Redux

林佳钰

智能支付 用户前端组

贵圈很乱，但是你也可能不需要 .它.

fe-qrcodemis例子

一段代码

React： state管理页面数据

```
export default class DealList extends Component {
    constructor() {
        super();
        this.state = {
            userId: window.initState.userId,
            sid: window.initState.sid,
            dealInfo: {},
            // shopInfo: {},
            refundInfoList: [],
            cardRefundInfo: {},
            billInfoQueryParam: {
                agentTradeNo: '',
                addTIme: '',
                modTime: ''
            },
            preAuthModalStatus: false,
            refundModalStatus: false,
            preAuthList: [],
            payinfoData: {},
            refundParam: {
                orderId: '',
                merchantId: '',
                transaction_id: '',
                status: '',
                origin: '',
                payType: '',
                refundFee: '',
                reason: '',
                notifierContact: '',
                notifierName: ''
            },
            qdFyStatus: {},
            qdFyStatusParam: {
                orderid: '',
                paytype: '',
                batchNo: '',
                traceno: '',
                orderTime: '',
                totalFee: '',
                deviceId: '',
                tradeNo: ''
            },
            refundRadioStatus: [false, false, false, false],
            deviceInfo: {},
            tabActive: '1',
            authLoading: true,
            refundType: 'all',
            sectionRefundNum: '',
            checkCodeStatus: false,
            codeBody: {},
            location: ''
        };
    }
    componentWillMount() {
        checkPermission(this.props.match, this.props.location, this.props.history);
```

this.setState变更数据

```
    const refundParam = {...this.state.refundParam};
    refundParam[key] = val;
    if (index !== undefined) {
        refundRadioStatus[index] = true;
    }
    // 正确使用方式: this.setState: 改变数据
    this.setState({
        refundParam,
        refundRadioStatus
    });

    // 不正确使用: 在object的数据强行被变更。
    const { refundParam } = this.state;
    refundParam.orderId = '数据被玷污了~ ';
}
```

不正确的直接修改state方式，不会重新出发render

fe-qrcodemis例子

一段代码的思考

1. 数据变化可预知

2. 页面可共享状态

函数式编程

Pure Function

- The function always evaluates the same result value given the same argument value(s)
- Evaluation of the result does not cause any semantically observable side effect or output,

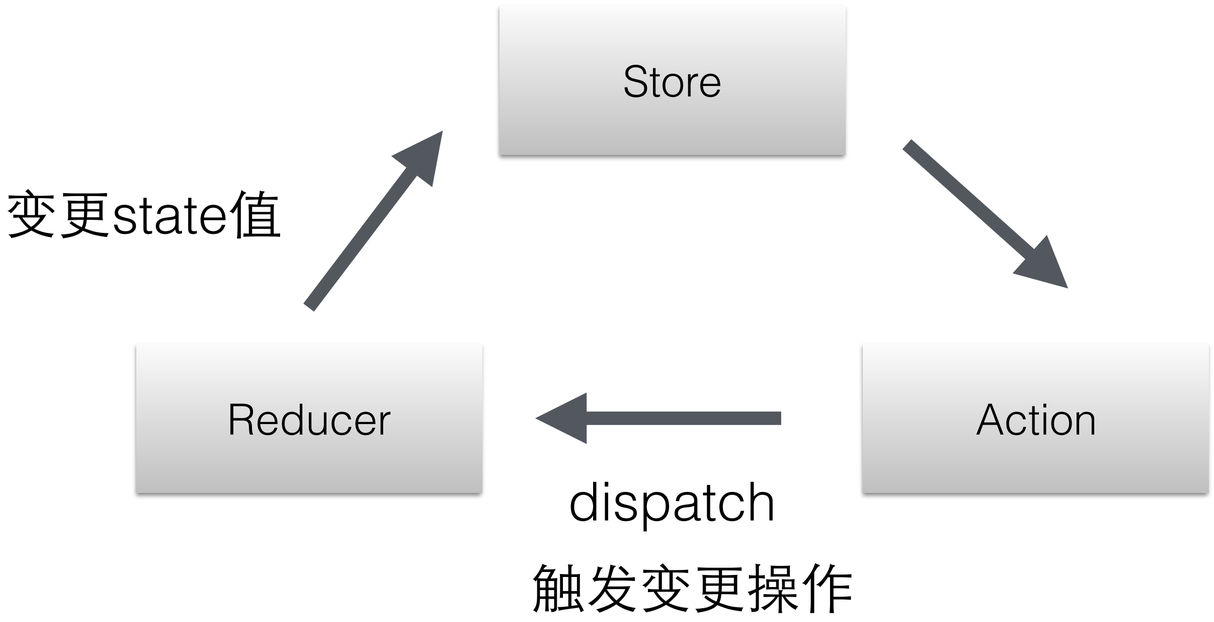
$$f(x) = x + 1$$

Compose

Currying

Redux

A predictable state container for JavaScript apps.



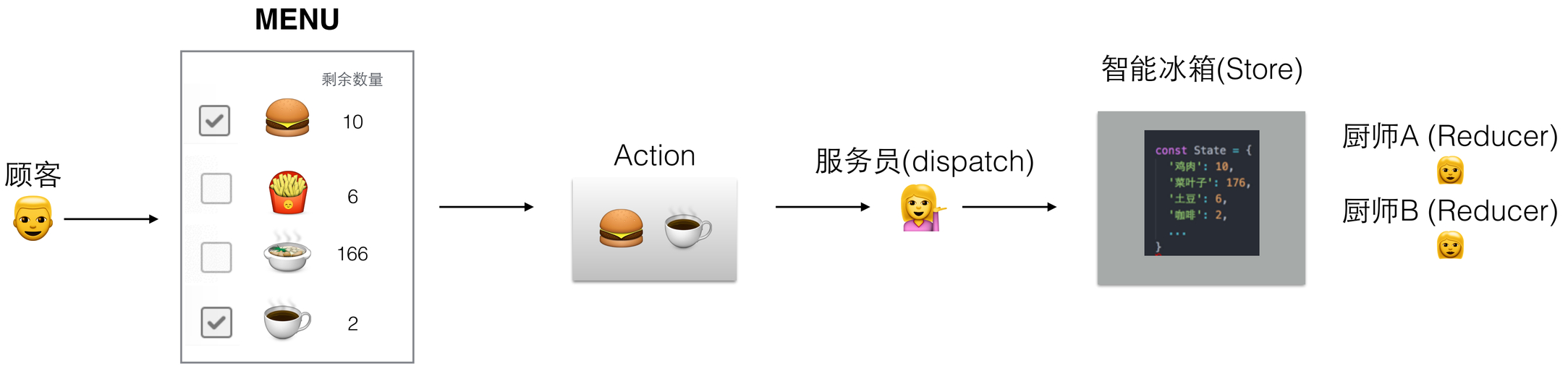
1. 单一数据源

2. 数据只读

3.使用纯函数执行修改

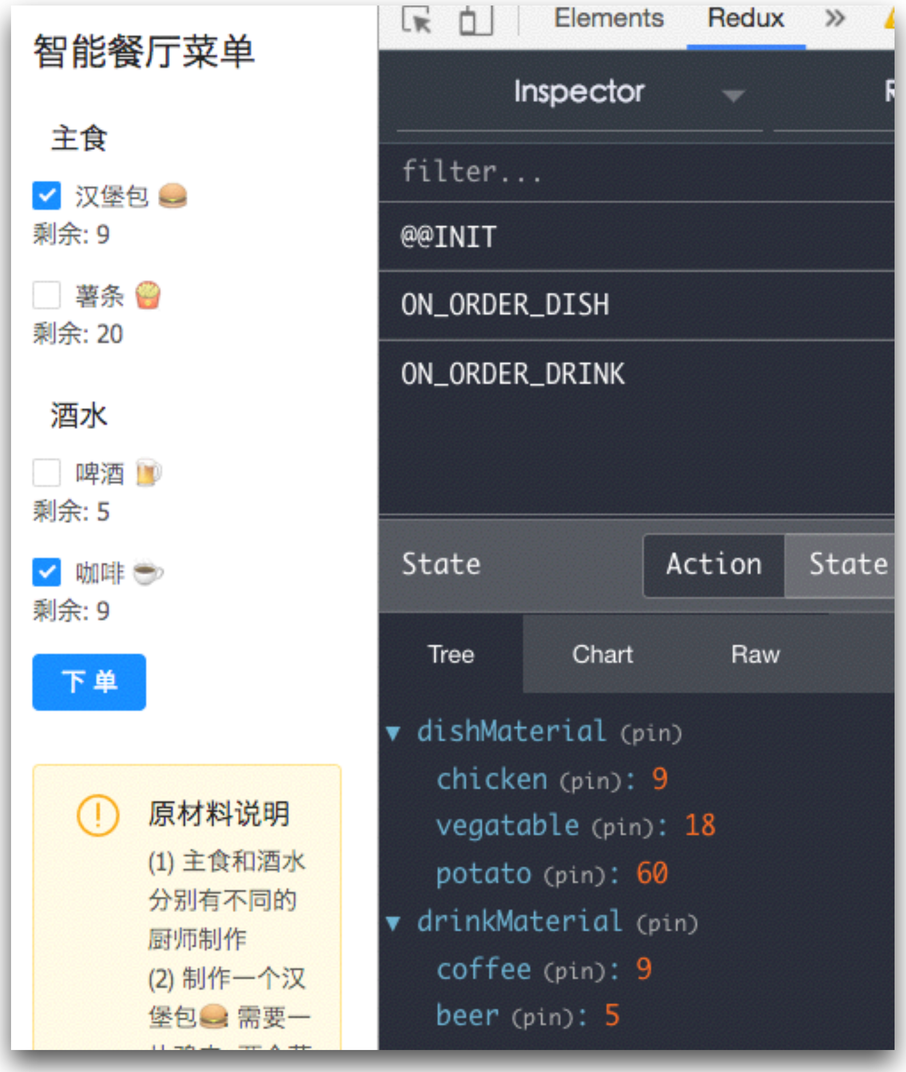
一个餐厅的故事

老板有一套食材管理系统叫(Redux)，所有的原材料都在智能冰箱(Store)里面放着。
菜单上面会显示每一种食材剩余的数量， 当顾客(View)选择(Action)几种食材并下单后。
服务员把单子输入(dispatch)到智能冰箱(Store)中，智能冰箱(Store)可根据几种食材找到相应做菜的厨师(Reducer)。只有(Reducer)厨师才可以从智能冰箱(Store)拿原材料。



Redux

Demo栗子： https://github.com/Linjiayu6/learn-redux-code



Redux 5个API

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

Redux - combineReducers

Turns an object whose values are different reducer functions, into a single reducer function

https://github.com/Linjiayu6/learn-redux-code/blob/master/code/combineReducers.js

```
export default function combineReducers(reducers) {
  const finalReducers = reducers;
  const finalReducerKeys = Object.keys(finalReducers);

  /**
   * combination: 形成一个新的rootReducer函数
   * @state: 整个state树
   * @action: 变更state树的唯一条件
   */
  return function combination(state = {}, action) {
    // ... 省略一些判断条件....

    let hasChanged = false; // 判断状态是否变化的标记变量
    const nextState = {};

    // 遍历每一个reducer函数，根据传入的action值，去改变对应的state内容
    for (let i = 0; i < finalReducerKeys.length; i++) {
      const key = finalReducerKeys[i];
      const reducer = finalReducers[key];

      // 获取当前对应key值的state内容
      const previousStateForKey = state[key];
      // * 划重点1: 根据传入的 action = { type, payload }，来更新state值。
      const nextStateForKey = reducer(previousStateForKey, action);
      // 将新获取的state值，放到nextState中。
      nextState[key] = nextStateForKey;

      // *划重点2: state值是否变化了，如果改变了，则更新整个state数.
      // * Redux 只通过比较新旧两个对象的存储位置来比较新旧两个对象是否相同。
      hasChanged = hasChanged || nextStateForKey !== previousStateForKey;
    }
    // 如果有更新则变更 state = nextState
    return hasChanged ? nextState : state;
  };
}
```

Redux - createStore

Creates a Redux store that holds the state tree.

createStore(reducer, [preloadedState], enhancer);

```
return {
    dispatch, // * 触发reducer, 以及subscribe绑定的监听函数
    getState, // * 用来获取当前最新的tate
    subscribe, // [使用不多] 主要作用就是绑定监听函数
    replaceReducer, // [使用不多] 主要目的就是用新的reducer取代当前的reducer
  }
```

https://github.com/Linjiayu6/learn-redux-code/blob/master/code/createStore.js

Redux - 例子

简化其他处理等流程，只保留主流程实现的例子

https://github.com/Linjiayu6/learn-redux-code/blob/master/code/example_basic.js

```
const combineReducer = (state = { name: 1 }, action) => {
  if (action.type === 'BURGER') {
    return { name: action.type };
  }
  if (action.type === 'COFFEE') {
    return { name: action.type };
  }
  return state;
};

const createStore = (reducer, preloadedState = { name: 'ljy' }) => {
  const currentReducer = reducer;
  let currentState = preloadedState;
  console.log('currentState....', currentState);

  const getState = () => currentState;
  const dispatch = (action) => {
    console.log('dispatch - currentState', currentState);
    currentState = currentReducer(currentState, action);
  };

  return {
    getState,
    dispatch,
  };
};
```

核心流程实现

函数式编程

Pure Function

Compose

Currying

Compose

组合函数，将函数串联起来执行，
一个函数的输出结果是另一个函数的输入参数。

像Domino一样，推倒一个，其他函数也跟着执行。

```
funcs.reduce((a, b) => (...args) => a(b(...args)))
```

https://github.com/Linjiayu6/learn-redux-code/blob/master/code/compose.js

Currying

所谓”柯里化"，就是把一个多参数的函数，转化为单参数函数

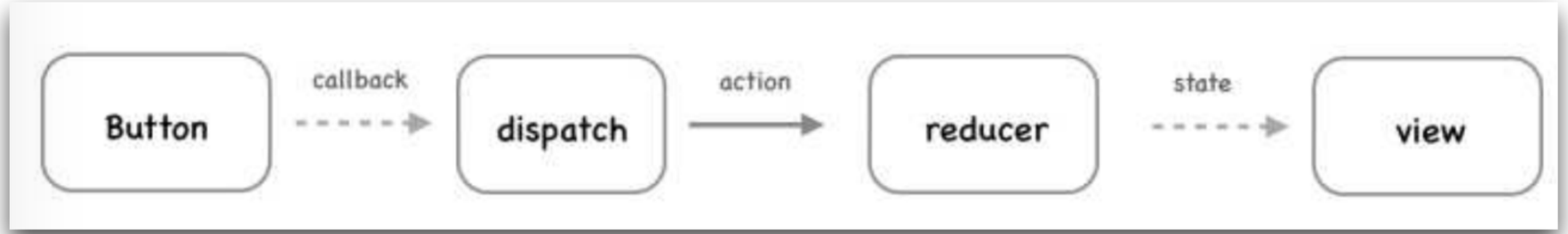
```
const add = (x, y, z) => x + y + z;
add(1, 2, 3);

// 把一个多参数的函数，转化为单参数函数    *每次只接受单参
// curry: 高阶函数，匿名函数，闭包
const addCurry = x => y => z => x + y + z;
addCurry(1)(2)(3);
```

简单问题复杂化? 为什么?

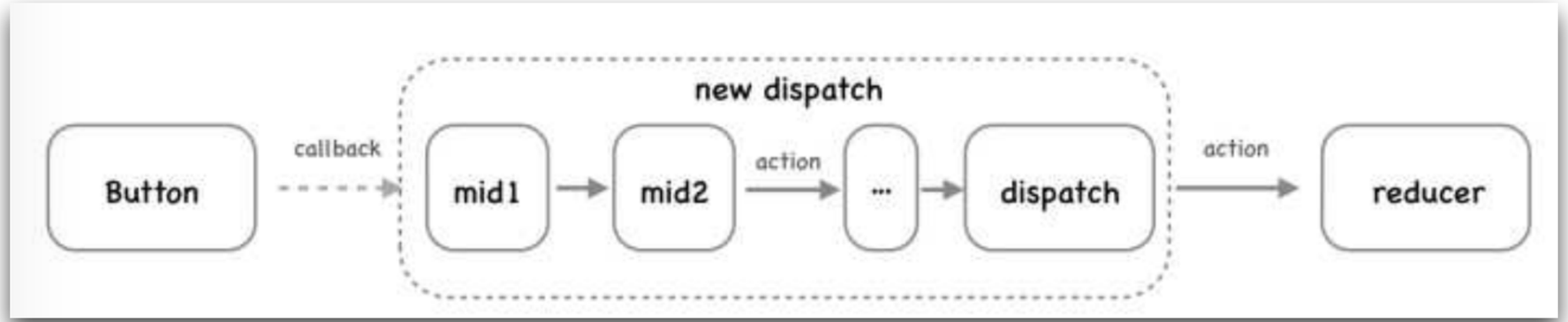
1. 问题细化，逐步处理　　2. 函数式处理

https://github.com/Linjiayu6/learn-redux-code/blob/master/code/currying.js

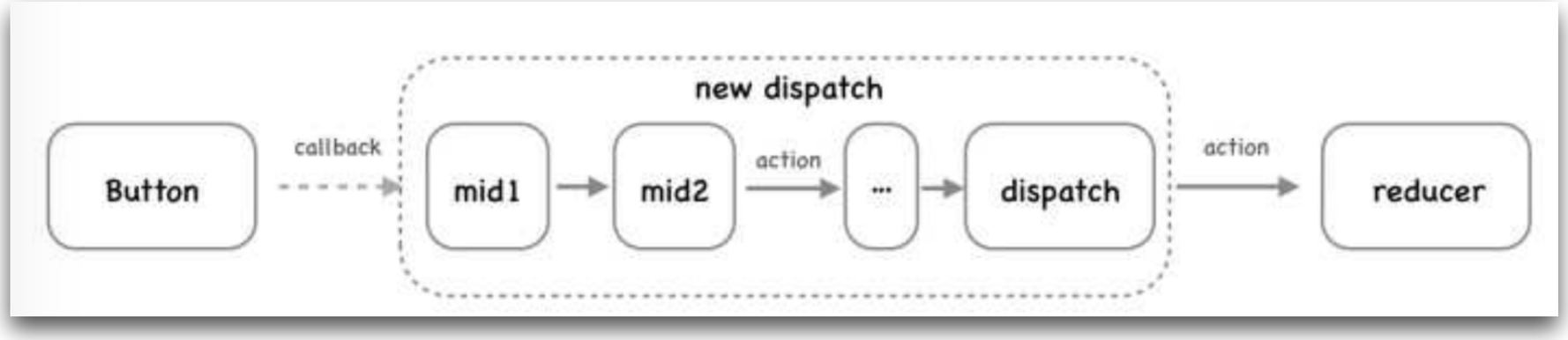
Redux ApplyMiddleware



Redux借鉴Koa中间件的思想



Redux ApplyMiddleware



1. 如何将中间件串联起来?
2. 如何保证最后执行dispatch(action)?
3. 如何像Koa获取ctx一样，每次都能获取到Store，并在执行过后，将处理权交给下一个中间件?

```
// koa中间件的例子
const logger = (ctx, next) => {
  console.log(`${Date.now()} ${ctx.request.method} ${ctx.request.url}`);
  next();
}
```

Redux ApplyMiddleware

- 如何将中间件串联起来?

  如何保证最后执行dispatch(action)?

```
/**
 * 目标: 想在每一次dispatch的时候，打印log
 *   store.dispatch({ type: 'BURGER' });
 *
 * 问题:
 *   1. 如何将中间件串联起来?  compose
 *   2. 如何保证最后执行dispatch(action)?  最后一个执行dispatch(action)就行了
 *   eg: dispatch(mid2(mid1(action)))
 */
const action = { type: 'BURGER' };
const mid1 = action => { console.log('(1)中间件1', action); return action; }
const mid2 = action => { console.log('(2)中间件2', action); return action; }
const final_dispatch = action => store.dispatch(action);


// ********************* 执行 ***********************************
compose(final_dispatch, mid2, mid1)(action);
console.log('更新的state: ', store.getState());
```

Redux ApplyMiddleware

- 如何每个中间件都可以访问Store?

```
// (2) 如何在mid1, mid2中访问store的内容?
// 给每个中间件传入store不就解决了吗
const mid_1 = (store, action) => { console.log('(1) mid1: ', action, store); return action; }
const mid_2 = (store, action) => { console.log('(2) mid2: ', action, store); return action; }

// 但是问题是如果使用compose只能传递单参数. 这样两个参数compose搞不了
// 利用currying概念, 大问题拆解处理
const a = store => action => { console.log('(1) mid1_a: ', action, store.getState()); return action; }
const b = store => action => { console.log('(2) mid2_b: ', action, store.getState()); return action; }
const c = store => action => { console.log('(3) dispatch:'), store.dispatch(action); };

// 先利用闭包原理, 将store保存在各个函数中 -> 循环执行处理
// ********************* 执行II ******************************************
const chain = [c, b, a].map(midItem => midItem(store));
compose(...chain)(action)
console.log('更新的state: ', store.getState());
```

Redux ApplyMiddleware

- 总有刁民想害朕~ 如何避免，在中间件执行过程中，提前执行dispatch(action)?
  同时各个中间件也可以期望dispatch自己action?

```
// (3) 总有刁民想害朕
const x = store => action => {
  console.log('(1) mid1_a: ');
  // 不想在最后执行store.dispatch(action), 中间件就执行吧
  // 执行后的结果是: 这一步已经执行了dispatch, 后面的store内容都是不准确的
  store.dispatch(action);
  return action;
}
const y = store => action => { console.log('(2) mid2_b: ', action, store.getState()); return action; }
const z = store => action => { console.log('(3) dispatch: '), store.dispatch(action); };

// 为了避免这种情况出现, 需要给定store.dispatch的权限,
// 也就是说, 只有最后一个执行的函数, 才是真正的store.dispatch(action)
```

(1) 收回store权限，只给可读state权限. 并给各个中间件全局store.dispatch权限，
而不是封装后的新的dispatch权限
(2) dispatch执行到最后才是调用store.dispatch(action) 给定一个标志，
像koa一样，next() 执行下一个

Redux ApplyMiddleware

- 只给中间件传递读权限，
  同时各个中间件也可以期望dispatch自己action?

```
// (1) const chain = [c, b, a].map(midItem => midItem({ getState: store.getState })); 只给读权限
// (2) 如何定义将b,c,dispatch抽象定义
const aa = store => next => action => { console.log('aa', store.getState()); next(action); };
const bb = store => next => action => { console.log('bb', store.getState()); next(action); };
const cc = store => next => action => { console.log('cc', store.getState()); next(action); console.

// ********************* 执行III ****************************************
const chain_new = [aa, bb, cc].map(midItem => midItem({ getState: store.getState }))
// compose处理
const dispath_new = compose(...chain_new)(store.dispatch);
dispath_new(action);
console.log('更新的state: ', store.getState());
```

Redux ApplyMiddleware

```
export default function applyMiddleware(...middlewares) {
  return (createStore) => (reducer, preloadedState, enhancer) => {
    const store = createStore(reducer, preloadedState, enhancer)
    let dispatch = store.dispatch
    let chain = []

    const middlewareAPI = {
      getState: store.getState,
      // 1. 为什么将dispatch封装了一层?
      // store.dispatch是最原始的使用，对每个中间件来说，不会影响其他的调用，故不包含其他中间件
      dispatch: (action) => dispatch(action)
    }
    chain = middlewares.map(middleware => middleware(middlewareAPI))
    // 新的dispatch: 将所有的中间件串联起来的新的dispatch，执行dispatch(aciton)会走所有中间件功能
    dispatch = compose(...chain)(store.dispatch)

    return {
      ...store,
      dispatch
    }
  }
}
```

Q&A