Redux学习笔记

1．State:状态树，可以包含数组，表示整个的状态，除了数组外还有一些表示整体状态的标识符 。state 的形式取决于你，可以是基本类型、数组、对象、

\* 甚至是 Immutable.js 生成的数据结构。惟一的要点是

\* 当 state 变化时需要返回全新的对象，而不是修改传入的参数

2.store包括了state和reducer们，但state是在运行过程中自动生成的，所以我们能事先写好的就是reducer们了，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

3.action 由你来抽象出来，就是一个事件，包含了哪些要素，自己想

4.reducer就是对不同的action进行反应，以达到改变状态树的目的

5.唯一改变state的办法是触发action

6. 不要把 Redux 当作万灵丹，如果你的应用没那么复杂，就没必要用它。另一方面，Redux 只是 Web 架构的一种解决方案，也可以选择其他方案

7. 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。

8. 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。

当前时刻的 State，可以通过store.getState()拿到。

9. 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

10. Action 描述当前发生的事情。改变 State 的唯一办法，就是使用 Action。它会运送数据到 Store。

11. store.dispatch()是 View 发出 Action 的唯一方法。上面代码中，store.dispatch接受一个 Action 对象作为参数，将它发送出去

12. 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。

const store = createStore(reducer);

12. 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。

你应该把要做的修改变成一个普通对象，这个对象被叫做 action，而不是直接修改 state，

即：action是一个对象

13. 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。由于 Reducer 是纯函数，就可以保证同样的State，必定得到同样的 View。但也正因为这一点，Reducer 函数里面不能改变 State，必须返回一个全新的对象

14. 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。

15. 3.8 store.subscribe()

Store 允许使用store.subscribe方法设置监听函数，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。

16. 可以发现 Store 提供了三个方法。

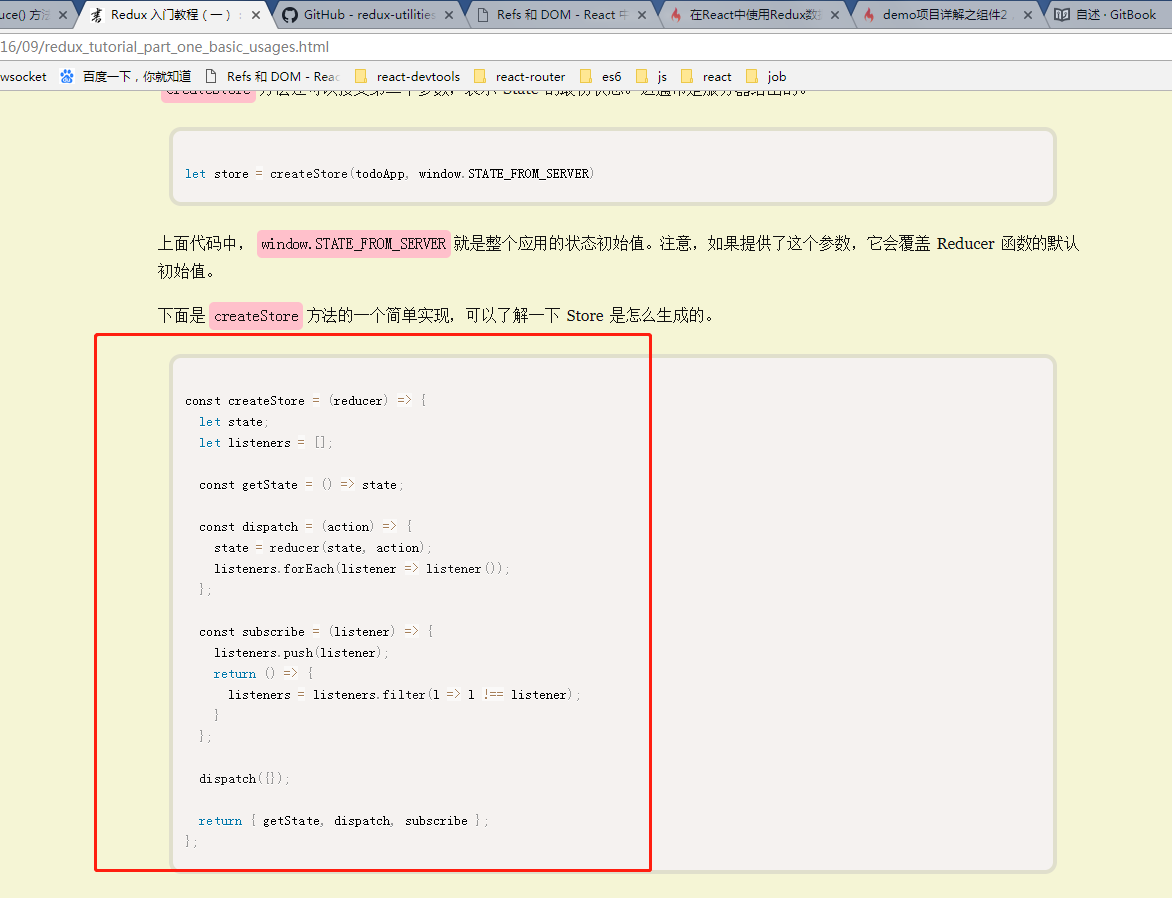
store.getState()

store.dispatch()

store.subscribe()

17.没懂

18.没看懂，但应该不用懂



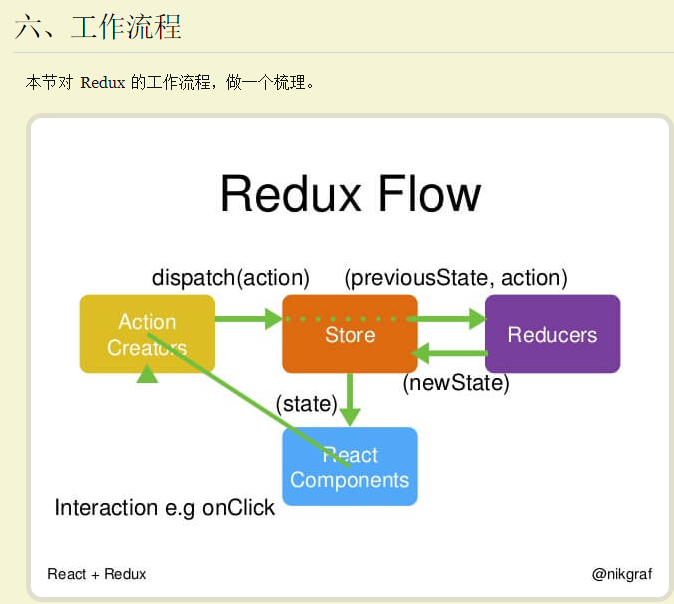
19. Reducer 函数负责生成 State。由于整个应用只有一个 State 对象，包含所有数据，对于大型应用来说，这个 State 必然十分庞大，导致 Reducer 函数也十分庞大。State 的三个属性之间没有联系，这提示我们可以把 Reducer 函数拆分。不同的函数负责处理不同属性，最终把它们合并成一个大的 Reducer 即可。

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。

20.



21.



22.react组件还可以这么定义，传参进去



22.中间件的概念

为了理解中间件，让我们站在框架作者的角度思考问题：如果要添加功能，你会在哪个环节添加？--action . 中间件就是一个函数，对store.dispatch方法进行了改造，在发出 Action 和执行 Reducer 这两步之间，添加了其他功能。

常用的中间件都有现成的，只要引用别人写好的模块即可

23.较透彻地理解异步和中间件

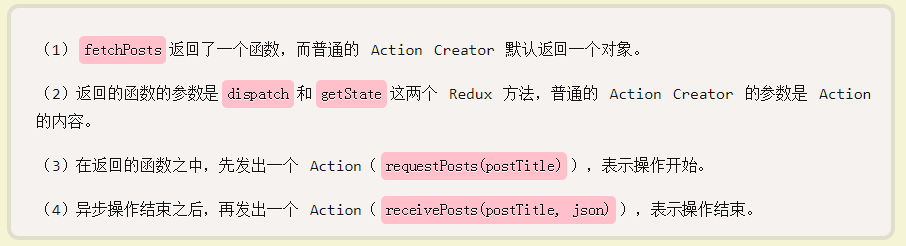
（1）中间件的雏形：即重写store.dispatch,在dispatch前和后都加一些其他功能。真实情况下是前后加dispatch, 不过中间件都有现成的，他们的作用都是改写store.dispatch函数。

中间件的用法是：作为createStore的第三个参数（要都用到applyMiddlewares）

（2）实现异步的过程中要用到中间件

（3）实现异步的过程（第一种异步的方法）：

1. 自己写的具体实现异步功能的函数，即action creator，（如例子里的那个fetchPost()函数），不过和平时的有不一样的几点：



1. 中间件作为createStore的第三个参数，目的是为了让store.dispatch能够接受和平时的action creator不一样的action creator.(因为这个action creator返回的是函数)

（4）其他的实现异步的过程见网页

<http://www.ruanyifeng.com/blog/2016/09/redux_tutorial_part_two_async_operations.html>

23. applyMiddlewares这个方法到底是干什么的？

它是 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。下面是它的源码。

24. 理解了中间件以后，就可以处理异步操作了。

同步操作只要发出一种 Action 即可，异步操作的差别是它要发出三种 Action。

25.什么意思？es6的语法吗？

是的，将this.props(对象)的dispatch属性和selectedPost属性分别赋给本地变量dispatch 和selectedPost



26. componentDidMount在哪里重写？在组件定义的函数里吗？是的

27. dispatch可以自己使用吗？不是store的方法？//例子里使用dispatch应该是因为以参数传进去了吧？不清楚



29. 因此，异步操作的第一种解决方案就是，写出一个返回函数的 Action Creator，然后使用redux-thunk中间件改造store.dispatch

30. <http://www.ruanyifeng.com/blog/2016/09/redux_tutorial_part_two_async_operations.html>

阮大神的入门（三）

1. React-Redux 将所有组件分成两大类：UI 组件（presentational component）和容器组件（container component）。
2. 总之，只要记住一句话就可以了：UI 组件负责 UI 的呈现，容器组件负责管理数据和逻辑。
3. 你可能会问，如果一个组件既有 UI 又有业务逻辑，那怎么办？回答是，将它拆分成下面的结构：外面是一个容器组件，里面包了一个UI 组件。前者负责与外部的通信，将数据传给后者，由后者渲染出视图。React-Redux 规定，所有的 UI 组件都由用户提供，容器组件则是由 React-Redux 自动生成。也就是说，用户负责视觉层，状态管理则是全部交给它。//当然，看视频发现，并不是完全交给它，程序员还是得写一些代码用于交互的
4. React-Redux 提供connect方法，用于从 UI 组件生成容器组件。connect的意思，就是将这两种组件连起来。
5. 1）输入逻辑：外部的数据（即state对象）如何转换为 UI 组件的参数

（2）输出逻辑：用户发出的动作如何变为 Action 对象，从 UI 组件传出去。

1. connect方法接受两个参数：mapStateToProps和mapDispatchToProps。它们定义了 UI 组件的业务逻辑。前者负责输入逻辑，即将state映射到 UI 组件的参数（props），后者负责输出逻辑，即将用户对 UI 组件的操作映射成 Action。
2. mapStateToProps是一个函数。它的作用就是像它的名字那样，建立一个从（外部的）state对象到（UI 组件的）props对象的映射关系。

mapStateToProps是一个函数，它接受state作为参数，返回一个对象mapStateToProps会订阅 Store，每当state更新的时候，就会自动执行，重新计算 UI 组件的参数，从而触发 UI 组件的重新渲染。

mapStateToProps的第一个参数总是state对象，还可以使用第二个参数，代表容器组件的props对象。

1. mapDispatchToProps是connect函数的第二个参数，用来建立 UI 组件的参数到store.dispatch方法的映射。也就是说，它定义了哪些用户的操作应该当作 Action，传给 Store。它可以是一个函数，也可以是一个对象。

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。

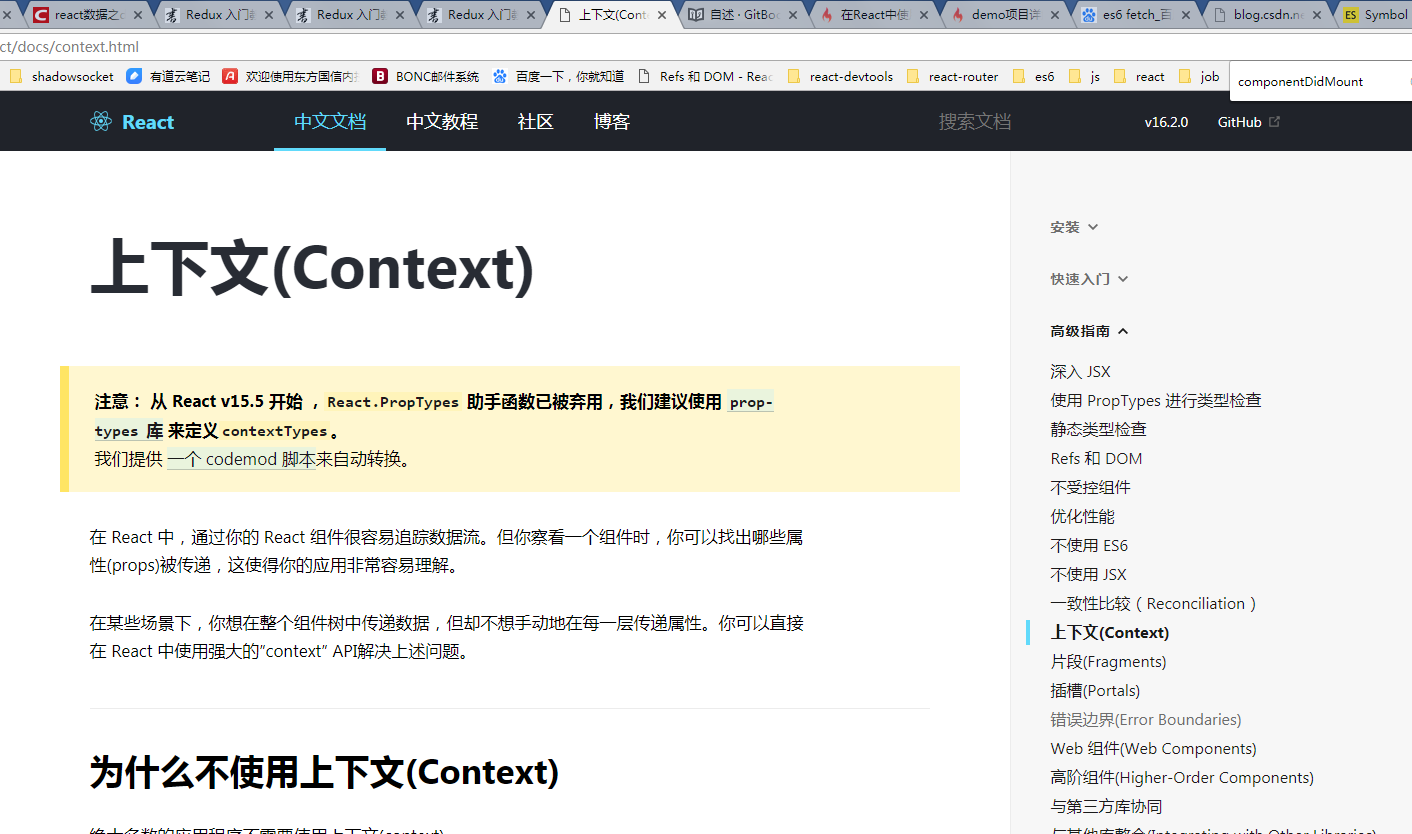
mapDispatchToProps若作为函数，应该返回一个对象，该对象的每个键值对都是一个映射，定义了 UI 组件的参数怎样发出 Action。

如果mapDispatchToProps是一个对象，它的每个键名也是对应 UI 组件的同名参数，键值应该是一个函数，会被当作 Action creator ，返回的 Action 会由 Redux 自动发出

10. <Provider> 组件

connect方法生成容器组件以后，需要让容器组件拿到state对象，才能生成 UI 组件的参数。React-Redux 提供Provider组件，可以让容器组件拿到state。方法是在根组件外面包了一层Provider，这样一来，App的所有子组件就默认都可以拿到state了。

1. 
2. 答案在这：



1. todo



1. react中const {dispatch} = this.props; 这个dispatch属性是怎么获得的？

（1）const {dispatch} = this.props;

这段代码你可以认为是这样：

const dispatch = this.props.dispatch;

那样写是 ES6 的简写形式。

（2）是由connect传入的，当你没有设置mapDispatchToProps的时候默认是会return { dispatch }的，如果你自定义了mapDispatchToProps方法就需要你手动把dispatch返回了//////////////待理解

19. Provider的唯一功能就是传入store对象

**Redux官方文档**

需要redux的原因是：因为react专注于视图层（不是mvc 也不是mvvm模型），所以没有对数据流的操作，因此要完成一个完整的项目需要对数据的处理，即redux.

1.这里的复杂性很大程度上来自于：我们总是将两个难以理清的概念混淆在一起：变化和异步。 我称它们为曼妥思和可乐。如果把二者分开，能做的很好，但混到一起，就变得一团糟。一些库如 React 试图在视图层禁止异步和直接操作 DOM 来解决这个问题。美中不足的是，React 依旧把处理 state 中数据的问题留给了你。Redux就是为了帮你解决这个问题。

2.当应用规模越来越大时，建议使用单独的模块或文件来存放 action。对于小应用来说，使用字符串做 action type 更方便些。不过，在大型应用中把它们显式地定义成常量还是利大于弊的

我们还需要再添加一个 action index 来表示用户完成任务的动作序列号。因为数据是存放在数组中的，所以我们通过下标 index 来引用特定的任务

3. Redux 中只需把 action 创建函数的结果传给 dispatch() 方法即可发起一次 dispatch 过程或者创建一个 被绑定的 action 创建函数 来自动 dispatch，然后直接调用它们store 里能直接通过 store.dispatch() 调用 dispatch() 方法，但是多数情况下你会使用 react-redux 提供的 connect() 帮助器来调用。bindActionCreators() 可以自动把多个 action 创建函数 绑定到 dispatch() 方法上。

4. 记住 actions 只是描述了有事情发生了这一事实，并没有描述应用如何更新 state。

5. Redux 可以用这三个基本原则来描述

（1）单一数据源

整个应用的 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 中。

（2）State 是只读的

唯一改变 state 的方法就是触发 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

（3）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。

5. 因为 reducer 只是函数，你可以控制它们被调用的顺序，传入附加数据，甚至编写可复用的 reducer 来处理一些通用任务，如分页器。

6. 开发复杂的应用时，不可避免会有一些数据相互引用。建议你尽可能地把 state 范式化，不存在嵌套。把所有数据放到一个对象里，每个数据以 ID 为主键，不同实体或列表间通过 ID 相互引用数据。把应用的 state 想像成数据库。这种方法在 normalizr 文档里有详细阐述。例如，实际开发中，在 state 里同时存放 todosById: { id -> todo } 和 todos: array<id> 是比较好的方式，本文中为了保持示例简单没有这样处理。

7. 永远不要在 reducer 里做这些操作：

修改传入参数；

执行有副作用的操作，如 API 请求和路由跳转；

调用非纯函数，如 Date.now() 或 Math.random()。

Redux 首次执行时，state 为 undefined，此时我们可借机设置并返回应用的初始 state。这里一个技巧是使用 ES6 参数默认值语法 来精简代码。(对参数设置默认值)

注意:

1.不要修改 state。 使用 Object.assign() 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的支持, 从而使用 { ...state, ...newState } 达到相同的目的。

2.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。

Object.assign 须知

Object.assign() 是 ES6 特性，但多数浏览器并不支持。你要么使用 polyfill，Babel 插件，或者使用其它库如 \_.assign() 提供的帮助方法。

如上，不直接修改 state 中的字段，而是返回新对象。新的 todos 对象就相当于旧的 todos 在末尾加上新建的 todo。而这个新的 todo 又是基于 action 中的数据创建的。

8.

注意 todos 依旧接收 state，但它变成了一个数组！现在 todoApp 只把需要更新的一部分 state 传给 todos 函数，todos 函数自己确定如何更新这部分数据。这就是所谓的 reducer 合成，它是开发 Redux 应用最基础的模式。

9.reducer拆分合成还需再看：<http://www.redux.org.cn/docs/basics/Reducers.html>

10. 最后，Redux 提供了 combineReducers() 工具类来做上面 todoApp 做的事情，这样就能消灭一些样板代码了。有了它，可以这样重构 todoApp：

combineReducers 接收一个对象，可以把所有顶级的 reducer 放到一个独立的文件中，通过 export 暴露出每个 reducer 函数，然后使用 import \* as reducers 得到一个以它们名字作为 key 的 object：

11.

Store 就是把它们联系到一起的对象。Store 有以下职责：

维持应用的 state；

提供 getState() 方法获取 state；

提供 dispatch(action) 方法更新 state；

通过 subscribe(listener) 注册监听器;

通过 subscribe(listener) 返回的函数注销监听器。

1. 再次强调一下 Redux 应用只有一个单一的 store。当需要拆分数据处理逻辑时，你应该使用 reducer 组合 而不是创建多个 store
2. 根据已有的 reducer 来创建 store 是非常容易的

15. createStore() 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。

let store = createStore(todoApp, window.STATE\_FROM\_SERVER)

1. 可以看到，在还没有开发界面的时候，我们就可以定义程序的行为。而且这时候已经可以写 reducer 和 action 创建函数的测试。不需要模拟任何东西，因为它们都是纯函数。只需调用一下，对返回值做断言，写测试就是这么简单。
2. redux的一些示例：源代码

<http://www.redux.org.cn/docs/introduction/Examples.html>

17．Redux 应用中数据的生命周期遵循下面 4 个步骤：

1.调用 store.dispatch(action)。

你可以在任何地方调用 store.dispatch(action)，包括组件中、XHR 回调中、甚至定时器中。

2. Redux store 调用传入的 reducer 函数

注意 reducer 是纯函数。它仅仅用于计算下一个 state。它应该是完全可预测的：多次传入相同的输入必须产生相同的输出。它不应做有副作用的操作，如 API 调用或路由跳转。这些应该在 dispatch action 前发生。

3. 根 reducer 应该把多个子 reducer 输出合并成一个单一的 state 树。

根 reducer 的结构完全由你决定。Redux 原生提供combineReducers()辅助函数，来把根 reducer 拆分成多个函数，用于分别处理 state 树的一个分支

4. Redux store 保存了根 reducer 返回的完整 state 树

这个新的树就是应用的下一个 state！所有订阅 store.subscribe(listener) 的监听器都将被调用；监听器里可以调用 store.getState() 获得当前 state。

现在，可以应用新的 state 来更新 UI。如果你使用了 React Redux 这类的绑定库，这时就应该调用 component.setState(newState) 来更新。

5. 这里需要再强调一下：Redux 和 React 之间没有关系。Redux 支持 React、Angular、Ember、jQuery 甚至纯 JavaScript。尽管如此，Redux 还是和 React 和 Deku 这类库搭配起来用最好，因为这类库允许你以 state 函数的形式来描述界面，Redux 通过 action 的形式来发起 state 变化

搭配react

1. 并不意味着容器组件必须位于组件树的最顶层。如果一个容器组件变得太复杂（例如，它有大量的嵌套组件以及传递数不尽的回调函数），那么在组件树中引入另一个容器，就像FAQ中提到的那样
2. 技术上讲你可以直接使用 store.subscribe() 来编写容器组件。但不建议这么做的原因是无法使用 React Redux 带来的性能优化。也因此，不要手写容器组件，而使用 React Redux 的 connect() 方法来生成，后面会详细介绍。
3. 展示组件：这些组件只定义外观并不关心数据来源和如何改变。传入什么就渲染什么。如果你把代码从 Redux 迁移到别的架构，这些组件可以不做任何改动直接使用。它们并不依赖于 Redux
4. 容器组件，例子里：

VisibleTodoList 根据当前显示的状态来对 todo 列表进行过滤，并渲染 TodoList。

FilterLink 得到当前过滤器并渲染 Link。

filter: string 就是当前过滤的状态

5. 实现展示组件

它们只是普通的 React 组件，所以不会详细解释。我们会使用函数式无状态组件除非需要本地 state 或生命周期函数的场景。这并不是说展示组件必须是函数 -- 只是因为这样做容易些。如果你需要使用本地 state，生命周期方法，或者性能优化，可以将它们转成 class

1. 检查//react的内容

Todo.propTypes = {

onClick: PropTypes.func.isRequired,

completed: PropTypes.bool.isRequired,

text: PropTypes.string.isRequired}

1. 容器组件：即把这些展示组件和 Redux 关联起来
2. mapDispatchToProps() 方法接收 dispatch() 方法并返回期望注入到展示组件的 props 中的回调方法//这就是为啥mapDispatchToProps这个函数需要传的第一个参是dispatch()了
3. AddTodo 组件的视图和逻辑混合在一个单独的定义之中。
4. 传入 Store

建议的方式是使用指定的 React Redux 组件 <Provider> 来 魔法般的 让所有容器组件都可以访问 store，而不必显示地传递它。只需要在渲染根组件时使用即可。

异步action

1. Action 发出以后，过一段时间再执行 Reducer，这就是异步, 怎么才能 Reducer 在异步操作结束后自动执行呢？这就要用到新的工具：中间件（middleware）
2. 当调用异步 API 时，有两个非常关键的时刻：发起请求的时刻，和接收到响应的时刻（也可能是超时）。

这两个时刻都可能会更改应用的 state；为此，你需要 dispatch 普通的同步 action。一般情况下，每个 API 请求都需要 dispatch 至少三种 action：

一种通知 reducer 请求开始的 action。

对于这种 action，reducer 可能会切换一下 state 中的 isFetching 标记。以此来告诉 UI 来显示加载界面。

一种通知 reducer 请求成功的 action。

对于这种 action，reducer 可能会把接收到的新数据合并到 state 中，并重置 isFetching。UI 则会隐藏加载界面，并显示接收到的数据。

一种通知 reducer 请求失败的 action。

对于这种 action，reducer 可能会重置 isFetching。另外，有些 reducer 会保存这些失败信息，并在 UI 里显示出来。

为了区分这三种 action，可能在 action 里添加一个专门的 status 字段作为标记位：

2. 错误处理须知

在实际应用中，网络请求失败时也需要 dispatch action。虽然在本教程中我们并不做错误处理，但是这个 真实场景的案例 会演示一种实现方案。

3..then()这个方法什么意

.then()方法是异步执行。

意思是：就是当.then()前的方法执行完后再执行then()内部的程序，这样就避免了，数据没获取到等的问题。

语法：promise.then(onCompleted, onRejected);

参数

promise必需。Promise 对象。

onCompleted必需。承诺成功完成时要运行的履行处理程序函数。

onRejected可选。承诺被拒绝时要运行的错误处理程序函数。

4. 默认情况下，createStore() 所创建的 Redux store 没有使用 middleware，所以只支持 同步数据流。

你可以使用 applyMiddleware() 来增强 createStore()。虽然这不是必须的，但是它可以帮助你用简便的方式来描述异步的 action。

像 redux-thunk 或 redux-promise 这样支持异步的 middleware 都包装了 store 的 dispatch() 方法，以此来让你 dispatch 一些除了 action 以外的其他内容，例如：函数或者 Promise。你所使用的任何 middleware 都可以以自己的方式解析你 dispatch 的任何内容，并继续传递 actions 给下一个 middleware。比如，支持 Promise 的 middleware 能够拦截 Promise，然后为每个 Promise 异步地 dispatch 一对 begin/end actions。

当 middleware 链中的最后一个 middleware 开始 dispatch action 时，这个 action 必须是一个普通对象。这是 同步式的 Redux 数据流 开始的地方（译注：这里应该是指，你可以使用任意多异步的 middleware 去做你想做的事情，但是需要使用普通对象作为最后一个被 dispatch 的 action ，来将处理流程带回同步方式）。

15. action 拥有一个不变的 type 帮助 reducer (或 Flux 中的 Stores ) 识别它们。我们建议你使用 string 而不是 符号（Symbols） 作为 action type ，因为 string 是可序列化的，并且使用符号会使记录和重演变得困难。

减少样板代码

1. 有空看看这个

