REDUX学习 源码阅读

Src

utils

actionTypes.js

isPlainObject.js

warning.js

applyMiddleware.js

bindActionCreators.js

combineReducers.js

compose.js

createStore.js

index.js

isPlanObject.js

```
export default function isPlainObject(obj) {
  if (typeof obj !== 'object' || obj === null) return false

  let proto = obj
  while (Object.getPrototypeOf(proto) !== null) {
    proto = Object.getPrototypeOf(proto)
  }

  return Object.getPrototypeOf(obj) === proto
}
```

什么是简单对象？

应该有一些小伙伴不理解，所谓的简单对象就是该对象的__proto__等于Object.prototype,用一句通俗易懂的话就是:凡不是用new Object()或字面量的方式构建出来的对象都不是简单对象

```
class Fruit{
  sayName(){
    console.log(this.name)
  }
}

class Apple extends Fruit{
  constructor(){
    super();
    this.name=""
  }
}

const apple = new Apple();
const fruit = new Fruit();
const cherry = new Object({
  name:''
});
const banana = {
  name:''
};

console.log(isPlainObject(apple));//false
console.log(isPlainObject(fruit));//false
console.log(isPlainObject(cherry));//true
console.log(isPlainObject(banana));//true
```

这里可能会有人不理解**isPlainObject(fruit)===false**，如果对这个不能理解的话，自己后面要补习一下原型链的相关知识，这里fruit.__proto__.__proto__才等价于Object.prototype。

逻辑代码

index.js

```
import createStore from './createStore'
import combineReducers from './combineReducers'
import bindActionCreators from './bindActionCreators'
import applyMiddleware from './applyMiddleware'
import compose from './compose'
import warning from './utils/warning'
import __DO_NOT_USE__ActionTypes from './utils/actionTypes'

function isCrushed() {}

if (
  process.env.NODE_ENV !== 'production' &&
  typeof isCrushed.name === 'string' &&
  isCrushed.name !== 'isCrushed'
) {
  warning(
    "You are currently using minified code outside of NODE_ENV === 'production'. " +
      'This means that you are running a slower development build of Redux. ' +
      'You can use loose-envify (https://github.com/zertosh/loose-envify) for browserify ' +
      'or DefinePlugin for webpack (http://stackoverflow.com/questions/30030031) ' +
      'to ensure you have the correct code for your production build.'
  )
}

export {
  createStore,
  combineReducers,
  bindActionCreators,
  applyMiddleware,
  compose,
  __DO_NOT_USE__ActionTypes
}
```

此时判断条件就成立了，错误信息就会打印出来。这个主要作用就是防止开发者在开发环境下对代码进行压缩。

createStore.js

函数createStore接受三个参数（reducer、preloadedState、enhancer），reducer和enhancer我们用的比较多，preloadedState用的比较少。第一个reducer很好理解，这里就不过多解释了，第二个preloadedState，它代表着初始状态，我们平时在项目里也很少用到它，主要说一下enhancer，中文名叫增强器，顾名思义就是来增强redux的，它的类型的是Function，createStore.js里有这么一行代码：

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

    return enhancer(createStore)(reducer, preloadedState)
  }
```

这行代码展示了enhancer的调用过程，根据这个调用过程我们可以推导出enhancer的函数体的架子应该是这样子的:

```
function enhancer(createStore) {
    return (reducer,preloadedState) => {
         //
        .......
    }
 }
```

常见的enhancer就是redux-thunk以及redux-saga，一般都会配合applyMiddleware一起使用，而applyMiddleware的作用就是将这些enhancer格式化成符合redux要求的enhancer。具体applyMiddleware实现，下面我们将会讲到。我们先看redux-thunk的使用的例子：

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

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

看完上面的代码，可能会有人有这么一个疑问“**createStore函数第二个参数不是preloadedState吗？这样不会报错吗？**” 首先肯定不会报错，毕竟官方给的例子，不然写个错误的例子也太大跌眼镜了吧！redux肯定是做了这么一层转换，我在createStore.js找到了这么一行代码：

```
 if (typeof preloadedState === 'function' && typeof enhancer === 'undefined') {
    enhancer = preloadedState
    preloadedState = undefined
  }
```

当第二个参数preloadedState的类型是Function的时候，并且第三个参数enhancer未定义的时候，此时preloadedState将会被赋值给enhancer，preloadedState会替代enhancer变成undefined的。有了这么一层转换之后，我们就可以大胆地第二个参数传enhancer了。

说完createStore的参数，下面我说一下函数createStore执行完之后返回的对象都有什么？在createStore.js最下面一行有这一行代码：

```
return {
    dispatch,
    subscribe,
    getState,
    replaceReducer,
    [$$observable]: observable
  }
```

定义的一些变量

```
let currentState = preloadedState //createStorepreloadedState
let currentReducer = reducer  //createStorereducer
let currentListeners = [] //
let nextListeners = currentListeners //
let isDispatching = false
```

其中变量isDispatching，作为锁来用，我们redux是一个统一管理状态容器，它要保证数据的一致性，所以同一个时间里，只能做一次数据修改，如果两个action同时触发reducer对同一数据的修改，那么将会带来巨大的灾难。所以变量isDispatching就是为了防止这一点而存在的。（redux如何防止两个action修改reducer）

dispatch

```
function dispatch(action) {
    if (!isPlainObject(action)) {
      throw new Error(
        'Actions must be plain objects. ' +
          'Use custom middleware for async actions.'
      )
    }

    if (typeof action.type === 'undefined') {
      throw new Error(
        'Actions may not have an undefined "type" property. ' +
          'Have you misspelled a constant?'
      )
    }

    if (isDispatching) {
      throw new Error('Reducers may not dispatch actions.')
    }

    try {
      isDispatching = true
      currentState = currentReducer(currentState, action)
    } finally {
      isDispatching = false
    }

    const listeners = (currentListeners = nextListeners)
    for (let i = 0; i < listeners.length; i++) {
      const listener = listeners[i]
      listener()
    }

    return action
  }
```

函数dispatch在函数体一开始就进行了三次条件判断，分别是以下三个：

- **判断action是否为简单对象**
- **判断action.type是否存在**
- **判断当前是否有执行其他的reducer操作**

当前三个预置条件判断都成立时，才会执行后续操作，否则抛出异常。在执行reducer的操作的时候用到了try-finally，可能大家平时try-catch用的比较多，这个用到的还是比较少。执行前isDispatching设置为true，阻止后续的action进来触发reducer操作，得到的state值赋值给currentState，完成之后再finally里将isDispatching再改为false，允许后续的action进来触发reducer操作。接着一一通知订阅者做数据更新，不传入任何参数。最后返回当前的action。

getState

```
function getState() {
    if (isDispatching) {
      throw new Error(
        'You may not call store.getState() while the reducer is executing. ' +
          'The reducer has already received the state as an argument. ' +
          'Pass it down from the top reducer instead of reading it from the store.'
      )
    }

    return currentState
  }
```

`createStorestore,statestate`

store通过getState得出的state是可以直接被更改的，但是redux不允许这么做，因为这样不会通知订阅者更新数据。