



一个基于 REACT 的简单待办列表功能实现

ReactJs + Redux + Redux-saga + Jest

摘要

My Todo 项目类似于一个备忘录，用户可以添加要做的任务。当用户完成该任务时，可以点击复选框，完成任务。如果该任务已经不再需要，可以点击删除按钮，删除对应任务。同时用户可以在 todo 界面中看到剩余未完成任务数量。

Ella Gao

273853065@qq.com

目录

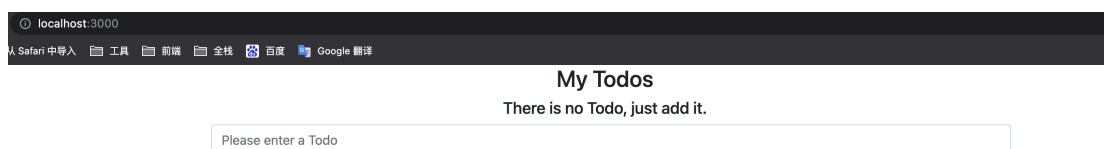
1	项目描述.....	2
2	操作手册.....	2
3	功能实现.....	4
3.1	项目目录	4
3.2	本地运行	7
3.3	实现.....	8
3.3.1	项目搭建.....	8
3.3.2	技术原理.....	9
3.3.3	My Todo redux-sagas 业务实现	12
4	单元测试.....	20
5	参考资料.....	24

1 项目描述

My Todo 项目类似于一个备忘录，用户可以添加要做的任务。当用户完成该任务时，可以点击复选框，完成任务。如果该任务已经不再需要，可以点击删除按钮，删除对应任务。同时用户可以在 todo 界面中看到剩余未完成任务数量。

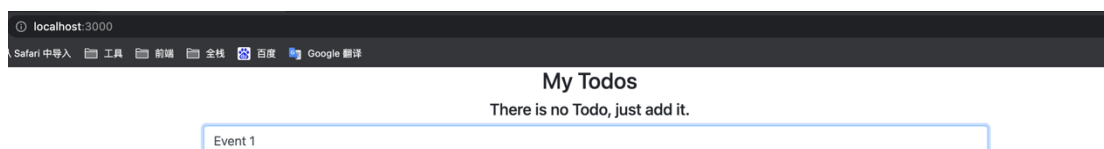
2 操作手册

- 1) 在本地启动项目（具体请参考功能实现），在浏览器中输入连接 <http://localhost:3000/>，可以看到项目主界面，由三部分组成，分别为项目标题、任务数量提示及任务输入框，如图一；



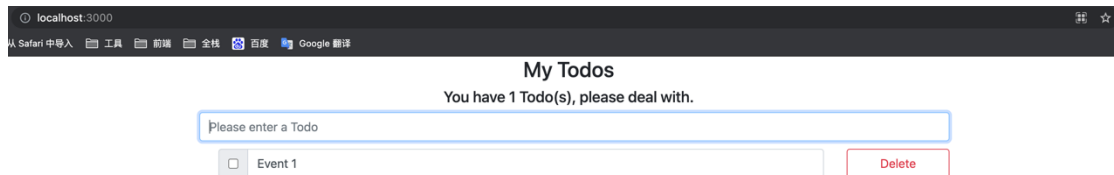
图一

- 2) 添加任务，在任务输入框中输入任务名称，这里命名为“Event 1”，如图二；



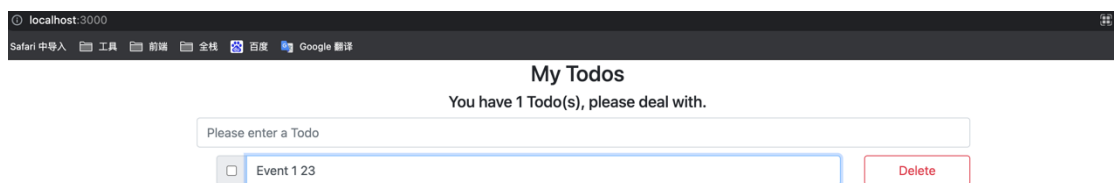
图二

点击回车添加任务，如图三，可以看到 Event 1 添加成功，会在输入框下方显示一个带复选框的任务，用户可以对添加的任务进行编辑，也可以对添加的任务进行删除操作，点击复选框表示任务已完成，同时任务数量提示也会根据用户操作同步更新。



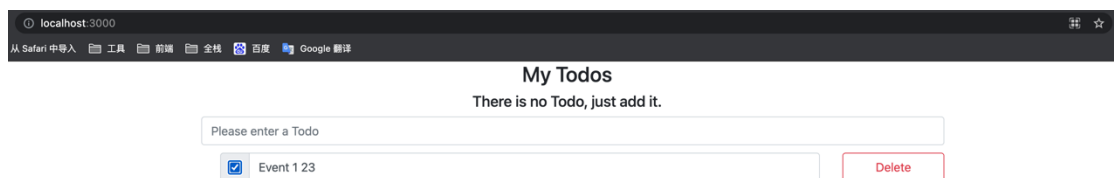
图三

- 3) 编辑任务，选中某一条已经添加的任务，对任务框进行编辑，可以实现更新任务列表功能，如图四。



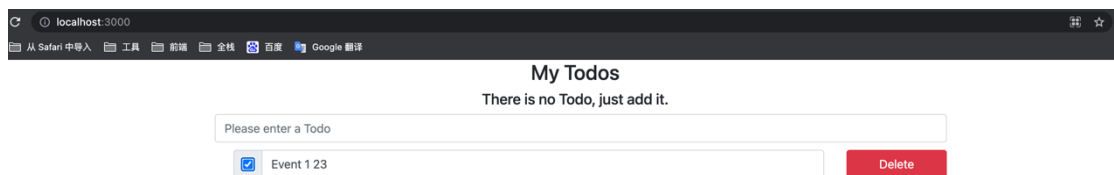
图四

- 点击该任务的复选框，完成该项目，同时任务数量提示也会同步更新，如图五。

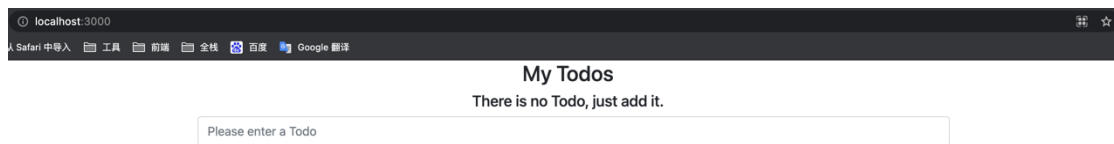


图五

- 4) 删除任务，点击任意一条任务的删除按钮，就可以删除该条任务，如图六、图七。



图六

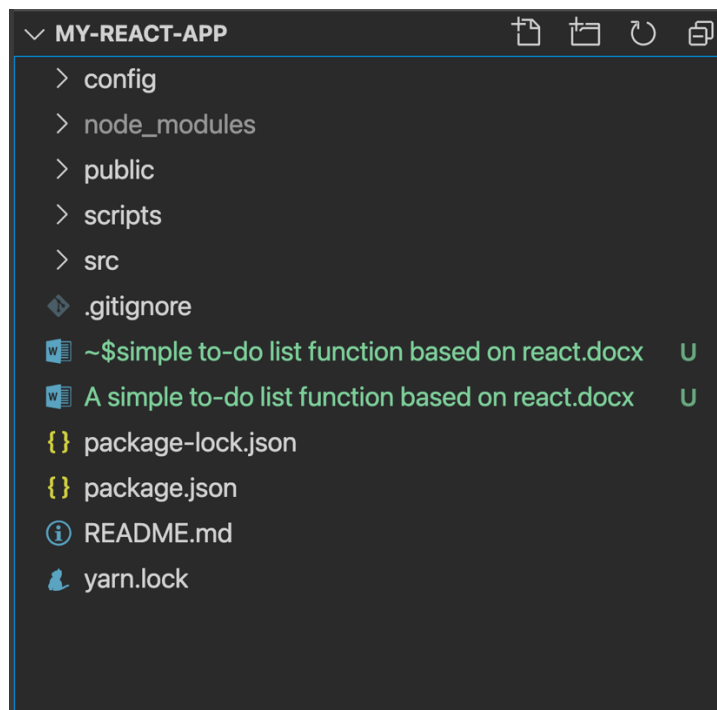


图七

3 功能实现

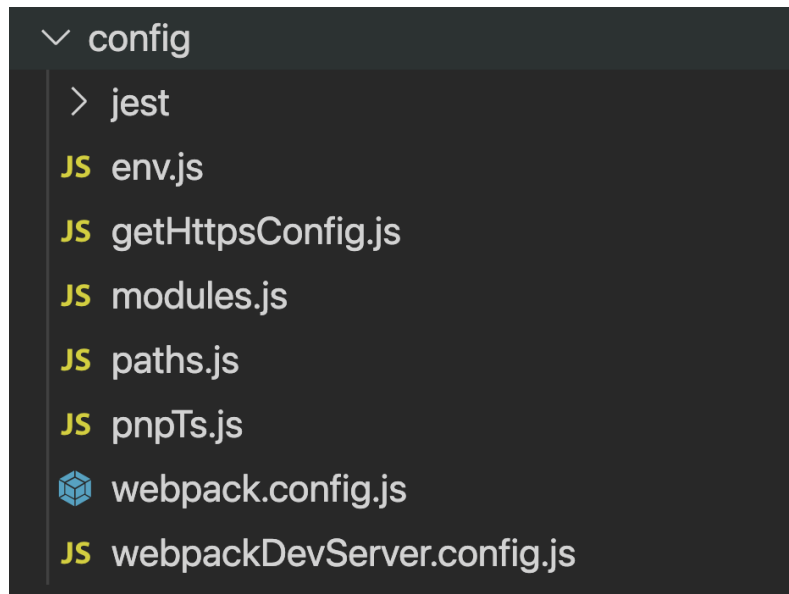
3.1 项目目录

该项目目录结构如图八所示, package.json 文件主要配置项目中的依赖, 以及启动、构建及测试项目相关配置。



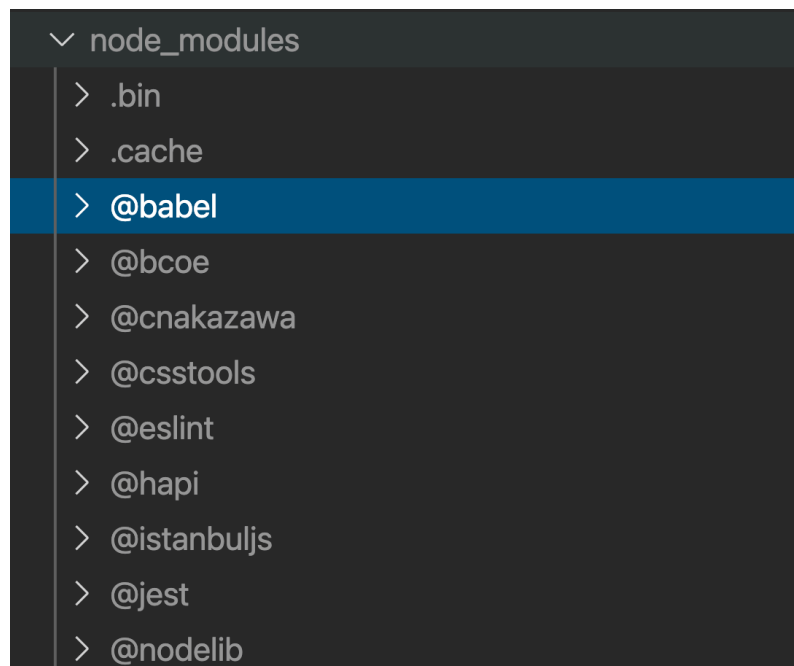
图八

config 文件夹为配置文件夹, 其中包含 webpack 配置, 单元测试 jest 配置, 环境信息配置等文件。



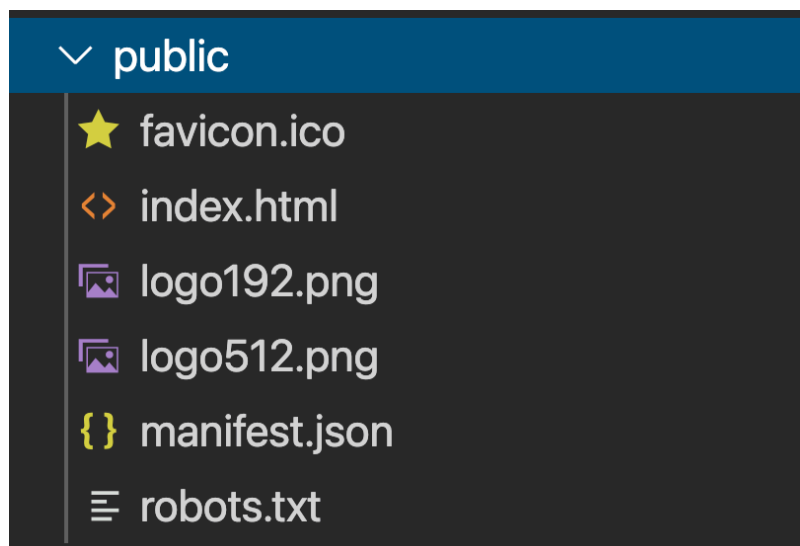
图九

node_modules 存放该项目中使用到的依赖包, 具体参考 package.json 文件。



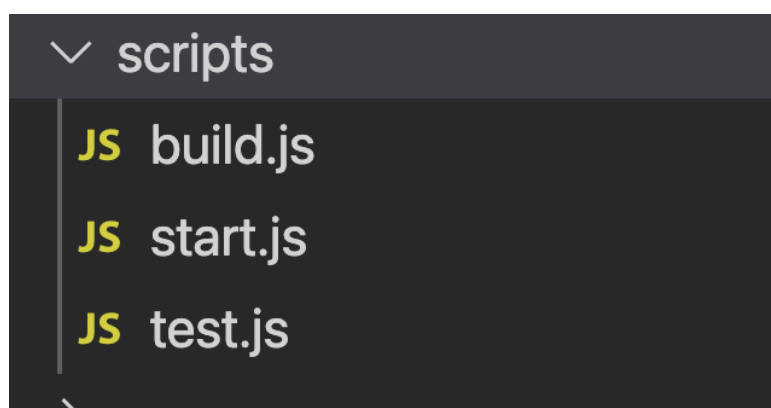
图十

public 存放公共文件, 其中 index.html 为 react app 的入口。



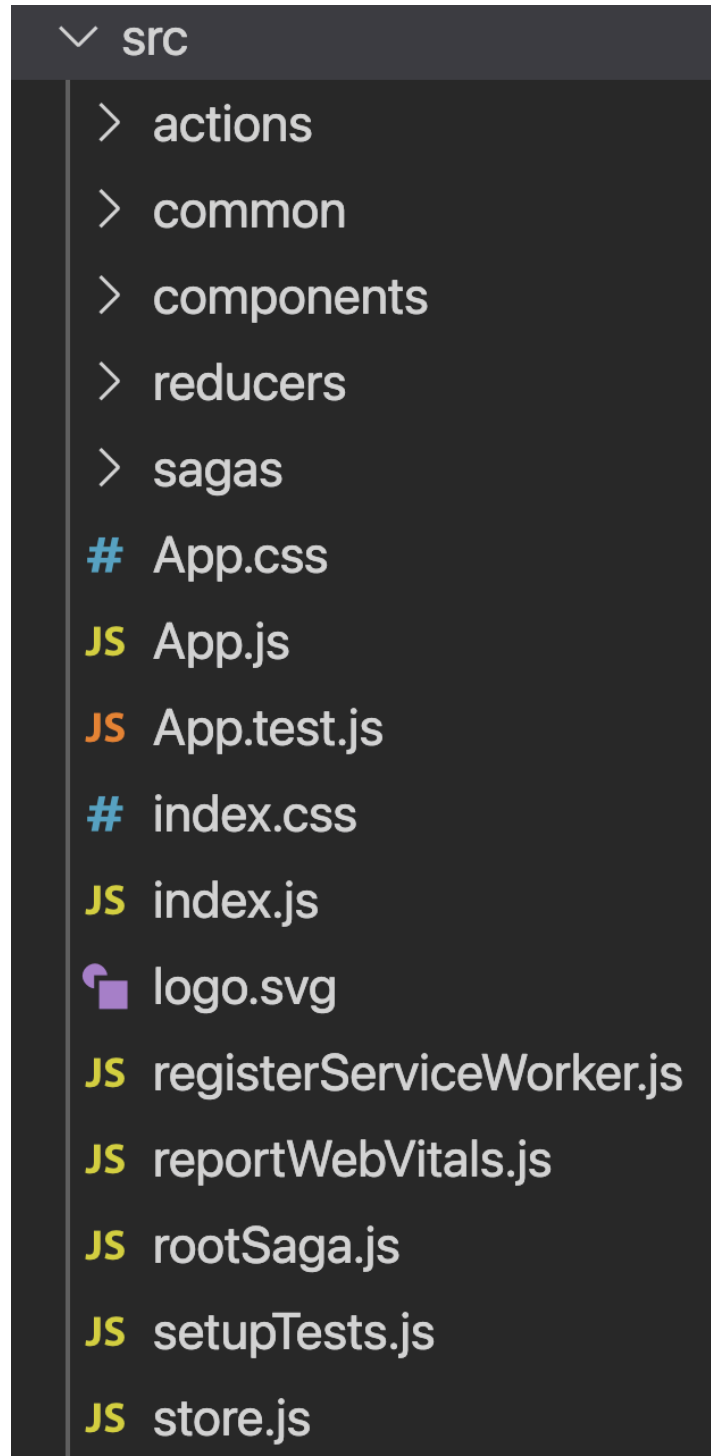
图十一

scripts 文件夹中的文件，对应 package.json 文件中的 scripts，主要用于启动，构建及测试项目。



图十二

src 文件主要为项目业务代码，该项目使用的 react, redux, redux-saga, 项目文件夹分为 actions、common、components、reducers 及 sagas, 同时提供了 app 的单元测试文件, App.test.js。



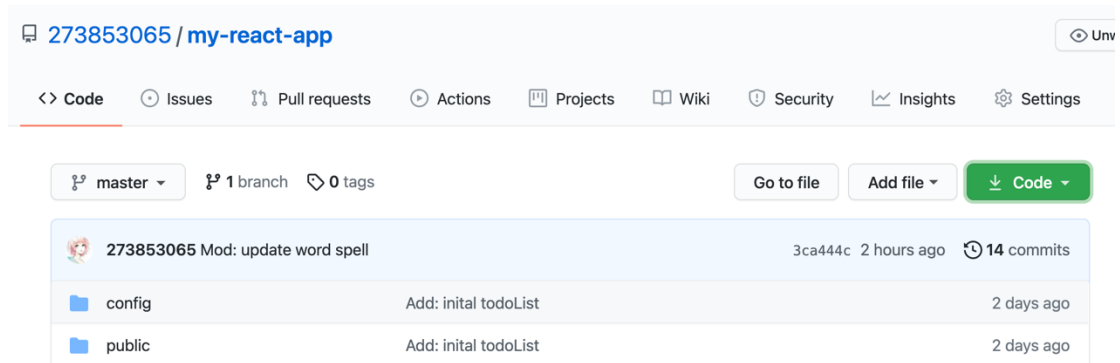
图十三

3.2 本地运行

我将项目上传到了 github 上，项目地址为：

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

代码分支为 master。



图十四

首先下载代码：

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

下载对应的依赖包，在本地执行：

```
npm install
```

启动：

```
yarn start
```

启动项目，<http://localhost:3000/>

3.3 实现

3.3.1 项目搭建

项目使用 Create React App 创建，该工具将大部分构建组件都封装在了 node_modules 中，例如 webpack，为了更了解项目整体，我在项目中执行了如下命令：

```
npm run eject
```

注意：这是一种单向操作。一旦弹出，就无法返回！

如果您对构建工具和配置选择不满意，可以随时弹出。此命令将从您的项目中删除单个构建依赖项。

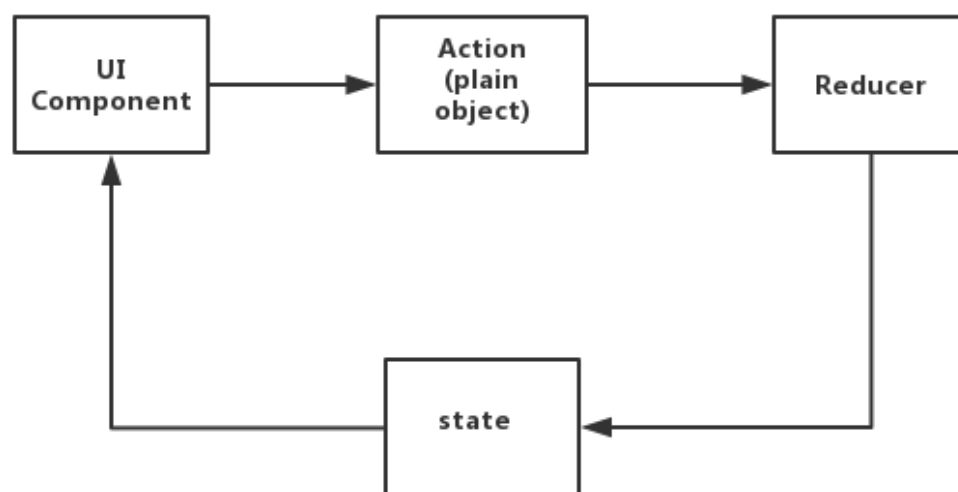
相反，它会将所有配置文件和可传递依赖项（webpack、Babel、ESLint 等）作为 package.json 中的依赖项复制到您的项目中。从技术上讲，对于生成静态包的前端应用程序，依赖项和开发依赖项之间的区别是非常随意的。

此外，它曾经导致某些未安装开发依赖项的托管平台出现问题（因此无法在服务器上构建项目或在部署前对其进行测试）。您可以根据需要在 package.json 中自由地重新排列您的依赖项。

3.3.2 技术原理

1) redux 中的数据流大致是：

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



图十五

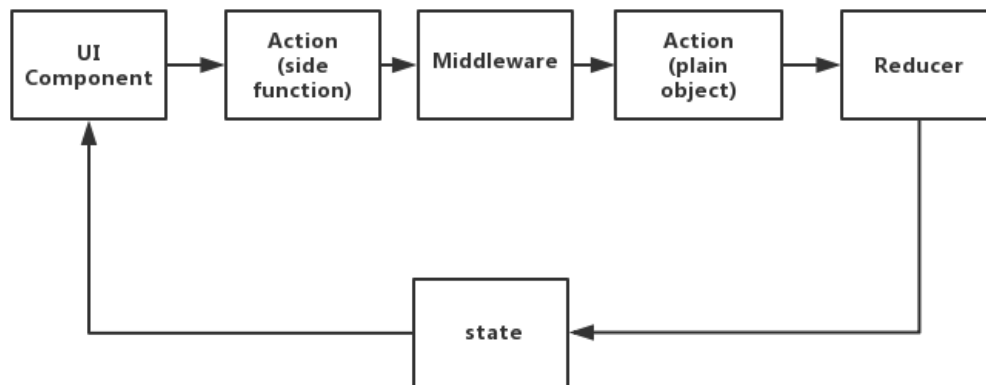
redux 是遵循函数式编程的规则，上述的数据流中，action 是一个原始 js 对象（plain object）且 reducer 是一个纯函数，对于同步且没有副作用的操作，上述的数据流起到可以管理数据，从而控制视图层更新的目的。

如果存在副作用函数，那么我们需要首先处理副作用函数，然后生成原始的

js 对象。如何处理副作用操作，在 redux 中选择在发出 action，到 reducer 处理函数之间使用中间件处理副作用。

redux 增加中间件处理副作用后的数据流大致如下：

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



图十六

在有副作用的 action 和原始的 action 之间增加中间件处理，从图中我们也可以看出，中间件的作用就是：

转换异步操作，生成原始的 action，这样，reducer 函数就能处理相应的 action，从而改变 state，更新 UI。

2) redux-thunk

在 redux 中，thunk 是 redux 作者给出的中间件，实现如下：

```
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;

```

判别 action 的类型，如果 action 是函数，就调用这个函数，调用的步骤为：

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

发现实参为 dispatch 和 getState，因此在定义 action 为 thunk 函数是，一般形参为 dispatch 和 getState。

3) redux-thunk 的缺点

redux-thunk 的缺点也是很明显的，thunk 仅仅做了执行这个函数，并不在乎函数主体内是什么，也就是说 thunk 使得 redux 可以接受函数作为 action，但是函数的内部可以多种多样。比如下面是一个获取商品列表的异步操作所对应的 action：

```

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

  fetch('/api/goodList',{ //fetch 返回的是一个 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);  
  });  
};
```

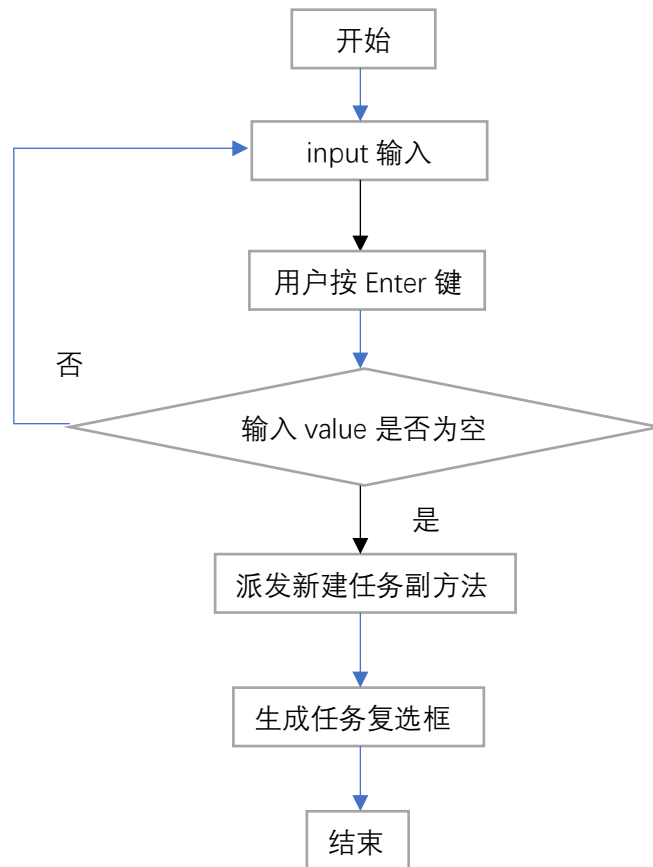
从这个具有副作用的 action 中，我们可以看出，函数内部极为复杂。如果需要为每一个异步操作都如此定义一个 action，显然 action 不易维护。

action 不易维护的原因：

- action 的形式不统一
- 就是异步操作太过分散，分散在了各个 action 中

3.3.3 My Todo redux-sagas 业务实现

以添加任务列表为例，业务流程图如下：



添加任务列表功能包括：

- 1) 在新增输入框中输入任务名，按下回车按钮，派发新增任务 action，清空新增任务输入框 value
- 2) 执行新增任务事件，生成任务复选框
- 3) 在新增输入框中输入任务名，按下回车按钮，派发新增任务 action，清空新增任务输入框 value

3.3.3.1 创建 input.js 文件，实现新增任务输入框

使用 connect 方法建立 state 和 action 的关联

路径 : /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 执行新增任务事件，生成任务复选框

创建一个 inputSaga.js 文件

路径：/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

    })

  }

}

```

在 redux 中使用 redux-saga 中间件

新建 store.js 文件

创建 store，关联 reducers 和 saga 中间件

路径: /src/store.js

```

const sagaMiddleware = createSagaMiddleware()

const store = createStore(

  reducers,

  applyMiddleware(sagaMiddleware)

)

sagaMiddleware.run(rootSaga)

```

```
export default store
```

使用 Provider 建立根组件和 store 关联

路径 : /src/index.js

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

新建 actions

/src/common/actionTypes.js

```
const actionTypes = {  
  ADD_ITEM: 'ADD_ITEM'  
}
```

/src/actions/index.js

```
export {actionTypes}  
  
import { actionTypes } from '../common/actionTypes'  
  
export function addItem(value) {  
  return {  
    type: actionTypes.ADD_ITEM,  
    value  
  }  
}
```

获取已经添加的任务列表

路径 : /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
```

多个 reducers 需要使用 combineReducers

路径 : /src/reducers/index.js

```
const reducer = combineReducers({  
  getTodoCount,  
  getTodoList
```

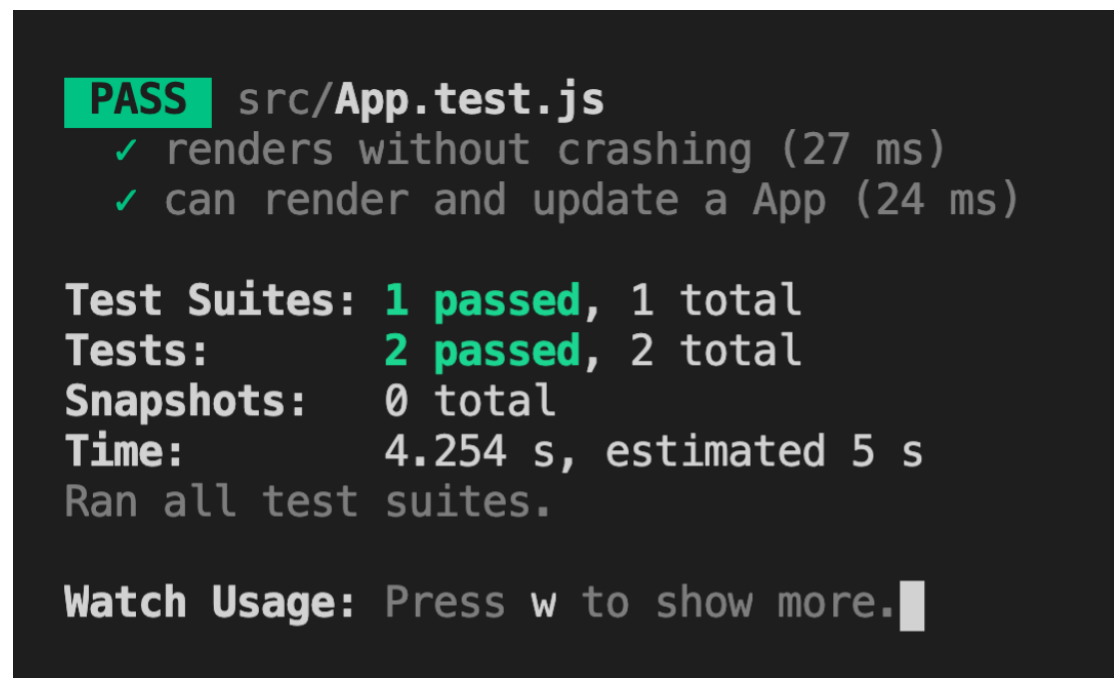
```
})
```

```
export default reducer
```

4 单元测试

该项目单元测试需要在命令行输入

```
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.
```

图十七

输出单元测试报表，输入命令

```
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.					

图十八

在需要测试的模块添加*.test.js 文件，单元测试工具 Jest 会自动识别测试用例，并根据断言进行测试。

redux-sagas 项目单元测试：

组件测试

以测试 App 组件 render,componentDidmount,componentDidUpdate 为例，可以根据组件逻辑，编写测试用例。

路径：/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 function

```

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("");

});
```

saga 测试、selector 测试及 utils 测试的用例写法继续研究中。

5 参考资料

- ◆ 使用 Jest 对 React 全家桶(react-saga, redux-actions, reselect)的单元测试 (<https://juejin.cn/post/6844903703128834062>)
- ◆ 测试技巧 (<https://reactjs.bootcss.com/docs/testing-recipes.html>)
- ◆ 记录一次艰难，却很有意思的问题解决经历-React input

(<https://juejin.cn/post/6844904128305430541>)

◆ 模拟登录 react 的页面 (<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>)

◆ 创建新的 React 应用

(<https://react.docschina.org/docs/create-a-new-react-app.html#create-react-app>)