React

1、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

2、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
  }
})
```

3、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

4、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

5、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

6、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

7、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

8、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
```

9、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

- `redux-logger`：提供日志输出

- `redux-thunk`：处理异步操作
- `redux-promise`：处理异步操作，`actionCreator`的返回值是`promise`

10、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。

11、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

12、react旧版生命周期函数

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
- `componentWillMount`：组件即将被装载、渲染到页面上
- `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

运行中状态

- `componentWillReceiveProps`:组件将要接收到属性的时候调用
- `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）
- `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

12 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:

- 使用`getDerivedStateFromProps`替换`componentWillMount`;
- 使用`getSnapshotBeforeUpdate`替换`componentWillUpdate`;
- 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:

reconciliation

- `componentWillMount`
- `componentWillReceiveProps`
- `shouldComponentUpdate`
- `componentWillUpdate`

commit

- `componentDidMount`
- `componentDidUpdate`
- `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() {}
```

```
17
18    // 组件更新后调用
19    componentDidUpdate() {}
20
21    // 组件即将销毁
22    componentWillUnmount() {}
23
24    // 组件已销毁
25    componentDidUnMount() {}
26  }
```

使用建议:

- 在`constructor`初始化`state`;
- 在`componentDidMount`中进行事件监听，并在`componentWillUnmount`中解绑事件;
- 在`componentDidMount`中进行数据的请求，而不是在`componentWillMount`;
- 需要根据`props`更新`state`时，使用`getDerivedStateFromProps(nextProps, prevState)`;
    - 旧 props 需要自己存储，以便比较;

```
1  public static getDerivedStateFromProps(nextProps, prevState) {
2      // 当新 props 中的 data 发生变化时，同步更新到 state 上
3      if (nextProps.data !== prevState.data) {
4          return {
5              data: nextProps.data
6          }
7      } else {
8          return null1
9      }
10 }
```

> 可以在componentDidUpdate监听 props 或者 state 的变化，例如:

```
1  componentDidUpdate(prevProps) {
2      // 当 id 发生变化时，重新获取数据
3      if (this.props.id !== prevProps.id) {
4          this.fetchData(this.props.id);
5      }
6  }
```

- 在componentDidUpdate使用setState时，必须加条件，否则将进入死循环;

- getSnapshotBeforeUpdate(prevProps, prevState)可以在更新之前获取最新的渲染数据，它的调用是在 render 之后， update 之前；
- 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。

13、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

14、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 `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
- 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

15、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

16、react性能优化方案

- 重写`shouldComponentUpdate`来避免不必要的dom操作
- 使用 `production` 版本的`react.js`
- 使用`key`来帮助`React`识别列表中所有子组件的最小变化

16、简述flux 思想

> `Flux` 的最大特点，就是数据的"单向流动"。

- 用户访问 `View`
- `View`发出用户的 `Action`
- `Dispatcher` 收到`Action`，要求 `Store` 进行相应的更新
- `Store` 更新后，发出一个`"change"`事件
- `View` 收到`"change"`事件后，更新页面

17、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>
}
```

2. 尽量不要在 `componentWillRevicesProps` 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

3. 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

4. 遍历子节点的时候，不要用 index 作为组件的 key 进行传入

18、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>
  }
```

19、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

20、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

21、介绍一下react

1. 以前我们没有jquery的时候，我们大概的流程是从后端通过ajax获取到数据然后使用jquery生成dom结果然后更新到页面当中，但是随着业务发展，我们的项目可能会越来越复杂，我们每次请求到数据，或则数据有更改的时候，我们又需要重新组装一次dom结构，然后更新页面，这样我们手动同步dom和数据的成本就越来越高，而且频繁的操作dom，也使我我们页面的性能慢慢的降低。
2. 这个时候mvvm出现了，mvvm的双向数据绑定可以让我们在数据修改的同时同步dom的更新，dom的更新也可以直接同步我们数据的更改，这个特定可以大大降低我们手动去维护dom更新的成本，mvvm为react的特性之一，虽然react属于单项数据流，需要我们手动实现双向数据绑定。
3. 有了mvvm还不够，因为如果每次有数据做了更改，然后我们都全量更新dom结构的话，也没办法解决我们频繁操作dom结构(降低了页面性能)的问题，为了解决这个问题，react内部实现了一套虚拟dom结构，也就是用js实现的一套dom结构，他的作用是讲真实dom在js中做一套缓存，每次有数据更改的时候，react内部先使用算法，也就是鼎鼎有名的diff算法对dom结构进行对比，找到那些我们需要新增、更新、删除的dom节点，然后一次性对真实DOM进行更新，这样就大大降低了操作dom的次数。 那么diff算法是怎么运作的呢，首先，diff针对类型不同的节点，会直接判定原来节点需要卸载并且用新的节点来装载卸载的节点的位置；针对于节点类型相同的节点，会对比这个节点的所有属性，如果节点的所有属性相同，那么判定这个节点不需要更新，如果节点属性不相同，那么会判定这个节点需要更新，react会更新并重渲染这个节点。
4. react设计之初是主要负责UI层的渲染，虽然每个组件有自己的state，state表示组件的状态，当状态需要变化的时候，需要使用setState更新我们的组件，但是，我们想通过一个组件重渲染它的兄弟组件，我们就需要将组件的状态提升到父组件当中，让父组件的状态来控制这两个组件的重渲染，当我们组件的层次越来越深的时候，状态需要一直往下传，无疑加大了我们代码的复杂度，我们需要一个状态管理中心，来帮我们管理我们状态state。
5. 这个时候，redux出现了，我们可以将所有的state交给redux去管理，当我们的某一个state有变化的时候，依赖到这个state的组件就会进行一次重渲染，这样就解决了我们的我们需要一直把state往下传的问题。redux有action、reducer

的概念，action为唯一修改state的来源，reducer为唯一确定state如何变化的入口，这使得redux的数据流非常规范，同时也暴露出了redux代码的复杂，本来那么简单的功能，却需要完成那么多的代码。

6. 后来，社区就出现了另外一套解决方案，也就是mobx，它推崇代码简约易懂，只需要定义一个可观测的对象，然后哪个组价使用到这个可观测的对象，并且这个对象的数据有更改，那么这个组件就会重渲染，而且mobx内部也做好了是否重渲染组件的生命周期shouldUpdateComponent，不建议开发者进行更改，这使得我们使用mobx开发项目的时候可以简单快速的完成很多功能，连redux的作者也推荐使用mobx进行项目开发。但是，随着项目的不断变大，mobx也不断暴露出了它的缺点，就是数据流太随意，出了bug之后不好追溯数据的流向，这个缺点正好体现出了redux的优点所在，所以针对于小项目来说，社区推荐使用mobx，对大项目推荐使用redux

22、React怎么做数据的检查和变化

> ```
> Model`改变之后（可能是调用了`setState`），触发了`virtual dom`的更新，再用`diff`算法来`把virtual DOM`比较`real DOM`，看看是哪个`dom`节点更新了，再渲染`real dom
> ```

23、react-router里的`<Link>`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

24、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`connect`

> `connect`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;
    }
  }
```

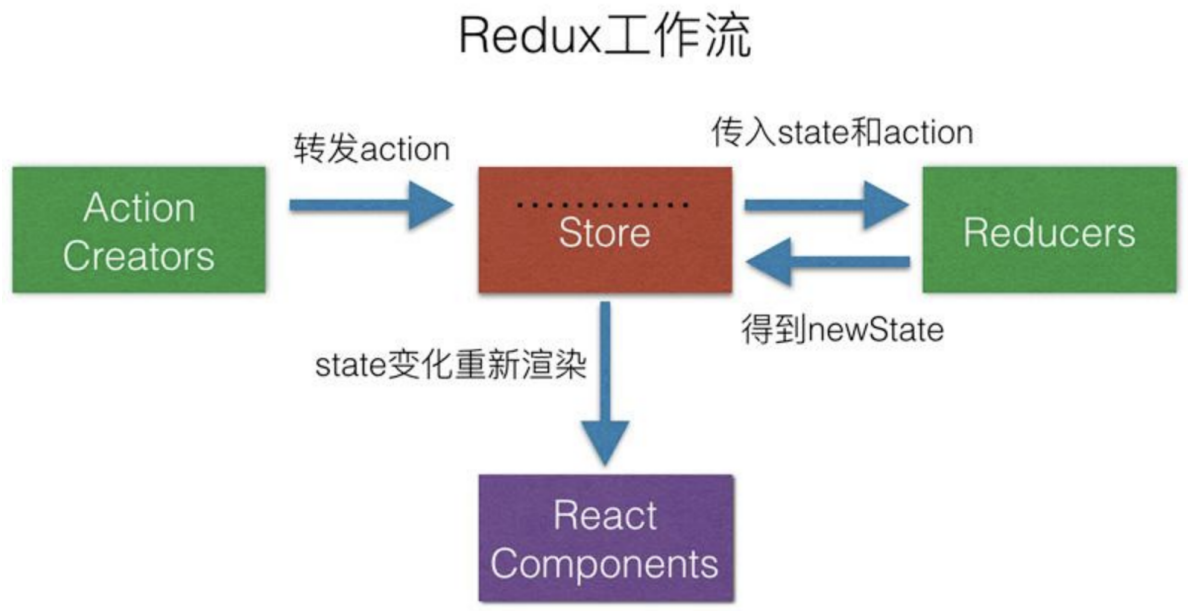
25、Redux实现原理解析

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



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`action`，`action`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};
```

26、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

27 react hooks，它带来了那些便利

- 代码逻辑聚合，逻辑复用
- HOC嵌套地狱
- 代替class

> 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

- 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；
- 时刻需要关注this的指向问题；
- 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；

1. 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useState，React 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

1. 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,
    })
}
```

1. 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。

useEffect(callback, [source])接受两个参数

- callback: 钩子回调函数;
- source: 设置触发条件，仅当 source 发生改变时才会触发;
- 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:

- componentDidMount: 传入[]时，就只会在初始化时调用一次

```
const useMount = (fn) => useEffect(fn, [])
```

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

```
const useUnmount = (fn) => useEffect(() => fn, [])
```

- 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()
})
```

1. 其它内置钩子:

- `useContext`: 获取 context 对象
- 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;
    - 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;
    - 配合useContext`的全局性，可以完成一个轻量级的 Redux;(easy-peasy)

- `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；
- `useMemo`: 用于缓存传入的 props，避免依赖的组件每次都重新渲染；
- `useRef`: 获取组件的真实节点；
- ```
  useLayoutEffect
  ```
  - 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
  - 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；
  - 可以获取更新后的 state；

1. 自定义钩子(useXxxxx): 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 (
        {title}
    )
}
```

28、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`componentDidMount`，`componentDidUpdate`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：

- `[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；
- `[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；
- `[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。

29、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

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

30、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
          };
      }
```

```
10
11        async componentDidMount() {
12            const isLogin = await getLoginStatus();
13            this.setState({ isLogin });
14        }
15
16        render() {
17          if (this.state.isLogin) {
18              return <WrappedComponent {...this.props} />;
19          }
20
21          return (<div>您还未登录...</div>);
22        }
23    }
24  }
```

31、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

- 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
- 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
- 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
- `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

32、Fiber

React 的核心流程可以分为两个部分:

- reconciliation(调度算法，也可称为render)
  - 更新 `state` 与 `props`;
  - 调用生命周期钩子;
  - 生成virtual dom

        - 这里应该称为 `Fiber Tree` 更为符合;
    - 通过新旧 vdom 进行 diff 算法，获取 vdom change
    - 确定是否需要重新渲染
- commit
    - 如需要，则操作 `dom` 节点更新

> 要了解 Fiber，我们首先来看为什么需要它

- **问题**: 随着应用变得越来越庞大，整个更新渲染的过程开始变得吃力，大量的组件渲染会导致主进程长时间被占用，导致一些动画或高频操作出现卡顿和掉帧的情况。而关键点，便是 同步阻塞。在之前的调度算法中，React 需要实例化每个类组件，生成一颗组件树，使用 同步递归 的方式进行遍历渲染，而这个过程最大的问题就是无法 暂停和恢复。
- **解决方**案: 解决同步阻塞的方法，通常有两种: 异步 与 任务分割。而 React Fiber 便是为了实现任务分割而诞生的
- 简述
    - 在 `React V16` 将调度算法进行了重构， 将之前的 `stack reconciler` 重构成新版的 fiber `reconciler`，变成了具有链表和指针的 单链表树遍历算法。通过指针映射，每个单元都记录着遍历当下的上一步与下一步，从而使遍历变得可以被暂停和重启
    - 这里我理解为是一种 任务分割调度算法，主要是 将原先同步更新渲染的任务分割成一个个独立的 小任务单位，根据不同的优先级，将小任务分散到浏览器的空闲时间执行，充分利用主进程的事件循环机制
- 核心
    - `Fiber` 这里可以具象为一个 数据结构

```
class Fiber {
    constructor(instance) {
        this.instance = instance
        // 指向第一个 child 节点
        this.child = child
        // 指向父节点
        this.return = parent
        // 指向第一个兄弟节点
        this.sibling = previous
    }
}
```

- 链表树遍历算法

: 通过 节点保存与映射，便能够随时地进行 停止和重启，这样便能达到实现任务分割的基本前提

  - 首先通过不断遍历子节点，到树末尾；
  - 开始通过 `sibling` 遍历兄弟节点；
  - return 返回父节点，继续执行2；
  - 直到 root 节点后，跳出遍历；
- 任务分割

  ，React 中的渲染更新可以分成两个阶段

  - **reconciliation 阶段**: vdom 的数据对比，是个适合拆分的阶段，比如对比一部分树后，先暂停执行个动画调用，待完成后再回来继续比对
  - **Commit 阶段**: 将 change list 更新到 dom 上，并不适合拆分，才能保持数据与 UI 的同步。否则可能由于阻塞 UI 更新，而导致数据更新和 UI 不一致的情况
- 分散执行:任务分割后，就可以把小任务单元分散到浏览器的空闲期间去排队执行，而实现的关键是两个新API:requestIdleCallback与requestAnimationFrame

  - 低优先级的任务交给 `requestIdleCallback` 处理，这是个浏览器提供的事件循环空闲期的回调函数，需要 `pollyfill`，而且拥有 `deadline` 参数，限制执行事件，以继续切分任务；
  - 高优先级的任务交给 `requestAnimationFrame` 处理；

```
// 类似于这样的方式
requestIdleCallback((deadline) => {
    // 当有空闲时间时，我们执行一个组件渲染；
    // 把任务塞到一个个碎片时间中去；
    while ((deadline.timeRemaining() > 0 ||
deadline.didTimeout) && nextComponent) {
        nextComponent = performWork(nextComponent);
    }
});
```

- **优先级策略:** 文本框输入 > 本次调度结束需完成的任务 > 动画过渡 > 交互反馈 > 数据更新 > 不会显示但以防将来会显示的任务

> - Fiber 其实可以算是一种编程思想，在其它语言中也有许多应用(Ruby Fiber)。
> - 核心思想是 任务拆分和协同，主动把执行权交给主线程，使主线程有时间空挡处理其他高优先级任务。
> - 当遇到进程阻塞的问题时，任务分割、异步调用 和 缓存策略 是三个显著的解决思路。

33、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:

- 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；

- 当组件已被销毁，如果再次调用setState，React 会报错警告，通常有两种解决办法
    - 将数据挂载到外部，通过 props 传入，如放到 Redux 或 父级中；
    - 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；

34、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:

- 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；
- 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；

1. 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} />
        }
    }
}
```

1. 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:

- 纯函数: 增强函数应为纯函数，避免侵入修改元组件;
- 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;
- 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;
- **引用传递**: 如果需要传递元组件的 refs 引用，可以使用React.forwardRef;

- 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:
  - 函数导出
  - 静态方法赋值
- **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;

35、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:
  - 属性代理
  - 反向继承
- 渲染属性
- react-hooks

36、你对 Time Slice的理解?

时间分片

- React 在渲染（render）的时候，不会阻塞现在的线程
- 如果你的设备足够快，你会感觉渲染是同步的
- 如果你设备非常慢，你会感觉还算是灵敏的
- 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
- 同样书写组件的方式

> 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?

37、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

- `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
- `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

- `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`，`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