书籍完整目录

3.4 redux 异步

在大多数的前端业务场景中，需要和后端产生异步交互，在本节中，将详细讲解 redux 中的异步方案以及一些异步第三方组件，内容有：

- redux 异步流
- redux-thunk
- redux-promise
- redux-saga

3.4.1 redux 异步流

前面讲的 redux 中的数据流都是同步的，流程如下：

`view -> actionCreator -> action -> reducer -> newState -> container component`
但同步数据不能满足真实业务开发，真实业务中异步才是主角，那如何将异步处理结合到上边的流程中呢？

3.4.2 实现异步的方式

其实 redux 并未有和异步相关的概念，我们可以用任何原来实现异步的方式应用到 redux 数据流中，最简单的方式就是延迟 dispatch action，以 setTimeout 为例：

```
this.dispatch({ type: 'SYNC_SOME_ACTION'})
window.setTimeout(() => {
  this.dispatch({ type: 'ASYNC_SOME_ACTION' })
}, 1000)
```

这种方式最简单直接，但是有如下问题：

1. 如果有多个类似的 action 触发场景，异步逻辑不能重用
2. 异步处理代码不能统一处理，最简单的例子就是节流

解决上面两个问题的办法很简单，把异步的代码剥离出来：

someAction.js

```
function dispatchSomeAction(dispatch, payload) {
    // ..调用控制逻辑...
    dispatch({ type: 'SYNC_SOME_ACTION'})
    window.setTimeout(() => {
      dispatch({ type: 'ASYNC_SOME_ACTION' })
    }, 1000)
}
```

然后组件只需要调用：

```
import {dispatchSomeAction} from 'someAction.js'

dispatchSomeAction(dispatch, payload);
```

基于这种方式上面的流程就改为了：

`view -> asyncActionDispatcher -> wait -> action -> reducer -> newState -> container component`

asyncActionDispatcher 和 actionCreator 是十分类似的, 所以简单而言就可以把它理解为 asyncActionCreator , 所以新的流程为：

`view -> asyncActionCreator -> wait -> action -> reducer -> newState -> container component`

但是上面的方法有一些缺点

同步调用和异步调用的方式不相同:

- 同步的情况: `store.dispatch(actionCreator(payload))`
- 异步的情况: `asyncActionCreator(store.dispatch, payload)`

幸运的是在 redux 中通过 middleware 机制可以很容易的解决上面的问题

通过 middleware 实现异步

我们已经很清楚一个 middleware 的结构 ，其核心的部分为

```
function(action) {
    // 调用后面的 middleware
    next(action)
}
```

middleware 完全掌控了 reducer 的触发时机， 也就是 action 到了这里完全由中间件控制，不乐意就不给其他中间件处理的机会，而且还可以控制调用其他中间件的时机。

举例来说一个异步的 ajax 请求场景，可以如下实现：

```
function (action) {
    // async call
    fetch('....')
      .then(
          function resolver(ret) {
            newAction = createNewAction(ret, action)
            next(newAction)
          },
          function rejector(err) {
            rejectAction = createRejectAction(err, action)
            next(rejectAction)
          })
    });
}
```

任何异步的 javascript 逻辑都可以，如： ajax callback, Promise, setTimeout 等等, 也可以使用 es7 的 async 和 await。

第三方异步组件

上面的实现方案只是针对具体的场景设计的，那如果是如何解决通用场景下的问题呢，其实目前已经有很多第三方 redux 组件支持异步 action，其中如：

- redux-thunk
- redux-promise
- redux-saga

这些组件都有很好的扩展性，完全能满足我们开发异步流程的场景，下面来一一介绍

3.4.3 redux-thunk

redux-thunk 介绍

redux-thunk 是 redux 官方文档中用到的异步组件，实质就是一个 redux 中间件，thunk 听起来是一个很陌生的词语，先来认识一下什么叫 thunk

> A thunk is a function that wraps an expression to delay its evaluation.

简单来说一个 thunk 就是一个封装表达式的函数，封装的目的是延迟执行表达式

```
// 1 + 2 立即被计算 = 3
let x = 1 + 2;

// 1 + 2 被封装在了 foo 函数内
// foo 可以被延迟执行
// foo 就是一个 thunk
let foo = () => 1 + 2;
```

redux-thunk 是一个通用的解决方案，其核心思想是让 action 可以变为一个 thunk ，这样的话:

1. 同步情况：dispatch(action)
2. 异步情况：dispatch(thunk)

我们已经知道了 thunk 本质上就是一个函数，函数的参数为 dispatch, 所以一个简单的 thunk 异步代码就是如下：

```
this.dispatch(function (dispatch){
    setTimeout(() => {
       dispatch({type: 'THUNK_ACTION'})
    }, 1000)
})
```

之前已经讲过，这样的设计会导致异步逻辑放在了组件中，解决办法为抽象出一个 asyncActionCreator, 这里也一样，我们就叫 thunkActionCreator 吧，上面的例子可以改为:

```
//actions/someThunkAction.js
export function createThunkAction(payload) {
    return function(dispatch) {
        setTimeout(() => {
           dispatch({type: 'THUNK_ACTION', payload: payload})
        }, 1000)
    }
}

// someComponent.js
this.dispatch(createThunkAction(payload))
```

安装和使用

第一步：安装

```
$ npm install redux-thunk
```

第二步: 添加 thunk 中间件

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers/index';

const store = createStore(
  rootReducer,
  applyMiddleware(thunk)
);
```

第三步：实现一个 thunkActionCreator

```
//actions/someThunkAction.js
export function createThunkAction(payload) {
    return function(dispatch) {
        setTimeout(() => {
           dispatch({type: 'THUNK_ACTION', payload: payload})
        }, 1000)
    }
}
```

第三步：组件中 dispatch thunk

```
this.dispatch(createThunkAction(payload));
```

> 拥有 dispatch 方法的组件为 redux 中的 container component

thunk 源码

说了这么多，redux-thunk 是不是做了很多工作，实现起来很复杂，那我们来看看 thunk 中间件的实现

```
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;
```

就这么简单，只有 14 行源码，但是这简短的实现却能完成复杂的异步处理，怎么做到的，我们来分析一下：

1. 判断如果 action 是 function 那么执行 action(dispatch, getState, ...)
    1. action 也就是一个 thunk
    2. 执行 action 相当于执行了异步逻辑
        1. action 中执行 dispatch
        2. 开始新的 redux 数据流，重新回到最开始的逻辑（thunk 可以嵌套的原因）
    3. 把执行的结果作为返回值直接返回
    4. 直接返回并没有调用其他中间件，也就意味着中间件的执行在这里停止了
    5. 可以对返回值做处理（后面会讲如果返回值是 Promise 的情况）
2. 如果不是函数直接调用其他中间件并返回

理解了这个过后是不是对 redux-thunk 的使用思路变得清晰了

thunk 的组合

根据 redux-thunk 的特性，可以做出很有意思的事情

1. 可以递归的 dispatch(thunk) => 实现 thunk 的组合；
2. thunk 运行结果会作为 dispatch返回值 => 利用返回值为 Promise 可以实现多个 thunk 的编排;

thunk 组合例子：

```
function thunkC() {
    return function(dispatch) {
        dispatch(thunkB())
    }
}
function thunkB() {
    return function (dispatch) {
        dispatch(thunkA())
    }
}
function thunkA() {
    return function (dispatch) {
        dispatch({type: 'THUNK_ACTION'})
    }
}
```

Promise 例子

```
function ajaxCall() {
    return fetch(...);
}

function thunkC() {
    return function(dispatch) {
        dispatch(thunkB(...))
        .then(
            data => dispatch(thunkA(data)),
            err  => dispatch(thunkA(err))
        )
    }
}
function thunkB() {
    return function (dispatch) {
        return ajaxCall(...)
    }
}

function thunkA() {
    return function (dispatch) {
        dispatch({type: 'THUNK_ACTION'})
    }
}
```

3.4.4 redux-promise

另外一个 redux 文档中提到的异步组件为 redux-promise, 我们直接分析一下其源码吧

```
import { isFSA } from 'flux-standard-action';

function isPromise(val) {
  return val && typeof val.then === 'function';
}

export default function promiseMiddleware({ dispatch }) {
  return next => action => {
    if (!isFSA(action)) {
      return isPromise(action)
        ? action.then(dispatch)
        : next(action);
    }

    return isPromise(action.payload)
      ? action.payload.then(
          result => dispatch({ ...action, payload: result }),
          error => {
            dispatch({ ...action, payload: error, error: true });
            return Promise.reject(error);
          }
        )
      : next(action);
  };
}
```

大概的逻辑就是：

1. 如果不是标准的 flux action，那么判断是否是 promise, 是执行 action.then(dispatch)，否执行 next(action)
2. 如果是标准的 flux action, 判断 payload 是否是 promise，是的话 payload.then 获取数据，然后把数据作为 payload 重新 dispatch({ ...action, payload: result})，否执行 next(action)

结合 redux-promise 可以利用 es7 的 async 和 await 语法，简化异步的 promiseActionCreator 的设计， eg:

```
export default async (payload) => {
  const result = await somePromise;
  return {
    type: "PROMISE_ACTION",
    payload: result.someValue;
  }
}
```

如果对 es7 async 语法不是很熟悉可以看下面两个例子：

1. async 关键字可以总是返回一个 Promise 的 resolve 结果或者 reject 结果

```
async function foo() {
    if(true)
        return 'Success!';
    else
        throw 'Failure!';
}

// 等价于

function foo() {
    if(true)
        return Promise.resolve('Success!');
    else
        return Promise.reject('Failure!');
}
```

1. 在 async 关键字中可以使用 await 关键字，其目的是 await 一个 promise， 等待 promise resolve 和 reject

eg：

```
async function foo(aPromise) {
    const a = await new Promise(function(resolve, reject) {
            // This is only an example to create asynchronism
            window.setTimeout(
                function() {
                    resolve({a: 12});
                }, 1000);
        })
    console.log(a.a)
    return  a.a
}

// in console
> foo()
> Promise {_c: Array[0], _a: undefined, _s: 0, _d: false, _v: undefined…}
> 12
```

可以看到在控制台中，先返回了一个 promise，然后输出了 12

async 关键字可以极大的简化异步流程的设计，避免 callback 和 thennable 的调用，看起来和同步代码一致。

3.4.5 redux-saga

redux-saga 介绍

redux-saga 也是解决 redux 异步 action 的一个中间件，不过和之前的设计有本质的不同

1. redux-saga 完全基于 Es6 的 Generator Function

2. 不使用 actionCreator 策略，而是通过监控 action, 然后在自动做处理
3. 所有带副作用的操作（异步代码，不确定的代码）都被放到 saga 中

那到底什么是 saga

redux-saga 实际也没有解释什么叫 saga ，通过引用的参考：

> The term saga is commonly used in discussions of CQRS to refer to a piece of code that coordinates and routes messages between bounded contexts and aggregates.

这个定义的核心就是 **CQRS-查询与责任分离** ，对应到 redux-sage 就是 action 与 处理函数的分离。 实际上在 redux-saga 中，一个 saga 就是一个 Generator 函数。

eg:

```
import { takeEvery, takeLatest } from 'redux-saga'
import { call, put } from 'redux-saga/effects'
import Api from '...'

/*
 * 一个 saga 就是一个 Generator Function
 *
 * 每当 store.dispatch `USER_FETCH_REQUESTED` action 的时候都会调用 fetchUser.
 */
function* mySaga() {
  yield* takeEvery("USER_FETCH_REQUESTED", fetchUser);
}

/**
 * worker saga:  真正处理 action 的 saga
 *
 * USER_FETCH_REQUESTED action 触发时被调用
 * @param {[type]} action  [description]
 * @yield {[type]} [description]
 */
function* fetchUser(action) {
   try {
      const user = yield call(Api.fetchUser, action.payload.userId);
      yield put({type: "USER_FETCH_SUCCEEDED", user: user});
   } catch (e) {
      yield put({type: "USER_FETCH_FAILED", message: e.message});
   }
}
```

一些基本概念

watcher saga

负责编排和派发任务的 saga

worker saga

真正负责处理 action 的函数

saga helper

如上面例子中的 takeEvery，简单理解就是用于监控 action 并派发 action 到 worker saga 的辅助函数

Effect

redux-saga 完全基于 Generator 构建，saga 逻辑的表达是通过 yield javascript 对象来实现，这些对象就是Effects。

这些对象相当于描述任务的规范化数据（任务如执行异步函数，dispatch action 到一个 store），这些数据被发送到 redux-saga 中间件中执行，如：

1. `put({type: "USER_FETCH_SUCCEEDED", user: user})` 表示要执行 `dispatch({{type: "USER_FETCH_SUCCEEDED", user: user}})` 任务
2. `call(fetch, url)` 表示要执行 `fetch(url)`

通过这种 effect 的抽象，可以避免 call 和 dispatch 的立即执行，而是描述要执行什么任务，这样的话就很容易对 saga 进行测试，saga 所做的事情就是将这些 effect 编排起来用于描述任务，真正的执行都会放在 middleware 中执行。

安装和使用

第一步：安装

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

第二步：添加 saga 中间件

```
import { createStore, applyMiddleware } from 'redux'
import createSagaMiddleware from 'redux-saga'

import reducer from './reducers'
import mySaga from './sagas'

// 创建 saga 中间件
const sagaMiddleware = createSagaMiddleware()

// 添加到中间件中
const store = createStore(
  reducer,
  applyMiddleware(sagaMiddleware)
)

// 立即运行 saga ，让监控器开始监控
sagaMiddleware.run(mySaga)
```

第三步：定义 sagas/index.js

```
import { takeEvery } from 'redux-saga'
import { put } from 'redux-saga/effects'

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

// 将异步执行 increment 任务
export function* incrementAsync() {
  yield delay(1000)
  yield put({ type: 'INCREMENT' })
}

// 在每个 INCREMENT_ASYNC action 调用后，派生一个新的 incrementAsync 任务
export default function* watchIncrementAsync() {
  yield* takeEvery('INCREMENT_ASYNC', incrementAsync)
}
```

第四步：组件中调用

```
this.dispatch({type: 'INCREMENT_ASYNC'})
```

redux-saga 基于 Generator 有很多高级的特性， 如：

1. 基于 take Effect 实现更自由的任务编排
2. fork 和 cancel 实现非阻塞任务
3. 并行任何和 race 任务
4. saga 组合 ，yield* saga

> 因篇幅有限，这部分内容在下一篇讲解