

React

React

A JavaScript library for building user interfaces

打造用户界面

响应各种事件

- React 虚拟dom, fiber 原理解
析，生命周期以及diff 算法，
- redux flux mobx 等状态管理
- Hooks 的前世今生

React会先将你的代码转换成一个JavaScript对象，然后这个JavaScript对象再转换成真实DOM。这个JavaScript对象就是所谓的虚拟DOM。

```
1 <div class="title">
2   <span>Hello ConardLi</span>
3   <ul>
4     <li>苹果</li>
5     <li>橘子</li>
6   </ul>
7 </div>
```

```
1 const VIRTUALDom = {
2   type: 'div',
3   props: { class: 'title' },
4   children: [
5     type: 'span',
6     children: 'Hello ConardLi'
7   ], {
8     type: 'ul',
9     children: [
10       type: 'li',
11       children: '苹果'
12     }, {
13       type: 'li',
14       children: '橘子'
15     }]
16   }
17 }
```

React 16.0之前的版本虚拟dom的跟新采用的是循环和递归

- 任务一旦开始，无法结束，直到任务结束，主线程一直被占用
- 导致大量组件实例存在时，执行效率变低
- 用户交互的动画效果，出现页面卡顿

Fiber

- 利用浏览器空闲时间执行，不会长时间占用主线程
- 将对比更新dom的操作碎片化
- 碎片化的任务，可以根据需要被暂停

`requestIdleCallback`是浏览器提供的API，其利用浏览器空闲时间执行任务，当前任务可被终止，优先执行更高级别的任务

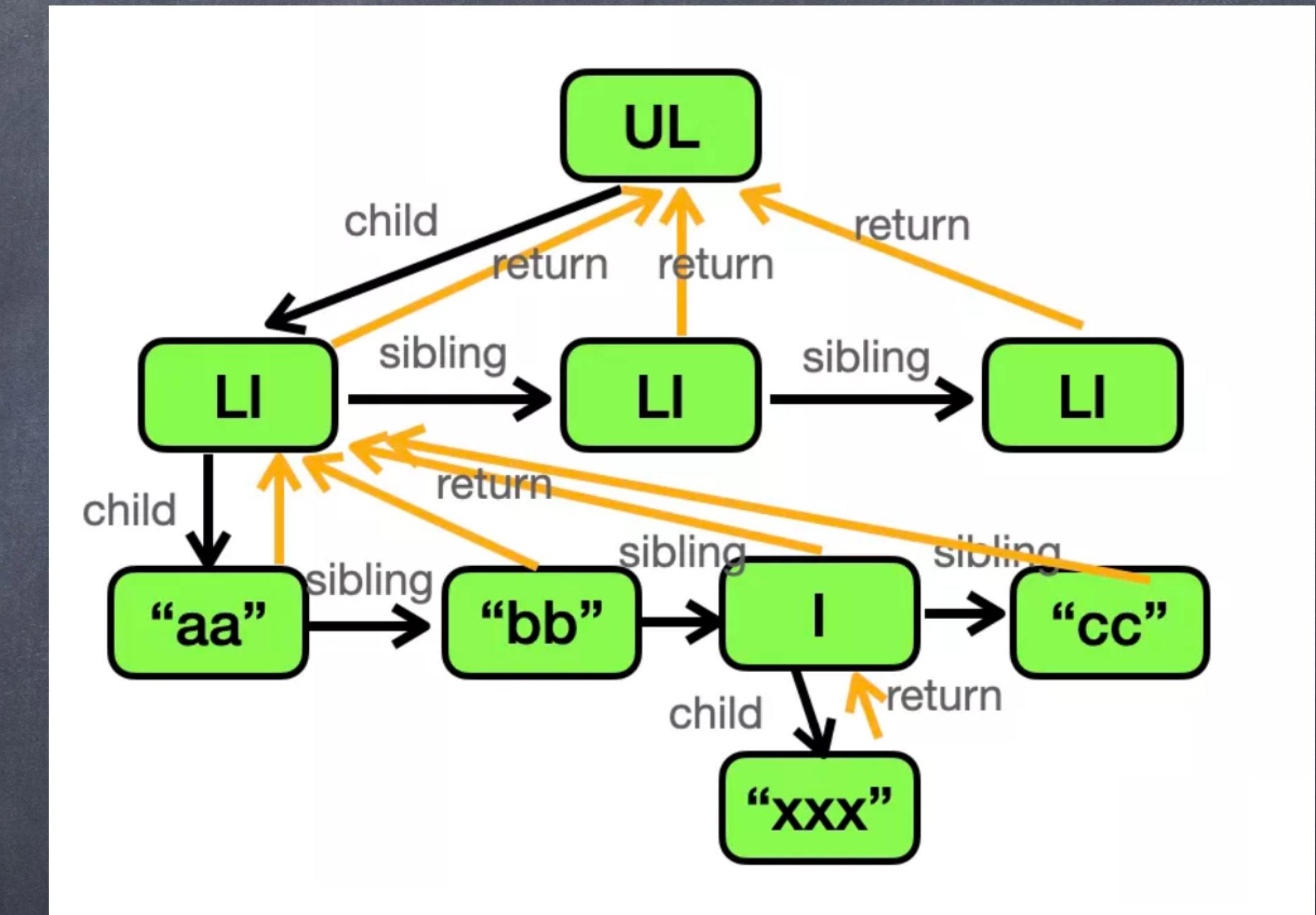
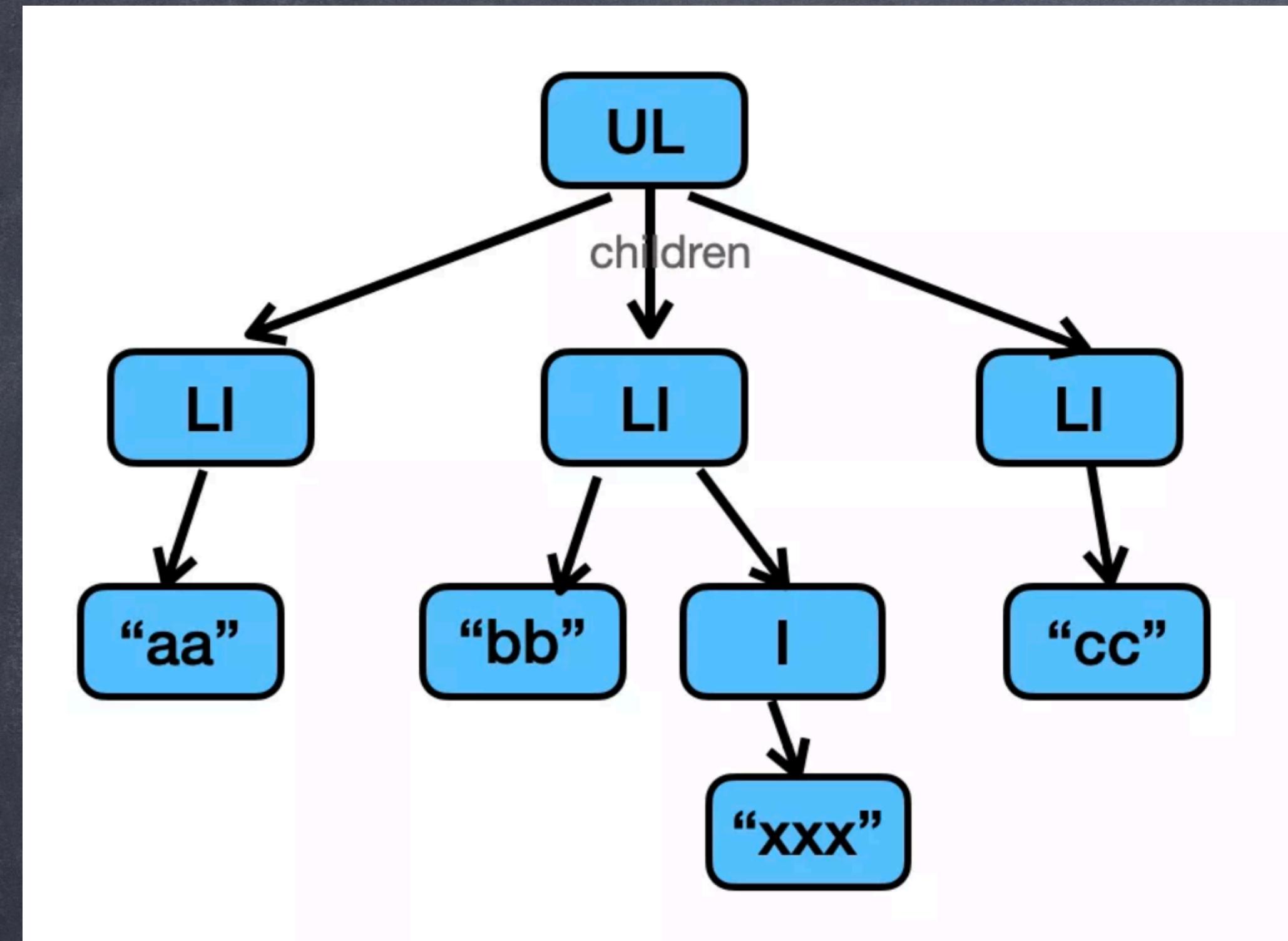
```
requestIdleCallback(function(deadline) {  
    // deadline.timeRemaining() 获取浏览器的空余时间  
})
```

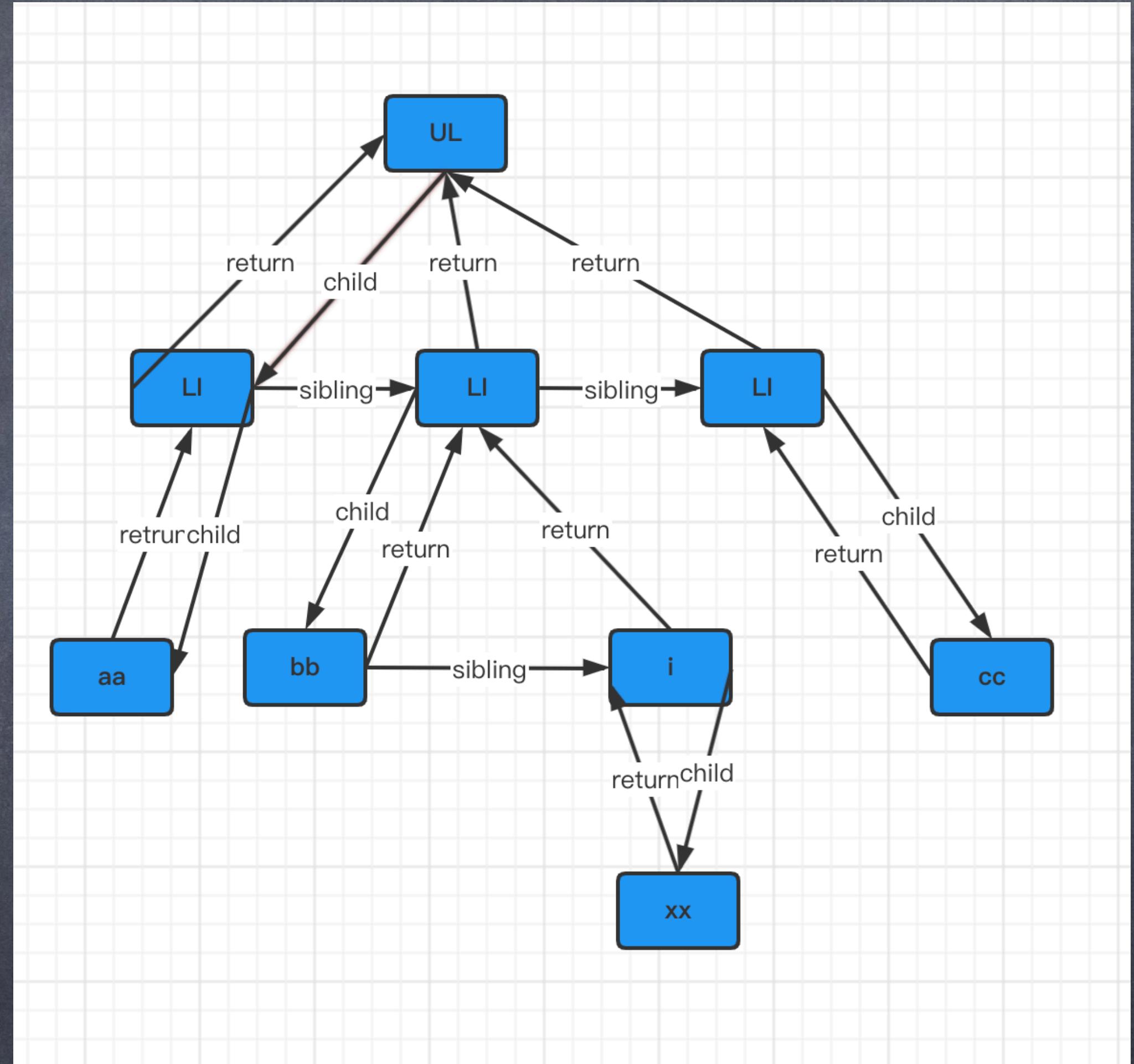
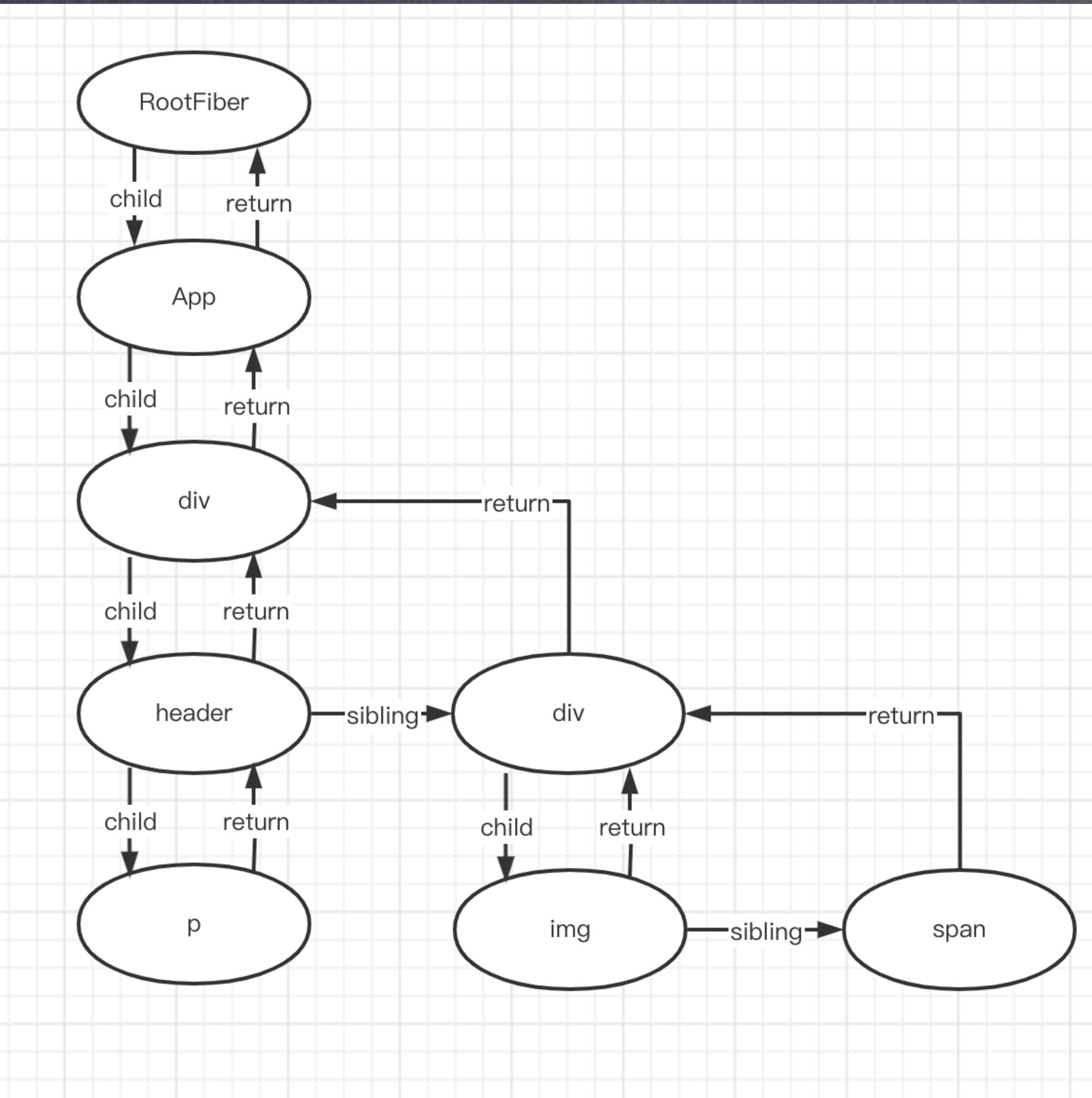
属性名	说明
type	节点类型，和虚拟Dom对象的type相同，用于区分元素、文本、组件
props	节点属性，同虚拟Dom对象
stateNode	节点Dom对象或者组件实例
tag	标记，用于标记节点
effects	存储包含自身和所有后代的Fiber数组
effectTag	标记当前节点需要进行的操作，包含插入、更新、移除等
parent	父Fiber对象，在React源码中叫Return
child	当前Fiber对象的子级Fiber对象
sibling	当前Fiber对象的下一级兄弟节点
alternate	Fiber对象备份，用于对比

```
react > fiber > ⚛ vdom.jsx > ...
1  const data = {
2    item1: 'bb',
3    item2: 'cc'
4  }
5
6  const jsx = <ul className="list">
7    <li className="item" style={{ background: 'blue', color: 'pink' }} onClick={() => alert(2)}>aa</li>
8    <li className="item">{data.item1}<i>xxx</i></li>
9    <li className="item">{data.item2}</li>
10 </ul>;
```

```
react > fiber > {} vdom.json > {} props > [ ] children > {} 2 > {} props > [ ] children > abc 0
```

```
1  {
2    "type": "ul",
3    "props": {
4      "className": "list",
5      "children": [
6        {
7          "type": "li",
8          "props": {
9            "className": "item",
10           "children": [
11             "aa"
12           ]
13         }
14       },
15       {
16         "type": "li",
17         "props": {
18           "className": "item",
19           "children": [
20             "bb"
21           ]
22         }
23       },
24       {
25         "type": "li",
26         "props": {
27           "className": "item",
28           "children": [
29             "cc"
30           ]
31         }
32       }
33     ]
34   }
35 }
```





- 将虚拟dom对象构建成Fiber对象
- 根据fiber对象渲染成真实dom

```
// 创建任务队列
const taskQueue = createTaskQueue()

// 空闲时间执行的具体方法
const performTask = deadline => {
  // 执行任务, workLoop方法后续补充
  workLoop(deadline)
  // 实现持续调用
  if (subTask || !taskQueue.isEmpty()) {
    requestIdleCallback(performTask)
  }
}

// 暴露的render方法
export const render = (element, dom) => {
  // 1. 添加任务 => 构建fiber对象
  taskQueue.push({
    dom,
    props: { children: element }
  })
  // 2. 指定浏览器空闲时间执行performTask 方法
  requestIdleCallback(performTask)
}
```

```
// 子任务
let subTask = null
// commit操作标志
let pendingCommit = null

const workLoop = deadline => {
    // 1. 构建根对象
    if (!subTask) {
        subTask = getFirstTask()
    }
    // 2. 通过while循环构建其余对象
    while (subTask && deadline.timeRemaining() > 1) {
        subTask = executeTask(subTask)
    }

    // 3. 执行commit操作，实现Dom挂载
    if (pendingCommit) {
        commitAllWork(pendingCommit)
    }
}
```

```
const getFirstTask = () => {
  // 获取任务队列中的任务
  const task = taskQueue.pop()

  // 返回Fiber对象
  return {
    props: task.props,
    stateNode: task.dom,
    // 代表虚拟Dom挂载的节点
    tag: "host_root",
    effects: [],
    child: null
  }
}
```

- ① react虚拟dom了解吗?
- ② 你了解react fiber吗?
- ③ Fiber的优势是什么?
- ④ Fiber怎么做到比之前的渲染要快的?
- ⑤ 你了解react虚拟dom渲染机制吗?

生命周期

16.0之前

- 初始化-挂载props-初始化state-render-完成
- 获取数据完毕-更新state-diff-render-完成

Initialization

setup props and state

Mounting

componentWillMount

render

componentDidMount

props

componentWillReceiveProps

shouldComponentUpdate

componentWillUpdate

render

componentDidUpdate

states

shouldComponentUpdate

componentWillUpdate

render

componentDidUpdate

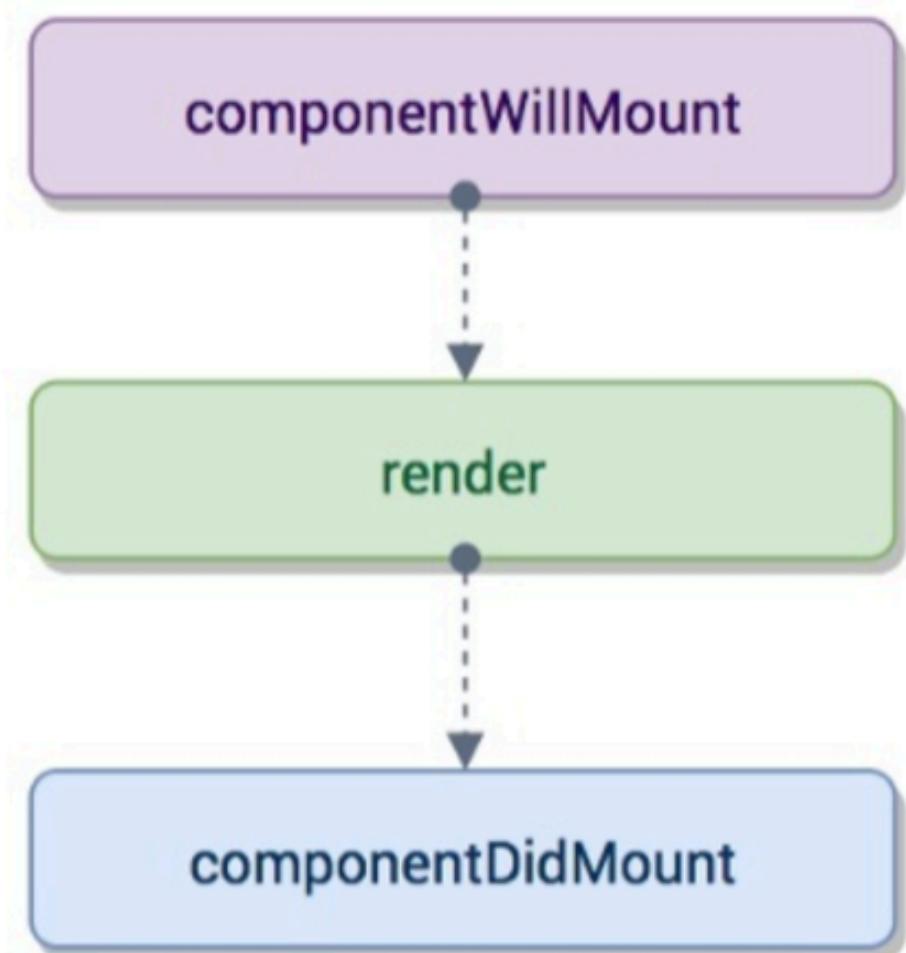
Unmounting

componentWillUnmount

react > life-cycle > JS initial.js > 📄 Test

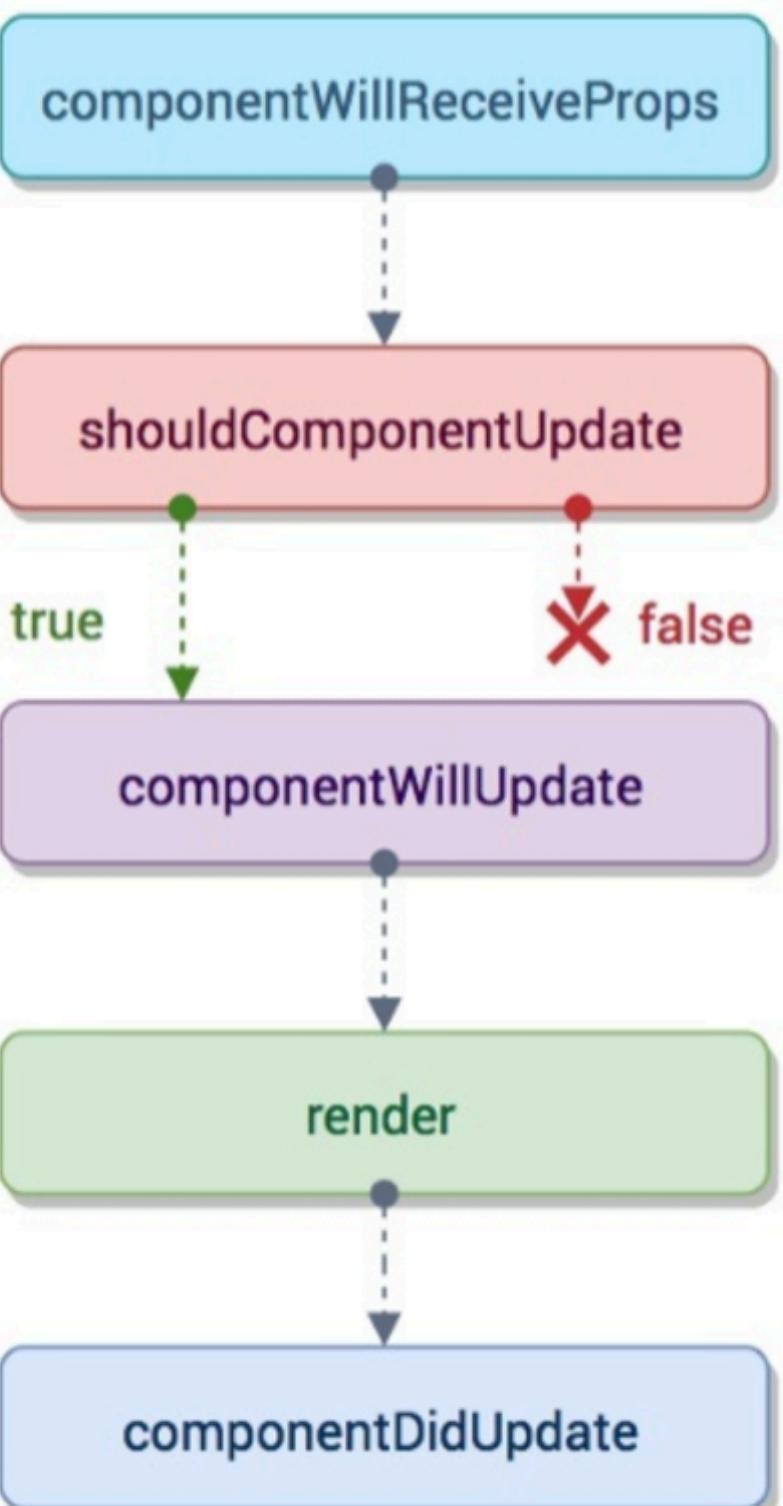
```
1 import React, { Component } from 'react';
2
3 class Test extends Component {
4   constructor(props) {
5     super(props);
6   }
7 }
```

Mounting

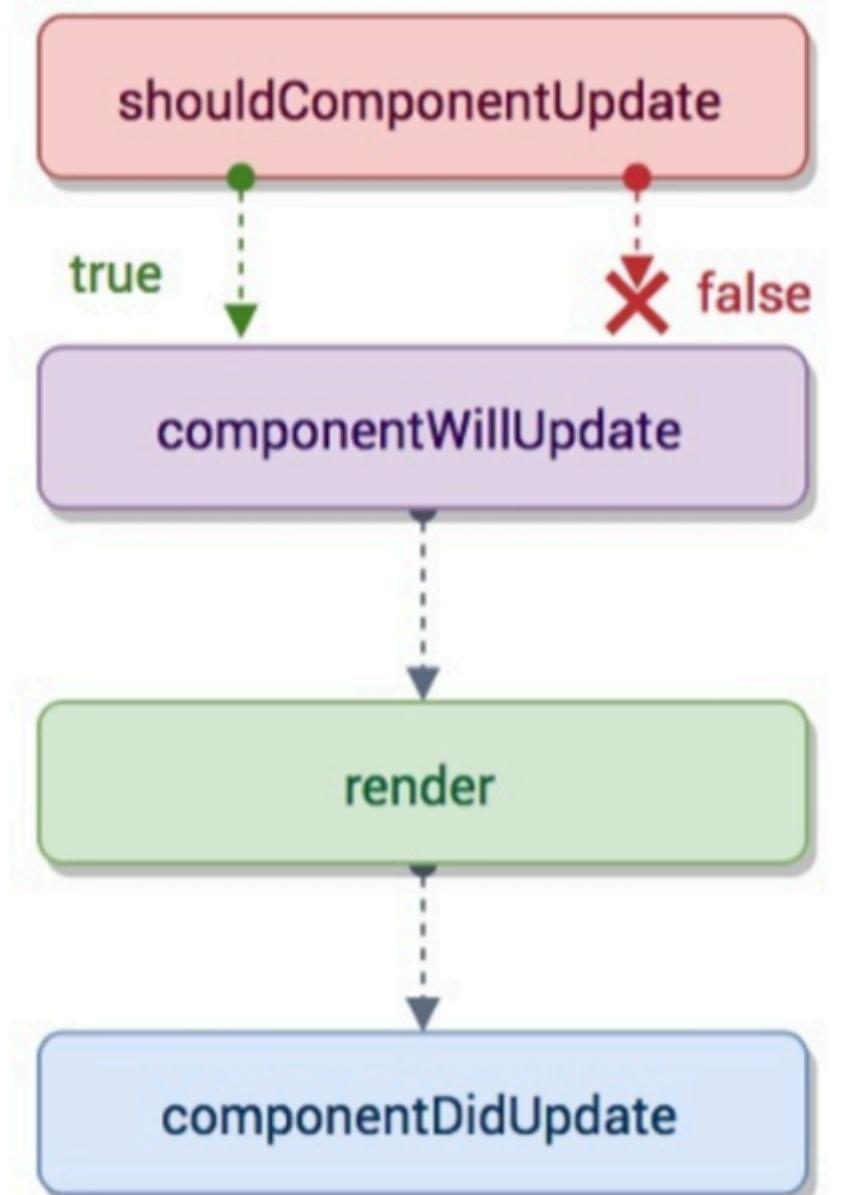


Updation

props



states



- ② 组件自己`setState`触发更新
- ③ 父组件触发传给子组件的`props`值变化，引发子组件更新

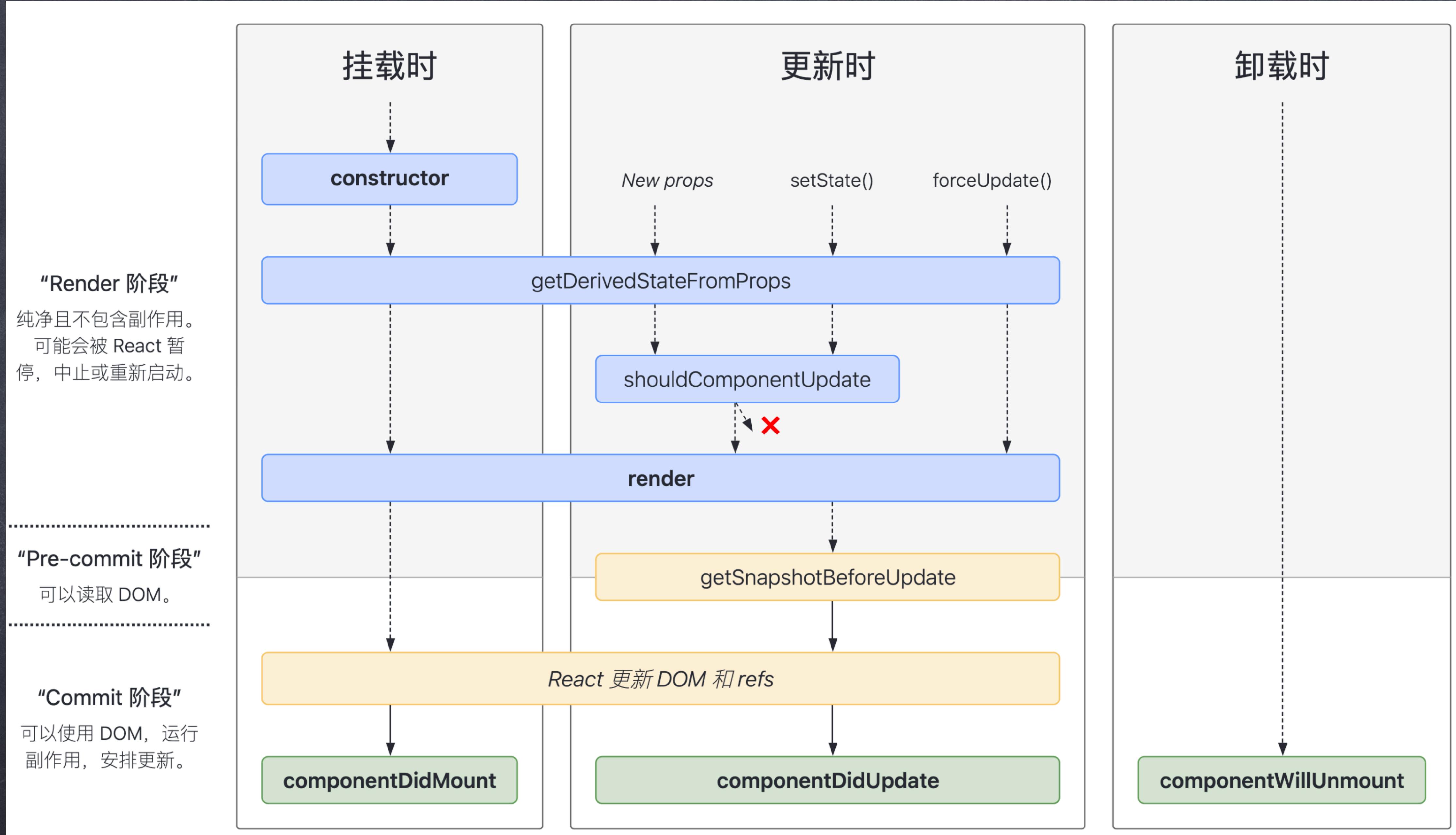
react > life-cycle > JS child2.js > SomeComponent > shouldComponentUpdate

```
1 // 父组件重新render, 重传props, 无论porps值有没有改变, 子组件都会接收并改变
2 class SomeComponent extends Component {
3     shouldComponentUpdate(nextProps) {
4         // 应该使用这个方法, 否则无论props是否有变化都将会导致组件跟着重新渲染
5         if (nextProps.someThings === this.props.someThings) {
6             return false;
7         }
8     }
9     render() {
10        return <div>{this.props.someThings}</div>;
11    }
12}
13
```

react > life-cycle > JS child.js > SomeComponent > componentWillReceiveProps

```
1  class SomeComponent extends Component {  
2      constructor(props) {  
3          super(props);  
4          this.state = {  
5              someThings: props.someThings  
6          };  
7      }  
8      componentWillReceiveProps(nextProps) {  
9          // 父组件重传props时就会调用这个方法  
10         this.setState({someThings: nextProps.someThings});  
11     }  
12     render() {  
13         return <div>{this.state.someThings}</div>  
14     }  
15 }
```

- componentWillMount
- componentWillReceiveProps
- componentWillUpdate
- shouldComponentUpdate



getDerivedStateFromProps(props, state)

`getDerivedStateFromProps` 会在调用 `render` 方法之前调用。

并且在初始挂载及后续更新时都会被调用。它应 `返回` 一个对象来更新 `state`，如果返回 `null` 则 `不更新` 任何内容。

getSnapshotBeforeUpdate(prevProps, prevState)

`getSnapshotBeforeUpdate()` 在最近一次渲染输出（提交到 DOM 节点）之前调用。

它使得组件能在发生更改之前从 DOM 中捕获一些信息（例如，滚动位置）。

此生命周期的任何返回值将作为参数传递给 `componentDidUpdate()`。

此用法并不常见，但它可能出现在 UI 处理中，如需要以特殊方式处理滚动位置的聊天线程等。

应返回 `snapshot` 的值（或 `null`），返回值会在 `componentDidUpdate(prevProps, prevState, snapshot)` 中的 `第三个参数`。

- React生命周期
- React生命周期函数以及各个函数的应用
- React 16.0前后生命周期函数的变化为什么
- React对于生命周期函数的变化你怎么理解
- React性能优化（虚拟dom渲染上优化）

diff算法

React diff算法策略

- 针对树结构(tree diff): 对UI层的DOM节点跨层级的操作进行忽略。 (数量少)
- 针对组件结构(component diff): 拥有相同类的两个组件生成相似的树形结构，拥有不同类的两个组件会生成不同的属性结构。
- 针对元素结构(element-diff): 对于同一层级的一组节点，使用具有唯一性的id区分 (key属性)

- 通过state计算出新的fiber节点
- 对比节点的tag和key确定节点操作（修改，删除，新增，移动）
- effectTag标记fiber对象
- 收集所有标记的fiber对象，形成effectList
- Commit阶段一次性处理所有变化的节点

```
// This API will tag the children with the side-effect of the reconciliation
// itself. They will be added to the side-effect list as we pass through the
// children and the parent.
//
function reconcileChildFibers(
  returnFiber: Fiber,
  currentFirstChild: Fiber | null,
  newChild: any,
  lanes: Lanes,
): Fiber | null {
  // This function is not recursive.
  // If the top level item is an array, we treat it as a set of children,
  // not as a fragment. Nested arrays on the other hand will be treated as
  // fragment nodes. Recursion happens at the normal flow.
```

旧: A

新: A

旧: A-B-C

新: B

旧: null

新: B

单节点diff, 针对以上三种情况
react处理成2种, 就看fibers链
是否为空, 为空视为新增, 不为空
视为更新。

旧: A-B-C-D

新: A-B-C-D

旧: A-B-C-D

新: A-B-C-D-E

旧: A-B-C-D

新: A-B-C

旧: A-B-C-D

新: A-B-D-C

旧: A-B-C-D-E

新: A-B-D-C

- ① diff原理和策略
- ② fiber diff的策略
- ③ react diff的优势

感谢

「狗哥」