1.

Redux是提供给Javascript App的一个数据存储地方，和React没有什么本质关系，Redux可以结合其他库正常使用。只不过Redux数据管理方式，跟React的数据驱动视图理念很合拍，它俩结合在一起，开发非常便利。并且Redux小组专门为React提供了Redux-React包。Redux-React包用来连接react组件和store仓库的桥梁。

2.

Redux是流行框架React不折不扣的最佳伴侣。React管理显示，贤内助Redux管理数据。

3.

在React应用中使用Redux并不是必须的，如果React应用中状态很少，或者状态变化很少， 将所有状态都存放于顶层组件便可以管理的时候，那么并不需要Redux， 我们只需要采用React管理状态的办法， 例如从顶层组件的state作为子组件的props传递给子组件， 子组件使用父组件的回调函数来改变状态(lift state up 概念)。

在大型React APP中，组件的数量以及状态变化很多，采用原生的React方式一层层的传递，采用回调函数让子组件改变上层组件的状态的方式过于复杂， 所以我们引入Redux. Redux将APP中所有的状态都放入一个Redux store中，所有的组件都可以通过Redux来直接访问改变状态，无须多层级调用。

4.

Redux可实现状态的Hot-Reloading. 虽然Webpack也实现Hot-Reloading，即是在代码改变的时候，webpack监听源码文件的reload event,将改变的代码自动刷新到页面上。但是局限性是webpack也将刷新组件的状态，回到最初的state状态，如果我们想调试组件在某个状态的表现，那么reload以后还要将组件一步步的变回该状态。

使用Redux以后，由于所有的状态都存在Redux的store中，我们可以实现在代码改变以后让组件直接回到Redux中store所代表的状态时间点。

**几个主要概念**

State

State直译为状态。它在Redux里就是数据模型，或者说是当前的某一部分数据。

这些数据是用于应用的显示，所以可以说这些数据就代表了应用的状态。

比如最简单的Todo list，每一个todo包含了todo内容、状态（完成、未完成）。这里每个todo可以称为todo state，而这个包含todo顺序的列表又可以称为todo list state。

Store

Store就是整个应用的数据仓库，它是State的集合，也是连接Action、Reducer、State这几个主要构成的桥梁。

Store实例通过redux中的createStore(reduer实例) 方法创建。

Action

Action是store.dispatch 方法传递的指令和相关参数、数据。

也就是说store.dispatch 会将Action指令发送给所有挂载在Store上的Reducer，让Reducer根据Action的类型和 “Action”提供的相关数据处理更新现有的State。

Action实例中将会说明这个Action对数据的操作类型action.type，以及包含更新的数据值等。

Redux作者给出的关于Action的实例通常如下：

store.dispatch({

type: TODO.COMPLETE\_TODO

data: 'xxx'

});

Reducer

Reducer负责接收store.dispatch(action)发送过来的action。Action是一个对象，里面含有具体的指令和相关数据，reducer接受后对现有State进行处理，并返回处理过后的State，更新我们的Store。

不论dispatch了什么action，所有挂载在Store上的Reducer都能依次接收到，并且可以对State进行处理，全看你代码怎么写。自由度很大。

Reducer中将会写明，当收到一个action以后，将会根据action实例中所包含的数据操作类型，进行不同的操作。如果action是数据读取类型，则返回数据。如果action是数据更新类型，则使用action中所带的数据对数据进行更新，然后返回最新数据。(需要注意的是如果处理的数据的引用类型数据，不要在Reducer中直接更改，而是应该使用immutable方法直接返回一个新的对象)

Reducer方法将接受两个参数。一个是datastore中此时的状态，一个是action对象。需要注意的是如果是更新，新建的操作，如果需要保留原有的状态的部分，使用...来原状态进行拷贝。另外当dataStore新建的时候也会调用一次reducer方法，此时action为空，但是reducer要求务必返回一个状态对象，不能为null.所以reducer中应该包含default条件，为初始时候状态。

在小型程序中可能只有一个根reducer,在大型的react的程序多，将会有多个reducer存在。

Reducer示例代码：

function counter (state = 0, action) {

switch (action.type) { // 根据action的类别

case 'INCREMENT':

return state + 1

case 'DECREMENT':

return state - 1

default:

return state // 最终返回一个新的状态

}

}

Reducer所接收的state可以是任何类型，可以是原始数据类型，例如数字，字符字符串等。也可以是数据对象等。**Reducer在接收action改变状态的时候，不在Reducer中直接改变而是最后返回一个新的状态。**

Selector

Selector是用来从Store中选出显示或者做其他处理通常所需要的一些数据，并做一定的整理。

Redux其实就是包装了一个"客户端数据库"出来。

Store => 数据库

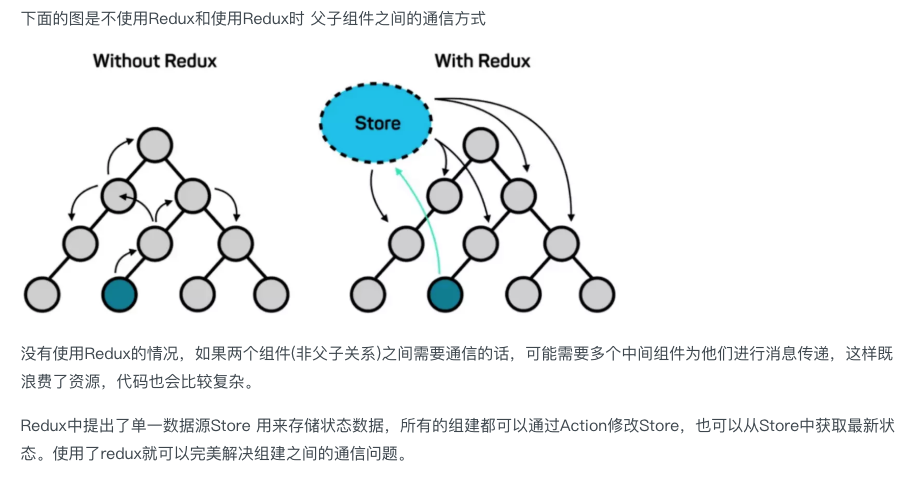
State => 数据表

Action => 查询指令

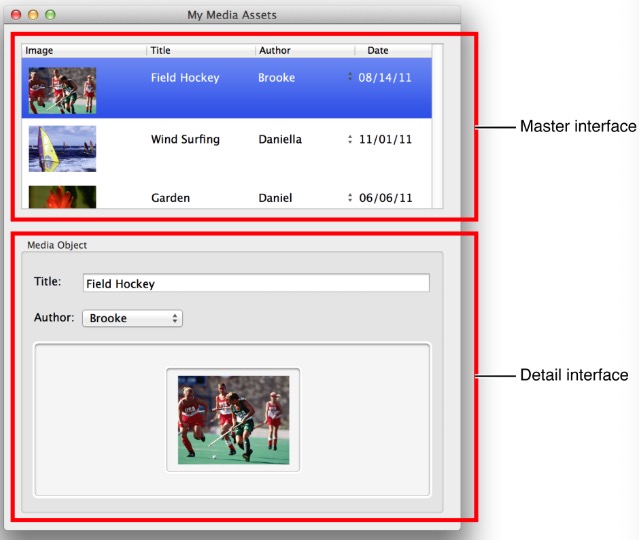
Reducer => 存储过程、数据锁等

Selector => 数据库视图

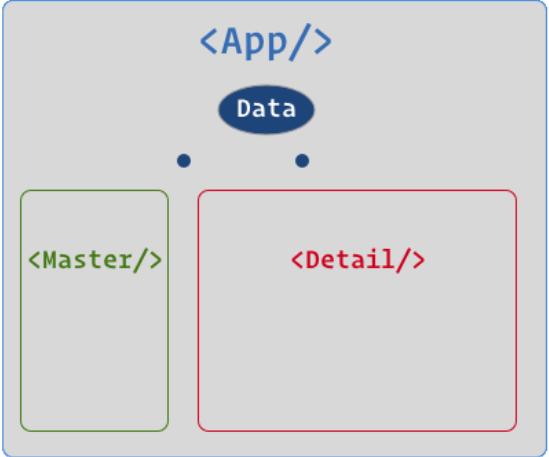
5.Redux实现组件的流畅通信：



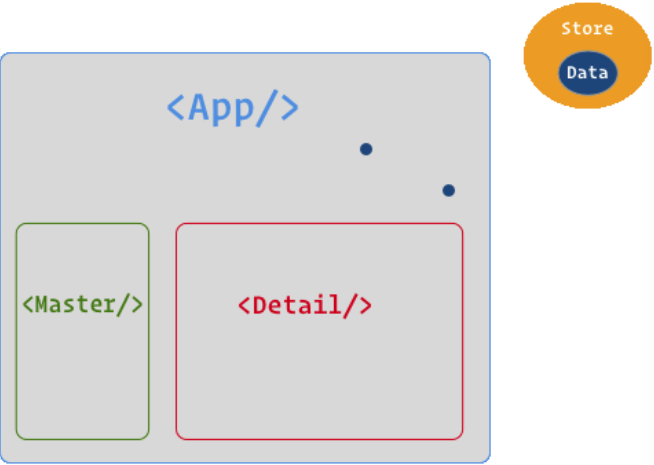
例子（开发一个master/detail的react app）：



在这样一个应用当中，Master Component显示概略信息而Detail Component显示对应的详细信息。我们需要这个这两个控件在状态上是统一的，如果按照纯React的做法，我们会按照React的life state up将状态放到他们的最近的共同祖先父组件当中。然后父组件作为子组件的props分别传给这两个控件。如下图：



如果Master控件和Detail控件在上一层便可以有共同的祖先组件，那么我们将状态放入该祖先组件当中。但是如果他们的共同祖先需要在上方几层以后才能找到，将状态放入该祖先组件当中便会产生很多不便。首先我们需要将状态通过该祖先组件，作为子组件的props传递给沿途的其他组件，最终传到Master以及Detail组件。这使得状态的追踪十分不便以及不直接。另外如果状态发生更新，那么相当于沿途的所有组件都需要更新而不仅仅是Master以及Detail组件。最后对于沿途的组件，这些props对于他们来说是没有意义的只是单纯的传递给他们的子组件。如果我们使用Redux以后：



Master和Detail组件可以直接从Redux的store中获取更改状态，直接方便了许多。对于沿途的控件也不需要获取额外的props, 也不会有无意义的更新。

5.

Redux安装命令: npm install redux, 一般还会安装其开发帮助包npm install redux-devtools

如果使用React-Redux则需要先安装redux(npm install redux), 再安装react-redux(npm install react-redux)

6.

**采用Redux机制，index.js示例代码, 用于创建redux store，产生组件以及和组件绑定：**

import React from 'react';

import ReactDOM from 'react-dom';

import Counter from './components/Counter'; //引入自己编写的React组件计数器

import {createStore} from 'redux'; //从redux包中解构引入得到createStore, 为store的创建做准备

import reducer from './reducers/index' //引入自行编写的reducer纯函数 该函数,根据action的type不同结合旧的state返回新的state

let store = createStore(reducer); //创建redux的核心store, 从而Redux的store中获取state的初始值以及注册商所有处理action的reducer

import {numAdd, numDel} from './actions/index'; //引入自行编写的action函数。它将返回一个action对象，将包括actiontype,以及其他数据。

// Render the main component into the DOM

//这里使用initializer变量来定义ReactDOM中的render函数, 是方便store中state更新之后, 页面的渲染

const initializer= () => {

ReactDOM.render(

<Counter

value={store.getState()} // value={store.getState()} ,从redux中取给组件Counter的数据 这里的store.getState()得到的值,就是store创建过程中reducer纯函数里面的初始state值

add = {() =>store.dispatch(numAdd())}

del = {() =>store.dispatch(numDel())}

// ()=>store.dispatch(numAdd())和()=>store.dispatch(numDel()) 定义函数传递给展示组件Counter

//store.dispatch(参数)会传递一个action对象作为参数例如{type:"add"},调用reducer纯函数, reducer根据传入的action，执行相对应的逻辑实现store中的state的更新

></Counter>,

document.getElementById('app')

);

};

initializer ();//该函数执行,就初始化了页面

store.subscribe(initializer);

//store.subscribe()用来监听store中的state是否发生改变,如果发生改变,就重新渲染页面,所以才跟initializer ()进行绑定

**Redux机制的Reducer示例代码：**

const reducer = (state=0, action) => { //定义初始state为0，将接收action以及返回新的状态   
 switch (action.type){  
 case 'add':  
 return state+1;  
 case 'del':  
 return state-1;  
 default:  
 return state  
 }  
};  
export default reducer;

**Redux机制的Action示例代码：**

export const numAdd = () => { //action返回一个必须包含type属性的对象作为reducer的参数

return {

type:"add"

}

};

export const numDel = () => {

return {

type:"del"

}

}

**Redux机制中组件本身示例代码：**

class Counter extends React.Component {

render () {

return (

<div>

<p style= {{color: 'red'}}>

点击次数{this.props.value}

</p>

<button onClick={this.props.add}>加一</button>

<button onClick={this.props.del}>减一</button>

</div>

)

}

}

6.

Redux三个重要概念：

1. 关于App的状态都存在于一个对象树中，而此对象树处于一个store中。通过每个时刻对象树的状态，我们可以确定App的整体状态，同时回到某个时刻对象树的状态，也就等于让App回到某个状态当中（redo操作原理）。
2. 状态是只可读的。所有修改状态的操作都需要经过action来进行触发。这些action将会严格的按照一定次序被接受执行，不会存在race condition的问题。
3. 由纯函数reducer来接受action以及现有的状态,返回新的状态.我们根据App的规模，可以有多个reducer也可能只有一个根reducer.

7.

使用Redux的项目结构，我们可以在/src文件夹下面创建一个redux专门用于放redux相关资料。在redux下，创建action, reducer各自放置action和reducer相关文件.然后在项目的index.js中进行createStore, 以及将组件和Store进行绑定等。

8.