**Redux教程**

**Redux（架构）部分**

**1.redux简介**

首先明确一点，Redux是一个有用的架构，但不是非用不可。事实上，大多数情况，你可

以不用它，只用React就够了。

如果你的UI层非常简单，没有很多互动，Redux就是不必要的，用了反而增加复杂性。

**三大原则：**

**A：单一数据源：**

**整个应用的**[**state**](http://www.redux.org.cn/docs/Glossary.html#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**](http://www.redux.org.cn/docs/Glossary.html#store)**中。**

**B：State 是只读的：**

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

**对象。**

**C：使用纯函数来执行修改：**

**为了描述 action 如何改变 state tree ，你需要编写**[**reducers**](http://www.redux.org.cn/docs/Glossary.html#reducer)**。**

**（1）不需要使用redux**

A：用户的使用方式非常简单

B：用户之间没有协作

C：不需要与服务器大量交互，也没有使用WebSocket

D：视图层（View）只从单一来源获取数据

**（2）需要使用redux（多交互、多数据源。）**

A：用户的使用方式复杂

B：不同身份的用户有不同的使用方式（比如普通用户和管理员）

C：多个用户之间可以协作

D：与服务器大量交互，或者使用了WebSocket

E：View要从多个来源获取数据

**从组件角度看，如果你的应用有以下场景，可以考虑使用 Redux。**

A：某个组件的状态，需要共享

B：某个状态需要在任何地方都可以拿到

C：一个组件需要改变全局状态

D：一个组件需要改变另一个组件的状态

发生上面情况时，如果不使用 Redux 或者其他状态管理工具，不按照一定规律处理状态

的读写，代码很快就会变成一团乱麻。你需要一种机制，可以在同一个地方查询状态、改变

状态、传播状态的变化。

**2.设计思想**

Redux 的设计思想很简单，就两句话。

**（1）Web 应用是一个状态机，视图与状态是一一对应的。**

**（2）所有的状态，保存在一个对象里面。**

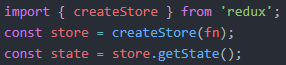
**3.基本概念和 API**

**（1）state**

Store对象包含所有数据。如果想得到某个时点的数据，就要对 Store 生成快照。这种时点的数据集合，就叫做 State。

app中的状态存放的地方，并且state是只读的，不同于React，Redux中state的更改，其实是创建了一个全新的state。

当前时刻的 State，可以通过store.getState()拿到。



Redux 规定，一个State对应一个View。只要State相同，View 就相同。

你知道State，就知道View是什么样，反之亦然。

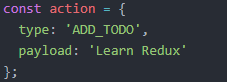
**（2）Action**

State的变化，会导致 View的变化。但是，用户接触不到State，只能接触到View。

所以，State的变化必须是View 导致的。**Action 就是View 发出的通知**，表示 State

应该要发生变化了。**改变 State 的唯一办法，就是使用 Action。它会运送数据到 Store。**

Action 是一个对象。其中的type属性是必须的，表示 Action 的名称。其他属性可以自由设置，社区有一个[规范](https://github.com/acdlite/flux-standard-action)可以参考。



上面代码中，Action 的名称是ADD\_TODO，它携带的信息是字符串Learn Redux。

[**flux下action的标准命名规范**](https://github.com/acdlite/flux-standard-action)**：**

鉴于redux是在flux的基础上的一个升华，很多本质的内容是相似的。因此Redux的

action命名规范是可以参考flux的命名规范的。

一个action应当

A：是一个普通的javascript对象

B：有一个type属性

一个action可以

A：包含一个error属性

B：包含一个payload属性

C：包含一个meta属性

D：包含一个index

一个action不可以包含这四个属性以外的其他属性。

**type：**

action中的type标识通常是向消费者说明action的属性。通常情况下，type是一个字符串常量或者是一个符号标记。如果两个type相同，那么他们必须全等(===)。这里的意思是，type具有唯一性和标志性，用来标记action。

**payload:**

payload标识是可选的，而且可以是任何类型的数据。它表示这个action携带的内容。任

何一个不属于type和action状态的值都应该属于payload域。

通常情况下，如果error是true，那么payload应该是一个error对象。

**error：**

error标识是可选的，而且当该action表示一个报错时，其值可以是true。当一个action

的error域值为true时，它等价于一个被拒绝的promise。通常情况下，这是的payload

值是一个error对象。

如果一个error有其它的非true的值，包括undefined和null，那么这个action就不应

当被解读为是一个error。

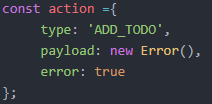
**meta：**

meta标识是可选的，而且可以是任何类型的数据。它旨在保存一切不属于payload的额

外信息。

**index：**

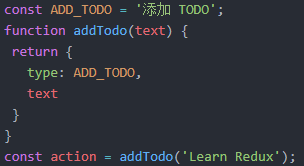
表示用户完成任务的动作序列号, 而实际项目中一般会在新建数据的时候生成唯一的 ID 作为数据的引用标识。



**（3）Action Creator（action 创建函数）**

View 要发送多少种消息，就会有多少种 Action。如果都手写，会很麻烦。可以定义一个

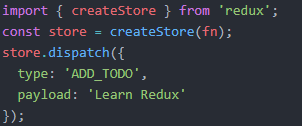
函数来生成 Action，这个函数就叫 Action Creator。



addTodo函数就是一个 Action Creator。

**（4）store.dispatch()**

store.dispatch()是View发出Action的唯一方法。



结合Action Creator，可以改写成:

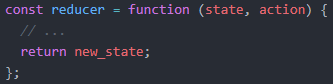


**（5）Reducer**

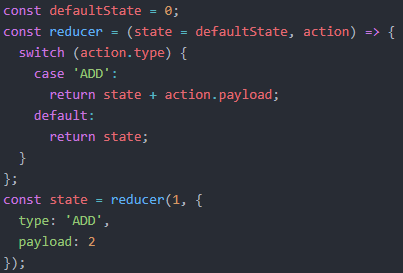
Store 收到Action以后，必须给出一个新的State，这样View才会发生变化。这种

State 的计算过程就叫做 Reducer。

Reducer 是一个函数，它接受Action和当前State作为参数，返回一个新的State。



例如：



实际应用中，Reducer 函数不用像上面这样手动调用store.dispatch方法会触发。

**Reducer 的自动执行**。为此，Store 需要知道 Reducer 函数，做法就是在生成Store 的

时候，将 Reducer 传入createStore方法。



上面代码中，createStore接受 Reducer 作为参数，生成一个新的 Store。以后每当

store.dispatch发送过来一个新的 Action，就会自动调用 Reducer，得到新的State。

**（6）纯函数**

Reducer 函数最重要的特征是，它是一个纯函数。也就是说，只要是同样的输入，必定得

到同样的输出。

纯函数是函数式编程的概念，必须遵守以下一些约束。

A：不得改写参数

B：不能调用系统 I/O 的API

C：不能调用Date.now()或者Math.random()等不纯的方法，因为每次会得到不一样的结果。

由于 Reducer是纯函数，就可以保证同样的State，必定得到同样的View。

所以，Reducer函数里面不能改变State，必须返回一个全新的对象，请参考下面的写法。

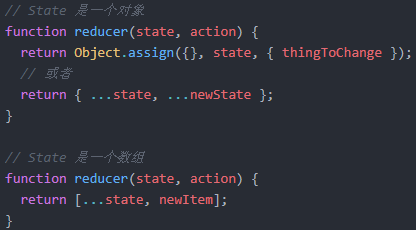
最好把State对象设成只读。你没法改变它，要得到新的State，唯一办法就是生成一个

新对象。这样的好处是，任何时候，与某个View对应的State总是一个不变的对象。

**不要修改 state。** 使用 [Object.assign()](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Object/assign" \t "_blank) 新建了一个副本。

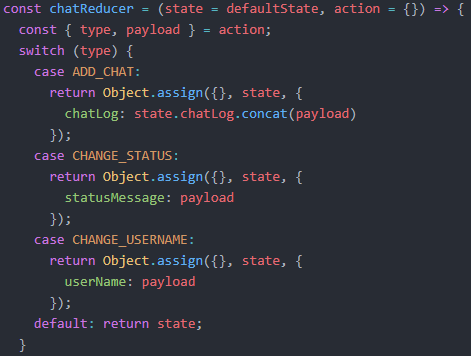
不能这样使用 Object.assign(state, { visibilityFilter: action.filter })，因为它会改变第一个参数的值。你**必须**把第一个参数设置为空对象。你也可以开启对ES7提案[对象展开运算符](http://www.redux.org.cn/docs/recipes/UsingObjectSpreadOperator.html)的支持, 从而使用 { ...state, ...newState } 达到相同的目的。

**在 default 情况下返回旧的 state。**遇到未知的 action 时，一定要返回旧的state。



**Reducer 的拆分**

Reducer 函数负责生成 State。由于整个应用只有一个 State 对象，包含所有数据，对于大型应用来说，这个 State 必然十分庞大，导致 Reducer 函数也十分庞大。



三种 Action 分别改变 State 的三个属性。

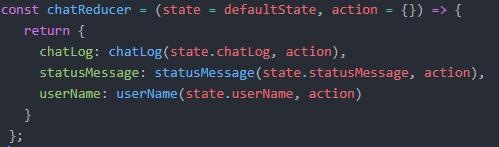
ADD\_CHAT：chatLog属性

CHANGE\_STATUS：statusMessage属性

CHANGE\_USERNAME：userName属性

这三个属性之间没有联系，这提示我们可以把 Reducer 函数拆分。不同的函数负责处理

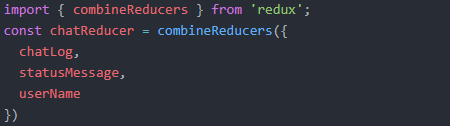
不同属性，最终把它们合并成一个大的 Reducer 即可。



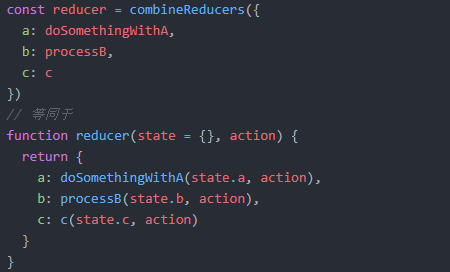
这样一拆，Reducer 就易读易写多了。而且，这种拆分与 React 应用的结构相吻合：

一个 React 根组件由很多子组件构成。这就是说，子组件与子 Reducer 完全可以对应。

Redux 提供了一个**combineReducers**方法，用于 Reducer 的拆分。你只要定义各个子Reducer 函数，然后用这个方法，将它们合成一个大的 Reducer。



这种写法有一个前提，就是 State 的属性名必须与子 Reducer 同名。如果不同名，就要采用下面的写法。



你可以把所有子 Reducer 放在一个文件里面，然后统一引入。



**注意每个reducer只负责管理全局 state 中它负责的一部分。每个reducer**

**的 state参数都不同，分别对应它管理的那部分 state数据。**

**（7）Store**

Store 就是**保存数据**的地方，你可以把它看成一个容器。整个应用只能有一个 Store。

Redux 提供**createStore**这个函数，用来生成 Store。

****

[createStore()](http://www.redux.org.cn/docs/api/createStore.html) 的第二个参数是可选的, 用于设置 state 初始状态。这对开发同构应用时

非常有用，服务器端redux应用的state结构可以与客户端保持一致, 那么客户端可以

将从网络接收到的服务端state直接用于本地数据初始化。

**功能：**

维持应用的 state。

提供 [getState()](http://www.redux.org.cn/docs/api/Store.html" \l "getState) 方法获取 state。

提供 [dispatch(action)](http://www.redux.org.cn/docs/api/Store.html#dispatch) 方法更新 state。

通过 [subscribe(listener)](http://www.redux.org.cn/docs/api/Store.html#subscribe) 注册监听器。

通过 [subscribe(listener)](http://www.redux.org.cn/docs/api/Store.html#subscribe) 返回的函数注销监听器。

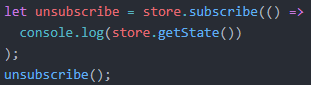
**store.subscribe()**设置监听函数，一旦 State 发生变化，就自动执行这个函数。



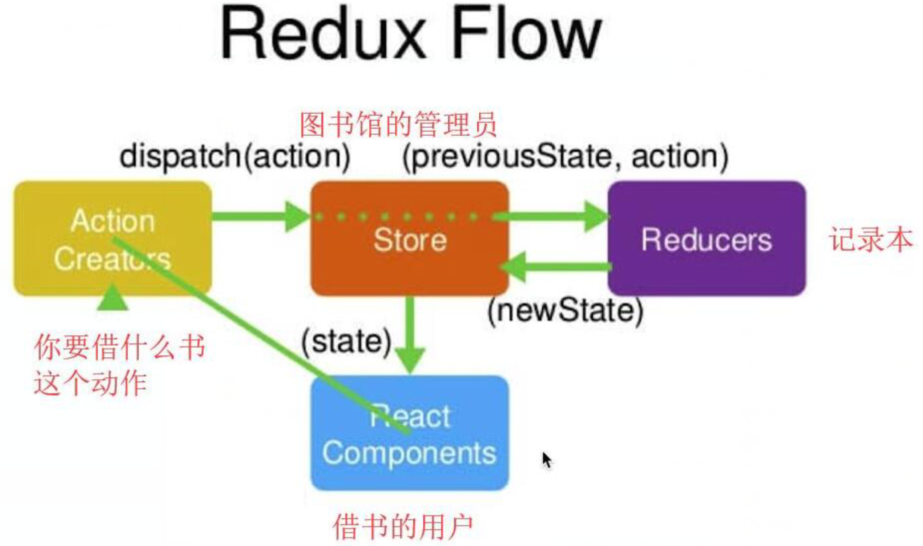
显然，只要把View的更新函数（对于React项目，就是组件的render方法或setState

方法）放入listen，就会实现 View 的自动渲染。

store.subscribe方法返回一个函数，调用这个函数就可以解除监听。



**（8）工作流程**

****

**说明：**

**A：户发出Action**

**store.dispatch(action);**

**B：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 。**

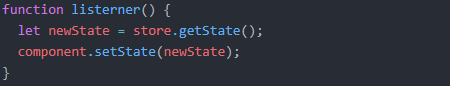
**let nextState = todoApp(previousState, action);**

**C：State 一旦有变化，Store 就会调用监听函数。**

**store.subscribe(listener);**

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。

****

**（9）数据流**

**严格的单向数据流是 Redux 架构的设计核心。**

Redux 应用中数据的生命周期遵循下面4个步骤：

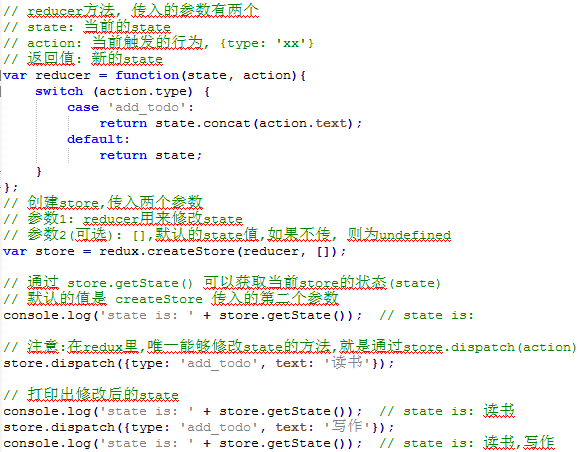
**A：调用** [store.](http://www.redux.org.cn/docs/api/Store.html" \l "dispatch)[dispatch(action)](http://www.redux.org.cn/docs/api/Store.html" \l "dispatch)。

B：**Redux store调用传入的reducer函数。**

C：**根 reducer应该把多个子reducer输出合并成一个单一的state树。**

D：**Redux store保存了根reducer返回的完整state树。**

**（10）redux的核心概念就是store**、**action**、**reducer**



**（11）react-redux库connect与Provider**

先简单说一下redux和react是怎么配合的。react-redux提供了connect和Provider

两个好基友，它们一个将组件与redux关联起来，一个将store传给组件。

组件通过dispatch发出action，store根据action的type属性调用对应的reducer并传入state和这个action，reducer对state进行处理并返回一个新的state放入store，connect监听到store发生变化，调用setState更新组件，此时组件的props也就跟着变化。

值得注意的是connect，Provider，mapStateToProps,mapDispatchToProps是

react-redux提供的，redux本身和react没有半毛钱关系，它只是数据处理中心，没有

和react产生任何耦合，是react-redux让它们联系在一起。

一旦我们选择使用了这个react-redux库，那么我们的组件概念就要加以区分了，从现在起我们的组件分为展示组件和容器组件两种 。

**A：展示组件（presentational component）**

也叫UI组件、纯组件 。

特点如下:

负责UI显示。

无状态不使用this.state。

数据来自this.props。

不使用任何redux的API。

展示组件其实就是把我们的普通组件的数据与逻辑抽离出来。

**B：容器组件（container component）**

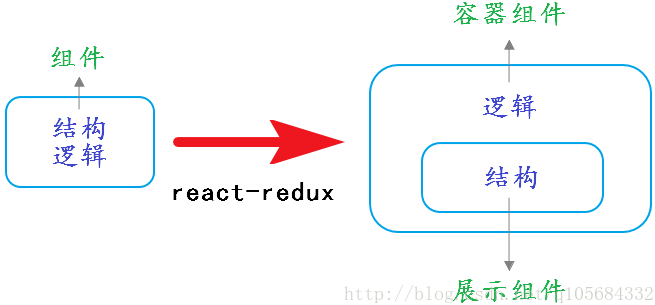
特点如下：

负责管理数据和业务逻辑。

带有内部状态。

使用redux的API。

容器组件是由我们react-redux库的API通过展示组件生成的。



我们原来是将结构和逻辑都装在一个组件中 。

现在将这个组件继续拆成负责视图的组件和负责逻辑数据的组件。

**这样做有如下优点：**

**理解 ：**数据与逻辑分开，更便于我们理解。

**分离 ：**必须将标签拆分，可用性更强。

**重用 ：**一个展示组件可以搭配不同容器组件。

**视图 ：**展示组件可以放到单独页面中调整UI。

**react-redux库的核心：connect()方法与Provider组件**

**A：connect**（连接展示组件与容器组件。）

为了加以区分，我用Container表示容器组件，用Component表示展示组件 。



结构就是这个样子：



但是容器组件中并没有数据和逻辑，只是一具空壳，毫无意义。

所以我们还需要向这个connect函数中传入两个参数 ：

它接收两个值作为参数：（实际是四个，另外两个不常用暂时不讲）

**mapStateToProps**（输入逻辑）

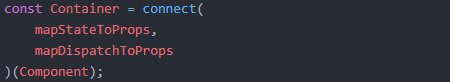
负责将通过state获得的数据映射到展示组件的this.props。

**mapDispatchToProps**（输出逻辑）

负责将用户操作转化为Action的功能函数映射到展示组件的this.props。

名字就和reducer一样，只是官方的概念性叫法（不过还是蛮形象的）。

使用的时候可以自定义名字（不过一定要语义化），所以完整的用法应该是这样的。

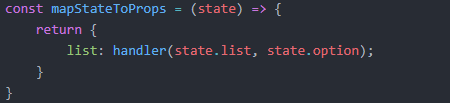


**mapStateToProps**是一个函数，接收参数state对象 。

如果有必要的话，还可以使用第二个参数：容器组件的props属性 。

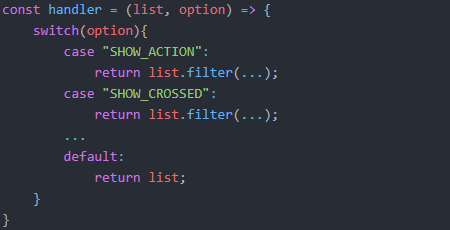
返回一个对象表示state到展示组件props的映射关系。

我们可以自定义一个处理函数返回要传递的数据。



比如说这里handler就是我们的处理函数。

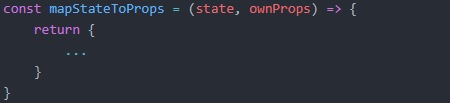
拿我上一篇文章的toDoList待办事项列表为例， 这个handler大概是这样的：



mapStateToProps会订阅store，state更新后，就会触发展示组件重绘 。

不过在connect( )函数中，我们可以省略mapStateToProps ，如果这么做的话，store更新就不会触发展示组件重绘了。

除了state我们还可以使用容器组件的属性props。



如果容器组件的props发生改变的话，同样会触发展示组件重绘。

**mapDispatchToProps**可以是函数也可以是对象。

如果mapDispatchToProps是一个函数，会得到dispatch和ownProps（容器组件的

props对象）两个参数。返回值就是一个表示映射关系的对象 。

用户如何发出Action（比如触发事件）：



**说明：**

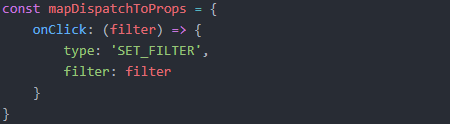
返回对象中的“值”—— () => {dispatch(...)}

表示我们要传递给内部展示组件的函数（函数功能：dispatch一个action）

返回对象中的“键”—— onClick

表示我们在展示组件中可以通过this.props.onClick来获取这个函数。

如果是作为对象的话，就更简单了 ：



这个对象的值是一个函数，它被认为是一个Action Creator 。

函数的参数可以填入容器组件的props 。

返回的Action会由redux自动dispatch。

以上的两种写法是等价。

**B：Provider**

在完成了Container与Componet的连接后，实现了Container的管理数据与业务逻辑之后还没完。

还有问题 ：我们使用了mapStateToProps，它的参数是state ，也就是说需要传入state。

如果我们手动将state对象一层一层的传入容器组件，应用小还好说，大应用深层的组件简直累死了，绝对让你传到怀疑人生。

react-redux提供了Provider组件让我们省了不少功夫，它就相当于我们整体的容器组件。

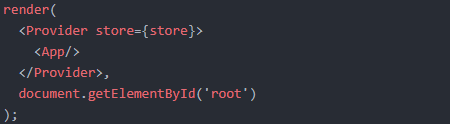
用法就是在我们根组件外部嵌套一层Provider，传入store。这样所有的子组件都可以开

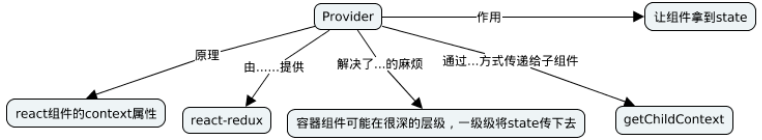
心地拿到state了。

**Provider组件主要有以下下两个作用：**

A：在整个应用上包一层，使整个应用成为Provider的子组件。

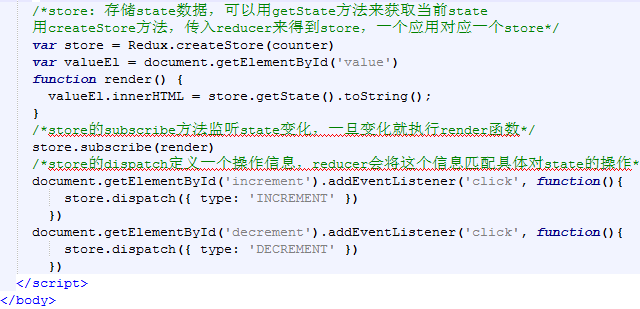
B：接收Redux的store作为props，通过context对象传递给子组件，所有的子组件都可以取得store。只是一个传递数据的组件。





**例子1：加减计数器的简单实现**

****

****

**运行结果：**

****

**例子2：Hello world的实现**

****

****

****

跳转到画面：World。点击按钮后，显示Click Me

开始画面：点击Hello

**运行结果：**

**  **

**4.中间件与异步操作**

**（1）中间件的概念**

在使用redux时，改变store state的一个固定套路是调用store.dispatch(action)方法，将action送到reducer中。

所谓中间件，就是在**dispatch发送action** 和 **action到达reducer** 之间，加入一些

中间层，对action进行处理。

在之前学习redux的异步操作时，用到的redux-thunk就是一个中间件。action抵达reducer之前先对其进行判断，如果是对象就直接送到reducer；如果是函数就执行。

**中间件特点：**

**A：是一个独立的函数，与其他中间件没有依赖关系**

**B：中间件可以有多个，按顺序组合使用**

**C：有统一的接口**

中间件说白了就是一些第三方的插件，用来增强dispatch的功能，让我们可以在action到达reducer之前对action作出判断、改造、log下信息、实现复杂逻辑等。

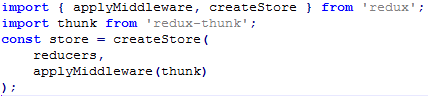
当我们使用了多个中间件时，action依次从前一个中间件传到下一个中间件，最后抵达

reducer。也可能在某个中间件进行处理后，发现没必要传到reducer了，就不传给下一

个中间件（相当于丢弃，不再去处理）。

**（2）applyMiddleware(...middlewares)**

是Redux的原生方法，作用是将所有中间件组成一个数组，依次执行。

****

直接将thunk中间件引入，放在applyMiddleware方法之中，传入createStore方法，

就完成了store.dispatch()的功能增强。即可以在reducer中进行一些异步的操作。

applyMiddleware(...middlewares)

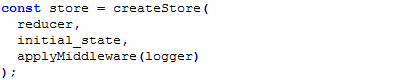
告诉redux我们会用到那些中间件，比如说要用到基础的异步，我们会用到thunk中间件。

其实applyMiddleware就是Redux的一个原生方法，将所有中间件组成一个数组，依次

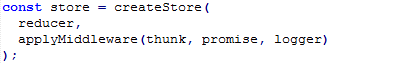
执行。中间件多了可以当做参数依次传进去。

注意点：

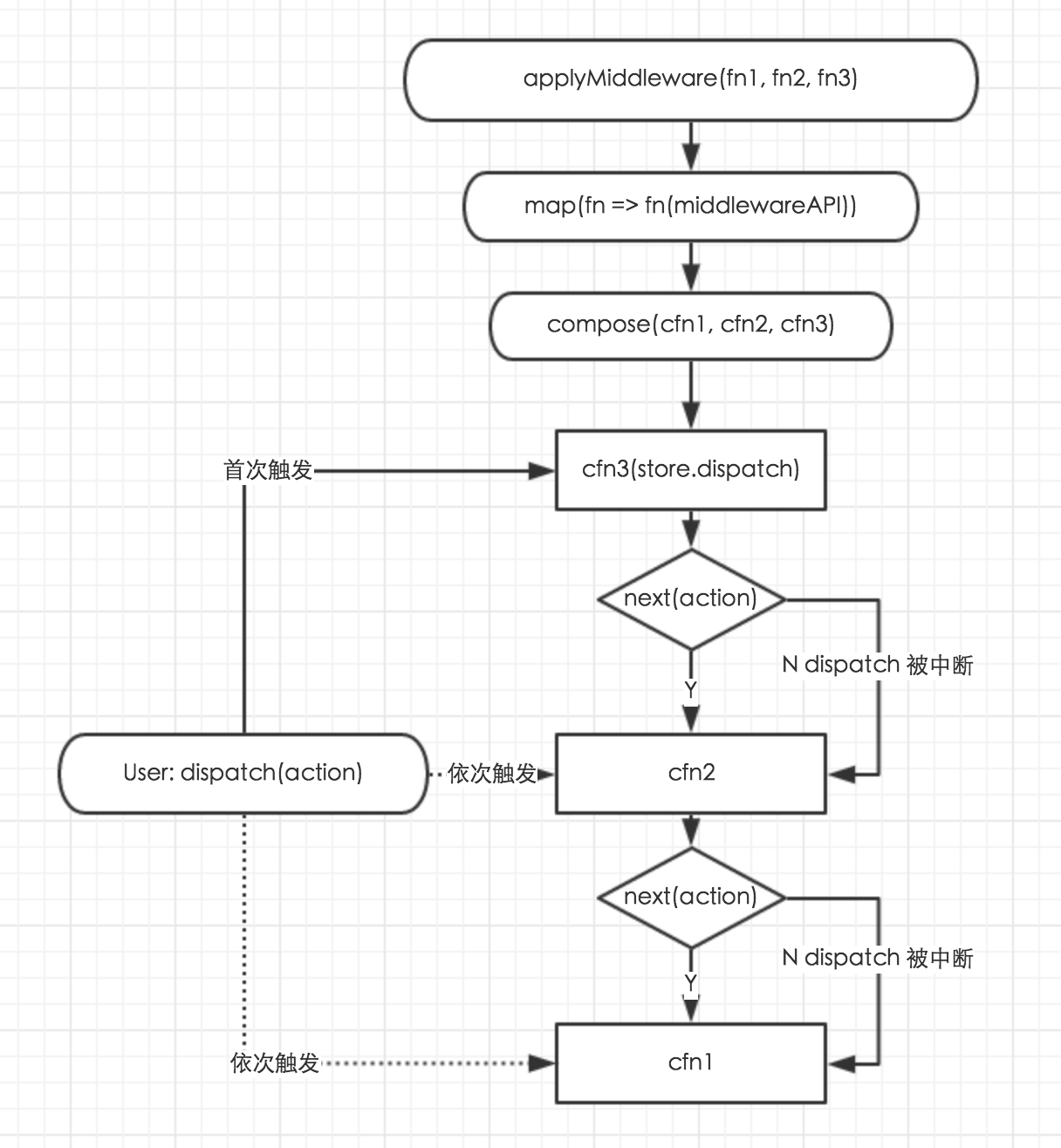
A：createStore方法可以接受整个应用的初始状态作为参数。



B：中间件的次序有讲究。

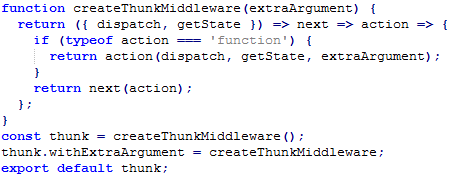


applyMiddleware方法的三个参数，就是三个中间件。有的中间件有次序要求，使用前要查一下文档。比如，logger就一定要放在最后，否则输出结果会不正确。

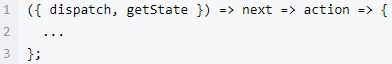


**（3）redux-thunk 中间件**

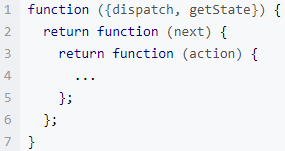
源码：



createThunkMiddleware方法里返回了一个奇怪的东西：



换一种写法：



第一个（最外层）方法的参数是一个包含dispatch和getState字段（方法）的对象，其

实就是store对象，所以也可以写成：



参数next是一个方法，这个方法的作用是通知下一个Redux中间件对这次的action进行

处理： next(action)，如果一个中间件中没有执行next(action)，则action会停止向后

续的中间件传递，并阻止reducer的执行（store将不会因为本次的action而更新）。

参数action就不用多说了，就是当前被触发的action。

在ReduxThunk这个中间件中，做的处理很简单：判断当前的action是否为一个方法，

如果是，就执行action这个方法，并将store.dispatch与store.getState方法作为参数

传递给action方法；如果不是，则执行next(action)将控制权转移给下一个中间件（如果

有）。

所以当我们给store.dispatch方法传入一个方法的时候，ReduxThunk就会去执行这个

方法，以达到自由控制action触发流程的一个目的。

**（4）异步action**

原来的store.dispatch方法只能接收一个普通的action对象作为参数，当我们加入了

ReduxThunk这个中间件之后，store.dispatch还可以接收一个方法作为参数，这个方法

会接收到两个参数，第一个是dispatch，等同于store.dispatch，第二个是getState，

等同于store.getState。

例如：现在可以这样来触发INCREASE



**例如：计数器的实现**



