一.react 和vue不同点

理念上的不同重点：

真正想深入的是这些方面：
//1.同：共同点单向数据流）
单向数据流核心是在于避免组件的自身(未来可复用)状态设计，它强调把 state 拿出来进行集中管理。

//2.1.异：Vue 进行数据拦截/代理，它对侦测数据的变化更敏感、更精确
而真正我认为 React 和 Vue 在理念上的差别，且对后续设计实现产生不可逆影响的是：Vue 进行数据拦截/代理，它对侦测数据的变化更敏感、更精确，也间接对一些后续实现(比如 hooks，function based API)提供了很大的便利。在Vue中，一个组件在渲染期间依赖于自动追踪，因此系统知道提前预判哪一个组件需要渲染当组件状态发生改变时。每个组件可以被认为具有自动为你实现shouldComponentUpdate，不需要注意嵌套的组件。

//2.2.react:单向数据流：setState() DOMdiff。
React 推崇函数式，它直接进行局部重新刷新(或者重新渲染)，这样更粗暴，但是更简单，让我们的开发的最初，就是刷新。但是 React 并不知道什么时候“应该去刷新”，触发局部重新变化是由开发者手动调用 setState 完成。在react中，当组件状态改变时，它会触发整个子组件数重新渲染，以根组件作为渲染基点。为了避免不必要的子组件重新渲染，你需要使用PureComponent或者实现 shouldComponentUpdate。(相比之下，Vue 由于采用依赖追踪，默认就是优化状态：你动了多少数据，就触发多少更新，而 React 对数据变化毫无感知，它就提供 React.createElement 调用已生成 virtual dom)。

另外 React 为了弥补不必要的更新，会对 setState 的行为进行合并操作。因此 setState 有时候会是异步更新，但并不是总是“异步”。

主流框架的数据单向/双向绑定实现原理？

不同点：

1.
---vue 给data赋值直接用this.属性
--react 改变state 需要this.setState()
虽然都是异步渲染，但是两者不太一样。
react不允许直接更改状态，react的setState是异步的,批量地对state进行更新以提高性能的,减少渲染次数.
没办法做到检测属性变化直接驱动render函数的执行，所以依赖setState调用.（如果需要通过setState第二个参数传入callback能让你拿到更新后的state）

vue响应式更新，通过过劫持可以精确响应更新,检测到值的改变就会重新渲染(vue 值会立刻改变，但是 DOM 是异步更新的)
关于Vue，当你把一个普通的 JavaScript 对象传给 Vue 实例的data选项，Vue 将遍历此对象所有的属性，并使用Object.defineProperty把这些属性全部转为getter/setter。从而实现响应式。
vue中属性变化,首先触发Object.defineProperty中属性的set监听,执行updateComponent方法(异步),通过vm._render()更新vNode(新旧node对比),最后渲染到html中

2.模板
vue template,在html 写js,开发效率高，api更多
react jsx,在js里面写html,一切都是js 可以实现复杂需求

3.优化
Vue 应用中，组件的依赖是在渲染过程中自动追踪的，所以系统能精确知晓哪个组件确实需要被重渲染

react 优化shouldComponentUpdate 决定组件是否需要重现渲染

4.状态管理

React是一个用于创建可重用且有吸引力的UI组件的库。它非常适合代表经常变化的数据的组件。使用React，您可以通过将它们分解为组件而不是使用模板或HTML来构建可重用的用户界面。

同：

1.数据驱动,单向数据流，简而言之，单向数据流就是model的更新会触发view的更新，view的更新不会触发model的更新，它们的作用是单向的
2.都是虚拟dom
3.都能实现mvvm

react不是mvvm也不是MVV,React just view,没有所谓的状态管理，只有数据到视图，ui = render (data)

vm是view mode的意思。所以mvvm框架是要有一个vm对象，来映射view。也就是vm对象的属性发生改变的时候，对应的视图部分会相对应更新。

react，它没有纯粹意义上的vm对象，它有的是属性和状态。用属性和状态去映射视图。

那么属性和状态和vm有什么区别呢？个人认为，vm对象不管你值是从外部传进来的还是自己内部定义的，最后都一视同仁，mvvm框架都有双向绑定的概念

而react，强调的是属性不可变性，单向数据流。内部的状态内部自己控制。这样的设计可能从设计上更复杂一些，但是从使用上变得更确定，更清晰了。如果react用的比较熟，给合适的组件管理合适的状态，做好状态的合理分层，会大大降低应用复杂度。然后，redux有个很先进的概念叫容器组件和纯展示组件，如果领悟了这个设计思路的话，把复杂的东西集中到少部分组件中，大部分组件就变成纯展示组件，进一步降低应用复杂性。

因为单纯的 react 本质就是一个 render 函数。入参是 state，也就是 model，出参是 html，也就是渲染结果交给浏览器。这也是为什么 react 会和函数式，以及不可变数据结构扯上关系的原因。
一个用于 render 的函数，划分为 v 是没错的吧。state 也就是 model 应该包含在 react 中吗？私以为是不包含的，因为那是入参，是使用者提供的，react 对 model并没有任何操作，你每一次状态更新的时候，在其看来都是一个“新的”state，然后返回一个“新的”结果。
反观 vue，内部也有 state，但是并不是作为入参而是作为 vue 组件的一部分，因为只有内部的这个 state 是和 view 绑定的，对这个 state 进行修改的时候会同步修改对应的那一部分 view。至于 c，你说我这么一个函数式设计思路的库，追求的就是无状态无副作用，被钦点为 c 是不是有点说不过去。

详细解释 React 组件的生命周期方法_重点

一些最重要的生命周期方法是：

1.componentWillMount() - 在渲染之前执行，在客户端和服务器端都会执行。---->React16弃用
2.componentDidMount() - 仅在初次渲染后在客户端执行。
3.componentWillReceiveProps() - 当从父类接收到 props 并且在调用另一个渲染器之前调用。---->React16弃用
4.shouldComponentUpdate() - 根据特定条件返回 true 或 false。如果你希望更新组件，请返回 true 否则返回 false。默认情况下，它返回 false。
5.componentWillUpdate() - 在 DOM 中进行渲染之前调用。---->React16弃用
6.componentDidUpdate() - 在渲染发生后立即调用。
7.componentWillUnmount() - 从 DOM 卸载组件后调用。用于清理内存空间。

react16之前的生命周期

- 初始化阶段

- - `constructor` 构造函数
  - `getDefaultProps``props`默认值
  - `getInitialState``state`默认值

- 挂载阶段

- - `componentWillMount` 组件初始化渲染前调用
  - `render` 组件渲染
  - `componentDidMount`组件挂载到 `DOM`后调用

- 更新阶段

- - `componentWillReceiveProps` 组件将要接收新 `props`前调用
  - `shouldComponentUpdate` 组件是否需要更新
  - `componentWillUpdate` 组件更新前调用
  - `render`组件渲染
  - `componentDidUpdate` 组件更新后调用

- 卸载阶段

- - `componentWillUnmount` 组件卸载前调用

总结:

只执行一次： constructor、componentWillMount、componentDidMount

执行多次：render 、子组件的componentWillReceiveProps、componentWillUpdate、componentDidUpdate

有条件的执行：componentWillUnmount（页面离开，组件销毁时）

不执行的：根组件（ReactDOM.render在DOM上的组件）的componentWillReceiveProps（因为压根没有父组件给传递props）

2.react16生命周期:

React16`新的生命周期弃用了 `componentWillMount、componentWillReceivePorps，componentWillUpdate`新增了 `getDerivedStateFromProps、getSnapshotBeforeUpdate`来代替弃用的三个钩子函数。

React16并没有删除这三个钩子函数，但是不能和新增的钩子函数混用， `React17`将会删除这三个钩子函数，新增了对错误的处理（ `componentDidCatch`）

```
- 初始化阶段

- - `constructor` 构造函数
  - getDefaultProps   props默认值
  - getInitialState tate默认值

- 挂载阶段

- - `staticgetDerivedStateFromProps(props,state)`    ---->react16新增
  - `render`
  - `componentDidMount`
```

新属性介绍:getDerivedStateFromProps

`getDerivedStateFromProps`：组件每次被 `rerender`的时候，包括在组件构建之后(虚拟 `dom`之后，实际 `dom`挂载之前)，每次获取新的 `props`或 `state`之后；每次接收新的props之后都会返回一个对象作为新的 `state`，返回null则说明不需要更新 `state`；配合 `componentDidUpdate`，可以覆盖 `componentWillReceiveProps`的所有用法.

```
- 更新阶段

- - `staticgetDerivedStateFromProps(props,state)`
  - `shouldComponentUpdate`
  - `render`
  - `getSnapshotBeforeUpdate(prevProps,prevState)`
  - `componentDidUpdate`
```

新属性介绍:`getSnapshotBeforeUpdate`

> `getSnapshotBeforeUpdate`：触发时间: `update`发生的时候，在 `render`之后，在组件`dom`渲染之前；返回一个值，作为 `componentDidUpdate`的第三个参数；配合`componentDidUpdate`, 可以覆盖 `componentWillUpdate`的所有用法

```
- 卸载阶段

- - `componentWillUnmount`

- 错误处理

- - `componentDidCatch`
```

三.React如何提高性能

1.适当地使用shouldComponentUpdate生命周期方法。 它避免了子组件的不必要的渲染。 如果树中有100个组件，则不重新渲染整个组件树来提高应用程序性能。

```
1.渲染次数:
    render里面尽量减少新建变量和bind函数
2.shouldComponentUpdate
//在render函数调用前判断：如果前后state中Number不变，通过return false阻止render调用
shouldComponentUpdate(nextProps,nextState){
   if(nextState.Number == this.state.Number){
     return false
   }
}
```

2.在显示列表或表格时始终使用 Keys，这会让 React 的更新速度更快

4.多使用无状态组件(Function)

5.不可变性是提高性能的关键。不要对数据进行修改，而是始终在现有集合的基础上创建新的集合，以保持尽可能少的复制，从而提高性能。

四.概念理解:

```
**纯函数**：纯函数是始终接受一个或多个参数并计算参数然后返回数据或是函数的函数。

**高阶函数：**高阶函数就是将 函数作为参数 或 返回函数 的函数 或者都有

这些高阶函数可以操纵其他函数

**什么是函数式编程：**函数式编程是声明式编程的一部分（声明式编程（注重过程）命令式编程（注重结果））

函数式编程的几个部分：1 不可变性 2纯函数 3数据转换 4高阶函数 5递归 6组合

在react中，我们将功能划分为小型可重用的纯函数，我们必须将这些可重用的函数放在一起，最终使其成为产品。
```

五.组件

1.有状态组件： 这类组件可以通过setState()来改变组件的状态，并且可以使用生命周期函数

2.容器组件： 容器组件用来包含展示其它组件或其它容器组件，但里面从来都没有html。
3.高阶组件：其实和高阶函数的意思差不多。意思是将组件作为参数并生成另一个组件的组件。

4.受控和非受控组件
例如input option radio 他们的状态是不受react控制的 而是控件本身具有的 我们把这样的组件称为非受控组件

```
1. 保持着自己的状态
2.数据由 DOM 控制
3. Refs 用于获取其当前值
```

4.2受控组件

```
1. 没有维持自己的状态
2.数据由父组件控制
3. 通过 props 获取当前值，然后通过回调通知更改
```

六.什么是上下文?

如果层级特别深 那么如果通过props层层传递是明显不合理的。这时我们就需要用到context上下文 通过设置上下文 任意一个后代元素都可以直接取到上下文的内容 不需要层层传递

示例：使用 context, 我们可以避免通过中间元素传递 props：

```
// Context 可以让我们无须明确地传遍每一个组件，就能将值深入传递进组件树。
// 为当前的 theme 创建一个 context（“light”为默认值）。
const ThemeContext = React.createContext('light');

class App extends React.Component {
render() {
 // 使用一个 Provider 来将当前的 theme 传递给以下的组件树。
 // 无论多深，任何组件都能读取这个值。
 // 在这个例子中，我们将 “dark” 作为当前的值传递下去。
 return (
   <ThemeContext.Provider value="dark">
     <Toolbar />
   </ThemeContext.Provider>
 );
}
}

// 中间的组件再也不必指明往下传递 theme 了。
function Toolbar(props) {
return (
 <div>
   <ThemedButton />
 </div>
);
}

class ThemedButton extends React.Component {
// 指定 contextType 读取当前的 theme context。
// React 会往上找到最近的 theme Provider，然后使用它的值。
// 在这个例子中，当前的 theme 值为 “dark”。
static contextType = ThemeContext;
render() {
 return <Button theme={this.context} />;
}
}
```

api-1:`React.createContext`

```
const MyContext = React.createContext(defaultValue);
/*
创建一个 Context 对象。当 React 渲染一个订阅了这个 Context 对象的组件，这个组件会从组件树中离自身最近的那个匹配的 Provider 中读取到当前的 context 值。
*/

/*
只有当组件所处的树中没有匹配到 Provider 时，其 defaultValue 参数才会生效。这有助于在不使用 Provider 包装组件的情况下对组件进行测试。注意：将 undefined 传递给 Provider 的 value 时，消费组件的 defaultValue 不会生效。
*/
```

api-02:`Context.Provider`

```
<MyContext.Provider value={/* 某个值 */}>
/*
每个 Context 对象都会返回一个 Provider React 组件，它允许消费组件订阅 context 的变化。

Provider 接收一个 value 属性，传递给消费组件。一个 Provider 可以和多个消费组件有对应关系。多个 Provider 也可以嵌套使用，里层的会覆盖外层的数据。

当 Provider 的 value 值发生变化时，它内部的所有消费组件都会重新渲染。Provider 及其内部 consumer 组件都不受制于 shouldComponentUpdate 函数，因此当 consumer 组件在其祖先组件退出更新的情况下也能更新。

通过新旧值检测来确定变化，使用了与 Object.is 相同的算法。
*/
```

api-03:`Class.contextType`

挂载在 class 上的 `contextType` 属性会被重赋值为一个由 [`React.createContext()`](https://zh-hans.reactjs.org/docs/context.html#reactcreatecontext) 创建的 Context 对象。这能让你使用 `this.context` 来消费最近 Context 上的那个值。你可以在任何生命周期中访问到它，包括 render 函数中。

```
class MyClass extends React.Component {
componentDidMount() {
 let value = this.context;
 /* 在组件挂载完成后，使用 MyContext 组件的值来执行一些有副作用的操作 */
}
componentDidUpdate() {
 let value = this.context;
 /* ... */
}
componentWillUnmount() {
 let value = this.context;
 /* ... */
}
render() {
 let value = this.context;
 /* 基于 MyContext 组件的值进行渲染 */
}
}
MyClass.contextType = MyContext;
```

context 重点理解：

很多优秀的React组件都通过Context来完成自己的功能，

比如react-redux的`<Provider />`，就是通过`Context`提供一个全局态的`store`，拖拽组件react-dnd，通过`Context`在组件中分发DOM的Drag和Drop事件。

路由组件react-router通过`Context`管理路由状态等等。在React组件开发中，如果用好`Context`，可以让你的组件变得强大，而且灵活。

示例：

```
import React from 'react';
import ReactDOM from 'react-dom';

const ThemeContext = React.createContext({
background: 'red',
color: 'white'
});

/*
通过静态方法React.createContext()创建一个Context对象，这个Context对象包含两个组件，
<Provider />和<Consumer />。
*/

class App extends React.Component {
render () {
 return (
   <ThemeContext.Provider value={{background: 'green', color: 'white'}}>
     <Header />
   </ThemeContext.Provider>
 );
}
}
```

示例2：`<Provider />`的`value`相当于现在的`getChildContext()`。

```
class Header extends React.Component {
render () {
 return (
   <Title>Hello React Context API</Title>
 );
}
}

class Title extends React.Component {
render () {
 return (
   <ThemeContext.Consumer>
     {context => (
       <h1 style={{background: context.background, color: context.color}}>
         {this.props.children}
       </h1>
     )}
   </ThemeContext.Consumer>
 );
}
```

```
}
/*
<Consumer />的children必须是一个函数，通过函数的参数获取<Provider />提供的Context。
可见，Context的新API更加贴近React的风格。
*/
```

在我们平时的开发中，用到作用域或者上下文的场景是很常见，很自然，甚至是无感知的，然而，在React中使用`Context`并不是那么容易。父组件提供`Context`需要通过`childContextTypes`进行“声明”，子组件使用父组件的`Context`属性需要通过`contextTypes`进行“申请”，所以，我认为React的`Context`是一种**“带权限”的组件作用域**。

这种“带权限”的方式有何好处？就我个人的理解，首先是保持框架API的一致性，和`propTypes`一样，使用声明式编码风格。另外就是，**可以在一定程度上确保组件所提供的Context的可控性和影响范围**。

React App的组件是树状结构，一层一层延伸，父子组件是一对多的线性依赖。随意的使用`Context`其实会破坏这种依赖关系，导致组件之间一些不必要的额外依赖，降低组件的复用性，进而可能会影响到App的可维护性。

在我看来，**通过Context暴露数据或者API不是一种优雅的实践方案**，尽管react-redux是这么干的。因此需要一种机制，或者说约束，去降低不必要的影响。

通过`childContextTypes`和`contextTypes`这两个静态属性的约束，可以在一定程度保障，只有组件自身，或者是与组件相关的其他子组件才可以随心所欲的访问`Context`的属性，无论是数据还是函数。因为只有组件自身或者相关的子组件可以清楚它能访问`Context`哪些属性，而相对于那些与组件无关的其他组件，无论是内部或者外部的 ，由于不清楚父组件链上各父组件的`childContextTypes`“声明”了哪些`Context`属性，所以没法通过`contextTypes`“申请”相关的属性。所以我理解为，给组件的作用域`Context`“带权限”，可以在一定程度上确保`Context`的可控性和影响范围。

用Context作为共享数据的媒介

官方所提到`Context`可以用来进行跨组件的数据通信。而我，把它理解为，好比一座桥，作为一种作为媒介进行**数据共享**。数据共享可以分两类：**App级**与**组件级**。

1.App级的数据共享重点 redux和context

App根节点组件提供的`Context`对象可以看成是App级的全局作用域，所以，我们利用App根节点组件提供的`Context`对象创建一些App级的全局数据。现成的例子可以参考react-redux，以下是`<Provider />`组件源码的核心实现：

App的根组件用`<Provider />`组件包裹后，本质上就为App提供了一个全局的属性`store`，相当于在整个App范围内，共享`store`属性。当然，`<Provider />`组件也可以包裹在其他组件中，在组件级的全局范围内共享`store`。

```
export function createProvider(storeKey = 'store', subKey) {
    const subscriptionKey = subKey || `${storeKey}Subscription`

    class Provider extends Component {
        getChildContext() {
          return { [storeKey]: this[storeKey], [subscriptionKey]: null }
        }

        constructor(props, context) {
          super(props, context)
          this[storeKey] = props.store;
        }
```

```
        render() {
          return Children.only(this.props.children)
        }
    }

    // ......

    Provider.propTypes = {
        store: storeShape.isRequired,
        children: PropTypes.element.isRequired,
    }
    Provider.childContextTypes = {
        [storeKey]: storeShape.isRequired,
        [subscriptionKey]: subscriptionShape,
    }

    return Provider
}

export default createProvider()
```

2.组件级的数据共享

如果组件的功能不能单靠组件自身来完成，还需要依赖额外的子组件，那么可以利用`Context`构建一个由多个子组件组合的组件。例如，react-router。

react-router的`<Router />`自身并不能独立完成路由的操作和管理，因为导航链接和跳转的内容通常是分离的，因此还需要依赖`<Link />`和`<Route />`等子组件来一同完成路由的相关工作。为了让相关的子组件一同发挥作用，react-router的实现方案是利用`Context`在`<Router />`、`<Link />`以及`<Route />`这些相关的组件之间共享一个`router`，进而完成路由的统一操作和管理。

下面截取`<Router />`、`<Link />`以及`<Route />`这些相关的组件部分源码，以便更好的理解上述所说的。

```
// Router.js

/**
 * The public API for putting history on context.
 */
class Router extends React.Component {
  static propTypes = {
    history: PropTypes.object.isRequired,
    children: PropTypes.node
  };

  static contextTypes = {
    router: PropTypes.object
  };

  static childContextTypes = {
    router: PropTypes.object.isRequired
  };

  getChildContext() {
    return {
      router: {
```

```
        ...this.context.router,
        history: this.props.history,
        route: {
          location: this.props.history.location,
          match: this.state.match
        }
      }
    };
  }

  // ......

  componentWillMount() {
    const { children, history } = this.props;

    // ......

    this.unlisten = history.listen(() => {
      this.setState({
        match: this.computeMatch(history.location.pathname)
      });
    });
  }

  // ......
}
/*
尽管源码还有其他的逻辑，但<Router />的核心就是为子组件提供一个带有router属性的Context，同时监听history，一旦history发生变化，便通过setState()触发组件重新渲染。
*/
```

```
// Link.js

/**
 * The public API for rendering a history-aware <a>.
 */
class Link extends React.Component {

  // ......

  static contextTypes = {
    router: PropTypes.shape({
      history: PropTypes.shape({
        push: PropTypes.func.isRequired,
        replace: PropTypes.func.isRequired,
        createHref: PropTypes.func.isRequired
      }).isRequired
    }).isRequired
  };

  handleClick = event => {
    if (this.props.onClick) this.props.onClick(event);

    if (
      !event.defaultPrevented &&
      event.button === 0 &&
      !this.props.target &&
```

```
      !isModifiedEvent(event)
    ) {
      event.preventDefault();
      // 使用<Router />组件提供的router实例
      const { history } = this.context.router;
      const { replace, to } = this.props;

      if (replace) {
        history.replace(to);
      } else {
        history.push(to);
      }
    }
  };

  render() {
    const { replace, to, innerRef, ...props } = this.props;

    // ...

    const { history } = this.context.router;
    const location =
      typeof to === "string"
        ? createLocation(to, null, null, history.location)
        : to;

    const href = history.createHref(location);
    return (
      <a {...props} onClick={this.handleClick} href={href} ref={innerRef} />
    );
  }
}
```

`<Link />`的核心就是渲染`<a>`标签，拦截`<a>`标签的点击事件，然后通过`<Router />`共享的`router`对`history`进行路由操作，进而通知`<Router />`重新渲染。

```
// Route.js

/**
 * The public API for matching a single path and rendering.
 */
class Route extends React.Component {

  // ......

  state = {
    match: this.computeMatch(this.props, this.context.router)
  };

  // 计算匹配的路径，匹配的话，会返回一个匹配对象，否则返回null
  computeMatch(
    { computedMatch, location, path, strict, exact, sensitive },
    router
  ) {
    if (computedMatch) return computedMatch;

    // ......
```

```
    const { route } = router;
    const pathname = (location || route.location).pathname;

    return matchPath(pathname, { path, strict, exact, sensitive }, route.match);
  }

  // ......

  render() {
    const { match } = this.state;
    const { children, component, render } = this.props;
    const { history, route, staticContext } = this.context.router;
    const location = this.props.location || route.location;
    const props = { match, location, history, staticContext };

    if (component) return match ? React.createElement(component, props) : null;

    if (render) return match ? render(props) : null;

    if (typeof children === "function") return children(props);

    if (children && !isEmptyChildren(children))
      return React.Children.only(children);

    return null;
  }
}
```

`<Route />`有一部分源码与`<Router />`相似，可以实现路由的嵌套，但其核心是通过`Context`共享的`router`，判断是否匹配当前路由的路径，然后渲染组件。

通过上述的分析，可以看出，整个react-router其实就是围绕着`<Router />`的`Context`来构建的。

总结：

- 相比`props`和`state`，React的`Context`可以实现跨层级的组件通信。
- Context API的使用基于生产者消费者模式。生产者一方，通过组件静态属性`childContextTypes`声明，然后通过实例方法`getChildContext()`创建`Context`对象。消费者一方，通过组件静态属性`contextTypes`申请要用到的`Context`属性，然后通过实例的`context`访问`Context`的属性。
- 使用`Context`需要多一些思考，不建议在App中使用`Context`，但如果开发组件过程中可以确保组件的内聚性，可控可维护，不破坏组件树的依赖关系，影响范围小，可以考虑使用`Context`解决一些问题。
- 通过`Context`暴露API或许在一定程度上给解决一些问题带来便利，但个人认为不是一个很好的实践，需要慎重。
- 旧版本的`Context`的更新需要依赖`setState()`，是不可靠的，不过这个问题在新版的API中得以解决。
- 可以把`Context`当做组件的作用域来看待，但是需要关注`Context`的可控性和影响范围，使用之前，先分析是否真的有必要使用，避免过度使用所带来的一些副作用。
- 可以把`Context`当做媒介，进行App级或者组件级的数据共享。
- 设计开发一个组件，如果这个组件需要多个组件关联组合的，使用`Context`或许可以更加优雅。

11.2.Redux 的基础概念和如何使用redux：

实现原理：参考上面的context和redux的联系

通过react-redux做连接，使用Provider：从最外部封装了整个应用，并向connect模块传递store。
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

简单地说就是：
1.顶层分发状态，让React组件被动地渲染。
2.监听事件，事件有权利回到所有状态顶层影响状态。

1. Redux 的核心是一个 store。

store:首先要创建一个对象store，这个对象有各种方法，用来让外界获取Redux的数据（store.getState），或者让外界来修改Redux中的数据（store.dispatch）

```
import { createStore } from 'redux';
const store = createStore(reducer);
```

2.action:action 是一个 JavaScript 对象，通常包含了 type、payload 等字段，用于描述发生的事件及相关信息

描述我要干啥，一般是一个对象的形式，其中有一个type字段是必须要有的，比如：{ type：‘请求增援’ }，还可以带点数据{ type：‘请求增援’，gun:"100" }
那怎么去触发这个操作（action）呢，就好比说我一连长发现敌情了，我怎么报告给通讯班，让通讯班来处理呢？
dispatch() 接受一个 action 作为参数，将这个 action 分发给所有订阅了更新的 reducer。
store.dispatch({ type：‘请求增援’，gun:"100" })

这样就可以触发action，执行reducer，得到一个全新的state。

reducer：撸开袖子真刀实枪的就去干了，比如一连长要求增援，增援要求是100杆枪，团长马上就给你加了100杆枪送了过去。reducer 不是一个对象，而是一个返回更新后 state 的纯函数

```
const defaultState = 0;
const reducer = (state = defaultState, action) => {
  switch (action.type) {
    case '请求增援':
      return state + action.gun;
    default:
      return state;
  }
};
```

4.步骤1包装根组件，Provider是一个普通组件，可以作为顶层app的分发点，它只需要store属性就可以了。它会将state分发给所有被connect的组件，不管它在哪里，被嵌套多少层。

```
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import todoApp from './reducers'
import App from './components/App'

let store = createStore(todoApp);

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

步骤2：包装component

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
mapStateToProps,
mapDispatchToProps
)(TodoList)
//我们将：mapStateToProps和mapDispatchToProps 作为参数传给 connect,connect 会返回一个生成组件函数，然后将TodoList 组件当作参数传给这个函数
上面代码中，connect方法接受两个参数：mapStateToProps和mapDispatchToProps。它们定义了 UI 组件的业务逻辑。
前者负责输入逻辑，即将state映射到 UI 组件的参数（props），
后者负责输出逻辑，即将用户对 UI 组件的操作映射成 Action。
```

5.connect方法理解：connect(mapStateToProps, mapDispatchToProps)(MyComponent)

```
connect是真正的重点，它是一个科里化函数，意思是先接受两个参数（数据绑定mapStateToProps和事件绑定mapDispatchToProps），再接受一个参数（将要绑定的组件本身）：
**5.1.mapStateToProps**：构建好Redux系统的时候，它会被自动初始化，但是你的React组件并不知道它的存在，因此你需要分拣出你需要的Redux状态，所以你需要绑定一个函数，它的参数是state，简单返回你关心的几个值。`
```

```
mapStateToProps：字面含义是把state映射到props中去，意思就是把Redux中的数据映射到React中的props中去。
//mapStateToProps是一个函数，它接受`state`作为参数，返回一个对象。里面的每一个键值对就是一个映射。
const mapStateToProps = (state) => {
  return {
    gun: state.gunOfErlian
  }
}
// 这样在该组件就可以使用数据：
render(){
    return(
```

```
        <div>this.props.gun</div>
    )
}

//还可以使用第二个参数，代表容器组件的`props`对象。
// 容器组件的代码
//    <FilterLink filter="SHOW_ALL">
//      All
//    </FilterLink>
const mapStateToProps = (state, ownProps) => {
    return {
     active: ownProps.filter === state.visibilityFilter
    }
}
```
使用`ownProps`作为参数后，如果容器组件的参数发生变化，也会引发 UI 组件重新渲染。

`connect`方法可以省略`mapStateToProps`参数，那样的话，UI 组件就不会订阅Store，就是说 Store 的更新不会引起 UI 组件的更新。

5.2.mapDispatchToProps:

`mapDispatchToProps`是`connect`函数的第二个参数，用来建立 UI 组件的参数到`store.dispatch`方法的映射。也就是说，它定义了哪些用户的操作应该当作 Action，传给 Store。它可以是一个函数，也可以是一个对象。

5.21.如果`mapDispatchToProps`是一个函数，会得到`dispatch`和`ownProps`（容器组件的`props`对象）两个参数。

在组件中使用：mapDispatchToProps的函数返回值会合并到你的组件props中去。你就能够直接调用它们来分发action

```
function Counter({ count, increment, decrement, reset }) {
  return (
    <div>
      <button onClick={decrement}>-</button>
      <span>{count}</span>
      <button onClick={increment}>+</button>
      <button onClick={reset}>reset</button>
    </div>
  );
}
```

```
const mapDispatchToProps = (
dispatch,
ownProps
) => {
return {
 onClick: () => {
   dispatch({
     type: 'SET_VISIBILITY_FILTER',
     filter: ownProps.filter
   });
 }
};
}
```

```
//从上面代码可以看到，`mapDispatchToProps`作为函数，应该返回一个对象，该对象的每个键值对都是一个映射，定义了 UI 组件的参数怎样发出 Action。
```

dispatch示例2

`mapDispatchToProps`函数调用时以`dispatch`作为第一个参数。通常情况下，你会利用这个参数来返回一个内部调用了`dispatch()`的新函数，然后内部传递一个纯的action对象或者action创建函数的返回值。

dispatch示例2
函数形式：更高自由度、能够访问`dispatch`和可选择的`ownProps`

对象形式：更声明式，更易于使用

注意到：

- 每个`mapDispatchToProps`对象的字段都被假设为一个action创建函数
- 你的组件不再接收`dispatch`作为一个`prop`

```
如果传递的是一个对象，其内部的每一个函数都被假定为一个Redux action 创建函数。这个对象会作为`props`传递给连接组件，且其内部每个字段名都与action creators相同，但被一个能调用`dispatch`方法的函数包装，这样一来这些分发函数就可以直接调用。

如果传递的是一个函数，其第一个参数为`dispatch`。你想如何使用`dispatch`去绑定action创建函数都可以。（贴士：你可以使用Redux提供的`bindActionCreators()`方法）
```

```
// React-Redux 自动为你做:
dispatch => bindActionCreators(mapDispatchToProps, dispatch);
```

因此，我们的`mapDispatchToProps`可以简写为：

```
import {increment, decrement, reset} from "./counterActions";

const actionCreators = {
  increment,
  decrement,
  reset
}
export default connect(mapState, actionCreators)(Counter);
```

注意：我们建议使用对象形式的`mapDispatchToProps`，除非你需要以某种自定义形式进行分发操作

```
const mapDispatchToProps = dispatch => {
  return {
    // 分发纯action对象
    increment: () => dispatch({ type: "INCREMENT" }),
    decrement: () => dispatch({ type: "DECREMENT" }),
    reset: () => dispatch({ type: "RESET" })
  };
};

你也可能需要把一些参数转发给你的action创建函数
const mapDispatchToProps = dispatch => {
  return {
    // 直接转发参数
    onClick: event => dispatch(trackClick(event)),
```

```
    // 间接转发参数
    onReceiveImpressions: (...impressions) =>
      dispatch(trackImpressions(impressions))
  };
};
```

额外的示例：把dispatch()函数的action 抽离到actionCreator，比较常用:

```
import React, { Component } from 'react';
import actionCreator from "../store/number/actionCreator"; //需要actionCreator中的
action
import {connect} from 'react-redux';
class Number2 extends Component {
  constructor(){
    super();
    this.state = {
        val: "0"
    }
  }
  render() {
    console.log(this.props)
    return (
      <div>
        <button>++</button>
        <input type="text" />
        <button >--</button>
      </div>
    );
  }
}

let mapDispatchToProps = (dispatch)=>{
    return {
        add(){
            dispatch(actionCreator.intNumber());
        },
        sub(){
            dispatch(actionCreator.decNumber());
        },
        input(val){
            dispatch(actionCreator.inputNumber(val));
        }
    }
}
let Connected = connect(state=>state, mapDispatchToProps)(Number2);
export default Connected;
```

5.22.如果`mapDispatchToProps`是一个对象，它的每个键名也是对应 UI 组件的同名参数，键值应该是一个函数，会被当作 Action creator ，返回的 Action 会由 Redux 自动发出。举例来说，上面的`mapDispatchToProps`写成对象就是下面这样。**

我们建议适中使用这种"对象简写"形式的mapDispatchToProps，除非你有特殊理由需要自定义dispatching行为。

```
const mapDispatchToProps = {
    onClick: (filter) => {
     type: 'SET_VISIBILITY_FILTER',
     filter: filter
    };
}
```

当有多个 reducer 时，创建 store 之前需要将它们先进行合并

```
import { combineReducers } from 'redux';

// 合并成一个 reducers
const reducers = combineReducers({
    a: doSomethingWithA,
    b: processB,
    c: c
});
```

redux中间件的理解

中间件就是要对redux的store.dispatch方法做一些改造，以实现其他的功能。

背景：Redux 的基本做法，是用户发出 Action，Reducer 函数立刻算出新的 State，View 重新渲染，但这是做同步。

而如果有异步请求时，那就不能知道什么时候获取的数据有存进store里面，因此此时需要在请求成功时返回一个标识或状态，并在此时再触发action给reducer传值。
因此，为了解决异步的问题，就引入了中间件的概念。
作用：redux-thunk 帮助你统一了异步和同步 action 的调用方式，把异步过程放在 action 级别解决，对 component 调用没有影响。

React中的事件是什么?

在 React 中，事件是对鼠标悬停、鼠标单击、按键等特定操作的触发反应。处理这些事件类似于处理 DOM 元素中的事件。但是有一些语法差异，如：

1. 用驼峰命名法对事件命名而不是仅使用小写字母。
2. 事件作为函数而不是字符串传递。

事件参数重包含一组特定于事件的属性。每个事件类型都包含自己的属性和行为，只能通过其事件处理程序访问。

事件绑定this的三种方式

绑定事件处理函数this的几种方法：
第一种方法：

```
    run(){
alert(this.state.name)
    }
    <button onClick={this.run.bind(this)}>按钮</button>
```

第二种方法：

```
    构造函数中改变
    this.run = this.run.bind(this);
    run(){
    alert(this.state.name)
     }
    <button onClick={this.run>按钮</button>
第三种方法：
     run=()=> {
        alert(this.state.name)
     }
    <button onClick={this.run>按钮</button>
```

解释：

这些老铁都说的什么啊。。
来，如果直接写，你这样函数在render的时候已经执行了呀,肯定不行嘿,

```
onClick={this.handleClick(i)}所以说要这样
```

但是 这个函数 需要携带一个 i 的参数过去：

```
onClick = {this.handleClick}
```

所以 就要用一个匿名函数把i 带过去啊。。

```
//不用传递事件
onClick = {()=> this.handleClick(i)}
//如果要传递事件：
onClick={(event)=>this.showModal("modal1",event)}
```

这样闭包 让 i 对 renderSquare 的i 保持引用

bind this 是另外一个问题吧 就像有一位老铁说的 `xx = ()=> { 代码 }` this就指向 当前 class

react-router

参考网址：https://www.cnblogs.com/cckui/p/11490372.html
Q1:window.location对象有什么特点?
专门用来获取当前页面的地址的相关信息，还可以通过给这些地址属性赋值的方式使得当前页面跳转。
window.location.href：获取url，并且可以可以赋值给他新的url，让页面跳转。

Q2:window.history对象有什么特点呢?

```
对当前浏览器的url历史记录的管理。可想成：window.history是一个堆栈，里面存放了当前浏览器Tab的所有浏览url并依照浏览顺序存放在堆栈中。
可以通过调用对象的方法从url堆栈中push或者pop url出来。
window.history.pushState(null,null,url)就是向当前文档的url堆栈中push一个新的url。
-----> 结论：使用window.history.pushState(null,null,url)是不会使用参数url和当前的url拼接产生新的url跳转的。
```

Q4:**BrowserRouter：h5路由（history API）和 HashRouter：哈希路由 的区别：早期实现页面哈希，使用的是锚点技术；**

BrowserRouter 和 HashRouter 都可以实现前端路由的功能，区别是前者基于url的pathname段，后者基于hash段。

前者：http://127.0.0.1:3000/article/num1

后者：http://127.0.0.1:3000/#/article/num1（不一定是这样，但#是少不了的）

这样的区别带来的直接问题就是当处于二级或多级路由状态时，刷新页面，会将当前路由发送到服务器（因为是pathname），而不会（因为是hash段）。

Q5:**第一种跳转：**

一，组件中跳转到另一个路由组件：withRouter 可以将一个非路由组件包裹为路由组件，使这个非路由组件也能访问到当前路由的match，location，history对象。

从react-router-dom中导入withRouter方法
```
import { withRouter } from 'react-router-dom';
```
使用withRouter方法加工需要触发路由跳转的组件
```
export default withRouter(Login);
```
通过withRouter加工后的组件会多出一个history props，这时就可以通过history的push方法跳转路由了。
```
this.props.history.push('/home');
```

注意，只有通过 Route 组件渲染的组件，才能在 this.props 上找到 history 对象
所以，如果想在路由组件的子组件中使用 history ，需要使用 withRouter 包裹：

```
import React, { PureComponent } from 'react';
import { withRouter } from 'react-router-dom';
class 子组件 extends PureComponent {
 goHome = () => {
     this.props.history.push('/home')
 }
 render() {
     console.log(this.props)
     return (
         <div onClick={this.goHome}>子组件</div>
     )
 }
}
export default withRouter(子组件);
```

实战：

```
componentDidMount() {
console.log("路由：", this.props.history)
}

let newCommitDiv = connect(mapStateToProps, mapDispatchToProps)(commitDiv);
// export default newCommitDiv;
export default withRouter(newCommitDiv);
```

非组件JS函数中触发路由跳转

从history中导入createHashHistory方法（如果您的react应用使用的是history路由则导入createBrowserHistory）

```
import { createHashHistory } from 'history'; // 如果是hash路由
import { createBrowserHistory } from 'history'; // 如果是history路由
```

React-Router v4.0上已经不推荐使用hashRouter，主推browserRouter，但是因为使用browserRouter需要服务端配合可能造成不便，有时还是需要用到hashRouter。

- 创建history实例

```
const history = createHashHistory();
```

- 跳转路由
  history.push('/login')

Q6.传参方式
1.query方式使用很简单，类似于表单中的get方法，传递参数为明文：
首先定义路由：

```
//传：
hashHistory.push({
  pathname:'/user',
  query:{id:3,name:sam,age:36},
});
//获取数据：
let data = this.props.location.query;
let {id,name,age} = data;
```

2.state方式类似于post方式，使用方式和query类似

```
hashHistory.push({
  pathname:'/user',
  state:{id:3,name:sam,age:36},
})
let data = this.props.location.state;
let {id,name,age} = data;
```

3.props.params

```
 console.log("搜索组装后数据：", queryData)

 const path = {
   pathname: '/result',
   query: queryData,
 }
 this.props.history.push(path);
或：
```

14.你对 React 的 refs 有什么了解?

```
Refs 是 React 中引用的简写。它是一个有助于存储对特定的 React 元素或组件的引用的属性，它将由组件渲染配置函数返回。用于对 render() 返回的特定元素或组件的引用。当需要进行 DOM 测量或向组件添加方法时，它们会派上用场
```

```
class ReferenceDemo extends React.Component{
  display() {
      const name = this.inputDemo.value;
      document.getElementById('disp').innerHTML = name;
  }
render() {
 return(
       <div>
         Name: <input type="text" ref={input => this.inputDemo = input} />
         <button name="Click" onClick={this.display}>Click</button>
         <h2>Hello <span id="disp"></span> !!!</h2>
       </div>
 );
}
}
```

列出一些应该使用 Refs 的情况:

以下是应该使用 refs 的情况：

- 需要管理焦点、选择文本或媒体播放时
- 触发式动画
- 与第三方 DOM 库集成

React 中 key 的重要性是什么?
key 用于识别唯一的 Virtual DOM 元素及其驱动 UI 的相应数据。它们通过回收 DOM 中当前所有的元素来帮助 React 优化渲染。这些 key 必须是唯一的数字或字符串，React 只是重新排序元素而不是重新渲染它们。这可以提高应用程序的性能。

setState是同步的还是异步的?

既可能是同步的，也可能是异步的。 准确地说，在React内部机制能检测到的地方， setState就是异步的；在React检测不到的地方，例如setInterval,setTimeout里，setState就是同步更新的。

生命周期和合成事件中

在 `React` 的生命周期和合成事件中， `React` 仍然处于他的更新机制中，这时无论调用多少次 `setState` ，都会不会立即执行更新，而是将要更新的·存入 `_pendingStateQueue` ，将要更新的组件存入 `dirtyComponent` 。
当上一次更新机制执行完毕，以生命周期为例，所有组件，即最顶层组件 `didmount` 后会将批处理标志设置为 `false` 。这时将取出 `dirtyComponent` 中的组件以及 `_pendingStateQueue` 中的 `state` 进行更新。这样就可以确保组件不会被重新渲染多次。

```
componentDidMount() {
this.setState({
   index: this.state.index + 1
})
 console.log('state', this.state.index);
}
/*
所以，如上面的代码，当我们在执行 `setState`后立即去获取 `state`，这时是获取不到更新后的`state`的，因为处于 `React`的批处理机制中， `state`被暂存起来，待批处理机制完成之后，统一进行更新。
所以。`setState`本身并不是异步的，而是 `React`的批处理机制给人一种异步的假象。
*/
```

异步代码和原生事件中

```
componentDidMount() {
 setTimeout(() => {
   console.log('调用setState');
this.setState({
     index: this.state.index + 1
})
   console.log('state', this.state.index);
}, 0);
}
/*
如上面的代码，当我们在异步代码中调用 `setState`时，根据 `JavaScript`的异步机制，会将异步代码先暂存，等所有同步代码执行完毕后在执行，这时 `React`的批处理机制已经走完，处理标志设被设置为`false`，这时再调用 `setState`即可立即执行更新，拿到更新后的结果。
在原生事件中调用 `setState`并不会出发 `React`的批处理机制，所以立即能拿到最新结果。
*/
```

最佳实践

`setState`的第二个参数接收一个函数，该函数会在`React`的批处理机制完成之后调用，所以你想在调用`setState`后立即获取更新后的值，请在该回调函数中获取。

```
this.setState({ index: this.state.index + 1 }, () => {
   console.log(this.state.index);
})
```

原生事件和React事件的区别?

- `React`事件使用驼峰命名，而不是全部小写。
- 通过`JSX`，你传递一个函数作为事件处理程序，而不是一个字符串。
- 在`React`中你不能通过返回`false`来阻止默认行为。必须明确调用`preventDefault`。

React和原生事件的执行顺序是什么？可以混用吗?

`React`的所有事件都通过 `document`进行统一分发。当真实 `Dom`触发事件后冒泡到 `document`后才会对 `React`事件进行处理。
所以原生的事件会先执行，然后执行 `React`合成事件，最后执行真正在 `document`上挂载的事件
`React`事件和原生事件最好不要混用。原生事件中如果执行了 `stopPropagation`方法，则会导致其他`React`事件失效。因为所有元素的事件将无法冒泡到 `document`上，导致所有的 `React`事件都将无法被触发。。