第三版：React 45 道

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。此外 React 还需要借助 Key 值来判断元素与本地状态的关联关系因此我们绝不可忽视转换函数中 Key 的重要性

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

```
this.setState(
  { username: 'tylermcginnis33' },
  () => console.log('setState has finished and the component has re-rendered.')
)
this.setState((prevState, props) => {
  return {
    streak: prevState.streak + props.count
  }
})
```

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 树中的句柄该值会作为回调函数的第一个参数返回

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 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

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

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

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

redux 中间件

中间件提供第三方插件的模式自定义拦截 action -> reducer 的过程。变为 action -> middlewares -> reducer。这种机制可以让我们改变数据流实现如异步 action action 过滤日志输出异常报告等功能

1、 redux-logger 提供日志输出

2、 redux-thunk 处理异步操作

3、 redux-promise 处理异步操作 actionCreator 的返回值是 promise

redux 有什么缺点

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。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。

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

react 旧版生命周期函数

初始化阶段

1、 getDefaultProps:获取实例的默认属性

2、 getInitialState:获取每个实例的初始化状态

3、 componentWillMount 组件即将被装载、渲染到页面上

4、 render:组件在这里生成虚拟的 DOM 节点

5、 componentDidMount:组件真正在被装载之后

运行中状态

1、 componentWillReceiveProps:组件将要接收到属性的时候调用

2、 shouldComponentUpdate:组件接受到新属性或者新状态的时候可以返回 false 接收数据后不更新阻止 render 调用后面的函数不会被继续执行了

3、 componentWillUpdate:组件即将更新不能修改属性和状态

4、 render:组件重新描绘

5、 componentDidUpdate:组件已经更新

销毁阶段

componentWillUnmount:组件即将销毁新版生命周期

在新版本中 React 官方对生命周期有了新的 变动建议:

1、 使用 getDerivedStateFromProps 替换 componentWillMount

2、 使用 getSnapshotBeforeUpdate 替换 componentWillUpdate

3、 避免使用 componentWillReceiveProps

其实该变动的原因正是由于上述提到的 Fiber。首先从上面我们知道 React 可以分成 reconciliation 与 commit 两个阶段对应的生命周期如下：

1、 reconciliation

2、 componentWillMount

3、 componentWillReceiveProps

4、 shouldComponentUpdate

5、 componentWillUpdate

6、 commit

7、 componentDidMoun

8、 componentDidUpdate

9、 componentWillUnmount

在 Fiber 中 reconciliation 阶段进行了任务分割涉及到 暂停 和 重启因此可能会导致 reconciliation 中的生命周期函数在一次更新渲染循环中被 多次调用 的情况产生一些意外错误

新版的建议生命周期如下：

```
class Component extends React.Component {
  // 替换 `componentWillReceiveProps`
  // 初始化和 update 时被调用
```

```
  // 静态函数无法使用 this
  static getDerivedStateFromProps(nextProps, prevState) {}

  // 判断是否需要更新组件
  // 可以用于组件性能优化
  shouldComponentUpdate(nextProps, nextState) {}

  // 组件被挂载后触发
  componentDidMount() {}

  // 替换 componentWillUpdate
  // 可以在更新之前获取最新 dom 数据
  getSnapshotBeforeUpdate() {}

  // 组件更新后调用
  componentDidUpdate() {}

  // 组件即将销毁
  componentWillUnmount() {}

  // 组件已销毁
  componentDidUnMount() {}
}
```


使用建议：

1、 在 constructor 初始化 state

2、 在 componentDidMount 中进行事件监听并在 componentWillUnmount 中解绑事件

3、 在 componentDidMount 中进行数据的请求而不是在 componentWillMount

4、 需要根据 props 更新 state 时使用 getDerivedStateFromProps(nextProps, prevState)

5、 旧 props 需要自己存储以便比较

```
public static getDerivedStateFromProps(nextProps, prevState) {
 // 当新 props 中的 data 发生变化时同步更新到 state 上
 if (nextProps.data !== prevState.data) {
  return {
   data: nextProps.data
  }
 } else {
  return null1
 }
}
```

可以在 componentDidUpdate 监听 props 或者 state 的变化例如：

```
componentDidUpdate(prevProps) {
 // 当 id 发生变化时重新获取数据
 if (this.props.id !== prevProps.id) {
  this.fetchData(this.props.id);
 }
}
```

在 componentDidUpdate 使用 setState`时必须加条件否则将进入死循环

shouldComponentUpdate: 默认每次调用 setState 一定会最终走到 diff 阶段但可以通过 shouldComponentUpdate 的生命钩子返回 false 来直接阻止后面的逻辑执行通常是用于做条件渲染优化渲染的性能。

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

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

1、 用 JavaScript 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。然后用新的树和旧的树进行比较记录两棵树差异

3、 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

diff 算法?

1、 把树形结构按照层级分解只比较同级元素。

2、 给列表结构的每个单元添加唯一的 key 属性方便比较。

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

4、 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.

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

react 性能优化方案

1、 重写 shouldComponentUpdate 来避免不必要的 dom 操作

2、 使用 production 版本的 react.js

3、 使用 key 来帮助 React 识别列表中所有子组件的最小变化

简述 flux 思想

1、 Flux 的最大特点就是数据的"单向流动"。

2、 用户访问 View

3、 View 发出用户的 Action

4、 Dispatcher 收到 Action 要求 Store 进行相应的更新

5、 Store 更新后发出一个"change"事件

6、 View 收到"change"事件后更新页面

说说你用 react 有什么坑点

1、 JSX 做表达式判断时候需要强转为 boolean 类型

如果不使用 !!b 进行强转数据类型会在页面里面输出 0。

```
render() {
  const b = 0;
  return <div>
    {
      !!b && <div>这是一段文本</div>
    }
  </div>
}
```

1、 尽量不要在 componentWillRevicеProps 里使用 setState 如果一定要使用那么需要判断结束条件不然会出现无限重渲染导致页面崩溃

2、 给组件添加 ref 时候尽量不要使用匿名函数因为当组件更新的时候匿名函数会被当做新的 prop 处理让 ref 属性接受到新函数的时候 react 内部会先清空 ref

也就是会以 null 为回调参数先执行一次 ref 这个 props 然后在以该组件的实例执行一次 ref 所以用匿名函数做 ref 的时候有的时候去 ref 赋值后的属性会取到 null

3、 遍历子节点的时候不要用 index 作为组件的 key 进行传入

我现在有一个 button 要用 react 在上面绑定点击事件要怎么做

```
class Demo {
  render() {
    return <button onClick={(e) => {
      alert('我点击了按钮')
    }}>
      按钮
    </button>
  }
}
```

你觉得你这样设置点击事件会有什么问题吗

由于 onClick 使用的是匿名函数所有每次重渲染的时候会把该 onClick 当做一个新的 prop 来处理会将内部缓存的 onClick 事件进行重新赋值所以相对直接使用函数来说可能有一点的性能下降

修改

```
class Demo {

  onClick = (e) => {
    alert('我点击了按钮')
```

```
}

render() {
  return <button onClick={this.onClick}>
    按钮
  </button>
}
```

react 的虚拟 dom 是怎么实现的

首先说说为什么要使用 Virturl DOM 因为操作真实 DOM 的耗费的性能代价太高所以 react 内部使用 js 实现了一套 dom 结构在每次操作在和真实 dom 之前使用实现好的 diff 算法对虚拟 dom 进行比较递归找出有变化的 dom 节点然后对其进行更新操作。为了实现虚拟 DOM 我们需要把每一种节点类型抽象成对象每一种节点类型有自己的属性也就是 prop 每次进行 diff 的时候 react 会先比较该节点类型假如节点类型不一样那么 react 会直接删除该节点然后直接创建新的节点插入到其中假如节点类型一样那么会比较 prop 是否有更新假如有 prop 不一样那么 react 会判定该节点有更新那么重渲染该节点然后在对其子节点进行比较一层一层往下直到没有子节点

react 的渲染过程中兄弟节点之间是怎么处理的也就是 key 值不一样的时候

通常我们输出节点的时候都是 map 一个数组然后返回一个 ReactNode 为了方便 react 内部进行优化我们必须给每一个 reactNode 添加 key 这个 key prop 在设计值处不是给开发者用的而是给 react 用的大概的作用就是给每一个 reactNode 添加一个身份标识方便 react 进行识别在重渲染过程中如果 key 一样

若组件属性有所变化则 react 只更新组件对应的属性没有变化则不更新如果 key 不一样则 react 先销毁该组件然后重新创建该组件

react-router 里的标签和<a>标签有什么区别

对比<a>,Link 组件避免了不必要的重渲染

connect 原理

首先 connect 之所以会成功是因为 Provider 组件在原应用组件上包裹一层使原来整个应用成为 Provider 的子组件接收 Redux 的 store 作为 props 通过 context 对象传递给子孙组件上的 connectconnect 做了些什么。它真正连接 Redux 和 React 它包在我们的容器组件的外一层它接收上面 Provider 提供的 store 里面的 state 和 dispatch 传给一个构造函数返回一个对象以属性形式传给我们的容器组件

connect 是一个高阶函数首先传入 mapStateToProps、mapDispatchToProps 然后返回一个生产 Component 的函数(wrapWithConnect)然后再将真正的 Component 作为参数传入 wrapWithConnect 这样就生产出一个经过包裹的 Connect 组件该组件具有如下特点

通过 props.store 获取祖先 Component 的 store props 包括 stateProps、dispatchProps、parentProps,合并在一起得到 nextState 作为 props 传给真正的 Component componentDidMount 时添加事件 this.store.subscribe(this.handleChange)实现页面交互 shouldComponentUpdate 时判断是否有避免进行渲染提升页面性能并得到 nextState componentWillUnmount 时移除注册的事件 this.handleChange

由于 connect 的源码过长我们只看主要逻辑

```
export default function connect(mapStateToProps, mapDispatchToProps,
mergeProps, options = {}) {
  return function wrapWithConnect(WrappedComponent) {
    class Connect extends Component {
      constructor(props, context) {
        // 从祖先 Component 处获得 store
        this.store = props.store || context.store
        this.stateProps = computeStateProps(this.store, props)
        this.dispatchProps = computeDispatchProps(this.store, props)
        this.state = { storeState: null }
        // 对 stateProps、dispatchProps、parentProps 进行合并
        this.updateState()
      }
      shouldComponentUpdate(nextProps, nextState) {
        // 进行判断当数据发生改变时 Component 重新渲染
        if (propsChanged
        || mapStateProducedChange
        || dispatchPropsChanged) {
          this.updateState(nextProps)
            return true
          }
        }
        componentDidMount() {
          // 改变 Component 的 state
          this.store.subscribe(() => {
            this.setState({
              storeState: this.store.getState()
            })
          })
        }
```

```
        render() {
          // 生成包裹组件 Connect
          return (
            <WrappedComponent {...this.nextState} />
          )
        }
      }
      Connect.contextTypes = {
        store: storeShape
      }
      return Connect;
    }
  }
```

Redux 实现原理解析

为什么要用 redux

在 React 中数据在组件中是单向流动的数据从一个方向父组件流向子组件通过 props,所以两个非父子组件之间通信就相对麻烦 redux 的出现就是为了解决 state 里面的数据问题

Redux 设计理念

Redux 是将整个应用状态存储到一个地方上称为 store,里面保存着一个状态树 store tree,组件可以派发(dispatch)行为(action)给 store,而不是直接通知其他组件组件内部通过订阅 store 中的状态 state 来刷新自己的视图

image

Redux 三大原则

唯一数据源

整个应用的 state 都被存储到一个状态树里面并且这个状态树只存在于唯一的 store 中

保持只读状态

state 是只读的唯一改变 state 的方法就是触发 actionaction 是一个用于描述以发生时间的普通对象

数据改变只能通过纯函数来执行

使用纯函数来执行修改为了描述 action 如何改变 state 的你需要编写 reducers

Redux 源码

```
let createStore = (reducer) => {
    let state;
    //获取状态对象
    //存放所有的监听函数
    let listeners = [];
    let getState = () => state;
    //提供一个方法供外部调用派发 action
    let dispath = (action) => {
```

```
            //调用管理员 reducer 得到新的 state
            state = reducer(state, action);
            //执行所有的监听函数
            listeners.forEach((l) => l())
        }
        //订阅状态变化事件当状态改变发生之后执行监听函数
        let subscribe = (listener) => {
            listeners.push(listener);
        }
        dispath();
        return {
            getState,
            dispath,
            subscribe
        }
}
let combineReducers=(renducers)=>{
        //传入一个 renducers 管理组返回的是一个 renducer
        return function(state={},action={}){
            let newState={};
            for(var attr in renducers){
                newState[attr]=renducers[attr](state[attr],action)

            }
            return newState;
        }
}
export {createStore,combineReducers};
```

pureComponent 和 FunctionComponent 区别

PureComponent 和 Component 完全相同但是在 shouldComponentUpdate 实现中 PureComponent 使用了 props 和 state 的浅比较。主要作用是用来提高某些特定场景的性能

react hooks 它带来了那些便利

1、 代码逻辑聚合逻辑复用

2、 HOC 嵌套地狱

3、 代替 class

4、 React 中通常使用 类定义 或者 函数定义 创建组件:

在类定义中我们可以使用到许多 React 特性例如 state、 各种组件生命周期钩子等但是在函数定义中我们却无能为力因此 React 16.8 版本推出了一个新功能 (React Hooks)通过它可以更好的在函数定义组件中使用 React 特性。

好处:

1、 跨组件复用: 其实 render props / HOC 也是为了复用相比于它们 Hooks 作为官方的底层 API 最为轻量而且改造成本小不会影响原来的* 组件层次结构和传说中的嵌套地狱

2、 类定义更为复杂

3、 不同的生命周期会使逻辑变得分散且混乱不易维护和管理

4、 时刻需要关注 this 的指向问题

5、 代码复用代价高高阶组件的使用经常会使整个组件树变得臃肿

6、 状态与 UI 隔离：正是由于 Hooks 的特性状态逻辑会变成更小的粒度并且极容易被抽象成一个自定义 Hooks 组件中的状态和 UI 变得更为清晰和隔离。

注意:

避免在 循环/条件判断/嵌套函数 中调用 hooks 保证调用顺序的稳定只有 函数定义组件 和 hooks 可以调用 hooks 避免在 类组件 或者 普通函数 中调用 不能在 useEffect 中使用 useStateReact 会报错提示类组件不会被替换或废弃 不需要强制改造类组件两种方式能并存重要钩子

状态钩子 (useState)：用于定义组件的 State 其到类定义中 this.state 的功能

```
// useState 只接受一个参数: 初始状态
// 返回的是组件名和更改该组件对应的函数
const [flag, setFlag] = useState(true);
// 修改状态
setFlag(false)

// 上面的代码映射到类定义中:
this.state = {
 flag: true
}
const flag = this.state.flag
const setFlag = (bool) => {
    this.setState({
        flag: bool,
    })
}
```

生命周期钩子（useEffect):

类定义中有许多生命周期函数而在 React Hooks 中也提供了一个相应的函数(useEffect)这里可以看做 componentDidMount、componentDidUpdate 和 componentWillUnmount 的结合。

1、 useEffect(callback, [source])接受两个参数

2、 callback: 钩子回调函数

3、 source: 设置触发条件仅当 source 发生改变时才会触发

4、 useEffect 钩子在没有传入[source]参数时默认在每次 render 时都会优先调用上次保存的回调中返回的函数后再重新调用回调

```
useEffect(() => {
 // 组件挂载后执行事件绑定
 console.log('on')
 addEventListener()
 // 组件 update 时会执行事件解绑
 return () => {
  console.log('off')
  removeEventListener()
 }
}, [source]);


// 每次 source 发生改变时执行结果(以类定义的生命周期便于大家理解):
// --- DidMount ---
```

```
// 'on'
// --- DidUpdate ---
// 'off'
// 'on'
// --- DidUpdate ---
// 'off'
// 'on'
// --- WillUnmount ---
// 'off'
```

通过第二个参数我们便可模拟出几个常用的生命周期：

1、 componentDidMount: 传入[]时就只会在初始化时调用一次

2、 const useMount = (fn) => useEffect(fn, [])

3、 componentWillUnmount: 传入[]回调中的返回的函数也只会被最终执行一次

4、 const useUnmount = (fn) => useEffect(() => fn, [])

5、 mounted: 可以使用 useState 封装成一个高度可复用的 mounted 状态

```
const useMounted = () => {
    const [mounted, setMounted] = useState(false);
    useEffect(() => {
        !mounted && setMounted(true);
        return () => setMounted(false);
    }, []);
    return mounted;
```

```
}
componentDidUpdate: useEffect每次均会执行其实就是排除了 DidMount 后即可
const mounted = useMounted()
useEffect(() => {
    mounted && fn()
})
```

其它内置钩子：

1、 useContext: 获取 context 对象

2、 useReducer: 类似于 Redux 思想的实现但其并不足以替代 Redux 可以理解成一个组件内部的 redux:并不是持久化存储会随着组件被销毁而销毁

3、 属于组件内部各个组件是相互隔离的单纯用它并无法共享数据

4、 配合 useContext 的全局性可以完成一个轻量级的 Redux(easy-peasy)

5、 useCallback: 缓存回调函数避免传入的回调每次都是新的函数实例而导致依赖组件重新渲染具有性能优化的效果

6、 useMemo: 用于缓存传入的 props 避免依赖的组件每次都重新渲染

7、 useRef: 获取组件的真实节点

8、 useLayoutEffect

9、 DOM 更新同步钩子。用法与 useEffect 类似只是区别于执行时间点的不同

10、 useEffect 属于异步执行并不会等待 DOM 真正渲染后执行而 useLayoutEffect 则会真正渲染后才触发

11、 可以获取更新后的 state

12、 自定义钩子(useXxxx): 基于 Hooks 可以引用其它 Hooks 这个特性我们可以编写自定义钩子如上面的 useMounted。又例如我们需要每个页面自定义标题:

```
function useTitle(title) {
  useEffect(
    () => {
      document.title = title;
    });
}

// 使用:
function Home() {
 const title = '我是首页'
 useTitle(title)

 return (
  <div>{title}</div>
 )
}
```

React Portal 有哪些使用场景

在以前 react 中所有的组件都会位于 #app 下而使用 Portals 提供了一种脱离 #app 的组件因此 Portals 适合脱离文档流(out of flow) 的组件特别是

position: absolute 与 position: fixed 的组件。比如模态框通知警告 goTop 等。

以下是官方一个模态框的示例可以在以下地址中测试效果

```
<html>
  <body>
    <div id="app"></div>
    <div id="modal"></div>
    <div id="gotop"></div>
    <div id="alert"></div>
  </body>
</html>
```

```
const modalRoot = document.getElementById('modal');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
```

```
    return ReactDOM.createPortal(
      this.props.children,
      this.el,
    );
  }
}
```

React Hooks 当中的 useEffect 是如何区分生命周期钩子的

useEffect 可以看成是 componentDidMountcomponentDidUpdate 和 componentWillUnmount 三者的结合。useEffect(callback, [source])接收两个参数调用方式如下

```
useEffect(() => {
  console.log('mounted');

  return () => {
      console.log('willUnmount');
  }
}, [source]);
```

生命周期函数的调用主要是通过第二个参数[source]来进行控制有如下几种情况

1、 [source]参数不传时则每次都会优先调用上次保存的函数中返回的那个函数然后再调用外部那个函数

2、 [source]参数传[]时则外部的函数只会在初始化时调用一次返回的那个函数也只会最终在组件卸载时调用一次

3、 [source]参数有值时则只会监听到数组中的值发生变化后才优先调用返回的那个函数再调用外部的函数。

react 和 vue 的区别

相同点

1、 数据驱动页面提供响应式的试图组件

2、 都有 virtual DOM,组件化的开发通过 props 参数进行父子之间组件传递数据都实现了 webComponents 规范

3、 数据流动单向都支持服务器的渲染 SSR

4、 都有支持 native 的方法 react 有 React native vue 有 wexx

不同点

1、 数据绑定 Vue 实现了双向的数据绑定 react 数据流动是单向的

2、 数据渲染大规模的数据渲染 react 更快

3、 使用场景 React 配合 Redux 架构适合大规模多人协作复杂项目 Vue 适合小快的项目

4、 开发风格 react 推荐做法 jsx + inline style 把 html 和 css 都写在 js 了

5、 vue 是采用 webpack + vue-loader 单文件组件格式 html, js, css 同一个文件

什么是高阶组件(HOC)

高阶组件(Higher Order Componennt)本身其实不是组件而是一个函数这个函数接收一个元组件作为参数然后返回一个新的增强组件高阶组件的出现本身也是为了逻辑复用举个例子

```
function withLoginAuth(WrappedComponent) {
  return class extends React.Component {

      constructor(props) {
          super(props);
          this.state = {
            isLogin: false
          };
      }

      async componentDidMount() {
          const isLogin = await getLoginStatus();
          this.setState({ isLogin });
      }

      render() {
        if (this.state.isLogin) {
            return <WrappedComponent {...this.props} />;
        }

        return (<div>您还未登录...</div>);
      }
  }
}
```

React 实现的移动应用中如果出现卡顿有哪些可以考虑的优化方案

1、 增加 shouldComponentUpdate 钩子对新旧 props 进行比较如果值相同则阻止更新避免不必要的渲染或者使用 PureReactComponent 替代 Component 其内部已经封装了 shouldComponentUpdate 的浅比较逻辑

2、 对于列表或其他结构相同的节点为其中的每一项增加唯一 key 属性以方便 React 的 diff 算法中对该节点的复用减少节点的创建和删除操作

render 函数中减少类似

```
onClick={() => {
    doSomething()
}}
```

的写法每次调用 render 函数时均会创建一个新的函数即使内容没有发生任何变化也会导致节点没必要的重渲染建议将函数保存在组件的成员对象中这样只会创建一次

1、 组件的 props 如果需要经过一系列运算后才能拿到最终结果则可以考虑使用 reselect 库对结果进行缓存如果 props 值未发生变化则结果直接从缓存中拿避免高昂的运算代价

2、 webpack-bundle-analyzer 分析当前页面的依赖包是否存在不合理性如果存在找到优化点并进行优化

setState

在了解 setState 之前我们先来简单了解下 React 一个包装结构: Transaction:

事务（Transaction)

是 React 中的一个调用结构用于包装一个方法结构为: initialize - perform(method) - close。通过事务可以统一管理一个方法的开始与结束处于事务流中表示进程正在执行一些操作

setState: React 中用于修改状态更新视图。它具有以下特点:

异步与同步: setState 并不是单纯的异步或同步这其实与调用时的环境相关:

在合成事件 和 生命周期钩子(除 componentDidUpdate) 中 setState 是"异步"的

原因: 因为在 setState 的实现中有一个判断: 当更新策略正在事务流的执行中时该组件更新会被推入 dirtyComponents 队列中等待执行否则开始执行 batchedUpdates 队列更新

在生命周期钩子调用中更新策略都处于更新之前组件仍处于事务流中而 componentDidUpdate 是在更新之后此时组件已经不在事务流中了因此则会同步执行

在合成事件中 React 是基于 事务流完成的事件委托机制 实现也是处于事务流中

问题：无法在 setState 后马上从 this.state 上获取更新后的值。

解决：如果需要马上同步去获取新值 setState 其实是可以传入第二个参数的。setState(updater, callback)在回调中即可获取最新值#

在 原生事件 和 setTimeout 中 setState 是同步的可以马上获取更新后的值

原因：原生事件是浏览器本身的实现与事务流无关自然是同步而 setTimeout 是放置于定时器线程中延后执行此时事务流已结束因此也是同步#

批量更新：在 合成事件 和 生命周期钩子 中 setState 更新队列时存储的是 合并状态(Object.assign)。因此前面设置的 key 值会被后面所覆盖最终只会执行一次更新

函数式：由于 Fiber 及 合并 的问题官方推荐可以传入 函数 的形式。setState(fn)在 fn 中返回新的 state 对象即可例如

```
this.setState((state, props) => newState)
```

使用函数式可以用于避免 setState 的批量更新的逻辑传入的函数将会被 顺序调用

注意事项：

1、 setState 合并在 合成事件 和 生命周期钩子 中多次连续调用会被优化为一次

2、 当组件已被销毁如果再次调用 setStateReact 会报错警告通常有两种解决办法

3、 将数据挂载到外部通过 props 传入如放到 Redux 或 父级中

4、 在组件内部维护一个状态量（isUnmounted）componentWillUnmount 中标记为 true 在 setState 前进行判断

HOC(高阶组件)

HOC(Higher Order Componennt) 是在 React 机制下社区形成的一种组件模式在很多第三方开源库中表现强大。

简述:

1、 高阶组件不是组件是 增强函数可以输入一个元组件返回出一个新的增强组件

2、 高阶组件的主要作用是 代码复用操作 状态和参数用法:

属性代理 (Props Proxy): 返回出一个组件它基于被包裹组件进行 功能增强默认参数: 可以为组件包裹一层默认参数

```
function proxyHoc(Comp) {
 return class extends React.Component {
  render() {
   const newProps = {
    name: 'tayde',
    age: 1,
   }
   return <Comp {...this.props} {...newProps} />
  }
 }
}
```

提取状态：可以通过 props 将被包裹组件中的 state 依赖外层例如用于转换受控组件：

```
function withOnChange(Comp) {
 return class extends React.Component {
  constructor(props) {
   super(props)
   this.state = {
    name: '',
   }
  }
  onChangeName = () => {
   this.setState({
    name: 'dongdong',
   })
  }
  render() {
   const newProps = {
    value: this.state.name,
    onChange: this.onChangeName,
   }
   return <Comp {...this.props} {...newProps} />
  }
 }
}
```

使用姿势如下这样就能非常快速的将一个 Input 组件转化成受控组件。

```
const NameInput = props => (<input name="name" {...props} />)
export default withOnChange(NameInput)
```

包裹组件：可以为被包裹元素进行一层包装

```
function withMask(Comp) {
  return class extends React.Component {
   render() {
   return (
     <div>
     <Comp {...this.props}/>
     <div style={{
       width: '100%',
       height: '100%',
       backgroundColor: 'rgba(0, 0, 0, .6)',
     }}
     </div>
   )
   }
  }
}
```

反向继承（Inheritance Inversion）：返回出一个组件继承于被包裹组件常用于以下操作

```
function IIHoc(Comp) {
    return class extends Comp {
        render() {
            return super.render();
        }
    };
}
```

渲染劫持（Render Highjacking）

条件渲染：根据条件渲染不同的组件

```
function withLoading(Comp) {
  return class extends Comp {
    render() {
      if(this.props.isLoading) {
          return <Loading />
      } else {
          return super.render()
      }
    }
  };
}
```

可以直接修改被包裹组件渲染出的 React 元素树

操作状态（Operate State）：可以直接通过 this.state 获取到被包裹组件的状态并进行操作。但这样的操作容易使 state 变得难以追踪不易维护谨慎使用。

应用场景：

权限控制通过抽象逻辑统一对页面进行权限判断按不同的条件进行页面渲染：

```
function withAdminAuth(WrappedComponent) {
  return class extends React.Component {
   constructor(props){
    super(props)
    this.state = {
        isAdmin: false,
```

```
    }
  }
  async componentWillMount() {
    const currentRole = await getCurrentUserRole();
    this.setState({
        isAdmin: currentRole === 'Admin',
    });
  }
  render() {
    if (this.state.isAdmin) {
      return <Comp {...this.props} />;
    } else {
      return (<div>您没有权限查看该页面请联系管理员</div>);
    }
  }
 };
}
```

性能监控包裹组件的生命周期进行统一埋点：

```
function withTiming(Comp) {
    return class extends Comp {
        constructor(props) {
            super(props);
            this.start = Date.now();
            this.end = 0;
        }
        componentDidMount() {
            super.componentDidMount &&
super.componentDidMount();
            this.end = Date.now();
```

```
            console.log(`${WrappedComponent.name} 组件渲染时间为
${this.end - this.start} ms`);
        }
        render() {
            return super.render();
        }
    };
}
```

代码复用可以将重复的逻辑进行抽象。

使用注意:

1、 纯函数：增强函数应为纯函数避免侵入修改元组件

2、 避免用法污染：理想状态下应透传元组件的无关参数与事件尽量保证用法不变

3、 命名空间：为 HOC 增加特异性的组件名称这样能便于开发调试和查找问题

4、 引用传递：如果需要传递元组件的 refs 引用可以使用 React.forwardRef

5、 静态方法：元组件上的静态方法并无法被自动传出会导致业务层无法调用解决：

6、 函数导出

7、 静态方法赋值

8、 重新渲染：由于增强函数每次调用是返回一个新组件因此如果在 Render 中使用增强函数就会导致每次都重新渲染整个 HOC 而且之前的状态会丢失

React 如何进行组件/逻辑复用?

抛开已经被官方弃用的 Mixin,组件抽象的技术目前有三种比较主流:

高阶组件:

1、 属性代理

2、 反向继承

3、 渲染属性

4、 react-hooks

你对 Time Slice 的理解?

时间分片

1、 React 在渲染 render 的时候不会阻塞现在的线程

2、 如果你的设备足够快你会感觉渲染是同步的

3、 如果你设备非常慢你会感觉还算是灵敏的

4、 虽然是异步渲染但是你将会看到完整的渲染而不是一个组件一行行的渲染出来

5、 同样书写组件的方式

6、 也就是说这是 React 背后在做的事情对于我们开发者来说是透明的具体是什么样的效果呢

setState 到底是异步还是同步?

先给出答案：有时表现出异步,有时表现出同步

1、 setState 只在合成事件和钩子函数中是“异步”的在原生事件和 setTimeout 中都是同步的

2、 setState 的“异步”并不是说内部由异步代码实现其实本身执行的过程和代码都是同步的只是合成事件和钩子函数的调用顺序在更新之前导致在合成事件和钩子函数中没法立马拿到更新后的值形成了所谓的“异步”当然可以通过第二个参数 setState(partialState, callback)中的 callback 拿到更新后的结果

3、 setState 的批量更新优化也是建立在“异步”合成事件、钩子函数之上的在原生事件和 setTimeout 中不会批量更新在“异步”中如果对同一个值进行多次 setState 的批量更新策略会对其进行覆盖取最后一次的执行如果是同时 setState 多个不同的值在更新时会对其进行合并批量更新