react 与 vue 的对比

相同点：1. 都用虚拟 DOM 实现快速渲染
2. react 和 vue 的数据流管理方案不同：react=>redux, vue=>vuex
3. 都是轻量级框架
4. 现在 vue 也在渐渐的向 react 中的一些语法，比如 JSX 语法，类式声明写法等

不同点：
1. React 属于单向数据流——MVC 模式，vue 则属于双向——MVVM 模式。
2. react 兼容性比 vue 好，vue 不兼容 IE8.
3. react 采用 JSX 语法，vue 采用的则是 html 模板语法。
4. react 比 vue 好的另一点是，它是团队维护，而 vue 属于个人，一般来说，大型项目更倾向于 react，小型则用 vue，但也不是绝对的。

比较 redux 和 vuex?

相同点：1.数据可以控制视图，提供响应式的视图组件
2.都有虚拟 DOM， 组件化开发，通过 props 参数进行父子组件数据的传递
3.都支持服务端渲染

不同点：1.vuex 是一个针对 VUE 优化的状态管理系统，而 redux 仅是一个常规的状态管理系统（Redux）与 React 框架的结合版本。
2.开发模式：React 本身，是严格的 view 层，MVC 模式；Vue 则是 MVVM 模式的一种方式实现
3.数据绑定：Vue 采取双向数据绑定的方式；React，则采取单向数据流的方式
4.数据更新：Vue 采取依赖追踪，默认是优化状态：按需更新；
React 在则有两种选择：
1）手动添加 shouldComponentUpdate，来避免重新 render 的情况
2）Components 可以使用 pureComponent，然后采用 redux 结构 + Immutable.js

总之：期待构建一个大型应用程序——选择 React，小型项目——选择 Vue

Immutalbe.js

js 中的对象是可变的，因为使用了引用赋值，新的对象引用了原始对象，改变新的对象会影响到原始对象。为了解决这个问题，一般就是使用浅拷贝或者深拷贝来避免被修改，但是这样会造成 cpu 和内存的浪费，所以可以使用 immutable

特点：
1 .一旦创建，就不能更改的数据，对 immutable 对象的任何修改或删除添加都会返回一个新的 immutable 对象

2 .实现原理就是持久化数据结构，在使用旧数据创建新数据的时候，会保证旧数据同时可用且不变，同时为了避免深度复制会复制所有节点带来的性能损耗，immutable 使用了结构共享，

什么是结构共享：如果一个节点发生变化，只修改这个节点和受他影响的父节点，其他节点就会共享。

immutable 优点

并发安全：传统的并发需要加锁，但是这个数据天生不可变，所以并发加锁就不需要了
节省内存
时间旅行，复制粘贴这些操作做起来非常简单

this.setState

1.Setstate 是同步还是异步

在 React 在合成事件（react 自己封装了一套机制代理原生事件）**和钩子函数中， setState 就是异步的；在 React 检测不到的地方，向 setInterval,setTimeout 里和原生事件， setState 就是同步更新的。**

异步指的不是执行异步代码，只是合成事件和钩子函数的调用顺序在更新之前，在合成事件和钩子函数中没法立马拿到更新后的值，但是可以通过 setstate 中的第二个参数获得更新后的结果。

2.什么是合成事件

React 合成事件（SyntheticEvent）是 React 模拟原生 DOM 事件所有能力的一个事件对象，即浏览器原生事件的跨浏览器包装器。如果 react 事件绑定在了真实 DOM 节点上，一个节点同时有多个事件时，页面的响应和内存的占用会受到很大的影响，所以用合成事件当中间层，它根据 W3C 规范 来定义合成事件，兼容所有浏览器，拥有与浏览器原生事件相同的接口。

8.setState 的更新过程会触发哪些生命周期

setState 调用引起的 React 的更新生命周期函数 4 个函数（比修改 prop 引发的生命周期少一个 componentWillReceiveProps 函数），这 4 个函数依次被调用。
shouldComponentUpdate
componentWillUpdate
render
componentDidUpdate

2.setState 的原理及用法？

当调用 setState 时，它并不会立即改变，而是会把要修改的状态放入一个任务队列，等到事件循环结束时在合并更新。
常见用法就是传入一个对象，还可以接收一个参数，因为 setState 是异步的，所以它还可以接收第二个参数，第二个参数是一个 callback 回调函数

3.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 React 会以相对高效的方式根据新的状态构建 React 元素树
并且重新渲染整个 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。

4.this.setState 为什么在原生中是同步的?

因为 react 自己封装了一套机制代理原生事件,这就是合成事件,
比如 onClick,在合成事件中有原生事件没有的功能,
　　点击 onClick 添加函数时，React 并没有将 Click 事件绑定在 DOM 上面。
　　而是在 document 处监听所有支持的事件，当事件发生并冒泡至 document 处时，React 将事件触发

5.为什么 react 中,this.setState({})为什么是异步的？

如果有多条 setState（{}）也会只重新 render 一次，因为 setState 会合并，异步的才可以进行合并。如果是同步的话，就会直接 render， 那样渲染效率会非常低。

6.setstate 重复调用

在 React 中 setState 不是每次调用就立刻渲染的。setState 多次调用并不会导致渲染多次。
原理:相当于对象合并 Object.assign，相同的操作后面会覆盖前面的
（比如在 click 事件过程中可能有很多 setState 在等待，等 Click 事件完成之后，setState 这个队列里面的内容才开始进行结算）

7.解决 setstate 重复调用会重复的问题

通过 this.setState 调用时，第一个参数传入一个参数，将前一个 state 参数（preState）的值

传进去

```
this.setState((prevState,props)=>({
count:prevState.count+1}));
```

10.一个组件中要读取当前状态用是访问 this.state，但是更新状态却是用 this.setState，不是直接在 this.state 上修改，为什么呢？

直接修改 this.state 的值，会发现的确能够改变状态，但是却不会引发重新渲染。

11.在 setState 之前能获取到更新之后的值吗？

不能，此时更新后的 state 值正在更新队列中，未归并到当前 state，所以无法获取

生命周期函数里可以 setState 吗？什么时候 setState 合适？

在 componentDidUpdate 中执行 setState 会导致组件刚刚完成更新，又要再更新一次，连续渲染两遍。 可以

在 componentWillReceiveProps 中可以 setState，不会造成二次渲染。由于只有 props 的变化才会触发 componentWillReceiveProps 事件，因为在这个事件里 setState 不会造成组件更新的死循环，可以在这个函数里 setState。 可以

在 componentWillUnmount 中执行 setState 不会更新 state，是不生效而且无意义的。不可以

禁止在 shouldComponentUpdate 和 componentWillUpdate 中调用 setState，这会造成循环调用，在两个函数又触发了 setState，然后再次触发这两个函数直至耗光浏览器内存后崩溃。不可以

React 组件

1.如何理解一切皆组件这句话？？

组件是 React 应用 UI 的构建块。这些组件将整个 UI 分成小的独立并可重用的部分。每个组件彼此独立，而不会影响 UI 的其余部分。

2.有状态组件和无状态组件的区别？？

无状态组是没有状态的组件即函数组件 它是一种函数式组件，没有 state，接收 Props，渲染 DOM，，如果一个组件不需要管理 state 只是纯的展示，那么就可以定义成无状态组件。

有状态的组件的话那么他就一定会触发生命周期定义的一些函数，一旦触发这些函数就会影响当前项目的运行，所以在尽可能的情况下使用无状态的组件

有状态组件能使用 this，无状态组件不能使用 this.refs

有状态组件：

1. 在内存中存储有关组件状态变化的信息 1. 计算组件的内部的状态
2. 有权改变状态 2. 无权改变状态
3. 包含过去、现在和未来可能的状态变化情况 3. 不包含过去，现在和未来可能发生的状态变化情况
4. 接受无状态组件状态变化要求的通知，然后将 props 发送给他们。 4.从有状态组件接收 props 并将其视为回调函数。

3.受控组件和非受控组件？

受控组件

受控组件只有继承 React.Component 才会有状态。

受控组件，表单元素的修改会实时映射到状态值上，此时就可以对输入的内容进行校验.

表单中有一个 input 标签，input 的 value 值必须是我们设置在 constructor 构造函数的 state 中的值，然后，通过 onChange 触发事件来改变 state 中保存的 value 值，所以说受控组件必须要在表单上使用 onChange 事件来绑定对应的事件.

value 和 onchange 两者在受控组件中缺一不可，一旦缺少其中一个就会报错。

获取文本框的值 ：文本框、下拉框都用的是 event.target.value，

多选框用的是 event.target.checked。

非受控组件就是不受状态的控制，获取数据就是相当于操作 DOM。

非受控也就意味着我可以不需要设置它的 state 属性，而通过 ref 来操作真实的 DOM。

一般没有 value，我们就可以认为这个组件是非受控组件，但是我们可以通过 defaultValue 来给初始值。

受控组件:即通过 setState 的形式控制输入的值及更新,

非受控组件:即通过 dom 的形式更新值，要获取其值可以通过 ref 的形式去获取。

受控和非受控元素都有其优点，根据具体情况选择。如果表单在UI反馈方面非常简单，则对ref进行控制是完全正确的，即使用非受控组件。

特征	非受控制	受控
一次性检索（例如表单提交）	yes	yes
及时验证	no	yes
有条件的禁用提交按钮	no	yes
执行输入格式	no	yes
一个数据的几个输入	no	yes
动态输入	no	yes

4.什么是高阶组件？？

高阶组件就是一个没有副作用的纯函数。接收一个组件（被包装组件），再返回一个新的包装组件，然后包装组件会向被包装组件传入特定的属性

高阶组件能干什么：高阶组件的主要功能是封装并分离组件的通用逻辑，让通用逻辑在组件间更好地被复用

代码复用（把一些公共功能包裹里面并添加一些属性封装里面那么一调用 connect 就会把属性添加给组件）

用到的高阶组件：

WithRouter：把不是通过路由切换过来的组件中，将 react-router 的 history、location、match 三个对象传入 props 对象上

Form.create： 传入的是 react 组件，返回一个新的 react 组件，在函数内部会对传入组件进行改造，添加上一定的方法，经 Form.create() 包装过的组件会自带 this.props.form 属性

5.react 组件性能优化:

1.类组件：可以在 shouldComponentUpdate 进行比较，是否进入组件的更新，但是 react 并没有实现这个方法，可以使用 pureComponetnt 进行浅层的比较，比对前后的 state 和 props 是否发生变化，决定是否重新 render

如果要是对象嵌套一层对象，就可以自己封装实现 shouldComponentUpdate 这个方法，遍历对象之后比对

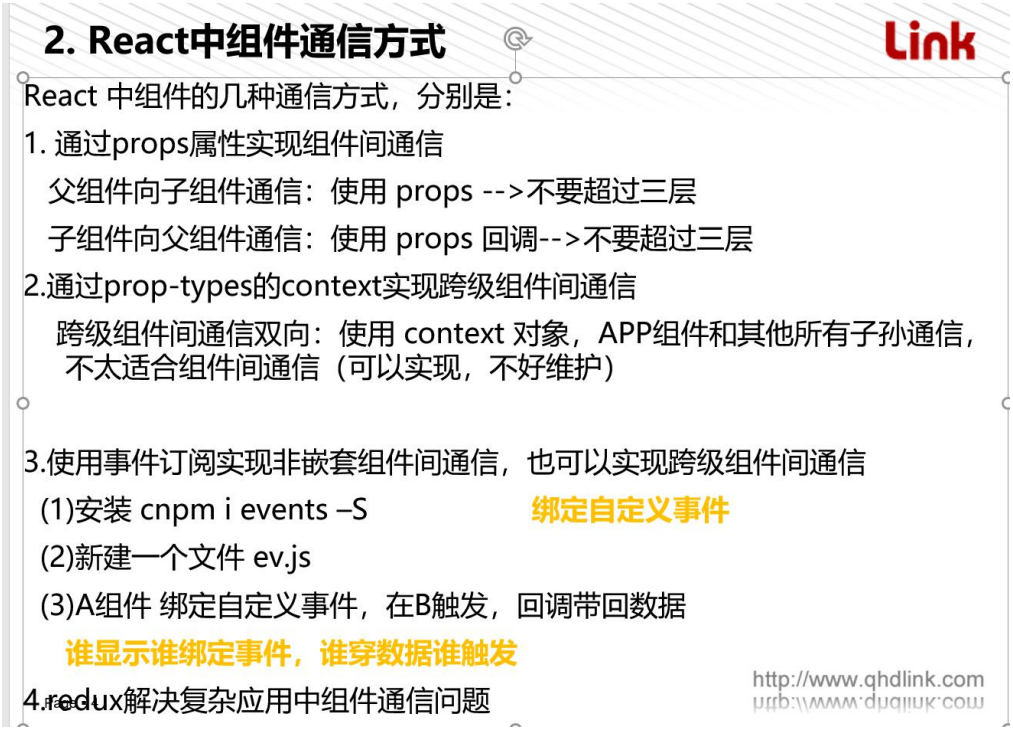
2.通过 key 值，使用 diff 算法，决定是否重新 render

3.静态组件中

React.Memo()高阶组件 结合 react hooks useCallback

react 传值问题

1.父子组件传值（父组件传值，组件通信）



2. 如何不通过 props 传值

1.通过 redux 的方式进行各个页面的数据传递，
2.路由传值：通过 url 传值，通过 localtion。search 和 match。parsms 传值
3.通过全局上下文 context 传值

父组件：

```
//需要先安装 prop-types 包
//设置传值的属性类型通过 propTypes
// 创建了一个全局 context 对象
App.childContextTypes={
    color:PropTypes.string,          //属性用于给后代传值
    callback:PropTypes.func          //方法用于后代给 App 传值,通过调用函数，
传递参数
 }

 //2.对 context 对象中，属性和方法进行初始化,（函数名不能变）
    getChildContext(){
```

```
                console.log("getChildContext 被调用了")
                return {
                    color:"red",            //传给后代的数据
                    callback:this.getMsg.bind(this) //后代回传数据调用的函数
                }
            }
                子组件：
                //声明当前组件支持 context 容器对象
                GrandChild.contextTypes={
                    color:PropTypes.string,
                    callback:PropTypes.func
                }
                //通过 this.context 使用数据
                <p>App 组件传过来的数据:{this.context.color}</p>
```

4.事件订阅

谁显示，谁绑定事件，　　谁传值谁触发事件

```
import {EventEmitter} from 'events'

import emitter from './ev.js'
  componentDidMount(){      //通过 emitter 绑定 addListener
        emitter.addListener("myEvent",(msg)=>{
            this.setState({msg})
        })
    }

//通过 emitter 的调用 emit 方法，触发事件
<button onClick={()=>emitter.emit("myEvent","测试哈哈")}>给 son1 穿得数据</button>
```

React 特点

1. 较高的性能
   --虚拟 DOM--js 对象
   --diff 算法
2. 虚拟 DOM
3. 组件化--主要为了实现代码高度复用
4. jsx 语法:在 js 中写 xml/xhtml 代码，快速生成虚拟 DOM
5. 单向响应数据流:父-->子 方便开发和维护

1.jsx 语法通过 babel 转换成什么（为什么浏览器无法读取 jsx）？

浏览器只能处理 JavaScript 对象，而不能读取常规 JavaScript 对象中的 JSX。所以为了使浏览器能够读取 JSX，首先，需要用像 Babel 这样的 JSX 转换器将 JSX 文件转换为 JavaScript 对象，然后再将其传给浏览器。

2. diff 算法 原理（常考，大厂必考）

在页面一开始打开的时候，React 会调用 render 函数构建一棵虚拟 DOM 树，在 state/props 发生改变的时候，render 函数会被再次调用渲染出另外一棵虚拟 DOM 树，接着 diff 算法会判断两棵 dom 树的差异，把差异更新到真实 DOM 中去。

React 基于两个假设：

a.两个相同的组件产生类似的 DOM 结构，不同组件产生不同 DOM 结构
b.对于同一层次的一组子节点，它们可以通过唯一的 key 区分
①节点类型不同，react 会直接删去旧的节点，新建一个新的节点。
②节点类型相同，但是属性不同，只改变需要改变的属性。
③列表比较，为每个子元素添加一个唯一的 key，React 使用 key 将原始树中的子元素与后续树中的子元素进行匹配。

3.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，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

4.真实 dom 和虚拟 dom 的区别？？

真实 dom:更新慢，可以直接更新 html,如果元素更新，则创建新 dom,dom 操作代价高，消耗内存较多。
虚拟 dom: 虚拟 DOM 就是一个 js 对象，他有 tag，props，和 children 属性，虚拟 DOM 更新快，无法直接更新 Html ，如果元素更新，则更新虚拟 DOM 操作非常简单 很好的内存消耗

5.虚拟 DOM 好在哪？为什么虚拟 dom 会提高性能?(必考)

1. 用 js 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。
2. 当状态变更的时候，重新构造一棵新的对象树。然后对比新旧虚拟 DOM 树，记录两棵树差异。
3. 把差异应用构建的真正的 DOM 树上，视图就更新了。

原因：虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 diff 算法减少了对真实 DOM 的操作次数，从而提高性能。

6.React 中 refs 的作用是什么？

可以通过 refs 获取 dom，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方法。我们可以为元素添加 ref 属性然后在回调函数中接受这个元素在 DOM 树中的 dom 结构

7.react 怎么从虚拟 dom 中拿出真实的 dom？

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，可以为元素添加 ref 属性然后在回调函数中接收该元素，该值会作为回调函数的第一个参数返回，或者 ref 可以传字符串

8.ref 的几种使用形式

1.字符串

通过 this.refs.a 来引用真实 dom 的节点 <input type ="text" ref="a"/>

再 dom 节点上使用

2.回调函数

回调函数就是在 dom 节点或组件上挂载函数，函数的入参是 dom 节点或组件实例，达到的效果与字符串形式是一样的，都是获取其引用。

<input type="text" ref={(input)=>{this.textInput=input}}

3.React.createRef()

在 React 16.3 版本后，使用 React.createRef()来创建 ref。将其赋值给一个变量，通过 ref 挂载在 dom 节点或组件上，该 ref 的 current 属性将能拿到 dom 节点或组件的实例

react 生命周期函数

0.生命周期函数

初始化阶段：
getDefaultProps:获取实例的默认属性
getInitialState:获取每个实例的初始化状态　　15.6 之前的

componentWillMount：组件即将被装载、渲染到页面上，只调用一次（可以在调用 echars 的时候，在 antd 中改变一个值将所有的颜色都改变）
render:组件在这里生成虚拟的 DOM 节点
componentDidMount:组件真正在被装载之后（在这步请求 ajax，因为在这一步渲染出了真实 dom 节点，不会去避免一些不必要的更新不到的值，在这一步我们能完全加载出来）
更新阶段：
componentWillReceiveProps:组件将要接收到属性的时候调用
shouldComponentUpdate（**nextProps, nextState**）:组件接受到新属性或者新状态的时候判断是否进入更新状态。
使用用 redux 传输数据,当前组件 connect 连接到仓库，另外一个组件的值连接到仓库，另外一个仓库的值发生改变，而我这个组件也连接了仓库，仓库里 state 发生改变，这个组件 view，state 都没改变的组件也是要重新渲染的这就极大的浪费了性能，所以需要在 shouldComponentUpdate 进行一层优化在进行浅层的对比，判断我们的 state 和 props 有没有发生改变，
（可以根据 nextProps 和 nextState 与当前的 props 和 state 值进行判断，未发生变化可以返回 false，接收数据后不更新，可以阻止 render 调用，）

优化：（可以用 purecomponent 来进行优化，就是在 shouldUpdate 的那一步再一次进行了 diff 算法封装，加了浅层的对比，）
componentWillUpdate:组件即将更新不能修改属性和状态
render:组件重新描绘
componentDidUpdate:组件已经更新
销毁阶段：
componentWillUnmount:组件即将销毁

2.废除的生命周期

componentWillMount
componentWillRecieveProps
componentWillUpdate

3.新增的生命周期

（1）静态函数 static getDerivedStateFromProps（nextProps，prevState）{}
用于替换 componentWillReceiveProps，放在挂载阶段 render 的前面（约束代码，因为它是一个静态函数，不能用 this,只能通过 prevState 而不是 prevProps 来做对比,保证了 state 和 props 之间的简单关系 ）

也就是这个函数不能通过 this 访问到 class 的属性，也并不推荐直接访问属性。而是应该通过参数提供的 nextProps 以及 prevState 来进行判断,根据新传入的 props 来映射到 state，并且不需要处理第一次渲染时 prevProps 为空的情况

如果 props 传入的内容不需要影响到你的 state，那么就需要返回一个 null，这个返回值是必须的，所以尽量将其写到函数的末尾

（2）getSnapshotBeforeUpdate（）{}四 nai p
用于替换 componentWillUpdate 放在 componentDidMount 后面 用于获得最新的 DOM 数据

// 返回的值作为 componentDidUpdate 的第三个参数
（由于 fiber 的出现，render 可能会被打断，willupdate 取到的值有可能跟我们想要的不一样）

（3）componentDidCatch（error，info）{}
如果一个组件定义了 componentDidCatch 生命周期，则他将成为一个错误边界(错误边界会捕捉渲染期间、在生命周期方法中和在它们之下整棵树的构造函数中的错误，就像使用了 try catch，不会将错误直接抛出了，保证应用的可用性)

1.componentUnmount 如何使用

组件卸载时调用 componentWillUnmount 方法，

卸载组件时触发生命周期方法：

```
unmountClick:function(){
        ReactDom.unmountComponentAtNode(document.getElementById("app"))
    },
```

使用场景：在这里清除一些不需要的监听和计时器

```
  componentWillUnmount() {
      document.removeEventListener("click", this. unmountClick);
   }
```

2.render 可以返回什么类型的数据

数组和字符串

3.组件移除是会触发那个生命周期方法？

Componentwillunmount()

4.react 中异步请求通常放在哪一个生命周期中？

1. 放在 constructor 或者 componentWillMount 里面反而会更加有效率。

解释：获取数据肯定是以异步方式进行，不会阻碍组件渲染（只会耽误请求发送这个时间），然后接着渲染，等异步返回数据后，如果成功再进行 setState 操作，setState 是将更新的状态放进了组件的__pendingStateQueue 队列，react 不会立即响应更新，会等到组件挂载完成后，统一的更新脏组件（需要更新的组件）。所以放在 constructor 或者 componentWillMount 里面反而会更加有效率。

5.shouldComponentUpdate 是做什么的，（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 diff 算法，可以极大的提高性能。
看返回值

redux?

Redux 的使用：

通用集中式数据流管理方案，实现组件跨层通信
函数式编程--redux 提供的 API 都是函数

store: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
State:包含所有数据，表示某个时点的数据集合
Action: State 的变化，会导致 View 的变化，就是 View 发出的通知，表示 State 应该就要发生变化了。

Action Creator:View 要发送多少种消息，就会有多少种 Action,我们定义 Action Creator 函数来生成 Action

Reducer: Store 收到 Action 以后，必须给出一个新的 State,这样 View 才会发生改变，这种 State 的计算过程就叫 Reducer

它是一个函数，它接收 Action 和当前 State 作为参数，返回一个新的 State

dispatch: 是 View 发出 Action 的唯一方法

redux 是一个应用数据流框架，主要是解决了组件间状态共享的问题，原理是集中式管理，主要有三个核心方法，action，store，reducer，

工作流程是 ：

用户点击了一个 ui 按钮，通过 dispatch，发出 Action，然后 store 自动调用 reducer,并且传入了两个参数，当前 State 和收到的 Action.Reducer 会返回新的 State.每当 state 更新之后，view（视图层）会根据 state 触发重新渲染

1.Redux 原理：

react-redux 是一个轻量级的封装库，它主要通过两个核心方法实现：

Provider： 从最外部封装了整个应用，并向 connect 模块传递 store，组件可以获得 store 中的 state。

Connect：

1、包装原组件，将 state 和 action 通过 props 的方式传入到原组件内部。

2、监听 store tree 变化，使其包装的原组件可以响应 state 变化

0. redux 优势：

redux 的三大优势：

1、单一数据源

2、状态是只读的

3、状态的改变只能通过纯函数操作

2.Redux 使用场景：

同一个 state 需要在多个 Component 中共享

需要操作一些全局性的常驻 Component，比如 Notifications，Tooltips 等

太多 props 需要在组件树中传递，其中大部分只是为了透传给子组件

业务太复杂导致 Component 文件太大，可以考虑将业务逻辑拆出来放到 Reducer 中

3.Redux/React 的性能优化？

1render 里要尽量减少新建变量和 bind 的使用，传递参数时尽量减少传递参数的数量，bind 一般放在 constructor 中
2shouldComponentUpdate 是决定 react 组件什么时候能够不重新渲染的函数。这个函数默认的实现方式就是简单的返回一个 true。
默认每次更新的时候都会调用所用的生命周期函数，包括 render 函数，重新渲染。
3 immutable 对象管理状态，让状态不能被更改。
4 key 值唯一 利用 diff 算法中的 key 值
5 将组件 component 更换为 pureComponent 也可以进行优化
6 可以用到 reselect（数据获取时优化的）都是优化渲染来提高性能的 原理是 只要相关的状态没发生改变，那么就直接使用上一次的缓存结果

4.redux 中间件？

Logger 中间件:负责打印我们之前的 state 和之后的 state

-thunk 中间件作用？

负责拦截 action, **可以改写 action function，让 action function 的返回值是一个函数**并且在函数中传入一个 dispatch—

1.可以在这个函数中执行 ajax，

2.然后通过 dispatch 发送 action, action 中携带 ajax 返回的数据给 reducer，
然后通过自己的 next 方法转发 action

next 方法的作用？如果只有一个中间件，就会直接转发给 reducer,
run 方法自动调用 next 方法并且执行。
thunk 用来执行异步操作的。
优点：使用方便，可以执行异步操作。
缺点：(1).一个异步请求的 action 代码过于复杂，且异步操作太分散，相对比 saga 只要调用一个 call 方法就显得简单多了。

(2).action 形式不统一，如果不一样的异步操作，就要写多个了。

Redux-promise 简化了 thunk 的写法，也可以用来执行异步操作

5.redux-saga?

根据 generator 函数来执行异步操作，用同步的写法写异步代码，有个 run 方法，会遍历 yield 语句
会返回一个遍历器对象，

优点：（1）集中处理了所有的异步操作，异步接口部分一目了然(有提供自己的方法)

（2）action 是普通对象，这跟 redux 同步的 action 一模一样({type:XXX})
（3）通过 Effect，方便异步接口的测试
（4） 异步操作的流程是可以控制的，可以随时取消相应的异步操作。
（5）通过 worker 和 watcher 可以实现非阻塞异步调用，并且同时可以实现非阻塞调用下的事件监听

6.redux-saga 中的方法的 redux-saga 的副作用？

takeEvery:两个参数，第一个是接收参数，第二个是要执行的 generaor 函数
takeLastest:防止重复提交，当点击一次 只会执行最后一次
put 作用：转发 action
call 作用？阻塞的作用，异步不执行完毕，不会执行后面的 yield 语句
fork 的作用？非阻塞，不会等待结果，直接执行后面的 yield 语句
all 的作用？合并 saga

7.reducer 是什么

reducer 是 redux 的三个核心概念之一，它指定了应用状态的变化如何响应 actions 并发送到 store，需要由开发人员自己定义，reducer 是一个纯函数

8.redux 中的 reducer 有啥特点

(1) 它必须是纯函数 - 这意味着在输入不变的情况下，永远应该返回相同的结果。
(2) 它应该没有副作用 - 像访问全局变量、发起异步请求或等待 promise 解析这样的操作都不应该用在此处。

react-router?路由

2.react-router 的优缺点：

1.风格: 与 React 融为一体,专为 react 量身打造，编码风格与 react 保持一致，例如路由的配置可以通过 component 来实现
2.简单: 不需要手工维护路由 state，使代码变得简单
3.强大: 强大的路由管理机制
路由配置: 可以通过组件、配置对象来进行路由的配置
路由切换: 可以通过<Link> Redirect 进行路由的切换
路由加载: 可以同步记载，也可以异步加载，这样就可以实现按需加载

1.react-router 的实现原理？

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，这就转变成 location 与 components 之间的同步问题。

withRouter 高阶组件，自带三个对象 history,location,match
通过高阶组件包裹可以获得这三个对象实现路由跳转 传值
动态路由？就是通过路由传值，一般通过 router 里的 path 路径后面要添加
传递的值可在 location 里的 search 获取我们传递的内容，也可以通过 link 标签的 to 的地址后面
---

exact 是路由的精确匹配

{/* Switch 只会匹配第一个路由， */}

Redirect 路由重定向，匹配路由后，加载其他路由

<Route path='/home' component={Home} />
Path 是匹配的路由，component 是加载的组件

<NavLink to="/" exact activeStyle={myStyle}>Home 首 页</NavLink>
相当于 a 标签
NavLink 在获取到 url 的时候有一个样式表，可以直接获得样式,默认绑定属性名：action 和 selected
---

react 的不足？

react 中只是 MVC 模式的 View 部分，要依赖引入很多其他模块开发。
当父组件进行重新渲染操作时，即使子组件的 props 或 state 没有做出任何改变，也会同样进行重新渲染。

Hooks

1.hooks 有哪些钩子

1. useState

具有类似 this.state 功能，让静态组件具备 state
参数一个是 state 名，一个是回调函数，useState 可以设置初始值
const [count,setCount]=useState(0);
改变 state，直接调用第二个方法即可
<button onClick={()=>setCount(count+1)}>+</button>

2. useEffect

具有生命周期方法的功能，让静态组件具有生命周期方法
--componentDidMount
--componentDidUpdate
--componentWillUnmount
useEffect 中的值，第一个是一个回调函数，可以再里面请求数据，写异步操作
第二个参数如果没有，任何 state 值改变都会调用，如果第二个参数是一个空数组，就相当于 componentDidMount，只在页面初始化时候调用，如果数组中有值，就在监控的 state 值发生改变时调用 userEffect

uerEffect 配合 async、await 使用
使用 useEffect 时，外侧不要有循环，条件或嵌套函数
react.Memo
类组件中使用 pureComponent 可以进行浅层的比较，函数组件中可以使用 react。Memo 对子组件进行包裹，使子组件在获得 props 属性的时候，可以进行前后的对比

3. useCallback

对匿名函进行缓存---重新 render，就不再重新创建匿名函数

4. useMemo

调用函数对函数返回的私有函数进行缓存 --重新 render，就不再重新创建私有函数
说明:useCallback useMemo 结合高级组件 React.memo() 对静态组件进行优化

<Child name={name} setName={useCallback(()=>{setName("jarry")},[])}/> --匿名，不会调用

<Child name={name} setName={useMemo(doSomeThing,[])}/> --有名，会调用

说明:useCallback 缓存函数的引用，useMemo 缓存计算数据的值。

唯一区别就是：
useCallback 是根据依赖缓存第一个入参的(callback)。
useMemo 是根据依赖缓存第一个入参(callback)调用后的值。
useMemo 会调用一次，useCallback 不会调用

5. useReducer

是 useState 的另一种替代方案，也可以修改 state，重新 render

```
//      --将初始 state initialState 赋值给 state
//      --注册 reducer，但是不调用
//      --解构出来一个 dispatch 函数
```

useReducer 会接收 reducer 和一个初始 state 值，解构可以获得 state 值和 dispatch，可以显示 state 值和通过 dispatch 发送 action

```
let [state,dispatch]=useReducer(reducer,初始 state 的值)
  // 与 react 不同的 useReducer 会注册 reducer，但是不会调用 reducer，
```

6. useContext

声明使用全局容器数据， 可以实现组件跨层通信，全局共享数据 context，类似于 prop-types 提供 context

--<1>.创建全局容器 const Context= React.createContext({defaultValue})
--<2>.向后代传递数据 <Context.Provider value={要传递的数据}></Contxt.Provider>
--<3>.声明使用全局容器数据获取 state 和 dispatch

```
let {state,dispatch} = useContext(CountContext)
```

说明:useReducer+useContext 实现 redux

7. 7. 自定义 hook

可以自定义 hook，返回 state 值，和一些事件方法，调用 useState 中的方法，修改 state，在组件中调用自定义 hook，可以返回 value 值，通过 value 值调用自定义 hook 中的 state 和自定义 hook 中的方法

1.怎样实现组件的复用，如何通过 hook 实现，什么是自定义 hook

使用 React 自定义 Hooks，我们可以把请求数据、更新加载状态、更新错误状态的代码全部写到自定义的 Hooks 中，然后在需要它的组件中调用即可，一般只需要一行代码。

React 自定义 Hooks 可以让业务逻辑从展示组件中抽离出来，并可以多次复用，极大的减少了代码量，提高了效率。

它的特点和注意事项：

自定义 hook 以 use 开头
可以接受参数
可以返回任何类型的返回值
可以使用其他内置或自定义的 hooks
每次调用 hook，其中的状态和逻辑都是隔离的

2.为什么使用 hooks，hooks 优势

1.React Hooks 不必写 class 组件就可以用 state 和其他的 React 特性；
2.你也可以编写自己的 hooks 在不同的组件之间复用；

hooks 优势：

2.更容易复用代码:它通过自定义 hooks 来复用状态，从而解决了类组件有些时候难以复用逻辑的问题
3.函数式编程风格:函数式组件、状态保存在运行环境、每个功能都包裹在函数中，整体风格更清爽，更优雅
4.代码量更少
5.更容易拆分组件
1.没有破坏性改动
完全可选的。 你无需重写任何已有代码就可以在一些组件中尝试 Hook。
100% 向后兼容的。 Hook 不包含任何破坏性改动。

3.如果子组件是函数组件不存在生命周期和钩子怎么进行性能优化？

（1）hooks 中的 usermemo　　（2） hooks 中的 useCallback

3-1.hooks 优化,useCallback 功能

用 React.memo 进行包裹

父组件向子组件重新 render 的话　子组件也会进行重新 render 即使传的值每次都是一样的子组件也会重新进行 render

但是向子组件传函数的话 还是每次都渲染子组件 可以用 usecallback 进行优化

用 usecallback 进行包裹

4.useCallback 和 useMemo 关系和区别?

相同点：

useCallback 和 useMemo 参数相同，第一个参数是函数，第二个参数是依赖项的数组。

useMemo、useCallback 都是使参数（函数）不会因为其他不相关的参数变化而重新渲染。

与 useEffect 类似，[] 内可以放入你改变数值就重新渲染参数（函数）的对象。如果 [] 为空就是只渲染一次，之后都不会渲染。

不同点，区别：

主要区别是 React.useMemo 将调用函数并返回其结果，

而 React.useCallback 将返回函数而不调用它

5.useEffect 的执行是怎样的？

useEffect 具有生命周期方法的功能，让静态组件具有生命周期方法

--componentDidMount

--componentDidUpdate

--componentWillUnMount

useEffetc(()=>{})　　任何状态变量改变 componentDidMount/componentDidUpdate 都会执行

useEffect(()=>{},[状态变量])　　仅这个状态变量改变 componentDidMount/componentDidUpdate 都会执行

useEffect(()=>{},[])　　仅调用 componentDidMount，不会调用 componentDidUpdate

userEffect(()=>{return 语句体}) 当移除组件时，回执行 return 后的语句

6.hooks 优化

用 react.memo 进行包裹
父组件向子组件重新 render 的话　子组件也会进行重新 render 即使穿的值每次都是一样的子组件也会重新进行 render
但是向子组件传函数的话 还是每次都渲染子组件 可以用 usecallback
用 usecallback 进行包裹