React面试题

7. 解读setState的同步与异步。
8. 函数组件与class组件如何选择?
9. React性能优化方案

资源

DebugReact

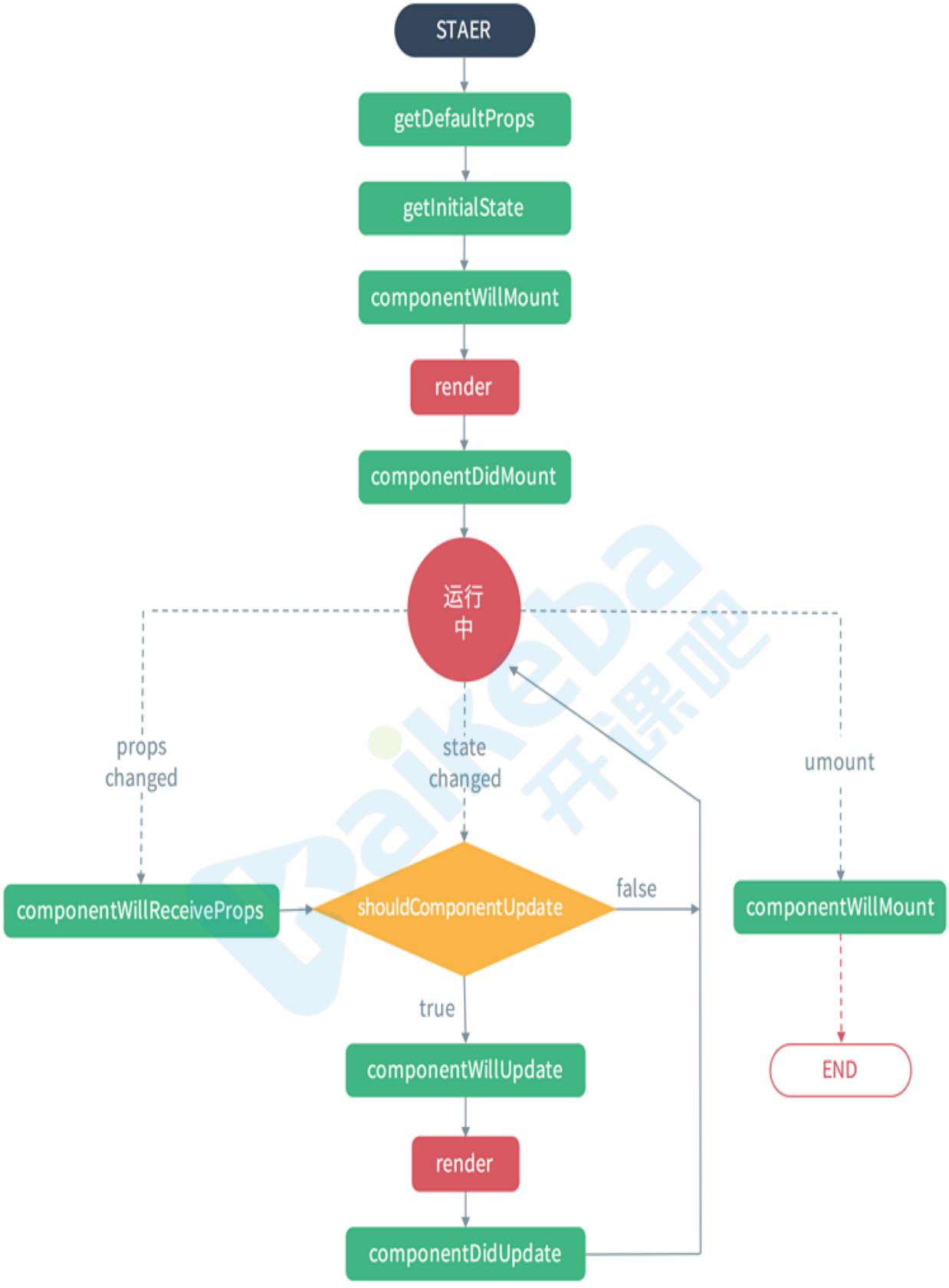
常见面试题

1. 生命周期方法

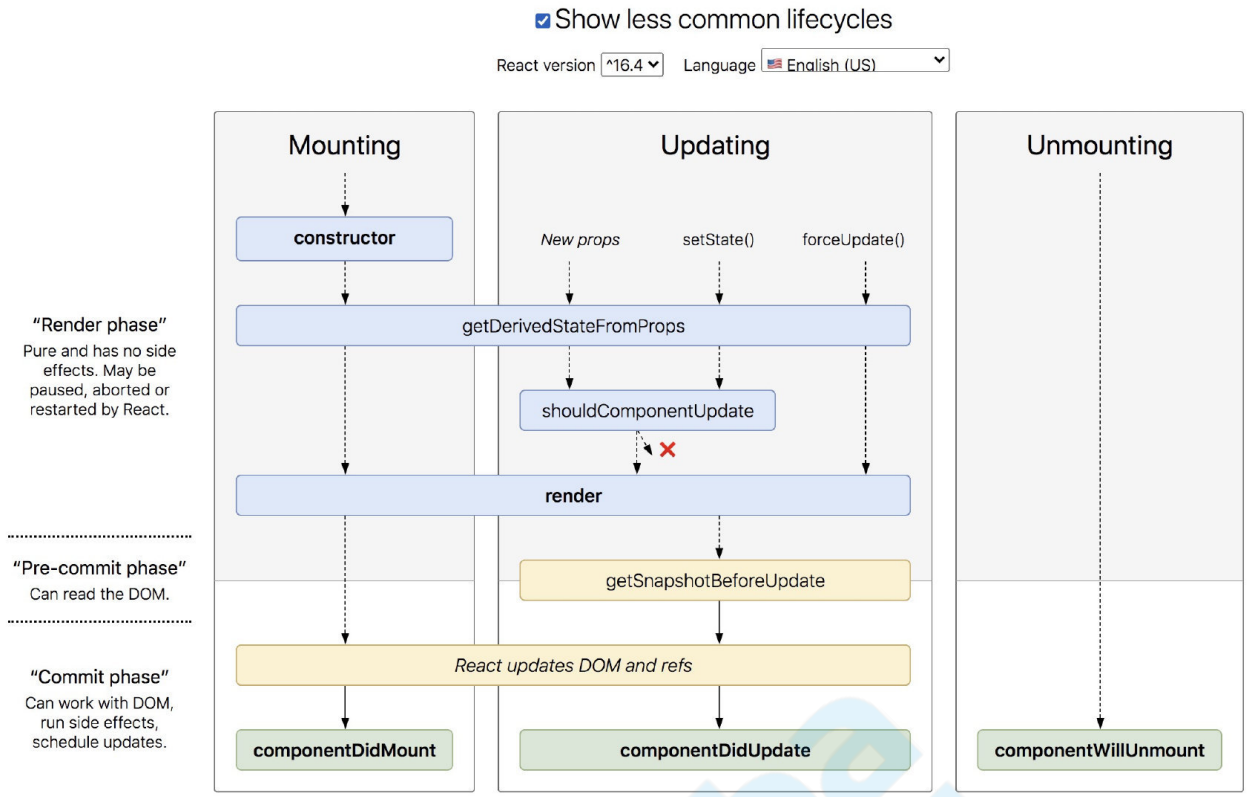
生命周期方法（新老）有哪些，执行过程是怎样的？为什么发生改变？新生命周期的执行场景?

生命周期方法，用于在组件不同阶段执行自定义功能。在组件被创建并插入到 DOM 时（即挂载中阶段（mounting）），组件更新时，组件取消挂载或从 DOM 中删除时，都有可以使用的生命周期方法。

React V16.3之前



V16.4之后的生命周期：



V17可能会废弃的三个生命周期函数用getDerivedStateFromProps替代，目前使用的话加上UNSAFE_：

- componentWillMount
- componentWillReceiveProps
- componentWillUpdate

引入两个新的生命周期函数：

- static getDerivedStateFromProps
- getSnapshotBeforeUpdate

如果不想手动给将要废弃的生命周期添加 `UNSAFE_` 前缀，可以用下面的命令。

```
npx react-codemod rename-unsafe-lifecycles <path>
```

新引入的两个生命周期函数

getDerivedStateFromProps

```
static getDerivedStateFromProps(props, state)
```

`getDerivedStateFromProps` 会在调用 render 方法之前调用，并且在初始挂载及后续更新时都会被调用。它应返回一个对象来更新 state，如果返回 null 则不更新任何内容。

请注意，不管原因是什么，都会在*每次*渲染前触发此方法。这与 `UNSAFE_componentWillReceiveProps` 形成对比，后者仅在父组件重新渲染时触发，而不是在内部调用 `setState` 时。

getSnapshotBeforeUpdate

```
getSnapshotBeforeUpdate(prevProps, prevState)
```

在render之后，在componentDidUpdate之前。

`getSnapshotBeforeUpdate()` 在最近一次渲染输出（提交到 DOM 节点）之前调用。它使得组件能在发生更改之前从 DOM 中捕获一些信息（例如，滚动位置）。此生命周期的任何返回值将作为参数传递给 `componentDidUpdate(prevProps, prevState, snapshot)`。

2. React Fiber到底是什么?

为什么需要fiber

[React Conf 2017 Fiber介绍视频]

React的killer feature： virtual dom

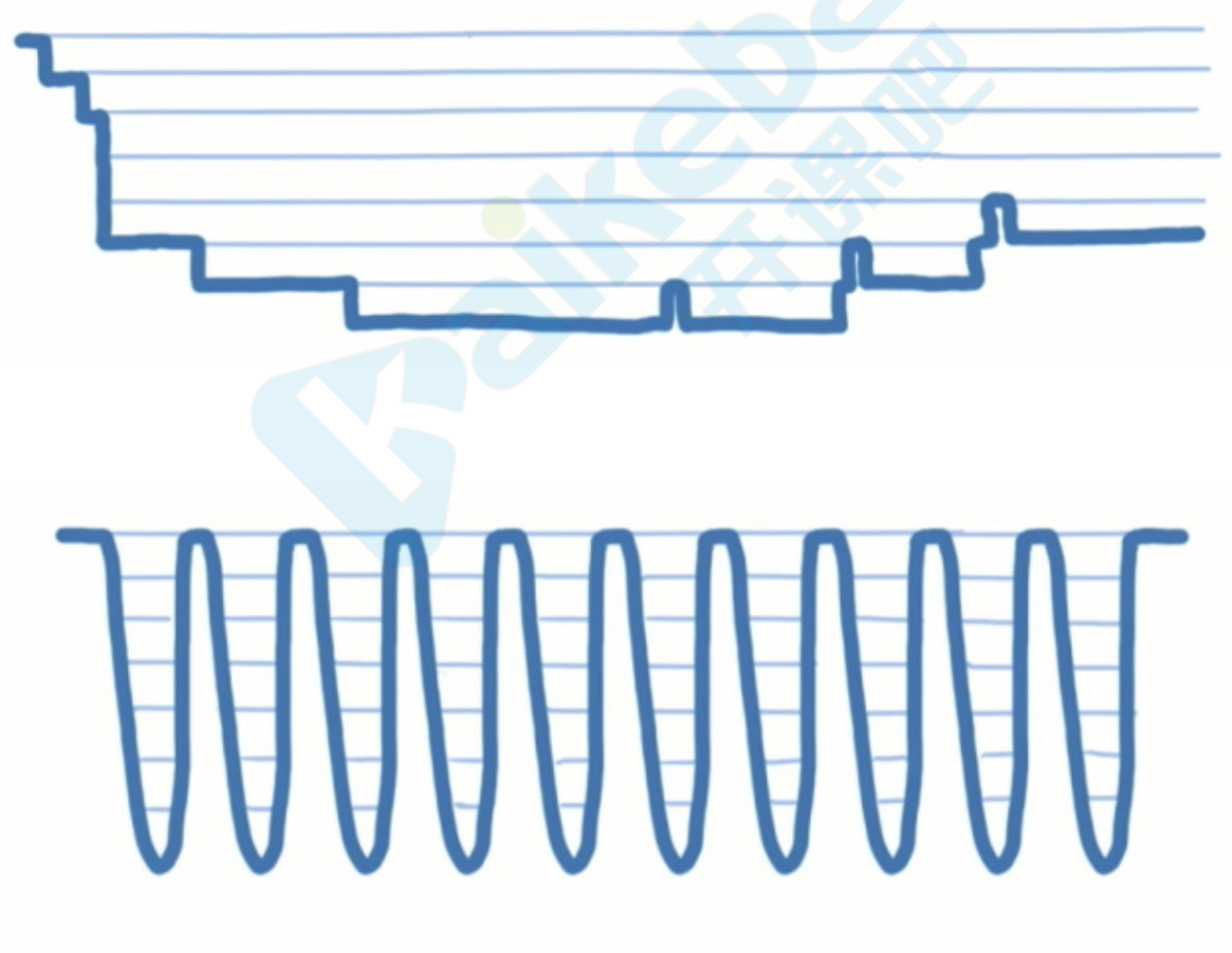
1. 为什么需要fiber

对于大型项目，组件树会很大，这个时候递归遍历的成本就会很高，会造成主线程被持续占用，结果就是主线程上的布局、动画等周期性任务就无法立即得到处理，造成视觉上的卡顿，影响用户体验。

2. 任务分解的意义

   解决上面的问题

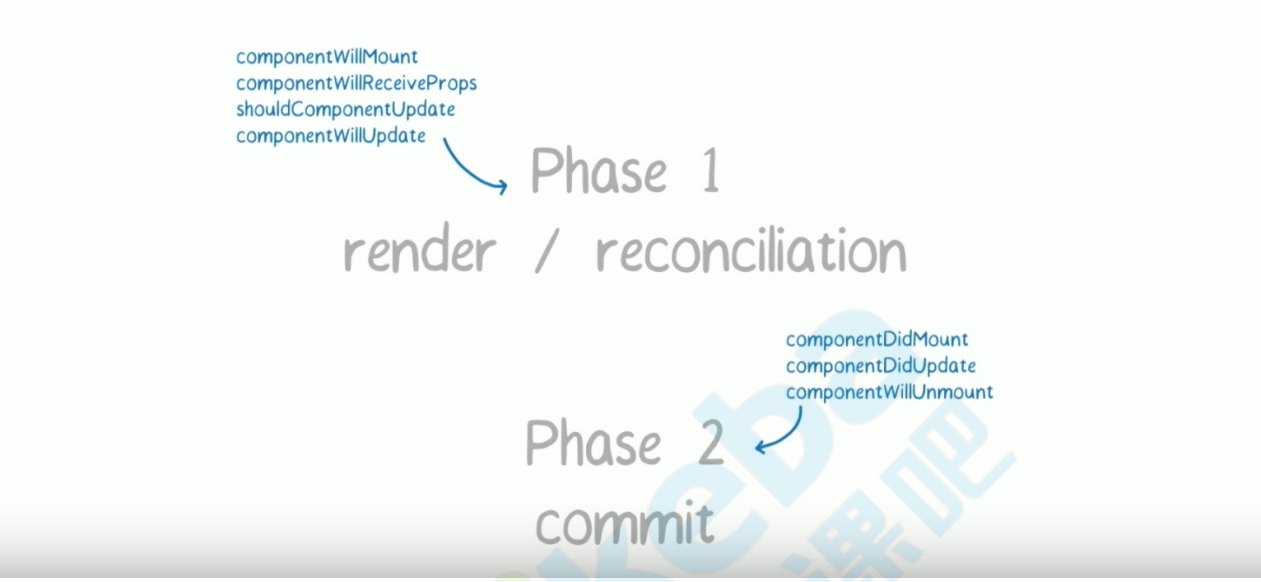
3. 增量渲染（把渲染任务拆分成块，匀到多帧）
4. 更新时能够暂停，终止，复用渲染任务
5. 给不同类型的更新赋予**优先级**
6. 并发方面新的基础能力
7. **更流畅**



什么是fiber

A Fiber is work on a Component that needs to be done or was done. There can be more than one per component.

fiber是指组件上将要完成或者已经完成的任务，每个组件可以一个或者多个。



fiber链表结构

3. 什么是React Hook，它带来了什么。

Hook简介

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。

1. Hooks是什么？为了拥抱~~王能量~~函数式
2. Hooks带来的变革，让函数组件有了状态，可以替代class，可以实现React class组件可以有的特性，如执行副作用等等。

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

function Example() {
  // 声明一个新的叫做 “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>
  );
}
```

Hook解决了什么问题

Hook 解决了我们五年来编写和维护成千上万的组件时遇到的各种各样看起来不相关的问题。无论你正在学习 React，或每天使用，或者更愿尝试另一个和 React 有相似组件模型的框架，你都可能对这些问题似曾相识。

1). 在组件之间复用状态逻辑很难

React 没有提供将可复用性行为“附加”到组件的途径（例如，把组件连接到 store）。如果你使用过 React 一段时间，你也许会熟悉一些解决此类问题的方案，比如 render props 和 高阶组件。但是这类方案需要重新组织你的组件结构，这可能会很麻烦，使你的代码难以理解。如果你在 React DevTools 中观察过 React 应用，你会发现由 providers，consumers，高阶组件，render props 等其他抽象层组成的组件会形成“嵌套地狱”。尽管我们可以在 DevTools 过滤掉它们，但这说明了一个更深层次的问题：React 需要为共享状态逻辑提供更好的原生途径。

你可以使用 Hook 从组件中提取状态逻辑，使得这些逻辑可以单独测试并复用。**Hook 使你在无需修改组件结构的情况下复用状态逻辑。** 这使得在组件间或社区内共享 Hook 变得更便捷。

2). 复杂组件变得难以理解

我们经常维护一些组件，组件起初很简单，但是逐渐会被状态逻辑和副作用充斥。每个生命周期常常包含一些不相关的逻辑。例如，组件常常在 `componentDidMount` 和 `componentDidUpdate` 中获取数据。但是，同一个 `componentDidMount` 中可能也包含很多其它的逻辑，如设置事件监听，而之后需在 `componentWillUnmount` 中清除。相互关联且需要对照修改的代码被进行了拆分，而完全不相关的代码却在同一个方法中组合在一起。如此很容易产生 bug，并且导致逻辑不一致。

在多数情况下，不可能将组件拆分为更小的粒度，因为状态逻辑无处不在。这也给测试带来了一定挑战。同时，这也是很多人将 React 与状态管理库结合使用的原因之一。但是，这往往会引入了很多抽象概念，需要你在不同的文件之间来回切换，使得复用变得更加困难。

为了解决这个问题，**Hook 将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据）**，而并非强制按照生命周期划分。你还可以使用 reducer 来管理组件的内部状态，使其更加可预测。

3). 难以理解的 class

除了代码复用和代码管理会遇到困难外，我们还发现 class 是学习 React 的一大屏障。你必须去理解 JavaScript 中 `this` 的工作方式，这与其他语言存在巨大差异。还不能忘记绑定事件处理器。没有稳定的语法提案，这些代码非常冗余。大家可以很好地理解 props，state 和自顶向下的数据流，但对 class 却一筹莫展。即便在有经验的 React 开发者之间，对于函数组件与 class 组件的差异也存在分歧，甚至还要区分两种组件的使用场景。

另外，React 已经发布五年了，我们希望它能在下一个五年也与时俱进。就像 Svelte，Angular，Glimmer等其它的库展示的那样，组件预编译会带来巨大的潜力。尤其是在它不局限于模板的时候。最近，我们一直在使用 Prepack 来试验 component folding，也取得了初步成效。但是我们发现使用 class 组件会无意中鼓励开发者使用一些让优化措施无效的方案。class 也给目前的工具带来了一些问题。例如，class 不能很好的压缩，并且会使热重载出现不稳定的情况。因此，我们想提供一个使代码更易于优化的 API。

为了解决这些问题，**Hook 使你在非 class 的情况下可以使用更多的 React 特性。** 从概念上讲，React 组件一直更像是函数。而 Hook 则拥抱了函数，同时也没有牺牲 React 的精神原则。Hook 提供了问题的解决方案，无需学习复杂的函数式或响应式编程技术。

Hook API

- 基础 Hook
  - `useState`
  - `useEffect`
  - `useContext`
- 额外的 Hook
  - `useReducer`

    在某些场景下，`useReducer` 会比 `useState` 更适用，例如 state 逻辑较复杂且包含多个子值，或者下一个 state 依赖于之前的 state 等。并且，使用 `useReducer` 还能给那些会触发深更新的组件做性能优化，因为你可以向子组件传递 `dispatch` 而不是回调函数 。
  - `useCallback`

    返回一个 memoized 回调函数。

    把内联回调函数及依赖项数组作为参数传入 `useCallback`，它将返回该回调函数的 memoized 版本，该回调函数仅在某个依赖项改变时才会更新。

- `useMemo`

  返回一个 memoized 值。

  把“创建”函数和依赖项数组作为参数传入 `useMemo`，它仅会在某个依赖项改变时才重新计算 memoized 值。这种优化有助于避免在每次渲染时都进行高开销的计算。

- `useRef`

  `useRef` 返回一个可变的 ref 对象，其 `.current` 属性被初始化为传入的参数（`initialValue`）。返回的 ref 对象在组件的整个生命周期内保持不变。

- `useImperativeHandle`

  `useImperativeHandle` 可以让你在使用 `ref` 时自定义暴露给父组件的实例值。在在大多数情况下，应当避免使用 ref 这样的命令式代码。`useImperativeHandle` 应当与 `forwardRef` 一起使用。

- `useLayoutEffect`

  函数签名与 `useEffect` 相同，但它会在所有的 DOM 变更之后同步调用 effect。

- `useDebugValue`

  `useDebugValue` 可用于在 React 开发者工具中显示自定义 hook 的标签。

4. 比较state与props

state是一种数据结构，在当前组件有效，使用setState进行更改，class组件中会引发组件渲染。

函数组件中可以用useState、useReducer。

与class组件中的setState不同的是，如果前后两次的值相同，函数组件中的useState和useReducer Hook会放弃更新，原地修改setState不会引起重新渲染。

通常，你不应该在 React 中修改本地 state。然而，作为一条出路，你可以用一个增长的计数器来在 state 没变的时候依然强制一次重新渲染：

```
const [ignored, forceUpdate] = useReducer(x => x + 1, 0);

function handleClick() {
  forceUpdate();
}
```

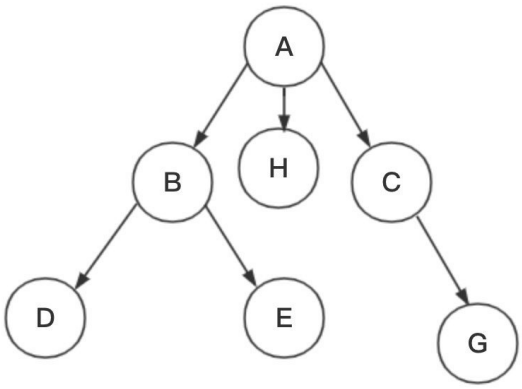
这也是function组件中类似class组件中的forceUpdate。

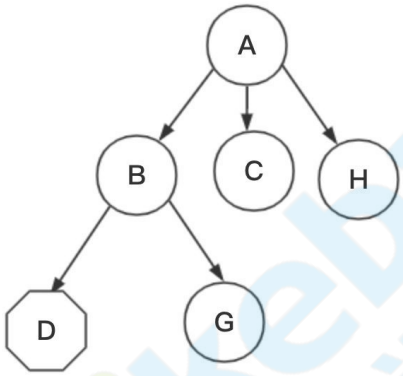
而props（属性“properties” 的缩写）是组件的属性值，由父组件传递过来，就组件自身来说，props是不可变的，组件不能改变自己的props，但是可以把子组件的props放在一起统一管理，props可以是多种数据类型，如对象、函数等。

`props`和 class `state` 都是普通的 JavaScript 对象。它们都是用来保存信息的，这些信息可以控制组件的渲染输出，而它们的一个重要的不同点就是：`props` 是*传递给*组件的（类似于函数的形参），而 `state` 是在组件*内*被组件自己管理的（类似于在一个函数内声明的变量）。

5. React diff

算法复杂度？怎么做到O(n)的（回答diff策略）？diff流程？优缺点？与Vue diff比较？





在某一时间节点调用React的render方法，会创建一棵由React元素组成的树，在下一次state或props更新时，相同的render方法会返回一棵不同的树。React需要基于这两棵树之间的差别来判断如何有效率的更新UI以保证当前UI与最新的树保持同步。

这个问题有一些通用的解决方案，即生成一棵树转换成另一棵树的最小操作数。然而，即使在最前沿的算法中，该算法的复杂程度为O(n 3)，其中 n 是树中元素的数量。

如果在 React 中使用了该算法，那么展示 1000 个元素所需要执行的计算量将在十亿的量级范围。这个开销实在是太过高昂。于是 React 在以下两个假设的基础之上提出了一套 O(n) 的启发式算法：

1. 两个不同类型的元素会产生出不同的树；
2. 开发者可以通过 `key` prop 来暗示哪些子元素在不同的渲染下能保持稳定；

   在实践中，我们发现以上假设在几乎所有实用的场景下都成立。

diffing算法

算法复杂度O(n)

diff 策略

1. 同级比较，Web UI 中 DOM 节点跨层级的移动操作特别少，可以忽略不计。
2. 拥有不同类型的两个组件将会生成不同的树形结构。

   例如：div->p, CompA->CompB
3. 开发者可以通过 `key` prop 来暗示哪些子元素在不同的渲染下能保持稳定；

1. React diff 实现

当对比两颗树时，React 首先比较两棵树的根节点，不同类型的根节点元素会有不同的行为。

1）比对不同类型的元素

当根节点为不同类型的元素时，React会卸载老树并创建新树。举个例子，从变成，从 `<Article>` 变成 `<Comment>`，或者从 `<Button>` 变成 `<div>`，这些都会触发一个完整的重建流程。

卸载老树的时候，老的DOM节点也会被销毁，组件实例会执行 componentWillUnmount。创建新树的时候，也会有新的DOM节点插入DOM，这个组件实例会执行 `componentWillMount()` 和 `componentDidMount()`。当然，老树相关的state也被消除。

2） 比对同类型的DOM元素

当对比同类型的DOM元素时候，React会比对新旧元素的属性，同时保留老的，只去更新改变的属性。

处理完DOM节点之后，React然后会去递归遍历子节点。

3）比对同类型的组件元素

这个时候，React更新该组件实例的props，调用`componentWillReceiveProps()`和`componentWillUpdate()`。下一步，render被调用，diff算法递归遍历新老树。

4） 对子节点进行递归

当递归DOM节点的子元素时，React会同时遍历两个子元素的列表。

下面是遍历子节点的源码，解析这段源码得出以下思路：

- 首先判断当前节点是否是没有key值的顶层fragment元素，如果是的话，需要遍历的newChild就是newChild.props.children元素。
- 判断newChild的类型，如果是object，并且$$typeof是REACT_ELEMENT_TYPE，那么证明这是一个单个的HTML标签元素，则首先执行reconcileSingleElement函数，返回协调之后得到的fiber，placeSingleChild函数则把这个fiber放到指定位置上。
- REACT_PORTAL_TYPE同上一条。
- 如果newChild是string或者number，即文本，则执行reconcileSingleTextNode函数，返回协调之后得到的fiber，依然是placeSingleChild把这个fiber放到指定的位置上。
- 如果是newChild数组，则执行reconcileChildrenArray对数组进行协调。

```
function reconcileChildFibers(
    returnFiber: Fiber,
    currentFirstChild: Fiber | null,
    newChild: any,
    expirationTime: ExpirationTime,
  ): Fiber | null {

    const isUnkeyedTopLevelFragment =
      typeof newChild === 'object' &&
      newChild !== null &&
```

```
      newChild.type === REACT_FRAGMENT_TYPE &&
      newChild.key === null;
    if (isUnkeyedTopLevelFragment) {
      newChild = newChild.props.children;
    }

    // Handle object types
    const isObject = typeof newChild === 'object' &&
newChild !== null;

    if (isObject) {
      switch (newChild.$$typeof) {
        case REACT_ELEMENT_TYPE:
          return placeSingleChild(
            reconcileSingleElement(
              returnFiber,
              currentFirstChild,
              newChild,
              expirationTime,
            ),
          );
        case REACT_PORTAL_TYPE:
          return placeSingleChild(
            reconcileSinglePortal(
              returnFiber,
              currentFirstChild,
              newChild,
              expirationTime,
            ),
          );
      }
    }

    if (typeof newChild === 'string' || typeof newChild ===
'number') {
      return placeSingleChild(
        reconcileSingleTextNode(
```

```
            returnFiber,
            currentFirstChild,
            '' + newChild,
            expirationTime,
          ),
        );
      }

      if (isArray(newChild)) {
        return reconcileChildrenArray(
          returnFiber,
          currentFirstChild,
          newChild,
          expirationTime,
        );
      }

      if (isObject) {
        throwOnInvalidObjectType(returnFiber, newChild);
      }

      // Remaining cases are all treated as empty.
      return deleteRemainingChildren(returnFiber,
currentFirstChild);
    }
```

2. React diff的优缺点

React由于设定的diff前提，达到了O(n)的算法复杂度，在目前所有diff中算的上是优秀的了，再加上fiber的架构，做到了增量渲染，防止了掉帧，这绝对是前端中的一大革新，其思想值得前端工程师深入学习。

但是也有一些目前没有解决的缺点。比如说由于React的fiber的单链表结构性质，导致React diff查找的时候不能从两端开始查找，这在时间复杂度上是非常值得优化的一个点，React目前也在寻求这种优化，也让我们拭目等待React的进一步优化。

3. React中diff与vue diff的对比

React本身架构与Vue不同，Vue中没有fiber，目前也用不到fiber。而且React diff算法的实现是基本自身架构的，React实现不了双向查找，而vue可以。

相同点是都是深度优先、同层级比较，都借助key，都有批量新增和删除等等。

6. React合成事件机制。

React中有自己的事件系统模式，通常被称为React合成事件。之所以采用这种自己定义的合成事件，一方面是为了抹平事件在不同平台体现出来的差异性，这使得React开发者不需要自己再去关注浏览器事件兼容性问题，另一方面是为了统一管理事件，提高性能，这主要体现在React内部实现事件委托，并且记录当前事件发生的状态上。

事件委托，也就是我们通常提到的事件代理机制，这种机制不会把事件处理函数直接绑定在真实的节点上，而是把所有的事件绑定到结构的最外层，使用一个统一的事件监听和处理函数。当组件挂载或卸载时，只是在这个统一的事件监听器上插入或删除一些对象；当事件放生时，首先被这个统一的事件监听器处理，然后在映射表里找到真正的事件处理函数并调用。这样做简化了事件处理和回收机制，效率也有很大提升。

记录当前事件发生的状态，即记录事件执行的上下文，这便于React来处理不同事件的优先级，达到谁优先级高先处理谁的目的，这里也就实现了React的增量渲染思想，可以预防掉帧，同时达到页面更顺滑的目的，提升用户体验。

ExecutionContext

React执行栈（React execution stack）中，所处于几种环境的值，所对应的的全局变量是react-reconciler/src/ReactFiberWorkLoop.js文件中的`executionContext`。

```
type ExecutionContext = number;

const NoContext = /*                      */ 0b000000;
const BatchedContext = /*                 */ 0b000001;
const EventContext = /*                   */ 0b000010;
const DiscreteEventContext = /*           */ 0b000100;
const LegacyUnbatchedContext = /*         */ 0b001000;
const RenderContext = /*                  */ 0b010000;
const CommitContext = /*                  */ 0b100000;
```

7. 解读setState的同步与异步。

React中理论上有三种模式可选，默认的legacy 模式、blocking模式和concurrent模式，legacy 模式在合成事件中有自动批处理的功能，但仅限于一个浏览器任务。非 React 事件想使用这个功能必须使用`unstable_batchedUpdates`。在 blocking 模式和 concurrent 模式下，所有的`setState`在默认情况下都是批处理的，但是这两种模式目前仅实验版本可用。

在目前的版本中，事件处理函数内部的setState是异步的，即批量执行，这样是为了避免子组件被多次渲染，这种机制可以在大型应用中得到很好的性能提升。但是React官网也提到这只是一个实现的细节，所以请不要直接依赖于这种机制。在以后的版本当中，React 会在更多的情况下默认地使用 state 的批更新机制。

8. 函数组件与class组件如何选择?

1. hook之前的函数组件是什么样子?

无状态，无副作用，只能做单纯的展示组件。

2. class组件有什么弊端，为什么要引入hook?
    - 在组件之间复用状态逻辑很难
    - 复杂组件变得难以理解
    - 难以理解的 class
3. 引入了hook之后的函数组件发生了哪些变化?

    函数组件可以存储和改变状态值（useState、useReducer），可以执行副作用（useEffect、useLayoutEffect），还可以复用状态逻辑（自定义hook）。

4. 函数组件与class组件如何选择?

    出现以上缺点的情况下都适合使用函数组件。

9. React性能优化方案

1. 减少不必要渲染，如用shouldComponentUpdate、PureComponent、React.memo实现。

```
function updateSimpleMemoComponent(
  current: Fiber | null,
  workInProgress: Fiber,
  Component: any,
  nextProps: any,
  updateExpirationTime,
  renderExpirationTime: ExpirationTime,
): null | Fiber {
  if (current !== null) {
    const prevProps = current.memoizedProps;
    if (
      shallowEqual(prevProps, nextProps) &&
      current.ref === workInProgress.ref &&
    ) {
      didReceiveUpdate = false;
```

```
      if (updateExpirationTime < renderExpirationTime) {

        workInProgress.expirationTime =
current.expirationTime;
        return bailoutOnAlreadyFinishedWork(
          current,
          workInProgress,
          renderExpirationTime,
        );
      }
    }
  }
  return updateFunctionComponent(
    current,
    workInProgress,
    Component,
    nextProps,
    renderExpirationTime,
  );
}
```

2. 数据缓存。
   - useMemo缓存参数、useCallback缓存函数。

```
function updateMemo<T>(
  nextCreate: () => T,
  deps: Array<mixed> | void | null,
): T {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  const prevState = hook.memoizedState;
  if (prevState !== null) {
    if (nextDeps !== null) {
      const prevDeps: Array<mixed> | null =
prevState[1];
```

```
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        return prevState[0];
      }
    }
  }
  const nextValue = nextCreate();
  hook.memoizedState = [nextValue, nextDeps];
  return nextValue;
}
```

```
function updateCallback<T>(callback: T, deps:
Array<mixed> | void | null): T {
  const hook = updateWorkInProgressHook();
  const nextDeps = deps === undefined ? null : deps;
  const prevState = hook.memoizedState;
  if (prevState !== null) {
    if (nextDeps !== null) {
      const prevDeps: Array<mixed> | null =
prevState[1];
      if (areHookInputsEqual(nextDeps, prevDeps)) {
        return prevState[0];
      }
    }
  }
  hook.memoizedState = [callback, nextDeps];
  return callback;
}
```

- 函数、对象尽量不要使用内联形式（如context的value object、refs function）。
- Router中的内联函数渲染时候使用render或者children，不要使用component。

当你用`component`的时候，Router会用你指定的组件和React.createElement创建一个新的[React element]。这意味着当你提供的是一个内联函数的时候，每次render都会创建一个新的组件。这会导致不再更新已经现有组件，而是直接卸载然后再去挂载一个新的组件。因此，当用到内联函数的内联渲染时，请使用render或者children。

3. 不要滥用功能项，如context、props等。
4. 懒加载，对于长页列表分页加载。
5. 减少http请求。

总结：减少计算、渲染和请求。