



React原理核心技术点预备

1. 数据结构和算法相关

数据结构和算法相关

链表、双向链表

队列

深度优先、广度优先

字典

2. 前端开发架构相关

前端知识

性能渲染requestAnimationFrame

性能优化指标FCP、FP、FID、CLS等

位运算

私仓、TypeScript、Jest单元测试、pnpm原理、CI CD、WebPack、Rollup

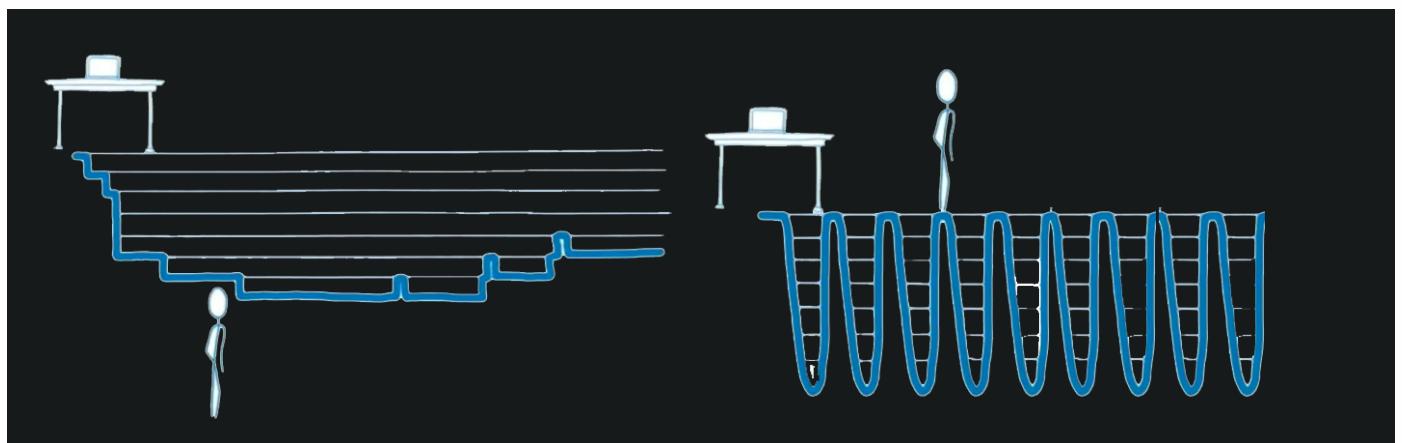
V8 Libuv原理

微任务宏任务等

3.Concurrent Mode（来源于Fiber架构）

React15 采用不可中断的递归方式更新的 Stack Reconciler (老架构)

React16 废弃~采用可中断的遍历方式更新的 Fiber Reconciler (新架构)
Fiber 的主要目标是通过异步渲染和更细粒度的任务调度来优化性能。



React 17 废弃了 React 16 中处理优先级采用的是 expirationTime 模型`。用了Lane

expirationTime 模型使用 expirationTime (一个时间长度) 来描述任务的优先级；而 Lane 模型则使用二进制数来表示任务的优先级：

Lane 模型提供了一个新的优先级排序的思路，相对于 **expirationTime** 来说，它对优先级的处理会更细腻，能够覆盖更多的边界条件。

React17带来了**Concurrent Mode**是一种提供更好用户体验的渲染模式，它允许React在处理大型和复杂应用时更灵活地分配和调度工作。

```
ReactDOM.render // sync 模式  
ReactDOM.createRoot(document.getElementById('root')).render() //  
concurrent 模式
```

React18继续废弃React17的开启方式，采用默认启动。但是请配合startTransition才能启动满血版本。

Concurrent Mode的实现涉及到以下几个方面：

1. 异步渲染：

- Concurrent Mode引入了异步渲染的概念，即将渲染任务分割为多个小任务，并使用调度器（Scheduler）来优先处理用户交互和高优先级任务。
- 异步渲染使得React能够以递增的方式对应用进行更新，并在每个更新阶段中尽快对用户提供反馈。

2. 调度器（Scheduler）：

- Concurrent Mode使用了一个全新的调度器，称为新的调度器（New Scheduler）。
- 新的调度器采用优先级调度算法，允许React根据任务的优先级和类型动态地安排任务的执行顺序。
- 调度器在Concurrent Mode中负责任务的调度和优先级排序，确保任务按照正确的优先级和顺序执行。

3. 任务优先级：

- Concurrent Mode引入了任务优先级的概念，使得React能够根据任务的紧迫程度和重要性来分配优先级。
- React为不同类型的任务（如用户交互、动画、数据更新等）赋予不同的优先级，确保紧急任务得到更快的响应和处理。

4. 时间切片 (Time Slicing) :

- 时间切片是Concurrent Mode的一个关键特性，它将渲染任务切分为一系列小的时间片段。
- 每个时间片段的执行时间被控制在一定范围内，确保不会阻塞主线程，使得浏览器能够及时响应用户输入和其他高优先级的任务。

通过这些机制的结合，Concurrent Mode使得React应用能够以更细粒度的方式进行渲染和更新，并提供更好的用户交互和响应能力。Concurrent Mode的实现使得React能够在处理复杂应用时，更好地平衡渲染和性能。需要注意的是，Concurrent Mode目前仍然处于实验性阶段，并且在React 18及以后版本中得到了更多的改进和优化。

```
import { useState, useEffect, useTransition, Suspense } from
'react';

// 模拟一个获取数据的异步函数
const fetchData = (resource: string): Promise<string> => {
  return new Promise(resolve =>
    setTimeout(() => resolve(`Data fetched: ${resource}`), 2000)
  );
}

const DataComponent: React.FC<{ resource: string }> = ({ resource }) => {
  const [data, setData] = useState<string | null>(null);

  useEffect(() => {
    fetchData(resource).then(fetchedData => {
      setData(fetchedData);
    });
  }, [resource]);

  if (!data) {
    throw new Error('Data is loading...');
  }

  return <div>{data}</div>;
}

const MyComponent: React.FC = () => {
```

```
const [resource, setResource] = useState<string>('initial resource');
const [startTransition, isPending] = useTransition();

const handleClick = (nextResource: string) => {
  startTransition(() => {
    setResource(nextResource);
  });
}

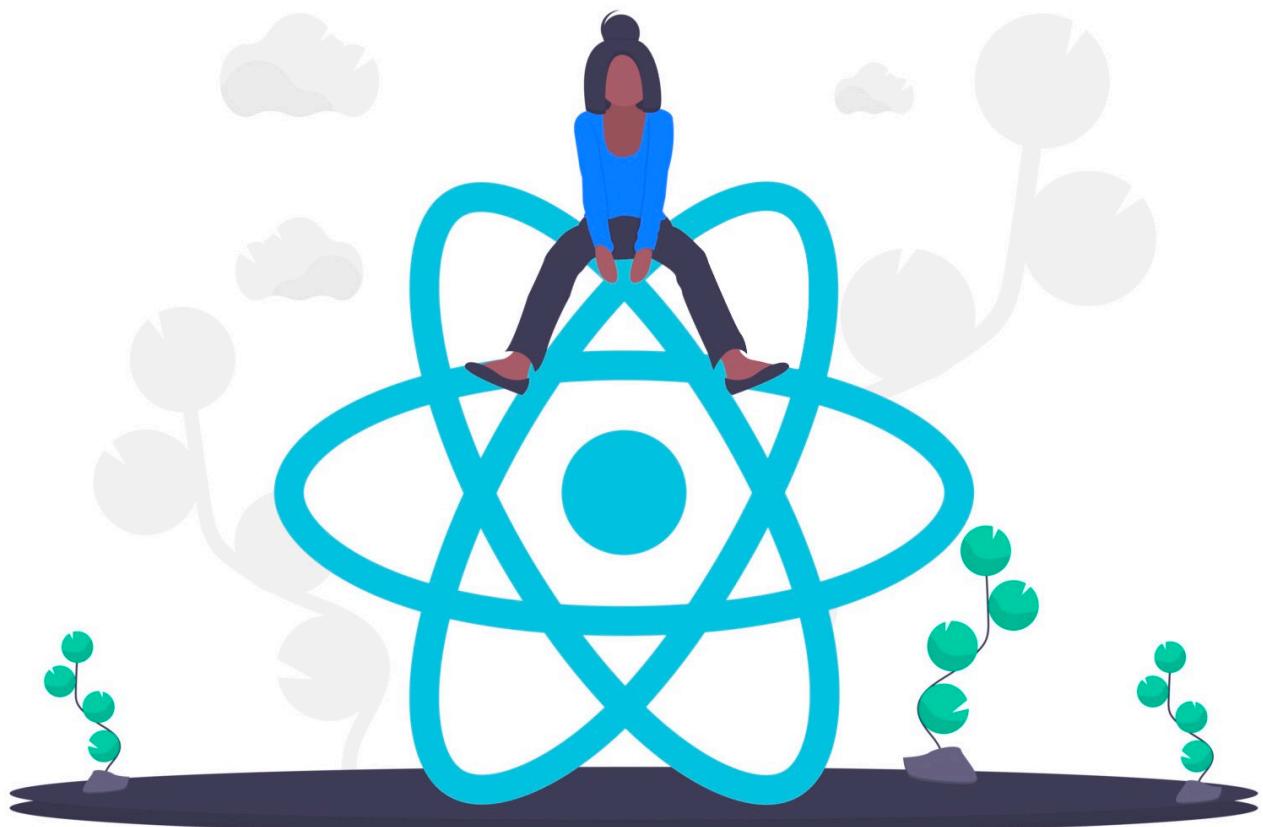
return (
  <div>
    <button onClick={() => handleClick('next resource')}
disabled={isPending}>
      Load Next Resource
    </button>
    <Suspense fallback={<h1>Loading...</h1>}>
      <DataComponent resource={resource} />
    </Suspense>
  </div>
);
}
```

总的来说，Fiber 是 React 内部的一种架构，用于管理组件的渲染；而 Concurrent Mode 是一种运行模式，它利用了 Fiber 的异步渲染能力，使得 React 可以在处理多个任务时进行更好的协调。

志佳老师



从0实现React16源码



React渲染阶段概念大全

1. 简化逻辑

1. createElement (JSX解析)
2. render方法；

3. Concurrent Mode (Fiber)

4. Fibers Node;

5. Render and Commit 阶段

6. Reconciliation

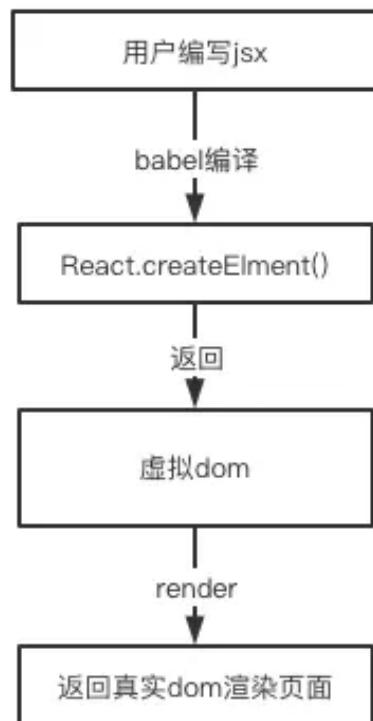
7. Function Components

8. hooks;

9. 类组件

总结 JSX -> FiberNode -> Concurrent Mode -> Fiber核心调度器 -> Scheduler -> 打断Reconciliation -> Render(执行Hooks) -> Commit Phases

2. 启航阶段 🚤



在React 17中，Facebook推出了一个新的JSX转换器：`@babel/plugin-transform-react-jsx`。它在编译时引入了`_jsx`和`_jsxs`两个新的函数，作为替代`React.createElement`的工具。这两个新的函数只在编译时使用，并不会出现在运行时代码中。

`_jsx` 和 `_jsxs` 与 `React.createElement` 相比做了以下优化：

1. **静态子元素的优化**: 在新的转换器中, 如果JSX元素有静态子元素, 那么它会被编译成 `_jsxs` 调用, 而非 `_jsx`。`_jsxs` 可以处理静态子元素的数组, 从而避免在每次渲染时创建新的数组, 以提高性能。
2. **key属性的优化**: 新的转换器能够在编译时分辨出静态和动态的key属性, 并将它们传递给 `_jsx` 或 `_jsxs` 函数。这样就避免了不必要的key属性计算, 提高了性能。
3. **开发模式的优化**: 新的转换器通过一个 `_source` 参数在开发模式下提供更多的调试信息。这个参数在生产模式下不会被包含, 以减小代码体积。
4. **减少了运行时的依赖**: 与 `React.createElement` 不同, `_jsx` 和 `_jsxs` 并不需要React库的运行时版本。这意味着, 在未来, React可以去除掉它的 `createElement` 函数, 从而减小库的体积。
5. **优化对象字面量**: 当 JSX 属性被编译为对象字面量时, 新的转换器可以在编译时标识出这些字面量, 从而避免在每次渲染时创建新的对象。

总的来说, 新的 `_jsx` 和 `_jsxs` 函数为JSX提供了更好的优化空间, 从而提高了性能并减小了代码体积。

3. 整体调度逻辑

- 初始化阶段:

创建根Fiber节点, 代表整个React应用的根组件。

调用根组件的render方法, 创建初始的虚拟DOM树。

- 调度阶段 (Scheduler) :

使用调度器 (Scheduler) 调度更新, 决定何时执行更新任务。

检查是否有高优先级任务需要执行, 如用户交互事件或优先级较高的异步操作。

根据优先级确定任务执行顺序, 并根据任务的优先级将任务添加到不同的任务队列 (lane) 中。

- 协调阶段 (Reconciliation) :

从任务队列中取出下一个任务。

对任务中涉及的组件进行协调, 比较前后两个虚拟DOM树的差异, 找出需要更新的部分。

使用DOM diff算法进行差异计算, 生成需要更新的操作指令。

- 生命周期阶段 (Lifecycle) :

在协调阶段和提交阶段，React会根据组件的生命周期方法调用相应的钩子函数。

生命周期方法包括componentDidMount、componentDidUpdate等，用于处理组件的生命周期事件。

- 5. 渲染阶段（Render）：

在Render阶段，React会根据组件的状态变化、props的更新或者父组件的重新渲染等触发条件，重新执行组件的函数体（函数组件）或者render方法（类组件）。

当React执行函数组件或render方法时，它会检测组件中是否包含了Hooks，如果包含了Hooks，那么React会根据Hooks的顺序依次调用它们。

Hooks执行：

在Render阶段，React会根据组件中Hooks的顺序，依次执行每个Hooks函数。

Hooks函数可能包括useState、useEffect、useContext等等，这些Hooks函数会在组件每次更新时被调用，让你能够在函数组件中使用状态、副作用和上下文等特性。

（批处理 setState 合并一次 setTimeout 16阶段不支持了 React18自动化批处理）

- Commit（提交）阶段：

- 在Commit阶段，React将Render阶段生成的更新应用到真实的DOM中，完成页面的渲染。

在Commit阶段，React可能会执行一些其他操作，比如调用生命周期方法（如componentDidMount、componentDidUpdate等）或执行其他副作用。

- 重复步骤2-7：

- 根据应用程序的交互和状态变化，React会重复执行调度、协调、渲染、提交的步骤，实现更新的循环流程。

4. 初始化渲染逻辑

首次渲染是指在React应用初始加载时进行的渲染过程。下面是首次渲染的流程：

1. 初始阶段：

- 创建根Fiber节点，代表整个React应用的根组件。

- 根据根组件的类型（函数组件或类组件），创建根Fiber节点的初始Fiber节点。
- 调用根组件的 `render` 方法（函数组件则执行函数体），生成初始的虚拟DOM树。

2. 渲染阶段：

- 根据初始的虚拟DOM树，React开始进行渲染阶段。
- React会遍历虚拟DOM树，递归创建组件的Fiber节点，并构建Fiber树。
- 在Fiber树的构建过程中，React会为每个组件创建对应的Fiber节点，建立组件之间的父子关系。

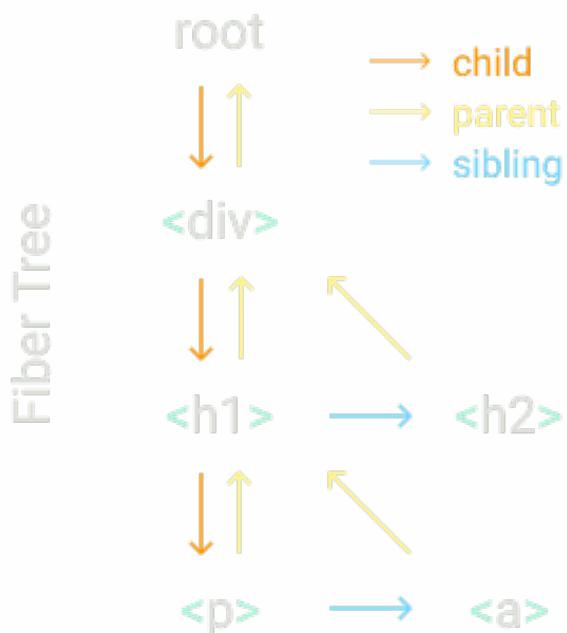
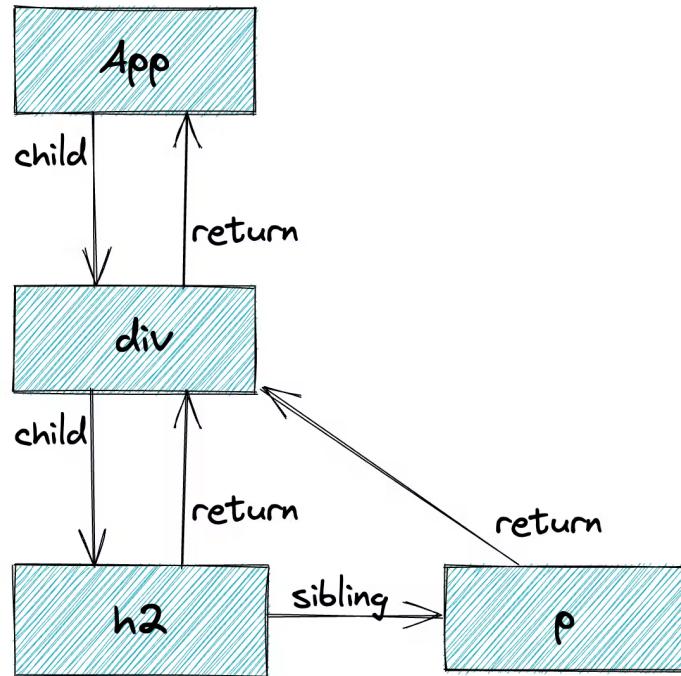
3. 提交阶段：

- 渲染阶段结束后，React进入提交阶段。
- React将Fiber树中的节点转换为真实的DOM节点，并将其插入到页面中，完成首次渲染。
- 在提交阶段，React还会触发一些生命周期方法，如 `componentDidMount`，以处理组件的挂载逻辑。

总结：首次渲染流程的关键步骤包括创建根Fiber节点、渲染阶段的Fiber树构建，以及提交阶段将虚拟DOM转换为真实DOM并插入页面。这个过程会触发组件的生命周期方法，完成组件的初始化和挂载。一旦首次渲染完成，后续的更新流程将进入到协调、更新和提交的循环中，以保持React应用的更新和渲染。

下一次的更新内存中挂载了一个虚拟的Fiber Tree <====> 真实的还有一棵树

5. Fiber Node



上图渲染过程详细描述如下：

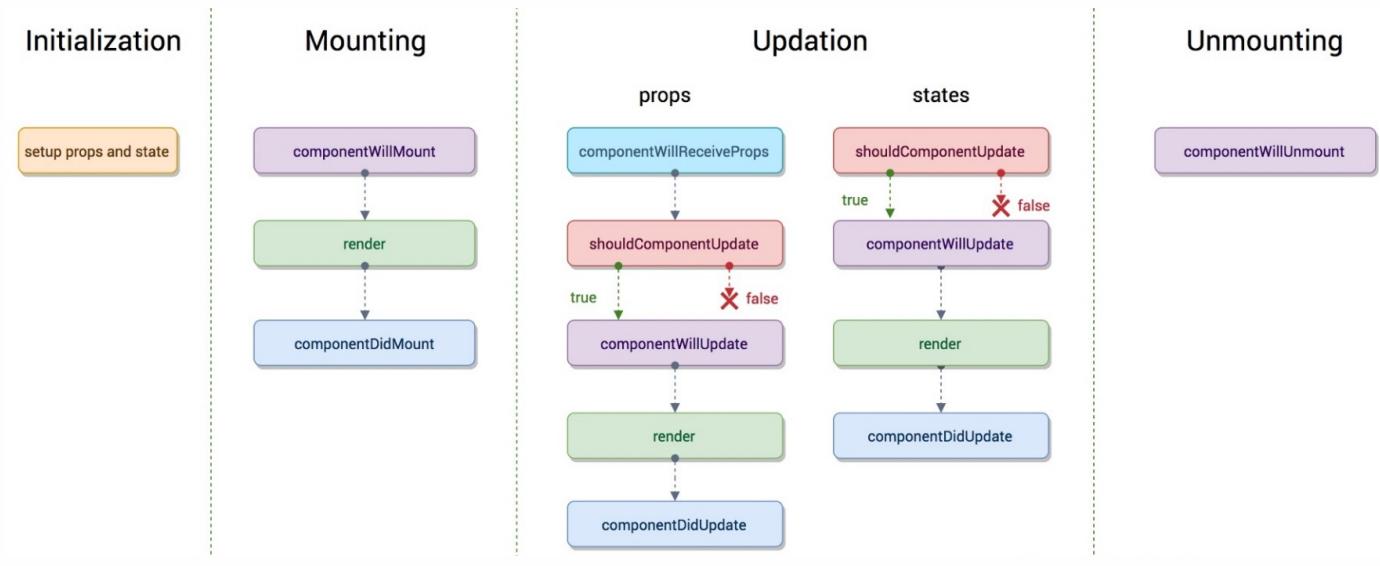
1. 当完成root的fiber工作时，如果有孩子，那么fiber是下一个工作的单元，root的子节点是div
2. 当完成div的fiber工作时，下一个工作单元是h1
3. h1的节点是p，继续下一个工作单元p
4. p没有子节点，去找兄弟节点a
5. a兄弟节点和子节点都没有，返回到父亲节点h1
6. h1的子节点都已经工作完成了，去找h1的兄弟节点h2

7. h2既没有兄弟节点，也没有子节点，返回到父节点div
8. 同上，div在返回到父节点root
9. 至此已经完成了所有的渲染工作

备注. 生命周期

新版 <https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>

老版：



志佳老师 2023年6月

react17.0 源码分析

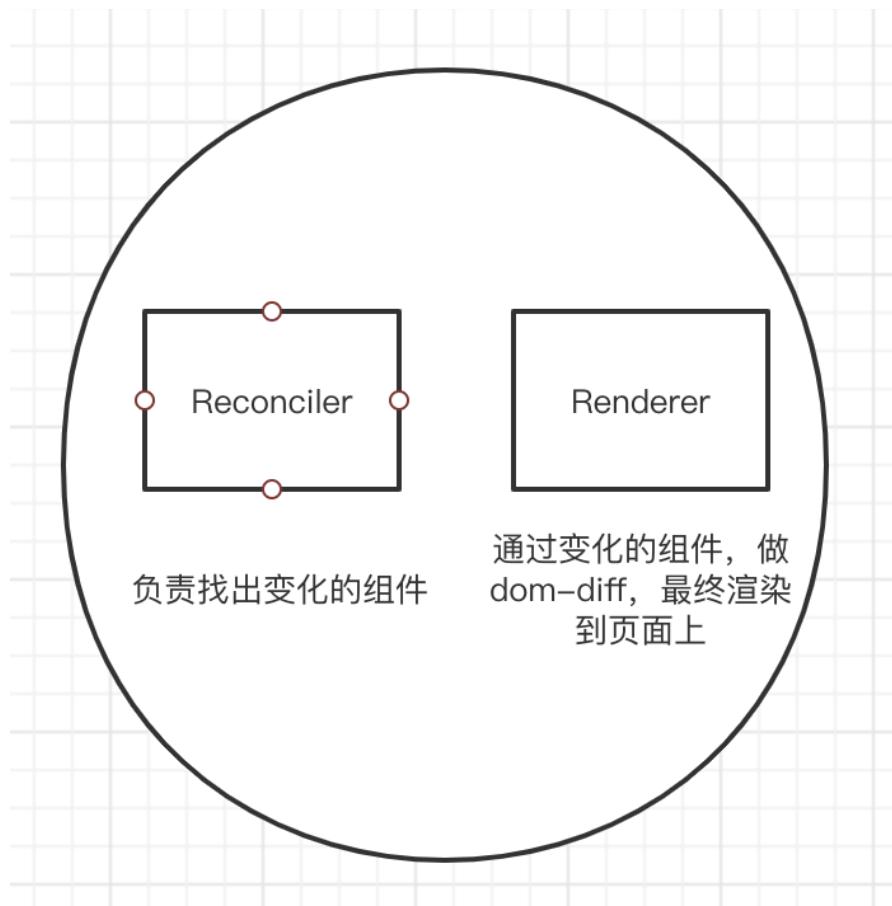
怎么样学习源码？

1. 要先掌握react的api的用法
2. 根据具体的某个api，通过debug来逐步的debug相关api的源码。
3. debug时，看不明白的，学会抓大放小。setState,只管主要的部分。
4. 至少先明白一个函数是做什么的，再弄懂具体怎么实现的。
5. 还看不明白的，可以看一些react相关的issue，别的开发者整理的文章。
6. 做一些简单的demo，来实际的运行看是怎么回事。

react16以前的整体架构

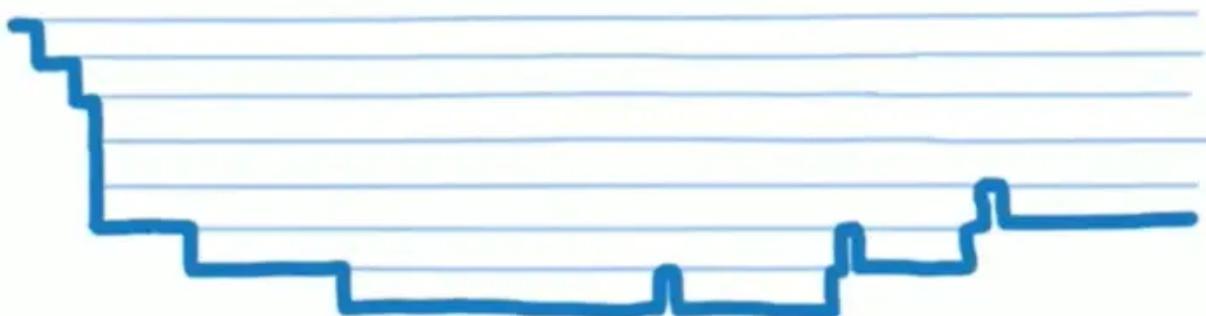
- a. Reconciler 找出更新的组件
- b. Renderer 渲染到页面

```
1 // 批处理的优化
2 componentDidMount(){
3   this.setState({
4     "a": ""
5 });
6   this.setState({
7     "b": ""
8 });
9 }
10 // 只发起一次更新
11 // react内部默认批处理
```



react15架构的缺点

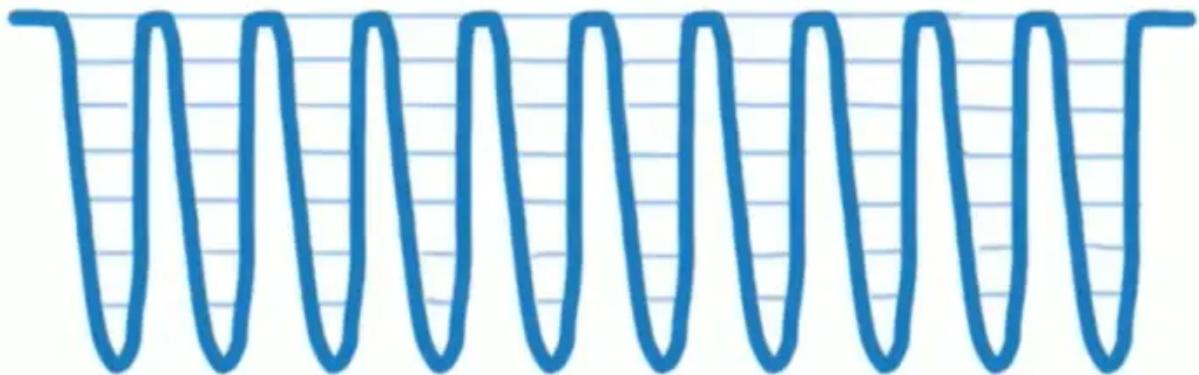
- c. react15架构递归的，一个长任务，会导致阻塞用户后续交互，会卡顿



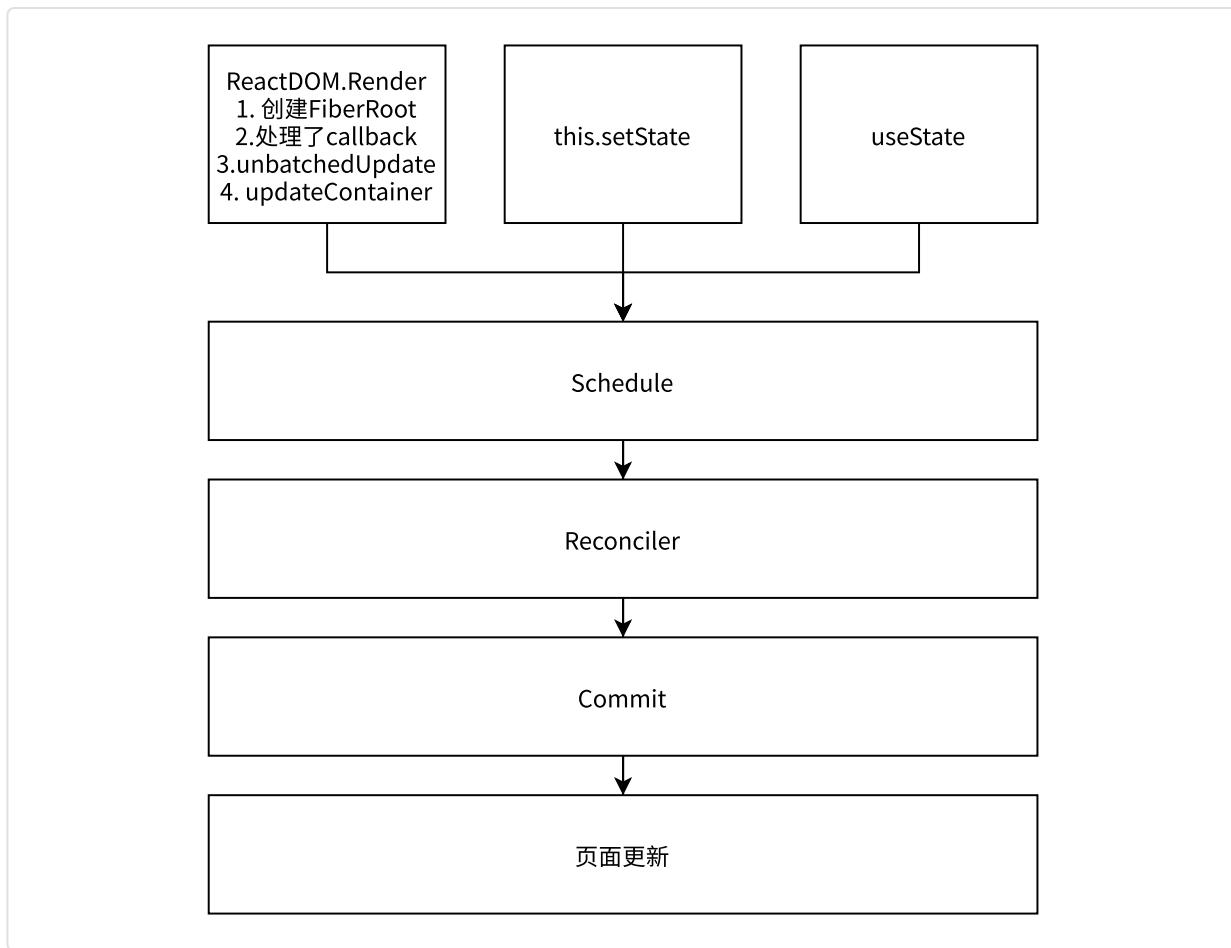
react16 Fiber架构的做法

异步调度

1. 执行异步的调度任务会在宏任务中执行，这样可以保证，不会让用户失去响应



2. 同时react16对所有的更新都做了一个优先级的绑定，当出现多个更新同时需要处理时，**可以中断低优先级更新，先执行高优先级的更新**。这就是——**Scheduler**模块，来调度任务的优先级。



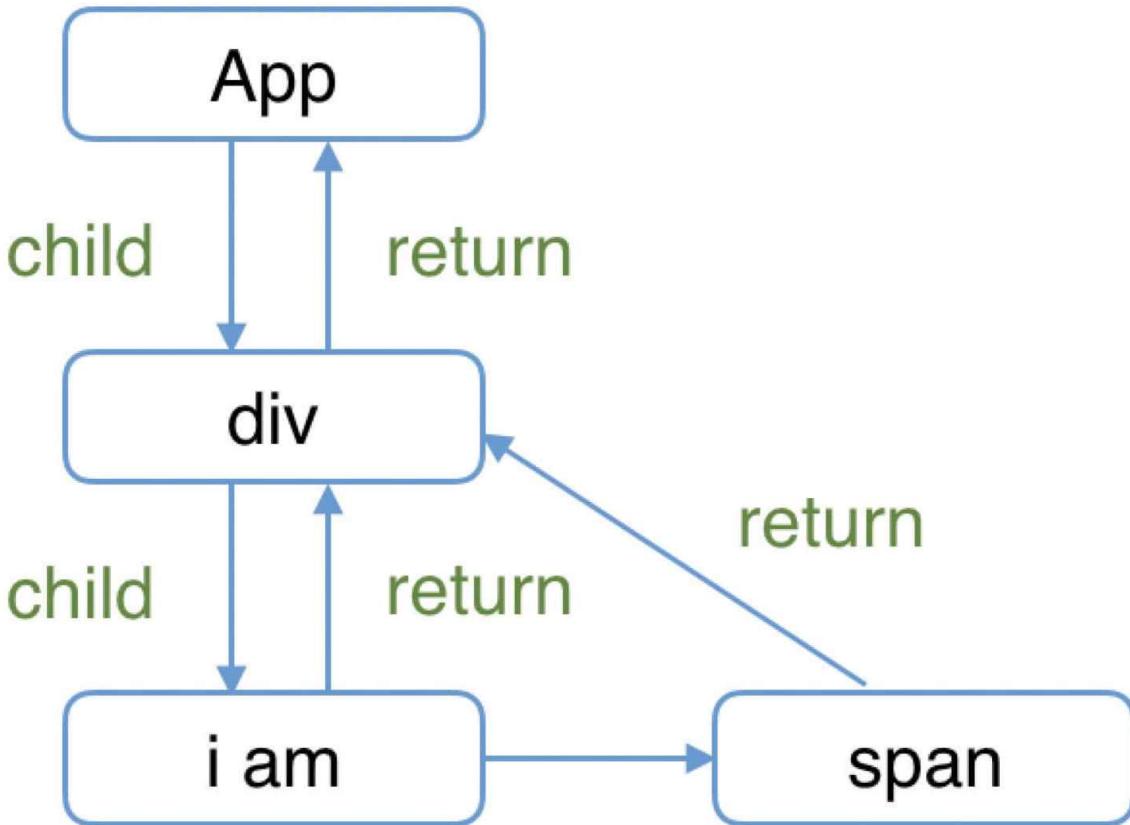
react里的优先级（和时间对应）

生命周期方法：同步执行

受控的用户输入：比如输入框内输入文字，同步执行

交互事件：比如动画，高优先级执行

其他：比如数据请求，低优先级执行



比如：

1. 当发起了一次优先级为C的更新，还没有更新完成，我们记为第一次更新。
2. 紧接着*iam*发起一次优先级为A的更新，记为第二次更新。优先级为A的更新会打断优先级为C的更新，先执行第二次更新，或者一起更新。A不更新，C肯定不会更新
3. 再对

节点发起一个优先级为A的更新，记为第三次更新。由于第一次更新的优先级会随着时间的流逝也会提升，如果这时第一次更新的优先级和第三次更新一致了，就会一起执行。否则，还是会先执行第三次更新，再执行第一次更新。但是由于优先级会随着时间变化，所以最后也还是会执行。

注意：

1. 什么是exprationTime?
2. 优先级会随着时间的变化而提升，当currentTime > exprationTime时，会以最高优先级执行
通过exprationTime比大小来对比优先级，exprationTime取决于任务的优先级+currentTime。
exprationTime会随着时间的变化而提升

JavaScript

- 1 react16的逻辑
- 2 优先级：
- 3 `this.setState({}) =>` 根据发起setState的情况，给不同的优先级 => 根据优先级计算 exprationTime
- 4 当前时间：unstable_ImmediatePriority 间隔数字 `exprationTime = currentTime + 间隔数字`

```
5  当前时间: unstable_NormalPriority 500 exprationTime = currentTime + 500
6  A 任务
7  B 任务
8  A.exprationTime > B.exprationTime
9
10 export const unstable_ImmediatePriority = 1;
11 export const unstable_UserBlockingPriority = 2;
12 export const unstable_NormalPriority = 3;
13 export const unstable_IdlePriority = 5;
14 export const unstable_LowPriority = 4;
15
16 // react把不同类型的任务分成不同的等级
17 function dispatchUserBlockingUpdate(
18   domEventName,
19   eventSystemFlags,
20   container,
21   nativeEvent,
22 ) {
23   runWithPriority(
24     UserBlockingPriority,
25     dispatchEvent.bind(
26       null,
27       domEventName,
28       eventSystemFlags,
29       container,
30       nativeEvent,
31     ),
32   );
33 }
34
35 expirationTime:
36 // expirationTime: 同任务的优先级和currentTime有关, 实际对比优先级就是对比两个任务
37 // 的expirationTime的大小
38 // 计算不同的过期时间
39 // 如果几个任务的expirationTime相同, 这几个任务会批量更新
40 // 任务之间的优先级就是比较expirationTime的大小
41 function computeExpirationForFiber(currentTime: ExpirationTime, fiber: Fiber) {
42   // var ImmediatePriority = 1; 最高优先级, 直接走messageChannel 直接处理
43   // var UserBlockingPriority = 2;
44   // var NormalPriority = 3; 默认
45   // var LowPriority = 4;
46   // var IdlePriority = 5;
47   // 优先级越高, ExpirationTime 超时时间越低
48   const priorityLevel = getCurrentPriorityLevel();
49
50   let expirationTime;
51   if ((fiber.mode & ConcurrentMode) === NoContext) {
52     // Outside of concurrent mode, updates are always synchronous.
```

```
53     // 在并发模式之外，更新始终是同步的。
54     expirationTime = Sync; //
55     // isWorking 在renderRoot或者CommitRoot
56     // isCommitting CommitRoot
57 } else if (isWorking && !isCommitting) {
58     // 在 renderRoot 阶段，优先级设置为当前正在工作的fiber的到期时间
59     // During render phase, updates expire during as the current render.
60     expirationTime = nextRenderExpirationTime;
61 } else {
62     // 在commit阶段，根据priorityLevel进行expirationTime更新
63     switch (priorityLevel) {
64         case ImmediatePriority:
65             // 立即执行
66             expirationTime = Sync; 2^31-1
67             break;
68         case UserBlockingPriority:
69             // 因用户交互阻塞的优先级
70             expirationTime = computeInteractiveExpiration(currentTime);
71             break;
72         case NormalPriority:
73             // 一般，默认优先级， 异步执行
74             // This is a normal, concurrent update
75             expirationTime = computeAsyncExpiration(currentTime);
76             break;
77         case LowPriority:
78         case IdlePriority:
79             // 低优先级或空闲状态
80             expirationTime = Never;
81             break;
82     default:
83         invariant(
84             false,
85             'Unknown priority level. This error is likely caused by a bug in ' +
86             'React. Please file an issue.',
87         );
88     }
89 }
90
91 /**
92 *
93 * @param {*} currentTime
94 * @param {*} expirationInMs 不同优先级任务会传不同的偏移量，把不同优先级的时间拉开一些差距
95 * @param {*} batchSizeMs batchSizeMs越大，批处理的间隔就越大
96 */
97 function computeExpirationBucket(
98     currentTime,
99     expirationInMs,
```

```
100     bucketSizeMs,
101  ): ExpirationTime {
102    return (
103      MAGIC_NUMBER_OFFSET -
104      ceiling(
105        MAGIC_NUMBER_OFFSET - currentTime + expirationInMs / UNIT_SIZE,
106        bucketSizeMs / UNIT_SIZE,
107      )
108    );
109 }
```

react17的扩展

expirationTime

在react 16中以expirationTime的大小来衡量优先级。当要进行优先级的比较的时候，通常就是比较两次update的expirationTime大小，当有高优先级的任务时，会将低优先级的任务给阻塞掉，先执行高优先级的。

相同的expirationTime会批处理。

不足

但Suspence出现后，出现了问题，高优先级IO任务会阻塞低优先级的cpu任务——expirationTime糅合了优先级和批处理

<https://codesandbox.io/s/demo-before-using-lane-model-txsnw?file=/src/index.js:1950-1958>

lanes

但随着react的功能迭代,react内部存在IO任务。

React 中 **IO 任务是指 Suspense 机制相关的任务，其他任务都是 CPU 任务**。如果一个任务会引起 Suspense 下子组件抛出 thenable 对象，那么它就是 IO 任务。

Suspense会等待接口返回后再执行后续的操作，所以是异步的。如果有一个高优先级的IO任务(Suspense)和低优先的CPU任务。那么按照**expirationTime** 模型，高优先级始终阻塞低优先级的。低优先级的任务就会一直等待高优先级的IO任务执行完毕才会执行低优先的CPU任务。导致页面卡顿
如何更好的处理高优先级io任务和低优先级cpu任务呢？

优先级和批处理分离

lane 表示优先级

lanes表示批处理多个优先级

lanes指定一个优先级区间，如果update的优先级在这个优先级区间内，则处理区间内包含的优先级生成对应页面快照。

lanes模型使用31位的二进制代表31种可能性。可以分别给IO任务，cpu任务不同的lane，最后可以并发的执行这几种类型的优先级。

从以前的只能执行1个优先级任务，到同时执行多个优先级任务。批处理和优先级分离

每发起一次更新：

1. React 会设置一个优先级
2. 根据优先级，去占用一个位置
3. 如果有新的更新，占用剩余的位置
4. 最终会把这个lanes区间的所有的更新一起处理

具体逻辑如下：

5. 发起update时，会指定优先级

JavaScript

```
1 <div onClick={}>
2     <div onFocus={}>dd</div>
3     <div onFocus={}>dd</div>
4 </div>
5
6 function dispatchUserBlockingUpdate(
7     domEventName,
8     eventSystemFlags,
9     container,
10    nativeEvent,
11 ) {
12     // lanes的优先级
13     setCurrentUpdateLanePriority(InputContinuousLanePriority);
14     // 在Scheduler会用到的优先级
15     runWithPriority(
16         UserBlockingPriority,
17         dispatchEvent.bind(
18             null,
19             domEventName,
20             eventSystemFlags,
21             container,
22             nativeEvent,
23         ),
24     );
25 }
26
27 // 用户操作可以和用户操作一起执行
28 case InputContinuousLanePriority:
29 case InputDiscreteLanePriority:
30 ...
31 return UserBlockingPriority;
32
33 // LowPriority 不是用户操作了
```

6. 通过优先级再去寻找lane，并占据一个位置，这个位置的优先级是右边的优先级更高

PHP

```
1 // 1. 每一个lane代表一个优先级
2 // 2. 右边的lane优先级比左边的lane优先级高
3 // 3. 按位运算比较的
4 // 4. 多个lane代表批处理多个任务
5 // 二进制的处理，按位处理
6
7 // 找到没有被占用的最后一位lane
8 function findUpdateLane(
9     lanePriority: LanePriority,
10    wipLanes: Lanes,
11 ): Lane {
12     switch (lanePriority) {
13         ...
14         case SyncLanePriority:
15             return SyncLane;
16         case SyncBatchedLanePriority:
17             return SyncBatchedLane;
18         case InputDiscreteLanePriority: {
19             // wipLanes 0b00000000000000000000000000000000 当前更新需要处理的哪些优先级
20             // ~wipLanes =          0b11111111111111111111111111111111
21             // InputDiscreteLanes = 0b00000000000000000000000000000000
22             // InputDiscreteLanes & ~wipLanes = 0b0000000000000000000000000000000011000
23             // lanes & -lanes
24             // lanes = 0b0000000000000000000000000000000011000
25             // (InputDiscreteLanes & ~wipLanes) & -(InputDiscreteLanes & ~wipLanes)
26             const lane = pickArbitraryLane(InputDiscreteLanes & ~wipLanes);
27             // InputDiscreteLanes的两个lane都被占据了
28             if (lane === NoLane) {
29                 // Shift to the next priority level
30                 return findUpdateLane(InputContinuousLanePriority, wipLanes);
31             }
32             return lane;
33         }
34         case InputContinuousLanePriority: {
35             // 第一个更新
36             // InputContinuousLanes = 0b0000000000000000000000000000000011000000
37             // wipLanes—已经被占用的lane 0b00000000000000000000000000000000
38             // 第二个更新
39             // InputContinuousLanes = 0b0000000000000000000000000000000011000000
40             // wipLanes—已经被占用的lane 0b00000000000000000000000000000000100000000
41             // 第3个更新
42             // wipLanes—已经被占用的lane 0b000000000000000000000000000000001100000000
43             // 最终可能:
44             // wipLanes — 0b0000000000000000000000000000000011111000000000
45             // 其他情况...  
.....
```

```

45     // 先执行: InputContinuousLanes & ~wipLanes
46     // 再执行: pickArbitraryLane = (lanes) => (lanes & -lanes)
47     const lane = pickArbitraryLane(InputContinuousLanes & ~wipLanes);
48     // wipLanes是现在已经被占用的lane, lane === NoLane 说明InputContinuousLanes的
两个lane都被占据了
49     if (lane === NoLane) {
50         // Shift to the next priority level
51         return findUpdateLane(DefaultLanePriority, wipLanes);
52     }
53     return lane;
54 }
55 case DefaultLanePriority: {
56     let lane = pickArbitraryLane(DefaultLanes & ~wipLanes);
57     if (lane === NoLane) {
58         // If all the default lanes are already being worked on, look for a
59         // lane in the transition range.
60         lane = pickArbitraryLane(TransitionLanes & ~wipLanes);
61         if (lane === NoLane) {
62             // All the transition lanes are taken, too. This should be very
63             // rare, but as a last resort, pick a default lane. This will have
64             // the effect of interrupting the current work-in-progress render.
65             lane = pickArbitraryLane(DefaultLanes);
66         }
67     }
68     return lane;
69 }
70 ...
71 }
72 }
73
74 // pickArbitraryLane最终会调到getHighestPriorityLane, 当存在多个可选择的lane时, 从后
往前取
75 // 负数以原码的补码形式表示
76 // lanes: 0b0000000000000000000000000000000011000000
77 // -lanes: 原:0b1000000000000000000000000000000011000000 =>
78 //       反: 0b1111111111111111111111110111111 + 1 =
79 //       补: 反 + 1 = 0b111111111111111111111111101000000
80 function getHighestPriorityLane(lanes: Lanes) {
81     return lanes & -lanes;
82 }
83 0b0000000000000000000000000000000011000
84 // InputDiscreteLanes = 0b0000000000000000000000000000000011000000
85 // wipLanes = 0b0000000000000000000000000000000000000000000000000000000000
86 // InputDiscreteLanes & ~wipLanes = 0b0000000000000000000000000000000011000000
87 // -lanes = 负数以原码的补码形式表示
88 //((InputDiscreteLanes & ~wipLanes) & -(InputDiscreteLanes & ~wipLanes))
89

```

Scheduler (调度器)

调度任务的优先级，控制那些任务优先进入 Reconciler

了解react里的优先级（和时间对应）

- 生命周期方法：同步执行
- 受控的用户输入：比如输入框内输入文字，同步执行
- 交互事件：比如动画，高优先级执行
- 其他：比如数据请求，低优先级执行

大致调度逻辑

Fiber的结构

container._reactRootContainer => root

root._internalRoot => fiberRoot

FiberRoot => FiberNode:

```
1 function FiberNode(){
2   // Instance 作为静态数据结构的属性
3   // Fiber对应组件的类型 Function/Class/Host/Fregment/Portail
4   this.tag = tag;
5   // 标志你的这个节点的唯一性,通过key值, 复用节点, 替换return, sibling, child。
6   this.key = key;
7   this.elementType = null;
8   // 对于 FunctionComponent, 指函数本身, 对于ClassComponent, 指class, 对于
9   // HostComponent, 指DOM节点tagName。
10  this.type = null;
11  // Fiber对应的真实DOM节点 <div>dfdf</div>
12  this.stateNode = null;
13  // Fiber 用于连接其他Fiber节点形成Fiber树
14  this.return = null;
15  this.child = null;
16  this.sibling = null;
17  // 保存本次更新造成的状态改变相关信息
18  this.pendingProps = pendingProps;
19  this.memoizedProps = null;
20  this.updateQueue = null;
21  // 存放hooks
22  this.memoizedState = null;
23  this.dependencies = null;
24  // SideEffects, 标志更新的类型。删除, 新增, 更改属性
25  this.flags = NoFlags;
26  this.subtreeFlags = NoFlags;
27  this.deletions = null;
28  // 调度优先级相关,以前这里是expirationTime.较之于优先级系统,有两个优势:
29  // 1. 即较高优先级的IO约束任务会阻止较低优先级的CPU约束任务无法完成
```

```

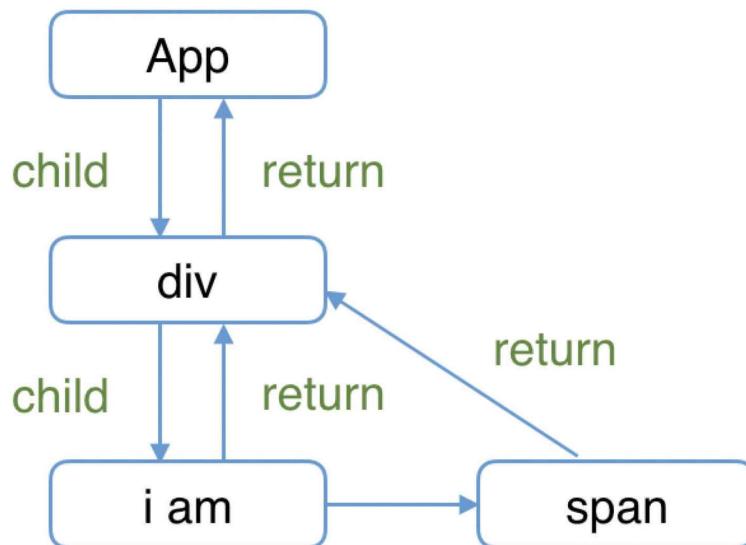
29 // 2. 能代表有限的一组的多个不同任务
30 // 详情看这个文档: https://github.com/facebook/react/pull/18796
31 this.lanes = NoLanes;
32 // this.expirationTime = null;
33 this.childLanes = NoLanes;
34 // 指向 workInProgress Fiber, 其实就是上一次构建的Fiber镜像。
35 this.alternate = null;
36 }

```

Fiber Tree

可以看到Fiber 与 Fiber之间是以链表的形式来连接的。

注：这种结构可以方便中断



大致调度逻辑

- 根据优先级区分同步任务和异步任务，同步任务立即同步执行在主线程执行的，最快渲染出来。异步任务走scheduler——宏任务里执行的。（ReactDOM.render 主线程执行） unBatchedUpdate

JavaScript

```

1 // 不要做批处理
2 unBatchedUpdate(() => {
3     this.setState({a: '122'});
4 });
5 executionContext = LegacyUnbatchedContext;

```

- 计算得到expirationTime, expirationTime = currentTime(当前时间) + timeout (不同优先级的时间间隔，时间越短，优先级越大)
- 对比startTime和currentTime，将任务分为及时任务和延时任务。
- 及时任务当即执行调度

5. 延时任务需要等到currentTime >= startTime的时候才会执行调度。
6. 及时任务执行完后，也会去判断是否有延时任务到了该执行之时，如果是，就执行延时任务
7. 每一批任务的执行在不同的宏任务中，不阻塞页面用户的交互。

调度具体代码分析

1. 根据优先级区分同步任务和异步任务，同步任务立即同步执行，最快渲染出来。异步任务走scheduler

基本概念：

JavaScript

```

1 SyncLane: 同步优先级，这个优先级是最高的，同步处理
2 executionContext: 执行上下文。由于react的整个处理周期很长，
3 为了较好的区分任务进行到哪个阶段，用executionContext来标志。
4 由于executionContext可能同时存在多种，为了方便计算是按位运算的
5
6     export const NoContext = /*          */ 0b00000000;
7     const BatchedContext = /*          */ 0b00000001;
8     const EventContext = /*          */ 0b00000010;
9     const DiscreteEventContext = /*      */ 0b00000100;
10    const LegacyUnbatchedContext = /*    */ 0b00010000;
11    const RenderContext = /*          */ 0b00100000;
12    const CommitContext = /*          */ 0b01000000;
13    export const RetryAfterError = /*    */ 0b10000000;
14
15    function RenderRoot (){
16        const prevExecutionContext = executionContext;
17        executionContext |= RenderContext;
18        fn();
19        executionContext = prevExecutionContext;
20    }
21
22    function CommitRoot (){
23        const prevExecutionContext = executionContext;
24        executionContext |= CommitContext;
25        fn();
26        executionContext = prevExecutionContext;
27    }
28

```

```

1 export function scheduleUpdateOnFiber(
2   fiber: Fiber,
3   lane: Lane,
4   eventTime: number,
5 ) {

```

```
6 ...
7
8 // 获得当前更新的优先级
9 const priorityLevel = getCurrentPriorityLevel();
10 // 同步任务，立即更新
11
12 // renderRoot的方法 => executionContext |= RenderContext
13 // ReactDOM.render() === vue().mount();
14 // 在unbatchedUpdates 执行回调 设置executionContext= LegacyUnbatchedContext;
15 if (lane === SyncLane) {
16   if (
17     // 处于unbatchedUpdates， 且不在Renderer渲染阶段， 立即执行
18     // Check if we're inside unbatchedUpdates
19     // executionContext = LegacyUnbatchedContext 执行上下文
20     // unbatchedUpdates: 非批处理 ReactDOM.render()
21     // batchedUpdates: 批处理
22     // 之前肯定调用过unbatchedUpdates
23     // executionContext === LegacyUnbatchedContext 这个也是位运算
24     (executionContext & LegacyUnbatchedContext) !== NoContext &&
25     // Check if we're not already rendering
26     // CommitContext: 渲染到页面的那个阶段
27     // RenderContext Render阶段
28     // LegacyUnbatchedContext 处于UnbatchUpdate之下
29     (executionContext & (RenderContext | CommitContext)) === NoContext
30   ) {
31     // Register pending interactions on the root to avoid losing traced
32     // interaction data.
33     schedulePendingInteractions(root, lane);
34     // This is a legacy edge case. The initial mount of a ReactDOM.render-ed
35     // root inside of batchedUpdates should be synchronous, but layout updates
36     // should be deferred until the end of the batch.
37     // 不会在宏任务中执行，在主线程中处理
38     performSyncWorkOnRoot(root);
39   } else {
40     ...
41     // 包含异步调度逻辑，和中断逻辑
42     ensureRootIsScheduled(root, eventTime);
43   }
44   ...
45   // Schedule other updates after in case the callback is sync.
46   ensureRootIsScheduled(root, eventTime);
47   schedulePendingInteractions(root, lane);
48 }
49 mostRecentlyUpdatedRoot = root;
50 }
51
52 // A: ensureRootIsScheduled root.callbackNode已被赋值
53 // B: ensureRootIsScheduled existingCallbackNode = root.callbackNode 是否存在
54 function ensureRootIsScheduled(root: FiberRoot, currentTime: number) {
```

```

55 // root.callbackNode的存活周期是从ensureRootIsScheduled开始—>到commitRootImpl截止
56 const existingCallbackNode = root.callbackNode;
57 // 检查是否存在现有任务。 我们也许可以重用它。
58 // Check if there's an existing task. We may be able to reuse it.
59 if (existingCallbackNode !== null) {
60     const existingCallbackPriority = root.callbackPriority;
61     // 优先级没有改变。 我们可以重用现有任务，现有任务的优先级和下一个任务的优先级相同。比如
62     // input连续的输入，优先级相同，可以执行用之前的任务
63     // 由于获取更新是从root开始，往下找到在这个优先级内的所有update.
64     // 比如存在连续的setState，会执行这个逻辑，不会新建一个新的update
65     // this.setState({
66     //   "a": ""
67     // });
68     // this.setState({
69     //   "b": ""
70     // });
71     // 不需要重新发起一个调度，用之前那个就可以了
72     if (existingCallbackPriority === newCallbackPriority) {
73         // The priority hasn't changed. We can reuse the existing task. Exit.
74         return;
75     }
76     // 优先级变了，先cancel掉，后续重新发起一个，// 中断逻辑
77     // The priority changed. Cancel the existing callback. We'll schedule a new
78     // one below.
79     cancelCallback(existingCallbackNode);
80 }
81 // Schedule a new callback.
82 // 发起一个新callBack
83 let newCallbackNode;
84 ...
85 // 新建一个scheduleCallback
86 newCallbackNode = scheduleCallback(
87     schedulerPriorityLevel,
88     performConcurrentWorkOnRoot.bind(null, root),
89 );
90 root.callbackPriority = newCallbackPriority;
91 // root.callbackNode的存活周期是从ensureRootIsScheduled开始—>到commitRootImpl截止
92 root.callbackNode = newCallbackNode;
93 }

```

2. 计算得到expirationTime, expirationTime = currentTime(当前时间) + timeout (不同优先级的时间间隔，时间越短，优先级越大)

```

1 var currentTime = getCurrentTime();
2 // 得到startTime，根据优先级的不同分别加上不同的间隔时间，构成expirationTime；当
3 // expirationTime越接近真实的时间，优先级越高
4 // 根据startTime 是否大于当前的currentTime，将任务分为了及时任务和延时任务。延时任务还不
5 // 会立即执行，它会在currentTime接近startTime的时候，才会执行
6 var startTime;

```

```

5  if (typeof options === 'object' && options !== null) {
6    var delay = options.delay;
7    if (typeof delay === 'number' && delay > 0) {
8      startTime = currentTime + delay;
9    } else {
10      startTime = currentTime;
11    }
12  } else {
13    startTime = currentTime;
14  }
15  var timeout;
16  // 根据优先级增加不同的时间间隔
17  switch (priorityLevel) {
18    // ImmediatePriority = -1;
19    case ImmediatePriority:
20      timeout = IMMEDIATE_PRIORITY_TIMEOUT;
21      break;
22    case UserBlockingPriority:
23      timeout = USER_BLOCKING_PRIORITY_TIMEOUT;
24      break;
25    case IdlePriority:
26      timeout = IDLE_PRIORITY_TIMEOUT;
27      break;
28    case LowPriority:
29      timeout = LOW_PRIORITY_TIMEOUT;
30      break;
31    case NormalPriority:
32    default:
33      timeout = NORMAL_PRIORITY_TIMEOUT;
34      break;
35  }
36  var expirationTime = startTime + timeout;
37
38 // expirationTime <= currentTime: 如果到这个时候, 还没有执行; 就立即执行

```

3. 对比startTime和currentTime，将任务分为及时任务和延时任务

```

1 if (startTime > currentTime) {
2   push(timerQueue, newTask);
3   // 当没有及时任务的时候
4   // Schedule a timeout.
5   // 在间隔时间之后, 调用一个handleTimeout, 主要作用是把timerQueue的任务加到
6   // taskQueue队列里来, 然后调用requestHostCallback
7   // 执行那个延时任务
8   // setTimeOut(handleTimeout, startTime - currentTime)
9   requestHostTimeout(handleTimeout, startTime - currentTime);
10 }
11 } else {
12   push(taskQueue, newTask);
13   // Schedule a host callback, if needed. If we're already performing work,

```

```

13     // wait until the next time we yield.
14     if (!isHostCallbackScheduled && !isPerformingWork) {
15         isHostCallbackScheduled = true;
16         // 这里会调度及时任务
17         requestHostCallback(flushWork);
18     }
19 }
20 return newTask;
21 }

```

4. 及时任务当即执行,但是为了不阻塞页面的交互,因此在宏任务中执行

```

1 // channel.port2.postMessage() => 会在宏任务里执行channel.port1.onmessage
2 const channel = new MessageChannel();
3 const port = channel.port2;
4 channel.port1.onmessage = performWorkUntilDeadline;
5 // a.port1.postMessage({}) => a.port2.onmessage 的回调会执行
6 // 在MessageChannel宏任务里执行真正的调度逻辑,可以保证任务与任务之间不是连续执行的,这样就
7 // 不会因为要一次性执行的任务多而阻塞用户的操作
8 requestHostCallback = function(callback) {
9     scheduledHostCallback = callback;
10    if (!isMessageLoopRunning) {
11        isMessageLoopRunning = true;
12        // 会在宏任务里执行performWorkUntilDeadline
13        port.postMessage(null);
14        // 在主线程执行performWorkUntilDeadline();
15        // performWorkUntilDeadline();
16        // setTimeout(() => {
17        //     // 进入宏任务
18        //     // }, 0)
19    }
20 };

```

5. 延时任务需要等到currentTime >= expirationTime的时候才会执行。每次调度及时任务的时候,都会去判断延时任务的执行时间是否到了,如果判断为true,则添加到及时任务中来。

```

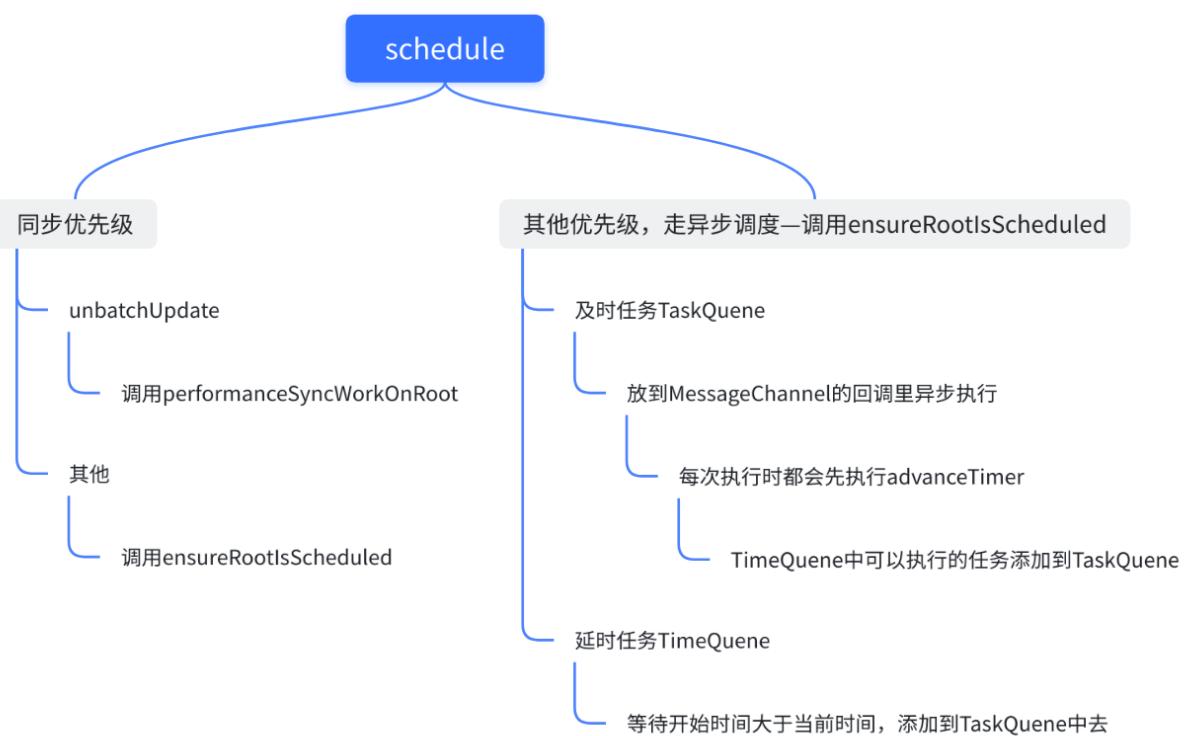
1 function advanceTimers(currentTime) {
2     // Check for tasks that are no longer delayed and add them to the queue.
3     let timer = peek(timerQueue);
4     while (timer !== null) {
5         if (timer.callback === null) {
6             // Timer was cancelled.
7             pop(timerQueue);
8         } else if (timer.startTime <= currentTime) {
9             // Timer fired. Transfer to the task queue.
10            pop(timerQueue);
11            timer.sortIndex = timer.expirationTime;

```

```

12     // 把timerQueue中的添加到taskQueue中来
13     push(taskQueue, timer);
14     if (enableProfiling) {
15         markTaskStart(timer, currentTime);
16         timer.isQueued = true;
17     }
18 } else {
19     // Remaining timers are pending.
20     return;
21 }
22 timer = peek(timerQueue);
23 }
24 }

```

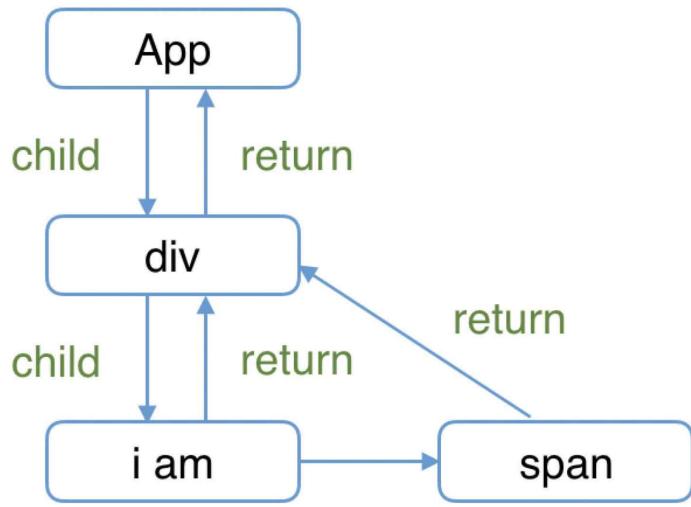


Reconciler (协调器)

负责构建fiber tree, 找出变化的组件, 标记组件的变化。

大致的找出变化的组件的逻辑

react发生一次更新的时候, 比如ReactDOM.render/setState, 都会从Fiber Root开始从上往下遍历, 然后逐一找到变化的节点。构建完成会形成一颗Fiber Tree。在react内部会同时存在两棵 Fiber 树。

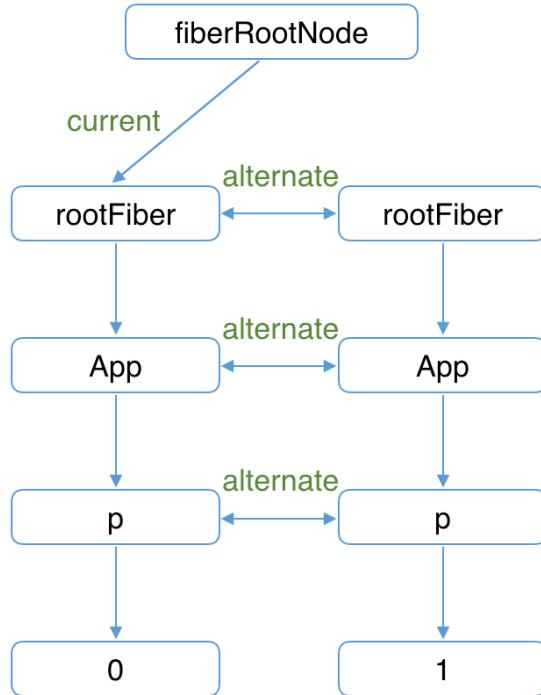


双缓存结构

在 React 中最多会同时存在两棵 Fiber 树。当前屏幕上显示内容对应的 Fiber 树称为 `current Fiber` 树，正在内存中构建的 Fiber 树称为 `workInProgress Fiber` 树。

`current Fiber` 树中的 Fiber 节点被称为 `current fiber`，`workInProgress Fiber` 树中的 Fiber 节点被称为 `workInProgress fiber`，他们通过 `alternate` 属性连接。

如果之前没有 Fiber Tree 就逐级创建 Fiber Tree；如果存在 Fiber Tree，会构建一个 `WorkInProgress Tree`，这个 tree 的 Fiber 节点可以复用 `Current Tree` 上没有发生变化的节点数据。



为什么是双缓存结构？

1. 可以很快的找到之前对应的 Fiber
2. 在某些情况下可以直接复用 fiber

3. 更新完毕后 current直接指向workInProgress root，完成了Fiber tree的更新

构建Fiber Tree

```
1 render(){
2   return <div>
3     i am
4     <span>yideng</span>
5   </div>
6 }
7
8 // ReactDOM.render()
```

Fiber Tree 构建工作的代码大致从 renderRootSync 函数开始，从优先级最高的Fiber Root开始递归。

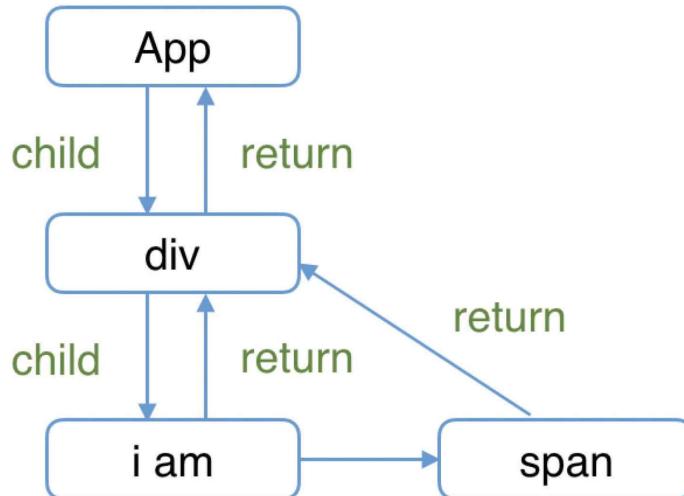
```
1 workInProgress:指的是 workInProgress节点，表示当前正在构建的哪一个fiber
2 function workLoopSync() {
3   // Already timed out, so perform work without checking if we need to yield.
4   // workInProgress:当前正在处理的节点,
5   while (workInProgress !== null) {
6     performUnitOfWork(workInProgress);
7   }
8 }
9
10 function performUnitOfWork(unitOfWork: Fiber): void {
11   // The current, flushed, state of this fiber is the alternate. Ideally
12   // nothing should rely on this, but relying on it here means that we don't
13   // need an additional field on the work in progress.
14   const current = unitOfWork.alternate;
15   ...
16   // 会创建一个Fiber，赋值给workInProgress.child并返回workInProgress.child。
17   // 注意，sibling Fiber，此处暂时还没处理
18   // 返回 next = workInProgress.child
19   // 1. 创建或者复用fiber节点
20   // 2. 对比是否节点需要更新
21   // 3. 打上flags标记—节点的更新类型，删除/新增/修改..
22   // 4. 返回子节点
23   // 5. 在beginWork里会执行render声明周期之前的所有声明周期+render声明周期
24   // render及render前的生命周期都是会执行的。Fragment, Partial, Function, Class, memo
25   next = beginWork(current, unitOfWork, subtreeRenderLanes);
26   ...
27   unitOfWork.memoizedProps = unitOfWork.pendingProps;
28   if (next === null) {
29     // If this doesn't spawn new work, complete the current work.
30     completeUnitOfWork(unitOfWork);
31   } else {
```

```

32     workInProgress = next;
33 }
34 ReactCurrentOwner.current = null;
35 }

```

在beginWork函数里：只创建了这副图中的App, div, iam 3个Fiber。当没有子节点时，进入到completeUnitOfWork里执行，执行completeUnitOfWork后如果存在兄弟Fiber节点，就从兄弟Fiber节点这里接着执行BeginWork.执行完了，又接着执行completeUnitOfWork。这就是Fiber Tree构建的整体流程。



介绍了Fiber Tree 的整体流程，接下来我们可以稍微更具体来看下beginWork和completeWork这两个函数还做了哪些事情。由于beginWork里的情况特别多，我们选取其中一种来加以分析，其他的也都差不多。

beginWork

1. 判断Fiber 节点是否需要更新，不需要更新，是不是就是直接可以复用

```
// Class Function ConText Fragment ..
```

2. 根据不同的Tag，生成不同的Fiber节点（调用reconcileChildren）

- a. Mount 阶段：创建Fiber 节点

- b. Update阶段 和现在的Fiber节点做对比，生成新的Fiber节点

- i. 单节点Diff

- ii. 多节点Diff

3. 给存在变更的Fiber节点打上标记 newFiber.flags = Placement|Update|Deletion|...

4. 创建的Fiber节点赋给WorkInProgress.child,返回WorkInProgress.child. 继续下一次的循环

5. 不同类型的fiber节点会执行相应的特殊逻辑

- a. Class组件 会调生命周期函数 shouldComponentUpdate / PureComponent/render(){}

- b. Function组件 会注入hooks上下文,函数体也会执行

```
1 // 生成Fiber节点—依赖的Render的元素
```

```
2 function beginWork(
3   current: Fiber | null,
4   workInProgress: Fiber,
5   renderLanes: Lanes,
6 ): Fiber | null {
7   const updateLanes = workInProgress.lanes;
8   // current tree 存在, 不是初次构建
9   if (current !== null) {
10     const oldProps = current.memoizedProps;
11     const newProps = workInProgress.pendingProps;
12     if (
13       oldProps !== newProps ||
14       hasLegacyContextChanged()
15     ) {
16       // If props or context changed, mark the fiber as having performed work.
17       // This may be unset if the props are determined to be equal later (memo).
18       didReceiveUpdate = true;
19     } else if (!includesSomeLane(renderLanes, updateLanes)) {
20       // 更新的优先级和current tree的优先级是否有一致的, 不一致才触发bailout
21       didReceiveUpdate = false;
22       // This fiber does not have any pending work. Bailout without entering
23       // the begin phase. There's still some bookkeeping we that needs to be done
24       // in this optimized path, mostly pushing stuff onto the stack.
25       ...
26       // didReceiveUpdate = false; 可以复用上次的fiber
27     // { this.extra }
28     return bailoutOnAlreadyFinishedWork(current, workInProgress, renderLanes);
29   } else {
30     if ((current.flags & ForceUpdateForLegacySuspense) !== NoFlags) {
31       // This is a special case that only exists for legacy mode.
32       // See https://github.com/facebook/react/pull/19216.
33       didReceiveUpdate = true;
34     } else {
35       // An update was scheduled on this fiber, but there are no new props
36       // nor legacy context. Set this to false. If an update queue or context
37       // consumer produces a changed value, it will set this to true. Otherwise,
38       // the component will assume the children have not changed and bail out.
39       didReceiveUpdate = false;
40     }
41   }
42 } else {
43   didReceiveUpdate = false;
44 }
45 // Before entering the begin phase, clear pending update priority.
46 // TODO: This assumes that we're about to evaluate the component and process
47 // the update queue. However, there's an exception: SimpleMemoComponent
48 // sometimes bails out later in the begin phase. This indicates that we should
49 // move this assignment out of the common path and into each branch.
50 workInProgress.lanes = NoLanes;
51 switch (workInProgress.tag) {
```

```
52     case IndeterminateComponent:  
53     ...  
54     case LazyComponent:  
55     ...  
56     case FunctionComponent:  
57         const Component = workInProgress.type;  
58         const unresolvedProps = workInProgress.pendingProps;  
59         const resolvedProps =  
60             workInProgress.elementType === Component  
61                 ? unresolvedProps  
62                 : resolveDefaultProps(Component, unresolvedProps);  
63         // 1. 调用renderWithHooks方法，注入hooks上下文，执行function函数体  
64         // 2. 判断节点是否可以复用，能复用则调bailoutHooks方法复用节点  
65         // 3. 设置flags  
66         // 4. 调用reconcileChildren，得到子Fiber。function函数体会重新执行  
67         return updateFunctionComponent(  
68             current,  
69             workInProgress,  
70             Component,  
71             resolvedProps,  
72             renderLanes,  
73         );  
74     case ClassComponent: {  
75         const Component = workInProgress.type;  
76         const unresolvedProps = workInProgress.pendingProps;  
77         const resolvedProps =  
78             workInProgress.elementType === Component  
79                 ? unresolvedProps  
80                 : resolveDefaultProps(Component, unresolvedProps);  
81         // 执行render()以及render之前调用的生命周期，reconcileChildren  
82         return updateClassComponent(  
83             current,  
84             workInProgress,  
85             Component,  
86             resolvedProps,  
87             renderLanes,  
88         );  
89     }  
90     // ReactDOM.render(<App />)  
91     case HostRoot:  
92         // 会调到reconcileChildren  
93         return updateHostRoot(current, workInProgress, renderLanes);  
94     }  
95     ...  
96  
97     return workInProgress.child;  
98 }  
99  
100 export function reconcileChildren(
```

```
102     current: Fiber | null,
103     workInProgress: Fiber,
104     nextChildren: any,
105     renderLanes: Lanes,
106   ) {
107     if (current === null) {
108       // If this is a fresh new component that hasn't been rendered yet, we
109       // won't update its child set by applying minimal side-effects. Instead,
110       // we will add them all to the child before it gets rendered. That means
111       // we can optimize this reconciliation pass by not tracking side-effects.
112       // Mount阶段, 创建Fiber节点.
113     workInProgress.child = mountChildFibers(
114       workInProgress,
115       null,
116       nextChildren,
117       renderLanes,
118     );
119   } else {
120     // If the current child is the same as the work in progress, it means that
121     // we haven't yet started any work on these children. Therefore, we use
122     // the clone algorithm to create a copy of all the current children.
123     // If we had any progressed work already, that is invalid at this point so
124     // let's throw it out.
125     // Update节点, diff后更新Fiber节点
126     workInProgress.child = reconcileChildFibers(
127       workInProgress,
128       current.child,
129       nextChildren,
130       renderLanes,
131     );
132   }
133 }
134 //reconcileChildFibers里会判断变更的类型是什么? 比如有新增, 删除, 更新等类型。每一种类型的变更, 调用不同的方法, 赋予flags一个值.在commit阶段, 会直接根据flags来做dom操作。
135 // deleteSingleChild
136 function placeSingleChild(newFiber: Fiber): Fiber {
137   // This is simpler for the single child case. We only need to do a
138   // placement for inserting new children.
139   // newFiber.alternate = current;
140   if (shouldTrackSideEffects && newFiber.alternate === null) {
141     newFiber.flags = Placement;
142   }
143   return newFiber;
144 }
145
146 //function A(){
147   // B节点使用 useMemo
148 //   return <div> { B } </div> : // React.createElement
149 //}
150
```

```
151 //const B = React.memo(() => {
152 //    return <div></div>
153 // })
```

diff算法

diff的瓶颈以及React如何应对

React Diff 会预设几个规则：

1. 只对同级节点，进行比较
2. 节点变化，直接删除，然后重建
3. 存在key值，对比节点的key值

```
1 // 代码在ReactChildFiber.new.js 下 reconcileChildFibers函数
2 // 判断节点是不是react 节点
3     // Handle object types
4     const isObject = typeof newChild === 'object' && newChild !== null;
5     if (isObject) {
6         // 根据不同的类型，处理不同的节点对比
7         switch (newChild.$$typeof) {
8             case REACT_ELEMENT_TYPE:
9                 return placeSingleChild(
10                     reconcileSingleElement(
11                         returnFiber,
12                         currentFirstChild,
13                         newChild,
14                         lanes,
15                     ),
16                 );
17             ...
18         }
19     }
20     if (typeof newChild === 'string' || typeof newChild === 'number') {
21         return placeSingleChild(
22             reconcileSingleTextNode(
23                 returnFiber,
24                 currentFirstChild,
25                 '' + newChild,
26                 lanes,
27             ),
28         );
29     }
30     // 多节点数组
31     if (isArray(newChild)) {
32         return reconcileChildrenArray(
33             returnFiber,
34             currentFirstChild,
```

```
35         newChild,
36         lanes,
37     );
38 }
```

单节点diff

1. 判断存在对应节点，key值是否相同，节点类型一致，可以复用
2. 存在对应节点，key值是否相同，节点类型不一致，标记删除
// Div => P/Span 会删除
3. 存在对应节点，key值不同，标记删除
4. 不存在对应节点，创建新节点

```
1 function reconcileSingleElement(
2     returnFiber: Fiber,
3     currentFirstChild: Fiber | null,
4     element: ReactElement,
5     lanes: Lanes,
6 ): Fiber {
7     const key = element.key;
8     let child = currentFirstChild;
9     // 是否存在对应节点
10    while (child !== null) {
11        // 比较key是否相同
12        if (child.key === key) {
13            switch (child.tag) {
14
15                default: {
16                    // 节点类型一致，可以复用
17                    if (
18                        child.elementType === element.type) {
19                        ...
20                        const existing = useFiber(child, element.props);
21                        existing.ref = coerceRef(returnFiber, child, element);
22                        existing.return = returnFiber;
23                        return existing;
24                    }
25                    break;
26                }
27            }
28        // div => p
29        // 节点类型不一致才会到这里，标记为删除
30        // Didn't match.
31        deleteRemainingChildren(returnFiber, child);
32        break;
33    } else {
34        // key不同，将该fiber标记为删除 flags
35        deleteChild(returnFiber, child);
```

```

36     }
37     child = child.sibling;
38   }
39   // 不存在对应节点，创建
40   const created = createFiberFromElement(element, returnFiber.mode, lanes);
41   created.ref = coerceRef(returnFiber, currentFirstChild, element);
42   created.return = returnFiber;
43   return created;
44 }

```

多节点diff

```

1 {
2   arr.map(item => {
3     return <Child key={item.id} onDelete={} onAdd={} />
4   })
5 }
6
7 // 1. 对比新旧children相同index的对象的key是否相等，如果是，返回该对象，如果不是，返回null
8 // 2. key值不等，不用对比下去了，节点不能复用，跳出
9 // 3. 判断节点是否存在移动，存在则返回新位置
10 // 4. 但可能存在新的数组小于老数组的情况，即老数组后面有剩余的，所以要删除
11 // 5. 新数组存在新增的节点，创建新阶段
12 // 6. 创建一个existingChildren代表所有剩余没有匹配掉的节点，然后新的数组根据key从这个 map
13 //      里面查找，如果有则复用，没有则新建
13 function reconcileChildrenArray(
14   returnFiber: Fiber,
15   currentFirstChild: Fiber | null,
16   newChildren: Array<*>,
17   lanes: Lanes,
18 ): Fiber | null {
19   let resultingFirstChild: Fiber | null = null;
20   let previousNewFiber: Fiber | null = null;
21   let oldFiber = currentFirstChild;
22   let lastPlacedIndex = 0;
23   let newIdx = 0;
24   let nextOldFiber = null;
25   for (; oldFiber !== null && newIdx < newChildren.length; newIdx++) {
26     // oldIndex 大于 newIndex，那么需要旧的 fiber 等待新的 fiber，一直等到位置相同
27     if (oldFiber.index > newIdx) {
28       nextOldFiber = oldFiber;
29       oldFiber = null;
30     } else {
31       nextOldFiber = oldFiber.sibling;
32     }
33     // 对比新旧children相同index的对象的key是否相等，如果是，返回该对象，如果不是，返回
34     // null
35     // newFiber如果有值，意味着可以复用
36     const newFiber = updateSlot(

```

```

36     returnFiber,
37     oldFiber,
38     newChildren[newIdx],
39     lanes,
40 );
41 // key值不等，不用对比下去了，节点不能复用，跳出
42 if (newFiber === null) {
43     if (oldFiber === null) {
44         oldFiber = nextOldFiber;
45     }
46     break;
47 }
48 // 判断节点是否存在移动，存在则返回新位置
49 lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
50 oldFiber = nextOldFiber;
51 }
52 // 第一个循环完毕
53 // newIdx === newChildren.length 说明中途没有跳出的情况
54 // 但可能存在新的数组小于老数组的情况，即老数组后面有剩余的，所以要删除
55 if (newIdx === newChildren.length) {
56     // We've reached the end of the new children. We can delete the rest.
57     deleteRemainingChildren(returnFiber, oldFiber);
58     return resultingFirstChild;
59 }
60 // oldFiber === null 说明数组中所有的都可以复用
61 if (oldFiber === null) {
62     // 新数组存在新增的节点，创建新节点
63     for (; newIdx < newChildren.length; newIdx++) {
64         const newFiber = createChild(returnFiber, newChildren[newIdx], lanes);
65         if (newFiber === null) {
66             continue;
67         }
68         // 添加到sibling
69         lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
70     }
71     return resultingFirstChild;
72 }
73 // 创建一个existingChildren代表所有剩余没有匹配掉的节点，然后新的数组根据key从这个
74 map 里面查找，如果有则复用，没有则新建
75 const existingChildren = mapRemainingChildren(returnFiber, oldFiber);
76 // Keep scanning and use the map to restore deleted items as moves.
77 for (; newIdx < newChildren.length; newIdx++) {
78     // 对比是否还有可以复用的
79     const newFiber = updateFromMap(
80         existingChildren,
81         returnFiber,
82         newIdx,
83         newChildren[newIdx],
84         lanes,

```

```
85     );
86     if (newFiber !== null) {
87         // 判断节点是否存在移动，存在则返回新位置
88         lastPlacedIndex = placeChild(newFiber, lastPlacedIndex, newIdx);
89     }
90 }
91 return resultingFirstChild;
92 }
```

打上Effect Tag flags

Gherkin

```
1 workInProgress.flags |= Placement;
```

completeUnitOfWork

1. 向上递归completedWork
2. 创建DOM节点，更新Dom节点； DOM节点赋值给stateNode属性
3. 把子节点的sideEffect统统附加到父节点的sideEffect链之上，在commit阶段使用
4. 存在兄弟节点，将workInProgress指向兄弟节点，执行兄弟节点的beginWork过程
5. 不存在兄弟节点，返回父节点。继续执行父节点的completeWork

即，构建整个Fiber Tree是处于2重循环之中的。

```
1 // 1. 向上递归completedWork
2 // 2. 创建DOM节点，更新Dom节点。
3 // 3. 把子节点的side Effect， flag统统附加到父节点的sideEffect(flags)链之上
4 // 4. 存在兄弟节点，将workInProgress指向兄弟节点，执行兄弟节点的beginWork过程
5 // 5. 不存在兄弟节点，返回父节点。继续执行父节点的completeWork
6 function completeUnitOfWork(unitOfWork: Fiber): void {
7     // Attempt to complete the current unit of work, then move to the next
8     // sibling. If there are no more siblings, return to the parent fiber.
9     let completedWork = unitOfWork;
10    do {
11        // The current, flushed, state of this fiber is the alternate. Ideally
12        // nothing should rely on this, but relying on it here means that we don't
13        // need an additional field on the work in progress.
14        const current = completedWork.alternate;
15        const returnFiber = completedWork.return;
16        // Check if the work completed or if something threw.
17        if ((completedWork.flags & Incomplete) === NoFlags) {
18            setCurrentDebugFiberInDEV(completedWork);
19            let next;
20            ...
```

```
21      // 1. Mount时，创建DOM节点，将子孙DOM节点插入刚生成的DOM节点中，DOM节点赋值给  
22      // stateNode保存备用  
23      next = completeWork(current, completedWork, subtreeRenderLanes);  
24      ...  
25      // 把子节点的side Effect统统附加到父节点的sideEffect链之上  
26      if (  
27          returnFiber !== null &&  
28          // Do not append effects to parents if a sibling failed to complete  
29          (returnFiber.flags & Incomplete) === NoFlags  
30      ) {  
31          // Append all the effects of the subtree and this fiber onto the effect  
32          // list of the parent. The completion order of the children affects the  
33          // side-effect order.  
34          if (returnFiber.firstEffect === null) {  
35              returnFiber.firstEffect = completedWork.firstEffect;  
36          }  
37          if (completedWork.lastEffect !== null) {  
38              if (returnFiber.lastEffect !== null) {  
39                  returnFiber.lastEffect.nextEffect = completedWork.firstEffect;  
40              }  
41              returnFiber.lastEffect = completedWork.lastEffect;  
42          }  
43          // If this fiber had side-effects, we append it AFTER the children's  
44          // side-effects. We can perform certain side-effects earlier if needed,  
45          // by doing multiple passes over the effect list. We don't want to  
46          // schedule our own side-effect on our own list because if end up  
47          // reusing children we'll schedule this effect onto itself since we're  
48          // at the end.  
49          const flags = completedWork.flags;  
50          // Skip both Nowork and PerformedWork tags when creating the effect  
51          // list. PerformedWork effect is read by React DevTools but shouldn't be  
52          // committed.  
53          if (flags > PerformedWork) {  
54              if (returnFiber.lastEffect !== null) {  
55                  returnFiber.lastEffect.nextEffect = completedWork;  
56              } else {  
57                  returnFiber.firstEffect = completedWork;  
58              }  
59              returnFiber.lastEffect = completedWork;  
60          }  
61      }  
62  }  
63  // 有兄弟节点，赋值给workInProgress，并return，接下来会继续执行beginWork  
64  const siblingFiber = completedWork.sibling;  
65  if (siblingFiber !== null) {  
66      // If there is more work to do in this returnFiber, do that next.  
67      workInProgress = siblingFiber;  
68  return;
```

```

69     }
70     // Otherwise, return to the parent
71     // 没有兄弟节点，返回到父节点，继续执行下一次的循环
72     completedWork = returnFiber;
73     // Update the next thing we're working on in case something throws.
74     workInProgress = completedWork;
75   } while (completedWork !== null);
76 }
77
78 function completedWork(){
79   ...
80   // client: 创建DOM节点 document.createElement();
81   const instance = createInstance(
82     type,
83     newProps,
84     rootContainerInstance,
85     currentHostContext,
86     workInProgress,
87   );
88   // 将子孙DOM节点插入刚生成的DOM节点中
89   appendAllChildren(instance, workInProgress, false, false);
90   // DOM节点赋值给stateNode保存备用
91   workInProgress.stateNode = instance;
92   // Certain renderers require commit-time effects for initial mount.
93   // (eg DOM renderer supports auto-focus for certain elements).
94   // Make sure such renderers get scheduled for later work.
95   if (
96     finalizeInitialChildren(
97       instance,
98       type,
99       newProps,
100      rootContainerInstance,
101      currentHostContext,
102    )
103   ) {
104     markUpdate(workInProgress);
105   }
106 }
107 }

```

render(commit)——负责将变化的组件渲染到页面上 CommitContext

- 优先级最高，RunWithPriority(最高优先级, () => { commitRoot(); });

分为3个阶段：

commitBeforeMutationEffects (DOM操作前)

- 处理 DOM 节点 渲染/删除后的 autoFocus、blur 逻辑。

2. 调用 `getSnapshotBeforeUpdate` 生命周期钩子。
3. 调度 `useEffect`. 也是一个异步的东西，不会阻塞。 `useEffect => componentDidMount + componentDidUpdate`

```
1 useLayoutEffect 同步的, componentDidMount生命周期的
2 useEffect 异步的
3 class A extends Component{
4     getSnapshotBeforeUpdate(){
5         console.log(1);
6     }
7     render(){
8         return <B> </B>
9     }
10 }
11 class B extends Component{
12     getSnapshotBeforeUpdate(){
13         console.log(2);
14     }
15     render(){
16         return <div> </div>
17     }
18 }
19
20 // Effect list
21 function commitBeforeMutationEffects(firstChild: Fiber) {
22     let fiber = firstChild;
23     while (fiber !== null) {
24         // 处理需要删除的fiber autoFocus、blur 逻辑。
25         if (fiber.deletions !== null) {
26             commitBeforeMutationEffectsDeletions(fiber.deletions);
27         }
28         // 递归调用处理子节点
29         if (fiber.child !== null) {
30             commitBeforeMutationEffects(fiber.child);
31         }
32         try {
33             // 调用 getSnapshotBeforeUpdate 生命周期
34             // 异步调度useEffect
35             commitBeforeMutationEffectsImpl(fiber);
36         } catch (error) {
37             captureCommitPhaseError(fiber, fiber.return, error);
38         } // 返回兄弟节点, 接着循环
39         fiber = fiber.sibling;
40     }
41 }
42
43 // 在beginWork里, 存在getSnapshotBeforeUpdate的时候, fiber.flags |= Snapshot;
44 function commitBeforeMutationEffectsImpl(fiber: Fiber) {
45     const current = fiber.alternate;
```

```

46 const flags = fiber.flags;
47 if ((flags & Snapshot) !== NoFlags) {
48     // 调用 getSnapshotBeforeUpdate 生命周期
49     commitBeforeMutationEffectOnFiber(current, fiber);
50 }
51 // tags
52 if ((flags & Passive) !== NoFlags) {
53     // If there are passive effects, schedule a callback to flush at
54     // the earliest opportunity.
55     if (!rootDoesHavePassiveEffects) {
56         rootDoesHavePassiveEffects = true;
57         // 异步调度 useEffect 的回调并不是在 dom 渲染前执行的。
58         // useLayoutEffect 在 dom 操作后同步执行回调
59         // useEffect 异步执行回调。不想阻塞主线程。可以先尝试使用useEffect。不行的话再使用
60         // useLayoutEffect。
61         scheduleCallback(NormalSchedulerPriority, () => {
62             flushPassiveEffects();
63             return null;
64         });
65     }

```

commitMutationEffects(执行DOM操作)

1. 遍历finishedWork，执行DOM操作
2. 对于删除的组件，会执行componentWillUnmount生命周期

```

1 // 结构和上面的一样
2 function commitMutationEffects(
3     firstChild: Fiber,
4     root: FiberRoot,
5     renderPriorityLevel: ReactPriorityLevel,
6 ) {
7     let fiber = firstChild;
8     while (fiber !== null) {
9         const deletions = fiber.deletions;
10        if (deletions !== null) {
11            // 需要卸载的组件，会调用componentWillUnmount
12            commitMutationEffectsDeletions(
13                deletions,
14                fiber,
15                root,
16                renderPriorityLevel,
17            );
18        }
19        if (fiber.child !== null) {
20            // 仍然是递归调用处理子Fiber
21            commitMutationEffects(fiber.child, root, renderPriorityLevel);
22        }

```

```

23   try {
24     // 区分不同情Flag 执行不同的Dom操作
25     // 已经构造好了dom元素了，存放在stateNode节点的。新增，删除，替换？
26     commitMutationEffectsImpl(fiber, root, renderPriorityLevel);
27     // 页面就终于渲染出来了
28   } catch (error) {
29     captureCommitPhaseError(fiber, fiber.return, error);
30   }
31   // 处理兄弟节点
32   fiber = fiber.sibling;
33 }
34 }
```

recursivelyCommitLayoutEffects(DOM操作后)

在这个阶段前current tree也发生了变化了，指向了最新构建的workInProgress tree。

1. layout阶段 也是深度优先遍历 effectList，调用生命周期，componentdidMount/componentdidUpdate；执行useLayoutEffect的函数体
2. 赋值 ref
3. 处理ReactDOM.render 回调

```

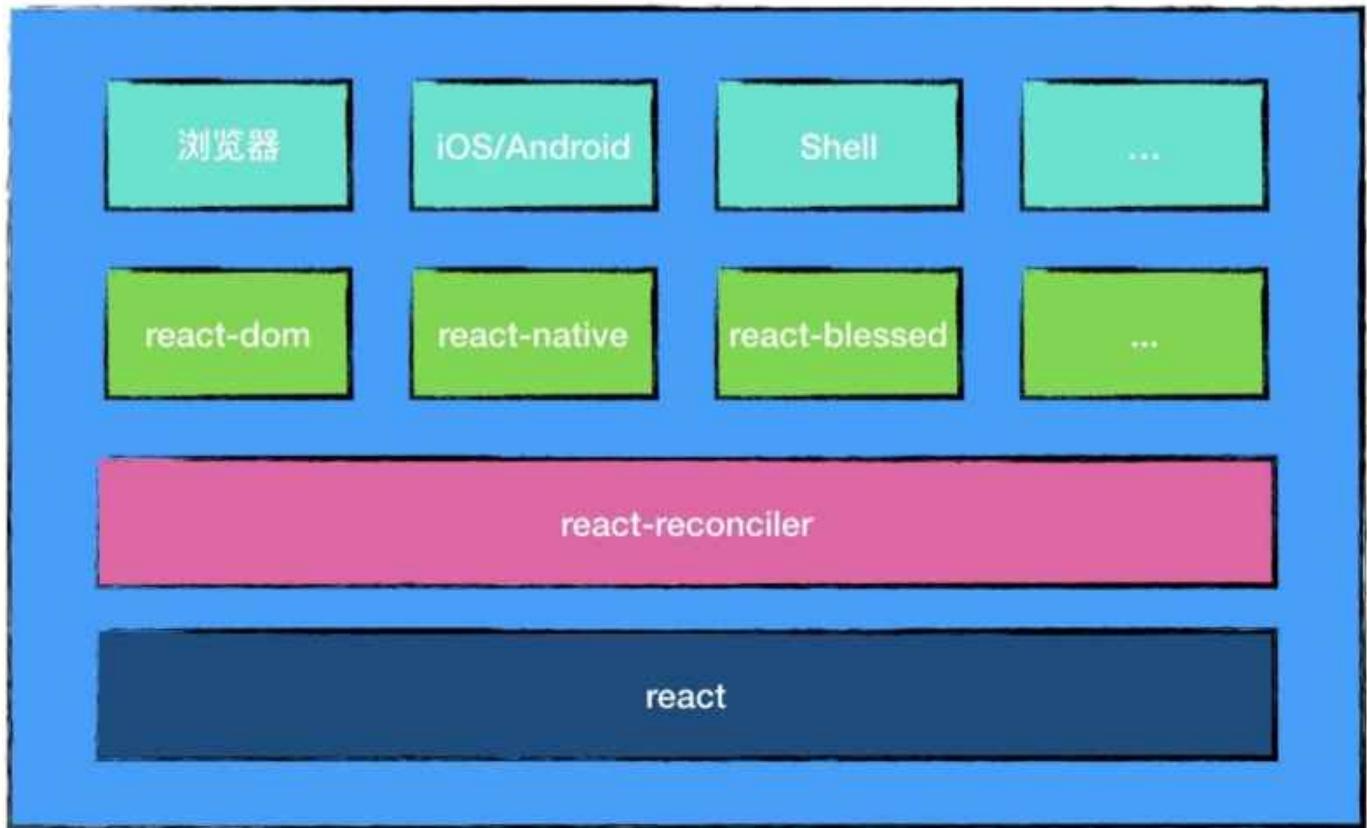
1 // A,B两个组件，生命周期getSnapshotBeforeUpdate, componentDidMount,
  componentWillMount, componentWillUnmount 问整体的执行顺序？
2 // 哪几个是在commit阶段执行的？哪几个是在BeginWork里执行的？
3 // render之前的声明周期是先执行的父节点的，再执行的子节点的
4 // componentDidMount先执行的子节点的，再执行的父节点的
5 // useEffect === componentDidMount 在LayoutEffect这个阶段执行的
6
7 // 首先要区分的是生命周期是render函数之前的还是render函数之后的
8 // render函数之后的生命周期是分开执行的，3个while，先执行的子节点再执行的父节点
9 class A extends Component{
10   componentWillMount(){
11     // 在前面
12     console.log('a')
13   }
14   componentDidMount(){
15     console.log(1);
16   }
17   getSnapshotBeforeUpdate(){
18     console.log(3);
19   }
20   render(){
21     return <><B></B> <C></C></>
22   }
23 }
24
25 const C = () => {
26   // useEffect是异步执行的
```

```
27     useLayoutEffect(() => {
28         console.log("ddd");
29     }, [])
30     // useEffect(() => {
31     //     console.log("ccc");
32     // }, []);
33     return <div>2323</div>
34 };
35 class B extends Component{
36     componentWillMount(){
37         // 在前面
38         console.log('b');
39     }
40
41     componentDidMount(){
42         console.log(2);
43     }
44     getSnapshotBeforeUpdate(){
45         console.log(4);
46     }
47     render(){
48         return <div> </div>
49     }
50 }
51
52 ReactDOM.render(<App/>, $("#app"), () => {
53     console.log('1');
54 });
55
56
57 function recursivelyCommitLayoutEffects(
58     finishedWork: Fiber,
59     finishedRoot: FiberRoot,
60 ) {
61     const {flags, tag} = finishedWork;
62     switch (tag) {
63         ...
64     default: {
65         let child = finishedWork.child;
66         while (child !== null) {
67             ...
68             try {
69                 // 仍然是递归
70                 recursivelyCommitLayoutEffects(child, finishedRoot);
71             } catch (error) {
72                 captureCommitPhaseError(child, finishedWork, error);
73             }
74             child = child.sibling;
75         }
76         const primaryFlags = flags & (Update | Callback);
```

```
77     if (primaryFlags !== NoFlags) {
78         switch (tag) {
79             case FunctionComponent:
80             case ForwardRef:
81             case SimpleMemoComponent:
82             case Block: {
83                 ...
84                 // 执行useLayoutEffect的回调
85                 commitHookEffectListMount(
86                     HookLayout | HookHasEffect,
87                     finishedWork,
88                 );
89                 break;
90             }
91             case ClassComponent: {
92                 // 执行componentDidMount/didUpdate 生命周期
93                 // NOTE: Layout effect durations are measured within this function.
94                 commitLayoutEffectsForClassComponent(finishedWork);
95                 break;
96             }
97             ...
98         }
99         ...
100        // 赋值ref
101        if (flags & Ref && tag !== ScopeComponent) {
102            commitAttachRef(finishedWork);
103        }
104        break;
105    }
106 }
107 }
```

react-reconciler

react将reconciler的逻辑封装成独立的子模块，这样做有一个好处，无论是浏览器环境，还是原生环境，都需要构建fiber tree，找出变化的组件，那么**Reconciler**就是完全可以拿来复用的。而事实上也正如此。



通过react-reconciler,我们完全可以自己写一个不同平台的渲染器，渲染到不同的平台。

JavaScript

```

1 const Reconciler = require('react-reconciler');
2
3 const HostConfig = {
4   // You'll need to implement some methods here.
5   // See below for more information and examples.
6   createInstance(type, props) {
7     // e.g. DOM renderer returns a DOM node
8     return document.createElement(type, props);
9   },
10  appendChild(parent, child) {
11    // e.g. DOM renderer would call .appendChild() here
12  },
13};
14
15 const MyRenderer = Reconciler(HostConfig);
16 const RendererPublicAPI = {
17   render(element, container, callback) {
18     // Call MyRenderer.updateContainer() to schedule changes on the roots.
19     // See ReactDOM, React Native, or React ART for practical examples.
20   }
21 };
22 module.exports = RendererPublicAPI;

```

更新流程

ReactDOM.render流程

```
unbatchedUpdate(() => {
```

```
});
```

this.setState / hooks setState流程

```
onCLick={() => {
```

```
    this.setState({});
```

```
}];
```

Hooks源码解读

1. 为什么会有hooks? 解决了什么样的问题

class Component为主，function Component为辅， 只能通过props把state传递过来。

function Component 做UI组件。没有定义state的能力。

function Component + hooks 可以做到更简洁

可以做到更方便的抽取逻辑

2. hooks 的基本用法

useContext + useReducer + createContext = redux

useEffect: 模拟生命周期 componentDidMount/didUpdate/willUnmount

useMemo: 缓存一个值

useCallback: 缓存一个function

useRef: ref 在function里使用

useState: 定义state和改变state的方式

hooks常见的几大问题：

0. 使用useState只能在顶层使用，不能出现在条件语句里.

1. 死循环

2. capture value 通过ref来解决。

JavaScript

```
1  function Test(){
2      const [ count, setCount ] = useState(() => {
3          return 0
4      })
5
6      const handleCountChange = () => {
7          setCount(count + 1)
8      }
9
10     return (
11         <div>
12             <p>Count: {count}</p>
13             <button onClick={handleCountChange}>+</button>
14         </div>
15     )
16 }
```

```
5         return v
6     });
7
8     const [ age, setAge ] = useState(0);
9     const [ name, setName ] = useState(0);
10    const [ age1, setAge1 ] = useState(0);
11    const nameRef = useRef(null)
12    const ref1 = useRef(0)
13
14    // useMemo 是缓存一个值
15    // 1. 每次都会被重新创建，消耗浏览器资源
16    // 2. 父组件任何的变化都会导致子组件的更新
17    const addCount = useCallback(() => {
18        // 既可以传值，也可以传函数
19        setCount((count)=>{
20            return count++;
21        })
22        console.log(age1);
23        ref1.current = count++;
24        // setTimeout(() => {
25        //     // 还是保存的之前的变量
26        //     // console.log(ref1.current)
27        //     // }, 3000)
28    }, [age1])
29
30    // 每运行一次，都是创建新的函数
31    const addCount1 = () => {
32        // 既可以传值，也可以传函数
33        setCount((count)=>{
34            return count++;
35        })
36        ref1.current = count++;
37        // setTimeout(() => {
38        //     // 还是保存的之前的变量
39        //     // console.log(ref1.current)
40        //     // }, 3000)
41    }
42
43    // componentDidUpdate
44    useEffect(() =>{
45        const onScroll = () => {}
46        window.addEventListener('scroll', onScroll);
47        // distory
48        return () => {
49            window.removeEventListener('scroll', onScroll);
50        }
51    }, [age])
52
53    useEffect(() =>{
54        1. 有判断逻辑，用ref来做
55        if(nameRef.current){
56            // 去除死循环的目的
57        }
58    })
59
```

```
52     }
53     request().then(res => {
54         setName(name + '_dd');
55         // nameRef.current = name + '_dd';
56         setName(name => name + '_dd')
57         // 1. 会导致name变化
58         // 2. name变化，导致useEffect的函数体会重新执行一次
59         // 3. 又会导致name变化
60     })
61 }, [])
62
63 // 1. 不写依赖数组，useEffect的回调每次渲染都会执行
64 // 2. age变化的时候，useEffect的回调才会执行
65 // 3. 可以返回一个函数，在函数销毁的时候会调用它
66 // componentDidMount
67 useEffect(() =>{
68     unstable_batchedUpdates(() => {
69         setAge();
70         setAge1();
71         setName();
72     });
73 }, [])
74
75 // componentDidUpdate
76 useEffect(() =>{
77     if(age === 0) return;
78     const onScroll = () => {}
79     window.addEventListener('scroll', onScroll)
80     // 组件销毁的时候，会调用这个useEffect的回调
81     return () => {
82         window.removeEventListener('scroll', onScroll)
83     }
84 }, [age])
85 return <div>
86 <div className='wrapper' ref={ref}>{count}</div>
87 <div className='btn' onClick={addCount1}>点击</div>
88     <Child addCount={addCount} />
89     <Child addCount={addCount1} />
90 </div>
91 }
92
93
94 const Child = ({addCount}) => {
95     return <div onClick={addCount}></div>
96 }
97 // 对Child组件的props做了一次浅比较 React.memo + useCallback
98 export React.memo(Child);
```

简版的实现

JavaScript

```
1  const [name, setName] = useState('yideng');
2  const [name, setName] = useState('yideng2');
3  const [name, setName] = useState('yideng3');
4  // 第一次执行函数体: name = 'yideng';
5  // setName: setName('yideng1');
6  // 第二次执行函数体: name = 'yideng1';
7
8  let state
9  function useState(defaultState) {
10    function setState(newState) {
11      state = newState;
12      // schedule();
13    }
14
15    if (!state) {
16      state = defaultState
17    }
18    return [state, setState]
19  }
20  // 1. 只能定义一个state
21  // 2. 没有上下文, 只能在同一个函数里使用
22  function functionA() {
23    const [name, setName] = useState(0);
24    console.log(name)
25    setState(name + 1)
26  }
27  function functionB() {
28    const [name, setName] = useState(0);
29    console.log(name)
30    setState(name + 1)
31 }
```

状态堆与上下文栈存储多个状态多个函数

JavaScript

```
1 let contextStack = []
2
3 function useState(defaultState=0) {
4   const context = contextStack[contextStack.length - 1]
5   const nu = context.nu++
6
7   const { states } = context
8 }
```

```

9  function setState(newState) {
10    states[nu] = newState
11  }
12
13  if (!states[nu]) {
14    states[nu] = defaultState
15  }
16
17  return [states[nu], setState]
18 }
19
20 function useState(func) {
21   const states = {}
22   return (...args) => {
23     contextStack.push({ nu: 0, states })
24     const result = func(...args)
25     contextStack.pop()
26     return result
27   }
28 }
29
30 const function1 = useState(
31   function render() {
32     // { nu: 0, states: {'0': 0} }
33     const [state, setState] = useState(0)
34     // { nu: 1, states: {'0': 0, 1: 'name'} }
35     const [name, setName] = useState("name")
36     render1()
37     console.log('render', state)
38     setState(state + 1)
39   }
40 )
41
42 const function2 = useState(
43   function render1() {
44     const [state, setState] = useState(0)
45     console.log('render1', state)
46     setState(state + 2)
47   }
48 )
49
50 function1()

```

useEffect

JavaScript

```
1 // 每一个state变化的时候，useEffect的函数体都会重新执行
2 useEffect(() => {
3
4 }, []);
5 function useEffect(callback, depArray=[]) {
6   const hasNoDeps = !depArray;
7   const deps = memoizedState[cursor];
8   const hasChangedDeps = deps
9     ? !depArray.every((el, i) => el === deps[i])
10    : true;
11   if (hasNoDeps || hasChangedDeps) {
12     callback();
13     memoizedState[cursor] = depArray;
14   }
15   cursor++;
16 }
```

hooks源码分析

useState

第一次执行App函数：

useState: MountState => 创建一个hooks，返回一个默认值和更改默认值的dispatchAction

调用dispatchAction的时候：

创建一个更新，添加到state

调用scheduleUpdateOnFiber

第二次执行App函数：

按顺序找到之前的hooks

计算Action得到最新的值，返回一个最新的值和按顺序找到之前的hooks

MountState

0. 创建hooks链表，放在FiberNode的memorizedState属性里
1. 默认值是function，执行function，得到初始state
2. state是存放在memoizedState
3. 新建一个queue
4. 把queue传递给dispatch

5. 返回默认值和dispatch

`dispatchAction(setName(() => {}), setAge)`

1. 创建一个update
2. update添加到quene里
3. 空闲的时候：提前计算出最新的state，保存在eagerState
4. 最后调用一次`scheduleUpdateOnFiber`，进入schedule，进入Reconciler，触发function重新执行一次

UpdateState

0. 按顺序找到之前的hooks
1. 递归执行quene里的update
2. 计算最新的state，赋值给memoizedState

useEffect

注册的时候——beginWork调App函数体的时候

```
useEffect(() => {  
  console.log('');  
  return () => {}  
}, ['a', 'b']);
```

初始化：

在创建Fiber的时候：

1. 函数体会执行
2. 执行useEffect本身的逻辑
 - a. 打上flags标记
 - b. push一个Effect

在commit阶段的LayoutEffects阶段：

3. 执行commitHookEffectListMount
4. 执行了create方法，有返回值返回给destory.

有state变化发生的时候：

5. dispatchAction，调用一次`scheduleUpdateOnFiber`
6. 函数体会执行
7. 给destory赋值

UpdateEffect

1. 对比依赖的值有没有变化，**有变化的话，重新发起一个Effect.** 依赖不变，不会发起Effect，执行不到callBack

2. 打上flags标记
3. push一个Effect

在commit阶段的LayoutEffects阶段：

1. 调用commitHookEffectListUnmount的时候，会执行destory()
2. 执行commitHookEffectListMount
3. 执行了create方法，有返回值 返回给destory.

mountEffect

1. 打上flags标记
2. push一个Effect

JavaScript

```
1 // 执行useEffect本身的逻辑的时候，还在beginWork阶段，useEffect回调在commit阶段执行
2 useEffect(() => {
3     return () => {
4
5     }
6 }, [])
```

updateEffect

0. 给destroy赋值
1. 对比依赖的值有没有变化，有变化的话，重新发起一个Effect
2. 打上flags标记
3. push一个Effect

useEffect回调的执行时机：

在LayoutEffects里调用commitHookEffectListMount方法，执行了create方法，返回给destory.

在调用commitHookEffectListUnmount的时候，会执行destory();

全新的JSX转换

当你使用 JSX 时，编译器会将其转换为浏览器可以理解的 React 函数调用。**旧的 JSX 转换会把 JSX 转换为 `React.createElement(...)` 调用。**

js代码：

JavaScript

```
1 import React, { useState } from 'react';
2 function App() {
3   const [name, setName] = useState(' ');
4   // React.createElement('h1', )
5   return <h1>Hello World</h1>;
6 }
```

旧转换：

JavaScript

```
1 import React from 'react';
2 function App() {
3   return React.createElement('h1', null, 'Hello world');
4 }
```

新的jsx转换：

JavaScript

```
1 // 由编译器引入（禁止自己引入！）import {jsx as _jsx} from 'react/jsx-runtime';
2 function App() {
3   return _jsx('h1', { children: 'Hello world' });
4 }
```

事件委托

在 React 组件中，对大多数事件来说，React 实际上并不会将它们附加到 DOM 节点上。相反，React 会直接在 `document` 节点上为每种事件类型附加一个处理器。。除了在大型应用程序上具有性能优势外，它还使添加类似于 `replaying events` 这样的新特性变得更加容易。

但是如果页面上有多个 React 版本，他们都将在顶层注册事件处理器。这会破坏

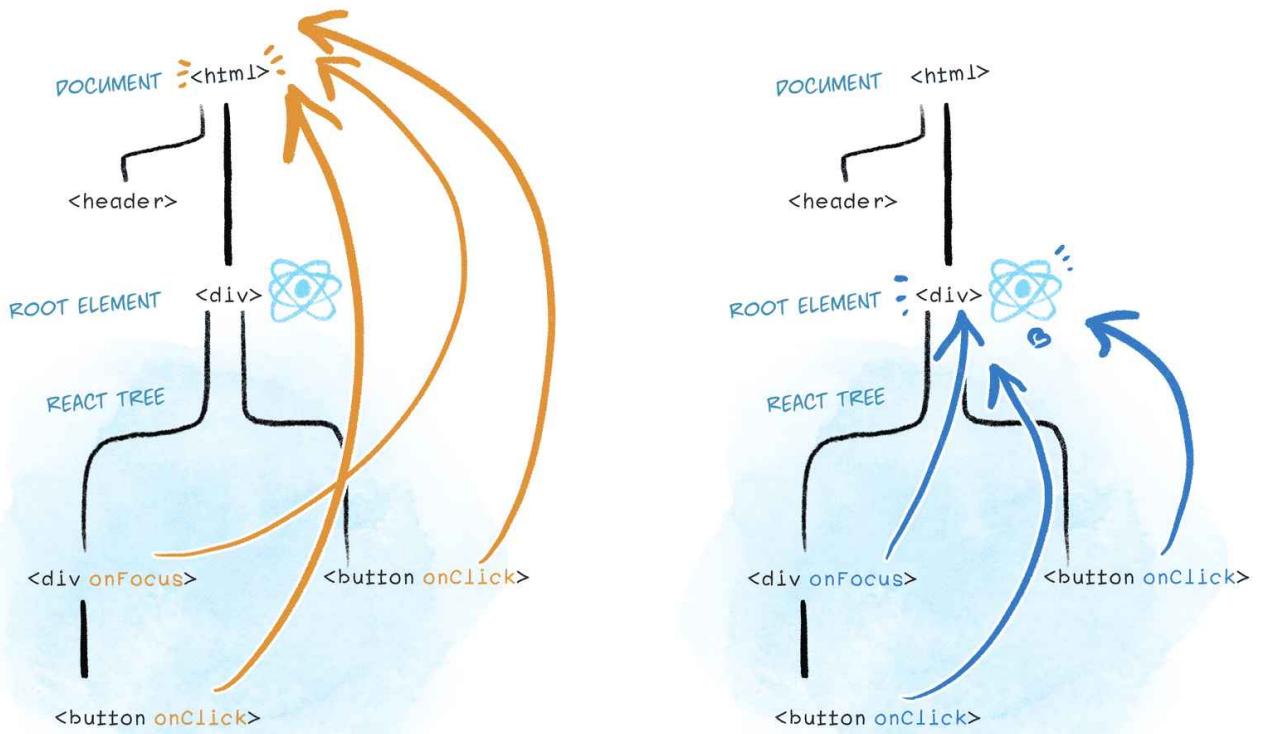
`e.stopPropagation()`：如果嵌套树结构中阻止了事件冒泡，但外部树依然能接收到它。这会使不同版本 React 嵌套变得困难重重。

在 React 17 中，React 将不再向 `document` 附加事件处理器。而会将事件处理器附加到渲染 React 树的根 DOM 容器中：

```
ReactDOM.render(<App />, $("#app"));
ReactDOM.render(<Header />, $("#header"));
ReactDOM.render(<Footer />, $("#footer"));
```

REACT 16

REACT 17



@Rachel Nabors

面试经常问的问题：

1. React fiber架构的核心原理是什么？

fiber架构分为了3个部分。

a. Scheduler 调度，通过任务的优先级来调度任务进入后面的构建流程

i. 同步任务和异步任务

1. 同步任务unbatchedUpdate (ReactDOM.render)

2. 异步的任务 `this.setState({})` 走MessageChannel调度优先级

3. 异步任务能不能同步？

a. `unbatchedUpdate(() => {`

`this.setState({});`

`});`

- ii. React 17是这么实现的；多个this.setState的批处理问题，判断了lanes一致，复用同一次更新。
 - iii. MessageChannel 不用 setTimeout
 - iv. 为什么不采用微任务呢？
 - 1. 微任务是会把页面卡死的
 - b. Reconciler 构建fiber tree，做DOM diff，给变更的fiber打上标记，方便后续处理
 - i. 构建fiber tree的具体流程是什么？
 - 1. 从上到下 创建fiber, diff, 生命周期
 - 2. 从下到上的过程
 - a. 创建dom元素，把effect全部给到跟节点
 - ii. 是可以被中断的
 - iii. 打上flags，生成一个effets链条
 - iv. 把dom都给生成了，保存起来
 - c. commit阶段——把变更的组件的变化，渲染到页面上了
 - i. 3个while，渲染前，渲染中，渲染后
 - ii. 优先子节点执行声明周期
 - iii. 不能被中断了，此时如果还能被中断，开销会更大。把commitRoot的优先级设为了最高
2. 为什么是双缓存结构？
 - a. 一个属性就立刻拿到了和它对应的节点
3. 通过学习react源码学到了什么？
4. react新特性
5. 如何实现一个redux
6. 为什么会有hooks？hooks解决什么问题？
7. React hooks有什么好处？
8. setState后面发生了什么
 - a. 确定setState在哪里调用的
 - i. onClick, 声明周期里
 - 1. 先提前标记优先级
 - 2. 进入schedule流程
 - 3. Reconciler
 - 4. commitRoot阶段
 - ii. hooks里调的
 - 1. 创建一个update
 - 2. update添加到quene里
 - 3. 空闲的时候：提前计算出最新的state，保存在eagerState

4. 最后调用一次scheduleUpdateOnFiber，进入schedule，触发function重新执行一次
 5. 进入schedule流程
 6. Reconciler
 7. commitRoot阶段
9. 为什么react一些生命周期加上了unsafe?
- a. reconciler阶段是可以被高优先级任务中断的，可能会反复执行。会反复执行的声明周期加上了unsafe
10. useEffect什么时候执行，和didMount的区别
- a. 异步调度的，在页面渲染完之后才会执行，didMount == useLayoutEffect
11. React useCallback和useMemo有什么区别
- a. 一个缓存function，一个缓存值
 - b. useCallback + React.memo做浅比较，避免无意义渲染
12. React capture value的原理

CoffeeScript

```
1 是闭包,
2 <Test1 onClick={() => {
3     setTimeout(() => {
4         console.log(data);
5     }, 5000);
6 }} data={data} />
```

13. React dom-diff的策略是什么
14. React 虚拟dom有什么好处，和真实的dom怎么对应
- a. dom是昂贵的，可以只更新一个DOM。Fiber Tree => current tree和真实的DOM一一对应。
15. 怎么在setState后执行回调
- a. this.setState({a: ''}, () => {});
16. 声明周期执行的顺序
17. useEffect和useLayoutEffect的区别
18. React 处理异常的一些方法
- a. ErrorBoundary
19. 实现一个自定义hooks

CoffeeScript

```
1 function useMount (callBack) {
2   useEffect(() => {
3     callBack();
4   }, [])
5   retun [a, b];
6 };
7
8 const Test = () => {
9   const ref = useRef();
10  const cb = useCallback(() => {
11
12 }, [])
13  useMount(() => {
14
15 });
16  return <div className='>
17    <Test1 ref={cb} />
18  </div>
19 }
```

20. useEffect死循环怎么破解

- a. useRef
- b. 使用dispatch传函数的方式 不是适用于所有场景

21. this.setState是同步还是异步?

22. 为什么只能在函数顶层使用hooks?

23. 函数组件使用ref需要注意什么

- a. 使用ref，通过useRef来创建ref，既可以用来得到DOM节点，也可以用来存值
- b. 想监听ref的改变，给ref字段传useCallback函数
- c. 转发ref forwardedRef
- d. useImperativeHandle 自定义暴露给父组件的实例值，可以让父组件访问子组件的方法

react18带来的变化

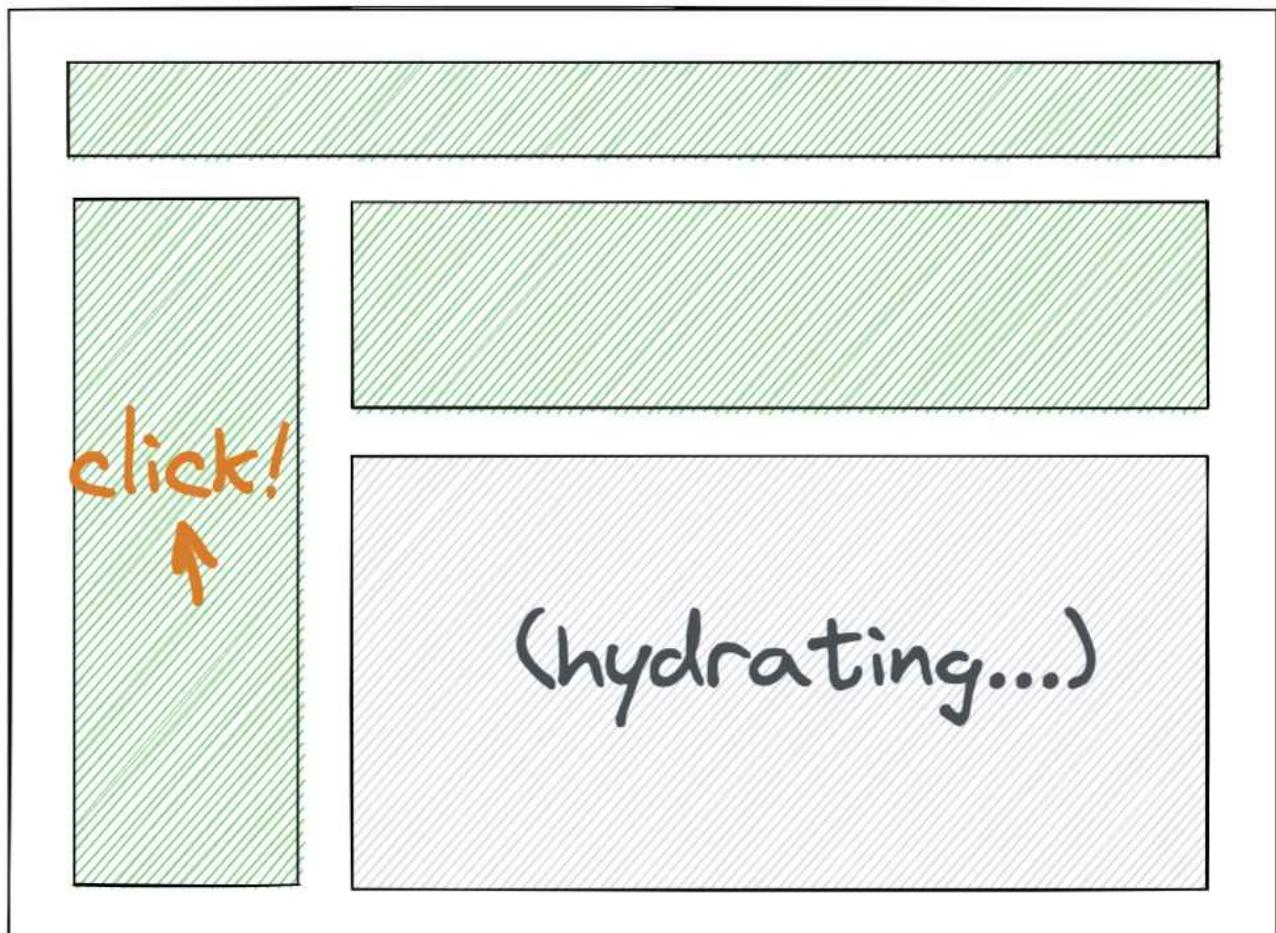
自动的batch update

startTransition

TypeScript

```
1 import { startTransition } from 'react';
2 // Urgent: Show what was typed
3 setInputValue(input);
4 // Mark any state updates inside as transitions
5 // 比 setTimeout 靠前
6 startTransition(() => {
7     // Transition: Show the results
8     setSearchQuery(input);
9});
```

Suspense SSR





React 17 与 React 18 的更新变化

前言

项目目前react17和react18都有使用，但在开发者角度绝大部分场景还是感知不到多大变化，但也要详细了解清楚具体更新了什么。本文就来一次性梳理下 react17 与 react18 的变化。

React 17 更新

首先，官方发布日志称react17最大的特点就是**无新特性**，这个版本主要目标是让React能渐进式升级，它允许多版本混用共存，可以说是为更远的未来版本做准备了。

去除事件池

在React17之前，如果使用异步的方式来获取事件e对象，会发现合成事件对象被销毁，如下：

```
function App() {
  const handleClick = (e: React.MouseEvent) => {
    console.log('直接打印e', e.target) // <button>React事件池
  }
  // v17以下在异步方法拿不到事件e，必须先调用 e.persist()
```

```
// e.persist()

// 异步方式获取事件e
setTimeout(() => {
  console.log('setTimeout打印e', e.target) // null
})
}

return (
  <div className="App">
    <button onClick={handleClick}>React事件池</button>
  </div>
)
}
```

```
直接打印e  <button>React事件池</button>                                         App.tsx:3
3 Warning: This synthetic event is reused for performance reasons. If you're seeing this, you're accessing the property 'target' on a released/nullified synthetic event. This is set to null. If you must keep the original synthetic event around, use event.persist(). See http://fb.me/react-event-pooling for more information. react-devtools-backend.js:4026
	setTimeout打印e null                                         App.tsx:8
```

如果你需要在事件处理函数运行之后获取事件对象的属性，你需要调用 `e.persist()`，它会将当前的合成事件从事件池中删除，并允许保留对事件的引用。

事件池：合成事件对象会被放入池中统一管理。这意味着合成事件对象可以被复用，当所有事件处理函数被调用之后，其所有属性都会被回收释放置空。

事件池的好处是在较旧浏览器中重用了不同事件的事件对象以提高性能，但它对现代浏览器的性能优化微乎其微，反而给开发者带来困惑，因此去除了事件池，因此也没有了事件复用机制。

```
function App() {
  // v17 去除了 React 事件池，异步方式使用e不再需要 e.persist()
  const handleClick = (e: React.MouseEvent) => {
    console.log('直接打印e', e.target) // <button>React事件池
  </button>

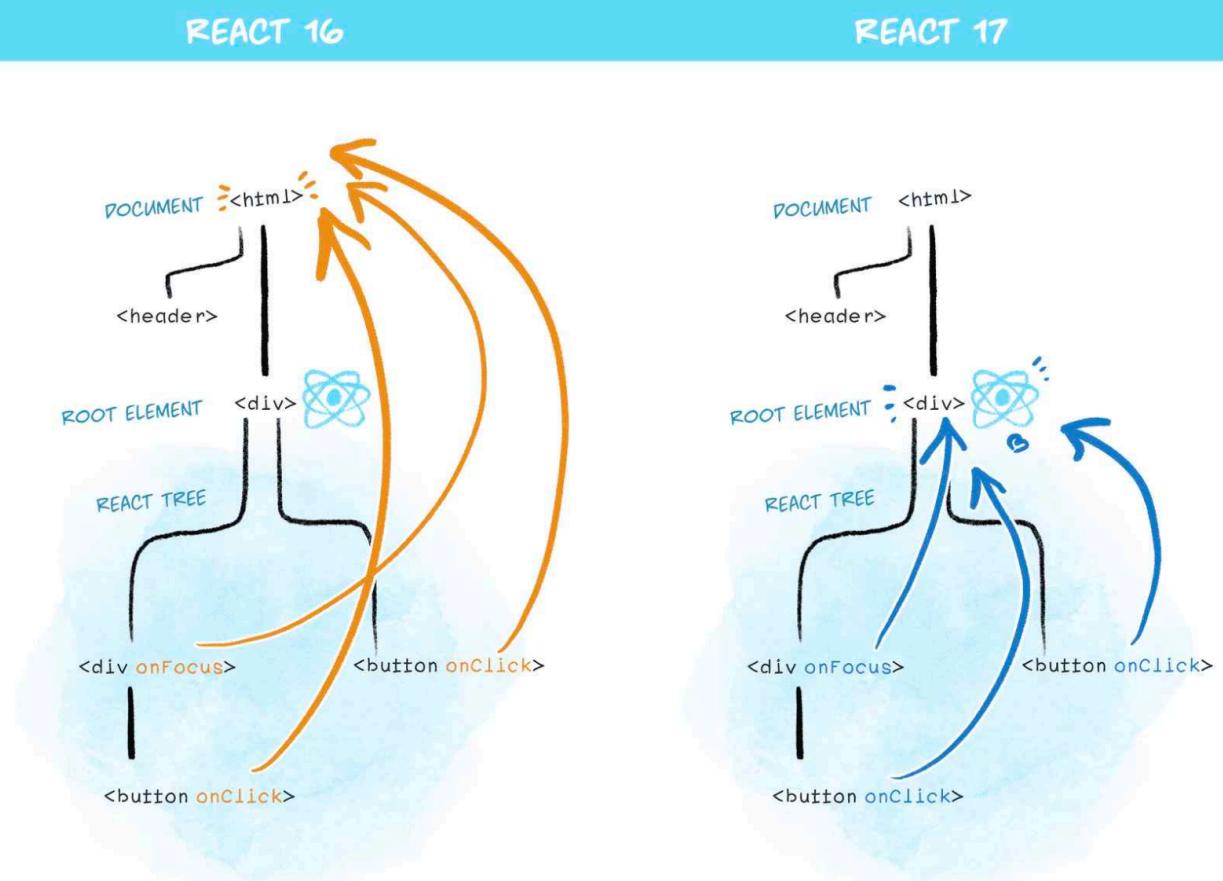
  setTimeout(() => {
    console.log('setTimeout打印e', e.target) // <button>React事件池</button>
  })
}
```

```
return (
  <div className="App">
    <button onClick={handleClick}>React事件池</button>
  </div>
)
```

事件委托到根节点

reactv17前，React 将事件委托到 document 上，在react17中，则委托到根节点

```
const rootNode = document.getElementById('root');
ReactDOM.render(<App />, rootNode);
```



```
import { useState, useEffect } from 'react'
```

```
function App() {
```

```
const [isShowText, setIsShowText] = useState(false)

const handleShowText = (e: React.MouseEvent) => {
  // e.stopPropagation() // v16无效
  // e.nativeEvent.stopImmediatePropagation() // 阻止监听同一事件
  // 的其他事件监听器被调用
  setIsShowText(true)
}

useEffect(() => {
  document.addEventListener('click', () => {
    setIsShowText(false)
  })
}, [])

return (
  <div className="App">
    <button onClick={handleShowText}>事件委托变更</button>

    {isShowText && <div>展示文字</div>}
  </div>
)
```

如上代码，在react16和v17版本，点击按钮时，都不会显示文字。这是因为react的合成事件是基于事件委托的，有事件冒泡，先执行React事件，再执行document上挂载的事件。

v16：出于对冒泡的了解，我们直接在按钮事件上加 `e.stopPropagation()`，这样就不会冒泡到document，`isShowText` 也不会被置为false了。但由于v16版本的事件委托是绑在 `document` 上的，它的事件源跟 `document` 就是同级了，而不是上下级，所以 `e.stopPropagation()` 并没有起作用。如果要阻止冒泡，可以使用原生的

`e.nativeEvent.stopImmediatePropagation()` 阻止同级冒泡，这样文字就可以显示了。

v17：由于事件委托到根目录root节点，与 `document` 属于上下级关系，所以可以直接使用 `e.stopPropagation()` 阻止

`stopImmediatePropagation()` 方法可以阻止监听同一事件的其他事件监听器被调用

这种更新不仅方便了局部使用 React 的项目，还可以用于项目的渐进式升级，解决不同版本的 React 组件嵌套使用时，`e.stopPropagation()`无法正常工作的问题

更贴近原生浏览器事件

对事件系统进行了一些较小的更改：

- `onScroll` 事件不再冒泡，以防止[出现常见的混淆](#)
- React 的 `onFocus` 和 `onBlur` 事件已在底层切换为原生的 `focusin` 和 `focusout` 事件。它们更接近 React 现有行为，有时还会提供额外的信息。

`blur`、`focus` 和 `focusin`、`focusout` 的区别：`blur`、`focus` 不支持冒泡，`focusin`、`focusout` 支持冒泡

- 捕获事件（例如，`onClickCapture`）现在使用的是实际浏览器中的捕获监听器。

这些更改会使 React 与浏览器行为更接近，并提高了互操作性。

尽管 React 17 底层已将 `onFocus` 事件从 `focus` 切换为 `focusin`，但请注意，这并未影响冒泡行为。在 React 中，`onFocus` 事件总是冒泡的，在 React 17 中会继续保持，因为通常它是一个更有用的默认值

全新的 JSX 转换器

总结下来就是两点：

- 用 `jsx()` 函数替换 `React.createElement()`
- 运行时自动引入 `jsx()` 函数，无需手写引入 `react`

在v16中，我们写一个React组件，总要引入

```
import React from 'react'
```

这是因为在浏览器中无法直接使用 jsx，所以要借助工具如 `@babel/preset-react` 将 jsx 语法转换为 `React.createElement` 的 js 代码，所以需要显式引入 React，才能正常调用 createElement。

通过 `React.createElement()` 创建元素是比较频繁的操作，本身也存在一些问题，无法做到性能优化，具体可见官方优化的 [动机](#)

v17之后，React 与 Babel 官方进行合作，直接通过将 `react/jsx-runtime` 对 jsx 语法进行了新的转换而不依赖 `React.createElement`，因此v17使用 jsx 语法可以不引入 React，应用程序依然能正常运行。

```
function App() {
  return <h1>Hello World</h1>;
}

// 新的 jsx 转换为
// 由编译器引入（禁止自己引入！）
import { jsx as _jsx } from 'react/jsx-runtime';

function App() {
  return _jsx('h1', { children: 'Hello world' });
}
```

如何升级至新的 JSX 转换

- 更新至支持新转换的 React 版本（v17）

如果你还在使用v16，也可升级至 React [v16.14.0](#) 的版本，官方在该版本也支持这个特性。

- 修改配置

1. `@babel/preset-react` 编译增加 `runtime: 'automatic'` 配置

```
// 如果你使用的是 @babel/preset-react
{
  "presets": [
    [ "@babel/preset-react", {
      "runtime": "automatic"
    } ]
  ]
}
```

```
]
}

// 如果你使用的是 @babel/plugin-transform-react-jsx
{
  "plugins": [
    ["@babel/plugin-transform-react-jsx", {
      "runtime": "automatic"
    }]
  ]
}
```

1. 修改 `tsconfig.json` 配置, 具体配置可见[TS官方文档](#)

```
{
  "compilerOptions": {
    // "jsx": "react",
    "jsx": "react-jsx",
  },
}
```

从 Babel 8 开始, "automatic" 会将两个插件默认集成在 runtime 中

副作用用清理时机

```
useEffect(() => {
  // This is the effect itself.
  return () => {
    // This is its cleanup.
  }
})
```

- v17前, 组件被卸载时, `useEffect` 的清理函数都是同步运行的; 对于大型应用程序来说, 同步会减缓屏幕的过渡 (如切换标签)
- v17后, `useEffect` 副作用清理函数是**异步执行的**, 如果要卸载组件, 则清理会在屏幕更新后运行

此外，v17 将在运行任何新副作用之前执行所有副作用的清理函数（针对所有组件），v16 只对组件内的副作用保证这种顺序。

不过需要注意

```
useEffect(() => {
  someRef.current.someSetupMethod();
  return () => {
    someRef.current.someCleanupMethod();
  };
})
```

问题在于 `someRef.current` 是可变的且因为异步的，在运行清除函数时，它可能已经设置为 `null`。

```
// 用一个变量量在 ref 每次变化时，将 someRef.current 保存起来，放到副作用清理回调函数的闭包中，来保证不可变性。
useEffect(() => {
  const instance = someRef.current
  instance.someSetupMethod()

  return () => {
    instance.someCleanupMethod()
  }
})
```

或者用 `useLayoutEffect`

```
useLayoutEffect(() => {
  someRef.current.someSetupMethod();
  return () => {
    someRef.current.someCleanupMethod();
  };
})
```

`useLayoutEffect` 可以保证回调函数同步执行，这样就能确保 `ref` 此时还是最后的值。

返回一致的 undefined 错误

在v17以前，组件返回 `undefined` 始终是一个错误。但是有漏网之鱼，React 只对类组件和函数组件执行此操作，但并不会检查 `forwardRef` 和 `memo` 组件的返回值。

```
function Button() {
  return; // Error: Nothing was returned from render
}

function Button() {
  // We forgot to write return, so this component returns
  // undefined.
  // React surfaces this as an error instead of ignoring it.
  <button />;
}
```

在 v17 中修复了这个问题，`forwardRef` 和 `memo` 组件的行为会与常规函数组件和类组件保持一致，在返回 `undefined` 时会报错

```
let Button = forwardRef(() => {
  // We forgot to write return, so this component returns
  // undefined.
  // React 17 surfaces this as an error instead of ignoring it.
  <button />;
});

let Button = memo(() => {
  // We forgot to write return, so this component returns
  // undefined.
  // React 17 surfaces this as an error instead of ignoring it.
  <button />;
});
```

原生组件栈

v16中错误调用栈的缺点：

- 缺少源码位置追溯，在控制台无法点击跳转到出错的地方
- 无法适用于生产环境

整体来说不如原生的 JavaScript 调用栈，不同于常规压缩后的 JavaScript 调用栈，它们可以通过 sourcemap 的形式自动恢复到原始函数的位置，而使用 React 组件栈，在生产环境下必须在调用栈信息和 bundle 大小间进行选择。

在v17使用了不同的机制生成组件调用栈，直接从 JavaScript 原生错误栈生成的，所以在生产环境也能按sourcemap 还原回来，且支持点击跳到源码位置。

想详细了解的可见该 [PR](#)

移除私有导出 API

v17 删除了一些私有 API，主要是 [React Native for Web](#) 使用的

另外，还删除了 `ReactTestUtils.SimulateNative` 工具方法，因为其行为与语义不符，如果你想要一种简便的方式来触发测试中原生浏览器的事件，可直接使用 [React Testing Library](#)

启发式更新算法更新

引用 [React17新特性：启发式更新算法](#)

- React16的 `expirationTimes` 模型只能区分是否 \geq `expirationTimes` 决定节点是否更新。
- React17的 `lanes` 模型可以选定一个更新区间，并且动态的向区间中增减优先级，可以处理更细粒度的更新。

React 18 更新

并发模式

v18的新特性是使用现代浏览器的特性构建的，彻底放弃对 IE 的支持。

v17 和 v18 的区别就是：从同步不可中断更新变成了异步可中断更新，v17可以通过一些试验性的API开启并发模式，而v18则全面开启并发模式。

并发模式可帮助应用保持响应，并根据用户的设备性能和网速进行适当的调整，该模式通过使渲染可中断来修复阻塞渲染限制。在 Concurrent 模式中，React 可以同时更新多个状态。

这里参考下文区分几个概念：

- **并发模式**是实现**并发更新**的基本前提
- v18 中，以是否使用**并发特性**作为是否开启**并发更新**的依据。
- **并发特性**指开启**并发模式**后才能使用的特性，比如：`useDeferredValue / useTransition`

可阅读参考 [Concurrent Mode（并发模式）](#)

更新 render API

v18 使用 ReactDOM.createRoot() 创建一个新的根元素进行渲染，使用该 API，会自动启用并发模式。如果你升级到v18，但没有使用 `ReactDOM.createRoot()` 代替 `ReactDOM.render()` 时，控制台会打印错误日志提醒你使用React，该警告也意味此项变更没有造成breaking change，而可以并存，当然尽量是不建议。

```
Warning: ReactDOM.render is no longer supported in React 18. Use createRoot instead. Until you switch to the new API, your app will behave as if it's running React 17. Learn more: https://reactjs.org/link/switch-to-createRoot
```

```
// v17
import ReactDOM from 'react-dom'
import App from './App'

ReactDOM.render(<App />, document.getElementById('root'))

// v18
import ReactDOM from 'react-dom/client'
import App from './App'

ReactDOM.createRoot(document.getElementById('root') as HTMLElement).render(<App />)
```

自动批处理

批处理是指 React 将多个状态更新，聚合到一次 render 中执行，以提升性能

在v17的批处理只会在事件处理函数中实现，而在Promise链、异步代码、原生事件处理函数中失效。而v18则所有的更新都会自动进行批处理。

```
// v17
const handleBatching = () => {
  // re-render 一次，这就是批处理的作用
  setCount((c) => c + 1)
  setFlag((f) => !f)
}

// re-render两次
setTimeout(() => {
  setCount((c) => c + 1)
  setFlag((f) => !f)
}, 0)

// v18
const handleBatching = () => {
  // re-render 一次
  setCount((c) => c + 1)
  setFlag((f) => !f)
}
```

```
// 自动批处理: re-render 一次
setTimeout(() => {
  setCount((c) => c + 1)
  setFlag((f) => !f)
}, 0)
```

如果在某些场景不想使用批处理，可以使用 `flushSync` 退出批处理，强制同步执行更新。

`flushSync` 会以函数为作用域，函数内部的多个 `setState` 仍然是批量更新

```
const handleAutoBatching = () => {
  // 退出批处理
  flushSync(() => {
    setCount((c) => c + 1)
  })
  flushSync(() => {
    setFlag((f) => !f)
  })
}
```

Suspense 支持 SSR

SSR 一次页面渲染的流程：

1. 服务器获取页面所需数据
2. 将组件渲染成 HTML 形式作为响应返回
3. 客户端加载资源
4. (hydrate) 执行 JS，并生成页面最终内容

上述流程是串行执行的，v18前的 SSR 有一个问题就是它不允许组件"等待数据"，必须收集好所有的数据，才能开始向客户端发送HTML。如果其中有一步比较慢，都会影响整体的渲染速度。

v18 中使用并发渲染特性扩展了 `Suspense` 的功能，使其支持流式 SSR，将 React 组件分解成更小的块，允许服务端一点一点返回页面，尽早发送 HTML 和选择性的 hydrate，从而可以使 SSR 更快的加载页面

```
<Suspense fallback={<Spinner />}>
  <Comments />
</Suspense>
```

具体可参考文章 [React 18 中新的 Suspense SSR 架构](#)

startTransition

`Transitions` 是 React 18 引入的一个全新的并发特性。它允许你将标记更新作为一个 `transitions`（过渡），这会告诉 React 它们可以被中断执行，并避免回到已经可见内容的 `Suspense` 降级方案。本质上是用于一些不是很急迫的更新上，用来进行并发控制

在 v18 之前，所有的更新任务都被视为急迫的任务，而 Concurrent Mode 模式能将渲染中断，可以让高优先级的任务先更新渲染。

React 的状态更新可以分为两类：

- 紧急更新：比如点击按钮、搜索框打字是需要立即响应的行为，如果没有立即响应给用户的体验就是感觉卡顿延迟
- 过渡/非紧急更新：将 UI 从一个视图过渡到另一个视图。一些延迟可以接受的更新操作，不需要立即响应

`startTransition` API 允许将更新标记为非紧急事件处理，被 `startTransition` 包裹的会延迟更新的 state，期间可能被其他紧急渲染所抢占。因为 React 会在高优先级更新渲染完成之后，才会渲染低优先级任务的更新

React 无法自动识别哪些更新是优先级更高的。比如用户的键盘输入操作后，`setInputValue` 会立即更新用户的输入到界面上，是紧急更新。而 `setSearchQuery` 是根据用户输入，查询相应的内容，是非紧急的。

```
const [inputValue, setInputValue] = useState()

const onChange = (e)=>{
  setInputValue(e.target.value) // 更新用户输入值（用户打字交互的优先级应该要更高）
  setSearchQuery(e.target.value) // 更新搜索列表（可能有点延迟，影响）
}

return (
  <input value={inputValue} onChange={onChange} />
)
```

React无法自动识别，所以它提供了 `startTransition` 让我们手动指定哪些更新是紧急的，哪些是非紧急的，从而让我们改善用户体验。

```
// 紧急的更新
setInputValue(e.target.value)
// 开启并发更新
startTransition(() => {
  setSearchQuery(input) // 非紧急的更新
})
```

这里有个比较好的[在线例子](#)，可以直接感受到 `startTransition` 的优化

useTransition

当有过渡任务（非紧急更新）时，我们可能需要告诉用户什么时候当前处于 pending（过渡）状态，因此v18提供了一个带有 `isPending` 标志的 Hook `useTransition` 来跟踪 transition 状态，用于过渡期。

`useTransition` 执行返回一个数组。数组有两个状态值：

- `isPending`：指处于过渡状态，正在加载中
- `startTransition`：通过回调函数将状态更新包装起来告诉 React这是一个过渡任务，是一个低优先级的更新

```
function TransitionTest() {
  const [isPending, startTransition] = useTransition()
```

```

const [count, setCount] = useState(0)

function handleClick() {
  startTransition(() => {
    setCount((c) => c + 1)
  })
}

return (
  <div>
    {isPending && <div>spinner...</div>}
    <button onClick={handleClick}>{count}</button>
  </div>
)
}

```

直观感觉这有点像 `setTimeout`，而防抖节流其实本质也是 `setTimeout`，区别是防抖节流是控制了执行频率，让渲染次数减少了，而 v18 的 transition 则没有减少渲染的次数。

useDeferredValue

`useDeferredValue` 和 `useTransition` 一样，都是标记了一次非紧急更新。`useTransition` 是处理一段逻辑，而 `useDeferredValue` 是产生一个新状态，它是延时状态，这个新的状态则叫 `DeferredValue`。所以使用 `useDeferredValue` 可以推迟状态的渲染

`useDeferredValue` 接受一个值，并返回该值的新副本，该副本将推迟到紧急更新之后。如果当前渲染是一个紧急更新的结果，比如用户输入，React 将返回之前的值，然后在紧急渲染完成后渲染新的值。

```

function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizing 告诉 React 仅当 deferredQuery 改变,
  // 而不是 query 改变的时候才重新渲染
  const suggestions = useMemo(() =>
    <SearchSuggestions query={deferredQuery} />,
  )
}

```

```
[deferredQuery]
);

return (
<>
<SearchInput query={query} />
<Suspense fallback="Loading results...">
  {suggestions}
</Suspense>
</>
);
}
```

这样一看，`useDeferredValue` 直观就是延迟显示状态，那用防抖节流有什么区别呢？

如果使用防抖节流，比如延迟300ms显示则意味着所有用户都要延时，在渲染内容较少、用户CPU性能较好的情况下也是会延迟300ms，而且你要根据实际情况来调整延迟的合适值；但是 `useDeferredValue` 是否延迟取决于计算机的性能。

- 感兴趣可以看下这篇文章：[usedeferredvalue-in-react-18](#)
- [在线例子](#)

useId

`useId` 支持同一个组件在客户端和服务端生成相同的唯一的 ID，避免 hydration 的不匹配，原理就是每个 id 代表该组件在组件树中的层级结构。

```
function Checkbox() {
  const id = useId()
  return (
    <>
      <label htmlFor={id}>Do you like React?</label>
      <input id={id} type="checkbox" name="react" />
    </>
  )
}
```

这里涉及到 SSR 部分知识，这里不展开了，可以阅读该篇文章理解：

- 为了生成唯一id，React18专门引入了新Hook：useId

提供给第三方库的 Hook

这两个 Hook 日常开发基本用不到，简单带过

useSyncExternalStore

`useSyncExternalStore` 一般是第三方状态管理库使用如 `Redux`。它通过强制的同步状态更新，使得外部 store 可以支持并发读取。它实现了对外部数据源订阅时不再需要 `useEffect``

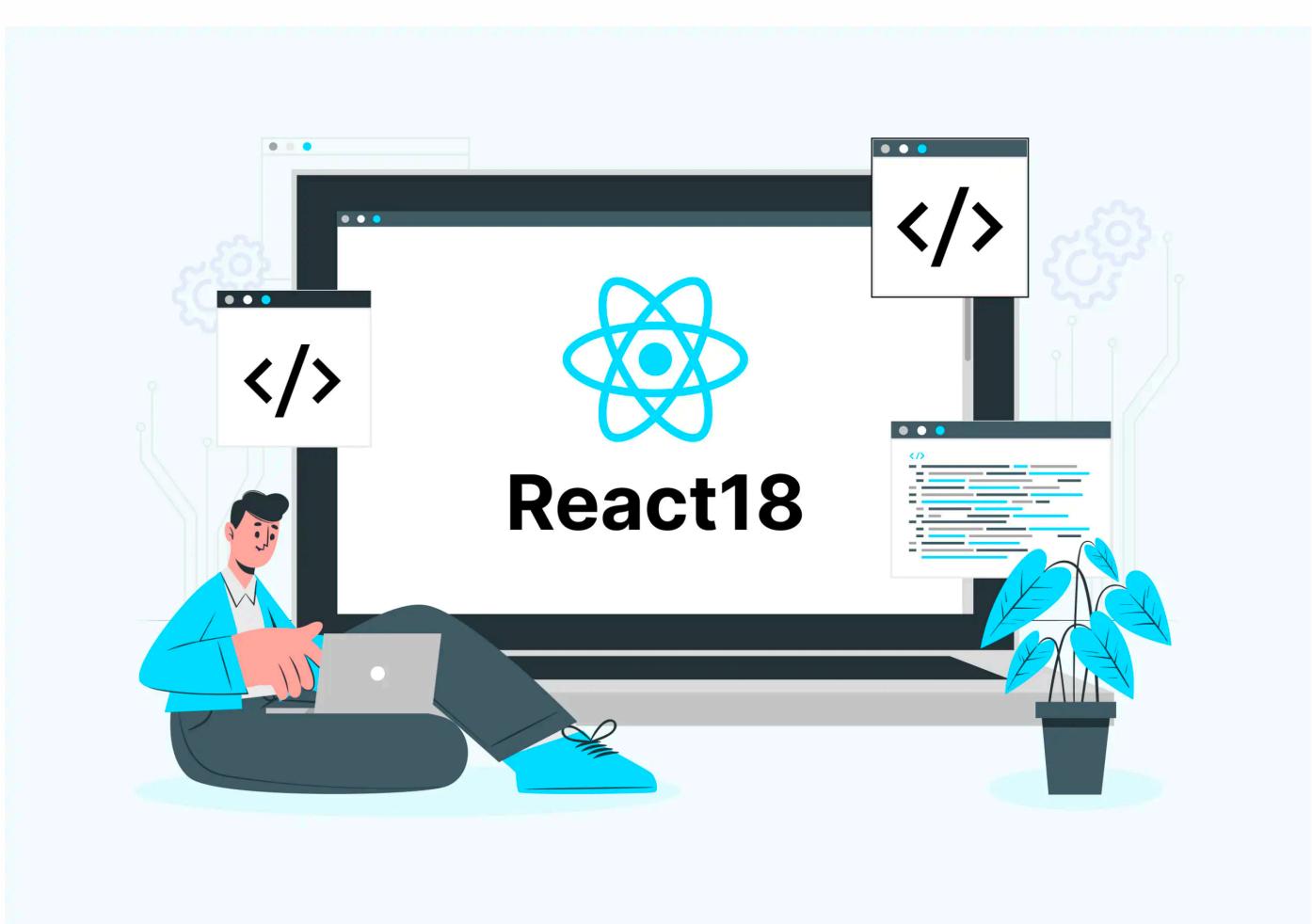
```
const state = useSyncExternalStore(subscribe, getSnapshot[,  
getServerSnapshot]);
```

useInsertionEffect

`useInsertionEffect` 仅限于 css-in-js 库使用。它允许 css-in-js 库解决在渲染中注入样式的性能问题。执行时机在 `useLayoutEffect` 之前，只是此时不能使用ref和调度更新，一般用于提前注入样式。

```
useInsertionEffect(() => {  
  console.log('useInsertionEffect 执行')  
}, [])
```

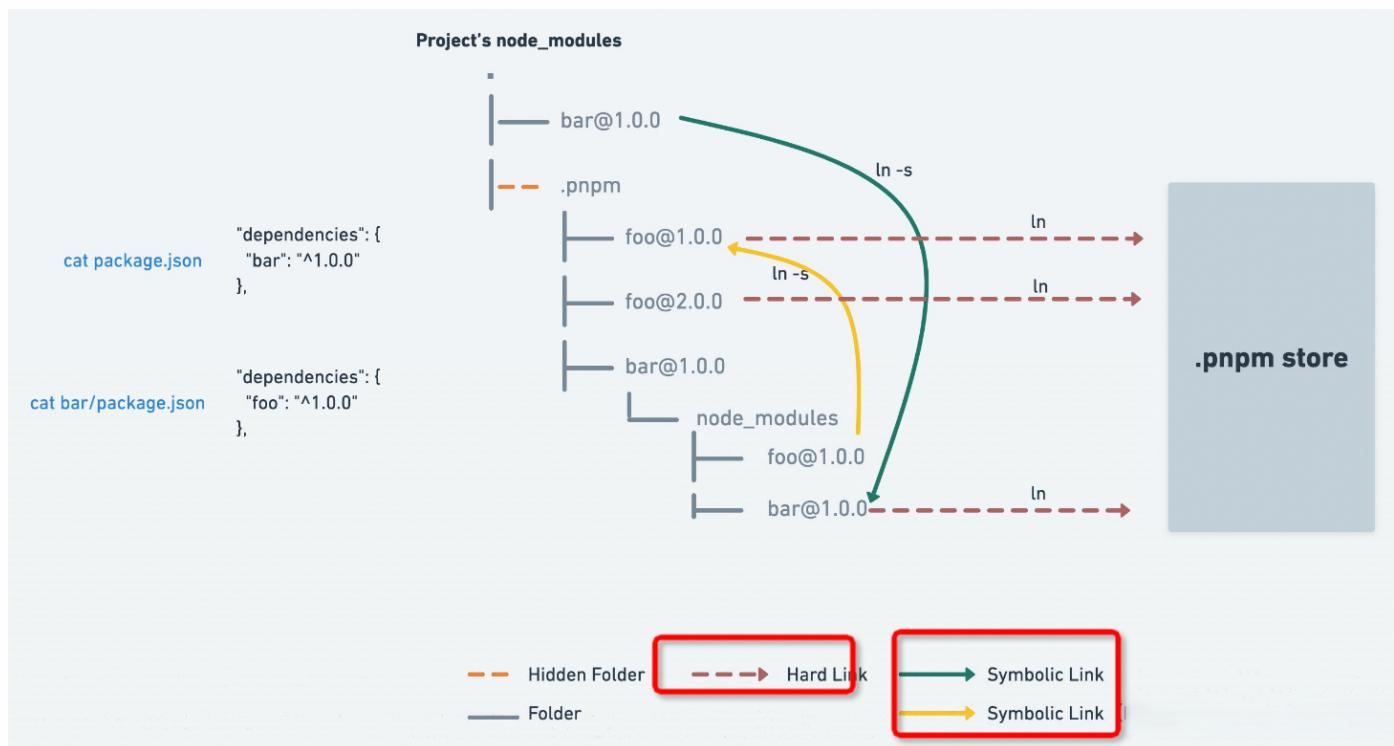
React18源码学习环境准备



1.<https://github.com/facebook/react>

千万不要读野文档！！！！

2.pnpm原理



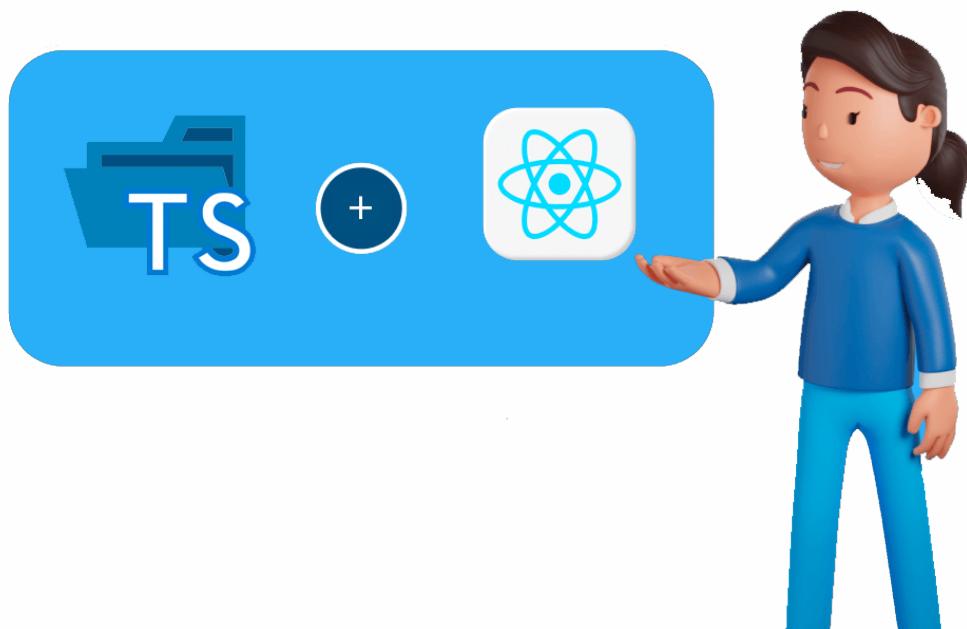
3.mono-repo常用命令

```
pnpm install xx -D #安装给根项目  
pnpm i lodash --filter #具体项目  
pnpm add lodash -r/-w #为每个项目或全局添加依赖  
pnpm run build --filter #具体要执行的项目  
pnpm install @qftjs/monorepo2 -r --filter @qftjs/monorepo1 #pkg1  
中将 pkg2 作为依赖进行安装。
```

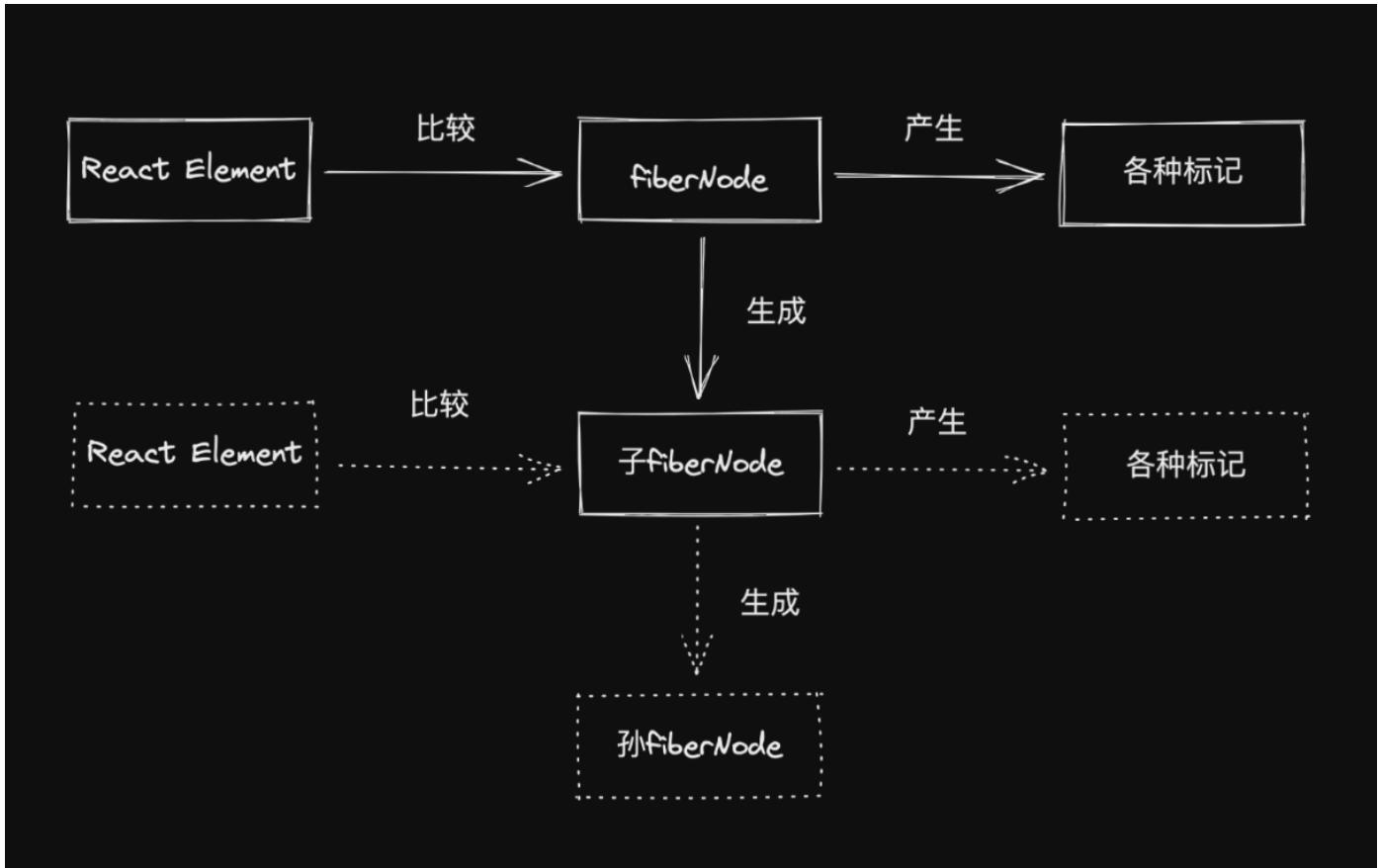


手写React18源码核心细节

节



1. Reconciler的工作方式



当前我们了解的节点类型：

- JSX
- ReactElement
- FiberNode
- DOMElement

比如，挂载 `<div></div>`：

```

// React Element <div></div>
jsx("div")
// 对应fiberNode
null
// 生成子fiberNode
// 对应标记
Placement

```

将 `<div></div>` 更新为 `<p></p>`：

```
// React Element <p></p>
jsx("p")
// 对应fiberNode
FiberNode {type: 'div'}
// 生成子fiberNode
// 对应标记
Deletion Placement
```

当所有 `ReactElement` 比较完后，会生成一棵 `fiberNode树`，一共会存在两棵 `fiberNode树`：

- `current`: 与视图中真实UI对应的 `fiberNode树`
- `workInProgress`: 触发更新后，正在 `reconciler` 中计算的 `fiberNode树`

2. 实现状态更新机制

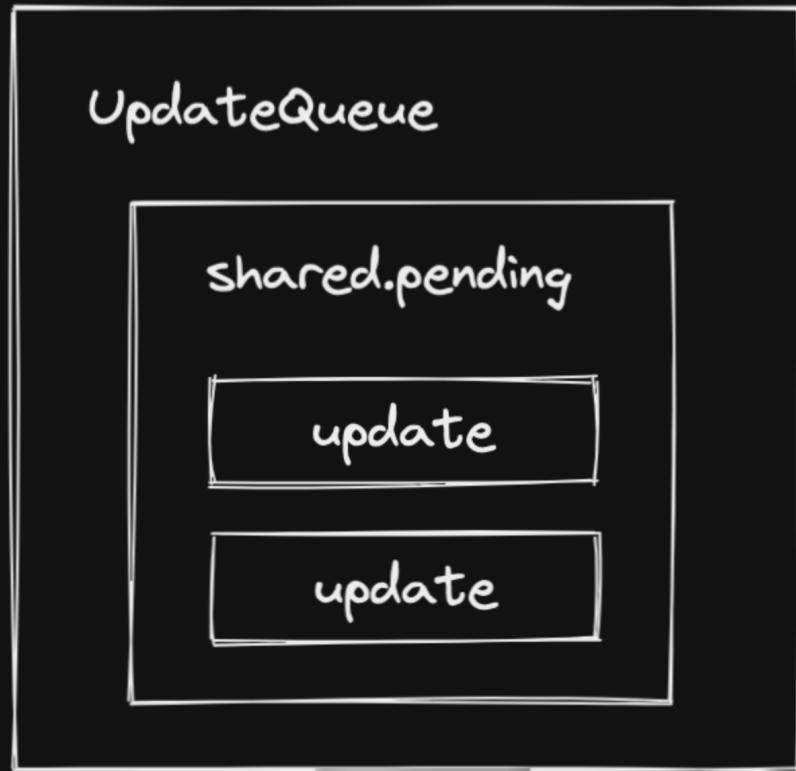
常见的触发更新的方式：

- `ReactDOM.createRoot().render` (或老版的 `ReactDOM.render`)
- `this.setState`
- `useState` 的 `dispatch` 方法

希望实现一套统一的更新机制~

更新机制的组成部分

- 代表更新的数据结构 —— `Update`
- 消费 `update` 的数据结构 —— `UpdateQueue`

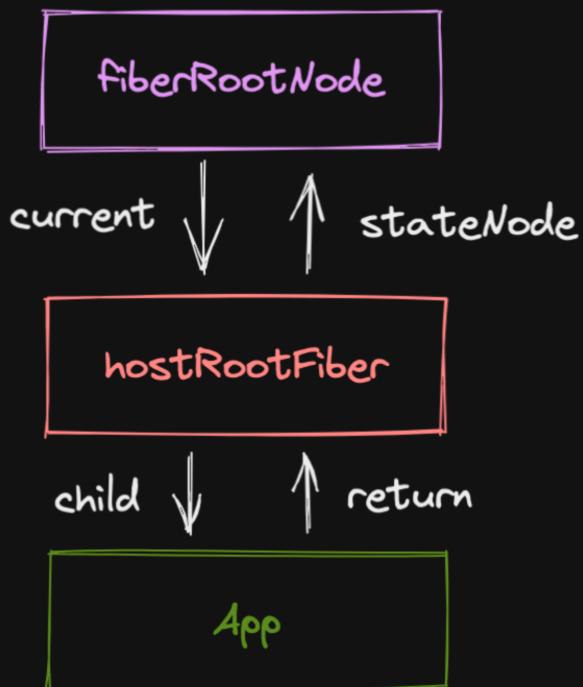


需要考慮的事情：

- 更新可能发生于任意组件，而更新流程是从根节点递归的
- 需要一个统一的根节点保存通用信息

```
ReactDOM.createRoot(rootElement).render(<App/>)
```

```
ReactDOM.createRoot( rootElement ).render( <App/> )
```



在React Fiber架构中，有两个概念叫做 `HostRootFiber` 和 `FiberRootNode`。这两个概念都是React内部实现的一部分，并且都与React组件树的根部相关。然而，它们在React内部的角色是不同的。

- `HostRootFiber`：这是一个特殊类型的Fiber节点，代表React应用的根。每个React应用都有一个对应的 `HostRootFiber`。在Fiber架构中，每个React组件都对应一个Fiber节点，这个节点包含了该组件的状态、属性（props）等信息，以及对父节点、子节点和兄弟节点的引用。`HostRootFiber` 也不例外，它的子节点通常是应用的顶级组件。
- `FiberRootNode`：这是一个存储在React内部的对象，代表React应用的根。它包含了很多关于React应用状态的信息，例如当前的Fiber树、下一个待处理的工作单元等。它的 `current` 属性指向了当前有效的 `HostRootFiber`。

所以，`HostRootFiber` 和 `FiberRootNode` 都代表了React应用的根，但一个是在Fiber树中的节点，一个是存储应用状态的对象。在处理React应用时，React内部会通过 `FiberRootNode` 来访问和管理应用的状态，通过 `HostRootFiber` 来处理组件的渲染和更新。

更深入一些具体细节

`FiberRootNode` 和 `HostRootFiber` 是React Fiber架构中的两个关键概念。 `FiberRootNode` 和 `HostRootFiber` 都代表着React应用的根，但它们的角色和用途是有所区别的。

- `FiberRootNode`：这是一个React内部的数据结构，它包含了许多关于React应用的信息，以及对应用状态的引用。它的一些主要属性包括：
 - `current`：指向当前活动的 `HostRootFiber`。React在渲染过程中可能会有两棵Fiber树在交替工作（这是为了实现React的异步渲染和时间切片功能），`current` 属性始终指向当前应用的状态。
 - `finishedWork`：在React完成一次更新后，`finishedWork` 属性会被设置为新的 `HostRootFiber`。然后，React会在提交阶段将这个新的Fiber树提交给渲染器，以便更新DOM。
- `HostRootFiber`：这是Fiber树中的一个特殊类型的节点，代表了React应用的根。每一个Fiber节点都代表了一个React元素或组件，而 `HostRootFiber` 则代表了整个应用。它的一些主要属性包括：
 - `child`：指向应用的顶级组件。例如，如果你的React应用由一个 `<App>` 组件构成，那么 `HostRootFiber` 的 `child` 属性将会指向代表 `<App>` 的Fiber节点。
 - `return`：对于所有的Fiber节点（除了 `HostRootFiber`），`return` 属性会指向它们的父节点。但对于 `HostRootFiber`，`return` 属性是 `null`，因为它没有父节点。

`FiberRootNode` 和 `HostRootFiber` 的关系是，`FiberRootNode` 的 `current` 属性会指向一个 `HostRootFiber`，而这个 `HostRootFiber` 则代表了应用当前的状态。当React完成一次更新后，`FiberRootNode` 的 `finishedWork` 属性会被设置为新的 `HostRootFiber`，然后React会在提交阶段将这个新的Fiber树提交给渲染器，以便更新DOM。

createContainer -**> hostRootFiber -**> updateQueue 返回<**-
FiberRootNode

FiberRootNode -**> hostRootFiber -**> updateQueue -**>
createUpdate(element) -**> 状态更新 -**> scheduleUpdateOnFiber

```
-**>markUpdateFromFiberToRoot -**> renderRoot -**> prepareFreshStack  
-**> createWorkInProgress -**> 第一次 mount阶段 ->
```

3.Render+Commit阶段实现

react内部3个阶段：

- schedule阶段
- render阶段 mount-> (beginWork completeWork) 入口函数： **workLoop**
- commit阶段 (commitWork) 入口函数： **commitRoot**

commit阶段的3个子阶段

- beforeMutation阶段
- mutation阶段 (突变阶段 意味着元素属性变化等)
- layout阶段 (useLayout等Hook发生的阶段)
- workInProgress和current切换就发生在mutation->layout之间

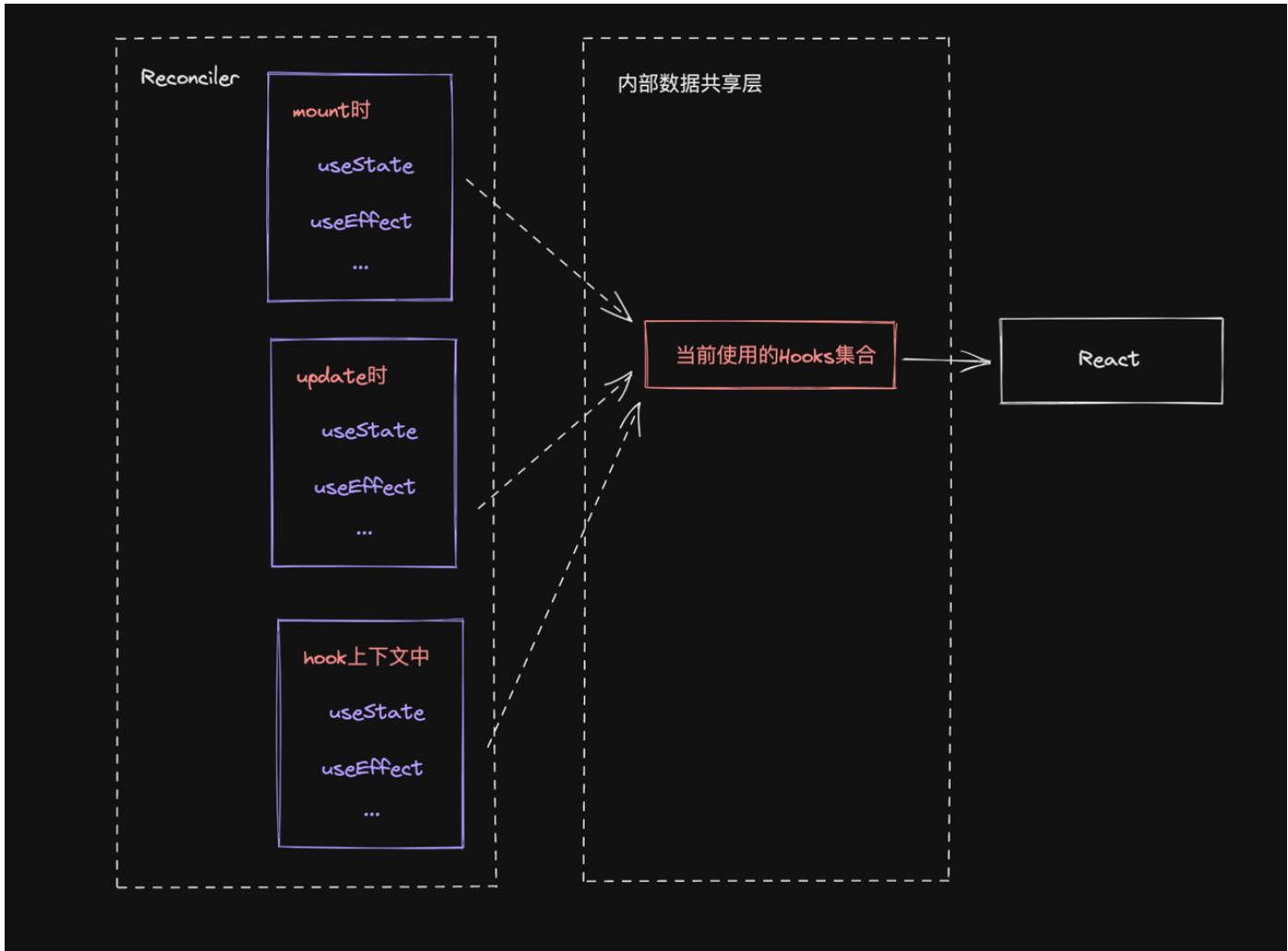
当前 **commit** 阶段要执行的任务：

- **fiber** 树的切换
- 执行 **Placement** 等对应操作 (flags和subtreeFlags的标记📌)

4.实现useState

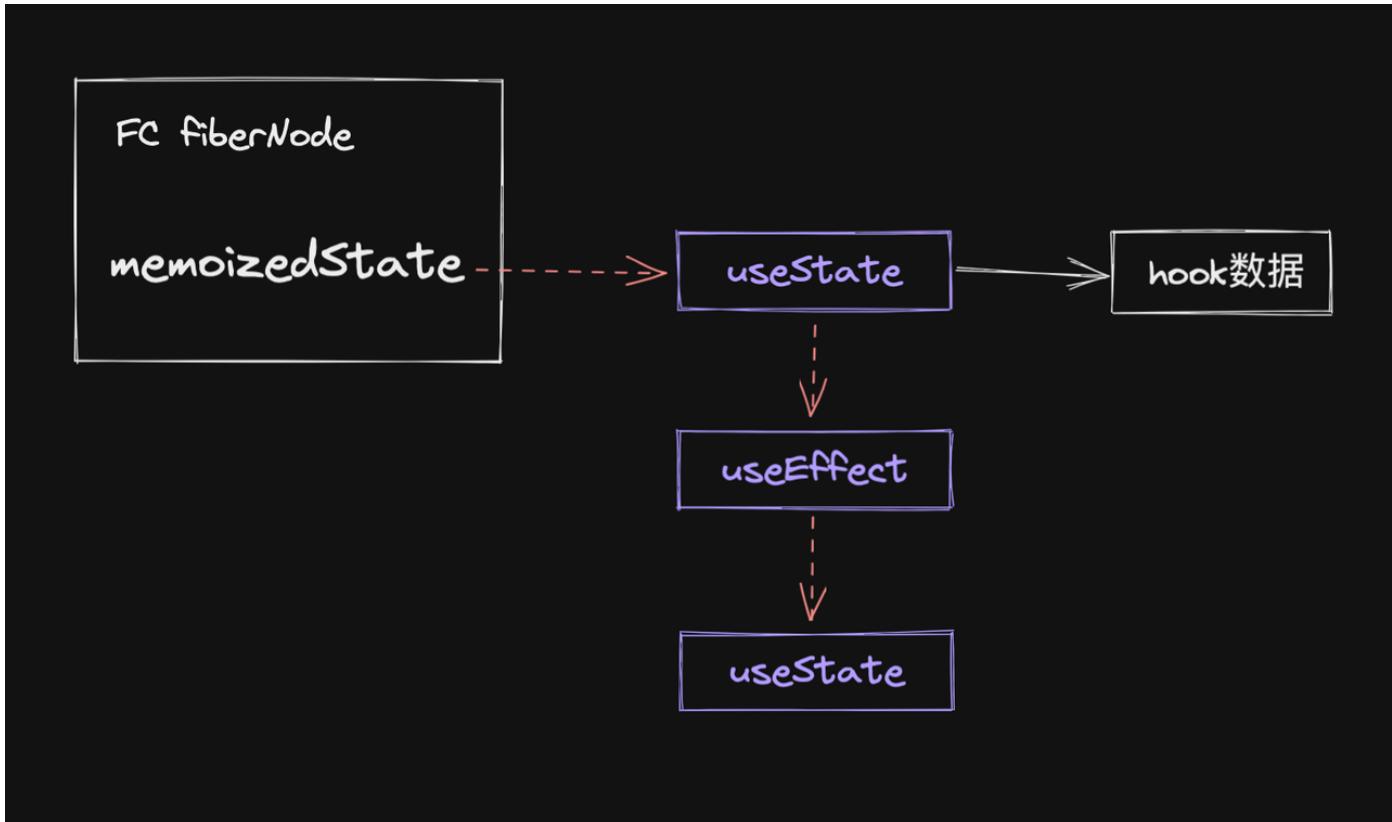
- **hook** 不能脱离 **FC** 上下文， **hook** 怎么知道当前是 **mount** 还是 **update** ?

解决方案： 「在不同上下文中调用的hook不是同一个函数」。



- **hook** 如何知道自身数据保存在哪?

「可以记录当前正在render的FC对应fiberNode，在fiberNode中保存hook数据」 Hooks的数据结构如下：



4.初探update流程

`update` 流程与 `mount` 流程的区别。

对于 `beginWork` :

- 需要处理 `ChildDeletion` 的情况
- 需要处理节点移动的情况 (`abc -> bca`)

对于 `completeWork` :

- 需要处理 `HostText` 内容更新的情况
- 需要处理 `HostComponent` 属性变化的情况

对于 `commitWork` :

- 对于 `ChildDeletion` , 需要遍历被删除的子树
- 对于 `Update` , 需要更新文本内容

对于 `useState` :

- 实现相对于 `mountState` 的 `updateState`

beginWork流程

- singleElement
- singleTextNode

处理流程为：

1. 比较是否可以复用 `current fiber`
2. 1. 比较 `key`，如果 `key` 不同，不能复用
2. 比较 `type`，如果 `type` 不同，不能复用
3. 如果 `key` 与 `type` 都相同，则可复用
3. 不能复用，则创建新的（同 `mount` 流程），可以复用则复用旧的

注意：对于同一个 `fiberNode`，即使反复更新，`current`、`wip` 这两个 `fiberNode` 会重复使用

completeWork流程

主要处理「标记Update」的情况，本节课我们处理 `HostText` 内容更新的情况。

commitWork流程

对于标记 `ChildDeletion` 的子树，由于子树中：

- 对于 `FC`，需要处理 `useEffect` `unmount` 执行、解绑 `ref`
- 对于 `HostComponent`，需要解绑 `ref`
- 对于子树的 `根HostComponent`，需要移除 `DOM`

所以需要实现「遍历ChildDeletion子树」的流程

对于useState

需要实现：

- 针对 `update` 时的 `dispatcher`
- 实现对标 `mountWorkInProgressHook` 的 `updateWorkInProgressHook`

- 实现 `updateState` 中「计算新state的逻辑」

其中 `updateWorkInProgressHook` 的实现需要考虑的问题：

- hook数据从哪来？
- 交互阶段触发的更新

5. 实现合成事件系统

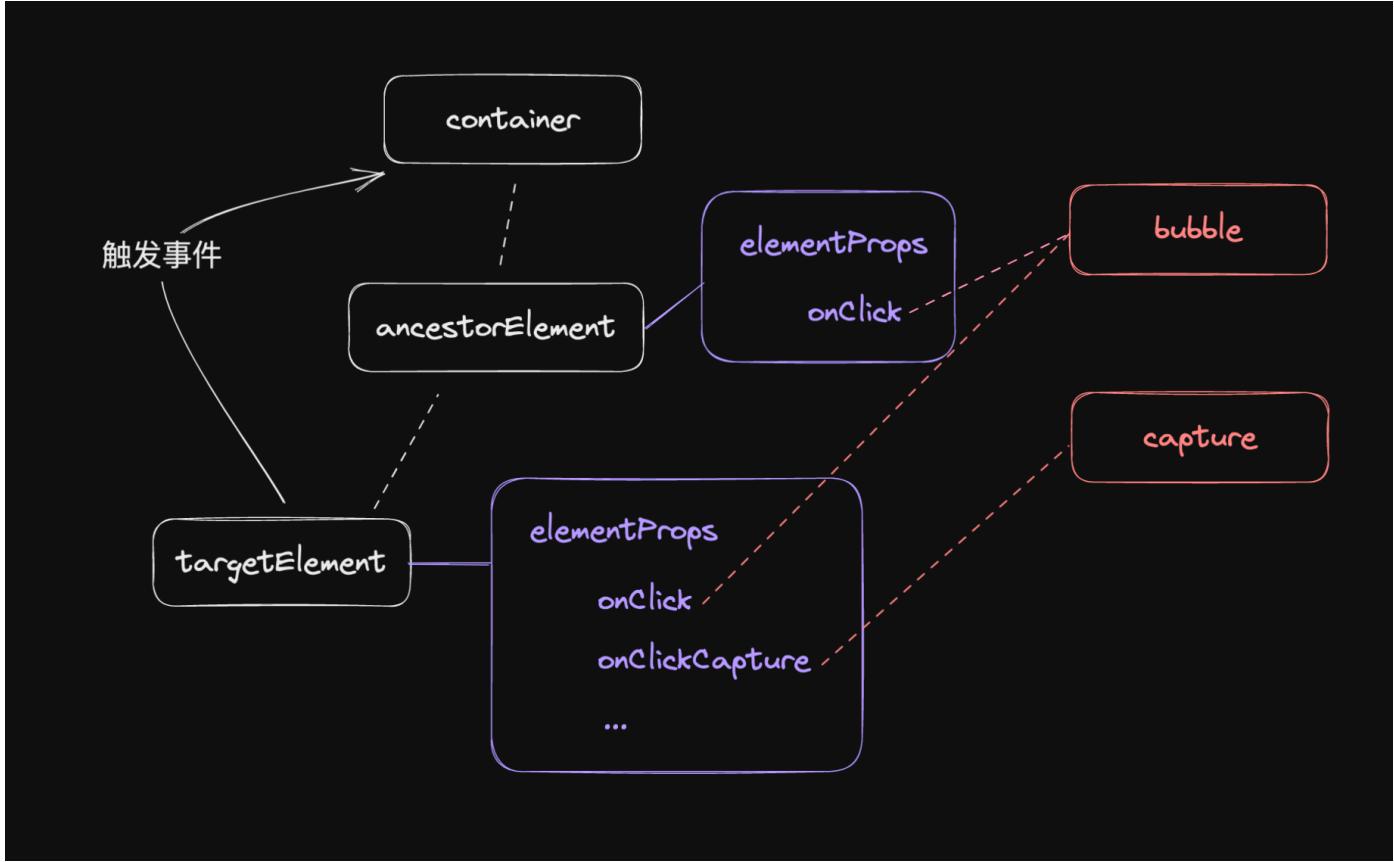
React的事件系统称为“合成事件系统”，原因是React在浏览器原生的事件系统之上实现了一套自己的事件系统。这个系统在内部创建了一种被称为“合成事件”的跨浏览器兼容性良好的事件对象。

React的合成事件与浏览器提供的原生事件相似，它们都提供了相同的接口，包括像 `stopPropagation()` 和 `preventDefault()` 这样的方法，以及像 `type`、`target` 等事件属性。然而，这些合成事件并不是浏览器原生事件的直接引用。它们是React内部创建的新对象，提供了和浏览器原生事件相同的接口，但具有更高的浏览器兼容性。

使用合成事件系统的好处是，React可以确保所有的事件在所有浏览器中具有一致的行为，因此开发者不需要担心处理兼容性问题。此外，由于React在事件系统上使用了事件委托，即所有的事件都被挂载在文档的根节点上，这样可以减少内存消耗，提高事件处理的性能。

最后，React的合成事件也让事件行为的模拟变得更容易，这对于在React的组件测试中模拟用户交互是非常有用的。

因此，虽然React的合成事件系统与浏览器的原生事件系统在工作方式上存在一些不同，但它们提供了相同的接口，使得在React中处理事件变得更简单、更一致，并具有更好的性能和可测试性。



6. 实现Dom Diff算法

当前支持的情况：

- A1 -> B1
- A1 -> A2

需要支持的情况：

- ABC -> A

「单/多节点」是指「更新后是单/多节点」。

更细致的，我们需要区分4种情况：

- `key` 相同, `type` 相同 == 复用当前节点

例如：A1 B2 C3 -> A1

- `key` 相同, `type` 不同 == 不存在任何复用的可能性

例如：A1 B2 C3 -> B1

- `key` 不同, `type` 相同 == 当前节点不能复用
- `key` 不同, `type` 不同 == 当前节点不能复用

整体流程分为4步。

1. 将 `current` 中所有同级 `fiber` 保存在 `Map` 中
2. 遍历 `newChild` 数组, 对于每个遍历到的 `element`, 存在两种情况:
3. ◦ 在 `Map` 中存在对应 `current fiber`, 且可以复用
 - 在 `Map` 中不存在对应 `current fiber`, 或不能复用
4. 判断是插入还是移动 (如下)
5. 最后 `Map` 中剩下的都标记删除

插入/移动判断 详解(reconcileChildrenArray函数的步骤③)

「移动」具体是指「向右移动」

移动的判断依据: `element` 的 `index` 与 「`element` 对应 `current fiber`」 的 `index` 的比较

A1 B2 C3 -> B2 C3 A1

012 012

当遍历 `element` 时, 「当前遍历到的 `element`」 一定是 「所有已遍历的 `element`」 中最靠右那个。

所以只需要记录 「最后一个可复用 `fiber`」 在 `current` 中的 `index` (`lastPlacedIndex`), 在接下来的遍历中:

- 如果接下来遍历到的 「可复用 `fiber`」 的 `index` < `lastPlacedIndex`, 则 标记 `Placement`
- 否则, 不标记

7. 实现Lanes（同步和异步调度流程）

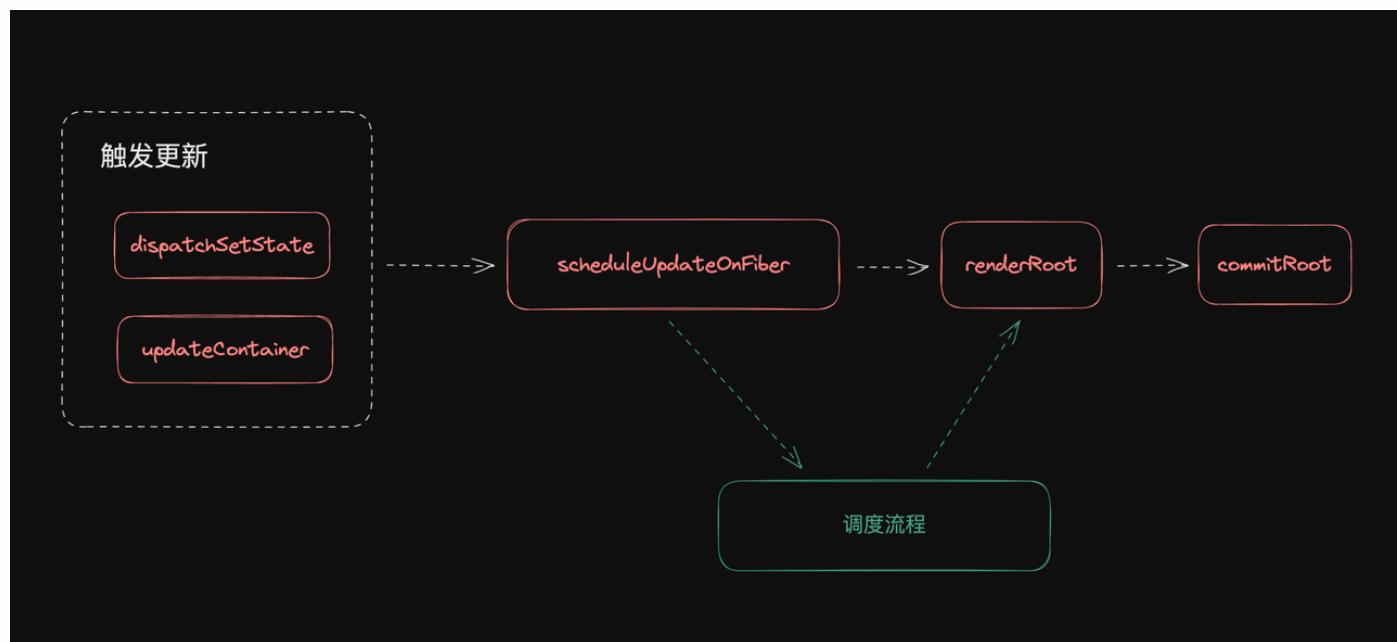
React 批处理的时机既有宏任务，也有微任务。启动并发更新的时候任务进入了微任务，其他更新在宏任务。React16版本的更新是同步更新的，但是用了时间碎片和MessageChannel。React18保留了**MessageChannel**增加了**setImmediate**的支持。所以日常状态更新在宏任务里，并行状态更新在微任务里。具体放弃时间碎片更新的方式请参考之前课程。

新增调度阶段

既然我们需要「多次触发更新，只进行一次更新流程」，意味着我们要将更新合并，所以在：

- **render** 阶段
- **commit** 阶段

的基础上增加 **schedule** 阶段（调度阶段）



对update的调整

「多次触发更新，只进行一次更新流程」中「多次触发更新」意味着对于同一个 **fiber**，会创建多个 **update**：

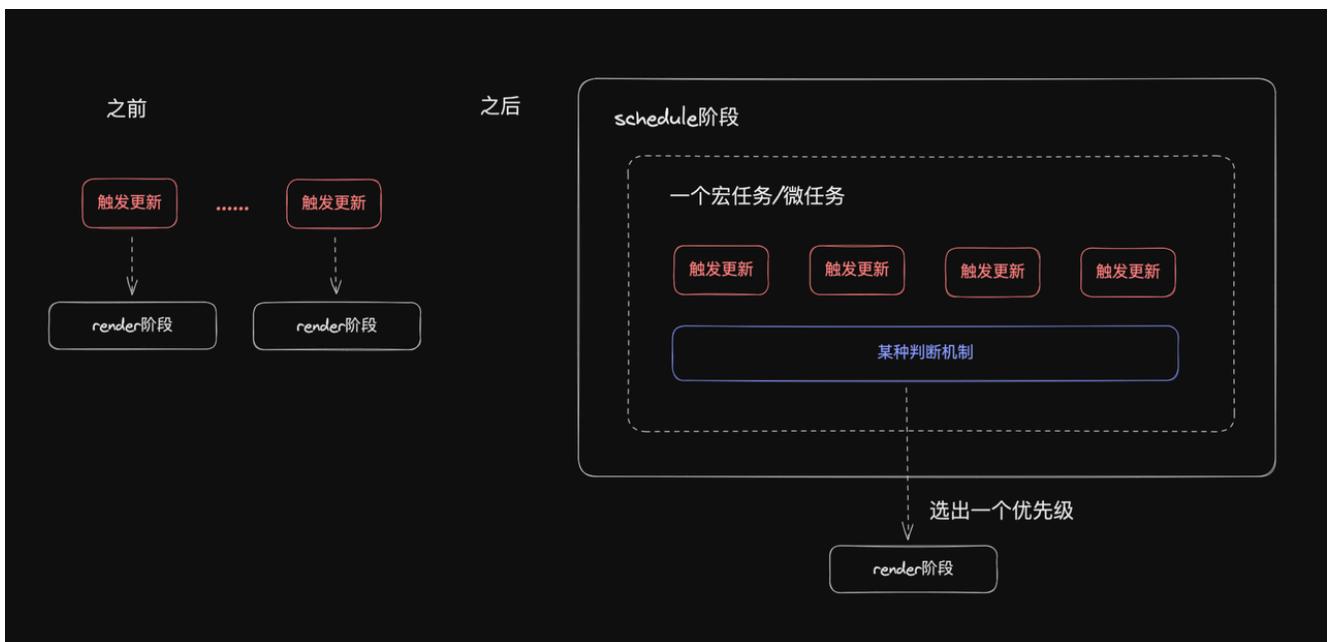
```

const onClick = () => {
  // 创建3个update
  updateCount((count) => count + 1);
  updateCount((count) => count + 1);
  updateCount((count) => count + 1);
};

```

「多次触发更新，只进行一次更新流程」，意味着要达成3个目标：

1. 需要实现一套优先级机制，每个更新都拥有优先级
2. 需要能够合并一个宏任务/微任务中触发的所有更新
3. 需要一套算法，用于决定哪个优先级优先进入render阶段



实现目标1：Lane模型

包括：

- `lane` (二进制位，代表优先级)
- `lanes` (二进制位，代表 `lane` 的集合)

其中：

- `lane` 作为 `update` 的优先级

- `lanes` 作为 `lane` 的集合

lane的产生

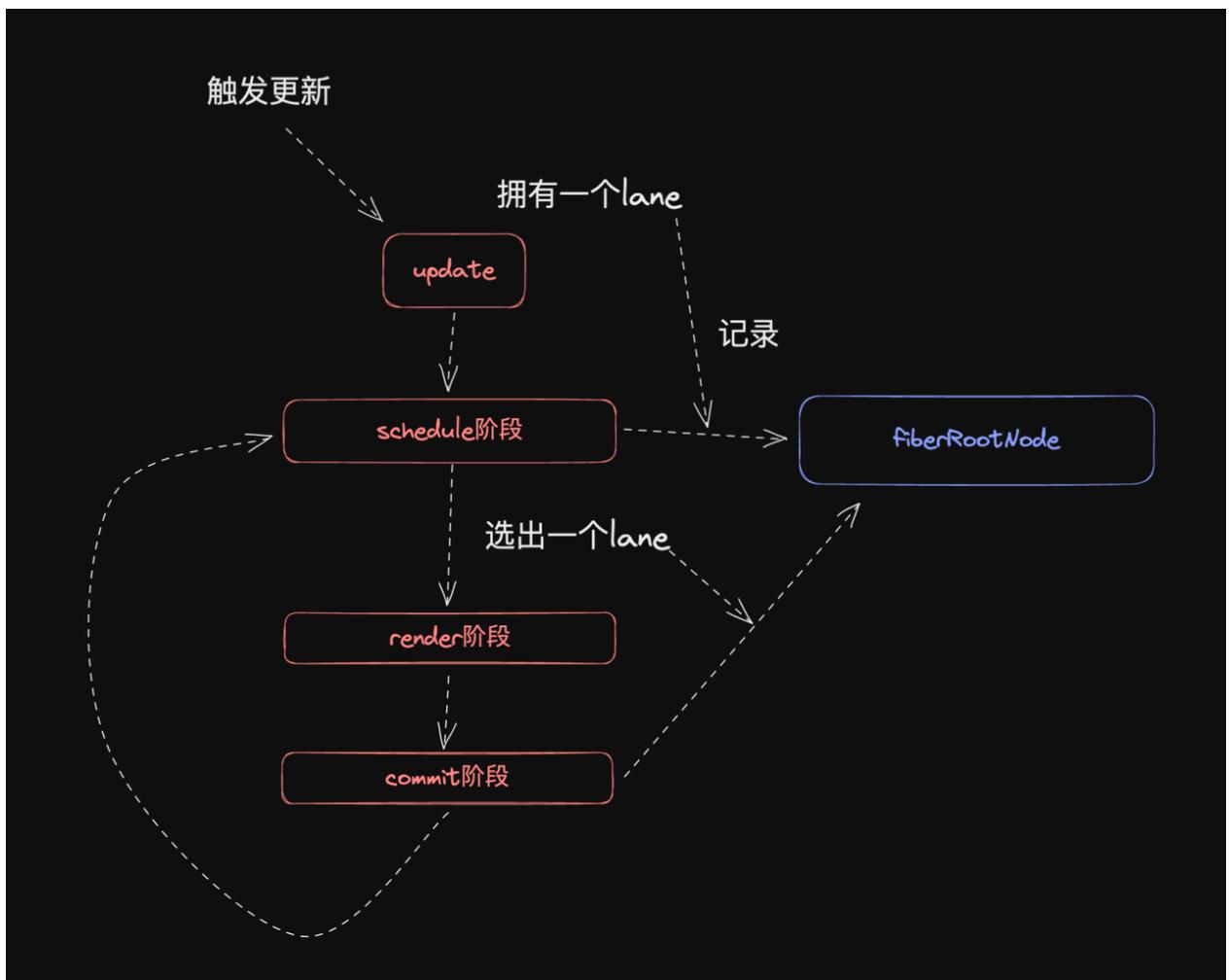
对于不同情况触发的更新，产生 `lane`。为后续不同事件产生不同优先级更新做准备。

如何知道哪些 lane 被消费，还剩哪些 lane 没被消费？

Lane对FiberRootNode的改造

需要增加如下字段：

- 代表所有未被消费的 lane 的集合
 - 代表本次更新消费的 lane lan2



实现目标2、3

需要完成两件事：

- 实现「某些判断机制」，选出一个 `lane`
- 实现类似防抖、节流的效果，合并宏/微任务中触发的更新

render阶段的改造

`processUpdateQueue` 方法消费 `update` 时需要考虑：

- `lane` 的因素
- `update` 现在是一条链表，需要遍历

commit阶段的改造

移除「本次更新被消费的`lane`」。

8.面试手写useEffect

```
const _memoizedState = []; // 多个 hook 存放在这个数组
let _idx = 0; // 当前 memoizedState 下标
const _effectDestroy = []; // 存储多个 useEffect 回调函数
/** 
 * 模拟实现 useEffect
 * // deps 的不同对应着不同的情况,
 * // 1. deps 不存在时：每次 state 的更新，都需要执行 callback
 * // 2. deps 存在，但数组为空时，只需要在挂载也就是初次渲染时执行
callback
 * // 3. deps 存在且有依赖项，则对应的依赖性更新时才执行 callback
 * @param {Function} callback 回调函数
 * @param {Array} deps 依赖项
 */
function useEffect(callback, deps) {
    if (Object.prototype.toString.call(depsAry) !== '[object Array]') {
        throw new Error('useEffect 函数的第二个参数必须是数组');
    }
    // 先根据当前下标获取到存储在全局 hooks 列表中当前位置原本的依赖项
    const memoizedDeps = _memoizedState[_idx];
```

```

// 如果当前没有, 则证明是初次渲染, 无论什么情况都执行一次 callback
if(!memoizedDeps) {
    const destroy = callback();
    // 同时更新依赖项
    _memoizedState[_idx] = deps;
    // 如果 callback 返回值是一个函数, 则先把函数存储在全局的
    // destroy 数组中, 随后在willUnmount阶段依次执行
    if(typeof destroy === "function") {
        _effectDestroy.push(destroy);
    }
    // 否则就是 重新渲染 的阶段
} else {
    // 没有依赖项直接执行 callback
    if(!deps) {
        callback();
    } else {
        // 依赖项不为空数组的时候且依赖项有更新了才去执行 callback
        deps.length !== 0 && !deps.every((dep, idx) => dep
        === memoizedDeps[idx]) && callback();
        // 别忘了更新依赖项
        _memoizedState[_idx] = deps;
    }
}
_idx++;
}

```

```

let onClick;
let onChange;
const willUnmount = () => {
    for(let destroy of _effectDestroy) {
        destroy();
    }
}

function render() {
    // _idx 表示当前执行到的 hooks 的位置
    // _idx 重新置为 0, 也是契合react每次更新时都从 hooks 头节点开始更新
    // 每一个 hook
    _idx = 0;
    const [count, setCount] = useState(0);
    const [name, setName] = useState("77");
}

```

```
useEffect(()=>{
    console.log("effect — count", count);
    return () => {
        console.log("count Effect Destroy");
    }
}, [count])
useEffect(()=>{
    console.log("effect — name", name);
}, [name])
// 使用 onClick, onChange 简单模拟一下更新操作
onClick = () => { setCount(count + 1) };
onChange = (name) => { setName(name) };
}

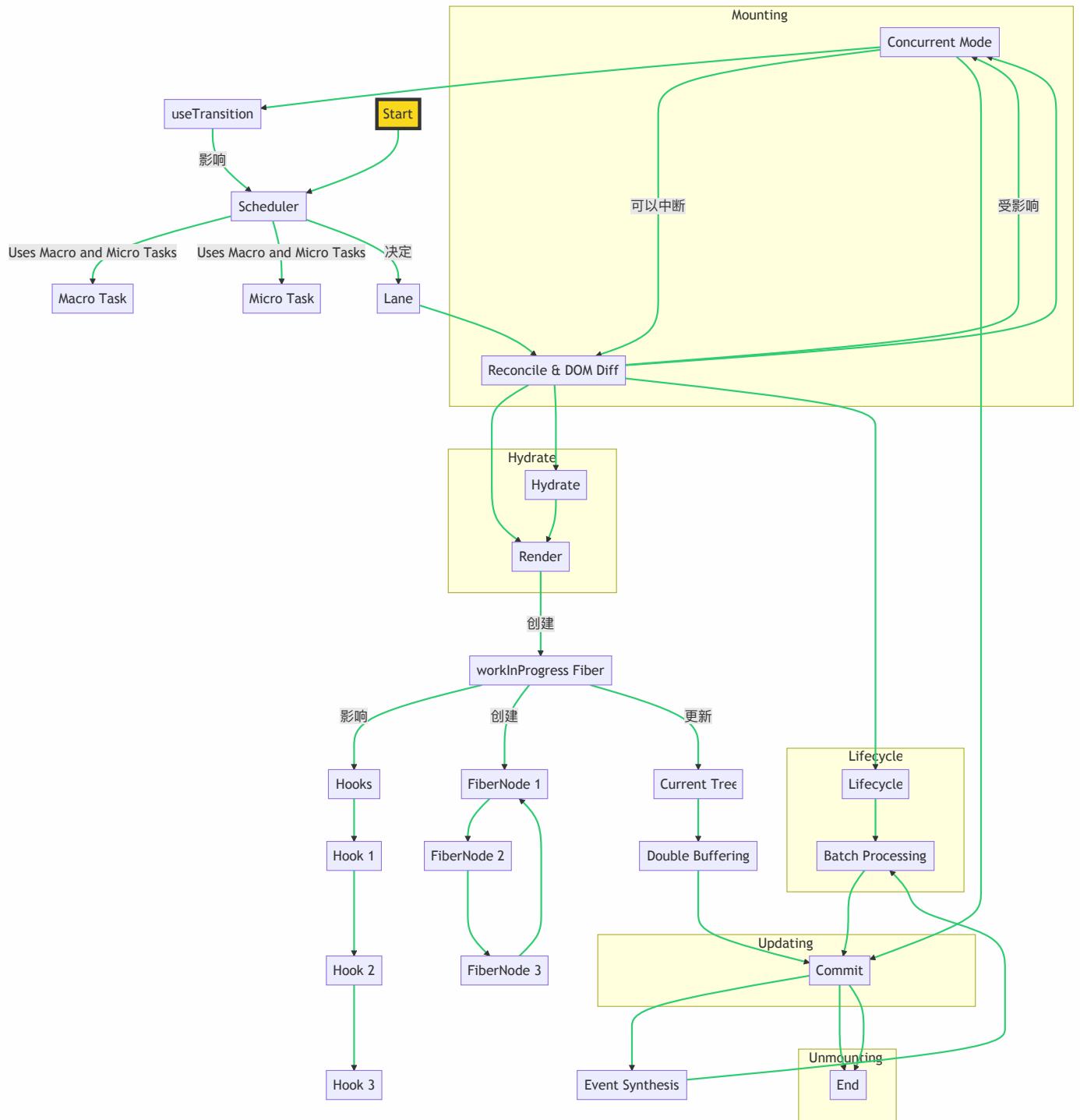
console.log("-----render-----");
render();
console.log("-----countChanged-----");
onClick();
onClick();
console.log("-----nameChanged-----");
onChange("1")
onChange("2")
console.log("-----willUnmount-----");
willUnmount();
```

志佳老师 2023年6月



React国内外大厂面试 30问

React架构图



本页考题为多年实践高频考题，但不承诺覆盖 100% 面试题。

1. API高频考题

高阶组件(HOC)是什么？你在业务中使用过解决了什么问题。

答案：

- HOC本质上是一个函数，它接收一个组件作为参数，然后返回一个新的组件。返回的新组件将拥有被包裹的组件的所有props，并且可以添加额外的props或状态。

HOC可以用于抽象出组件之间的共享代码，以增强组件的复用性和可维护性。也可以用于控制props，封装组件状态，或者通过引用（ref）来访问组件实例。

例如，以下是一个简单的HOC示例：

```
function withExtraProp(Component) {  
  return function(props) {  
    return <Component extraProp="someValue" {...props} />;  
  };  
}
```

在这个示例中，`withExtraProp` 是一个HOC，它接收一个组件 `Component`，然后返回一个新的组件，新的组件会向 `Component` 添加一个额外的prop。

在业务中可能会使用到HOC的例子：

1. **授权和权限管理：** 例如，你可能有一些组件只允许认证过的用户访问。你可以创建一个HOC，它接收一个组件并返回一个新的组件，新的组件在渲染之前会检查用户是否已经认证。
2. **数据获取：** 另一个常见的HOC用例是用于数据获取。例如，你可以创建一个HOC，它接收一个组件并返回一个新的组件，新的组件在挂载时获取数据，并将数据通过props传递给被包裹的组件。
3. **错误处理：** 你可以创建一个HOC，它接收一个组件并返回一个新的组件，新的组件包裹原组件的渲染，并在发生错误时显示错误信息或其他备用内容。

什么时候应该使用类组件而不是函数组件？React组件错误捕获怎么做？

答案：

- 在早期的React版本中，类组件和函数组件有明显的区别。类组件提供了生命周期方法（例如componentDidMount, componentDidUpdate等）和state，而函数组件则是无状态的，并且没有生命周期方法。因此，如果你的组件需要维护状态或者需要使用生命周期方法，那么你需要使用类组件。

然而，从React 16.8版本开始，引入了Hooks特性，允许在函数组件中使用状态(useState) 和生命周期方法(useEffect)。所以，目前你几乎在所有场景下都可以使用函数组件替代类组件。

不过，有一个情况下，你可能还需要使用类组件，那就是错误边界（Error Boundaries）。错误边界是React中的一种特性，它允许你在子组件树中捕获JavaScript错误，并在发生错误时显示备用内容，而不是让整个组件树崩溃。错误边界在React中只能通过类组件来实现。

以下是一个简单的错误边界类组件示例：

```
class ErrorBoundary extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { hasError: false };  
  }  
  
  static getDerivedStateFromError(error) {  
    // 当发生错误时，更新状态使下一次渲染显示备用UI  
    return { hasError: true };  
  }  
  
  componentDidCatch(error, info) {  
    // 你也可以将错误日志上报给一个错误报告服务  
    logErrorToMyService(error, info);  
  }  
  
  render() {  
    if (this.state.hasError) {  
      // 你可以自定义备用UI  
      return <h1>Something went wrong.</h1>;  
    }  
  }  
}
```

```
    return this.props.children;
}
}
```

然后你可以像这样使用 `ErrorBoundary` 组件来包裹其它组件：

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

这样，如果 `MyWidget` 组件或者它的子组件在渲染过程中发生错误，`ErrorBoundary` 就会捕获到这个错误，并显示备用UI，而不会导致整个应用崩溃。

如何在 React 中对 props 应用验证？

答案：

- React 提供了一个名为 `PropTypes` 的库，可以用于在开发环境下验证传递给组件的 `props` 是否符合期望的类型。

以下是使用 `PropTypes` 的基本示例：

```
import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  render() {
    // ...
  }
}

MyComponent.propTypes = {
  aStringProp: PropTypes.string,
  aNumberProp: PropTypes.number,
  aFunctionProp: PropTypes.func,
  aRequiredProp: PropTypes.number.isRequired,
  anArrayOfNumbers: PropTypes.arrayOf(PropTypes.number),
  anObjectOfShape: PropTypes.shape({
    color: PropTypes.string,
  })
}
```

```
    fontSize: PropTypes.number
  ) ,
};


```

在上述代码中，`MyComponent.propTypes` 对象定义了 `MyComponent` 组件所期望的 props 及其类型。例如，`aStringProp` 应当是一个字符串，`aNumberProp` 应当是一个数字，等等。如果 `MyComponent` 组件接收到类型不匹配的 props，那么在开发环境下，控制台会打印出警告。

在这个例子中，`isRequired` 属性被用来指明 `aRequiredProp` 是一个必须的 props。如果它没有被提供，那么在开发环境下会有警告信息打印出来。

注意，`PropTypes` 是在开发环境下运行的，对于生产环境不会产生任何影响。这意味着，`PropTypes` 并不能保证代码在生产环境下的正确性。为了在生产环境下也能保证类型安全，你可能需要使用 `TypeScript` 或者 `Flow` 这样的静态类型检查工具。

另外，从 `React 15.5.0` 版本开始，`PropTypes` 不再内置在 `React` 中，而是从 '`prop-types`' 库中导入。

在 `TypeScript` 中，你可以使用接口（`Interfaces`）或类型别名（`Type Aliases`）来进行类型验证。以下是一个简单的例子：

```
import React from 'react';

interface MyComponentProps {
  aStringProp?: string;
  aNumberProp?: number;
  aFunctionProp?: () => void;
  aRequiredProp: number;
  anArrayOfNumbers?: number[];
  anObjectOfShape?: {
    color?: string;
    fontSize?: number;
  };
}

const MyComponent: React.FC<MyComponentProps> = ({
  aStringProp,
  aNumberProp,
  aFunctionProp,
  aRequiredProp,
```

```
    anArrayOfNumbers,
    anObjectOfShape
) => {
//...
return <div>{aRequiredProp}</div>;
};
```

在上述代码中，我们首先定义了一个名为 `MyComponentProps` 的接口来描述 `MyComponent` 组件的props。在TypeScript中，可以使用 `?` 来表示一个属性是可选的。然后，我们用 `React.FC<MyComponentProps>` 来定义 `MyComponent` 组件的类型，这表明 `MyComponent` 是一个函数组件，它接收 `MyComponentProps` 类型的props。

使用TypeScript进行类型验证的好处是它不仅在开发环境下有效，而且在编译阶段就能捕获类型错误，从而提高代码质量和可维护性。

React 中如何创建Refs? 创建Refs的方式有什么区别?

答案：

- 在React中，Refs主要用于获取和操作DOM元素或者React组件的实例，是一种逃脱props传递的方法。React中创建Refs的主要方式有两种：`React.createRef()` 和回调Refs。
 1. `React.createRef()`：这是React 16.3版本后推出的新API，使用这种方式创建的Ref可以在整个组件的生命周期中保持不变。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }

  componentDidMount() {
    const node = this.myRef.current;
    // 你现在可以访问DOM节点或者组件实例
  }

  render() {
    return <div ref={this.myRef} />;
  }
}
```

```
    }  
}
```

在上述代码中，`this.myRef` 被赋值为 `React.createRef()` 的返回值。然后在 `render` 方法中，我们将 `this.myRef` 传递给了 `div` 的 `ref` 属性。这样，当 `div` 被挂载后，我们就可以在 `this.myRef.current` 中访问到 `div` 的 DOM 节点。

2. 回调Refs: 除了 `React.createRef()` 外，你还可以使用一个函数作为 Ref。这个函数将会在组件挂载和卸载时分别被调用，并将 DOM 节点或组件实例作为参数。

```
class MyComponent extends React.Component {  
  componentDidMount() {  
    // 你现在可以访问DOM节点或者组件实例  
  }  
  
  render() {  
    return <div ref={node => (this.myRef = node)} />;  
  }  
}
```

在上述代码中，我们将一个函数传递给 `div` 的 `ref` 属性。当 `div` 被挂载后，这个函数就会被调用，并将 `div` 的 DOM 节点作为参数传入。然后我们将 `div` 的 DOM 节点赋值给 `this.myRef`，这样就可以在其他地方访问到 `div` 的 DOM 节点。

这两种Refs创建方式的主要区别在于：

- `React.createRef()` 创建的 Ref 更简洁，API 更一致，而且 Ref 的值在组件的整个生命周期中保持不变。
- 回调Refs 更灵活，它允许你在组件挂载和卸载时执行一些额外的逻辑。但是，如果回调函数是在 `render` 方法中定义的，那么每次 `render` 时都会创建一个新的函数实例，可能会导致一些性能问题。

总的来说，除非你有特殊需求，否则建议使用 `React.createRef()` 来创建 Ref。

在 **函数组件** 中，你可以使用 `React.useRef()` 和 回调 Refs 来创建 Refs。

1. `React.useRef()`：这是 React Hooks API 的一部分，可以在函数组件中使用。使用 `useRef` 创建的 Ref 在整个组件的生命周期中保持不变。

```
import React, { useRef } from 'react';

function MyComponent() {
  const myRef = useRef(null);

  React.useEffect(() => {
    const node = myRef.current;
    // 你现在可以访问 DOM 节点或者组件实例
  }, []);

  return <div ref={myRef} />;
}
```

在上述代码中，我们首先使用 `useRef` 创建了一个 Ref，并将其赋值给 `myRef`。然后，在 `render` 方法中，我们将 `myRef` 传递给 `div` 的 `ref` 属性。这样，当 `div` 被挂载后，我们就可以在 `myRef.current` 中访问到 `div` 的 DOM 节点。

2. 回调 Refs: 你也可以在函数组件中使用回调 Refs。这种方式下，你会提供一个函数，该函数会在组件挂载和卸载时分别被调用，并将 DOM 节点或组件实例作为参数。

```
import React, { useEffect, useState } from 'react';

function MyComponent() {
  const [myRef, setMyRef] = useState(null);

  useEffect(() => {
    if (myRef) {
      // 你现在可以访问 DOM 节点或者组件实例
    }
  }, [myRef]);

  return <div ref={node => setMyRef(node)} />;
}
```

这两种创建 Ref 的方式主要区别在于：

- `React.useRef()` 创建的 Ref 更简洁，API 更一致，而且 Ref 的值在组件的整个生命周期中保持不变。
- 回调 Refs 更灵活，它允许你在组件挂载和卸载时执行一些额外的逻辑。但是，如果回调函数是在 `render` 方法中定义的，那么每次 `render` 时都会创建一个新的函数实例，可能会导致一些性能问题。

除非你有特殊需求，否则建议使用 `React.useRef()` 来创建 Ref。

createContext解决了什么问题？React父组件如何与子组件通信？子组件如何改变父组件的状态？

答案：

- `createContext` 解决了什么问题？

React的 `createContext` API主要解决了props“穿透”的问题，即你需要把一个prop层层传递给深层嵌套的子组件。这在应用中有一些全局可用的数据时（如主题，用户信息等）非常有用。通过创建一个context，你可以让组件直接访问这些数据，无需通过props层层传递。

```
import React, { createContext, useContext } from 'react';

// 创建一个Context对象
const ThemeContext = createContext('light');

function App() {
  // 使用Provider组件来提供context的值
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar() {
  return <Button />;
}
```

```
function Button() {
  // 使用useContext Hook来消费context的值
  const theme = useContext(ThemeContext);
  return <button>{theme}</button>;
}
```

在上述代码中，我们首先使用 `createContext` 创建了一个 `ThemeContext`。然后，我们在 `App` 组件中使用 `ThemeContext.Provider` 来提供 context 的值。最后，在 `Button` 组件中，我们使用 `useContext` Hook 来消费 context 的值。

2. React 父组件如何与子组件通信？

父组件通过 `props` 向子组件传递数据和函数。子组件可以通过 `props` 获取到这些数据和函数。

3. 子组件如何改变父组件的状态？

子组件无法直接改变父组件的状态。但是，父组件可以通过 `props` 向子组件传递一个函数，子组件调用这个函数就能间接地改变父组件的状态。

```
import React, { useState } from 'react';

function Parent() {
  const [count, setCount] = useState(0);

  const handleIncrement = () => {
    setCount(prevCount => prevCount + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <Child onIncrement={handleIncrement} />
    </div>
  );
}

function Child({ onIncrement }) {
  return <button onClick={onIncrement}>Increment</button>;
}
```

在上述代码中，`Parent` 组件向 `Child` 组件传递了一个名为 `onIncrement` 的函数。当 `Child` 组件的按钮被点击时，它就会调用 `onIncrement` 函数，从而间接地改变 `Parent` 组件的状态。

memo有什么用途，useMemo和memo区别是什么？useCallback和useMemo有什么区别？

答案：

- React 中的 `memo` 是一个高阶组件，它用于优化组件的渲染性能。`memo` 可以将一个纯函数组件（无状态组件）包装起来，以避免不必要的重新渲染。
- useMemo专用的hooks，缓存值。useCallback一般缓存我们的函数。
- `React.memo` 的用途：

`React.memo` 是一个高阶组件，它类似于 `React.PureComponent`，但只适用于函数组件，不适用于类组件。`React.memo` 对一个组件进行封装，使其仅在 `props` 改变时进行重新渲染，而不是每次父组件重新渲染时都进行渲染。这样可以避免不必要的渲染，提高性能。

```
const MyComponent = React.memo(function MyComponent(props) {  
  /* 使用 props 渲染 */  
});
```

2. `useMemo` 和 `memo` 的区别：

`useMemo` 是一个 Hook，它用于避免执行昂贵的计算操作。当依赖项改变时，`useMemo` 将重新计算缓存的值。

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b),  
  [a, b]);
```

`React.memo` 是一个高阶组件，它用于避免函数组件进行不必要的重新渲染。简单来说，`useMemo` 是用于优化计算操作，而 `React.memo` 是用于优化渲染。

3. `useCallback` 和 `useMemo` 的区别：

`useCallback` 和 `useMemo` 都是用于优化的 Hook，但它们用于优化的对象不同。

`useCallback` 用于返回一个 memoized 的回调函数：

```
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

`useMemo` 用于返回一个 memoized 的值：

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b),
[a, b]);
```

如果你有一个依赖于某些值的函数，并且你想防止这个函数在这些值未改变时被重新创建，那么你应该使用 `useCallback`。如果你有一个依赖于某些值的昂贵计算，并且你想防止这个计算在这些值未改变时被重新执行，那么你应该使用 `useMemo`。

React新老生命周期的区别是什么？合并老生命周期的理由是什么？

答案：

- React 16.3 版本以后，对组件的生命周期函数进行了一些修改和增强，主要出于优化异步渲染和性能的考虑。
 1. React 新老生命周期的区别：
 - 移除的生命周期方法：`componentWillMount`、`componentWillReceiveProps` 和 `componentWillUpdate`。这些生命周期方法在新的版本中被认为是不安全的，因为在异步渲染（React 16.3 引入的新特性）中，它们可能会被意外地多次调用。使用它们可能会导致一些难以调试的问题。
 - 新增的生命周期方法：`getDerivedStateFromProps` 和 `getSnapshotBeforeUpdate`。这两个方法是为了替代被移除的生命周期方法，同时提供更好的异步渲染支持。

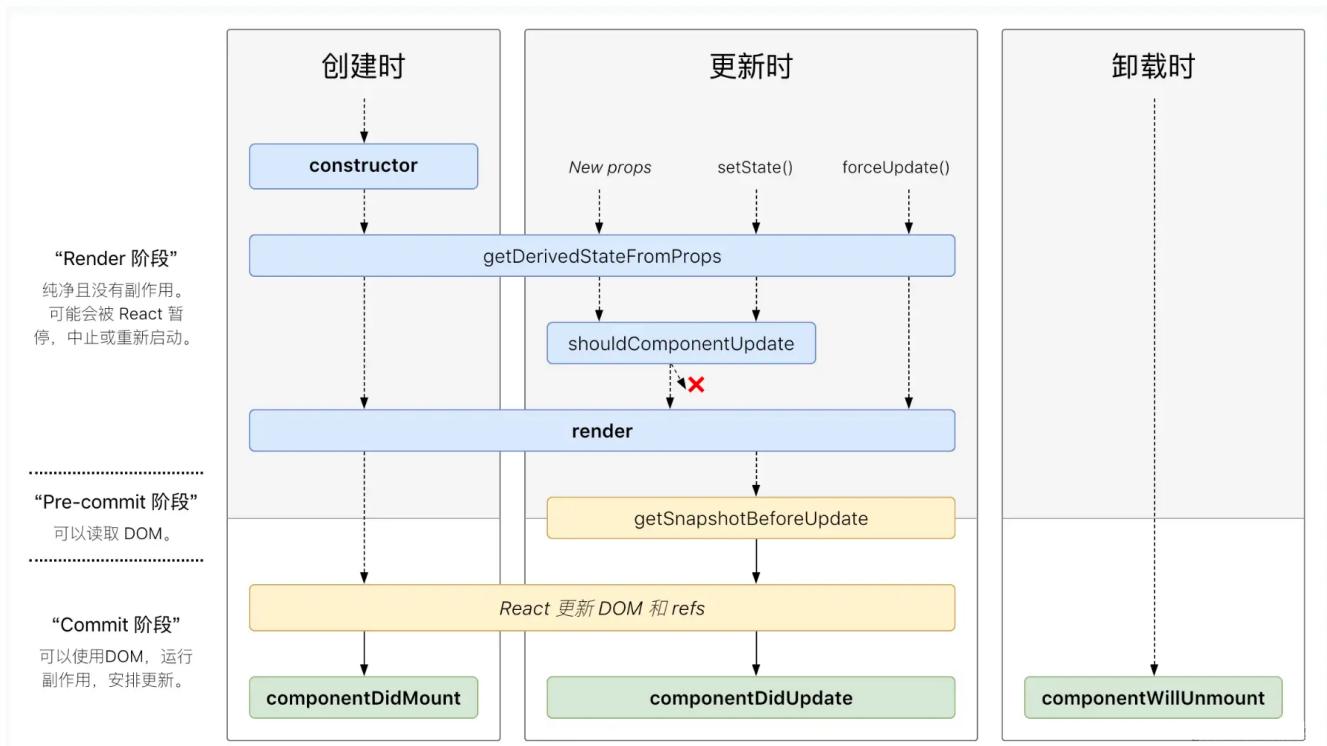
- **修改的生命周期方法:** `componentDidUpdate` 和 `componentDidCatch`。它们的功能没有改变，但是它们现在会在提交阶段被调用，这是异步渲染引入的新阶段。
- **被认为是安全的生命周期方法:** `componentDidMount`、`shouldComponentUpdate` 和 `componentWillUnmount`。这些生命周期方法并没有改变，而且在异步渲染中被认为是安全的。

2. 合并老生命周期的理由:

老生命周期函数在异步渲染模式下存在一些潜在的问题，比如可能会多次调用 `componentWillUpdate` 和 `componentWillReceiveProps`，这可能会导致状态的不一致。为了解决这些问题，React 团队提出了新的生命周期函数，以更好地支持异步渲染和性能优化。

例如，新的 `getDerivedStateFromProps` 生命周期函数在每次渲染前都会被调用，包括初始化渲染和后续更新，这使得组件可以在渲染前更新状态，以此来替换 `componentWillReceiveProps` 的部分功能。`getSnapshotBeforeUpdate` 则在 DOM 更新前被调用，能够在可能的情况下捕获一些 DOM 信息，以此来替代 `componentWillUpdate` 的部分功能。

这些改变使得 React 的生命周期函数更易于理解，同时提供更好的性能，并更适合未来 React 的发展，包括异步渲染等新特性。



React中的状态管理库你如何选择？什么是状态撕裂？useState同步还是异步？

- Redux作者的加入 useState底层也被彻底重写都是基于useReducer
- 在 react 18 版本以前在同步环境中异步，在异步环境中同步。在 react 18 版本以后，**setState ()** 不论在同步环境还是异步环境都是异步的。
- setState本身并不具备绝对的同步/异步概念。比如：在promise的then()方法中、setTimeOut()、setInterVal()，ajax的回调等异步环境中，setState就是同步的。同步环境下就是异步的。

react会有一个上下文环境，在同步环境中，setState处于react的上下文中，react会监控动作合并，所以这个时候setState()是异步的。

而在异步环境中，比如promise的then()方法中、setTimeOut()、setInterVal()中，react实际上已经脱离了react的上下文环境。所以setState()是同步执行的

- 在 React 中，状态管理库的选择取决于你的项目需求和个人/团队的熟悉程度。下面是一些常见的状态管理库，以及你可能会考虑选择它们的理由：
 1. **Redux**: 这是最常见的状态管理库，适用于大型应用和需要跨组件共享状态的情况。Redux 提供了一个集中式的状态存储，可以让你在任何地方访问和修改状态。如果你的应用状态逻辑比较复杂，或者你需要在应用的不同部分共享大量的状态，那么 Redux 可能是一个好选择。
 2. **MobX**: 这是另一个常见的状态管理库，它采用更灵活的、响应式的状态管理模型。与 Redux 的 "单一数据源，不可变状态" 的原则不同，MobX 允许有多个状态源，并且状态可以是可变的。如果你对 Redux 的严格性有些不满，或者你需要一个更加适应快速原型开发的状态管理库，那么 MobX 可能是一个好选择。
 3. **Context API 和 useState/useReducer Hooks**: 对于一些小型应用或组件，你可能不需要一个完整的状态管理库。React 的 Context API 和 Hooks（例如 useState 和 useReducer）可能已经足够满足你的需求。使用这些 React 内置的功能，你可以在组件间共享状态，而无需添加额外的依赖。
 4. 当然，还有Jotai。Jotai 是一个较新的状态管理库，旨在提供一个原子状态管理的解决方案。其核心理念在于将状态分解为最小的、可组合的单元，即“原子”。

下面是一些关于 Jotai 的详细信息：

Jotai: Jotai 的主要目标是提供一个简单且轻量的全局状态管理工具，它对 Concurrent Mode 完全友好，并尝试解决 React 状态共享的问题。Jotai 的 API 极其简单，它将状态分解为 "atom"（即最小的状态单元）。这个库的优势在于其原子状态可以被细粒度地订阅，因此，只有当状态改变时，依赖这个状态的组件才会重新渲染，这避免了无效的组件重新渲染。如果你的应用有许多独立但又需要共享的状态，Jotai 可能是一个很好的选择。

比如，你可以这样创建一个 atom：

```
import { atom } from 'jotai'

const countAtom = atom(0)
```

然后你可以使用 `useAtom` hook 在组件中读取或写入这个 atom：

```
import { useAtom } from 'jotai'
import { countAtom } from './atoms'

function Counter() {
  const [count, setCount] = useAtom(countAtom)
  return (
    <div>
      <div>Count: {count}</div>
      <button onClick={() => setCount(count +
1)}>increment</button>
    </div>
  )
}
```

每一个使用 `useAtom(countAtom)` 的组件只会在 `countAtom` 更新时重新渲染。

综上，选择状态管理库应当根据你的应用需求、开发团队熟悉程度和你对库理念的认同程度来进行。如果你的应用的状态管理需求较为复杂，或者需要全局状态管理，那么 Redux 或 MobX 可能是一个不错的选择。而如果你需要一个更轻量级、原子级别的状态管理库，Jotai 也是一个很好的选择。如果你的应用规模较小，或者状态管理需求较为简单，那么 React 自身的 Context API 和 Hook API 就足以满足你的需求。

- “状态撕裂”是指在并发渲染（Concurrent Mode）中，由于渲染的优先级不同，可能导致应用中的不同部分看到的同一份共享状态不一致的问题。这是因为在 Concurrent Mode 下，React 可以选择暂停、中断或延迟某些更新，以优先处理更重要的更新。如果你的状态更新和组件的渲染不是同步的，那么就可能出现状态撕裂的问题。React 团队正在开发一个新的特性（React Server Components）和新的 Hook（如 useTransition 和 useDeferredValue）来帮助开发者解决这个问题。

在 React 中什么是 Portal ?

答案：

- 在 React 中，Portal 提供了一种将子节点渲染到存在于父组件 DOM 层次结构之外的 DOM 节点的方式。

在很多场景下，当父组件有一些样式（比如：overflow: hidden 或 z-index）时，这些样式可能会影响或剪裁其子组件的布局表现。常见的应用场景如模态框（Modal）、提示框（Tooltips）等，我们期望这些组件能够“跳出”其父组件的容器，渲染到 DOM 树的顶层，从而避免这些问题。这就是 Portal 的作用。

你可以使用 `ReactDOM.createPortal` 方法创建一个 portal，这个方法有两个参数：第一个参数（child）是任何可渲染的 React 子元素，比如一个元素，字符串或者 fragment；第二个参数（container）是一个 DOM 元素。

下面是一个简单的模态框例子，它使用了 portal：

```
class Modal extends React.Component {  
  el = document.createElement('div');  
  
  componentDidMount() {  
    // 在 Modal 的所有子元素被挂载后,  
    // 这个 div 元素被添加到 body 中  
    document.body.appendChild(this.el);  
  }  
  
  componentWillUnmount() {  
    // 在 Modal 的所有子元素被卸载后,  
    // 这个 div 元素被从 body 中移除  
    document.body.removeChild(this.el);  
  }  
}
```

```
render() {
  return ReactDOM.createPortal(
    // 任何可以被渲染的元素，比如一个组件
    this.props.children,
    // 指向一个 DOM 元素
    this.el,
  );
}
```

在这个例子中，`Modal` 组件的子元素被渲染到了传入 `ReactDOM.createPortal` 的 `div` 元素中，而这个 `div` 元素是直接添加到 `body` 中的。这样，即使 `Modal` 组件在 DOM 树中的位置深入其它组件，它的子元素也能被渲染到 DOM 树的顶层。

自己实现一个Hooks的关键点在哪里？

答案：

- 创建自定义 Hook 本质上就是创建一个 JavaScript 函数，但这个函数需要遵循两个主要的规则：
 1. **函数名称必须以 `use` 开头**：这是一个命名约定，React 依赖于此约定来自动检查你的 Hook 是否遵循 Hook 的规则。这也有助于在阅读代码时明确一个函数是否是 Hook。
 2. **只能在函数组件或者其他 Hook 中调用 Hook**：这是因为 React 需要保持 Hook 调用的顺序一致以正确地维护内部状态。

自定义 Hook 可以调用其他的 Hook，这意味着你可以在自定义 Hook 中使用状态（`useState`），副作用（`useEffect`）等 React 提供的 Hook。

例如，你可以创建一个自定义 Hook 来提供和管理一个计数器的状态：

```
import { useState } from 'react';

function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue);

  const increment = () => setCount(prevCount => prevCount +
1);
  const decrement = () => setCount(prevCount => prevCount -
1);

  return { count, increment, decrement };
}
```

然后，你就可以在组件中像使用其他 Hook 一样使用这个自定义 Hook：

```
function Counter() {
  const { count, increment, decrement } = useCounter();

  return (
    <div>
      <button onClick={decrement}>-</button>
      <span>{count}</span>
      <button onClick={increment}>+</button>
    </div>
  )
}
```

在这个例子中，`useCounter` 就是一个自定义 Hook，它使用了 React 的 `useState` Hook 来提供计数器的状态，并提供了增加和减少计数器的函数。

如果你在 TypeScript 中编写自定义 Hook，你会需要指定一些类型信息以确保代码的正确性和利于代码的理解。以下是使用 TypeScript 创建自定义 Hook 的一些注意事项：

1. **为 Hook 的参数指定类型**：你应该为 Hook 的输入参数指定类型。例如，在上述 `useCounter` Hook 的例子中，我们可以为 `initialValue` 参数指定类型为 `number`。

```
function useCounter(initialValue: number = 0) {  
  // ...  
}
```

2. **为 Hook 的返回值指定类型**: 对于 Hook 返回的对象或其他值，你也应该提供类型注解。在上述 `useCounter` 的例子中，我们可以创建一个返回值类型：

```
type UseCounterReturn = {  
  count: number;  
  increment: () => void;  
  decrement: () => void;  
};  
  
function useCounter(initialValue: number = 0):  
  UseCounterReturn {  
  // ...  
}
```

3. **为内部状态和函数指定类型**: 使用 `useState` 或 `useReducer` 时，你可以为内部状态指定类型。同样，如果你在 Hook 中创建函数，也应该为这些函数的参数和返回值提供类型注解。

总的来说，与在其他 TypeScript 中编写的代码一样，使用 TypeScript 编写自定义 Hook 的主要好处是能够在编译时就捕获到可能的类型错误，以及提供更好的编辑器自动补全和提示。在使用 TypeScript 编写自定义 Hook 时，应该尽可能地为参数、状态和返回值等提供准确的类型注解。

- 最后大家一定要注意的就是 `useCallback`、和 `return` 元组类型的结构。

你去实现React的具体业务的时候TS类型不知道怎么设置你会怎么办？

答案：

- 翻阅 `React.d.ts` 的文档（这个要着重说一下）
- 搜索谷歌

React和其他框架对比优缺点是什么？你们团队选择React的理由是什么？

答案：

- React 是一个用于构建用户界面的 JavaScript 库，由 Facebook 大团队维护，已经在很多知名项目和公司中得到广泛使用。以下是一些 React 的优点和缺点：

优点：

1. **丝滑函数组件和hooks**：React 通过组件化的方式帮助开发者构建复杂的用户界面。每个组件都有自己的状态和逻辑，可以单独测试和重用。
2. **性能**：并发更新模式、FID的提前
3. **强大的社区支持**：由于其广泛的使用，React 拥有大量的开源库，丰富的学习资源和活跃的社区。
4. **方便的状态管理**：React 提供了 Context API 和 Hooks，简化了状态管理。另外，也可以使用 Redux、MobX、Jotai等状态管理库。

缺点：

1. **React 只是视图层**：相对于像 Angular 这样的完整框架，React 只关注视图层。这意味着开发者需要选择其他库来处理路由和状态管理等功能，这可能会增加项目的复杂性。
2. **频繁更新**：React 社区非常活跃，经常会有新的特性和改进。然而，这也意味着开发者需要不断学习新的概念和最佳实践。
3. **学习曲线相对陡峭**：想维护一个高性能的React应用需要你掌握非常详细的React使用技巧，比如配合Why Did You Render

至于为什么选择 React，这通常取决于团队的具体需求和偏好。以下是一些常见的理由：

1. **技术栈的一致性**：如果团队已经在其他项目中使用了 React，那么选择 React 可以保持技术栈的一致性，提高开发效率。
2. **团队的技能和经验**：如果团队成员已经熟悉 React，那么选择 React 可以减少学习新框架的时间。
3. **项目需求**：如果项目需要构建复杂的用户界面，并且希望有更高的灵活性，那么 React 可能是一个好选择。

以上是一些一般性的考量，实际的选择应当基于团队和项目的具体情况。

React16/17/18都有哪些新变化？useTransition是啥提解决了什么？

答案：

- **React 16** 的主要更新是引入了新的核心算法——Fiber 架构，它提供了如下一些新特性和更新：

1. 错误边界 (Error Boundaries)：错误边界是 React 16 中引入的一种新的错误处理机制，让你可以捕获和打印发生在子组件树任何地方的 JavaScript 错误，防止整个应用崩溃。
2. Fragments 和 Strings: React 16 允许组件可以返回多个元素 (Fragments) 或者字符串。
3. Portals: Portals 提供了一种将子节点渲染到存在于父组件 DOM 层次结构之外的 DOM 节点的方式。
4. 更好的服务器端渲染：包括新的 SSR API，支持流式渲染和组件缓存。

React 17 没有引入新的特性，但它做了许多更改来使 React 更容易升级，并且支持在同一个应用中运行多个版本的 React。

1. Event Delegation 的改变：在 React 17 中，React 不再将事件处理程序附加到 document，而是附加到根 DOM 容器。
2. Gradual Upgrades: React 17 的一个主要目标是使 React 的升级更加平滑，允许在同一应用中使用不同版本的 React。

React 18 做了很大的改变将如并行和模式和用户自定义控制优先级等进行了正式发版：

1. 并发模式 (Concurrent Mode)：并发模式是一个让 React 能在渲染过程中让出控制权给浏览器，以便浏览器能及时处理用户交互的新模式。
2. React Server Components: 服务器组件是一种只在服务器上运行，无需发送到客户端的新组件类型，它旨在提升渲染性能和减少客户端代码。

`useTransition` 是在并发模式下使用的一个新 Hook，它可以让你避免在 UI 中创建阻塞的视觉更新。例如，在数据加载时，你可能希望先显示一个加载的指示器，等数据加载完再更新 UI。在这种情况下，`useTransition` 可以让你的应用保持响应，并且让视觉更新的过程看起来更加平滑。

这个 Hook 返回一个数组，第一项是一个函数，可以用来触发一个过渡状态的更新；第二项是一个布尔值，表示是否处于过渡状态。当处于过渡状态时，你可以选择显示一个加载指示器或者其他备用 UI。

```
import { useState, useTransition, Suspense } from 'react';
```

```
// 模拟异步数据加载
const fetchData = () => {
  return new Promise(resolve =>
    setTimeout(() => resolve('Loaded data'), 2000)
  );
}

const AsyncComponent = () => {
  const [data, setData] = useState(null);
  const [startTransition, isPending] = useTransition();

  const loadData = () => {
    startTransition(() => {
      fetchData().then(result => setData(result));
    });
  };

  return (
    <div>
      <button onClick={loadData}>Load Data</button>
      {isPending ? 'Loading...' : data}
    </div>
  );
};

// 在 App 组件中使用 Suspense
const App = () => {
  return (
    <Suspense fallback="Loading...">
      <AsyncComponent />
    </Suspense>
  );
};
```

2. 源码高频考题

React整体渲染流程请描述一下？嗯，你描述的蛮好。那你能说下双缓存是在哪个阶段设置的么？优缺点是什么？

- 初始阶段：

创建根Fiber节点，代表整个React应用的根组件。
调用根组件的render方法，创建初始的虚拟DOM树。
- 调度阶段（Scheduler）：

使用调度器（Scheduler）调度更新，决定何时执行更新任务。
检查是否有高优先级任务需要执行，如用户交互事件或优先级较高的异步操作。
根据优先级确定任务执行顺序，并根据任务的优先级将任务添加到不同的任务队列（lane）中。
- 协调阶段（Reconciliation）：

从任务队列中取出下一个任务。
对任务中涉及的组件进行协调，比较前后两个虚拟DOM树的差异，找出需要更新的部分。
使用DOM diff算法进行差异计算，生成需要更新的操作指令。
- 生命周期阶段（Lifecycle）：

在协调阶段和提交阶段，React会根据组件的生命周期方法调用相应的钩子函数。
生命周期方法包括componentDidMount、componentDidUpdate等，用于处理组件的生命周期事件。
- 5. 渲染阶段（Render）：

在Render阶段，React会根据组件的状态变化、props的更新或者父组件的重新渲染等触发条件，重新执行组件的函数体（函数组件）或者render方法（类组件）。
当React执行函数组件或render方法时，它会检测组件中是否包含了Hooks，如果包含了Hooks，那么React会根据Hooks的顺序依次调用它们。
Hooks执行：
在Render阶段，React会根据组件中Hooks的顺序，依次执行每个Hooks函数。
Hooks函数可能包括useState、useEffect、useContext等等，这些Hooks函数会在组件每次更新时被调用，让你能够在函数组件中使用状态、副作用和上下文等特性。
(批处理 setState 合并一次 setTimeout 16阶段不支持了 React18自动化批处理)

- Commit (提交) 阶段：
 - 在Commit阶段，React将Render阶段生成的更新应用到真实的DOM中，完成页面的渲染。
在Commit阶段，React可能会执行一些其他操作，比如调用生命周期方法（如componentDidMount、componentDidUpdate等）或执行其他副作用。
- 重复步骤2-7：
 - 根据应用程序的交互和状态变化，React会重复执行调度、协调、渲染、提交的步骤，实现更新的循环流程。

Fiber架构原理你能细致描述下么？

- 答案：也可顶部或参考React渲染整体流程解析，以下是一些文字描述
- Fiber 是 React 16 中引入的新的调和（reconciliation）引擎。在这个新的架构下，React 能够做到调度和优先级，使得 React 可以在执行过程中暂停、中断和恢复工作，从而实现了时间切片（time slicing）和并发模式（Concurrent Mode）等特性。

Fiber 的核心原理可以用以下几个关键概念来理解：

1. **Fiber Node**：在 Fiber 架构中，每一个 React 组件都有一个对应的 Fiber 节点，它是一个保存了组件状态、组件类型和其他信息的对象。每一个 Fiber 节点都链接到一个父节点、第一个子节点、兄弟节点，形成了一个 Fiber 树。
2. **双缓冲技术**：Fiber 架构中采用了双缓冲技术，即有两棵 Fiber 树：当前在屏幕上显示的 Fiber 树（current Fiber tree）和下一帧要显示的 Fiber 树（work-in-progress Fiber tree）。这种方式避免了在渲染过程中直接修改 DOM，提升了性能，并能在出现错误时回退到稳定状态。
3. **工作循环**：React 的渲染过程可以看作是一个工作循环。在这个循环中，React 会遍历 Fiber 树，为每个节点调用对应的生命周期方法，并生成对应的 DOM 更新。如果在遍历过程中，有更高优先级的更新出现，React 可以将当前的工作暂停，去处理更高优先级的更新。
4. **时间切片和暂停（React18去掉了时间切片，改成了微任务+宏任务）**：
在 Fiber 架构中，React 将渲染工作分解为多个小任务，每个任务的执行时间不超过一个阈值。这使得 React 可以在长时间的渲染任务中，让出控制权给浏览器，处理其他更重要的工作，如用户的输入和动画。这就是所谓的“时间切片”。

5. **优先级和并发**: 在工作循环中，不同类型的更新可以有不同的优先级。例如，用户的交互会有更高的优先级，因为它们需要立即响应。这使得 React 能在需要的时候，打断当前的工作，去处理更紧急的任务。这就是所谓的“并发”。
6. **错误边界**: 在 Fiber 架构中，如果一个组件在渲染过程中发生错误，React 会寻找最近的错误边界组件，并将错误传递给它。错误边界组件可以捕获这个错误，并显示一个备用的 UI，防止整个应用崩溃。

Fiber 架构是 React 中的一项重要技术创新，它带来了许多新的特性和性能优化，让 React 在处理复杂的用户界面时更加高效。

React Scheduler核心原理 React 16/17/18变化都有哪些？Batching在这个阶段里么，解决了什么原理是什么？

答案：

- React Scheduler 是一个 React 内部的任务调度库。它主要用于在长期执行的渲染任务中切分任务，让浏览器在执行长期任务的空闲时间内有机会处理其他的任务，比如用户输入和动画，以提高应用的响应性。这也是 React 中时间切片（Time Slicing）的核心实现。

在 React 16 中，Scheduler 作为一个实验性的库被引入，用于实现新的 Fiber 架构和时间切片。

在 React 17 中，Scheduler 并没有明显的变化，React 17 主要在事件系统上做了改进，使得 React 能和其他 JavaScript 库更好的共存。

在 React 18 中 Scheduler 将被更完整的利用，以实现并发模式（Concurrent Mode）和新的 Suspense 特性。

- Batching 是 React 的一个重要特性，它允许 React 将多个状态更新合并为一次渲染，以减少不必要的渲染次数和 DOM 更新，从而提高性能。在 React 的历史版本中，Batching 主要在 React 的事件处理函数和生命周期方法中生效。在其他的异步代码中，Batching 不生效，每个状态更新都会导致一次渲染。

然而，在 React 18 中，引入了一个新的特性叫做 automatic batching。这个特性使得在任何地方，只要是连续的多个状态更新，都会被自动合并为一次渲染。这使得性能优化变得更容易，开发者无需考虑是否在一个 batch 中。

例如，在以下的代码中，两个状态更新会被合并为一次渲染：

```
setTimeout(() => {
  setState(count + 1);
  setState(count + 1);
}, 1000);
```

这是一个重要的改进，它使得 React 的性能优化更加自动化，开发者无需过多的考虑性能问题。

- 在 React 中，"batching" 是指把多个状态更新操作合并成一个操作，从而减少不必要的渲染和 DOM 更新，提高应用的性能。这个机制一直存在于 React 中，但在不同的情况下工作方式是不同的。

传统的 batching：在 React 的历史版本中，只有在 React 的事件处理函数和生命周期方法中，多个状态更新会被自动合并为一次渲染。这是因为在这些方法中，React 有足够的上下文信息来决定何时开始和结束一次 batch。

例如，以下的代码会触发一次渲染：

```
handleClick() {
  this.setState({count: this.state.count + 1});
  this.setState({count: this.state.count + 1});
}
```

但是在其他的异步代码中，React 没有足够的信息来决定何时开始和结束一次 batch，所以每个状态更新都会导致一次渲染。例如，以下的代码会触发两次渲染：

```
setTimeout(() => {
  this.setState({count: this.state.count + 1});
  this.setState({count: this.state.count + 1});
}, 1000);
```

Automatic batching：在 React 18 中，引入了一个新的特性叫做 automatic batching。这个特性使得在任何地方，只要是连续的多个状态更新，都会被自动合并为一次渲染，无论它们发生在哪。这使得性能优化变得更简单，开发者无需考虑是否在一个 batch 中。

例如，以下的代码在 React 18 中也会触发一次渲染：

```
setTimeout(() => {
  setCount(count + 1);
  setCount(count + 1);
}, 1000);
```

这是一个重要的改进，它使得 React 的性能优化更加自动化，开发者无需过多的考虑性能问题。

Hooks为什么不能写在条件判断、函数体里。我现在有业务场景就需要在if里写怎么办呢？

答案

- React Hooks 使用一个单向链表来保存组件的状态和副作用。在每次组件渲染时，React 会遍历这个链表，按照定义的顺序依次执行每个 Hook 对应的状态更新和副作用函数。通过链表的形式，React 可以保持 Hook 的调用顺序一致，并正确地跟踪每个 Hook 的状态和更新。
- 请使用组件外状态 zustand等

setState直接在函数组件调用会造成无限渲染，原因是什么。怎么监控React无意义渲染，监控的原理是什么？

答案：

- 当你在函数组件的主体部分直接调用 `useState`，每次组件渲染时都会调用 `useState`，而每次调用 `useState` 又会触发组件的重新渲染，从而形成了一个无限循环。以下是一个这种情况的例子：

```
function MyComponent() {
  const [count, setCount] = React.useState(0);

  // 直接调用 setCount 会导致无限循环
  setCount(count + 1);

  return <div>{count}</div>;
}
```

为了避免这种情况，你应该在一个事件处理函数或者效果（Effect）中调用 `useState`。以下是一个正确的例子：

```
function MyComponent() {
  const [count, setCount] = React.useState(0);

  React.useEffect(() => {
    // 在 useEffect 中调用 setCount，只在组件挂载时执行一次
    setCount(count + 1);
  }, []); // 空数组 [] 表示这个 effect 无依赖，只在组件挂载和卸载时执行

  return <div>{count}</div>;
}
```

- `@welldone-software/why-did-you-render` 是一个用于调试 React 中不必要的重渲染的库。它可以帮助你追踪和识别因 prop 或 state 的不必要的变化而触发的重渲染，从而提高应用的性能。

当你在应用中使用该库时，它会通过包装 React 的组件，重写它们的 `shouldComponentUpdate`（对于类组件）或 `React.memo`（对于函数组件）方法。在这些方法中，它会比较前后两次渲染时 props 和 state 的值。

如果 props 或 state 的值在两次渲染之间没有发生真正的变化，但组件仍然被重新渲染了，那么 `@welldone-software/why-did-you-render` 就会在控制台中打印一条警告消息。这条消息将包含关于组件名称、prop 或 state 的前后值，以及可能的解决建议等信息。

需要注意的是，这个库并不能自动解决重渲染问题，它只能提供信息帮助你找出可能的问题。实际上，解决重渲染问题的关键通常在于正确地使用 React 的 `useState`, `useEffect`, `useMemo`, `useCallback` 等 Hook，以及 `shouldComponentUpdate` 或 `React.memo` 等方法来优化组件的渲染行为。

Dom Diff细节请详细描述一下？Vue使用了双指针，React为什么没采用呢？

- React的更新过程包括 `新旧虚拟DOM树的对比过程` 和 `更新DOM过程`。

16版本之前，是 **对比、更新** 同时进行，对比的过程采用 **递归** 的方式，技术实现方式是不断的将各个节点、各个节点的子节点压入 **栈** 中，采用深度遍历的方式不断的访问子节点，回溯直到diff完整棵树。整个过程由于是 **递归** 实现的，中间不能中断、中断后必须要重新开始，如果树的层级较深，会导致整个更新过程（js执行）时间过长，**阻碍页面渲染和造成用户交互卡顿** 等问题，体验较差。

- 由于 **递归算法、栈本身的局限性**，16之后将 **递归改成迭代**，而且只Diff同层节点(div->span就不要了)，并将 **栈结构改进成fiber链表结构**，实现了更新过程可以随时中断的功能。

React如何实现自身的事件系统？什么叫合成事件？

- React 合成事件（SyntheticEvent）是 React 框架自己实现的一套事件系统。这套系统模拟了原生的 DOM 事件，但同时提供了一些额外的优点：
 1. **跨浏览器兼容性**：不同浏览器的原生事件行为可能会存在差异。React 的合成事件为所有浏览器提供了一致的 API 和行为，从而消除了这种差异。
 2. **性能优化**：React 使用了事件委托（event delegation）机制。这意味着对于同一类型的事件，React 并不会直接将事件处理器绑定到 DOM 节点上，而是将一个统一的事件监听器绑定到文档的根节点上。当事件发生时，React 会根据其内部映射确定真正的事件处理器。这样做可以有效减少事件监听器的数量，节省内存，提高性能。
 3. **集成到 React 的状态系统**：React 合成事件系统与其组件生命周期和状态系统紧密集成，可以在事件处理函数中调用 setState，React 会正确地批处理更新和重新渲染。
 4. **提供更多的信息**：React 的合成事件提供了比原生事件更多的信息，例如 event.target。

合成事件的名称（例如 onClick, onChange 等）和它们在组件中的使用方式，都与你在 JavaScript 中使用 DOM 事件的方式非常相似。实际上，大多数情况下，你可以把它们当作标准的 DOM 事件来使用。

- 在React组件中，对大多数事件来说，React 实际上并不会将它们附加到 DOM 节点上。相反，React 会直接在 document 节点上为每种事件类型附加一个处理器。除了在大型应用程序上具有性能优势外，它还使添加类似于replaying events 这样的新特性变得更加容易。

但是如果页面上有多个 React 版本，他们都将再顶层注册事件处理器。这会破坏`e.stopPropagation()`：如果嵌套树结构中阻止了事件冒泡，但外部树依然能接收到它。这会使不同版本 React 嵌套变得困难重重。

在React 17中，React 将不再向 document 附加事件处理器。而是将事件处理器附加到渲染 React 树的根 DOM 容器中：

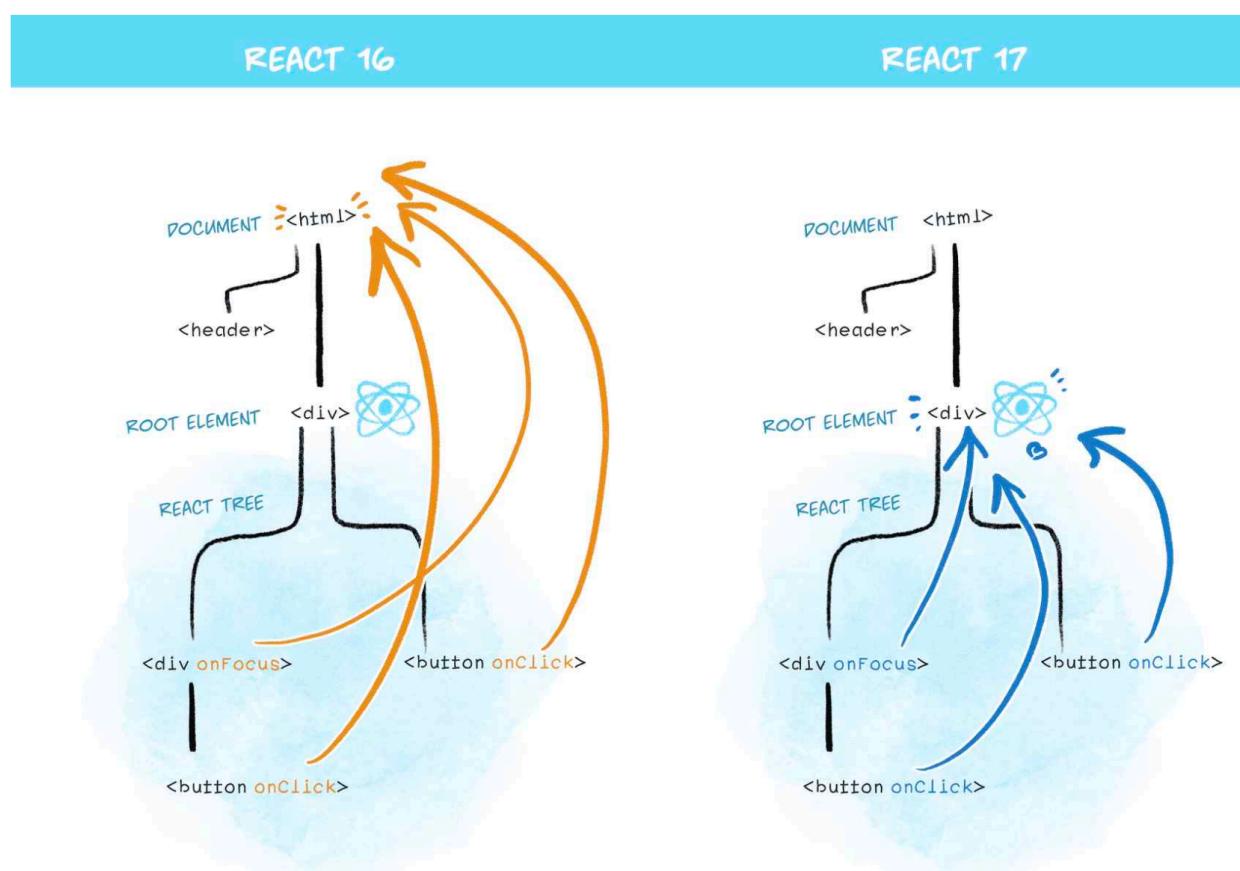
```
ReactDOM.render(, $("#app");
```

```
ReactDOM.render(
```

```
, $("#header");
```

```
ReactDOM.render(
```

```
, $("#footer"));
```



React Concurrent Mode是什么？React18是怎么实现的？他和useTransition有联系么？

答案：

- React 的 Concurrent Mode 是一种新的渲染模式，它使 React 能够在多个状态更新中进行“时间切片”，从而使得长时间运行的渲染任务不会阻塞浏览器的主线程。这种模式可以提高应用的响应性，特别是在复杂的用户界面和/或设备性能较低的情况下。

在传统的同步渲染模式中，React 会在一个状态更新发生时阻塞主线程，直到所有的组件都渲染完成。在一些情况下，这可能会导致应用变得不响应，因为主线程在渲染过程中无法处理其他任务，比如用户输入和动画。

而在 Concurrent Mode 中，React 会把渲染任务分解成多个小任务，每个任务的执行时间都很短。在这些任务之间，React 会给出一些空闲的时间，让浏览器有机会处理其他的任务。这就是所谓的“时间切片”。

`useTransition` 是一个在 React 18 中引入的新的 Hook，它与 Concurrent Mode 紧密相关。`useTransition` 使你可以告诉 React 你的状态更新可能需要一些时间来准备数据，例如发起一个网络请求。在这个状态更新的数据准备好之前，React 会继续显示旧的 UI，而不是立即渲染一个加载的状态。这可以避免界面的抖动，提高用户体验。

在 Concurrent Mode 中，`useTransition` 可以让你的应用在等待新的数据时保持响应，同时在数据准备好之后再平滑的过渡到新的状态。

需要注意的是，Concurrent Mode 和 `useTransition` 都是 React 18 中的新特性。

将Vue换成 React能提高FPS么？请给出理由

答案：

- 将 Vue 替换为 React，或反之，不一定会提高应用的帧率（FPS）。实际的性能表现取决于许多因素，包括但不限于你如何使用这些框架，你的应用的具体需求，以及用户的设备性能。

React 和 Vue 在设计上有一些关键的不同，这些不同可能会影响它们在特定场景下的性能：

- **虚拟 DOM 实现**：React 和 Vue 都使用虚拟 DOM 来提高渲染性能，但它们的实现方式略有不同。Vue 在一些情况下可以跟踪依赖关系，只更新改变的部分，而不是重新渲染整个组件树。React 从另一方面提供了一些优化技巧，如 `shouldComponentUpdate` 和 `React.memo`，开发者可以用它们来避免不必要的渲染。

- **异步渲染**: React 的 Concurrent Mode 和新的 `useTransition` Hook 支持异步渲染，这可以让 React 在处理大量更新时保持界面的响应。这可能有助于提高帧率，特别是在处理复杂交互和动画时。虽然 Vue 在此时没有类似的特性，但它也在寻求实现类似的优化。
- **框架的大小**: React 和 Vue 的大小相近，但 Vue 通常稍微小一些。较小的框架可以更快地加载和解析，这可能对首次渲染时间有所帮助，但对帧率的影响可能较小。

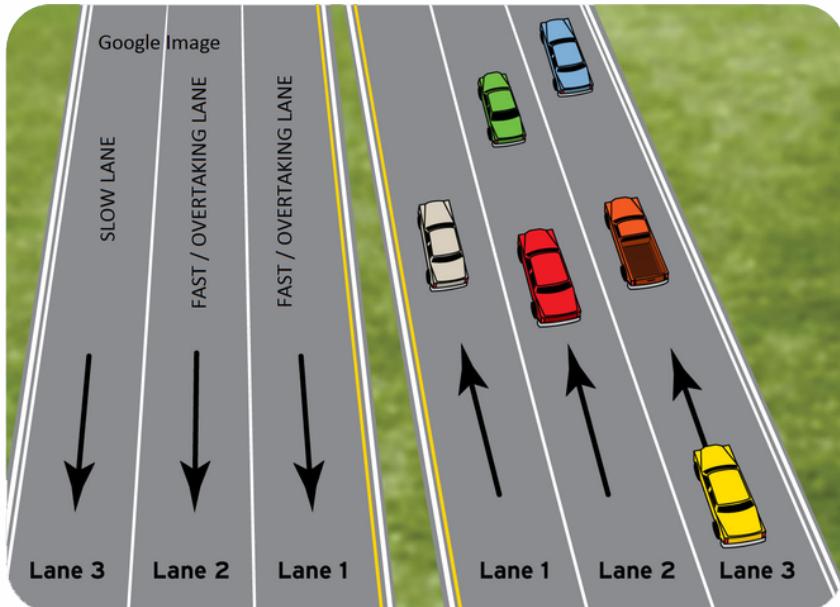
总的来说，从一个框架迁移到另一个框架通常需要大量的工作，并且可能带来不确定的结果。如果你在使用 Vue 的应用中遇到性能问题，我会建议首先寻找优化现有代码的机会，例如使用 Vue 的异步组件，优化依赖追踪，或者使用 Webpack 的代码分割等。

但当你在确认某框架给你带来了FPS的提升，请保持代码在同样的环境和你对该框架有足够深了解。

Lane是什么？解决了React什么问题。原理是什么？

答案

- React 17 的 lanes 模型和 Concurrent Mode 都是为了更好地支持 React Suspense，它们在一起可以更好地处理复杂的异步更新和任务调度。
- 在 React 16 中，即使是在启用了 Concurrent Mode 的情况下，Suspense 也可能在一些情况下表现得不够理想。当多个 Suspense 组件同时进行数据加载时，它们可能会阻塞其他的更新，甚至阻塞整个应用，直到所有的数据都加载完成。这可能会导致不必要的渲染延迟和用户体验下降。
- 在 React 17 中，通过引入 lanes 模型，React 可以更智能地处理并调度各种不同的更新。Suspense 组件现在可以被分配到不同的 lane 上，这使得 React 能够更好地管理和调度 Suspense 组件的加载和渲染。对于那些被 Suspense 捕获的异步更新，React 可以暂时将它们推迟，而去优先处理其他更高优先级的更新，从而改善应用的响应速度和性能。



3. 手写高频考题

- React高频Hooks手写(基础的SetState)

```
let globalState = {};
let globalSubscribers = {};
let stateIndex = 0;

function useState(initialValue) {
  const currentIndex = stateIndex;
  stateIndex++;
  if (!(currentIndex in globalState)) {
    globalState[currentIndex] = initialValue;
    globalSubscribers[currentIndex] = new Set();
  }

  const setState = (newState) => {
    if (typeof newState === 'function') {
      newState = newState(globalState[currentIndex]);
    }
    globalState[currentIndex] = newState;
    //触发所有的订阅者 进行数据的更新
    for (const subscriber of globalSubscribers[currentIndex]) {
      subscriber(newState);
    }
  };
}

const subscribe = (subscriber) => {
```

```

globalSubscribers[currentIndex].add(subscriber);

return () => {
  globalSubscribers[currentIndex].delete(subscriber);
};

};

return [globalState[currentIndex], setState, subscribe];
}

// 使用例子
const [count, setCount, subscribeCount] = useState(0);
subscribeCount((newValue) => {
  console.log('count changed:', newValue);
});
console.log('count:', count);
setCount(1);

// 使用例子
const [count1, setCount1, subscribeCount1] = useState(1);
subscribeCount1((newValue) => {
  console.log('count changed 1', newValue);
});
console.log('count1:', count1);
setCount1((count) => count + 2);

```

React FierNode链表伪代码

答案：

- 在编程和数据结构中，链表是一种基础的数据结构类型。它由一系列的节点组成，每个节点包含数据和指向下一个节点的引用。以下是一个简单的链表节点的 JavaScript 实现：

```

class ListNode {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}

```

在这个 `ListNode` 类中，`data` 属性用于存储节点的数据，`next` 属性用于存储对下一个节点的引用。当 `next` 为 `null` 时，表示这是链表的末尾。在 React 的 Fiber 架构中，Fiber 节点就形成了一种类似链表的数据结构。每个 Fiber 节点都有一个 `child` 属性和一个 `sibling` 属性，它们分别用于引用该节点的第一个子节点和下一个兄弟节点。以下是一个简化的示意代码：

```
class FiberNode {
  constructor(component) {
    this.component = component;
    this.child = null;
    this.sibling = null;
    this.return = null; // 父节点
  }
}
```

在这个 `FiberNode` 类中，`component` 属性用于存储节点的组件数据，`child` 属性用于存储对第一个子节点的引用，`sibling` 属性用于存储对下一个兄弟节点的引用，`return` 属性则是对父节点的引用。通过这种方式，React 构建了一个 Fiber 树，其中的每个节点都可以通过链表的方式访问其子节点和兄弟节点。这种数据结构使得 React 能够有效地遍历和渲染组件树。

React Scheduler涉及到核心微任务、宏任务代码输出结果考题

答案：见我们的参考代码

React Dom Diff原理

答案：

- 我们需要区分4种情况：

- `key` 相同，`type` 相同 == 复用当前节点

例如：A1 B2 C3 -> A1

- `key` 相同，`type` 不同 == 不存在任何复用的可能性

例如：A1 B2 C3 -> B1

- `key` 不同, `type` 相同 == 当前节点不能复用
- `key` 不同, `type` 不同 == 当前节点不能复用

整体流程分为4步。

1. 将 `current` 中所有同级 `fiber` 保存在 `Map` 中
2. 遍历 `newChild` 数组, 对于每个遍历到的 `element`, 存在两种情况:
 3. ■ 在 `Map` 中存在对应 `current fiber`, 且可以复用
 - 在 `Map` 中不存在对应 `current fiber`, 或不能复用
4. 判断是插入还是移动 (如下)
5. 最后 `Map` 中剩下的都标记删除

4. 同构考题

React的同构开发你是如何部署的? 使用Next还是自己开发的? 流式渲染是什么有什么好处?

答案:

- React 的同构开发, 又称为Isomorphic JavaScript, 是指同一份 React 代码能够在服务器端执行并生成HTML, 然后在浏览器端接管渲染并进一步响应用户交互的开发方式。这样做的优势包括:
 1. 提升首屏渲染速度, 增强用户体验。
 2. 对于搜索引擎优化(SEO)有益, 因为搜索引擎爬虫可以直接解析服务器返回的 HTML 内容。

关于部署方式, 两种常见的策略是:

1. 使用 Next.js: Next.js 是一个流行的、基于 React 的通用 JavaScript框架。它处理了很多同构开发中的常见问题, 包括路由、数据预取和预渲染等。使用 Next.js 通常可以节省大量开发时间, 并且提供了一个成熟稳定的平台。
2. 自己开发: 这需要使用 Node.js 服务器 (如 Express) 来预渲染 React 应用, 然后发送生成的 HTML 到浏览器。这种方式需要解决很多细节问题, 例如: 代码分割、数据预取、路由匹配等。

3. 云平台选择：参考视频课程 主要有AWS和Cloudflare

在部署同构应用时，你需要在服务器上部署 Node.js 环境，并在其中运行你的 Next.js 或自己开发的应用。你还需要配置适当的反向代理（如 Nginx），以便将请求转发到 Node.js 服务器。

关于流式渲染，它是指服务器生成 HTML 的过程可以以数据流（stream）的形式逐步发送到客户端，而不是等待全部内容生成完成后一次性发送。这样做的优势主要有两点：

1. 提升首屏渲染速度：浏览器可以更早地开始解析和渲染页面，提高用户体验。
2. 减少服务器内存使用：由于服务器不需要为每个请求保存完整的 HTML 字符串，因此可以显著减少服务器的内存使用。

React服务端渲染需要进行Hydrate么？哪些版本需要？据我所了解Qwik是去调和的，为什么呢？

答案：

- 在React服务端渲染（Server Side Rendering, SSR）中，hydrate是一种重要的步骤。在服务器端渲染的过程中，React会生成HTML字符串，然后在浏览器端，React需要接管这些已经渲染好的HTML，使其变得可以交互。这个过程称为hydration。

从React 16开始，React引入了 `ReactDOM.hydrate` 来替代 `ReactDOM.render` 进行hydration。如果你在服务器端渲染了一些内容，然后想在客户端上接管这个接口，你需要使用 `ReactDOM.hydrate` 而不是 `ReactDOM.render`。

从React 18开始，React官方计划引入部分hydration的能力，这将改善用户体验，提高性能。部分hydration的主要思想是：React应用在初始渲染后并不立即hydrate全部组件，而是根据用户的交互行为或者组件的优先级，按需进行hydrate。这意味着，React 18的应用在启动时可能不需要进行全部的hydrate，但在具体的使用过程中，还是需要hydrate操作来使得服务端渲染的HTML变得可交互。

- Qwik 是由 Google 推出的一款面向前端的，专注于性能优化的 JavaScript 框架。Qwik 的设计理念是提供最小的启动时间，并保持应用运行时的速度。为了实现这一目标，Qwik 采取了一种被称为 "HTML-oriented programming" 的方法。

Qwik 认为，页面的 HTML 本身就应该能够描述出应用的状态，以及如何从一个状态转变到另一个状态。在这个思想的指导下，Qwik 将组件直接链接到 HTML 元素，并将事件处理器直接链接到用户的操作（如点击）。这使得在没有 JavaScript 的情况下，浏览器仍然可以正确地呈现和导航 Qwik 应用。

由于 Qwik 通过 HTML 描述应用状态，因此在页面加载后，Qwik 可以直接加载和执行与当前页面状态对应的代码，而无需进行 diff 或 hydration 操作。这是因为，与其他前端框架（如 React）不同，Qwik 并不需要通过 JavaScript 去创建或更新 DOM，它直接使用了服务器渲染的 HTML，大大提高了加载速度。

因此，Qwik 是一个 "无调和 (diffing)" 的框架，它避免了在客户端进行复杂的 diff 计算和 DOM 操作，从而减少了首次内容绘制（First Contentful Paint, FCP）的时间。

但是需要注意的是，"HTML-oriented programming" 的方法也有其局限性，例如它可能不适合复杂的单页应用（SPA）或者需要频繁更新状态的应用。在选择使用 Qwik 或者其它框架时，你需要根据你的具体需求来做出决定。

React同构渲染如果提高性能问题？你是怎么落地的。同构解决了哪些性能指标。

答案：

- React同构渲染，也就是服务器端渲染（SSR），在某些情况下可以提高性能，并改善用户体验。以下是其可能的性能优势：
 1. 提升首屏加载速度：SSR会将React应用的初始状态渲染为HTML，然后发送到客户端，这样浏览器就可以更早地开始显示页面内容，而不用等待所有的JavaScript代码下载、解析和执行完毕。
 2. 对SEO友好：搜索引擎爬虫更容易解析服务器渲染的HTML，而不是客户端渲染的JavaScript。

同构渲染一般会对以下性能指标产生影响：

- 首次内容绘制（FCP，First Contentful Paint）：SSR可以减少FCP时间，因为浏览器可以在接收到服务器返回的HTML后立即开始渲染页面。
- 首次有效绘制（FMP，First Meaningful Paint）：SSR也可能减少FMP时间，因为服务器预渲染的页面通常包含了页面的主要内容。
- 其次FMP（First Meaningful Paint，首次有意义的绘制）已经从Web性能指标中被移除了。现在，一般使用以下指标来评估网页的加载性能和用户体验：

1. **LCP (Largest Contentful Paint, 最大内容绘制)** : LCP度量了在页面加载过程中，最大的（通常是主要的）内容元素何时呈现在屏幕上。比如页面上的一张主图或者一段文字。LCP提供了用户观看到页面主要内容的速度。
2. **FID (First Input Delay, 首次输入延迟)** : FID度量用户首次交互（例如点击链接、按钮等）和浏览器开始处理此交互之间的时间。这个指标关注的是网页的交互反应速度。
3. **CLS (Cumulative Layout Shift, 累积布局偏移)** : CLS度量视觉内容在加载过程中发生意外布局偏移的程度。这个指标关注的是页面的视觉稳定性。

以上这些指标都被纳入了Google的Web Vitals，用于评估和改善用户体验。LCP、FID和CLS是核心Web Vitals，它们是Google推荐所有网站关注的最重要的性能指标。

在实际落地过程中，可能会遇到以下一些挑战：

- 数据预取：在服务器端，我们需要在渲染页面之前预取数据。常见的解决方案包括在React组件中使用静态的loadData方法，或者使用像Next.js这样的框架。
- 组件的生命周期方法：只有部分生命周期方法会在服务器端渲染过程中被调用，例如 `componentDidMount` 和 `componentDidUpdate` 就不会被调用。因此需要确保应用的逻辑不依赖这些只在客户端执行的生命周期方法。
- 性能和资源使用：服务器端渲染会占用更多的服务器资源（CPU和内存）。需要通过适当的架构和优化来保证服务器的性能，例如使用流式渲染、设置合理的缓存策略等。

实践中，可以采用Next.js等同构框架来简化部分工作，这些框架提供了对React SSR的良好支持，包括自动数据预取、路由管理、优化构建等。

同构注水脱水是什么意思？React 进行ServerLess渲染时候项目需要做哪些改变？

答案：

- React同构渲染涉及到“注水”和“脱水”的概念，这两个术语来源于React服务器端渲染（SSR）的两个重要步骤。

1. **脱水 (Dehydration)** : 在服务器端, React将组件树渲染为HTML字符串, 并生成与应用状态相关的数据 (即"脱水"数据)。这个过程就被称为“脱水”。然后, 服务器将渲染后的HTML和脱水数据一起发送到客户端。
2. **注水 (Rehydration)** : 在客户端, React使用服务器发送过来的脱水数据来恢复应用的状态, 这个过程被称为“注水”。然后React会对服务器渲染的HTML和客户端组件树进行匹配, 如果匹配成功, React将接管这些已经存在的DOM节点, 使其变得可以交互。

这种同构渲染的方式有几个好处: 首先, 用户可以更早地看到页面内容, 因为浏览器无需等待所有的JavaScript代码下载和执行完毕就可以显示服务器渲染的HTML; 其次, 由于服务器已经生成了初始状态, 客户端无需再次获取数据; 最后, React可以复用服务器渲染的DOM, 避免了额外的DOM操作, 从而提高了性能。

但需要注意, 这种方式也有一些缺点, 例如服务器端渲染可能会增加服务器的负载, 以及在注水过程中可能会出现一些问题, 如数据不一致或者错误的状态等。

- 具体next.js部署请参阅官方文档 <https://developers.cloudflare.com/pages/framework-guides/deploy-a-nextjs-site/>

5. 附加题

刚才你也提到了可以部署的平台有很多, 那么每个平台进行JS冷启动的区别是什么呢?

答案:

- 首先, 让我们了解一下 Cloudflare Workers 和 AWS Lambda (作为 AWS 的一部分 Serverless 技术)。

Cloudflare Workers 基于 V8 Isolate (基于 Google V8 引擎的轻量级实例) 环境运行, 可以在全球范围内的 Cloudflare 数据中心部署并运行 JavaScript 代码。每个请求都会在一个新的 Isolate 中执行, 每个 Isolate 都拥有自己的全局变量和函数执行栈。Isolate 非常轻量, 创建和销毁都很快, 这使得 Cloudflare Workers 能够实现接近零的冷启动时间。

AWS Lambda 则采用了稍微不同的方法。AWS Lambda 将函数打包成容器，并在请求时启动这些容器。这种方法相比于 Cloudflare Workers 更为重量级，因此 Lambda 函数的冷启动时间通常会更长。不过，一旦一个函数被启动，AWS 会尽可能地保持其“热”的状态，以便能够快速响应后续请求。因此，频繁使用的函数通常会有很好的性能。

对于 JavaScript 函数，AWS Lambda 使用 Node.js 运行环境。在该环境中，函数在首次被调用时（或者在一段时间内未被调用后）需要进行初始化，这会花费一些时间。这个过程称为“冷启动”。在冷启动过程中，AWS 需要找到可用的计算资源，加载函数的代码和所有依赖项，然后执行函数的初始化代码。这个过程可能会花费一些时间，尤其是对于大型函数和/或具有许多依赖项的函数。

总结一下，Cloudflare Workers 的冷启动时间几乎为零，因为它们基于轻量级的 V8 Isolate，可以快速创建和销毁。而 AWS Lambda 的冷启动时间通常会更长，因为它需要启动更重量级的容器，并执行函数的初始化代码。然而，一旦 Lambda 函数被启动并保持“热”状态，它可以快速响应后续请求。这两种方法各有优缺点，适用于不同的应用场景。

6. 变态题

- ✓ 请问源码scheduler下fork的Scheduler.js 209行实现了什么？参数有几个
有几个 

志佳老师