1. **动机**

随着 JavaScript 单页应用开发日趋复杂，****JavaScript 需要管理比任何时候都要多的 state （状态）****。 这些 state 可能包括服务器响应、缓存数据、本地生成尚未持久化到服务器的数据，也包括 UI 状态，如激活的路由，被选中的标签，是否显示加载动效或者分页器等等。

管理不断变化的 state 非常困难。如果一个 model 的变化会引起另一个 model 变化，那么当 view 变化时，就可能引起对应 model 以及另一个 model 的变化，依次地，可能会引起另一个 view 的变化。直至你搞不清楚到底发生了什么。****state 在什么时候，由于什么原因，如何变化已然不受控制。**** 当系统变得错综复杂的时候，想重现问题或者添加新功能就会变得举步维艰。

通过限制更新发生的时间和方式，****Redux 试图让 state 的变化变得可预测****。这些限制条件反映在 Redux 的[三大原则](https://www.redux.org.cn/docs/introduction/ThreePrinciples.html)中。

1. **核心概念**



**核心为Store类**

**工作流程**

1. 用户发出 Action

store.dispatch(action);

1. Store 自动调用 Reducer，并且传入两个参数：当前 State 和收到的 Action。 Reducer 会返回新的 State 。

//todoApp为具体的reducer方法

let nextState = todoApp(previousState, action);

1. State 一旦有变化，Store 就会调用监听函数。

//TODO,需补充listener的具体是什么

//设置监听函数

store.subscribe(listener);

1. listener可以通过store.getState()得到当前状态。如果使用的是 React，这时可以触发重新渲染 View。

function listerner() {

let newState = store.getState();

component.setState(newState);

}

**场景介绍（todo 应用）**

state

{

todos: [{

text: 'Eat food',

completed: true

}, {

text: 'Exercise',

completed: false

}],

visibilityFilter: 'SHOW\_COMPLETED'

}

这个对象就像 “Model”，区别是它并没有 setter（修改器方法）。因此其它的代码不能随意修改它，**限制更新的方式，使变化可控**，避免难以复现的 bug。

action

普通对象，描述一个行为事件，携带事件名称和参数

{ type: 'ADD\_TODO', text: 'Go to swimming pool' }

{ type: 'DEL\_TODO', index: 1 }

{ type: 'SET\_VISIBILITY\_FILTER', filter: 'SHOW\_ALL' }

reducer

只是一个接收 state 和 action，并返回新的 state 的函数

function todos(state = [], action) {

switch (action.type) {

case 'ADD\_TODO':

return {

...state, //保留原来的

todos: state.concat([{ text: action.text, completed: false }]) //改变的

};

return ;

case ''SET\_VISIBILITY\_FILTER'':

return {

...state,

visibilityFilter: action.filter

};

default:

return state;

}}

listener

//监听器，解决state改变后，如何通知页面

store.subscribe(listener);

1. **三大原则**

##### 单一数据源

****整个应用的**[state](https://www.redux.org.cn/docs/Glossary.html" \l "state)**被储存在一棵 object tree 中，并且这个 object tree 只存在于唯一一个**[store](https://www.redux.org.cn/docs/Glossary.html" \l "store)**中。****

单一的 state tree ，以前难以实现的如“撤销/重做”这类功能也变得轻而易举。

****//TODO 添加图示和源代码截图说明****

##### **State 是只读的**

****唯一改变 state 的方法就是触发**[action](https://www.redux.org.cn/docs/Glossary.html" \l "action)**，action 是一个用于描述已发生事件的普通对象。****

 Action 就是普通对象而已，因此它们可以被日志打印、序列化、储存、后期调试或测试时回放出来。

##### **使用纯函数来执行修改**

刚开始你可以只有一个 reducer，随着应用变大，你可以把它拆成多个小的 reducers，分别独立地操作 state tree 的不同部分。

//TODO 思考，reducer 不一定就是按照页面来划分，是否是按照state的管理来划分的？

单一数据源、只有action可修改state、纯函数（数据可回放），这3个特性就决定了可以实现撤销、重做、回放功能

1. **中间件**

[redux-thunk](http://github.com/gaearon/redux-thunk" \t "https://www.redux.org.cn/docs/introduction/_blank) — 用最简单的方式搭建异步 action 构造器

[redux-promise](https://github.com/acdlite/redux-promise" \t "https://www.redux.org.cn/docs/introduction/_blank) — 遵从 [FSA](https://github.com/acdlite/flux-standard-action" \t "https://www.redux.org.cn/docs/introduction/_blank) 标准的 promise 中间件

[redux-axios-middleware](https://github.com/svrcekmichal/redux-axios-middleware" \t "https://www.redux.org.cn/docs/introduction/_blank) — 使用 axios HTTP 客户端获取数据的 Redux 中间件

[redux-observable](https://github.com/blesh/redux-observable/" \t "https://www.redux.org.cn/docs/introduction/_blank) — Redux 的 RxJS 中间件

[redux-rx](https://github.com/acdlite/redux-rx" \t "https://www.redux.org.cn/docs/introduction/_blank) — 给 Redux 用的 RxJS 工具，包括观察变量的中间件

[redux-logger](https://github.com/fcomb/redux-logger" \t "https://www.redux.org.cn/docs/introduction/_blank) — 记录所有 Redux action 和下一次 state 的日志

[redux-immutable-state-invariant](https://github.com/leoasis/redux-immutable-state-invariant" \t "https://www.redux.org.cn/docs/introduction/_blank) — 开发中的状态变更提醒

[redux-unhandled-action](https://github.com/socialtables/redux-unhandled-action" \t "https://www.redux.org.cn/docs/introduction/_blank) — 开发过程中，若 Action 未使 State 发生变化则发出警告

[redux-analytics](https://github.com/markdalgleish/redux-analytics" \t "https://www.redux.org.cn/docs/introduction/_blank) — Redux middleware 分析

[redux-gen](https://github.com/weo-edu/redux-gen" \t "https://www.redux.org.cn/docs/introduction/_blank) — Redux middleware 生成器

[redux-saga](https://github.com/yelouafi/redux-saga" \t "https://www.redux.org.cn/docs/introduction/_blank) — Redux 应用的另一种副作用 model

redux-saga 采用了另外一种思路，它没有把异步操作放在 action creator 中，也没有去处理 reductor，而是把所有的异步操作看成“线程”，可以通过普通的action去触发它，当操作完成时也会触发action作为输出。saga 的意思本来就是一连串的事件。

[redux-action-tree](https://github.com/cerebral/redux-action-tree" \t "https://www.redux.org.cn/docs/introduction/_blank) — Redux 的可组合性 Cerebral-style 信号

[apollo-client](https://github.com/apollostack/apollo-client" \t "https://www.redux.org.cn/docs/introduction/_blank) — 针对 GraphQL 服务器及基于 Redux 的 UI 框架的缓存客户端

## **Dva //TODO 单独说明**

dva 首先是一个基于 redux 和 redux-saga 的数据流方案，然后为了简化开发体验，dva 还额外内置了 react-router 和 fetch，所以也可以理解为一个轻量级的应用框架。

简单理解，就是让使用 react-redux 和 redux-saga 编写的代码组织起来更合理，维护起来更方便。

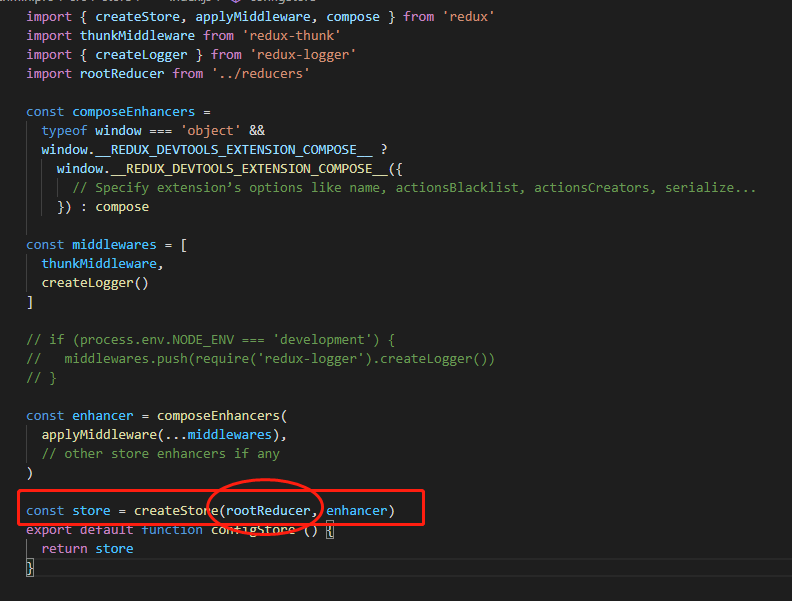
1. **Redux如何工作的**

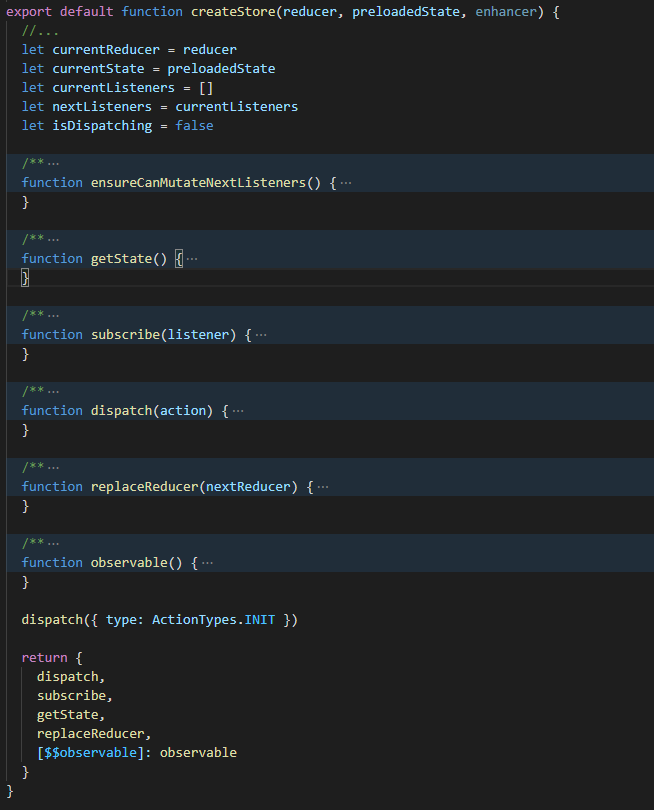
核心为Store类

todo 拿健康小程序项目代码做示例



Store





返回的store对象主要包含以下内容：

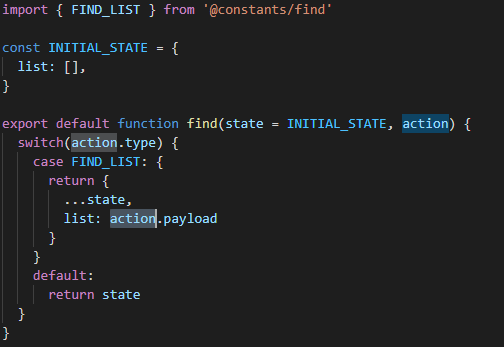
subscribe: store的订阅函数，用于component组件的注册监控函数

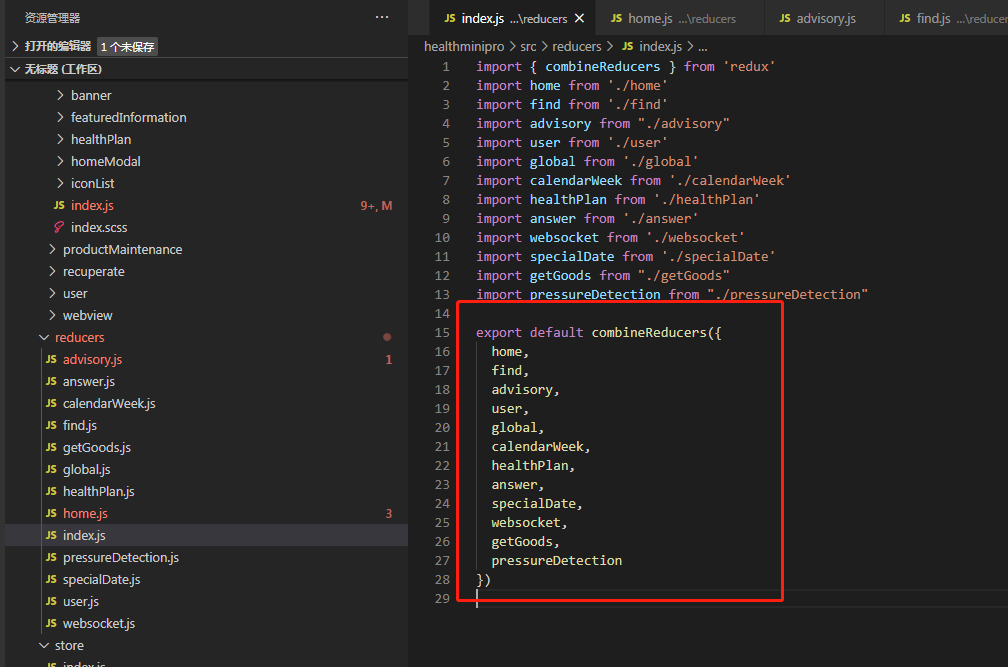
getState:获取store中的state树

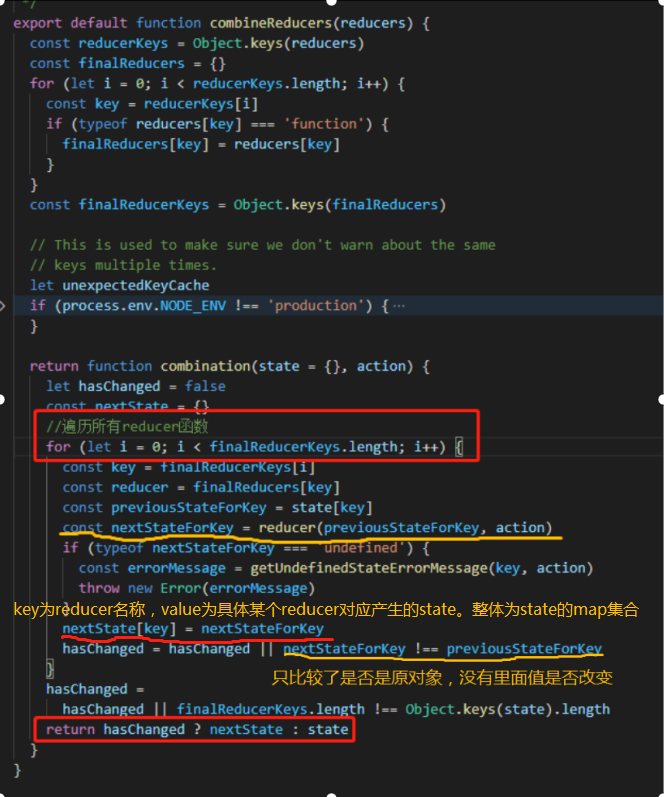
dispatch:触发action

Reducer

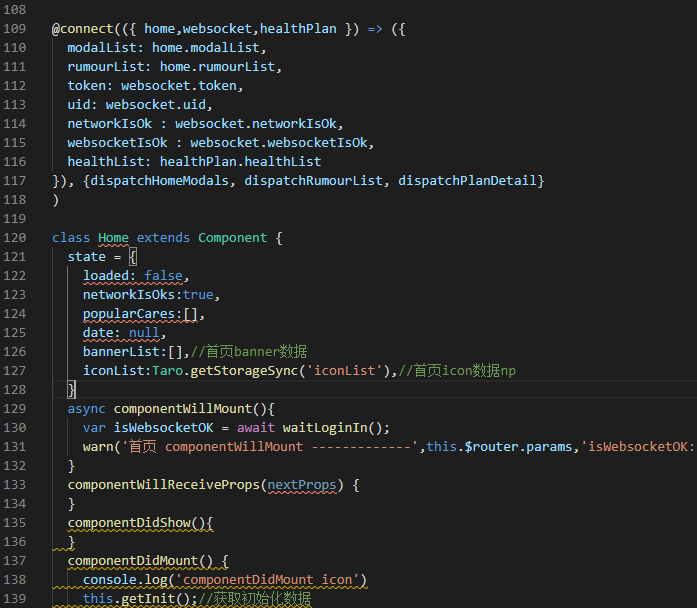
1. 根据业务功能或页面模块定义多个reducer方法
2. 使用combineReducers函数生成一个汇总的reducers函数对象

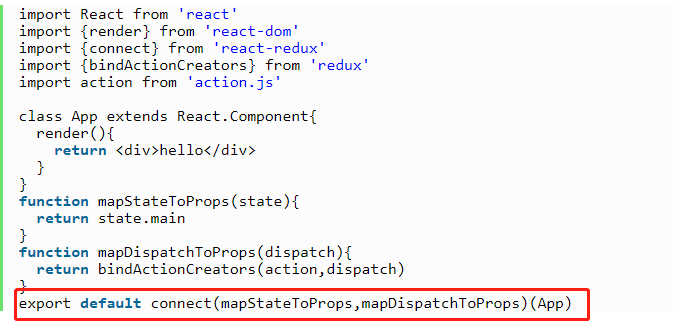






Components





@connect(...)

class Home extends Component {

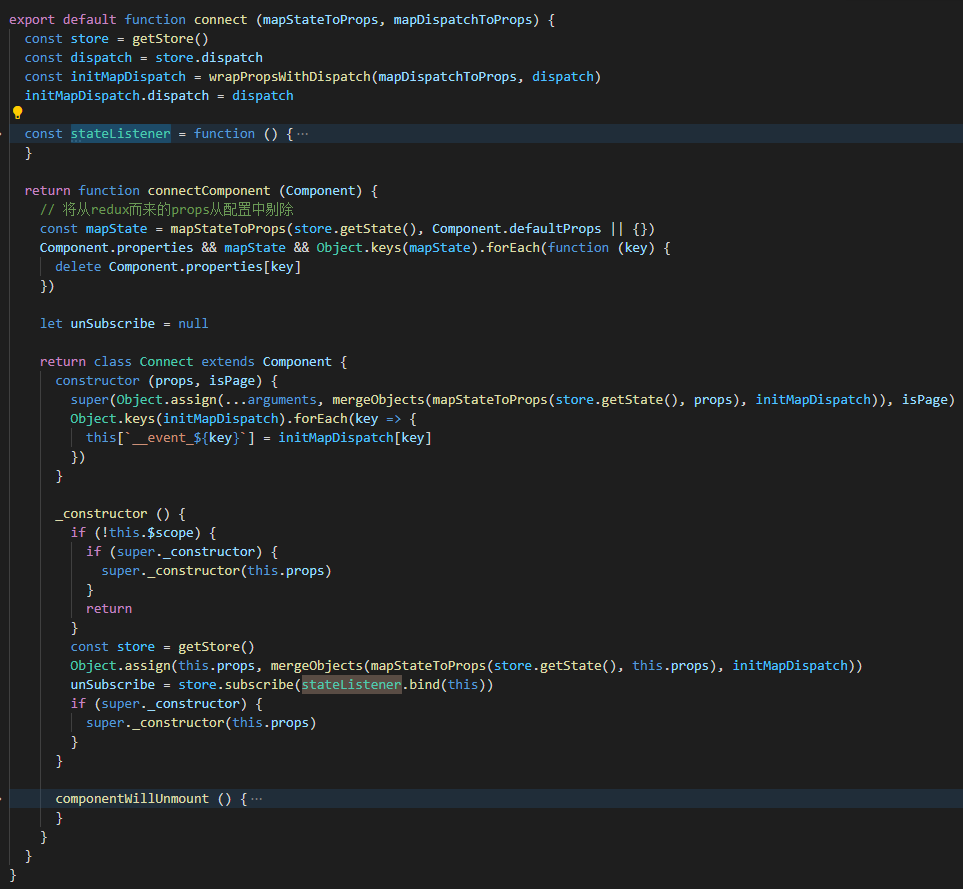
...

}

上面是简单写法

connect(mapStateToProps,mapDispatchToProps)(App)返回了一个根据原始组件，生成的一个新组件，这个组件可以使用redux的特性

Connect的作用



1. 本身是一个函数，返回值也是一个函数
2. 返回的函数可以传入一个组件作为参赛，调用后生成一个新的组件

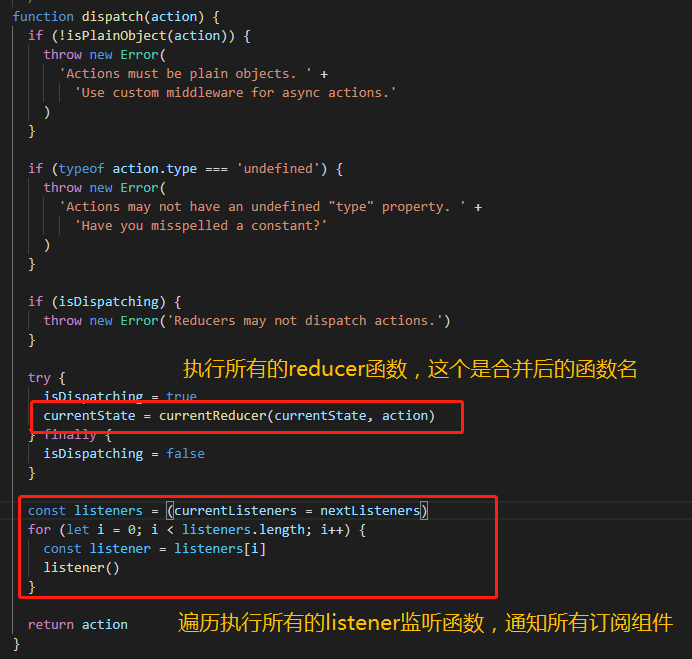
具体主要流程：

1. 生成监听函数stateListener
2. 生成一个新的组件类，在新的组建类的构造函数中调用store的订阅监控函数的方法，传入刚生成的监听函数stateListener

store.subscribe(stateListener.bind(this))

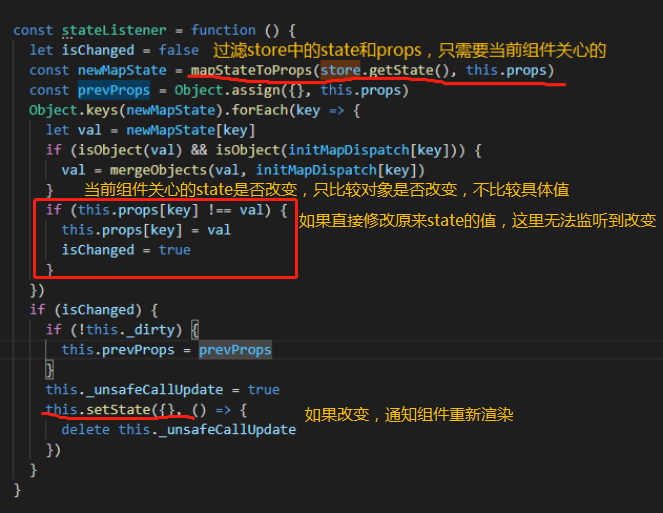
Action

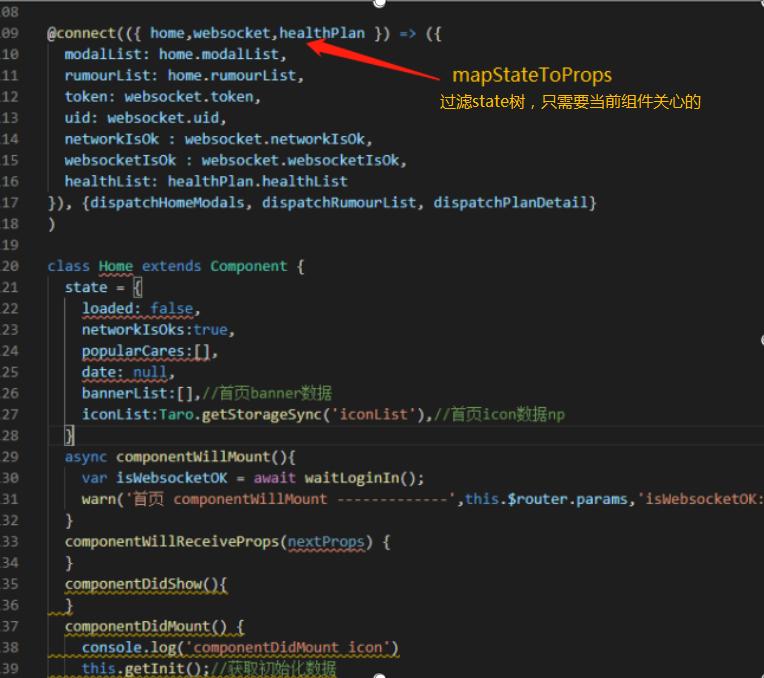
store.dispatch(action)



1. 执行所有reducer函数，生成新的state状态树
2. 执行所有（组件的）监听函数

Listener





1. 获取并过滤store中的state树，只需要当前组件关心的state
2. 逐个判断state树中，当前组件关心的所有子state的对象是否改变，只比较对象是否改变，不关心对象内的值是否改变
3. 如果有改变，通知当前组件重新渲染，调用setState

浅比较（对象地址比较）：

1. Listener监听函数只比较执行reducers后的state是否和原来是同一个对象，不关心对象内部的值是否改变，所有reducer函数中，必须返回新的对象，不能修改原对象内部的值。
2. Reducers执行的过程中，每个reducer生成的子state也只和原state进行对象本身是否改变的比较，比关心对象内部的值。