React原理解析03

资源

1. React Hook简介
2. React源码

课堂目标

1. 掌握useEffect、useLayoutEffect
2. 掌握Hook
3. 掌握React中的协调

知识点

Hook

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~~正能量~~函数式
2. Hooks带来的变革，让函数组件有了状态和其他的React特性，可以替代class

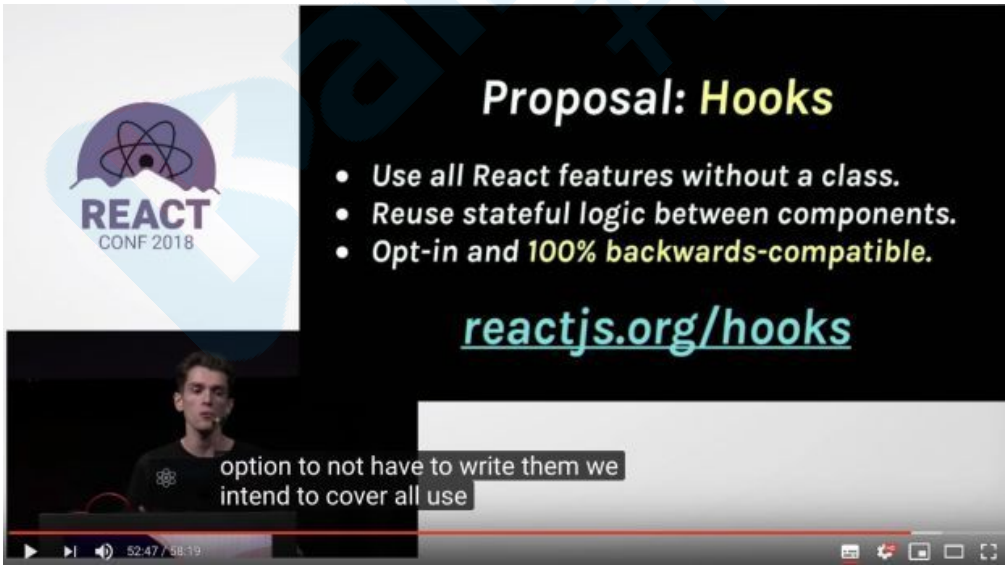
```
import React, { useState } from 'react';

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 {a}
      </button>
    </div>
  );
}
```

视频介绍

在 React Conf 2018 上，Sophie Alpert 和 Dan Abramov 介绍了 Hook，紧接着 Ryan Florence 演示了如何使用 Hook 重构应用。你可以在这里看到这个视频： https://www.youtube.com/embed/dpw9EHDh2bM



没有破坏性改动

在我们继续之前，请记住 Hook 是：

- **完全可选的。** 你无需重写任何已有代码就可以在一些组件中尝试 Hook。但是如果你不想，你不必现在就去学习或使用 Hook。
- **100% 向后兼容的。** Hook 不包含任何破坏性改动。
- **现在可用。** Hook 已发布于 v16.8.0。

没有计划从 React 中移除 class。

Hook 不会影响你对 React 概念的理解。 恰恰相反，Hook 为已知的 React 概念提供了更直接的 API：props， state，context，refs 以及生命周期。稍后我们将看到，Hook 还提供了一种更强大的方式来组合他们。

Hook解决了什么问题

Hook 解决了我们那些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。

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。

具体将在自定义 Hook 中对此展开更多讨论。

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。

我们将在使用 Effect Hook 中对此展开更多讨论。

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`
  - `useCallback`
  - `useMemo`
  - `useRef`
  - `useImperativeHandle`
  - `useLayoutEffect`

实现useEffect与useLayoutEffect

```
// import React, {useReducer, useEffect, useLayoutEffect} from "react";
// import ReactDOM from "react-dom";
import {useReducer, useEffect, useLayoutEffect} from "./kreact/";
import ReactDOM from "./kreact/react-dom";
import "./index.css";

function FunctionComponent(props) {
  // *
  // old fiber.memoizedState（hook0）-> next(hook1) -> next(hook2)-
>hook3(workInProgressHook)
  // workInProgressHook 全局变量
  //hook3
  // new fiber
  //*
  const [count1, setCount1] = useReducer((x) => x + 2, 0);
  const [count2, setCount2] = useReducer((x) => x + 1, 0); //2

  useEffect(() => {
    console.log("omg useEffect", count2); //sy-log
  }, [count2]);

  useLayoutEffect(() => {
    console.log("omg useLayoutEffect", count2); //sy-log
  }, [count2]);

  return (
    <div className="border">
      <p>{props.name}</p>
      <p>{count1}</p>
      <button
        onClick={() => {
          setCount1();
        }}>
        click
      </button>

      <p>{count2}</p>
      <button
        onClick={() => {
          setCount2();
        }}>
        click
      </button>

      {/* {count2 % 2 ? <div>123</div> : <span>456</span>} */}

      <ul>
        <li key="0">0</li>
        <li key="1">1</li>
        {count2 % 2 ? <li key="2">2</li> : null}
        <li key="3">3</li>
        <li key="4">4</li>
      </ul>
    </div>
  );
```

```
}

const jsx = (
  <div className="border">
    <h1>全栈</h1>
    <a href="https://www.kaikeba.com/">kkb</a>
    <FunctionComponent name="函数" />
  </div>
);

ReactDOM.render(jsx, document.getElementById("root"));
```

hooks.js

```
import {scheduleUpdateOnFiber} from "./ReactFiberWorkLoop";
import {areHookInputsEqual, HookPassive, HookLayout} from "./utils";

// fiber.memoizedState 第0个hook
// hook = {
// memoizedState : 状态值
// next // 下一个hook

// 当前正在工作fiber
let currentlyRenderingFiber = null;
// 当前正在工作的hook
let workInProgressHook = null;

let currentHook = null;

export function renderHooks(fiber) {
  currentlyRenderingFiber = fiber;
  currentlyRenderingFiber.memoizedState = null; // hook0
  currentlyRenderingFiber.updateQueueOfEffect = [];
  currentlyRenderingFiber.updateQueueOfLayout = [];
  workInProgressHook = null;
}

function updateWorkInProgressHook() {
  let hook = null;

  // todo get hook
  // 老节点
  let current = currentlyRenderingFiber.alternate;
  if (current) {
    // 更新阶段 新的hook在老的hook基础上更新
    currentlyRenderingFiber.memoizedState = current.memoizedState;
    if (workInProgressHook) {
      // 不是第0个hook
      hook = workInProgressHook = workInProgressHook.next;
      currentHook = currentHook.next;
    } else {
      // 是第0个hook
      hook = workInProgressHook = current.memoizedState;
      currentHook = current.memoizedState;
    }
  } else {
    // 初次渲染阶段
```

```
    currentHook = null;
    hook = {
      memoizedState: null, // 状态值
      next: null, // 下一个hook
    };
    if (workInProgressHook) {
      // 不是第0个hook
      workInProgressHook = workInProgressHook.next = hook;
    } else {
      // 是第0个hook
      workInProgressHook = currentlyRenderingFiber.memoizedState = hook;
    }
  }

  return hook;
}

export function useReducer(reducer, initalState) {
  const hook = updateWorkInProgressHook();

  if (!currentlyRenderingFiber.alternate) {
    // 初次渲染
    hook.memoizedState = initalState;
  }

  const dispatch = (action) => {
    hook.memoizedState = reducer(hook.memoizedState, action);
    scheduleUpdateOnFiber(currentlyRenderingFiber);
  };

  return [hook.memoizedState, dispatch];
}

export function useEffect(create, deps) {
  updateEffectIml(HookPassive, create, deps);
}

export function useLayoutEffect(create, deps) {
  updateEffectIml(HookLayout, create, deps);
}

export function updateEffectIml(hookFlags, create, deps) {
  const hook = updateWorkInProgressHook();

  if (currentHook) {
    // deps 是否发生变化
    const prevEffect = currentHook.memoizedState;
    if (deps) {
      const prevDeps = prevEffect.deps;
      if (areHookInputsEqual(deps, prevDeps)) {
        return;
      }
    }
  }

  const effect = {hookFlags, create, deps};
  hook.memoizedState = effect;
  if (hookFlags & HookPassive) {
```

```
    currentlyRenderingFiber.updateQueueOfEffect.push(effect);
  } else if (hookFlags & HookLayout) {
    currentlyRenderingFiber.updateQueueOfLayout.push(effect);
  }
}
```

ReactFiberWorkLoop.js简版

```
//根节点

import {
  updateFragmentComponent,
  updateFunctionComponent,
  updateHostComponent,
} from "./ReactFiberReconciler";
import {scheduleCallback, shouldYield} from "./scheduler";
import {isStringOrNumber, isFn, Placement, Update, updateNode} from "./utils";

// wip work in progress 当前正在工作当中的
let wipRoot = null;
// 将要更新的下一个fiber节点;
let nextUnitOfWork = null;

// 处理更新
export function scheduleUpdateOnFiber(fiber) {
  fiber.alternate = {...fiber};
  wipRoot = fiber;
  wipRoot.sibling = null;

  nextUnitOfWork = wipRoot;

  scheduleCallback(workLoop);
}

// 协调
function performUnitOfWork(wip) {
  // * 1. 更新自己
  // todo 原生标签 函数组件 类组件等
  const {type} = wip;
  if (isFn(type)) {
    updateFunctionComponent(wip);
  } else if (isStringOrNumber(type)) {
    // 原生标签
    updateHostComponent(wip);
  } else {
    // fragment
    updateFragmentComponent(wip);
  }

  // * 2. 返回下一个要更新的fiber
  // 深度优先遍历
  // 王朝的故事
  if (wip.child) {
    return wip.child;
  }
```

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

function workLoop() {
  while (nextUnitOfWork && !shouldYield()) {
    nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
  }

  //提交
  if (!nextUnitOfWork && wipRoot) {
    commitRoot();
  }
}

// requestIdleCallback(workLoop);

// 提交
function commitRoot() {
  isFn(wipRoot.type) ? commitWorker(wipRoot) : commitWorker(wipRoot.child);
}

function invokesHooks(wip) {
  const {updateQueueOfEffect, updateQueueOfLayout} = wip;
  for (let i = 0; i < updateQueueOfLayout.length; i++) {
    const effect = updateQueueOfLayout[i];
    effect.create();
  }

  for (let i = 0; i < updateQueueOfEffect.length; i++) {
    const effect = updateQueueOfEffect[i];
    scheduleCallback(() => {
      effect.create();
    });
  }
}
// 找父dom节点
function getParentNode(fiber) {
  let next = fiber.return;
  while (!next.stateNode) {
    next = next.return;
  }

  return next.stateNode;
}
function commitWorker(fiber) {
  if (!fiber) {
    return;
  }
  const {type, flags, stateNode} = fiber;
```

```
  if (isFn(type)) {
    invokesHooks(fiber);
  }

  // 父dom节点
  // 所有fiber都有dom节点吗 0
  let parentNode = getParentNode(fiber); //fiber.return.stateNode;
  // 1.提交自己

  // 插入
  if (flags & Placement && stateNode) {
    parentNode.appendChild(stateNode);
  }

  // 更新属性
  if (flags & Update && stateNode) {
    updateNode(stateNode, fiber.alternate.props, fiber.props);
  }

  if (fiber.deletions) {
    commitDeletions(fiber.deletions, stateNode || parentNode);
    fiber.deletions = null;
  }
  // 2.提交孩子
  commitWorker(fiber.child);
  // 3.提交下一个兄弟
  commitWorker(fiber.sibling);
}

function commitDeletions(deletions, parentNode) {
  for (let i = 0; i < deletions.length; i++) {
    const deletion = deletions[i];
    parentNode.removeChild(getStateNode(deletion));
  }
}

function getStateNode(fiber) {
  let tem = fiber;
  while (!tem.stateNode) {
    tem = tem.child;
  }
  return tem.stateNode;
}
```

ReactFiberWorkLoop.js完整版

```
//根节点

import {
  updateFragmentComponent,
  updateFunctionComponent,
  updateHostComponent,
} from "./ReactFiberReconciler";
import {scheduleCallback, shouldYield} from "./scheduler";
import {isStringOrNumber, isFn, Placement, Update, updateNode} from "./utils";
```

```
// wip work in progress 当前正在工作当中的
let wipRoot = null;
// 将要更新的下一个fiber节点;
let nextUnitOfWork = null;

// 处理更新
export function scheduleUpdateOnFiber(fiber) {
  fiber.alternate = {...fiber};
  wipRoot = fiber;
  wipRoot.sibling = null;

  nextUnitOfWork = wipRoot;

  scheduleCallback(workLoop);
}

// 协调
function performUnitOfWork(wip) {
  // * 1. 更新自己
  // todo 原生标签 函数组件 类组件等
  const {type} = wip;
  if (isFn(type)) {
    updateFunctionComponent(wip);
  } else if (isStringOrNumber(type)) {
    // 原生标签
    updateHostComponent(wip);
  } else {
    // fragment
    updateFragmentComponent(wip);
  }

  // * 2. 返回下一个要更新的fiber
  // 深度优先遍历
  // 王朝的故事
  if (wip.child) {
    return wip.child;
  }

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

function workLoop() {
  while (nextUnitOfWork && !shouldYield()) {
    nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
  }

  //提交
  if (!nextUnitOfWork && wipRoot) {
    commitRoot();
  }
```

```
}

// requestIdleCallback(workLoop);

// 提交
function commitRoot() {
  isFn(wipRoot.type) ? commitWorker(wipRoot) : commitWorker(wipRoot.child);
}

function invokesHooks(wip) {
  const {updateQueueOfEffect, updateQueueOfLayout} = wip;
  for (let i = 0; i < updateQueueOfLayout.length; i++) {
    const effect = updateQueueOfLayout[i];
    effect.create();
  }

  for (let i = 0; i < updateQueueOfEffect.length; i++) {
    const effect = updateQueueOfEffect[i];
    scheduleCallback(() => {
      effect.create();
    });
  }
}
// 找父dom节点
function getParentNode(fiber) {
  let next = fiber.return;
  while (!next.stateNode) {
    next = next.return;
  }

  return next.stateNode;
}
function commitWorker(fiber) {
  if (!fiber) {
    return;
  }
  const {type, flags, stateNode} = fiber;

  if (isFn(type)) {
    invokesHooks(fiber);
  }

  // 父dom节点
  // 所有fiber都有dom节点吗 0
  let parentNode = getParentNode(fiber); //fiber.return.stateNode;
  // 1.提交自己

  // 插入
  if (flags & Placement && stateNode) {
    let hasSiblingNode = foundSiblingNode(fiber, parentNode);
    if (hasSiblingNode) {
      parentNode.insertBefore(stateNode, hasSiblingNode);
    } else {
      parentNode.appendChild(fiber.stateNode);
    }
  }

  // 更新属性
```

```
  if (flags & Update && stateNode) {
    updateNode(stateNode, fiber.alternate.props, fiber.props);
  }

  if (fiber.deletions) {
    commitDeletions(fiber.deletions, stateNode || parentNode);
    fiber.deletions = null;
  }
  // 2.提交孩子
  commitWorker(fiber.child);
  // 3.提交下一个兄弟
  commitWorker(fiber.sibling);
}

function foundSiblingNode(fiber, parentNode) {
  let siblingHasNode = fiber.sibling;
  let node = null;
  while (siblingHasNode) {
    node = siblingHasNode.stateNode;
    if (node && parentNode.contains(node)) {
      return node;
    }
    siblingHasNode = siblingHasNode.sibling;
  }

  return null;
}

function commitDeletions(deletions, parentNode) {
  for (let i = 0; i < deletions.length; i++) {
    const deletion = deletions[i];
    parentNode.removeChild(getStateNode(deletion));
  }
}

function getStateNode(fiber) {
  let tem = fiber;
  while (!tem.stateNode) {
    tem = tem.child;
  }
  return tem.stateNode;
}
```

utils.js

```
// ! flags
export const NoFlags = /*                      */ 0b0000000000000000000000;

export const Placement = /*                    */ 0b0000000000000000000010; // 2
export const Update = /*                       */ 0b0000000000000000000100; // 4
export const Deletion = /*                     */ 0b0000000000000000001000; // 8

//**********************************************************************************
*************
```

```
// ! HookFlags
export const HookLayout = /*    */ 0b010;
export const HookPassive = /*   */ 0b100;

//********************************************************************************
*************

export function isFn(fn) {
  return typeof fn === "function";
}

export function isStr(s) {
  return typeof s === "string";
}

export function isStringOrNumber(s) {
  return typeof s === "string" || typeof s === "number";
}

export function isArray(arr) {
  return Array.isArray(arr);
}

export function areHookInputsEqual(nextDeps, prevDeps) {
  if (prevDeps === null) {
    return false;
  }

  for (let i = 0; i < prevDeps.length && i < nextDeps.length; i++) {
    if (Object.is(nextDeps[i], prevDeps[i])) {
      continue;
    }
    return false;
  }
  return true;
}

export function updateNode(node, prevVal, nextVal) {
  Object.keys(prevVal)
    // .filter(k => k !== "children")
    .forEach((k) => {
      if (k === "children") {
        // 有可能是文本
        if (isStringOrNumber(prevVal[k])) {
          node.textContent = "";
        }
      } else if (k.slice(0, 2) === "on") {
        const eventName = k.slice(2).toLocaleLowerCase();
        node.removeEventListener(eventName, prevVal[k]);
      } else {
        if (!(k in nextVal)) {
          node[k] = "";
        }
      }
    });

  Object.keys(nextVal)
```

```
    // .filter(k => k !== "children")
    .forEach((k) => {
      if (k === "children") {
        // 有可能是文本
        if (isStringOrNumber(nextVal[k])) {
          node.textContent = nextVal[k] + "";
        }
      } else if (k.slice(0, 2) === "on") {
        const eventName = k.slice(2).toLocaleLowerCase();
        node.addEventListener(eventName, nextVal[k]);
      } else {
        node[k] = nextVal[k];
      }
    });
}
```

实现协调

ReactFiberReconciler.js

```
export function updateFunctionComponent(wip) {
  renderHooks(wip);

  const {type, props} = wip;
  const children = type(props);

  // 协调子节点
  reconcileChildren(wip, children);
}
```

ReactChildFiber.js 简版

```
import {createFiber} from "./fiber";
import {isStringOrNumber, Update} from "./utils";

function deleteChild(returnFiber, childToDelete) {
  if (returnFiber.deletions) {
    returnFiber.deletions.push(childToDelete);
  } else {
    returnFiber.deletions = [childToDelete];
  }
}

// 协调子节点
// diff
// 1 2 3 4
// 2 3 4
export function reconcileChildren(returnFiber, children) {
  if (isStringOrNumber(children)) {
    return;
  }
```

```
  const newChildren = Array.isArray(children) ? children : [children];

  let previousNewFiber = null;
  // 老节点的头结点
  let oldFiber = returnFiber.alternate && returnFiber.alternate.child;
  for (let i = 0; i < newChildren.length; i++) {
    const newChild = newChildren[i];
    if (newChild === null) {
      continue;
    }
    const newFiber = createFiber(newChild, returnFiber);
    const same = sameNode(oldFiber, newFiber);
    if (same) {
      // 更新
      Object.assign(newFiber, {
        alternate: oldFiber, // 老节点
        stateNode: oldFiber.stateNode, // dom节点
        flags: Update, // fiber标记更新
      });
    }

    if (!same && oldFiber) {
      deleteChild(returnFiber, oldFiber);
    }

    if (oldFiber) {
      oldFiber = oldFiber.sibling;
    }

    if (previousNewFiber === null) {
      returnFiber.child = newFiber;
    } else {
      previousNewFiber.sibling = newFiber;
    }

    previousNewFiber = newFiber;
  }
}

// 同一个节点， 调用前提是同一个层级下
function sameNode(a, b) {
  return !!(a && b && a.key === b.key && a.type === b.type);
}
```

ReactChildFiber.js 完整版

```
import {createFiber} from "./fiber";
import {
  isArray,
  isStr,
  isStringOrNumber,
  Update,
  updateNode,
  Placement,
} from "./utils";
```

```
function placeChild(
  newFiber,
  lastPlacedIndex,
  newIndex,
  shouldTrackSideEffects // 初次渲染（false）还是更新（true）
) {
  newFiber.index = newIndex;
  if (!shouldTrackSideEffects) {
    return lastPlacedIndex;
  }
  const current = newFiber.alternate;

  if (current) {
    const oldIndex = current.index;
    if (oldIndex < lastPlacedIndex) {
      // move
      newFiber.flags = Placement;
      return lastPlacedIndex;
    } else {
      return oldIndex;
    }
  } else {
    newFiber.flags = Placement;
    return lastPlacedIndex;
  }
}

// 删除单个节点
function deleteChild(returnFiber, childToDelete) {
  // returnFiber.deletoins = [...]
  const deletoins = returnFiber.deletoins;
  if (deletoins) {
    returnFiber.deletoins.push(childToDelete);
  } else {
    returnFiber.deletoins = [childToDelete];
  }
}

// 删除节点链表,头结点是currentFirstChild
function deleteRemainingChildren(returnFiber, currentFirstChild) {
  let childToDelete = currentFirstChild;
  while (childToDelete) {
    deleteChild(returnFiber, childToDelete);
    childToDelete = childToDelete.sibling;
  }
}

function mapRemainingChildren(currentFirstChild) {
  const existingChildren = new Map();
  let existingChild = currentFirstChild;
  while (existingChild) {
    existingChildren.set(
      existingChild.key || existingChild.index,
      existingChild
    );
    existingChild = existingChild.sibling;
  }
```

```
  return existingChildren;
}

// old 1 2 3 4
// new 2 3 4
export function reconcileChildren(returnFiber, children) {
  if (isStringOrNumber(children)) {
    return;
  }

  const shouldTrackSideEffects = !!returnFiber.alternate;

  const newChildren = isArray(children) ? children : [children];

  let previousNewFiber = null;
  let oldFiber = returnFiber.alternate && returnFiber.alternate.child;
  let nextOldFiber = null; // 记录下一个oldFiber节点
  // 记录遍历newChildren的下标
  let newIndex = 0;
  let lastPlacedIndex = 0;

  // ! 1. 更新阶段，找到能复用的节点，如果不能复用，则停止这轮循环
  for (; oldFiber && newIndex < newChildren.length; newIndex++) {
    const newChild = newChildren[newIndex];
    if (newChild === null) {
      continue;
    }
    if (oldFiber.index > newIndex) {
      nextOldFiber = oldFiber;
      oldFiber = null;
    } else {
      nextOldFiber = oldFiber.sibling;
    }

    const same = sameNode(newChild, oldFiber);

    if (!same) {
      if (oldFiber === null) {
        oldFiber = nextOldFiber;
      }

      break;
    }

    const newFiber = createFiber(newChild, returnFiber);
    Object.assign(newFiber, {
      alternate: oldFiber,
      stateNode: oldFiber.stateNode,
      flags: Update,
    });

    lastPlacedIndex = placeChild(
      newFiber,
      lastPlacedIndex,
      newIndex,
      shouldTrackSideEffects
    );
```

```
    if (previousNewFiber === null) {
      returnFiber.child = newFiber;
    } else {
      previousNewFiber.sibling = newFiber;
    }
    previousNewFiber = newFiber;
    oldFiber = nextOldFiber;
  }

  // ! 2.检验newChildren是否到头，如果newChildren已经遍历完，那么就把剩下的oldFiber链表删除就行了
  if (newIndex === newChildren.length) {
    deleteRemainingChildren(returnFiber, oldFiber);
    return;
  }

  // !3.
  // 1). 初次渲染
  // 2). oldFiber已经为null，也就意味着老fiber链表已经被复用完成，但是新的Children还有fiber要生成，就只能新增插入了
  if (!oldFiber) {
    for (; newIndex < newChildren.length; newIndex++) {
      const newChild = newChildren[newIndex];
      if (newChild === null) {
        continue;
      }
      const newFiber = createFiber(newChild, returnFiber);
      lastPlacedIndex = placeChild(
        newFiber,
        lastPlacedIndex,
        newIndex,
        shouldTrackSideEffects
      );

      if (previousNewFiber === null) {
        returnFiber.child = newFiber;
      } else {
        previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
    }
    return;
  }

  // !4.同时遍历新子节点数组和老链表，能复用的节点就复用，没法复用的老节点删除，新节点新增插入
  // old 1: {} 2:{} 3:{} 4:{} (查找、删除)
  // new 2 4 5
  const existingChildren = mapRemainingChildren(oldFiber);
  for (; newIndex < newChildren.length; newIndex++) {
    const newChild = newChildren[newIndex];
    if (newChild === null) {
      continue;
    }
    const newFiber = createFiber(newChild, returnFiber);
    lastPlacedIndex = placeChild(
      newFiber,
      lastPlacedIndex,
      newIndex,
```

```
      shouldTrackSideEffects
    );

    // 从老链表上找能复用的节点
    let matchedFiber = existingChildren.get(newFiber.key || newFiber.index);
    if (matchedFiber) {
      // 找到能复用的节点
      existingChildren.delete(newFiber.key || newFiber.index);
      Object.assign(newFiber, {
        alternate: matchedFiber,
        stateNode: matchedFiber.stateNode,
        flags: Update,
      });
    }

    if (previousNewFiber === null) {
      returnFiber.child = newFiber;
    } else {
      previousNewFiber.sibling = newFiber;
    }
    previousNewFiber = newFiber;
  }

  if (shouldTrackSideEffects) {
    existingChildren.forEach((child) => deleteChild(returnFiber, child));
  }
}

export function sameNode(a, b) {
  return !!(a && b && a.key === b.key && a.type === b.type);
}
```

回顾

React原理解析03

- 资源
- 课堂目标
- 知识点
  - Hook
    - Hook简介
      - 视频介绍
      - 没有破坏性改动
    - Hook解决了什么问题
      - 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
    - Hook API
    - 实现useEffect与useLayoutEffect
      - hooks.js
      - ReactFiberWorkLoop.js简版
      - ReactFiberWorkLoop.js完整版
      - utils.js
  - 实现协调
    - ReactFiberReconciler.js
    - ReactChildFiber.js 简版

作业

1. 查看useMemo以及useCallback，理解源码，口述原理。**这个作业不用提交~**
2. 使用useCallback与useMemo，修改下面给出的例子，使其可以实现值与函数的缓存。**这个作业提交到学习中心，两张代码截图即可。**

```
import * as React from "react";
import {useState, useCallback, PureComponent} from "react";

export default function UseCallbackPage(props) {
  const [count, setCount] = useState(0);
  const addClick = () => {
    let sum = 0;
    for (let i = 0; i < count; i++) {
      sum += i;
    }
    return sum;
  };
  const [value, setValue] = useState("");
  return (
    <div>
      <h3>UseCallbackPage</h3>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>add</button>
      <input value={value} onChange={event => setValue(event.target.value)} />
      <Child addClick={addClick} />
    </div>
  );
}

class Child extends PureComponent {
  render() {
    console.log("child render");
    const {addClick} = this.props;
    return (
      <div>
        <h3>Child</h3>
        <button onClick={() => console.log(addClick())}>add</button>
      </div>
    );
  }
}
```

```
import * as React from "react";
import {useState, useMemo} from "react";

export default function UseMemoPage(props) {
```

```
  const [count, setCount] = useState(0);
  const [value, setValue] = useState("");
  const expensive = () => {
    console.log("compute");
    let sum = 0;
    for (let i = 0; i < count; i++) {
      sum += i;
    }
    return sum;
  };

  return (
    <div>
      <h3>UseMemoPage</h3>
      <p>expensive:{expensive()}</p>
      <p>{count}</p>
      <button onClick={() => setCount(count + 1)}>add</button>
      <input value={value} onChange={event => setValue(event.target.value)} />
    </div>
  );
}
```

下节课内容

1. 事件系统
2. setState、forceUpdate、render的具体更新流程