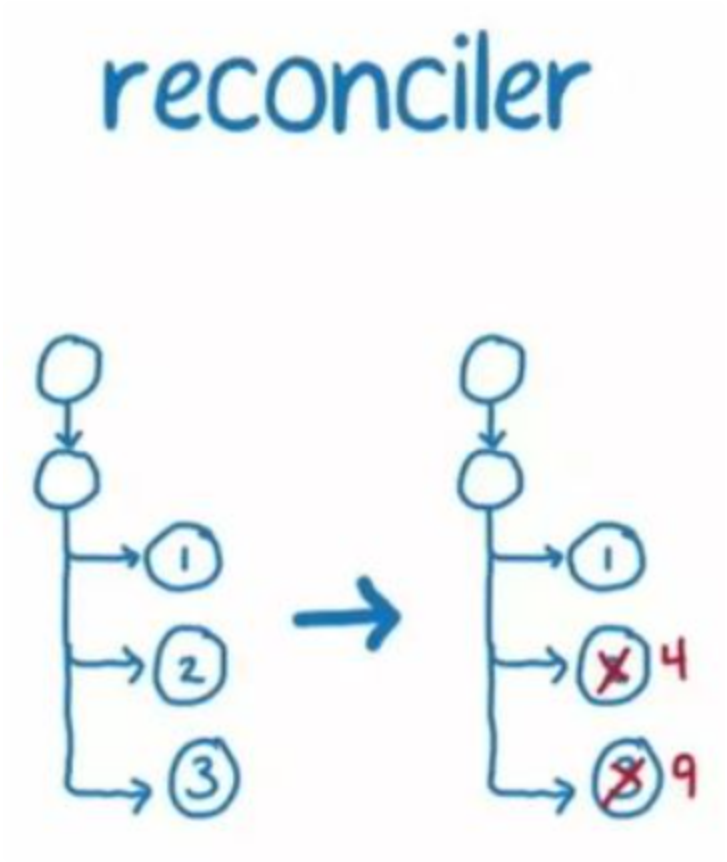


20.2. 是什么

React Fiber 是 Facebook 花费两年余时间对 React 做出的一个重大改变与优化，是对 React 核心算法的一次重新实现。从Facebook在 React Conf 2017 会议上确认，React Fiber 在React 16 版本发布

在 `react` 中，主要做了以下的操作：

- 为每个增加了优先级，优先级高的任务可以中断低优先级的任务。然后再重新，注意是重新执行优先级低的任务
- 增加了异步任务，调用requestIdleCallback api，浏览器空闲的时候执行
- dom diff树变成了链表，一个dom对应两个fiber（一个链表），对应两个队列，这都是为找到被中断的任务，重新执行

从架构角度来看， `Fiber` 是对 `React` 核心算法（即调和过程）的重写

从编码角度来看， `Fiber` 是 `React` 内部所定义的一种数据结构，它是 `Fiber` 树结构的节点单位，也就是 `React 16` 新架构下的虚拟 `DOM`

一个 `fiber` 就是一个 `JavaScript` 对象，包含了元素的信息、该元素的更新操作队列、类型，其数据结构如下：

JavaScript | 复制代码

```
type Fiber = {
  // 用于标记fiber的WorkTag类型，主要表示当前fiber代表的组件类型如FunctionComponent、ClassComponent等
  tag: WorkTag,
  // ReactElement里面的key
  key: null | string,
  // ReactElement.type，调用`createElement`的第一个参数
  elementType: any,
  // The resolved function/class/ associated with this fiber.
  // 表示当前代表的节点类型
  type: any,
  // 表示当前FiberNode对应的element组件实例
  stateNode: any,

  // 指向他在Fiber节点树中的`parent`，用来在处理完这个节点之后向上返回
  return: Fiber | null,
  // 指向自己的第一个子节点
  child: Fiber | null,
  // 指向自己的兄弟结构，兄弟节点的return指向同一个父节点
  sibling: Fiber | null,
  index: number,

  ref: null | (((handle: mixed) => void) & { _stringRef: ?string }) | RefObject,

  // 当前处理过程中的组件props对象
  pendingProps: any,
  // 上一次渲染完成之后的props
  memoizedProps: any,

  // 该Fiber对应的组件产生的Update会存放在这个队列里面
  updateQueue: UpdateQueue<any> | null,

  // 上一次渲染的时候的state
  memoizedState: any,

  // 一个列表，存放这个Fiber依赖的context
  firstContextDependency: ContextDependency<mixed> | null,

  mode: TypeOfMode,

  // Effect
  // 用来记录Side Effect
  effectTag: SideEffectTag,

```

```
    // 单链表用来快速查找下一个side effect
    nextEffect: Fiber | null,

    // 子树中第一个side effect
    firstEffect: Fiber | null,
    // 子树中最后一个side effect
    lastEffect: Fiber | null,

    // 代表任务在未来的哪个时间点应该被完成，之后版本改名为 lanes
    expirationTime: ExpirationTime,

    // 快速确定子树中是否有不在等待的变化
    childExpirationTime: ExpirationTime,

    // fiber的版本池，即记录fiber更新过程，便于恢复
    alternate: Fiber | null,
}
```

20.3. 如何解决

`Fiber` 把渲染更新过程拆分成多个子任务，每次只做一小部分，做完看是否还有剩余时间，如果有继续下一个任务；如果没有，挂起当前任务，将时间控制权交给主线程，等主线程不忙的时候在继续执行

即可以中断与恢复，恢复后也可以复用之前的中间状态，并给不同的任务赋予不同的优先级，其中每个任务更新单元为 `React Element` 对应的 `Fiber` 节点

实现的上述方式的是 `requestIdleCallback` 方法

`window.requestIdleCallback()` 方法将在浏览器的空闲时段内调用的函数排队。这使开发者能够在主事件循环上执行后台和低优先级工作，而不会影响延迟关键事件，如动画和输入响应

首先 React 中任务切割为多个步骤，分批完成。在完成一部分任务之后，将控制权交回给浏览器，让浏览器有时间再进行页面的渲染。等浏览器忙完之后有剩余时间，再继续之前 React 未完成的任务，是一种合作式调度。

该实现过程是基于 `Fiber` 节点实现，作为静态的数据结构来说，每个 `Fiber` 节点对应一个 `React element`，保存了该组件的类型（函数组件/类组件/原生组件等等）、对应的 DOM 节点等信息。

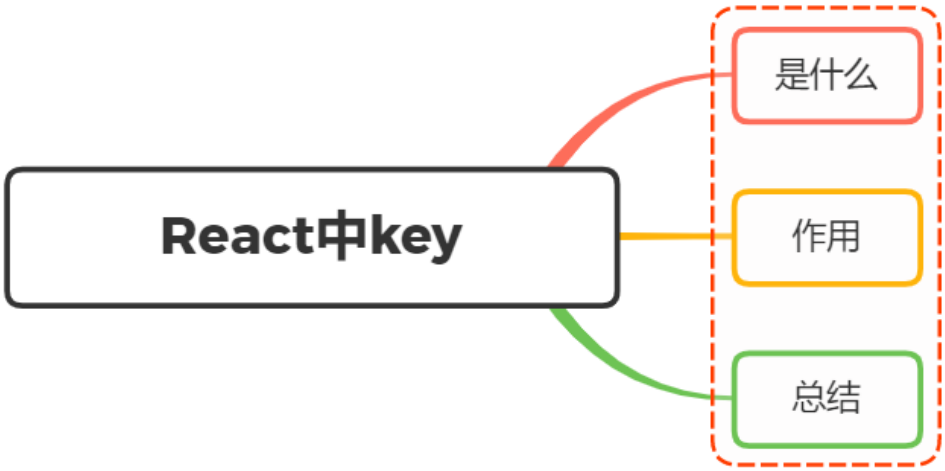
作为动态的工作单元来说，每个 `Fiber` 节点保存了本次更新中该组件改变的状态、要执行的工作。

每个 Fiber 节点有个对应的 `React element`，多个 `Fiber` 节点根据如下三个属性构建一颗树：

```javascript
// 指向父级Fiber节点
this.return = null
// 指向子Fiber节点
this.child = null
// 指向右边第一个兄弟Fiber节点
this.sibling = null
```

通过这些属性就能找到下一个执行目标

21. React中的key有什么作用?



21.1. 是什么

首先，先给出 `react` 组件中进行列表渲染的一个示例：

JSX | 复制代码

```jsx
const data = [
  { id: 0, name: 'abc' },
  { id: 1, name: 'def' },
  { id: 2, name: 'ghi' },
  { id: 3, name: 'jkl' }
];

const ListItem = (props) => {
  return <li>{props.name}</li>;
};

const List = () => {
  return (
    <ul>
      {data.map((item) => (
        <ListItem name={item.name}></ListItem>
      ))}
    </ul>
  );
};
```

然后在输出就可以看到 `react` 所提示的警告信息：

LaTeX | 复制代码

```latex
Each child in a list should have a unique "key" prop.
```

根据意思就可以得到渲染列表的每一个子元素都应该需要一个唯一的 `key` 值

在这里可以使用列表的 `id` 属性作为 `key` 值以解决上面这个警告

JSX | 复制代码

```jsx
const List = () => {
  return (
    <ul>
      {data.map((item) => (
        <ListItem name={item.name} key={item.id}></ListItem>
      ))}
    </ul>
  );
};
```

21.2. 作用

跟 `Vue` 一样， `React` 也存在 `Diff` 算法，而元素 `key` 属性的作用是用于判断元素是新创建的还是被移动的元素，从而减少不必要的元素渲染

因此 `key` 的值需要为每一个元素赋予一个确定的标识

如果列表数据渲染中，在数据后面插入一条数据， `key` 作用并不大，如下：

JSX | 复制代码

```jsx
this.state = {
    numbers:[111,222,333]
}

insertMovie() {
  const newMovies = [...this.state.numbers, 444];
  this.setState({
    movies: newMovies
  })
}

<ul>
    {
        this.state.movies.map((item, index) => {
            return <li>{item}</li>
        })
    }
</ul>
```

前面的元素在 `diff` 算法中，前面的元素由于是完全相同的，并不会产生删除创建操作，在最后一个比较的时候，则需要插入到新的 `DOM` 树中

因此，在这种情况下，元素有无 `key` 属性意义并不大

下面再来看看在前面插入数据时，使用 `key` 与不使用 `key` 的区别：

JavaScript | 复制代码

```javascript
insertMovie() {
  const newMovies = [000 ,...this.state.numbers];
  this.setState({
    movies: newMovies
  })
}
```

当拥有 `key` 的时候， `react` 根据 `key` 属性匹配原有树上的子元素以及最新树上的子元素，像上述情况只需要将000元素插入到最前面位置

当没有 `key` 的时候，所有的 `li` 标签都需要进行修改

同样，并不是拥有 `key` 值代表性能越高，如果说只是文本内容改变了，不写 `key` 反而性能和效率更高

主要是因为不写 `key` 是将所有的文本内容替换一下，节点不会发生变化

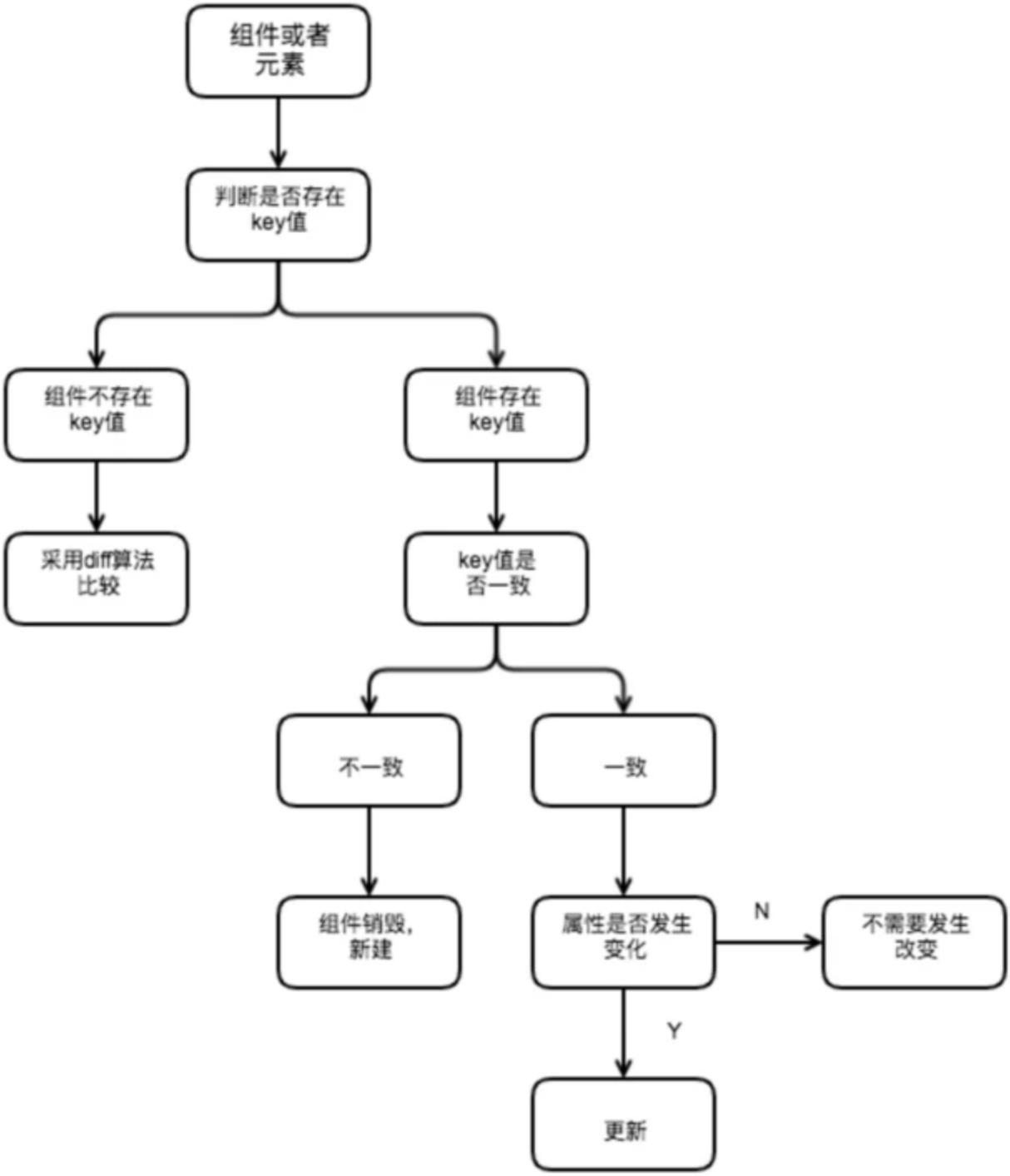
而写 `key` 则涉及到了节点的增和删，发现旧 `key` 不存在了，则将其删除，新 `key` 在之前没有，则插入，这就增加性能的开销

21.3. 总结

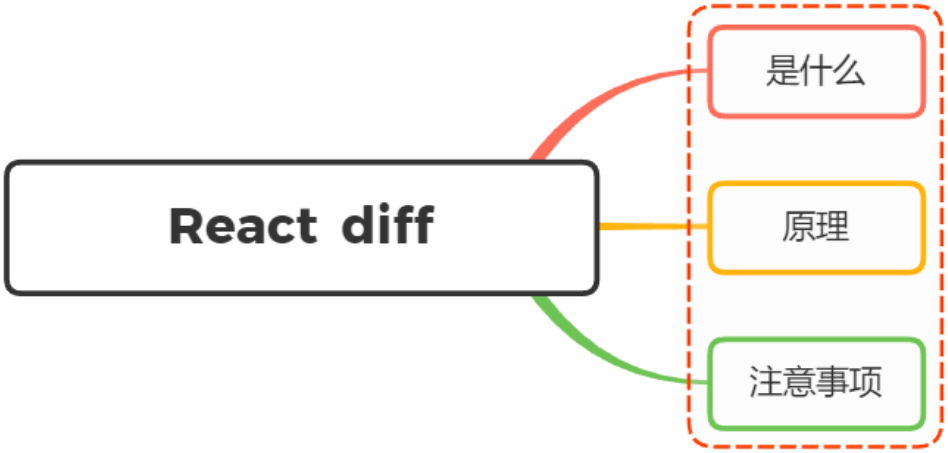
良好使用 `key` 属性是性能优化的非常关键的一步，注意事项为：

- key 应该是唯一的
- key不要使用随机值（随机数在下一次 render 时，会重新生成一个数字）
- 使用 index 作为 key值，对性能没有优化

`react` 判断 `key` 的流程具体如下图：



22. 说说React diff的原理是什么?

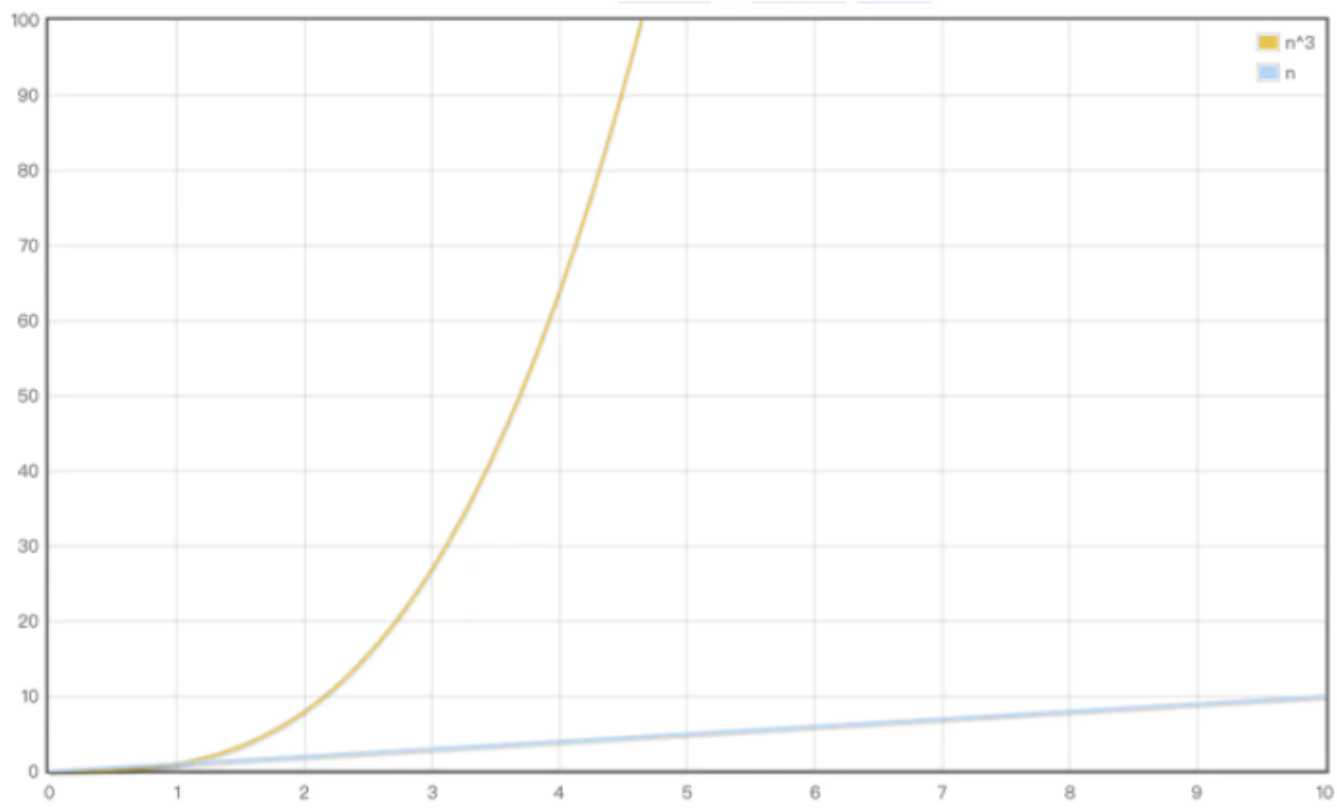


22.1. 是什么

跟 `Vue` 一致， `React` 通过引入 `Virtual DOM` 的概念，极大地避免无效的 `Dom` 操作，使我们的页面的构建效率提到了极大的提升

而 `diff` 算法就是更高效地通过对比新旧 `Virtual DOM` 来找出真正的 `Dom` 变化之处

传统diff算法通过循环递归对节点进行依次对比，效率低下，算法复杂度达到 O(n^3)， `react` 将算法进行一个优化，复杂度 `O(n)` ，两者效率差距如下图：



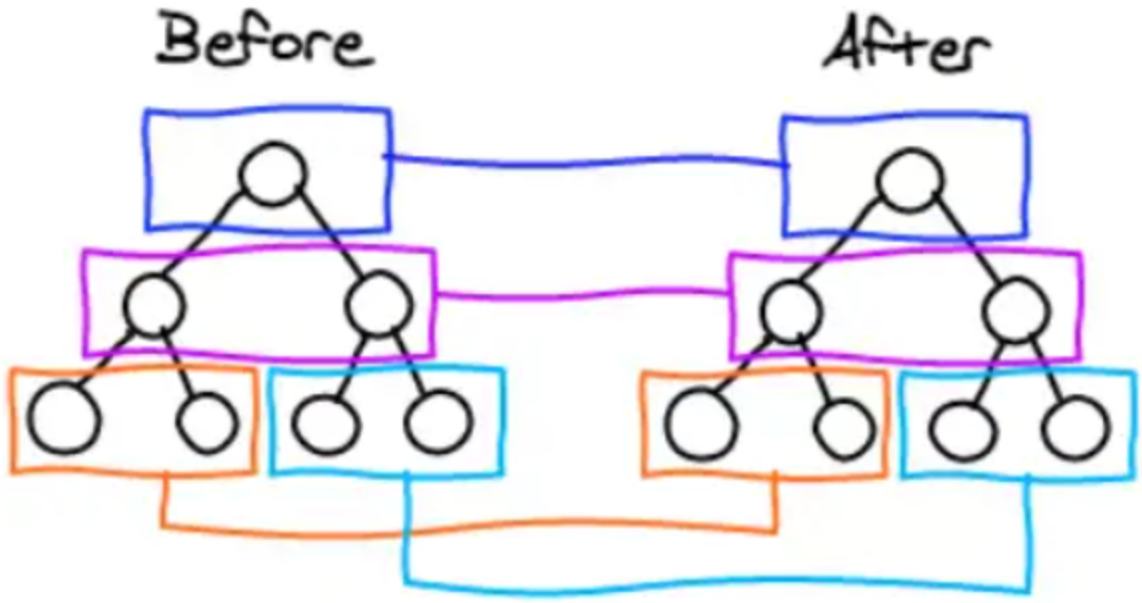
22.2. 原理

`react` 中 `diff` 算法主要遵循三个层级的策略：

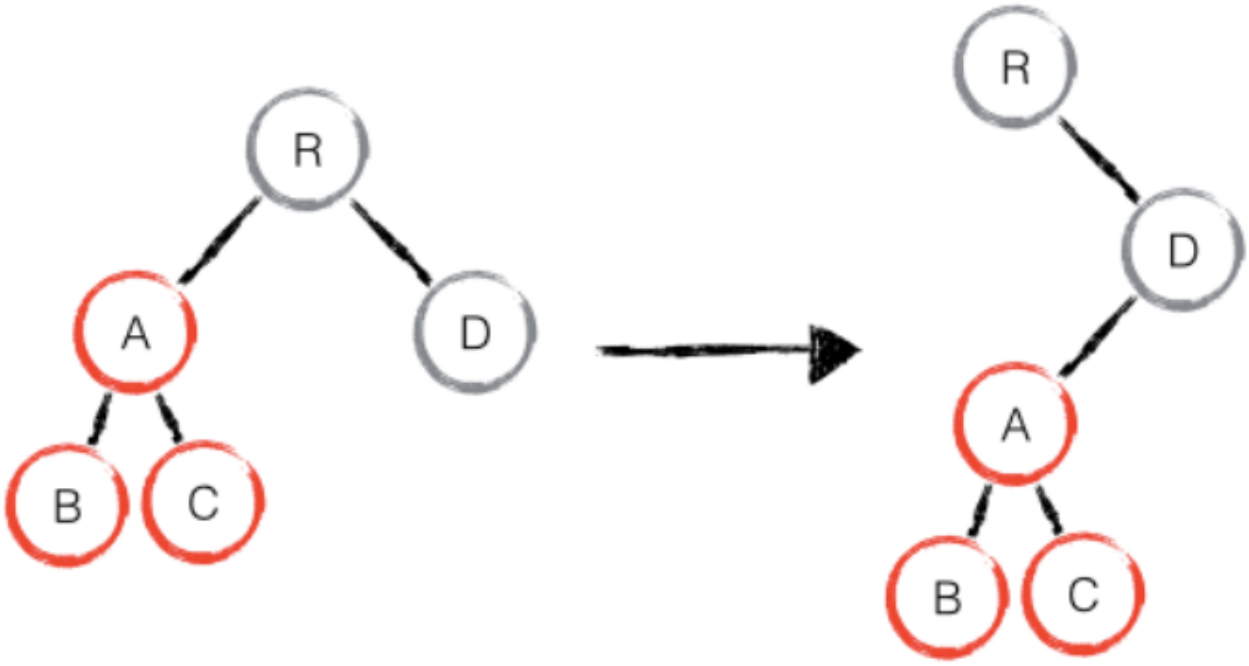
- tree层级
- conponent 层级
- element 层级

22.2.1. tree层级

`DOM` 节点跨层级的操作不做优化，只会对相同层级的节点进行比较



只有删除、创建操作，没有移动操作，如下图：

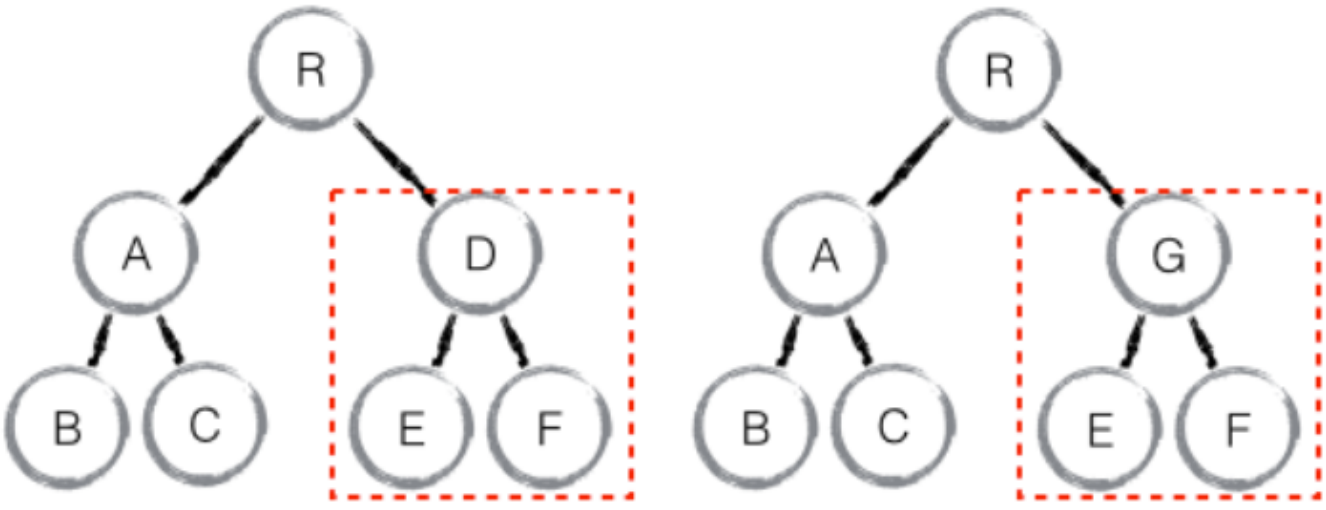


`react` 发现新树中，R节点下没有了A，那么直接删除A，在D节点下创建A以及下属节点

上述操作中，只有删除和创建操作

22.2.2. conponent层级

如果是同一个类的组件，则会继续往下 `diff` 运算，如果不是一个类的组件，那么直接删除这个组件下的所有子节点，创建新的



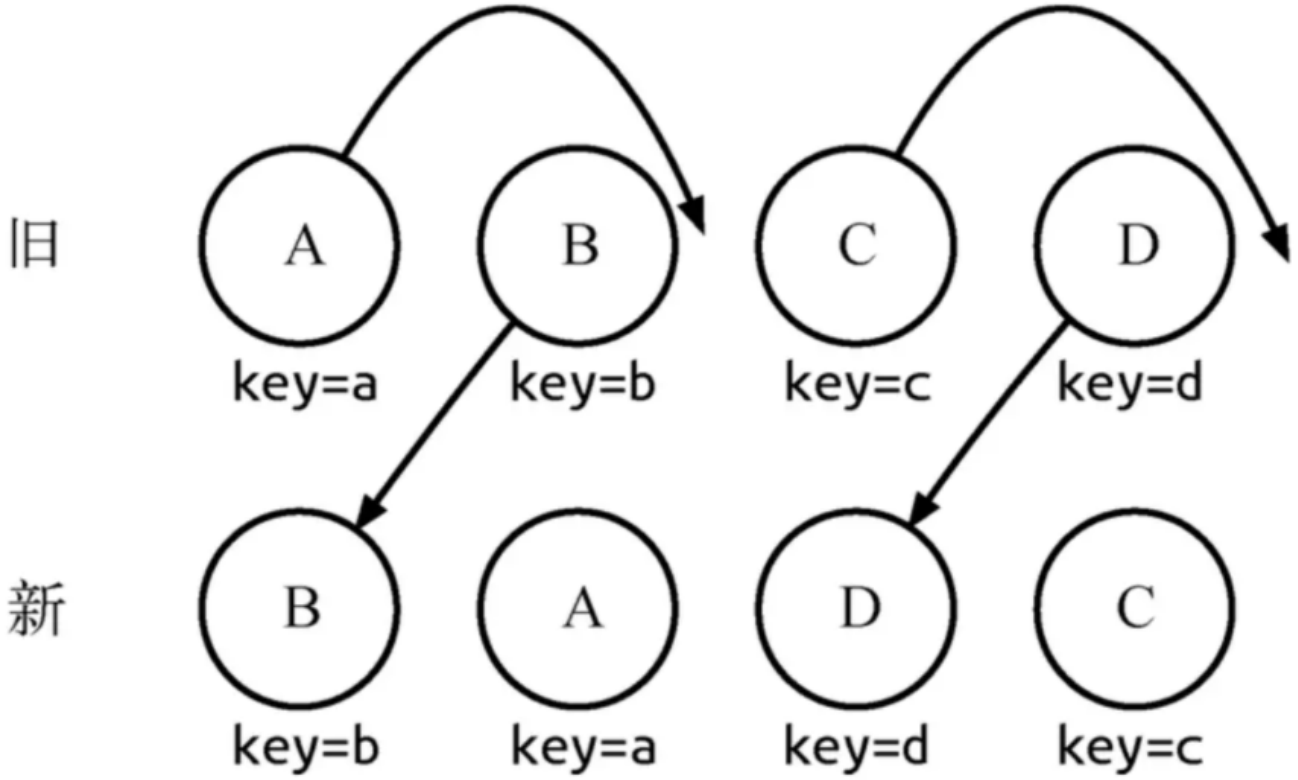
当 `component D` 换成了 `component G` 后，即使两者的结构非常类似，也会将 `D` 删除再重新创建 `G`

22.2.3. element层级

对于比较同一层级的节点们，每个节点在对应的层级用唯一的 `key` 作为标识

提供了 3 种节点操作，分别为 `INSERT_MARKUP`(插入)、`MOVE_EXISTING`(移动)和 `REMOVE_NODE`(删除)

如下场景：



通过 `key` 可以准确地发现新旧集合中的节点都是相同的节点，因此无需进行节点删除和创建，只需要将旧集合中节点的位置进行移动，更新为新集合中节点的位置

流程如下表：

index	节点	oldIndex	maxIndex	操作
0	B	1	0	oldIndex(1)>maxIndex(0),maxIndex=oldIndex,maxIndex变为1
1	A	0	1	oldIndex(0)<maxIndex(1),节点A移动至index(1)的位置
2	D	3	1	oldIndex(3)>maxIndex(1),maxIndex=oldIndex,maxIndex变为3
3	C	2	3	oldIndex(2)<maxIndex(3),节点C移动至index(3)的位置

- index： 新集合的遍历下标。
- oldIndex：当前节点在老集合中的下标
- maxIndex：在新集合访问过的节点中，其在老集合的最大下标

如果当前节点在新集合中的位置比老集合中的位置靠前的话，是不会影响后续节点操作的，这里这时候被动字节不用动

操作过程中只比较oldIndex和maxIndex，规则如下：

- 当oldIndex>maxIndex时，将oldIndex的值赋值给maxIndex
- 当oldIndex=maxIndex时，不操作
- 当oldIndex<maxIndex时，将当前节点移动到index的位置

`diff` 过程如下：

- 节点B：此时 maxIndex=0，oldIndex=1；满足 maxIndex< oldIndex，因此B节点不动，此时maxIndex= Math.max(oldIndex, maxIndex)，就是1
- 节点A：此时maxIndex=1，oldIndex=0；不满足maxIndex< oldIndex，因此A节点进行移动操作，此时maxIndex= Math.max(oldIndex, maxIndex)，还是1
- 节点D：此时maxIndex=1, oldIndex=3；满足maxIndex< oldIndex，因此D节点不动，此时maxIndex= Math.max(oldIndex, maxIndex)，就是3
- 节点C：此时maxIndex=3，oldIndex=2；不满足maxIndex< oldIndex，因此C节点进行移动操作，当前已经比较完了

当ABCD节点比较完成后， `diff` 过程还没完，还会整体遍历老集合中节点，看有没有没用到的节点，有的话，就删除

22.3. 注意事项

对于简单列表渲染而言，不使用 `key` 比使用 `key` 的性能，例如：

将一个[1,2,3,4,5]，渲染成如下的样子：

HTML | 复制代码

```
<div>1</div>
<div>2</div>
<div>3</div>
<div>4</div>
<div>5</div>
```

后续更改成[1,3,2,5,4]，使用 `key` 与不使用 `key` 作用如下：

```HTML
1.加key
<div key='1'>1</div>             <div key='1'>1</div>
<div key='2'>2</div>             <div key='3'>3</div>
<div key='3'>3</div>  ========>  <div key='2'>2</div>
<div key='4'>4</div>             <div key='5'>5</div>
<div key='5'>5</div>             <div key='4'>4</div>
操作：节点2移动至下标为2的位置，节点4移动至下标为4的位置。

2.不加key
<div>1</div>             <div>1</div>
<div>2</div>             <div>3</div>
<div>3</div>  ========>  <div>2</div>
<div>4</div>             <div>5</div>
<div>5</div>             <div>4</div>
操作：修改第1个到第5个节点的innerText
```

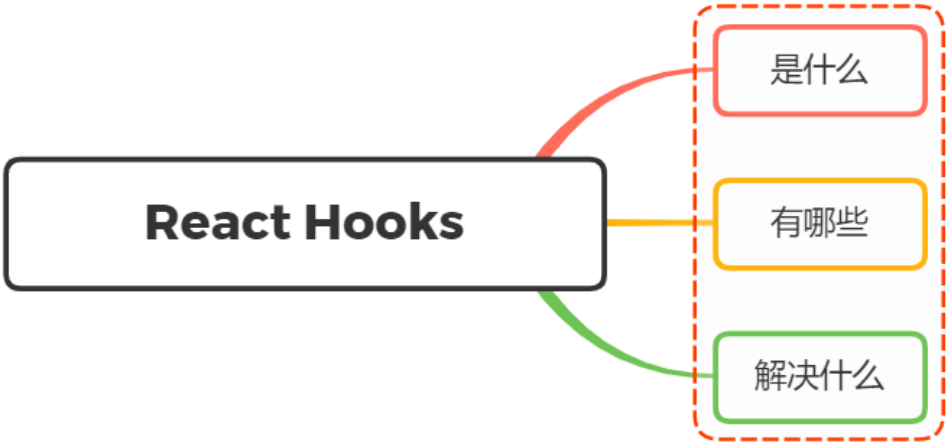
如果我们对这个集合进行增删的操作改成[1,3,2,5,6]

```HTML
1.加key
<div key='1'>1</div>             <div key='1'>1</div>
<div key='2'>2</div>             <div key='3'>3</div>
<div key='3'>3</div>  ========>  <div key='2'>2</div>
<div key='4'>4</div>             <div key='5'>5</div>
<div key='5'>5</div>             <div key='6'>6</div>
操作：节点2移动至下标为2的位置，新增节点6至下标为4的位置，删除节点4。

2.不加key
<div>1</div>             <div>1</div>
<div>2</div>             <div>3</div>
<div>3</div>  ========>  <div>2</div>
<div>4</div>             <div>5</div>
<div>5</div>             <div>6</div>
操作：修改第1个到第5个节点的innerText
```

由于 `dom` 节点的移动操作开销是比较昂贵的，没有 `key` 的情况下要比有 `key` 的性能更好

23. 说说对React Hooks的理解？解决了什么问题？



23.1. 是什么

`Hook` 是 React 16.8 的新增特性。它可以让你在不编写 `class` 的情况下使用 `state` 以及其他的 `React` 特性

至于为什么引入 `hook` ，官方给出的动机是解决长时间使用和维护 `react` 过程中常遇到的问题，例如：

- 难以重用和共享组件中的与状态相关的逻辑
- 逻辑复杂的组件难以开发与维护，当我们的组件需要处理多个互不相关的 local state 时，每个生命周期函数中可能会包含着各种互不相关的逻辑在里面
- 类组件中的this增加学习成本，类组件在基于现有工具的优化上存在些许问题
- 由于业务变动，函数组件不得不改为类组件等等

在以前，函数组件也被称为无状态的组件，只负责渲染的一些工作

因此，现在的函数组件也可以是有状态的组件，内部也可以维护自身的状态以及做一些逻辑方面的处理

23.2. 有哪些

上面讲到， `Hooks` 让我们的函数组件拥有了类组件的特性，例如组件内的状态、生命周期

最常见的 `hooks` 有如下：

- useState
- useEffect
- 其他

23.2.1. useState

首先给出一个例子，如下：

JavaScript | 复制代码

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

function Example() {
  // 声明一个叫 "count" 的 state 变量
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p >
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

在函数组件中通过 `useState` 实现函数内部维护 `state` ，参数为 `state` 默认的值，返回值是一个数组，第一个值为当前的 `state` ，第二个值为更新 `state` 的函数

该函数组件等价于的类组件如下：

JavaScript | 复制代码

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p >
        <button onClick={() => this.setState({ count: this.state.count + 1
})}>
          Click me
        </button>
      </div>
    );
  }
}
```

从上述两种代码分析，可以看出两者区别：

- state声明方式：在函数组件中通过 useState 直接获取，类组件通过constructor 构造函数中设置
- state读取方式：在函数组件中直接使用变量，类组件通过 `this.state.count` 的方式获取
- state更新方式：在函数组件中通过 setCount 更新，类组件通过this.setState()

总的来讲，useState 使用起来更为简洁，减少了 `this` 指向不明确的情况

23.2.2. useEffect

`useEffect` 可以让我们在函数组件中进行一些带有副作用的操作

同样给出一个计时器示例：

JavaScript | 复制代码

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p >
        <button onClick={() => this.setState({ count: this.state.count + 1
 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

从上面可以看见，组件在加载和更新阶段都执行同样操作

而如果使用 `useEffect` 后，则能够将相同的逻辑抽离出来，这是类组件不具备的方法

对应的 `useEffect` 示例如下：

JSX | 复制代码

```jsx
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {    document.title = `You clicked ${count} times`;  });
  return (
    <div>
      <p>You clicked {count} times</p >
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

`useEffect` 第一个参数接受一个回调函数，默认情况下， `useEffect` 会在第一次渲染和更新之后都会执行，相当于在 `componentDidMount` 和 `componentDidUpdate` 两个生命周期函数中执行回调

如果某些特定值在两次重渲染之间没有发生变化，你可以跳过对 effect 的调用，这时候只需要传入第二个参数，如下：

JavaScript | 复制代码

```javascript
useEffect(() => {
   document.title = `You clicked ${count} times`;
}, [count]); // 仅在 count 更改时更新
```

上述传入第二个参数后，如果 `count` 的值是 `5` ，而且我们的组件重渲染的时候 `count` 还是等于 `5` ，React 将对前一次渲染的 `[5]` 和后一次渲染的 `[5]` 进行比较，如果是相等则跳过 `effects` 执行

回调函数中可以返回一个清除函数，这是 `effect` 可选的清除机制，相当于类组件中 `componentwillUnmount` 生命周期函数，可做一些清除副作用的操作，如下：

```jsx
useEffect(() => {
    function handleStatusChange(status) {
        setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
        ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
});
```

所以，`useEffect` 相当于 `componentDidMount`，`componentDidUpdate` 和 `componentWillUnmount` 这三个生命周期函数的组合

23.2.3. 其它 hooks

在组件通信过程中可以使用 `useContext`，`refs` 学习中我们也用到了 `useRef` 获取 `DOM` 结构.....

还有很多额外的 `hooks`，如：

- useReducer
- useCallback
- useMemo
- useRef

23.3. 解决什么

通过对上面的初步认识，可以看到 `hooks` 能够更容易解决状态相关的重用的问题：

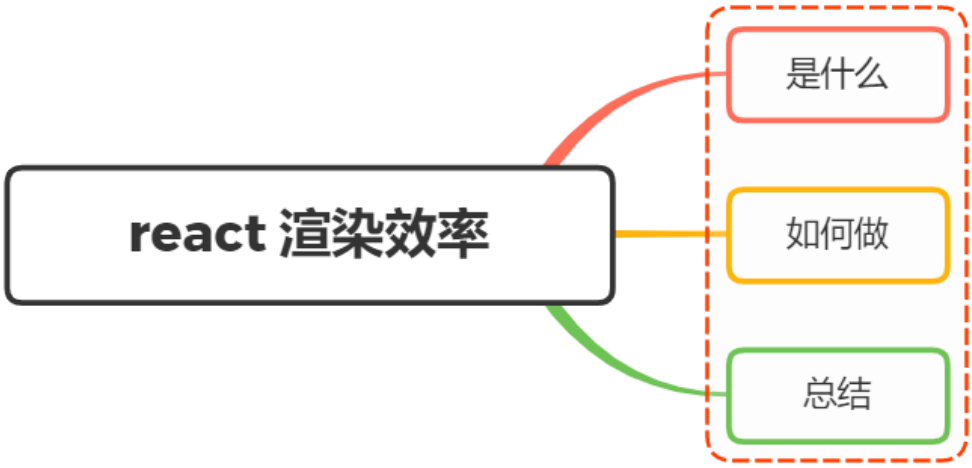
- 每调用useHook一次都会生成一份独立的状态
- 通过自定义hook能够更好的封装我们的功能

编写 `hooks` 为函数式编程，每个功能都包裹在函数中，整体风格更清爽，更优雅

`hooks` 的出现，使函数组件的功能得到了扩充，拥有了类组件相似的功能，在我们日常使用中，使用 `hooks` 能够解决大多数问题，并且还拥有代码复用机制，因此优先考虑 `hooks`

24. 说说你是如何提高组件的渲染效率的？在React中如

何避免不必要的render?



24.1. 是什么

`react` 基于虚拟 `DOM` 和高效 `Diff` 算法的完美配合，实现了对 `DOM` 最小粒度的更新，大多数情况下，`React` 对 `DOM` 的渲染效率足以我们的业务日常

复杂业务场景下，性能问题依然会困扰我们。此时需要采取一些措施来提升运行性能，避免不必要的渲染则是业务中常见的优化手段之一

24.2. 如何做

在之前文章中，我们了解到 `render` 的触发时机，简单来讲就是类组件通过调用 `setState` 方法，就会导致 `render` ，父组件一旦发生 `render` 渲染，子组件一定也会执行 `render` 渲染

从上面可以看到，父组件渲染导致子组件渲染，子组件并没有发生任何改变，这时候就可以从避免无谓的渲染，具体实现的方式有如下：

- shouldComponentUpdate
- PureComponent
- React.memo

24.2.1. shouldComponentUpdate

通过 `shouldComponentUpdate` 生命周期函数来比对 `state` 和 `props` ，确定是否要重新渲染

默认情况下返回 `true` 表示重新渲染，如果不希望组件重新渲染，返回 `false` 即可

24.2.2. PureComponent

跟 `shouldComponentUpdate` 原理基本一致，通过对 `props` 和 `state` 的浅比较结果来实现 `shouldComponentUpdate`，源码大致如下：

JavaScript | 复制代码

```
if (this._compositeType === CompositeTypes.PureClass) {
    shouldUpdate = !shallowEqual(prevProps, nextProps) || ! shallowEqual(inst.state, nextState);
}
```

`shallowEqual` 对应方法大致如下：

JavaScript | 复制代码

```javascript
const hasOwnProperty = Object.prototype.hasOwnProperty;

/**
 * is 方法来判断两个值是否是相等的值，为何这么写可以移步 MDN 的文档
 * https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/is
 */
function is(x: mixed, y: mixed): boolean {
  if (x === y) {
    return x !== 0 || y !== 0 || 1 / x === 1 / y;
  } else {
    return x !== x && y !== y;
  }
}

function shallowEqual(objA: mixed, objB: mixed): boolean {
  // 首先对基本类型进行比较
  if (is(objA, objB)) {
    return true;
  }

  if (typeof objA !== 'object' || objA === null ||
      typeof objB !== 'object' || objB === null) {
    return false;
  }

  const keysA = Object.keys(objA);
  const keysB = Object.keys(objB);

  // 长度不相等直接返回false
  if (keysA.length !== keysB.length) {
    return false;
  }

  // key相等的情况下，再去循环比较
  for (let i = 0; i < keysA.length; i++) {
    if (
      !hasOwnProperty.call(objB, keysA[i]) ||
      !is(objA[keysA[i]], objB[keysA[i]])
    ) {
      return false;
    }
  }

  return true;
```