**React的生命周期**

1. **constructor()——构造函数，组件最先调用，并且值执行一次。**

**作用：用来存放初始状态值，或者函数**

**2. componentWillMount() 组件初始渲染（render()被调用前）前调用，并且仅调用一次**

**作用：如果在这个函数中调用setState改变某些状态机，react会等待setState完成后再渲染组件**

**注意：子组件也有componentWillMount函数，在父组件的该函数调用后再被调用。**

1. **render() componentWillMount之后，componentDidMount之前。**

**作用：渲染挂载组件**

**注意：组件所必不可少的核心函数；不能在该函数中修改状态机state**

1. **componentDidMount() render之后被调用，并且仅调用一次**

**作用：渲染挂载组件；可以使用refs（备注：React支持一个特殊的属性，你可以将这个属性加在任何通过render()返回的组件中。这也就是说对render()返回的组件进行一个标记，可以方便的定位的这个组件实例。**

**注意：子组件也有该函数，在父组件的该函数调用前被调用；如果在该函数中修改某些状态机state，会重新渲染render组件，所以有些组件为减少渲染次数，可以将某些修改状态机的操作放在componentWillMount函数中；如果需要再程序启动显示初始化页面后从网路获取数据，可以将网络请求代码放在该函数中**

1. **componentWillReceiveProps(nextProps)**

**执行时间：组件渲染后，当组件接收到新的props时被调用；这个函数接收一个object参数（新的props）；props是父组件传递给子组件的。父组件发生render的时候子组件就会调用**

**作用：渲染挂载组件；可以使用refs（备注：React支持一个特殊的属性，你可以将这个属性加在任何通过render()返回的组件中。这也就是说对render()返回的组件进行一个标记，可以方便的定位的这个组件实例。）**

**注意：react初次渲染时，该函数并不会被触发，因此有时该函数需要和componentWillMount或componentDidMount组合使用;使用该函数一定要加nextProps参数，首次使用了解的可以先打印结果（console.log(nextProps)）**

1. **shouldComponentUpdate(nextProps, nextState)**

**执行时间：组件挂载后（即执行完render），接收到新的state或props时被调用，即每次执行setstate都会执行该函数，来判断是否重新render组件，默认返回true；接收两个参数：第一个是心的props，第二个是新的state。**

**作用：如果有些变化不需要重新render组件，可以在该函数中阻拦**

**注意：该方法在初始化渲染的时候不会调用，在使用 forceUpdate 方法的时候也不会**

1. **componentWillUpdate()**

**执行时间：在接收到新的props 或者 state，重新渲染 之前立刻调用，在初始化渲染的时候该方法不会被调用**

**作用：为即将发生的重新渲染做一些准备工作**

**注意：不能再该函数中通过this.setstate再次改变状态机，如果需要，则在componentWillReceiveProps函数中改变**

**8.componentDidUpdate()**

**执行时间：重新渲染后调用，在初始化渲染的时候该方法不会被调用**

**作用：使用该方法可以在组件更新之后操作DOM 元素**

**9.componentWillUnmount()**

**执行时间：组件被卸载前调用，**

**作用：在该方法中执行任何必要的清理，比如无效的定时器，或者清除在 componentDidMount 中创建的 DOM 元素。**

**为什么虚拟dom会提高性能?**

虚拟dom相当于在js和真实dom中间加了一个缓存，利用dom diff算法避免了没有必要的dom操作，从而提高性能。

具体实现步骤如下：

用 Java 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中

当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异

把2所记录的差异应用到步骤1所构建的真正的DOM树上，视图就更新了。

diff算法?

把树形结构按照层级分解，只比较同级元素。

给列表结构的每个单元添加唯一的key属性，方便比较。

React 只会匹配相同 class 的 component（这里面的class指的是组件的名字）

合并操作，调用 component 的 setState 方法的时候, React 将其标记为 dirty.到每一个事件循环结束, React 检查所有标记 dirty 的 component 重新绘制.

选择性子树渲染。开发人员可以重写shouldComponentUpdate提高diff的性能。

**简述flux 思想**

Flux 的最大特点，就是数据的"单向流动"。

1.用户访问 View

2.View 发出用户的 Action

3.Dispatcher 收到 Action，要求 Store 进行相应的更新

4.Store 更新后，发出一个"change"事件

5.View 收到"change"事件后，更新页面

**调用 setState 之后发生了什么？**

在代码中调用setState函数之后，React 会将传入的参数对象与组件当前的状态合并，根据新的状态计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。这就保证了按需更新，而不是全部重新渲染。

**在生命周期中的哪一步你应该发起 AJAX 请求？**

我们应当将AJAX 请求放到 componentDidMount 函数中执行，主要原因有下：

React 可能会多次频繁调用 componentWillMount。如果我们将 AJAX 请求放到 componentWillMount 函数中，那么显而易见其会被触发多次，自然也就不是好的选择。

如果我们将 AJAX 请求放置在生命周期的其他函数中，我们并不能保证请求仅在组件挂载完毕后才会要求响应。如果我们的数据请求在组件挂载之前就完成，并且调用了setState函数将数据添加到组件状态中，对于未挂载的组件则会报错。而在 componentDidMount 函数中进行 AJAX 请求则能有效避免这个问题。

**shouldComponentUpdate 的作用是啥以及为何它这么重要？**

shouldComponentUpdate 允许我们手动地判断是否要进行组件更新，根据组件的应用场景设置函数的合理返回值能够帮我们避免不必要的更新。

React-redux

## redux

redux就是用来统一管理项目中的状态（state）。state它可以是前后端的各种数据，也可以是UI上的一些信息。简单点，它就是个对象，包含了项目中可能用于改变的一些信息。

redux**重要关注**的几点：**Actions**，**Reducers**，**Store**，下面就分别来看看吧。

#### 1、Actions:

function changeTable(index) {

return { type: "channgTable", data:index }

}

以上例子就是一个action，我们不用纠结它定义个一个函数形式还是其它，最终它就是一个对象。包含type、data或者还有其他元素的对象。

action就是用来告诉我们的状态管理器需要做什么样的一种操作。拿以上例子来说，就是为了做一个切换table的操作，那么我就定义了这么一个action。data就是你做这个操作需要处理的一些数据等。

#### 2、Reducers :

const reducer = function(state={"tableIndex":0}, action={}) {

switch(action.type){

//当发出type为changeTable的action对state的操作

case "changeTable":

let backup = state;

backup["tableIndex"] = action.data;

return Object.assign({}, state,backup);

default :

return Object.assign({}, state);

}

}

以上例子就是一个reducer，它是一个会对不同action做出不同操作的函数。

如当我发出切换table的action时，就是把我们之前定义action的data传递给state下的tableIndex变量，用来告诉state，我要切换table的序号。

在没有任何操作情况下，我们返回初始的state。

我们不直接去改变state的值，而是返回一个新的对象，保持state的唯一性。缓存可以在这里做，后面会有叙述

#### 3、Store:

var store = createStore(reducer);

这就是Store了，用来管理state的单一对象。其中有三个方法：

* store.getState()：获取state，如上，经过reducer已经返回了一个新的state，那么就可以用该函数获取；
* store.dispatch(action)：发出操作，更新state。action内有操作的类型，就可以出发不同的对state的更新；
* store.subscribe(listener)：监听变化，当state发生更新时，就可以在这个函数的回调中监听。

上述为基本redux的用法和含义，发出的操作也是同步的，如要更深入了解异步操作和对state更合理的逻辑管理，可以查看Middleware、combineReducers函数等，本文不再详述。

## react-redux

讲完了基本的redux，那么理解和react结合就简单的多了。

在使用react-redux之前，别忘了下载该模块。

react-redux在redux的基础上，就关注两点：**Provider**和**connect**。

#### 1、Provider:

<Provider store={store}>

<Router ref="router" history={hashHistory}>

<Route path='/' component={Index}>

<IndexRoute component={MainPage}></IndexRoute>

</Route>

</Router>

</Provider>

Provider就是把我们用rudux创建的store传递到内部的其他组件。让内部组件可以享有这个store并提供对state的更新。

#### 2、connect:

export default connect(mapStateToProps,mapDispatchToProps)(MainPage);

connect()一共有四个参数，但我这里只说基本的两个，**mapStateToProps**和**mapDispatchToProps**。

* mapStateToProps：简单来说，就是把状态绑定到组件的属性当中。我们定义的state对象有哪些属性，在我们组件的props都可以查阅和获取。
* const mapStateToProps = (state, ownProps) => {
* return {tableIndex:state.tableIndex}
* }

查看props

在props中，我们就看到了我们绑定的状态。

它的初值就是reducer默认返回state中的值。

* mapDispatchToProps：在redux中介绍过，用store.dispatch(action)来发出操作，那么我们同样可以把这个方法封装起来，即绑定到我们的方法中。
* const mapDispatchToProps = (dispatch, ownProps) => {
* return {
* changeActive:(args)=>dispatch({type:"changeTable",data:args})
* }
* }

可以看到，这个方法return的就是一个dispatch函数，将该方法绑定到属性上，我们同样可以在props查看和调用。

查看props

这样，我们只要在组件中调用该属性中的方法，就可以发出一个特定的action，触发reducer对state进行更新。这里的reducer就是之前在redux中定义的reducer。

这时候state如果发生更新，因为我们已经把state也绑定到props中，那么会把这些新的props重新传递下去，组件会进行更新渲染，达到我们需要的目的。

如果需要缓存怎么办？

还记得我们定义的reducer么。我们只需要将最新的状态存起来（无论是sessionStorage、数据库还是其它），然后默认的state去取缓存过的状态进行初始化渲染即可。

const reducer = function(state={"tableIndex":0}, action={}) {

switch(action.type){

//当发出type为changeTable的action对state的操作

case "changeTable":

let backup = state;

backup["tableIndex"] = action.data;

//定义新的state，用于存储

let newState = Object.assign({}, state,backup);

sessionStorage.setItem("state",JSON.stringify(newState));

return newState;

default :

if(sessionStorage.getItem("state")){

//获取缓存过的state

return Object.assign({},JSON.parse(sessionStorage.getItem("state")));

}else{

return Object.assign({}, state);

}

}

}