## React 的工作原理

React 会创建一个虚拟 DOM(virtual DOM)。当一个组件中的状态改变时，React 首先会通过diff算法来标记虚拟 DOM 中的改变，第二步是调节(reconciliation)，会用 diff 的结果来更新 DOM。

## vue和react的异同

react和vue大体上是相同的，比如**都使用[虚拟DOM](https://so.csdn.net/so/search?q=%E8%99%9A%E6%8B%9FDOM&spm=1001.2101.3001.7020" \t "/Users/gdj/Documents\\x/_blank)高效的更新视图，提倡组件化，数据驱动视图，diff算法，对diff算法进行了优化，有router库实现url到组件的映射，有状态管理**等等

### 组件化的异同

相同点：

react和vue都推崇组件化，通过将页面拆分成一个一个小的可复用单元来提高代码的复用率和开发效率。在开发时react和vue有相同的套路，比如都有父子组件传参，都有数据状态管理，都有前端路由等。

不同点：

React推荐的做法是JSX + inline style, 也就是把 HTML 和 CSS 全都写进 JavaScript 中,即 all in js;

Vue 推荐的做法是 template 的单文件组件格式(简单易懂，从传统前端转过来易于理解),即 html,css,JS 写在同一个文件(vue也支持JSX写法)

### 2.虚拟DOM异同

相同点：

Vue与React都使用了 Virtual DOM + Diff算法， 不管是Vue的Template模板+options api 写法， 还是React的Class或者Function写法,最后都是生成render函数，而render函数执行返回VNode(虚拟dom树, 虚拟DOM的数据结构，本质上是棵树)。

当每一次UI更新时，总会根据render重新生成最新的虚拟dom树，然后跟以前缓存起来老的虚拟dom树进行比对，再使用Diff算法（框架核心）去真正更新真实DOM（虚拟DOM是JS对象结构，同样在JS引擎中，而真实DOM在浏览器渲染引擎中，所以操作虚拟DOM比操作真实DOM开销要小的多）

不同点：

react函数式组件思想，在发生数据（setState）更改后，会重新生成新的虚拟dom树，然后进行新旧虚拟dom树的diff（自顶向下的全量diff）

vue是组件响应思想，采用代理监听（watcher）数据，当一个组件内数据更改，可以明确知道并响应这个组件进行diff比较（局部订阅）

#### diff算法源码异同

在处理老节点部分，都需要把节点处理 key - value 的 Map 数据结构，方便在往后的比对中可以快速通过节点的 key 取到对应的节点。同样在比对两个新老节点是否相同时，key 是否相同也是非常重要的判断标准。所以不管是 React, 还是 Vue，在写动态列表的时候，都需要设置一个唯一值 key，这样在 diff 算法处理的时候性能才最大化。

不同点：react 会自顶向下全diff。vue会跟踪每一个组件的依赖关系,不需要重新渲染整个组件树。

在react中，当状态发生改变时，组件树就会自顶向下的全diff, 重新render页面， 重新生成新的虚拟dom tree, 新旧dom tree进行比较， 进行patch打补丁方式，局部更新dom。所以react为了避免父组件更新而引起不必要的子组件更新， 可以在shouldComponentUpdate做逻辑判断，减少没必要的render， 以及重新生成虚拟dom，做差量对比过程。

在vue中， 通过Object.defineProperty 把 data 属性全部转为 getter/setter。同时watcher实例对象会在组件渲染时，将属性记录为dep, 当dep 项中的 setter被调用时，通知watch重新计算，使得关联组件更新。（vue中的diff是用双端比较的方式实现的，但是vue2与vue3有区别）

### 3.数据驱动视图

#### vue中的数据驱动视图

Vuejs的数据驱动是通过MVVM这种框架来实现的。MVVM框架主要包含3个部分:model、view和 viewModel。

* Model:指的是数据部分，对应到前端就是javascript对象
* View:指的是视图部分，对应前端就是dom
* ViewModel:就是连接视图与数据的中间件

ViewModel是实现数据驱动视图的核心，当数据变化的时候，ViewModel能够监听到这种变化，并及时的通知view做出修改。同样的，当页面有事件触发时，ViewModel也能够监听到事件，并通知model进行响应。ViewModel就相当于一个观察者，监控着双方的动作，并及时通知对方进行相应的操作。

首先，vuejs在实例化的过程中，会对遍历传给实例化对象选项中的data 选项，遍历其所有属性并使用 Object.defineProperty 把这些属性全部转为 getter/setter。

同时每一个实例对象都有一个watcher实例对象，他会在模板编译的过程中,用getter去访问data的属性，watcher此时就会把用到的data属性记为依赖，这样就建立了视图与数据之间的联系。当之后我们渲染视图的数据依赖发生改变（即数据的setter被调用）的时候，watcher会对比前后两个的数值是否发生变化，然后确定是否通知视图进行重新渲染。这样就实现了所谓的数据对于视图的驱动。

#### react的数据驱动视图

首先了解一些列内容：

* pending：当前所有等待更新的state队列。
* isBatchingUpdates：React中用于标识当前是否处理批量更新状态，默认false。
* dirtyComponent：当前所有待更新state的组件队列。

React通过setState实现数据驱动视图，通过setState来引发一次组件的更新过程从而实现页面的重新渲染(除非shouldComponentUpdate返回false)。

* setState()首先将接收的第一个参数state存储在pending队列中；（state）
* 判断当前React是否处于批量更新状态，是的话就将需要更新state的组件添加到dirtyComponents中；（组件）
* 不是的话，它会遍历dirtyComponents的所有组件，调用updateComponent方法更新每个dirty组件（开启批量更新事务）

## 调用 setState 之后发生了什么？

在代码中调用 setState 函数之后，React 会将传入的参数对象与组件当前的状态合并，然后触发所谓的调和过程（Reconciliation）。经过调和过程，React 会以相对高效的方式根据新的状态构建 React 元素树并且着手重新渲染整个 UI 界面。在 React 得到元素树之后，React 会自动计算出新的树与老树的节点差异，然后根据差异对界面进行最小化重渲染。在差异计算算法中，React 能够相对精确地知道哪些位置发生了改变以及应该如何改变，这就保证了按需更新，而不是全部重新渲染。

## react 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

生命周期指的是组件实例从创建到销毁的流程，函数组件没有生命周期，只有类组件才有，因为只有class组件会创建组件实例。  
组件的生命周期可以分为挂载、更新、卸载阶段。  
render是class组件中唯一必须实现的方法

### 挂载

constructor 可以进行state和props的初始化

static getDerivedStateFromProps

render

componentDidMount 第一次渲染后调用，可以访问DOM，进行异步请求和定时器、消息订阅

### 更新（当组件的props或state变化会触发更新）

static getDerivedStateFromProps

shouldComponentUpdate 返回一个布尔值，默认返回true，可以通过这个生命周期钩子进行性能优化，确认不需要更新组件时调用

### render

getSnapShotBeforeUpdate

componentDidUpdate 在组件完成更新后调用

### 卸载

componentWillUnmount 组件从DOM中被移除的时候调用

### 错误捕获

static getDerivedStateFromError 在errorBoundary中使用

componentDidCatch

## shouldComponentUpdate 是做什么的，（react 性能优化是哪个周期函数？）

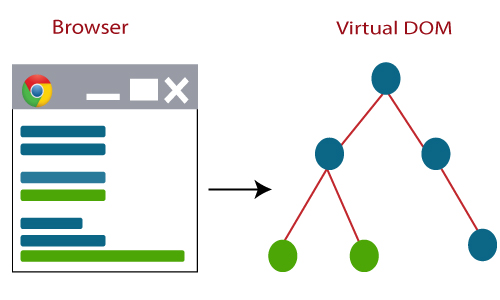
shouldComponentUpdate 这个方法用来判断是否需要调用 render 方法重新描绘 dom。因为 dom 的描绘非常消耗性能，如果我们能在 shouldComponentUpdate 方法中能够写出更优化的 dom diff 算法，可以极大的提高性能。

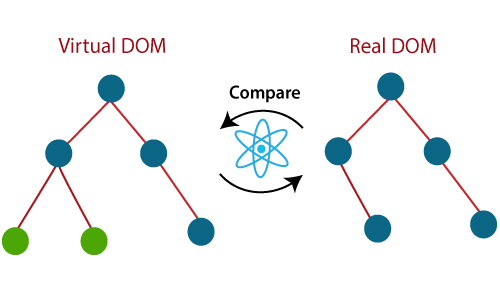
## 虚拟(Virtual)DOM 是什么？

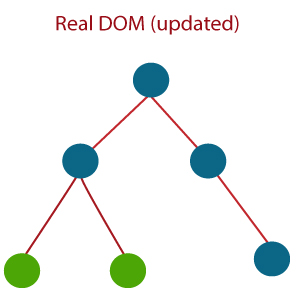
虚拟(Virtual)DOM是一个轻量级的 JavaScript 对象，它是真实 DOM 的内存表示。它是调用渲染函数和在屏幕上显示元素之间的中间步骤。它类似于将元素、它们的属性和内容作为对象及其属性列出的节点树。渲染函数创建 React 组件的节点树，然后更新此节点树以响应由用户或系统执行的各种操作导致的数据模型中的突变。

## 虚拟(Virtual)DOM 的工作原理是什么？

虚拟(Virtual)DOM 分三个步骤工作：

第1步： 每当 React App 中的任何数据发生变化时，整个 UI 都会以 Virtual DOM 表示形式重新渲染。  


第2步： 计算之前的 DOM 表示和新的 DOM 之间的差异。  


第3步： 一旦计算完成，真实的 DOM 只更新那些被改变的东西。  


## 真实 DOM 和虚拟 DOM 有什么区别？

下表指定了 Real DOM 和 Virtual DOM 之间的主要区别：

如果元素更新，真实的 DOM 会创建一个新的 DOM。

|  |  |
| --- | --- |
| 真实 DOM | 虚拟 DOM |
| 真正 DOM 更新较慢。 | 虚拟 DOM 更新更快。 |
| 真正 DOM 可以直接更新 HTML。 | 虚拟 DOM 不能直接更新 HTML。 |
| 如果元素更新 | 虚拟 DOM 会更新 JSX。 |
| 在真实 DOM 中，DOM 操作非常昂贵。 | 在虚拟 DOM 中，DOM 操作非常容易。 |
| 真实 DOM 中存在大量内存浪费。 | 虚拟 DOM 中没有内存浪费。 |

## 为什么虚拟 dom 会提高性能?(必考)

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。

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，视图就更新了。

## react diff 原理（常考，必考）

* 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。

## React性能优化手段

1. shouldComponentUpdate
2. memo
3. getDerviedStateFromProps
4. 使用Fragment
5. 循环元素使用正确的key
6. 拆分尽可能小的可复用组件，ErrorBoundary
7. 使用React.lazy和React.Suspense延迟加载不需要立马使用的组件

## React代码层面有哪些性能优化的方式？

React.memo()：可以缓存组件的渲染结果，避免不必要的重渲染。它接受一个函数组件，并返回一个新的组件，新组件将只在props发生变化时才重新渲染。

useMemo和useCallback。

shouldComponentUpdate：在类组件中，可以通过实现shouldComponentUpdate()方法来判断组件是否需要重新渲染。SCU接收两个参数：nextProps和nextState，我们可以在这个方法中比较当前props和state与下一个props和state的变化来决定是否需要重新渲染组件。

使用React.lazy()和Suspense进行组件懒加载。

对于大型应用，可以使用不可变数据的三方库比如Immer.js结合shouldComponentUpdate来做性能优化。

## 如何避免组件的重新渲染？

React 中最常见的问题之一是组件不必要地重新渲染。React 提供了两个方法，在这些情况下非常有用：

* React.memo(): 这可以防止不必要地重新渲染函数组件
* PureComponent: 这可以防止不必要地重新渲染类组件

这两种方法都依赖于对传递给组件的props的浅比较，如果 props 没有改变，那么组件将不会重新渲染。虽然这两种工具都非常有用，但是浅比较会带来额外的性能损失，因此如果使用不当，这两种方法都会对性能产生负面影响。

通过使用 React Profiler，可以在使用这些方法前后对性能进行测量，从而确保通过进行给定的更改来实际改进性能。

## 当调用setState时，React render 是如何工作的？

可以将"render"分为两个步骤：

虚拟 DOM 渲染：当render方法被调用时，它返回一个新的组件的虚拟 DOM 结构。当调用setState()时，render会被再次调用，因为默认情况下shouldComponentUpdate总是返回true，所以默认情况下 React 是没有优化的。

原生 DOM 渲染：React 只会在虚拟DOM中修改真实DOM节点，而且修改的次数非常少，它优化了真实DOM的变化，使React变得更快。

## 如何避免在React重新绑定实例？

1.将事件处理程序定义为内联箭头函数

<button onClick={() => { this.setState({ xx: xx }); }}>Submit</button>

2.使用箭头函数来定义方法

handleSubmit = () => { this.setState({ isFormSubmitted: true }); }

3.使用带有 Hooks 的函数组件

## setState是同步还是异步？

**v18之前：**

React是希望setState表现为异步的，因为批量更新可以优化性能。因此**在React能够管控到的地方，比如生命周期钩子和合成事件回调函数内，表现为异步**。

**在定时器和原生事件里，因为React管控不到，所以表现为同步。**

在某些情况下，我们需要立即获取更新后的状态，这时可以使用第二个可选参数callback，在状态更新后立即执行回调函数来获取更新后的状态。例如：

this.setState({ counter: this.state.counter + 1 }, () => {

console.log(this.state.counter); // 输出更新后的值

});

**v18之后：**

React18之后，**默认所有的操作都放到批处理中，因此setState不管在那儿调用都是异步**的了。

如果希望**同步更新**，可以使用**flushSync**这个API。

## React组件通信方式有哪些？

Props: 父组件可以通过 props 将数据传递给子组件。子组件可以通过 this.props 访问这些数据。

Callback: 父组件可以通过回调函数将函数传递给子组件。子组件可以在适当的时候调用这些回调函数，以便与父组件通信。

Context: 上下文是一种在组件树中共享数据的方法。通过 context，可以在组件树中传递数据，而不需要在每个级别显式地将 props 传递给所有组件。

Redux: 复杂应用全局状态管理可以使用Redux、Mobx等状态管理库，项目里一般使用React Redux或者RTK工具包。

Pub/Sub: 发布/订阅模式是一种通过事件来进行任意组件间通信的方法，和Vue里的事件总线原理一样。

Hooks里也可以通过useReducer和useContext来实现全局组件通信。

## React的特点是什么？

* JSX
* 组件
* 单向数据绑定
* 虚拟 DOM
* 简单
* 性能

## React最大的限制(缺点)是什么？

* React 只是一个库。它不是一个完整的框架。
* React有一个庞大的库，需要时间来理解。
* 新程序员可能很难理解和编码。
* React 使用内联模板和 JSX，这可能会很困难并充当障碍。它也使编码变得复杂。

## JSX是什么？

JSX 代表 JavaScript XML。它是一个 React 扩展，允许编写看起来类似于 HTML 的 JavaScript 代码。它使 HTML 文件易于理解。JSX 文件使 React 应用程序健壮并提高其性能。JSX 允许在编写 JavaScript 代码的同一文件中编写类似 XML 的语法，然后预处理器(即，像 Babel 这样的转译器)将这些表达式转换为实际的 JavaScript 代码。就像 XML/HTML 一样，JSX 标签也有标签名称、属性和子标签。

### JSX是什么，它和JS有什么区别

JSX是react的语法糖，它允许在html中写JS，它不能被浏览器直接识别，需要通过webpack、babel之类的编译工具转换为JS执行

### JSX与JS的区别：

* JS可以被打包工具直接编译，不需要额外转换，jsx需要通过babel编译，它是React.createElement的语法糖，使用jsx等价于React.createElement。
* jsx是js的语法扩展，允许在html中写JS；JS是原生写法，需要通过script标签引入。

## 浏览器为什么不能读取 JSX？

浏览器不能直接读取 JSX，因为它们只能理解 JavaScript 对象，而 JSX 不是常规的 JavaScript 对象。因此，我们需要使用 Babel 等转译器将 JSX 文件转换为 JavaScript 对象，然后将其传递给浏览器。

## 为什么要使用JSX？

* 它比常规 JavaScript 更快，因为它在将代码转换为 JavaScript 时执行优化。
* React 不是通过将标记和逻辑放在单独的文件中来分离技术，而是使用包含两者的组件。
* JSX是类型安全的，大部分错误都可以在编译时发现。
* JSX使创建模板变得更加容易。

## 从“在 React 中，一切都是组件”中理解了什么?

在 React 中，组件是 React 应用程序的构建块。这些组件将整个 React 应用程序的 UI 划分为小的、独立的、可重用的代码片段。React 独立渲染这些组件中的每一个，而不会影响应用程序 UI 的其余部分。因此，可以说，在 React 中，一切都是组件。

## 高阶组件(HOC)是什么？

在 React 中，高阶组件是一种重用组件逻辑的高级技术。它是一个接受一个组件并返回一个新组件的函数。换句话说，它是一个接受另一个函数作为参数的函数。根据官网的说法，它不是 React API 中的特性(部分)，而是从 React 的组合性质中出现的一种模式。

## 可以用 HOC 做什么？

* 代码可重用性
* Props 操作
* 状态操纵
* 渲染劫持

## 为什么必须以大写字母开头的组件名称？

在 React 中，组件名称必须以大写字母开头。如果以小写开头的组件名称，它将作为无法识别的标签抛出错误。这是因为，在 JSX 中，小写标记名称被视为 HTML 标记

## React 中 keys 的作用是什么？

密钥是唯一标识符。在 React 中，它用于识别哪些项目已从列表中更改、更新或删除。当动态创建组件或用户更改列表时，它很有用。它还有助于确定集合中的哪些组件需要重新渲染，而不是每次都重新渲染整个组件集。它提高了应用程序的性能。

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。

## 展示组件(Presentational component)和容器组件(Container component)之间有何不同

* 展示组件关心组件看起来是什么。展示专门通过 props 接受数据和回调，并且几乎不会有自身的状态，但当展示组件拥有自身的状态时，通常也只关心 UI 状态而不是数据的状态。
* 容器组件则更关心组件是如何运作的。容器组件会为展示组件或者其它容器组件提供数据和行为(behavior)，它们会调用 Flux actions，并将其作为回调提供给展示组件。容器组件经常是有状态的，因为它们是(其它组件的)数据源。

## 类组件(Class component)和函数式组件(Functional component)之间有何不同

语法：类组件使用ES6的class语法创建组件，而函数式组件使用函数声明来创建组件。

状态管理：类组件可以使用state来管理组件的内部状态，而函数式组件则通常使用userState Hook来管理状态。

生命周期：类组件可以使用生命周期方法，如componentDidMount、componentDidUpdate等来管理组件的生命周期，而函数式组件则使用useEffect Hook来管理。

调用方式：如果是一个函数组件，调用则是执行函数即可，如果是一个类组件，则需要将组件进行实例化，然后调用实例对象的render方法

性能：函数式组件通常比类组件更轻量级，因为类组件需要实例化，而函数式组件只是普通函数调用。

## (组件的)状态(state)和属性(props)之间有何不同

* State 是一种数据结构，用于组件挂载时所需数据的默认值。State 可能会随着时间的推移而发生突变，但多数时候是作为用户事件行为的结果。
* Props(properties 的简写)则是组件的配置。props 由父组件传递给子组件，并且就子组件而言，props 是不可变的(immutable)。组件不能改变自身的 props，但是可以把其子组件的 props 放在一起(统一管理)。Props 也不仅仅是数据--回调函数也可以通过 props 传递。

## 何为受控组件(controlled component)

在 HTML 中，类似 <input>, <textarea> 和 <select> 这样的表单元素会维护自身的状态，并基于用户的输入来更新。当用户提交表单时，前面提到的元素的值将随表单一起被发送。但在 React 中会有些不同，包含表单元素的组件将会在 state 中追踪输入的值，并且每次调用回调函数时，如 onChange 会更新 state，重新渲染组件。一个输入表单元素，它的值通过 React 的这种方式来控制，这样的元素就被称为"受控元素"。

## 何为高阶组件(higher order component)

高阶组件是一个以组件为参数并返回一个新组件的函数。HOC 运行你重用代码、逻辑和引导抽象。最常见的可能是 Redux 的 connect 函数。除了简单分享工具库和简单的组合，HOC 最好的方式是共享 React 组件之间的行为。如果你发现你在不同的地方写了大量代码来做同一件事时，就应该考虑将代码重构为可重用的 HOC。

## 为什么建议传递给 setState 的参数是一个 callback 而不是一个对象

因为 this.props 和 this.state 的更新可能是异步的，不能依赖它们的值去计算下一个 state。

## 除了在构造函数中绑定 this，还有其它方式吗

你可以使用属性初始值设定项(property initializers)来正确绑定回调，create-react-app 也是默认支持的。在回调中你可以使用箭头函数，但问题是每次组件渲染时都会创建一个新的回调。

## (在构造函数中)调用 super(props) 的目的是什么

在 super() 被调用之前，子类是不能使用 this 的，在 ES2015 中，子类必须在 constructor 中调用 super()。传递 props 给 super() 的原因则是便于(在子类中)能在 constructor 访问 this.props。

## 应该在 React 组件的何处发起 Ajax 请求

在 React 组件中，应该在 componentDidMount 中发起网络请求。这个方法会在组件第一次“挂载”(被添加到 DOM)时执行，在组件的生命周期中仅会执行一次。更重要的是，你不能保证在组件挂载之前 Ajax 请求已经完成，如果是这样，也就意味着你将尝试在一个未挂载的组件上调用 setState，这将不起作用。在 componentDidMount 中发起网络请求将保证这有一个组件可以更新了。

## 描述事件在 React 中的处理方式。

为了解决跨浏览器兼容性问题，您的 React 中的事件处理程序将传递 SyntheticEvent 的实例，它是 React 的浏览器本机事件的跨浏览器包装器。

这些 SyntheticEvent 与您习惯的原生事件具有相同的接口，除了它们在所有浏览器中都兼容。有趣的是，React 实际上并没有将事件附加到子节点本身。React 将使用单个事件监听器监听顶层的所有事件。这对于性能是有好处的，这也意味着在更新 DOM 时，React 不需要担心跟踪事件监听器。

## createElement 和 cloneElement 有什么区别？

React.createElement():JSX 语法就是用 React.createElement()来构建 React 元素的。它接受三个参数，第一个参数可以是一个标签名。如 div、span，或者 React 组件。第二个参数为传入的属性。第三个以及之后的参数，皆作为组件的子组件。

React.createElement(

type,

[props],

[...children]

)

React.cloneElement()与 React.createElement()相似，不同的是它传入的第一个参数是一个 React 元素，而不是标签名或组件。新添加的属性会并入原有的属性，传入到返回的新元素中，而就的子元素奖杯替换。

React.cloneElement(

element,

[props],

[...children]

)

## React 中有三种构建组件的方式

React.createClass()、ES6 class 和无状态函数。

## 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。

## 为什么在文件中没有使用react，也要在文件顶部import React from “react”

只要使用了jsx，就需要引用react，因为jsx本质就是React.createElement

## React组件为什么不能返回多个元素(React组件为什么只能有一个根元素)

React组件最后会编译为render函数，函数的返回值只能是1个，如果不用单独的根节点包裹，就会并列返回多个值，这在js中是不允许的。

react的虚拟DOM是一个树状结构，树的根节点只能是1个，如果有多个根节点，无法确认是在哪棵树上进行更新，vue的根节点为什么只有一个也是同样的原因。

## React组件怎样可以返回多个组件

* 使用HOC（高阶函数）
* 使用React.Fragment,可以让你将元素列表加到一个分组中，而且不会创建额外的节点（类似vue的template)
* 使用数组返回

## **React中元素和组件的区别**

* react组件有类组件、函数组件
* react元素是通过jsx创建的

## React逻辑复用方式有哪些？

Mixin：有很多缺点，已被弃用，可以不考虑。

HOC(高阶组件)：高阶组件是一个函数，它接收一个组件作为参数并返回一个新的组件。高阶组件可以将一些通用的逻辑（如：数据获取、权限验证、错误处理等）封装到一个函数中，并将其作为高阶组件的参数传递给其他组件使用，HOC一般以withXxx命名，并可以结合装饰器优雅地使用。

Render Props：通过在组件中传递一个函数作为prop，该函数将用于渲染组件的内容。这个函数可以接收组件需要的数据和方法，并返回React元素。

Hooks：自定义Hooks，将通用逻辑封装到useXxx函数中，可以在多个组件内使用，常见的像数据请求、表单、防抖节流、拖拽等。

## react中如何修改state中的数据？this.setState是同步的还是异步的？this.setState中的第二个参数的作用是什么？为什么是异步的？

1.通过this.setState来修改state中的数据；

2.this.setState是异步的；

其中有两个参数，第一个参数是一个对象或者是一个函数（必须返回一个对象），

函数中的第一个值为（prevState），第二个参数是（prevProps）

例：

this.setState((prevState,prevProps)=>({ }))

3.为什么是异步的，一位state可以批量执行，也就是说当多个setState一起同时执行时会被合并，提高DOM的渲染效率；

1. this.setState本身其实是一个同步的，异步不是因为本身的运行机制或者代码，而是因为他所在的合成事件和钩子函数的调用顺序在更新之前，导致函数内没法立即拿到更新后的值，形成了所谓的异步，可以通过第二个参数中的callback拿到更新后的结果；

setState是同步还是异步，不是绝对的，根据不同情况，是不一样的，如下：

5.同步的情况

1）原生js绑定的事件，setTimeout/setInterval，ajax，promise.then内等 React 无法掌控的APIs情况下

6.异步的情况

1）由**React控制的事件处理函数**，以及**生命周期函数调用setState**时表现为异步 。

大部分开发中用到的都是React封装的事件，比如onChange、onClick、onTouchMove等（合成事件中），这些事件处理函数中的setState都是异步处理的。

## Redux中同步 action 与异步 action 最大的区别是什么

同步只返回一个普通 action 对象。而异步操作中途会返回一个 promise 函数。当然在 promise 函数处理完毕后也会返回一个普通 action 对象。thunk 中间件就是判断如果返回的是函数，则不传导给 reducer，直到检测到是普通 action 对象，才交由 reducer 处理。

## react三大属性

指与组件生命周期和状态管理密切相关的三个核心特性或钩子函数

props,state,ref

### Ref

Props 是单向数据流，以 “声明式” 渲染组件；Ref 则是以 “命令式” 操作组件。

#### Ref 使用场景

处理 focus、视频播放 等

操作 dom 进行的动画

集成第三方的 dom 库

#### Ref 各类使用姿势

回调式的 Ref

Ref 转发

React.forwardRef 使用

## 什么是纯函数？

纯函数是不依赖并且不会在其作用域之外修改变量状态的函数。

本质上，纯函数始终在给定相同参数的情况下返回相同结果。

## Context上下文

context 是 React 提供的一个跨层级通信的方式，它允许父组件向其下面整个树提供数据，让我们可以不用显式地通过组件树逐层传递 props。

### 特点

* 1、跨级传递数据：Context 允许您在组件树中任意深度传递数据，而不必每层手动传递 props。
* 2、避免多余的层次：通过 Context，您可以避免将数据传递到组件树中不需要它的组件。
* 3、易于管理：Context 允许您在一个地方管理数据，而不必在每个组件中手动管理

### 使用场景

* 主题
* 当前账户
* 路由
* 还有一些状态会被远程组件用到或修改，结合 reducer 和 context 来使用

### 如何使用 Context

createContext(defaultValue) 创建上下文

SomeContext.Provider 向组件提供上下文值

useContext 函数组件中读取上下文值的方法

SomeContext.Consumer 函数组件中读取上下文值的方法（传统方法）

static contextType 类组件中指定上下文的方法，然后通过 this.content 来读取上下文值

### 例子:

父组件 Test.tsx

import React, { createContext, useCallback, useState } from 'react';

import TestContext from './TestContext';

**export const ContextDemo = createContext({**

**setNum: (num: number) => {},**

**num: 0,**

**});**

function Test() {

const [num, setNum] = useState(0)

return (

**<ContextDemo.Provider value={{ setNum, num }}>**

<div className='Test'>

**<TestContext />**

**<TestContext2 />**

**<TestContext3 />**

</div>

**</ContextDemo.Provider>**

)

}

export default Test;

子孙组件TestContext.tsx

函数组件中读取 context 的方法：ContextDemo.Consumer 和 useContext

// ContextDemo.Consumer方法

function TestContext(props: IProps) {

return (

<ContextDemo.Consumer>

{/\*

直接使用Consumer

这种方式获取参数

但是这种方式 不方便在render意外的地方使用传下来的参数

\*/}

{context => (

<div>

<div>函数组件 子孙组件TestContext ~ ContextDemo.Consumer</div>

<div>传下来的{context.num}</div>

<button onClick={() => context.setNum(Math.random() \* 10)}>设置随机数</button>

</div>

)}

</ContextDemo.Consumer>

);

}

// useContext用法

function TestContext2() {

const context = useContext(ContextDemo);

return (

<div className='TestContext2'>

<div>函数组件 子孙组件TestContext2 ~ useContext</div>

<div>传下来的{context.num}</div>

<button onClick={() => context.setNum(Math.random() \* 10)}>设置随机数</button>

</div>

);

}

类组件 contextType

export default class TestContext3 extends Component {

static contextType = ContextDemo;

render() {

return (

<div>

<div>类组件 子孙组件TestContext3 ~ contextType</div>

<div>传下来的{this.context.num}</div>

<button onClick={() => this.context.setNum(Math.random() \* 10)}>设置随机数</button>

</div>

);

}

}

### 注意

* useContext 作为一个 hook，不能在循环、条件或 map() 调用中调用。
* 不要过度使用 context，因为 context 值有改动时，其父组件以及整棵树都会重新渲染。所以使用 context 前，先尝试通过 props 传递数据，或者把 JSX 作为 props 来传递
* createContext需要放到单独文件中(不要放在jsx/tsx中),否则在类组件中将获得undefined

### 拓展

* antd element等UI组件库,Form表单的values和Form.Item之间的传值就是使用Context上下文关联的

## React Fiber

React Fiber 是 React 16 引入的一种全新的协调引擎，旨在解决旧版 React 在性能和灵活性方面的不足。本文将深入探讨 React Fiber 的工作原理、其背后的设计理念，以及它如何提升应用的性能。我们会用通俗易懂的语言，帮助你轻松理解这个复杂的概念，并通过代码示例来进一步解释。

1. 什么是 React Fiber？  
React Fiber 是对 React 核心算法的一次彻底重构。旧版的 React 使用的是“Stack Reconciler”，它会在一次更新中同步地遍历整个组件树，这样的方式对于大型应用来说，可能会导致卡顿和不流畅的用户体验。而 Fiber 则采用了一种增量式的更新方式，使得渲染过程可以被中断和恢复，从而提升性能和响应速度。  
2. React Fiber 的设计理念  
2.1 异步可中断的渲染  
React Fiber 最重要的设计目标之一就是使渲染过程变得可中断。传统的同步渲染方式在处理大型组件树时，可能会阻塞主线程，导致用户界面无响应。Fiber 通过将渲染工作分成一个个小任务，使得渲染过程可以在必要时被中断，从而让浏览器有机会处理用户输入等高优先级的任务。  
任务切片  
Fiber 的异步渲染是通过任务切片实现的。任务切片将渲染过程分解为多个小步骤，这样在执行过程中可以插入高优先级任务，如用户输入或动画。  
2.2 优先级控制  
Fiber 允许为不同类型的更新分配不同的优先级。例如，用户输入的更新通常比动画更新优先级更高。通过这种机制，React 可以确保高优先级的任务（如用户输入）能够快速得到响应，而不会被低优先级的任务（如动画或网络请求）所阻塞。  
优先级调度  
Fiber 的调度器会根据任务的类型和优先级来调度任务，确保高优先级任务能够尽快得到处理。这是通过 React 内部的“优先级队列”实现的。  
示例代码

class MyComponent extends React.Component {

state = {

text: '',

items: []

};

handleChange = event => {

this.setState({ text: event.target.value });

};

handleClick = () => {

this.setState(prevState => ({

items: [...prevState.items, prevState.text],

text: ''

}));

};

render() {

return (

<div>

<input

type="text"

value={this.state.text}

onChange={this.handleChange}

/>

<button onClick={this.handleClick}>Add Item</button>

<ul>

{this.state.items.map((item, index) => (

<li key={index}>{item}</li>

))}

</ul>

</div>

);

}

}

在示例中，handleChange 方法处理用户输入，handleClick 方法处理按钮点击。这两者的优先级会高于渲染 items 列表，从而保证用户输入和交互的流畅性。

#### 2.3 更好的错误边界处理

React Fiber 提供了更强大的错误边界处理机制。当组件在渲染过程中抛出错误时，Fiber 可以捕获这些错误并执行恢复操作，而不会影响到整个应用。这使得开发者可以更容易地处理错误并提供更好的用户体验。

示例代码

class ErrorBoundary extends React.Component {

state = { hasError: false };

static getDerivedStateFromError(error) {

return { hasError: true };

}

componentDidCatch(error, info) {

console.error("Error caught by Error Boundary:", error, info);

}

render() {

if (this.state.hasError) {

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

class MyComponent extends React.Component {

render() {

if (this.props.shouldThrow) {

throw new Error("Intentional error");

}

return <div>My Component</div>;

}

}

// 使用 ErrorBoundary 包裹 MyComponent

function App() {

return (

<ErrorBoundary>

<MyComponent shouldThrow={true} />

</ErrorBoundary>

);

}

在这个示例中，ErrorBoundary 组件捕获了 MyComponent 抛出的错误，并展示了一个错误信息，而不会导致整个应用崩溃。

### 3. React Fiber 的工作原理

#### 3.1 Fiber 节点

在 Fiber 架构中，每个组件对应一个 Fiber 节点。这些 Fiber 节点构成了一个链表结构，每个节点都包含了该组件的状态、更新队列以及指向子组件、兄弟组件和父组件的指针。通过这种结构，React 可以灵活地遍历和操作组件树。

##### Fiber 节点的结构

const fiberNode = {

type: MyComponent, // 组件类型

key: null, // 唯一键

stateNode: null, // 组件实例

child: null, // 子节点

sibling: null, // 兄弟节点

return: null, // 父节点

// 其他属性

};

#### 3.2 双缓存机制

React Fiber 使用了一种双缓存机制，分别称为“current”和“workInProgress”。“current”表示当前屏幕上显示的 UI，而“workInProgress”表示正在构建的新 UI 树。当“workInProgress”构建完成后，它将替换“current”成为新的 UI 树。这种机制确保了 UI 的一致性和稳定性。

双缓存机制示意

// current 表示当前屏幕上显示的树

let current = {

// 当前树的 Fiber 节点

};

// workInProgress 表示正在构建的新树

let workInProgress = {

// 新树的 Fiber 节点

};

// 当 workInProgress 构建完成后，将其设为 current

current = workInProgress;

#### 3.3 调度和协调

React Fiber 的调度器负责管理任务的优先级和执行顺序。调度器会根据任务的优先级决定何时执行每个任务，并在需要时中断和恢复任务。协调器则负责遍历组件树，比较新旧状态并生成需要更新的部分。通过这种方式，React 可以高效地更新 UI，而不会造成卡顿。

调度器示意代码

function performUnitOfWork(fiber) {

// 执行当前任务

// 省略具体逻辑

return fiber.child; // 返回下一个任务

}

function workLoop(deadline) {

let shouldYield = false;

while (nextUnitOfWork && !shouldYield) {

nextUnitOfWork = performUnitOfWork(nextUnitOfWork);

shouldYield = deadline.timeRemaining() < 1; // 检查剩余时间

}

requestIdleCallback(workLoop); // 请求下一个空闲时间继续执行

}

let nextUnitOfWork = initialFiber; // 初始任务

requestIdleCallback(workLoop); // 开始工作循环

### 4. React Fiber 带来的性能提升

#### 4.1 平滑的用户体验

由于 Fiber 的可中断特性，React 在处理大型组件树时不会阻塞主线程，从而保证了用户界面的流畅性和响应速度。这对于需要频繁更新的应用（如动画、复杂交互）来说尤为重要。

#### 4.2 精细的控制

开发者可以利用 Fiber 提供的优先级控制机制，精细地管理不同任务的执行顺序，从而优化应用的性能。例如，可以将用户输入和动画更新设置为高优先级，而将数据加载和后台任务设置为低优先级。

#### 4.3 更好的错误处理

通过增强的错误边界处理机制，React Fiber 可以在组件渲染过程中捕获并处理错误，从而避免整个应用崩溃。这不仅提高了应用的稳定性，也简化了开发者的调试工作。

### 5. 结论

React Fiber 是 React 框架的一次重要升级，它通过异步可中断的渲染、优先级控制和增强的错误处理机制，显著提升了应用的性能和用户体验。尽管 Fiber 的内部实现相对复杂，但其核心理念和优势对于开发者来说是非常有价值的。通过理解和应用这些理念，开发者可以创建更高效、更流畅的 React 应用。

## React为什么要推出Fiber架构？

React推出Fiber架构是为了解决React中的某些性能问题，并提供一些新的功能。

在React的传统架构中，采用了递归的方式创建虚拟DOM，递归过程是不能中断的。如果组件树的层级很深，递归会占用线程很多时间，因为JS是单线程的，所以当递归占用线程时，UI渲染线程就会一直处于等待中，所以用户会感受到页面卡顿。

为此，React推出了Fiber架构，Fiber 是 React 核心算法的重新实现，Fiber架构中有两个特性：动态优先级和可中断渲染。

其中，动态优先级会给每个任务都分配一个权重值，表示它的优先级。react在协调过程中，会动态调整任务的执行顺序，优先执行优先级高的任务，后执行优先级低的任务，解决了传统架构中所有任务都是相同优先级都是一次性执行的问题，从而提升了性能。

可中断渲染允许React在渲染过程中，对于任务进行暂停，然后适当时机再重启。这样的机制，解决了在传统架构中，因为渲染过程是单次执行不停止，而引发的渲染阻塞页面卡顿问题。

在React Fiber中，渲染过程是分"帧"进行的。当React在执行任务时，如果发现某个任务的执行时间超过了当前帧的剩余时间，它就会暂停执行，将控制权返回给浏览器去执行UI渲染，然后在下一帧继续执行未完成的任务，从而避免阻塞整个 UI。

此外，Fiber架构还提供了许多其他新功能，包括更好的动画支持、更好的浏览器事件处理、更好的同步和异步渲染支持以及更好的可访问性支持。

## ****jsx、ReactElement、FiberNode、dom之间的关系****

**jsx > **ReactElement > FiberNode > dom****

* 写JSX来描述React组件的结构和内容。
* JSX被Babel转译成React.createElement(\_\_jsx或\_\_jsxs)的调用，生成ReactElement。
* 在React的协调过程中，ReactElement被转换成FiberNode。
* FiberNode是React用来进行高效渲染和更新的数据结构，它支持并发渲染和优先级调度。
* 最终，FiberNode的信息被用来更新浏览器中的真实DOM，从而呈现用户界面。

## React Fiber 是怎么实现的

主要是通过两个原生的 API 来实现的 requestAnimationFrame 和 requestIdleCallback

显示器每秒 60 帧我们看着才不会感觉到卡嘛，比如动画的时候，一帧的时间内布局和绘制结束，还有剩余时间，JS 就会拿到主线程使用权，如果 JS 某个任务执行过长，动画下一帧开始时 JS 还没有执行完，就会导致掉帧，出现卡顿。

所以就通过把 JS 任务分成更小的任务块，分到每一帧上的方式，一帧时间到先暂停 JS 执行，然后下一帧绘制任完成再把主线程交给 JS，在每一帧绘制之前调用 requestAnimationFrame；在每一帧空间阶段，就是一帧动画任务完成，下一帧还没到开始时间，这中间还有时间的话就调用 requetIdleCallback，执行它里面的任务

### 还有其他方式能实现吗

函数式编程嘛，generator 就可以控制函数的运行过程中断和恢复

// 任务列表

const tasks = []

function \* run () {

let task

while(task = task.shift()) {

// 如果有高优先级的任务

if (hasHighPriorityTask()) {

// 中断

yield

}

}

}

// 中断后恢复

const iterator = run()

// 这样就恢复了

iterator.next()

## 为什么不直接用 generator

这个问题 React 开发者有在源码里回答过：

* 一是因为 React 是迭代的，而使用 generator + yield 的话需要把所有代码都包装成这个形式，非常麻烦，工作量很大
* 二是 generator 内部是有状态的，比如一个函数里有用到多个 yield 中断，就像 await 一样，有时候后面的会依赖前面的结果，可当后面的执行前，前面的又更新了，后面就无法拿到最新的值，这样就不可控了

所以就自己实现一个完全可控的

## 怎么判断当前是否有高优先级的任务

这个据我所知 JS 好像没法判断，而是按帧的时间，如果一帧内任务还没执行完，就中断当前任务，把控制权交给浏览器

## 浏览器什么时候才有空呢

每秒60帧算，每帧就是 1000/60 = 16.7ms 差不多，每帧任务执行完会调 requetIdleCallback(callback)，并且在 callback 中会有一个参数告诉我们当前帧还有多少时间给我们执行任务

## 那浏览器一帧内要做哪些事情

比如布局(layout)、绘制(paint)、JS 的执行、处理用户输入事件、requestAnimation 调用等，如果在一帧内处理完了这些剩余时间就用来执行 requetIdleCallback，直到下一帧开始

## 如果浏览器很忙，一帧结束了还没执行怎么办

requetIdleCallback(callback, options)，还有第二个参数，里面有个 timeout 字段(毫秒)，如果超过这个时间还没有被执行的话，那么下一帧就会强制执行

## 就是说超时后一定会被执行咯

也不一定，React 里有 5 个优先级的等级，高优先级的会被优先执行，低优先级的慢慢等下去，等级是这样的

Immediate: 最高优先级，会马上执行的不能中断

UserBlocking: 这一般是用户交互的结果，需要及时反馈

Normal: 普通等级的，比如网络请求等不需要用户立即感受到变化的

Low: 低优先级的，这种任务可以延后，但最后始终是要执行的

Idle: 最低等级的任务，可以被无限延迟的，比如 console.log()

如果是相同优先级的任务，就会按推入任务队列的顺序来执行

## 兼容性怎么样

requetIdleCallback 兼容性是很差的，React 也是通过 messageChannel 模拟实现的 requetIdleCallback 的功能

## 除了 MessageChannel 还有没有其他类似方法

还有一个功能类似的 BroadcastChannel