**做React需要会什么？**

react的功能其实很单一，主要负责渲染的功能，现有的框架，比如angular是一个大而全的框架，用了angular几乎就不需要用其他工具辅助配合，但是react不一样，他只负责ui渲染，想要做好一个项目，往往需要其他库和工具的配合，比如用redux来管理数据，react-router管理路由，react已经全面拥抱es6，所以es6也得掌握，webpack就算是不会配置也要会用，要想提高性能，需要按需加载，immutable.js也得用上，还有单元测试。。。。

**React 是什么**

用脚本进行DOM操作的代价很昂贵。有个贴切的比喻，把DOM和JavaScript各自想象为一个岛屿，它们之间用收费桥梁连接，js每次访问DOM，都要途径这座桥，并交纳“过桥费”,访问DOM的次数越多，费用也就越高。 因此，推荐的做法是尽量减少过桥的次数，努力待在ECMAScript岛上。因为这个原因react的虚拟dom就显得难能可贵了，它创造了虚拟dom并且将它们储存起来，每当状态发生变化的时候就会创造新的虚拟节点和以前的进行对比，让变化的部分进行渲染。整个过程没有对dom进行获取和操作，只有一个渲染的过程，所以react说是一个ui框架。

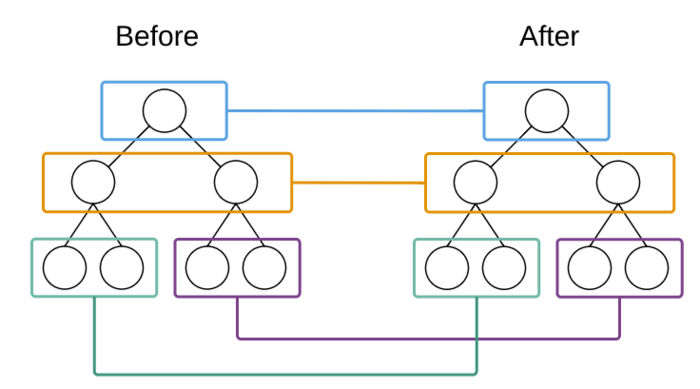
**React的组件化**

react的一个组件很明显的由dom视图和state数据组成，两个部分泾渭分明。state是数据中心，它的状态决定着视图的状态。这时候发现似乎和我们一直推崇的MVC开发模式有点区别，没了Controller控制器，那用户交互怎么处理，数据变化谁来管理？然而这并不是react所要关心的事情，它只负责ui的渲染。与其他框架监听数据动态改变dom不同，react采用setState来控制视图的更新。setState会自动调用render函数，触发视图的重新渲染，如果仅仅只是state数据的变化而没有调用setState，并不会触发更新。 组件就是拥有独立功能的视图模块，许多小的组件组成一个大的组件，整个页面就是由一个个组件组合而成。它的好处是利于重复利用和维护。

**React的 Diff算法**

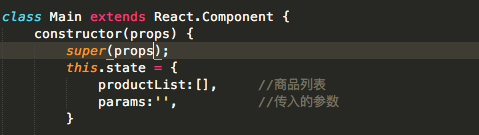
react的diff算法用在什么地方呢？当组件更新的时候，react会创建一个新的虚拟dom树并且会和之前储存的dom树进行比较，这个比较多过程就用到了diff算法，所以组件初始化的时候是用不到的。react提出了一种假设，相同的节点具有类似的结构，而不同的节点具有不同的结构。在这种假设之上进行逐层的比较，如果发现对应的节点是不同的，那就直接删除旧的节点以及它所包含的所有子节点然后替换成新的节点。如果是相同的节点，则只进行属性的更改。

对于列表的diff算法稍有不同，因为列表通常具有相同的结构，在对列表节点进行删除，插入，排序的时候，单个节点的整体操作远比一个个对比一个个替换要好得多，所以在创建列表的时候需要设置key值，这样react才能分清谁是谁。当然不写key值也可以，但这样通常会报出警告，通知我们加上key值以提高react的性能。



**React组件是怎么来的**

组件的创造方法为React.createClass() ——创造一个类，react系统内部设计了一套类系统，利用它来创造react组件。但这并不是必须的，我们还可以用es6的class类来创造组件,这也是Facebook官方推荐的写法。

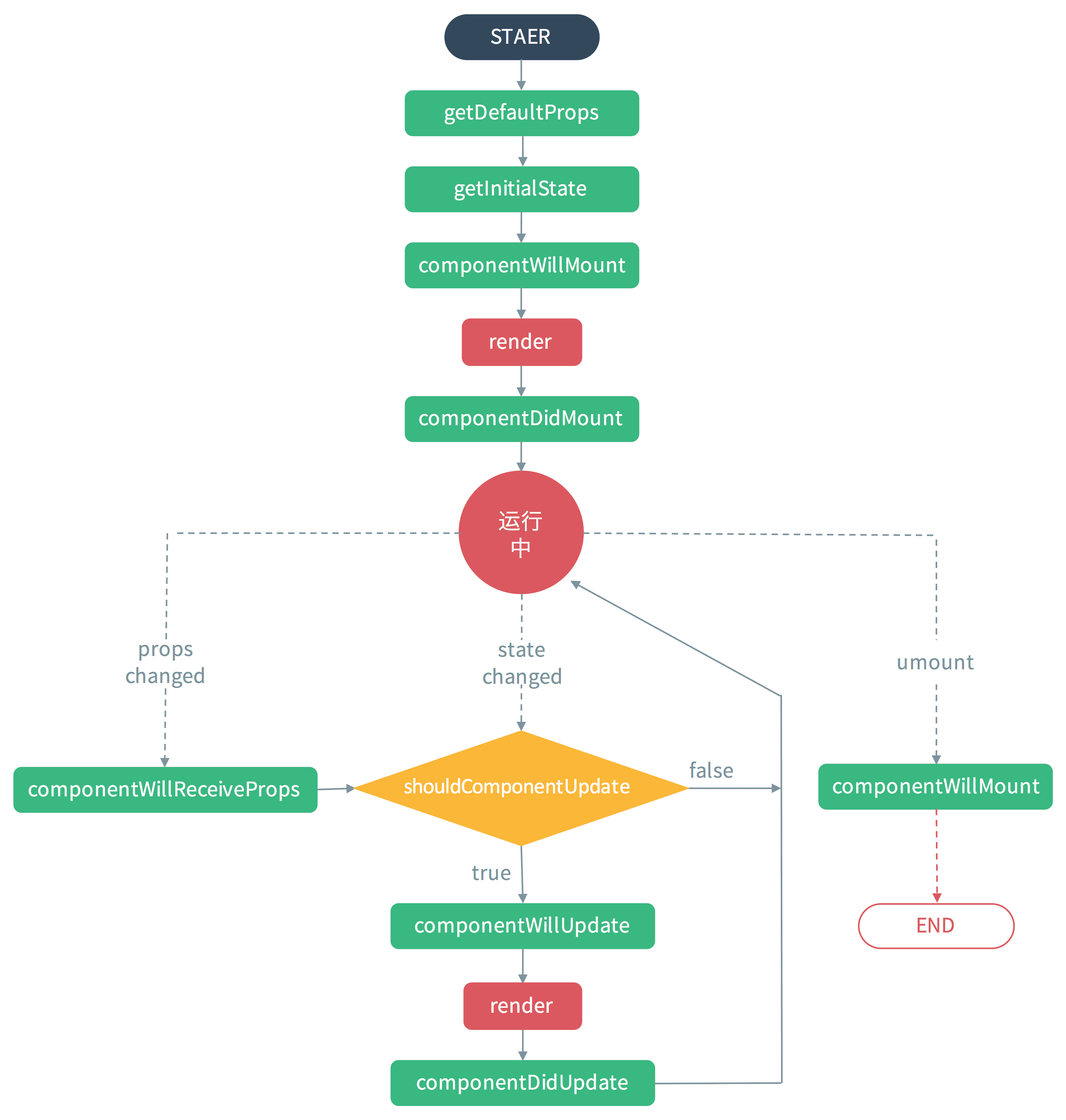


这两种写法实现的功能一样但是原理却是不同，es6的class类可以看作是构造函数的一个语法糖，可以把它当成构造函数来看，extends实现了类之间的继承 —— 定义一个类Main 继承React.Component所有的属性和方法，组件的生命周期函数就是从这来的。constructor是构造器，在实例化对象时调用，super调用了父类的constructor创造了父类的实例对象this，然后用子类的构造函数进行修改。这和es5的原型继承是不同的，原型继承是先创造一个实例化对象this，然后再继承父级的原型方法。了解了这些之后我们在看组件的时候就清楚很多。

当我们使用组件< Main />时，其实是对Main类的实例化——new Main，只不过react对这个过程进行了封装，让它看起来更像是一个标签。

有三点值得注意：1、定义类名字的首字母必须大写 2、因为class变成了关键字，类选择器需要用className来代替。 3、类和模块内部默认使用严格模式，所以不需要用use strict指定运行模式。

**组件的生命周期**



**组件在初始化时会触发5个钩子函数：**

**1、getDefaultProps()**

设置默认的props，也可以用dufaultProps设置组件的默认属性。

**2、getInitialState()**

在使用es6的class语法时是没有这个钩子函数的，可以直接在constructor中定义this.state。此时可以访问this.props。

**3、componentWillMount()**

组件初始化时只调用，以后组件更新不调用，整个生命周期只调用一次，此时可以修改state。

**4、 render()**

react最重要的步骤，创建虚拟dom，进行diff算法，更新dom树都在此进行。此时就不能更改state了。

**5、componentDidMount()**

组件渲染之后调用，可以通过this.getDOMNode()获取和操作dom节点，只调用一次。

**在更新时也会触发5个钩子函数：**

**6、componentWillReceivePorps(nextProps)**

组件初始化时不调用，组件接受新的props时调用。

**7、shouldComponentUpdate(nextProps, nextState)**

react性能优化非常重要的一环。组件接受新的state或者props时调用，我们可以设置在此对比前后两个props和state是否相同，如果相同则返回false阻止更新，因为相同的属性状态一定会生成相同的dom树，这样就不需要创造新的dom树和旧的dom树进行diff算法对比，节省大量性能，尤其是在dom结构复杂的时候。不过调用this.forceUpdate会跳过此步骤。

**8、componentWillUpdata(nextProps, nextState)**

组件初始化时不调用，只有在组件将要更新时才调用，此时可以修改state

**9、render()**

不多说

**10、componentDidUpdate()**

组件初始化时不调用，组件更新完成后调用，此时可以获取dom节点。

还有一个卸载钩子函数

**11、componentWillUnmount()**

组件将要卸载时调用，一些事件监听和定时器需要在此时清除。

以上可以看出来react总共有10个周期函数（render重复一次），这个10个函数可以满足我们所有对组件操作的需求，利用的好可以提高开发效率和组件性能。

**React-Router路由**

Router就是React的一个组件，它并不会被渲染，只是一个创建内部路由规则的配置对象，根据匹配的路由地址展现相应的组件。Route则对路由地址和组件进行绑定，Route具有嵌套功能，表示路由地址的包涵关系，这和组件之间的嵌套并没有直接联系。Route可以向绑定的组件传递7个属性：children，history，location，params，route，routeParams，routes，每个属性都包涵路由的相关的信息。比较常用的有children（以路由的包涵关系为区分的组件），location（包括地址，参数，地址切换方式，key值，hash值）。react-router提供Link标签，这只是对a标签的封装，值得注意的是，点击链接进行的跳转并不是默认的方式，react-router阻止了a标签的默认行为并用pushState进行hash值的转变。切换页面的过程是在点击Link标签或者后退前进按钮时，会先发生url地址的转变，Router监听到地址的改变根据Route的path属性匹配到对应的组件，将state值改成对应的组件并调用setState触发render函数重新渲染dom。

当页面比较多时，项目就会变得越来越大，尤其对于单页面应用来说，初次渲染的速度就会很慢，这时候就需要按需加载，只有切换到页面的时候才去加载对应的js文件。react配合webpack进行按需加载的方法很简单，Route的component改为getComponent，组件用require.ensure的方式获取，并在webpack中配置chunkFilename。

**组件之间的通信**

react推崇的是单向数据流，自上而下进行数据的传递，但是由下而上或者不在一条数据流上的组件之间的通信就会变的复杂。解决通信问题的方法很多，如果只是父子级关系，父级可以将一个回调函数当作属性传递给子级，子级可以直接调用函数从而和父级通信。

组件层级嵌套到比较深，可以使用上下文getChildContext来传递信息，这样在不需要将函数一层层往下传，任何一层的子级都可以通过this.context直接访问。

兄弟关系的组件之间无法直接通信，它们只能利用同一层的上级作为中转站。而如果兄弟组件都是最高层的组件，为了能够让它们进行通信，必须在它们外层再套一层组件，这个外层的组件起着保存数据，传递信息的作用，这其实就是redux所做的事情。

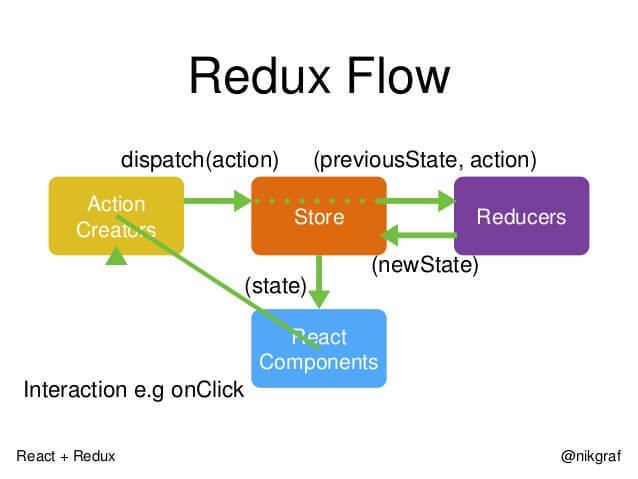
组件之间的信息还可以通过全局事件来传递。不同页面可以通过参数传递数据，下个页面可以用location.param来获取。其实react本身很简单，难的在于如何优雅高效的实现组件之间数据的交流。

**Redux**

首先，redux并不是必须的，它的作用相当于在顶层组件之上又加了一个组件，作用是进行逻辑运算、储存数据和实现组件尤其是顶层组件的通信。如果组件之间的交流不多，逻辑不复杂，只是单纯的进行视图的渲染，这时候用回调，context就行，没必要用redux，用了反而影响开发速度。但是如果组件交流特别频繁，逻辑很复杂，那redux的优势就特别明显了。我第一次做react项目的时候并没有用redux，所有的逻辑都是在组件内部实现，当时为了实现一个逻辑比较复杂的购物车，洋洋洒洒居然写了800多行代码，回头一看我自己都不知道写的是啥，画面太感人。

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。

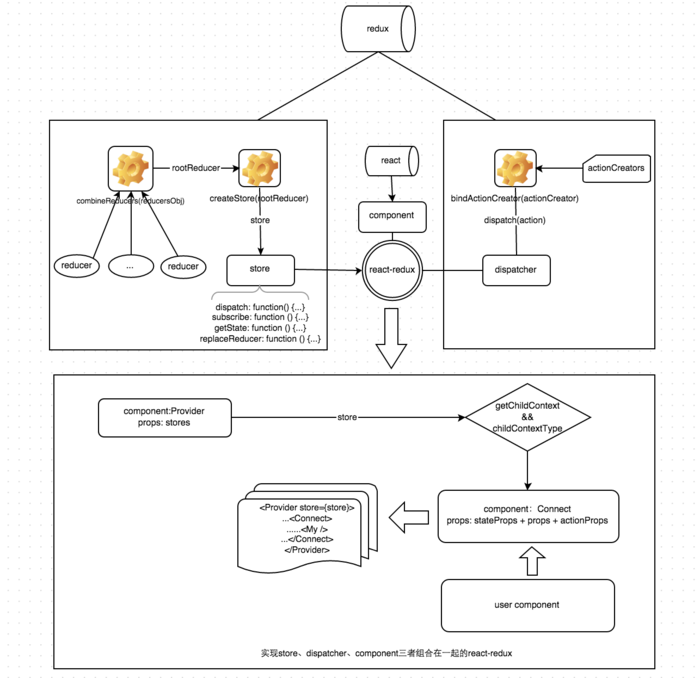
流程是这个样子的：



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。

**接下来具体分析一下，redux以及react-redux到底是怎么实现的。**

**先上一张图**



明显比第一张要复杂，其实两张图说的是同一件事。从上而下慢慢分析：

先说说redux：

**redux主要由三部分组成：store，reducer，action。**

**\*\*store\*\*是一个对象，它有四个主要的方法：**

**\*\*1、dispatch:\*\***

> 用于action的分发——在createStore中可以用middleware中间件对dispatch进行改造，比如当action传入dispatch会立即触发reducer，有些时候我们不希望它立即触发，而是等待异步操作完成之后再触发，这时候用redux-thunk对dispatch进行改造，以前只能传入一个对象，改造完成后可以传入一个函数，在这个函数里我们手动dispatch一个action对象，这个过程是可控的，就实现了异步。

**\*\*2、subscribe：\*\***

> 监听state的变化——这个函数在store调用dispatch时会注册一个listener监听state变化，当我们需要知道state是否变化时可以调用，它返回一个函数，调用这个返回的函数可以注销监听。

let unsubscribe = store.subscribe(() => {console.log('state发生了变化')})

**\*\*3、getState：\*\***

> 获取store中的state——当我们用action触发reducer改变了state时，需要再拿到新的state里的数据，毕竟数据才是我们想要的。getState主要在两个地方需要用到，一是在dispatch拿到action后store需要用它来获取state里的数据，并把这个数据传给reducer，这个过程是自动执行的，二是在我们利用subscribe监听到state发生变化后调用它来获取新的state数据，如果做到这一步，说明我们已经成功了。

**\*\*4、replaceReducer:\*\***

> 替换reducer，改变state修改的逻辑。

store可以通过createStore()方法创建，接受三个参数，经过combineReducers合并的reducer和state的初始状态以及改变dispatch的中间件，后两个参数并不是必须的。store的主要作用是将action和reducer联系起来并改变state。

**\*\*action:\*\***

>action是一个对象，其中type属性是必须的，同时可以传入一些数据。action可以用actionCreactor进行创造。dispatch就是把action对象发送出去。

**\*\*reducer:\*\***

>reducer是一个函数，它接受一个state和一个action，根据action的type返回一个新的state。根据业务逻辑可以分为很多个reducer，然后通过combineReducers将它们合并，state树中有很多对象，每个state对象对应一个reducer，state对象的名字可以在合并时定义。

像这个样子：

const reducer = combineReducers({

a: doSomethingWithA,

b: processB,

c: c

})

**\*\*combineReducers:\*\***

>其实它也是一个reducer，它接受整个state和一个action，然后将整个state拆分发送给对应的reducer进行处理，所有的reducer会收到相同的action，不过它们会根据action的type进行判断，有这个type就进行处理然后返回新的state，没有就返回默认值，然后这些分散的state又会整合在一起返回一个新的state树。

接下来分析一下整体的流程，首先调用store.dispatch将action作为参数传入，同时用getState获取当前的状态树state并注册subscribe的listener监听state变化，再调用combineReducers并将获取的state和action传入。combineReducers会将传入的state和action传给所有reducer，reducer会根据state的key值获取与自己对应的state，并根据action的type返回新的state，触发state树的更新，我们调用subscribe监听到state发生变化后用getState获取新的state数据。

redux的state和react的state两者完全没有关系，除了名字一样。

**\*\*上面分析了redux的主要功能，那么react-redux到底做了什么？\*\***

**React-Redux**

如果只使用redux，那么流程是这样的：

> component --> dispatch(action) --> reducer --> subscribe --> getState --> component

用了react-redux之后流程是这样的：

> component --> actionCreator(data) --> reducer --> component

store的三大功能：dispatch，subscribe，getState都不需要手动来写了。react-redux帮我们做了这些，同时它提供了两个好基友Provider和connect。

**\*\*Provider\*\***是一个组件，它接受store作为props，然后通过context往下传，这样react中任何组件都可以通过contex获取store。也就意味着我们可以在任何一个组件里利用dispatch(action)来触发reducer改变state，并用subscribe监听state的变化，然后用getState获取变化后的值。但是并不推荐这样做，它会让数据流变的混乱，过度的耦合也会影响组件的复用，维护起来也更麻烦。

**\*\*connect --connect(mapStateToProps, mapDispatchToProps, mergeProps, options)\*\***是一个函数，它接受四个参数并且再返回一个函数--wrapWithConnect，wrapWithConnect接受一个组件作为参数wrapWithConnect(component)，它内部定义一个新组件Connect(容器组件)并将传入的组件(ui组件)作为Connect的子组件然后return出去。

所以它的完整写法是这样的：connect(mapStateToProps, mapDispatchToProps, mergeProps, options)(component)

**\*\*mapStateToProps(state, [ownProps])：\*\***

>mapStateToProps 接受两个参数，store的state和自定义的props，并返回一个新的对象，这个对象会作为props的一部分传入ui组件。我们可以根据组件所需要的数据自定义返回一个对象。ownProps的变化也会触发mapStateToProps

function mapStateToProps(state) {

return { todos: state.todos };

}

**\*\*mapDispatchToProps(dispatch, [ownProps])：\*\***

> mapDispatchToProps如果是对象，那么会和store绑定作为props的一部分传入ui组件。如果是个函数，它接受两个参数，bindActionCreators会将action和dispatch绑定并返回一个对象，这个对象会和ownProps一起作为props的一部分传入ui组件。所以不论mapDispatchToProps是对象还是函数，它最终都会返回一个对象，如果是函数，这个对象的key值是可以自定义的

unction mapDispatchToProps(dispatch) {

return {

todoActions: bindActionCreators(todoActionCreators, dispatch),

counterActions: bindActionCreators(counterActionCreators, dispatch)

};

}

mapDispatchToProps返回的对象其属性其实就是一个个actionCreator，因为已经和dispatch绑定，所以当调用actionCreator时会立即发送action，而不用手动dispatch。ownProps的变化也会触发mapDispatchToProps。

**\*\*mergeProps(stateProps, dispatchProps, ownProps)：\*\***

> 将mapStateToProps() 与 mapDispatchToProps()返回的对象和组件自身的props合并成新的props并传入组件。默认返回 Object.assign({}, ownProps, stateProps, dispatchProps) 的结果。

**\*\*options：\*\***

> pure = true 表示Connect容器组件将在shouldComponentUpdate中对store的state和ownProps进行浅对比，判断是否发生变化，优化性能。为false则不对比。

其实connect函数并没有做什么，大部分的逻辑都是在它返回的wrapWithConnect函数内实现的，确切的说是在wrapWithConnect内定义的Connect组件里实现的。

下面是一个完整的 react --> redux --> react 流程：

一、Provider组件接受redux的store作为props，然后通过context往下传。

二、connect函数在初始化的时候会将mapDispatchToProps对象绑定到store，如果mapDispatchToProps是函数则在Connect组件获得store后，根据传入的store.dispatch和action通过bindActionCreators进行绑定，再将返回的对象绑定到store，connect函数会返回一个wrapWithConnect函数，同时wrapWithConnect会被调用且传入一个ui组件，wrapWithConnect内部使用class Connect extends Component定义了一个Connect组件，传入的ui组件就是Connect的子组件，然后Connect组件会通过context获得store，并通过store.getState获得完整的state对象，将state传入mapStateToProps返回stateProps对象、mapDispatchToProps对象或mapDispatchToProps函数会返回一个dispatchProps对象，stateProps、dispatchProps以及Connect组件的props三者通过Object.assign()，或者mergeProps合并为props传入ui组件。然后在ComponentDidMount中调用store.subscribe，注册了一个回调函数handleChange监听state的变化。

三、此时ui组件就可以在props中找到actionCreator，当我们调用actionCreator时会自动调用dispatch，在dispatch中会调用getState获取整个state，同时注册一个listener监听state的变化，store将获得的state和action传给combineReducers，combineReducers会将state依据state的key值分别传给子reducer，并将action传给全部子reducer，reducer会被依次执行进行action.type的判断，如果有则返回一个新的state，如果没有则返回默认。combineReducers再次将子reducer返回的单个state进行合并成一个新的完整的state。此时state发生了变化。Connect组件中调用的subscribe会监听到state发生了变化，然后调用handleChange函数，handleChange函数内部首先调用getState获取新的state值并对新旧两个state进行浅对比，如果相同直接return，如果不同则调用mapStateToProps获取stateProps并将新旧两个stateProps进行浅对比，如果相同，直接return结束，不进行后续操作。如果不相同则调用this.setState()触发Connect组件的更新，传入ui组件，触发ui组件的更新，此时ui组件获得新的props，react --> redux --> react 的一次流程结束。

**\*\*上面的有点复杂，简化版的流程是：\*\***

一、Provider组件接受redux的store作为props，然后通过context往下传。

二、connect函数收到Provider传出的store，然后接受三个参数mapStateToProps，mapDispatchToProps和组件，并将state和actionCreator以props传入组件，这时组件就可以调用actionCreator函数来触发reducer函数返回新的state，connect监听到state变化调用setState更新组件并将新的state传入组件。

connect可以写的非常简洁，mapStateToProps，mapDispatchToProps只不过是传入的回调函数，connect函数在必要的时候会调用它们，名字不是固定的，甚至可以不写名字。

简化版本：

connect(state => state, action)(Component);

**项目搭建**

上面说了react，react-router和redux的知识点。但是怎么样将它们整合起来，搭建一个完整的项目。

1、先引用 react.js，redux，react-router 等基本文件，建议用npm安装，直接在文件中引用。

2、从 react.js，redux，react-router 中引入所需要的对象和方法。

import React, {Component, PropTypes} from 'react';

import ReactDOM, {render} from 'react-dom';

import {Provider, connect} from 'react-redux';

import {createStore, combineReducers, applyMiddleware} from 'redux';

import { Router, Route, Redirect, IndexRoute, browserHistory, hashHistory } from 'react-router';

3、根据需求创建顶层ui组件，每个顶层ui组件对应一个页面。

4、创建actionCreators和reducers，并用combineReducers将所有的reducer合并成一个大的reduer。利用createStore创建store并引入combineReducers和applyMiddleware。

5、利用connect将actionCreator，reuder和顶层的ui组件进行关联并返回一个新的组件。

6、利用connect返回的新的组件配合react-router进行路由的部署，返回一个路由组件Router。

7、将Router放入最顶层组件Provider，引入store作为Provider的属性。

8、调用render渲染Provider组件且放入页面的标签中。

可以看到顶层的ui组件其实被套了四层组件，Provider，Router，Route，Connect，这四个组件并不会在视图上改变react，它们只是功能性的。

通常我们在顶层的ui组件打印props时可以看到一堆属性：



上图的顶层ui组件属性总共有18个，如果刚刚接触react，可能对这些属性怎么来的感到困惑，其实这些属性来自五个地方：

组件自定义属性1个，actionCreator返回的对象6个，reducer返回的state4个，Connect组件属性0个，以及Router注入的属性7个。

项目源码地址：[https://github.com/bailicangdu/pxq](https://link.jianshu.com?t=https:/github.com/bailicangdu/pxq)