to be tested, the unit testing tool Jest will automatically identify the test case and perform the test based on the assertion.

Redux-sagas project unit test:

Component Test

Take the test App component render, componentDidmount, componentDidUpdate as an example, You can write test cases based on component logic.

Path: /src/App.test.js

```

//Prepare a component for the assertion,

//wrap the code to be rendered and perform the update when act() is called.

//This will bring the test closer to how React works in the browser.

let container = null;

beforeEach(() => {

    // Create a DOM element as the rendering target

    container = document.createElement('div');

    document.body.appendChild(container);

});

afterEach(() => {

    // Clean up on exit

    unmountComponentAtNode(container);

    container.remove();

    container = null;

});

//smoking test

it('renders without crashing', () => {

    render(<Provider store={store}>

        <App />

    </Provider>, container);

});

//component unit test

```

```

it('can render and update a App', () => {

  //test render and componentDidMount

  act(() => {

    render(<Provider store={store}>

      <App />

    </Provider>, container);

  });

  const input = container.querySelector('#todo_input');

  const h5 = container.querySelector('h5');

  expect(h5.textContent).toBe('There is no Todo, just add it.');
```

//test input funciton

```

  let lastValue = input.value;

  input.value = 'testtest';

  let tracker = input._valueTracker;

  if (tracker) {

    tracker.setValue(lastValue);

  }

  // text render and componentDidUpdate

  act(() => {

    // You need to pass {bubbles: true} in each event created to reach the React
listener,
```

// because React will automatically delegate the event to root.


```

    input.dispatchEvent(new InputEvent('input', { bubbles: true }));

  });

  expect(input.value).toBe('testtest');

  act(() => {

    input.focus();

    input.dispatchEvent(new KeyboardEvent('keydown', {

      ctrlKey: false,

      metaKey: false,

      altKey: false,

      which: 13,

      keyCode: 13,

      key: 'Enter',

      code: 'Enter',

      bubbles: true

    }));

  });

  expect(input.value).toBe("");

});

```

The use case writing of saga test, selector test and utils test is still under study.

5 References

- ◆ Use Jest to unit test the React family bucket (react-saga, redux-actions, reselect)

(<https://juejin.cn/post/6844903703128834062>)

- ◆ Testing skills (<https://reactjs.bootcss.com/docs/testing-recipes.html>)
- ◆ Record a difficult but interesting problem solving experience-React input
(<https://juejin.cn/post/6844904128305430541>)
- ◆ Simulate login to react page (<https://www.jianshu.com/p/78f5a4baf88c>)
- ◆ Testing Components in React Using Jest: The Basics
(<https://code.tutsplus.com/articles/testing-components-in-react-using-jest-the-basics--cms-28934>)
- ◆ Create a new React application
(<https://react.docschina.org/docs/create-a-new-react-app.html#create-react-app>)