React

render函数():

https://www.cnblogs.com/accordion/p/4501118.html

1、在使用React进行构建应用时，我们总会有一个步骤将组建或者虚拟DOM元素渲染到真实的DOM上，将任务交给浏览器，进而进行layout和paint等步骤，这个函数就是React.render()

ReactComponent render(ReactElement element, DOMElement container, [function callback])

2、接收2-3个参数，并返回ReactComponent类型的对象，当组件被添加到DOM中后，执行回调。在这里涉及到了两个React类型--ReactComponent和ReactElement，着重分析。

ReactElement类型解读

ReactElement类型通过函数React.createElement()创建，接口定义如下：

ReactElement createElement(string/ReactClass type, [object props], [children ...])

第一个参数可以接受字符串（如“p”，“div”等HTML的tag）或ReactClass，第二个参数为传递的参数，第三个为子元素，ReactElement有4个属性：type，ref，key，props，并且轻量，没有状态，是一个虚拟化的DOM元素

React解决了哪些问题：

1、组件复用问题

2、性能问题：React做的一件事情，就是对于每个组件，它在内存里生成一棵虚拟dom树，当组件发生变化的时候，它会生成一颗新的虚拟dom树，然后和老的进行比较，找出有差异的节点，然后只更新发生变化的dom节点，所以更新的成本会比较小，性能也会更好。

3、浏览器兼容问题（Browser Suppert):主要是在事件方面，我们知道，各个浏览器的事件处理有可能不一致，比如取消事件冒泡：React使用SyntheticEvent，包装和规范了原生的浏览器事件，这样各个浏览器的事件行为都能够得到统一

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，然后触发所谓的调和过程（Reconciliation）。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个UI界面。在 React 得到元素树之后，React 会自动计算出新的树

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。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

React 中 Element 与 Component 的区别是

简单而言，React Element 是描述屏幕上所见内容的数据结构，是对于 UI 的对象表述。典型的 React Element 就是利用 JSX 构建的声明式代码片然后被转化为createElement的调用组合。而 React Component 则是可以接收参数输入并且返回某个 React Element 的函数或者类*f*

在什么情况下你会优先选择使用 Class Component 而不是 Functional Component

在组件需要包含内部状态或者使用到生命周期函数的时候使用 Class Component ，否则使用函数式组件

React 中 refs 的作用是什么

Refs 是 React 提供给我们的安全访问 DOM 元素或者某个组件实例的句柄。我们可以为元素添加ref属性然后在回调函数中接受该元素在 DOM 树中的句柄，refs 并不是类组件的专属，函数式组件同样能够利用闭包暂存其值:

React 中 keys 的作用是什么?

Keys 是 React 用于追踪哪些列表中元素被修改、被添加或者被移除的辅助标识。在开发过程中，我们需要保证某个元素的 key 在其同级元素中具有唯一性。在 React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染。

Controlled Component 与 Uncontrolled Component 之间的区别是什么?

受控组件（Controlled Component）代指那些交由 React 控制并且所有的表单数据统一存放的组件。譬如下面这段代码中username变量值并没有存放到DOM元素中，而是存放在组件状态数据中。任何时候我们需要改变username变量值时，我们应当调用setState函数进行修改。

```
class ControlledForm extends Component {
  state = {
    username: ''
  }
  updateUsername = (e) => {
    this.setState({
      username: e.target.value,
    })
  }
  handleSubmit = () => {}
  render () {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          value={this.state.username}
          onChange={this.updateUsername} />
        <button type='submit'>Submit</button>
      </form>
    )
  }
}
```

而非受控组件（Uncontrolled Component）则是由DOM存放表单数据，并非存放在 React 组件中。我们可以使用 refs 来操控DOM元素：

```
class UnControlledForm extends Component {
  handleSubmit = () => {
    console.log("Input Value: ", this.input.value)
  }
  render () {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          ref={(input) => this.input = input} />
        <button type='submit'>Submit</button>
      </form>
    )
  }
}
```

实际开发中我们并不提倡使用非受控组件，因为实际情况下我们需要更多的考虑表单验证、选择性的开启或者关闭按钮点击、强制输入格式等功能支持，而此时我们将数据托管到 React 中有助于我们更好地以声明式的方式完成这些功能。引入 React 或者其他 MVVM 框架最初的原因就是为了将我们从繁重的直接操作 DOM 中解放出来。

在生命周期中的哪一步你应该发起 AJAX 请求

- React 下一代调和算法 Fiber 会通过开始或停止渲染的方式优化应用性能，其会影响到 componentWillMount 的触发次数。对于 componentWillMount 这个生命周期函数的调用次数会变得不确定，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

如何告诉 React 它应该编译生产环境版本

通常情况下我们会使用 Webpack 的 DefinePlugin 方法来将 NODE_ENV 变量值设置为 production。编译版本中 React 会忽略 propType 验证以及其他的告警信息，同时还会降低代码库的大小，React 使用了 Uglify 插件来移除生产环境下不必要的注释等信息。

为什么我们需要使用 React 提供的 Children API 而不是 JavaScript 的 map

props.children并不一定是数组类型，如果我们使用props.children.map函数来遍历时会受到异常提示，因为在这种情况下props.children是对象（object）而不是数组（array）。React 当且仅当超过一个子元素的情况下会将props.children设置为数组，这也就是我们优先选择使用React.Children.map函数的原因，其已经将props.children不同类型的情况考虑在内了

概述下 React 中的事件处理逻辑

为了解决跨浏览器兼容性问题，React 会将浏览器原生事件（Browser Native Event）封装为合成事件（SyntheticEvent）传入设置的事件处理器中。这里的合成事件提供了与原生事件相同的接口，不过它们屏蔽了底层浏览器的细节差异，保证了行为的一致性。另外有意思的是，React 并没有直接将事件附着到子元素上，而是以单一事件监听器的方式将所有的事件发送到顶层进行处理。这样 React 在更新 DOM 的时候就不需要考虑如何去处理附着在 DOM 上的事件监听器，最终达到优化性能的目的。

createElement 与 cloneElement 的区别是什么

createElement 函数是 JSX 编译之后使用的创建 React Element 的函数，而 cloneElement 则是用于复制某个元素并传入新的 Props。

React Diff算法：

1、在React中，两棵DOM树只会对同一层的节点进行比较，若发现节点已不存在，则该节点及其子节点会被完全删除，不会用于进一步的比较。这样，只需要对树进行一次遍历，就能完成整个DOM树的比较

2、对于同层节点，React在数组遍历的增减关键字Key,若节点本身完全相同(类型相同，属性相同)，只是位置不同，则React只需要考虑同层节点的位置变换，不需要进行节点的销毁和重新创建

3、对于不同层的节点，只能销毁和重新创建

详细解释如下：

1）节点类型不同:

当在树中的同一位置前后的节点类型不同，React会直接删除原节点，然后创建并插入新的节点。

注意：删除节点即彻底销毁该节点，也就是说，后续不会查找是否有另外一个节点等同于删除的该节点。如果删除的该节点有子节点，那么子节点也会被删除。这也是diff算法复杂度能降到O(n)的原因。

同理，当树的同一个位置遇到前后不同的组件时，也是销毁原组件，把新的组件加上去。这应用了第一个假设，不同的组件一般会产生不同的DOM结构，与其浪费时间去比较不同的DOM结构，还不如完全创建一个新的组件加上去。

2)节点类型相同，但是属性不同:

React会对属性进行重设从而实现节点的转换

3）节点类型相同且属性相同

对于同层节点，若节点本身完全相同（类型相同且属性相同），只是位置不同，则React只需要考虑同层节点的位置变换，不需要进行节点的销毁和重新创建，这就需要用到下面介绍的key属性。

对于不同层的节点，即使节点本身完全相同（类型相同且属性相同），也只能销毁和重新创建。

4）key属性

为列表节点提供唯一的key属性，可以帮助React定位到正确的节点进行比较，从而大幅减少DOM操作的次数，提高性能

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

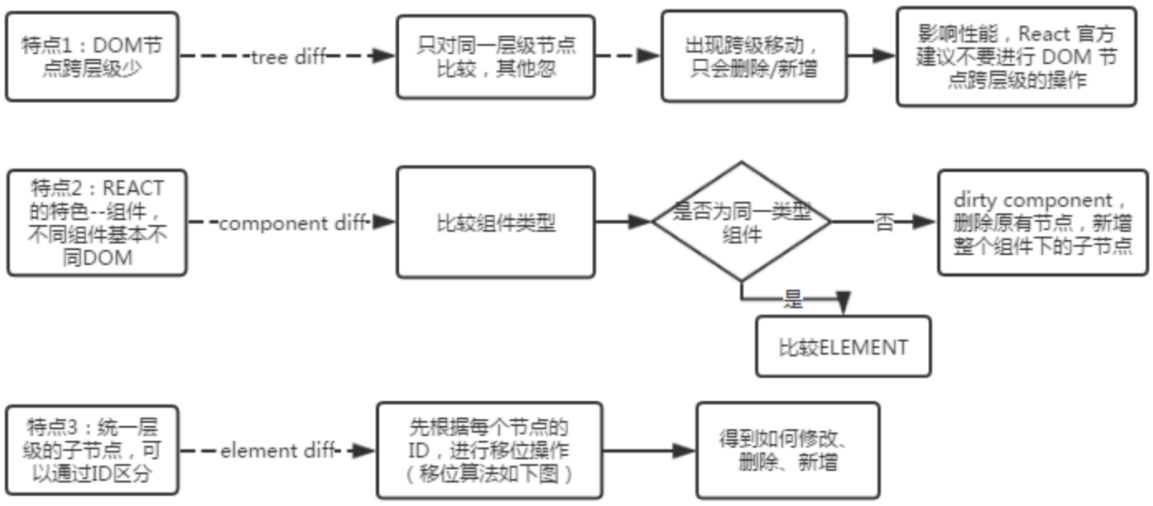
prop和state的主要区别在于：

1. prop是由外部传入的，是组件无法修改的
2. state是用于记录组件内部的状态的，因此组件可以修改

1、react的理解 ，以及如何优化 ，如何更新界面，当一个列表 有很多项，只修改一项的时候 如何做到不全部更新

1、React整个的渲染机制就是在state/props发生改变的时候，重新渲染所有的节点，构造出新的虚拟Dom tree跟原来的Dom tree用Diff算法进行比较，得到需要更新的地方在批量造作在真实的Dom上，由于这样做就减少了对Dom的频繁操作，从而提升的性能。

diff算法的特点如下图：传统 diff 算法的复杂度为 O(n^3)，单纯从demo看，复杂度不到n3，但实际上。React 通过制定大胆的策略，将 O(n^3) 复杂度的问题转换成 O(n) 复杂度的问题。



2、react给我们提供了一个方法shouldComponentUpdate(),当这个方法返回true的时候，需要重新渲染，false的时候不需要（默认是true).用Perf跟why-did-you-update两个工具已经可以很好帮我们判断哪部分不需要重新渲染，帮助我们做出优化。

react组件的划分业务组件技术组件?

根据组件的职责通常把组件分为UI组件和容器组件。

UI 组件负责 UI 的呈现，容器组件负责管理数据和逻辑。

两者通过React-Redux 提供connect方法联系起来。

2、react生命周期

生命周期共提供了10个不同的API。

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

在完成首次渲染之前调用，此时仍可以修改组件的state。

4.render

必选的方法，创建虚拟DOM，该方法具有特殊的规则：

- 只能通过this.props和this.state访问数据
- 可以返回null、false或任何React组件
- 只能出现一个顶级组件（不能返回数组）

- 不能改变组件的状态
- 不能修改DOM的输出

5.componentDidMount

真实的DOM被渲染出来后调用，在该方法中可通过this.getDOMNode()访问到真实的DOM元素。此时已可以使用其他类库来操作这个DOM。

在服务端中，该方法不会被调用。

6.componentWillReceiveProps

组件接收到新的props时调用，并将其作为参数nextProps使用，此时可以更改组件props及state。

```
componentWillReceiveProps: function(nextProps) {
    if (nextProps.bool) {
        this.setState({
            bool: true
        });
    }
}
```

7.shouldComponentUpdate

组件是否应当渲染新的props或state，返回false表示跳过后续的生命周期方法，通常不需要使用以避免出现bug。在出现应用的瓶颈时，可通过该方法进行适当的优化。

在首次渲染期间或者调用了forceUpdate方法后，该方法不会被调用

8.componentWillUpdate

接收到新的props或者state后，进行渲染之前调用，此时不允许更新props或state。

9.componentDidUpdate

完成渲染新的props或者state后调用，此时可以访问到新的DOM元素。

10.componentWillUnmount

组件被移除之前被调用，可以用于做一些清理工作，在componentDidMount方法中添加的所有任务都需要在该方法中撤销，比如创建的定时器或添加的事件监听器。

react性能优化是哪个周期函数?

shouldComponentUpdate 这个方法用来判断是否需要调用render方法重新描绘dom。因为dom的描绘非常消耗性能，如果我们能在shouldComponentUpdate方法中能够写出更优化的dom diff算法，可以极大的提高性能。

react中哪个生命周期监听props

componentWillReceiveProps

setState not working in componentDidMount() - React

The component does not listen to state modifications in didMount; that means that even if the state is updated the repaint is not triggered. a re-rendering

初始化-render-Did，不会再重复渲染

react this.setState有哪些函数?

setState(state,props){}本是一个异步函数，setState() 可以接收一个函数，这个函数接受两个参数，第一个参数表示上一个状态值，第二参数表示当前的 props，state拿到状态值之后，会根据react的元素树进行新旧差异化对比，把不一样的部分重新渲染出来

传入 setState 函数的第二个参数的作用是什么?

该函数会在setState函数调用完成并且组件开始重渲染的时候被调用，我们可以用该函数来监听渲染是否完成，

为什么是一个回调函数：this.setState 是在 render 时, state 才会改变调用的, 也就是说, setState 是异步的. 组件在还没有渲染之前, this.setState 还没有被调用.这么做的目的是为了提升性能, 在批量执行 State 转变时让 DOM 渲染更快.

现象：setState() does not immediately mutate

在哪个生命周期发送AJAX请求:

Didmount

原因：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，被触发多次，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。

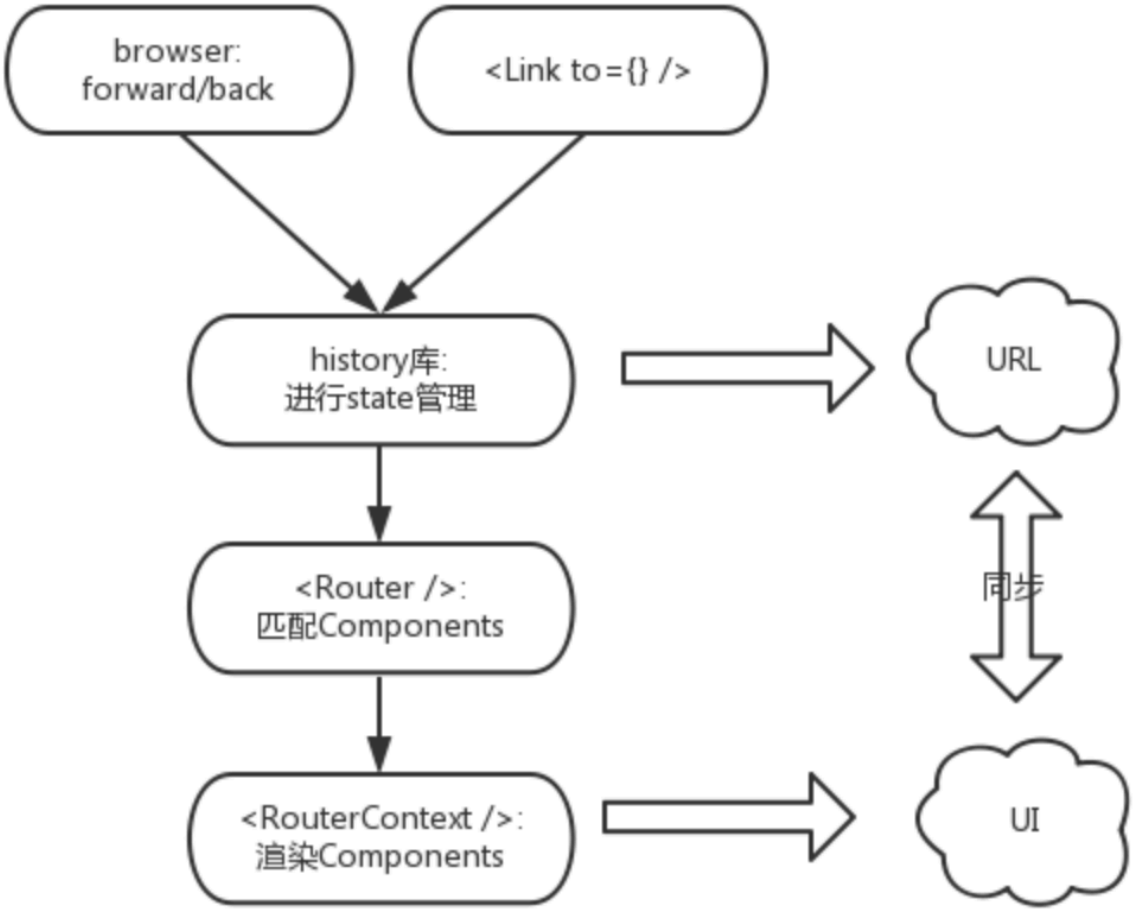
4、React router的实现原理

实现URL与UI界面的同步。其中在react-router中，URL对应Location对象，而UI是由react components来决定的，这样就转变成location与components之间的同步问题。

5、React router实现过程

react-router在history库的基础上，实现了URL与UI的同步，分为两个层次来描述具体的实现。

1、组件层面在react-router中最主要的component是
Router RouterContext Link，history库起到了中间桥梁的作用。



react-router的上层实现过程

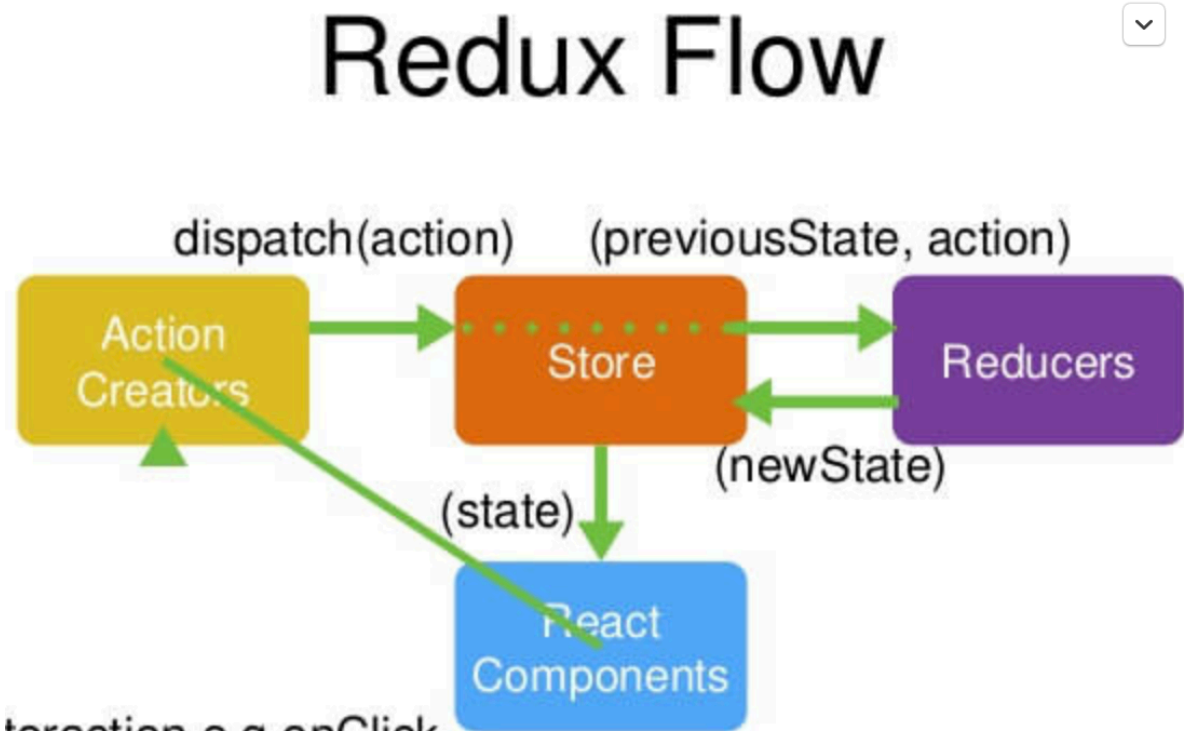
组件之间的通信:
父传子：this.props
子传父：利用回调函数，使用 props 回调,父组件将一个函数作为 props 传递给子组件，子组件调用该回调函数，便可以向父组件通信。
跨级或同级组件：redux
1、react推崇的是单向数据流，自上而下进行数据的传递，但是由下而上或者不在一条数据流上的组件之间的通信就会变的复杂。解决通信问题的方法很多，如果只是父子级关系，父级可以将一个回调函数当作属性传递给子级，子级可以直接调用函数从而和父级通信。
2、组件层级嵌套到比较深，可以使用上下文getChildContext来传递信息，这样在不需要将函数一层层往下传，任何一层的子级都可以通过this.context直接访问。

3、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。
4、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有什么缺点

1.一个组件所需要的数据，必须由父组件传过来，而不能像flux中直接从store取。
2.当一个组件相关数据更新时，即使父组件不需要用到这个组件，父组件还是会重新render，可能会有效率影响，或者需要写复杂的shouldComponentUpdate进行判断。

Redux:



1、首先，用户发出 Action
store.dispatch(action);
2、然后，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);
3、State 一旦有变化，Store 就会调用监听函数。

```
// 设置监听函数
store.subscribe(listener);
```

4、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。

```
function listerner() {
  let newState = store.getState();
  component.setState(newState);
}
```

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

7、react中如何获取一个dom元素的所有方法和属性

在react中，我们已经知道，组件并不是真实的DOM节点，而是通过虚拟DOM渲染出来的节点，只有当它被插入到文档后，才成为了真实的DOM。

要从组件中获取真实的DOM节点，则可在jsx标签中加入ref属性

写一个react的高阶函数

```
import React, { Component } from 'react';
import simpleHoc from './simple-hoc';
class Usual extends Component{
    render(){
        console.log(this.props,'props');
        renturn(
            <div>Usual</div>
        )
    }
}
export default simpleHoc(Usual);
const simpleHoc=WrappedComponent=>{
    console.log('simpleHoc');
    return class extends Component{
        render(){
            return <WrappedComponent{...this.props}/>
        }
    }
}
export default simpleHoc;
```

```
//操作props
import React,{Component} from 'react';
const propsProxyHoc=WrappedComponent=>class extends Component{
    handleClick(){
        console.log('click');
    }
    render(){
        return(
            <WrappedComponent
            {...this.props}
            handleClick={this.handleClick}
            />
        );
    }
};
export default propsProxyHoc;
```

```
const addFunc=WrappedComponent=>class extends Component{
    handleClick(){
        console.log('click');
    }
    render(){
        const props={
            ...this.props,
            handleClick:this.handleClick
        }
        return <WrappedComponent{...props}/>;
    }
};
const addStyle=WrappedComponent=>class extends Component{
    render(){
        return(
            <div style={{color:'red'}}>
            <WrappedComponent {...this.props}/>
            </div>
        )
    }
}
const WrappedComponent=addStyle(addFunc(Usual));
class WrappedComponent extends Component{
    render(){
        console.log(this.props,'props');
        return(
            <div>
            <WrappedComponent/>
            </div>
        )
    }
}
```