社招前端二面react面试题整理

解释 React 中 render() 的目的。

每个React组件强制要求必须有一个 **render()**。它返回一个 React 元素，是原生 DOM 组件的表示。如果需要渲染多个 HTML 元素，则必须将它们组合在一个封闭标记内，例如 `<form>` 、 `<group>` 、 `<div>` 等。此函数必须保持纯净，即必须每次调用时都返回相同的结果。

createElement过程

> React.createElement()： 根据指定的第一个参数创建一个React元素

javascript 复制代码

```
React.createElement(
  type,
  [props],
  [...children]
)
```

- 第一个参数是必填，传入的是似HTML标签名称，eg: ul, li
- 第二个参数是选填，表示的是属性，eg: className
- 第三个参数是选填, 子节点，eg: 要显示的文本内容

javascript 复制代码

```
//写法一：

var child1 = React.createElement('li', null, 'one');
    var child2 = React.createElement('li', null, 'two');
    var content = React.createElement('ul', { className: 'teststyle' }, child1, child2); // 第三个参
      ReactDOM.render(
          content,
        document.getElementById('example')
      );

//写法二：

var child1 = React.createElement('li', null, 'one');
    var child2 = React.createElement('li', null, 'two');
    var content = React.createElement('ul', { className: 'teststyle' }, [child1, child2]);
```

```
ReactDOM.render(
    content,
  document.getElementById('example')
);
```

组件之间传值

- 父组件给子组件传值

  在父组件中用标签属性的=形式传值

  在子组件中使用props来获取值

- 子组件给父组件传值

  在组件中传递一个函数

  在子组件中用props来获取传递的函数，然后执行该函数

  在执行函数的时候把需要传递的值当成函数的实参进行传递

- 兄弟组件之间传值

  利用父组件

  先把数据通过 【子组件】===》【父组件】

  然后在数据通过 【父组件】===〉【子组件】

  消息订阅

  使用PubSubJs插件

使用 React Hooks 好处是啥?

首先，Hooks 通常支持提取和重用跨多个组件通用的有状态逻辑，而无需承担高阶组件或渲染 `props` 的负担。 `Hooks` 可以轻松地操作函数组件的状态，而不需要将它们转换为类组件。Hooks 在类中不起作用，通过使用它们，咱们可以完全避免使用生命周期方法，例如

`componentDidMount` 、 `componentDidUpdate` 、 `componentWillUnmount` 。相反，使用像 `useEffect` 这样的内置钩子。

React 中的 `useState()` 是什么?

下面说明 `useState(0)` 的用途:

javascript 复制代码

```
const [count, setCounter] = useState(0);
const [moreStuff, setMoreStuff] = useState();

const setCount = () => {
  setCounter(count + 1);
  setMoreStuff();
};
```

`useState` 是一个内置的 React Hook。 `useState(0)` 返回一个元组，其中第一个参数 `count` 是计数器的当前状态， `setCounter` 提供更新计数器状态的方法。 咱们可以在任何地方使用 `setCounter` 方法更新计数状态-在这种情况下，咱们在 `setCount` 函数内部使用它可以做更多的事情，使用 Hooks，能够使咱们的代码保持更多功能，还可以避免过多使用基于类的组件。

什么是 React Hooks?

Hooks是 React 16.8 中的新添加内容。它们允许在不编写类的情况下使用 `state` 和其他 React 特性。使用 Hooks，可以从组件中提取有状态逻辑，这样就可以独立地测试和重用它。Hooks 允许咱们在不改变组件层次结构的情况下重用有状态逻辑，这样在许多组件之间或与社区共享 Hooks 变得很容易。

参考 前端进阶面试题详细解答

调和阶段 setState内部干了什么

- 当调用 setState 时，React会做的第一件事情是将传递给 setState 的对象合并到组件的当前状态
- 这将启动一个称为和解 （ `reconciliation` ） 的过程。和解 （ `reconciliation` ） 的最终目标是以最有效的方式，根据这个新的状态来更新 `UI` 。 为此， `React` 将构建一个新的 `React` 元素树（您可以将其视为 `UI` 的对象表示）

- 一旦有了这个树，为了弄清 UI 如何响应新的状态而改变，React 会将这个新树与上一个元素树相比较（ diff ）

> 通过这样做， React 将会知道发生的确切变化，并且通过了解发生什么变化，只需在绝对必要的情况下进行更新即可最小化 UI 的占用空间

为什么 JSX 中的组件名要以大写字母开头

因为 React 要知道当前渲染的是组件还是 HTML 元素

为什么不直接更新 `state` 呢？

如果试图直接更新 `state` ，则不会重新渲染组件。

javascript 复制代码

```
// 错误
This.state.message = 'Hello world';
```

需要使用 `setState()` 方法来更新 `state` 。它调度对组件 `state` 对象的更新。当 `state` 改变时，组件通过重新渲染来响应：

javascript 复制代码

```
// 正确做法
This.setState({message: ‘Hello World’});
```

React 组件生命周期有哪些不同阶段?

在组件生命周期中有四个不同的阶段：

1. **Initialization**：在这个阶段，组件准备设置初始化状态和默认属性。
2. **Mounting**：react 组件已经准备好挂载到浏览器 DOM 中。这个阶段包括 `componentWillMount` 和 `componentDidMount` 生命周期方法。
3. **Updating**：在这个阶段，组件以两种方式更新，发送新的 props 和 state 状态。此阶段包括 `shouldComponentUpdate` 、 `componentWillUpdate` 和 `componentDidUpdate` 生命周期方法。

4. **Unmounting**：在这个阶段，组件已经不再被需要了，它从浏览器 DOM 中卸载下来。这个阶段包含 `componentWillUnmount` 生命周期方法。 除以上四个常用生命周期外，还有一个错误处理的阶段： **Error Handling**：在这个阶段，不论在渲染的过程中，还是在生命周期方法中或是在任何子组件的构造函数中发生错误，该组件都会被调用。这个阶段包含了 `componentDidCatch` 生命周期方法。

在 React中元素（ element）和组件（ component）有什么区别?

简单地说，在 React中元素（虚拟DOM）描述了你在屏幕上看到的DOM元素。 换个说法就是，在 React中元素是页面中DOM元素的对象表示方式。在 React中组件是一个函数或一个类，它可以接受输入并返回一个元素。 注意：工作中，为了提高开发效率，通常使用JSX语法表示 React元素（虚拟DOM）。在编译的时候，把它转化成一个 React. createElement调用方法。

为什么类方法需要绑定到类实例?

在 JS 中， `this` 值会根据当前上下文变化。在 React 类组件方法中，开发人员通常希望 `this` 引用组件的当前实例，因此有必要将这些方法绑定到实例。通常这是在构造函数中完成的:

javascript 复制代码

```javascript
class SubmitButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isFormSubmitted: false,
    };
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleSubmit() {
    this.setState({
      isFormSubmitted: true,
    });
  }
  render() {
    return <button onClick={this.handleSubmit}>Submit</button>;
  }
}
```

React 的生命周期方法有哪些?

- `componentWillMount` :在渲染之前执行，用于根组件中的 App 级配置。

- `componentDidMount`：在第一次渲染之后执行，可以在这里做AJAX请求，DOM 的操作或状态更新以及设置事件监听器。
- `componentWillReceiveProps`：在初始化 `render` 的时候不会执行，它会在组件接受到新的状态(Props)时被触发，一般用于父组件状态更新时子组件的重新渲染
- `shouldComponentUpdate`：确定是否更新组件。默认情况下，它返回 `true`。如果确定在 `state` 或 `props` 更新后组件不需要在重新渲染，则可以返回 `false`，这是一个提高性能的方法。
- `componentWillUpdate`：在 `shouldComponentUpdate` 返回 `true` 确定要更新组件之前件之前执行。
- `componentDidUpdate`：它主要用于更新DOM以响应 `props` 或 `state` 更改。
- `componentWillUnmount`：它用于取消任何的网络请求，或删除与组件关联的所有事件监听器。

什么是高阶组件?

高阶组件(HOC)是接受一个组件并返回一个新组件的函数。基本上，这是一个模式，是从 React 的组合特性中衍生出来的，称其为**纯组件**，因为它们可以接受任何动态提供的子组件，但不会修改或复制输入组件中的任何行为。

```javascript
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

javascript 复制代码

HOC 可以用于以下许多用例

- 代码重用、逻辑和引导抽象
- 渲染劫持
- state 抽象和操作
- props 处理

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。

什么是 React的refs？为什么它们很重要

refs允许你直接访问DOM元素或组件实例。为了使用它们，可以向组件添加个ref属性。 如果该属性的值是一个回调函数，它将接受底层的DOM元素或组件的已挂载实例作为其第一个参数。可以在组件中存储它。

javascript 复制代码

```
export class App extends Component {
  showResult() {
    console.log(this.input.value);
  }
  render() {
    return (
      <div>
        <input type="text" ref={(input) => (this.input = input)} />
        <button onClick={this.showResult.bind(this)}>展示结果</button>
      </div>
    );
  }
}
```

如果该属性值是一个字符串， React将会在组件实例化对象的refs属性中，存储一个同名属性，该属性是对这个DOM元素的引用。可以通过原生的 DOM API操作它。

javascript 复制代码

```
export class App extends Component {
  showResult() {
    console.log(this.refs.username.value);
  }
  render() {
    return (
      <div>
        <input type="text" ref="username" />
        <button onClick={this.showResu1t.bind(this)}>展示结果</button>
      </div>
    );
  }
}
```

React中的状态是什么？它是如何使用的

状态是 React 组件的核心，是数据的来源，必须尽可能简单。基本上状态是确定组件呈现和行为的对象。与props 不同，它们是可变的，并创建动态和交互式组件。可以通过 `this.state()` 访问它们。

**

React 与 Vue 的 diff 算法有何不同?

diff 算法是指生成更新补丁的方式，主要应用于虚拟 DOM 树变化后，更新真实 DOM。所以 diff 算法一定存在这样一个过程：触发更新 → 生成补丁 → 应用补丁。

React 的 diff 算法，触发更新的时机主要在 state 变化与 hooks 调用之后。此时触发虚拟 DOM 树变更遍历，采用了深度优先遍历算法。但传统的遍历方式，效率较低。为了优化效率，使用了分治的方式。将单一节点比对转化为了 3 种类型节点的比对，分别是树、组件及元素，以此提升效率。

- 树比对：由于网页视图中较少有跨层级节点移动，两株虚拟 DOM 树只对同一层次的节点进行比较。
- 组件比对：如果组件是同一类型，则进行树比对，如果不是，则直接放入到补丁中。
- 元素比对：主要发生在同层级中，通过标记节点操作生成补丁，节点操作对应真实的 DOM 剪裁操作。

以上是经典的 React diff 算法内容。自 React 16 起，引入了 Fiber 架构。为了使整个更新过程可随时暂停恢复，节点与树分别采用了 FiberNode 与 FiberTree 进行重构。fiberNode 使用了双链表的结构，可以直接找到兄弟节点与子节点。整个更新过程由 current 与 workInProgress 两株树双缓冲完成。workInProgress 更新完成后，再通过修改 current 相关指针指向新节点。

Vue 的整体 diff 策略与 React 对齐，虽然缺乏时间切片能力，但这并不意味着 Vue 的性能更差，因为在 Vue 3 初期引入过，后期因为收益不高移除掉了。除了高帧率动画，在 Vue 中其他的场景几乎都可以使用防抖和节流去提高响应性能。

对React实现原理的理解

简版

- react 和 vue 都是基于 vdom 的前端框架，之所以用 vdom 是因为可以精准的对比关心的属性，而且还可以跨平台渲染
- 但是开发不会直接写 vdom，而是通过 jsx 这种接近 html 语法的 DSL，编译产生 render function，执行后产生 vdom
- vdom 的渲染就是根据不同的类型来用不同的 dom api 来操作 dom
- 渲染组件的时候，如果是函数组件，就执行它拿到 vdom。class 组件就创建实例然后调用 render 方法拿到 vdom。vue 的那种 option 对象的话，就调用 render 方法拿到 vdom
- 组件本质上就是对一段 vdom 产生逻辑的封装，函数、class、option 对象甚至其他形式都可以
- react 和 vue 最大的区别在状态管理方式上，vue 是通过响应式，react 是通过 setState 的 api。我觉得这个是最大的区别，因为它导致了后面 react 架构的变更
- react 的 setState 的方式，导致它并不知道哪些组件变了，需要渲染整个 vdom 才行。但是这样计算量又会比较大，会阻塞渲染，导致动画卡顿。所以 react 后来改造成了 fiber 架构，目标是可打断的计算
- 为了这个目标，不能变对比变更新 dom 了，所以把渲染分为了 render 和 commit 两个阶段，render 阶段通过 schedule 调度来进行 reconcile，也就是找到变化的部分，创建 dom，打上增删改的 tag，等全部计算完之后，commit 阶段一次性更新到 dom
- 打断之后要找到父节点、兄弟节点，所以 vdom 也被改造成了 fiber 的数据结构，有了 parent、sibling 的信息
- 所以 fiber 既指这种链表的数据结构，又指这个 render、commit 的流程
- reconcile 阶段每次处理一个 fiber 节点，处理前会判断下 shouldYield，如果有更高优先级的任务，那就先执行别的
- commit 阶段不用再次遍历 fiber 树，为了优化，react 把有 effectTag 的 fiber 都放到了 effectList 队列中，遍历更新即可
- 在 dom 操作前，会异步调用 useEffect 的回调函数，异步是因为不能阻塞渲染
- 在 dom 操作之后，会同步调用 useLayoutEffect 的回调函数，并且更新 ref
- 所以，commit 阶段又分成了 before mutation、mutation、layout 这三个小阶段，就对应上面说的那三部分

> 理解了 vdom、jsx、组件本质、fiber、render(reconcile + schedule) + commit(before mutation、mutation、layout) 的渲染流程，就算是对 react 原理有一个比较深的理解

下面展开分析

vdom

为什么 `react` 和 `vue` 都要基于 `vdom` 呢？直接操作真实 `dom` 不行么？

考虑下这样的场景：

- 渲染就是用 `dom api` 对真实 `dom` 做增删改，如果已经渲染了一个 `dom`，后来要更新，那就要遍历它所有的属性，重新设置，比如 `id`、`clasName`、`onclick` 等。
- 而 `dom` 的属性是很多的：

- 有很多属性根本用不到，但在更新时却要跟着重新设置一遍。
- 能不能只对比我们关心的属性呢?
- 把这些单独摘出来用 JS 对象表示不就行了?

- 这就是为什么要有 vdom，是它的第一个好处。
- 而且有了 vdom 之后，就没有和 dom 强绑定了，可以渲染到别的平台，比如 native、canvas 等等。
- 这是 vdom 的第二个好处。
- 我们知道了 vdom 就是用 JS 对象表示最终渲染的 dom 的，比如：

javascript 复制代码

```
{
  type: 'div',
  props: {
    id: 'aaa',
    className: ['bbb', 'ccc'],
    onClick: function() {}
  },
  children: []
}
```

然后用渲染器把它渲染出来，但是要让开发去写这样的 vdom 么？那肯定不行，这样太麻烦了，大家熟悉的是 html 那种方式，所以我们要引入编译的手段

dsl 的编译

- dsl 是 domain specific language，领域特定语言的意思，html、css 都是 web 领域的 dsl
- 直接写 vdom 太麻烦了，所以前端框架都会设计一套 dsl，然后编译成 render function，执行后产生 vdom。
- vue 和 react 都是这样

> 这套 dsl 怎么设计呢？前端领域大家熟悉的描述 dom 的方式是 html，最好的方式自然是也设计成那样。所以 vue 的 template，react 的 jsx 就都是这么设计的。vue 的 template compiler 是自己实现的，而 react 的 jsx 的编译器是 babel 实现的，是两个团队合作的结果。

编译成 `render function` 后再执行就是我们需要的 `vdom` 。接下来渲染器把它渲染出来就行了。那渲染器怎么渲染 `vdom` 的呢？

渲染 vdom

渲染 `vdom` 也就是通过 `dom api` 增删改 `dom` 。比如一个 `div` ，那就要 `document.createElement` 创建元素，然后 `setAttribute` 设置属性， `addEventListener` 设置事件监听器。如果是文本，那就要 `document.createTextNode` 来创建。所以说根据 `vdom` 类型的不同，写个 `if else` ，分别做不同的处理就行了。没错，不管 `vue` 还是 `react` ，渲染器里这段 `if else` 是少不了的：

javascript 复制代码

```
switch (vdom.tag) {
  case HostComponent:
    // 创建或更新 dom
  case HostText:
    // 创建或更新 dom
  case FunctionComponent:
    // 创建或更新 dom
  case ClassComponent:
    // 创建或更新 dom
}
```

> `react` 里是通过 `tag` 来区分 `vdom` 类型的，比如 `HostComponent` 就是元素， `HostText` 就是文本， `FunctionComponent` 、 `ClassComponent` 就分别是函数组件和类组件。那么问题来了，组件怎么渲染呢？这就涉及到组件的原理了：

组件

> 我们的目标是通过 `vdom` 描述界面，在 `react` 里会使用 `jsx` 。这样的 `jsx` 有的时候是基于 `state` 来动态生成的。如何把 `state` 和 `jsx` 关联起来呢？封装成 `function` 、 `class` 或者 `option` 对象的形式。然后在渲染的时候执行它们拿到 `vdom` 就行了。

这就是组件的实现原理：

javascript 复制代码

```
switch (vdom.tag) {
  case FunctionComponent:
      const childVdom = vdom.type(props);
```

```
        render(childVdom);
        //...
    case ClassComponent:
        const instance = new vdom.type(props);
        const childVdom = instance.render();

        render(childVdom);
        //...
}
```

如果是函数组件，那就传入 `props` 执行它，拿到 `vdom` 之后再递归渲染。如果是 `class` 组件，那就创建它的实例对象，调用 `render` 方法拿到 `vdom`，然后递归渲染。所以，大家猜到 `vue` 的 `option` 对象的组件描述方式怎么渲染了么？

javascript 复制代码

```
{
    data: {},
    props: {}
    render(h) {
        return h('div', {}, '');
    }
}
```

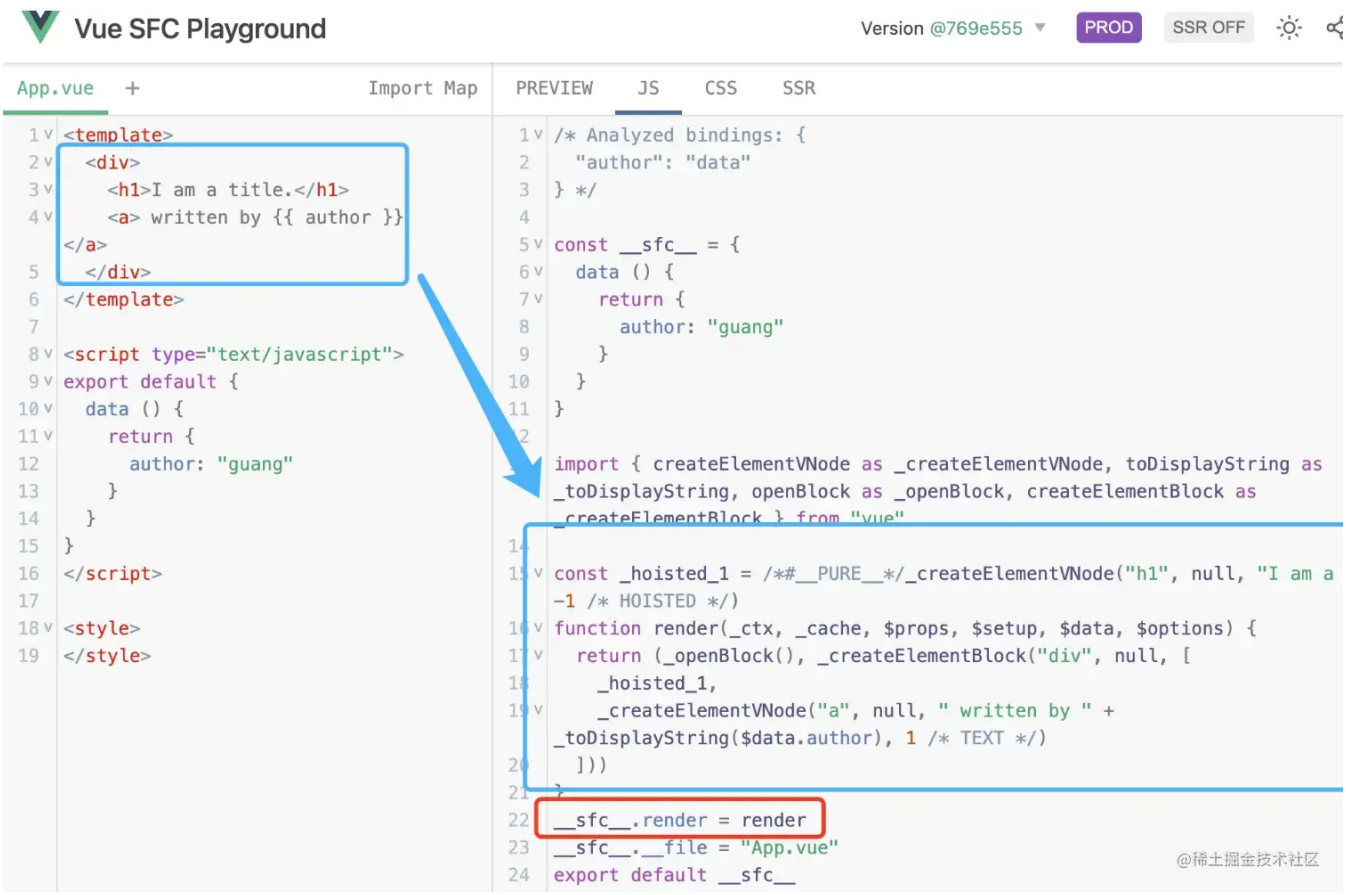
没错，就是执行下 `render` 方法就行：

javascript 复制代码

```
const childVdom = option.render();

render(childVdom);
```

大家可能平时会写单文件组件 `sfc` 的形式，那个会有专门的编译器，把 `template` 编译成 `render function`，然后挂到 `option 对象的` render` 方法上



所以组件本质上只是对产生 `vdom` 的逻辑的封装，函数的形式、 `option` 对象的形式、 `class` 的形式都可以。就像 `vue3` 也有了函数组件一样，组件的形式并不重要。基于 `vdom` 的前端框架渲染流程都差不多，vue 和 react 很多方面是一样的。但是管理状态的方式不一样， `vue` 有响应式，而 `react` 则是 `setState` 的 `api` 的方式。真说起来，vue 和 react 最大的区别就是状态管理方式的区别，因为这个区别导致了后面架构演变方向的不同。

状态管理

> `react` 是通过 `setState` 的 `api` 触发状态更新的，更新以后就重新渲染整个 `vdom` 。而 `vue` 是通过对状态做代理， `get` 的时候收集以来，然后修改状态的时候就可以触发对应组件的 `render` 了。

有的同学可能会问，为什么 `react` 不直接渲染对应组件呢？

想象一下这个场景：

父组件把它的 `setState` 函数传递给子组件，子组件调用了它。这时候更新是子组件触发的，但是要渲染的就只有那个组件么？明显不是，还有它的父组件。同理，某个组件更新实际上可能触发任意位置的其他组件更新的。所以必须重新渲染整个 `vdom` 才行。

那 vue 为啥可以做到精准的更新变化的组件呢？因为响应式的代理呀，不管是子组件、父组件、还是其他位置的组件，只要用到了对应的状态，那就会被作为依赖收集起来，状态变化的时候就可以触发它们的 render，不管是组件是在哪里的。这就是为什么 react 需要重新渲染整个 vdom，而 vue 不用。这个问题也导致了后来两者架构上逐渐有了差异。

react 架构的演变

- react15 的时候，和 vue 的渲染流程还是很像的，都是递归渲染 vdom，增删改 dom 就行。但是因为状态管理方式的差异逐渐导致了架构的差异。
- react 的 setState 会渲染整个 vdom，而一个应用的所有 vdom 可能是很庞大的，计算量就可能很大。浏览器里 js 计算时间太长是会阻塞渲染的，会占用每一帧的动画、重绘重排的时间，这样动画就会卡顿。作为一个有追求的前端框架，动画卡顿肯定是不行的。但是因为 setState 的方式只能渲染整个 vdom，所以计算量大是不可避免的。那能不能把计算量拆分一下，每一帧计算一部分，不要阻塞动画的渲染呢？顺着这个思路，react 就改造为了 fiber 架构。

fiber 架构

优化的目标是打断计算，分多次进行，但现在递归的渲染是不能打断的，有两个方面的原因导致的：

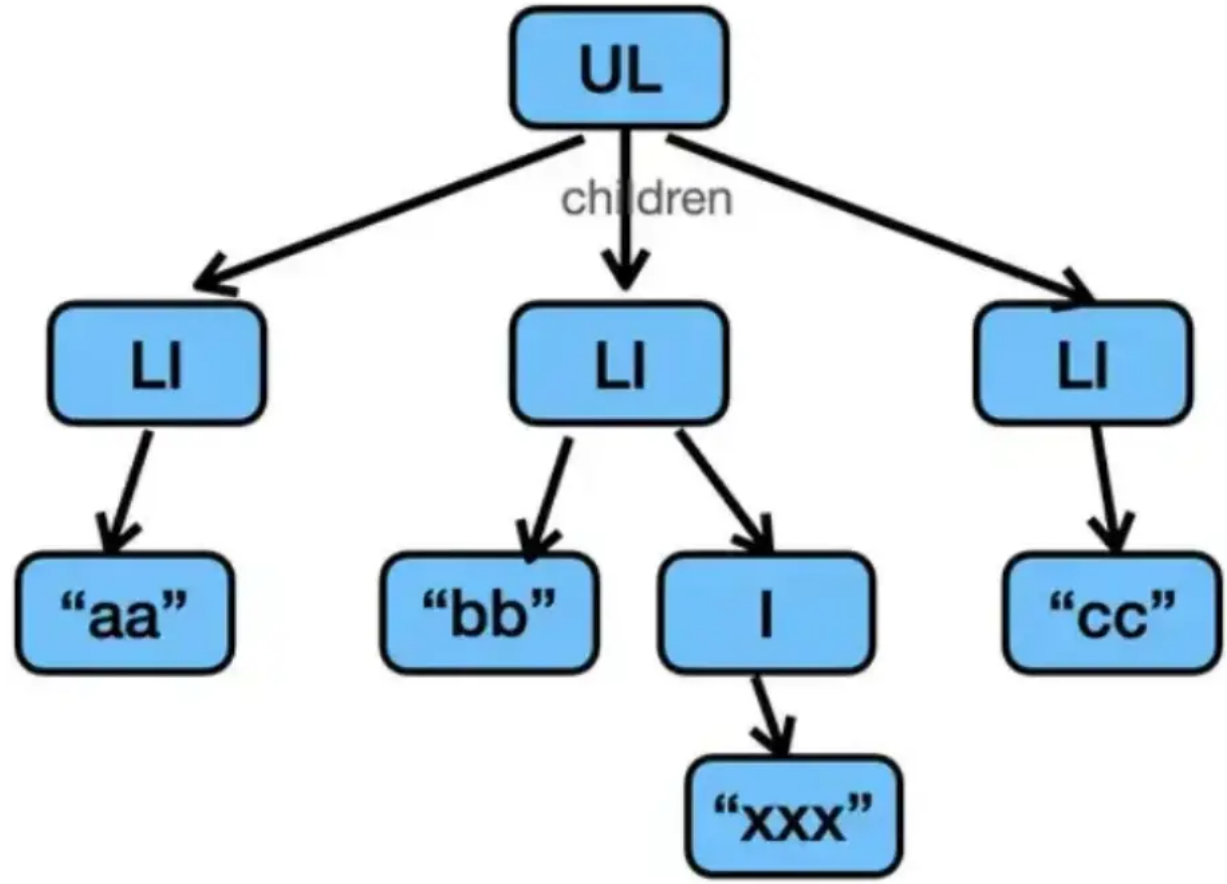
- 渲染的时候直接就操作了 dom 了，这时候打断了，那已经更新到 dom 的那部分怎么办？
- 现在是直接渲染的 vdom，而 vdom 里只有 children 的信息，如果打断了，怎么找到它的父节点呢？

第一个问题的解决还是容易想到的：

- 渲染的时候不要直接更新到 dom 了，只找到变化的部分，打个增删改的标记，创建好 dom，等全部计算完了一次性更新到 dom 就好了。
- 所以 react 把渲染流程分为了两部分： render 和 commit。
- render 阶段会找到 vdom 中变化的部分，创建 dom，打上增删改的标记，这个叫做 reconcile，调和。
- reconcile 是可以打断的，由 schedule 调度。
- 之后全部计算完了，就一次性更新到 dom，叫做 commit。
- 这样，react 就把之前的和 vue 很像的递归渲染，改造成了 render（reconcile + schdule） + commit 两个阶段的渲染。
- 从此以后，react 和 vue 架构上的差异才大了起来。

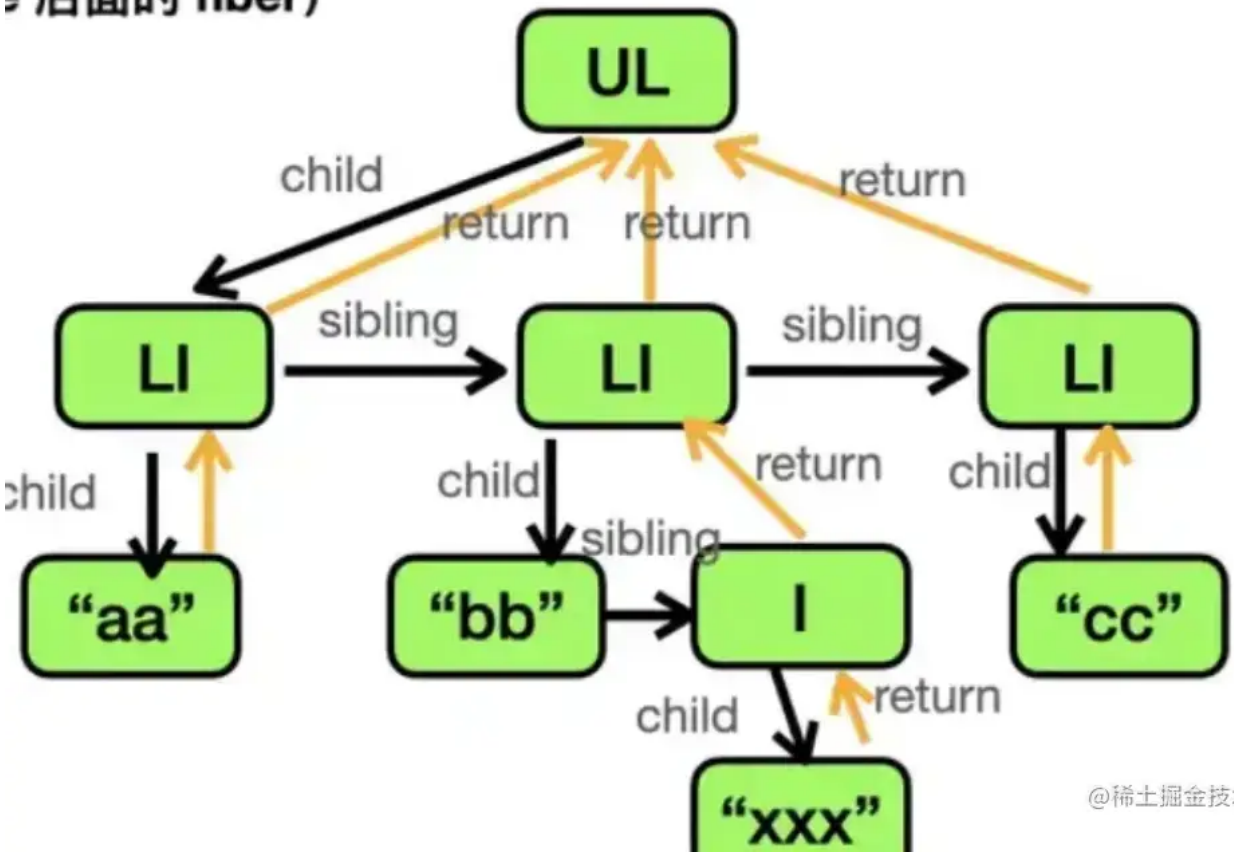
第二个问题，如何打断以后还能找到父节点、其他兄弟节点呢?

现有的 `vdom` 是不行的，需要再记录下 `parent` 、 `silbing` 的信息。所以 `react` 创造了 `fiber` 的数据结构。



reconcile (vdom 转 fiber，确定

e 后面的 fiber)



- 除了 children 信息外，额外多了 sibling 、 return ，分别记录着兄弟节点、父节点的信息。
- 这个数据结构也叫做 fiber 。（ fiber 既是一种数据结构，也代表 render + commit 的渲染流程） react 会先把 vdom 转换成 fiber ，再去进行 reconcile ，这样就是可打断的了。
- 为什么这样就可以打断了呢？因为现在不再是递归，而是循环了：

javascript 复制代码

```
function workLoop() {
  while (wip) {
    performUnitOfWork();
  }

  if (!wip && wipRoot) {
    commitRoot();
  }
}
```

- react 里有一个 workLoop 循环，每次循环做一个 fiber 的 reconcile ，当前处理的 fiber 会放在 workInProgress 这个全局变量上。
- 当循环完了，也就是 wip 为空了，那就执行 commit 阶段，把 reconcile 的结果更新到 dom 。
- 每个 fiber 的 reconcile 是根据类型来做的不同处理。当处理完了当前 fiber 节点，就把 wip 指向 sibling 、 return 来切到下个 fiber 节点。：

javascript 复制代码

```
function performUnitOfWork() {
  const { tag } = wip;

  switch (tag) {
    case HostComponent:
      updateHostComponent(wip);
      break;

    case FunctionComponent:
      updateFunctionComponent(wip);
      break;

    case ClassComponent:
      updateClassComponent(wip);
      break;
    case Fragment:
      updateFragmentComponent(wip);
      break;
    case HostText:
      updateHostTextComponent(wip);
```

```
      break;
    default:
      break;
  }

  if (wip.child) {
    wip = wip.child;
    return;
  }

  let next = wip;

  while (next) {
    if (next.sibling) {
      wip = next.sibling;
      return;
    }
    next = next.return;
  }

  wip = null;
}
```

> 函数组件和 `class` 组件的 `reconcile` 和之前讲的一样，就是调用 `render` 拿到 `vdom`，然后继续处理渲染出的 `vdom`：

javascript 复制代码

```
function updateClassComponent(wip) {
  const { type, props } = wip;
  const instance = new type(props);
  const children = instance.render();

  reconcileChildren(wip, children);
}

function updateFunctionComponent(wip) {
  renderWithHooks(wip);

  const { type, props } = wip;

  const children = type(props);
  reconcileChildren(wip, children);
}
```

- 循环执行 `reconcile`，那每次处理之前判断一下是不是有更高优先级的任务，就能实现打断了。

- 所以我们在每次处理 `fiber` 节点的 `reconcile` 之前，都先调用下 `shouldYield` 方法：

javascript 复制代码

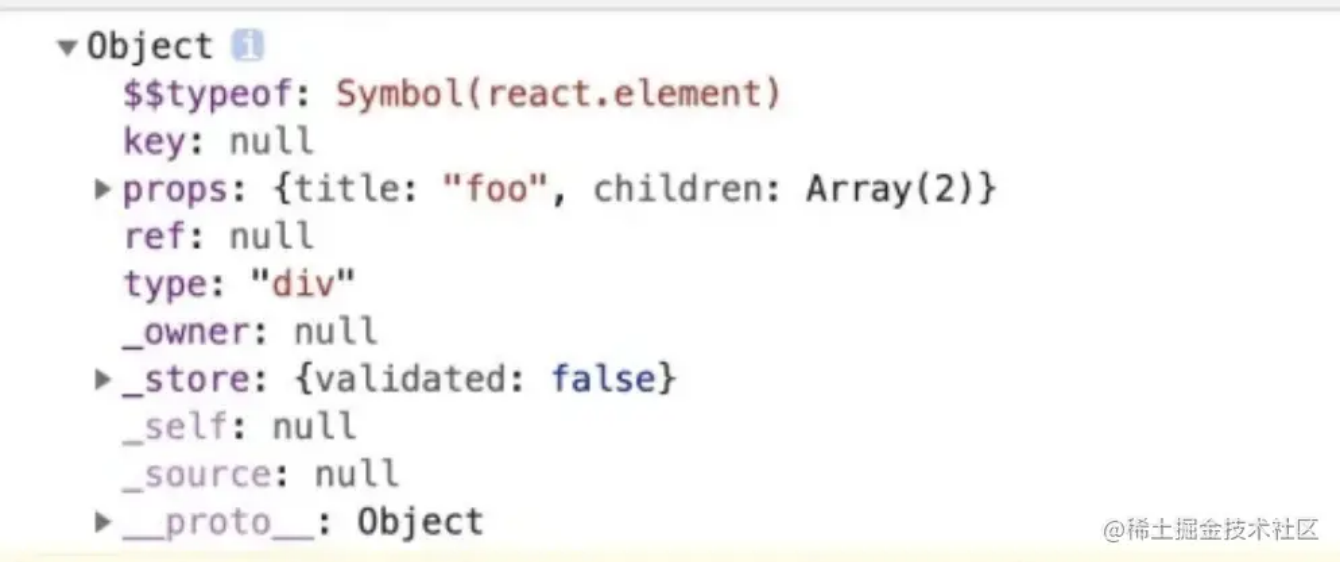
```
function workLoop() {
  while (wip && shouldYield()) {
    performUnitOfWork();
  }

  if (!wip && wipRoot) {
    commitRoot();
  }
}
```

- `shouldYiled` 方法就是判断待处理的任务队列有没有优先级更高的任务，有的话就先处理那边的 `fiber`，这边的先暂停一下。
- 这就是 `fiber` 架构的 `reconcile` 可以打断的原理。通过 `fiber` 的数据结构，加上循环处理前每次判断下是否打断来实现的。
- 聊完了 `render` 阶段（`reconcile + schedule`），接下来就进入 `commit` 阶段了。
- 前面说过，为了变为可打断的，`reconcile` 阶段并不会真正操作 `dom`，只会创建 `dom` 然后打个 `effectTag` 的增删改标记。
- `commit` 阶段就根据标记来更新 `dom` 就可以了。
- 但是 `commit` 阶段要再遍历一次 `fiber` 来查找有 `effectTag` 的节点，更新 `dom` 么?
- 这样当然没问题，但没必要。完全可以在 reconcile 的时候把有 effectTag 的节点收集到一个队列里，然后 commit 阶段直接遍历这个队列就行了。
- 这个队列叫做 `effectList`。
- `react` 会在 `commit` 阶段遍历 `effectList`，根据 `effectTag` 来增删改 `dom`。
- `dom` 创建前后就是 `useEffect`、`useLayoutEffect` 还有一些函数组件的生命周期函数执行的时候。
- `useEffect` 被设计成了在 `dom` 操作前异步调用，`useLayoutEffect` 是在 `dom` 操作后同步调用。
- 为什么这样呢?
- 因为都要操作 `dom` 了，这时候如果来了个 `effect` 同步执行，计算量很大，那不是把 fiber 架构带来的优势有毁了么?
- 所以 `effect` 是异步的，不会阻塞渲染。
- 而 `useLayoutEffect`，顾名思义是想在这个阶段拿到一些布局信息的，dom 操作完以后就可以了，而且都渲染完了，自然也就可以同步调用了。
- 实际上 `react` 把 `commit` 阶段也分成了 `3` 个小阶段。
- `before mutation`、`mutation`、`layout`。
- `mutation` 就是遍历 `effectList` 来更新 `dom` 的。
- 它的之前就是 `before mutation`，会异步调度 `useEffect` 的回调函数。

- 它之后就是 `layout` 阶段了，因为这个阶段已经可以拿到布局信息了，会同步调用 `useLayoutEffect` 的回调函数。而且这个阶段可以拿到新的 `dom` 节点，还会更新下 `ref` 。
- 至此，我们对 `react` 的新架构， `render` 、 `commit` 两大阶段都干了什么就理清了。

为什么 React 元素有一个 $$typeof 属性



> 目的是为了防止 XSS 攻击。因为 Symbol 无法被序列化，所以 React 可以通过有没有 $$typeof 属性来断出当前的 element 对象是从数据库来的还是自己生成的。

- 如果没有 $$typeof 这个属性，react 会拒绝处理该元素。
- 在 React 的古老版本中，下面的写法会出现 XSS 攻击：

javascript 复制代码

```
// 服务端允许用户存储 JSON
let expectedTextButGotJSON = {
  type: 'div',
  props: {
    dangerouslySetInnerHTML: {
      __html: '/* 把你想的搁着 */'
    },
  },
  // ...
};
let message = { text: expectedTextButGotJSON };

// React 0.13 中有风险
<p>
```

```
  {message.text}
</p>
```