React原理解析03

资源

1. React Hook简介
2. React源码

课堂目标

1. 掌握fiber更新策略
2. 掌握hook原理

知识点

Hook

Hook简介

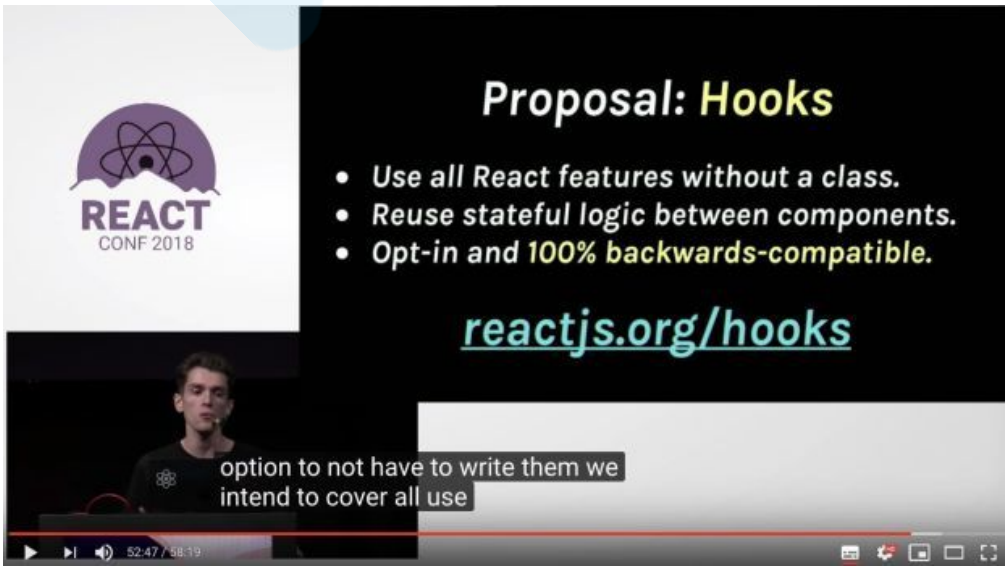
Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

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，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
      </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。

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`

Hooks原理

```
function FunctionComponent({name}) {
  const [count, setCount] = useState(0);

  return (
    <div className="border">
      {name}
      <button onClick={() => setCount(count + 1)}> {count}: count add</button>
      <div className="border">
        {count % 2 ? (
          <button onClick={() => console.log("omg")}>click</button>
        ) : (
          <div>omg</div>
        )}
      </div>
    </div>
  );
}
```

```
function FunctionalComponent () {
  const [state1, setState1] = useState(1)
  const [state2, setState2] = useState(2)
  const [state3, setState3] = useState(3)
}

hook1 => Fiber.memoizedState
state1 === hook1.memoizedState
hook1.next => hook2
state2 === hook2.memoizedState
hook2.next => hook3
state3 === hook2.memoizedState
```

实现useState

```
// !hook 实现
// 当前正在工作的fiber
let wipFiber = null;
let hookIndex = null;
export function useState(init) {
  const oldHook = wipFiber.base && wipFiber.base.hooks[hookIndex];
  const hook = {state: oldHook ? oldHook.state : init, queue: []};
  const actions = oldHook ? oldHook.queue : [];
  actions.forEach(action => (hook.state = action));
  const setState = action => {
    hook.queue.push(action);
    wipRoot = {
      node: currentRoot.node,
      props: currentRoot.props,
      base: currentRoot
    };
    nextUnitOfWork = wipRoot;
    deletions = [];
  };
```

```
  wipFiber.hooks.push(hook);
  hookIndex++;
  return [hook.state, setState];
}

function updateFunctionComponent(fiber) {
  wipFiber = fiber;
  wipFiber.hooks = [];
  hookIndex = 0;
  const {type, props} = fiber;
  const children = [type(props)];
  reconcileChildren(fiber, children);
}
```

遍历子节点，判断删除更新

```
function reconcileChildren(workInProgressFiber, children) {
  //  构建fiber结构
  // 更新 删除 新增
  let prevSibling = null;
  let oldFiber = workInProgressFiber.base && workInProgressFiber.base.child;
  for (let i = 0; i < children.length; i++) {
    let child = children[i];
    let newFiber = null;
    const sameType = child && oldFiber && child.type === oldFiber.type;
    if (sameType) {
      // 类型相同 复用
      newFiber = {
        type: oldFiber.type,
        props: child.props,
        node: oldFiber.node,
        base: oldFiber,
        return: workInProgressFiber,
        effectTag: UPDATE
      };
    }
    if (!sameType && child) {
      // 类型不同 child存在 新增插入
      newFiber = {
        type: child.type,
        props: child.props,
        node: null,
        base: null,
        return: workInProgressFiber,
        effectTag: PLACEMENT
      };
    }
    if (!sameType && oldFiber) {
      // 删除
      oldFiber.effectTag = DELETION;
      deletions.push(oldFiber);
    }

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

```
    }

    // 形成链表结构
    if (i === 0) {
      workInProgressFiber.child = newFiber;
    } else {
      // i>0
      prevSibling.sibling = newFiber;
    }
    prevSibling = newFiber;
  }
}
```

Commit阶段加上删除更新

```
// ! commit阶段
function commitRoot() {
  deletions.forEach(commitWorker);
  commitWorker(wipRoot.child);
  currentRoot = wipRoot;
  wipRoot = null;
}

function commitWorker(fiber) {
  if (!fiber) {
    return;
  }

  // 向上查找
  let parentNodeFiber = fiber.return;
  while (!parentNodeFiber.node) {
    parentNodeFiber = parentNodeFiber.return;
  }

  const parentNode = parentNodeFiber.node;
  if (fiber.effectTag === PLACEMENT && fiber.node !== null) {
    parentNode.appendChild(fiber.node);
  } else if (fiber.effectTag === UPDATE && fiber.node !== null) {
    updateNode(fiber.node, fiber.base.props, fiber.props);
  } else if (fiber.effectTag === DELETION && fiber.node !== null) {
    commitDeletions(fiber, parentNode);
  }

  commitWorker(fiber.child);
  commitWorker(fiber.sibling);
}

function commitDeletions(fiber, parentNode) {
  if (fiber.node) {
    parentNode.removeChild(fiber.node);
  } else {
    commitDeletions(fiber.child, parentNode);
  }
}
```

节点更新

```
import {TEXT, PLACEMENT, UPDATE, DELETION} from "./const";

// 下一个单元任务
let nextUnitOfWork = null;
// work in progress fiber root
let wipRoot = null;
// 现在的根节点
let currentRoot = null;

let deletions = null;

// fiber 结构
/**
 * child 第一个子元素
 * sibling 下一个兄弟节点
 * return 父节点
 * node 存储当前node节点
 */

function render(vnode, container) {
  wipRoot = {
    node: container,
    props: {
      children: [vnode]
    },
    base: currentRoot
  };
  nextUnitOfWork = wipRoot;
  deletions = [];
}

function updateNode(node, preVal, nextVal) {
  Object.keys(preVal)
    .filter(k => k !== "children")
    .forEach(k => {
      if (k.slice(0, 2) === "on") {
        // 简单处理 on开头当做事件
        let eventName = k.slice(2).toLowerCase();
        node.removeEventListener(eventName, preVal[k]);
      } else {
        if (!(k in nextVal)) {
          node[k] = "";
        }
      }
    });

  Object.keys(nextVal)
    .filter(k => k !== "children")
    .forEach(k => {
      if (k.slice(0, 2) === "on") {
        // 简单处理 on开头当做事件
        let eventName = k.slice(2).toLowerCase();
        node.addEventListener(eventName, nextVal[k]);
```

```
      } else {
        node[k] = nextVal[k];
      }
    });
}
```

如何调试源码

步骤如下：

```
clone文件: git clone https://github.com/bubucuo/DebugReact.git
安装： yarn
启动： yarn start
```

方便查看逻辑，去webpack里把dev设置为false，参考上面的git地址。

React中的数据结构

Fiber

SideEffectTag

注： 这里的effectTag都是二进制，这个和React中用到的位运算有关。首先我们要知道位运算只能用于整数，并且是直接对二进制位进行计算，直接处理每一个比特位，是非常底层的运算，运算速度极快。

比如说workInProgress.effectTag为132，那这个时候，workInProgress.effectTag & Update 和 workInProgress.effectTag & Ref在布尔值上都是true，这个时候就是既要执行`update effect`，还要执行`ref update`。

还有一个例子如workInProgress.effectTag |= Placement;这里就是说给workInProgress添加一个Placement的副作用。

这种处理不仅速度快，而且简洁方便，是非常巧妙的方式，很值得我们学习借鉴。

ReactWorkTag

library > DebugReact > src > react > packages > shared > ReactWorkTags.js

```
35  export const FunctionComponent = 0;
36  export const ClassComponent = 1;
37  export const IndeterminateComponent = 2; // Before we know
38  export const HostRoot = 3; // Root of a host tree. Could be
39  export const HostPortal = 4; // A subtree. Could be an entr
40  export const HostComponent = 5;
41  export const HostText = 6;
42  export const Fragment = 7;
43  export const Mode = 8;
44  export const ContextConsumer = 9;
45  export const ContextProvider = 10;
46  export const ForwardRef = 11;
47  export const Profiler = 12;
48  export const SuspenseComponent = 13;
49  export const MemoComponent = 14;
50  export const SimpleMemoComponent = 15;
51  export const LazyComponent = 16;
52  export const IncompleteClassComponent = 17;
53  export const DehydratedFragment = 18;
54  export const SuspenseListComponent = 19;
55  export const FundamentalComponent = 20;
56  export const ScopeComponent = 21;
57  export const Block = 22;
```

Update & UpdateQueue

tag的标记不同类型，如执行forceUpdate的时候，tag值就是2。

这个的payload是参数，比如setState更新时候，payload就是partialState，render的时候，payload就是第一个参数，即element。

src > react > packages > react-reconciler > src > ReactUpdateQueue.js

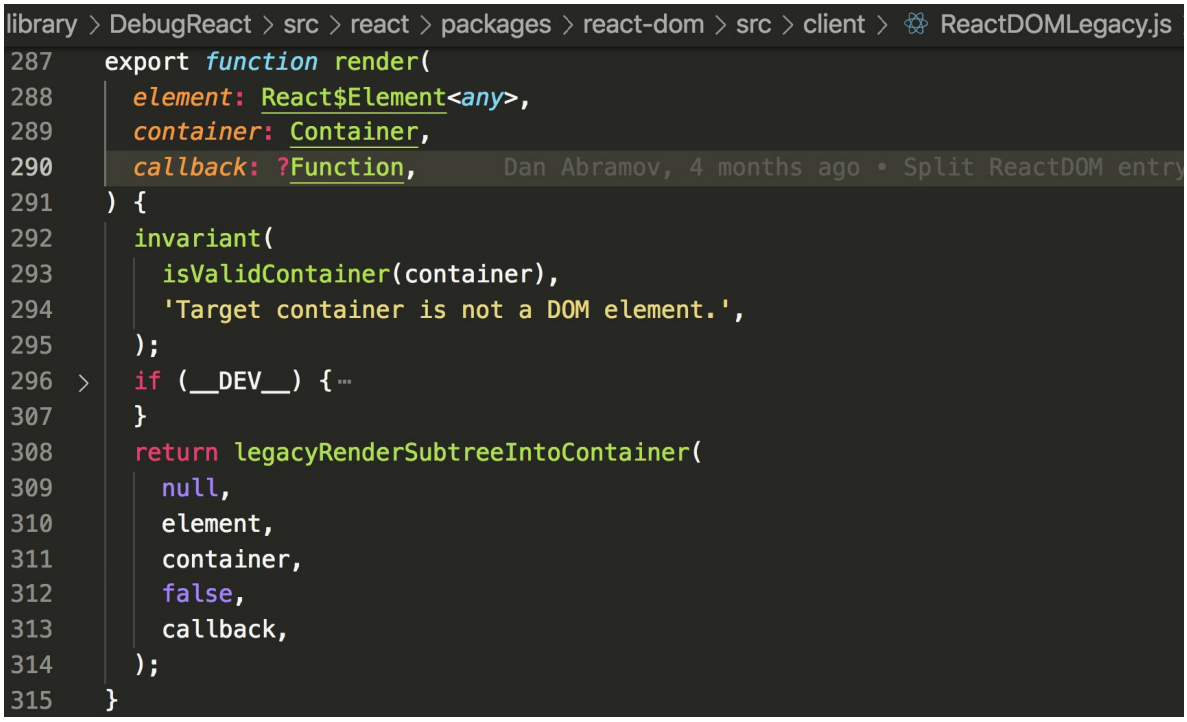
```
export type Update<State> = {|
  expirationTime: ExpirationTime,
  suspenseConfig: null | SuspenseConfig,

  tag: 0 | 1 | 2 | 3,
  payload: any,
  callback: (() => mixed) | null,

  next: Update<State>,

  // DEV only
  priority?: ReactPriorityLevel,
|};

type SharedQueue<State> = {|pending: Update<State> | null|};

export type UpdateQueue<State> = {|
  baseState: State,
  baseQueue: Update<State> | null,
  shared: SharedQueue<State>,
  effects: Array<Update<State>> | null,
|};
```

library > DebugReact > src > react > packages > react-reconciler > src > ReactUpdateQueue.js

```
132
133   export const UpdateState = 0;
134   export const ReplaceState = 1;
135   export const ForceUpdate = 2;
136   export const CaptureUpdate = 3;
```

创建更新

ReactDOM.render



上面render调用legacyRenderSubtreeIntoContainer，可以看到parentComponent设置为null。

初次渲染，生成fiberRoot，以后每次update，都要使用这个fiberRoot。

callback是回调，如果为function，则每次渲染和更新完成，都会执行，调用originalCallback.call(instance)。

updateContainer中计算过期时间并做返回，同时创建update，并给update.payload赋值为element，即这里的子元素，（这里可以setState做对比）。

296行入栈更新

297行进入任务调度。

DebugReact > src > react > packages > react-reconciler > src > JS ReactFiberReconciler.old.js > updateContainer

```
243  export function updateContainer(
244    element: ReactNodeList,
245    container: OpaqueRoot,
246    parentComponent: ?React$Component<any, any>,
247    callback: ?Function,
248  ): ExpirationTime {
249    const current = container.current;
250    const currentTime = requestCurrentTimeForUpdate();
251
252    const suspenseConfig = requestCurrentSuspenseConfig();
253    const expirationTime = computeExpirationForFiber(
254      currentTime,
255      current,
256      suspenseConfig,
257    );
258
259    const context = getContextForSubtree(parentComponent);
260    if (container.context === null) {
261      container.context = context;
262    } else {
263      container.pendingContext = context;
264    }
265
266    const update = createUpdate(expirationTime, suspenseConfig);
267    // Caution: React DevTools currently depends on this property
268    // being called "element".
269    update.payload = {element};
270
271    callback = callback === undefined ? null : callback;
272    if (callback !== null) {
273      update.callback = callback;
274    }
275
276    enqueueUpdate(current, update);
277    scheduleUpdateOnFiber(current, expirationTime);
278
279    return expirationTime;
280  }
```

253行生成一个update，实现如下：

library > DebugReact > src > react > packages > react-reconciler > src > ReactUpdateQueue.js

```
184  export function createUpdate(
185    expirationTime: ExpirationTime,
186    suspenseConfig: null | SuspenseConfig,
187  ): Update<*> {
188    let update: Update<*> = {
189      expirationTime,
190      suspenseConfig,
191
192      tag: UpdateState,
193      payload: null,
194      callback: null,
195
196      next: (null: any),
197    };
198    update.next = update;
199    if (__DEV__) {…
201    }
202    return update;      lunaruan, 8 months ago • Update Suspense Priority War
203  }
```

setState与forceUpdate

setState调用updater的enqueueSetState，这里的payload就是setState的第一个参数partialState，是个对象或者function。

相应的forceUpdate调用updater.enqueueForceUpdate，并没有payload，而有一个标记为ForceUpdate（2）的tag，对比上面createUpdate的tag是UpdateState（0）。

library > DebugReact > src > react > packages > react-reconciler > src > ReactUpdateQueue.js

```
132
133   export const UpdateState = 0;
134   export const ReplaceState = 1;
135   export const ForceUpdate = 2;
136   export const CaptureUpdate = 3;
```

DebugReact > src > react > packages > react-reconciler > src > ReactFiberClassComponent.old.js > classComponentUpdater

```
192
193   const classComponentUpdater = {
194     isMounted,
195     enqueueSetState(inst, payload, callback) {
196       const fiber = getInstance(inst);
197       const currentTime = requestCurrentTimeForUpdate();
198       const suspenseConfig = requestCurrentSuspenseConfig();
199       const expirationTime = computeExpirationForFiber(
200         currentTime,
201         fiber,
202         suspenseConfig,
203       );
204       const update = createUpdate(expirationTime, suspenseConfig);
205       update.payload = payload;
206       if (callback !== undefined && callback !== null) {
207         update.callback = callback;
208       }
209       enqueueUpdate(fiber, update);
210       scheduleUpdateOnFiber(fiber, expirationTime);
211     },
212     enqueueReplaceState(inst, payload, callback) {
213       const fiber = getInstance(inst);
214       const currentTime = requestCurrentTimeForUpdate();
215       const suspenseConfig = requestCurrentSuspenseConfig();
216       const expirationTime = computeExpirationForFiber(
217         currentTime,
218         fiber,
219         suspenseConfig,
220       );
221       const update = createUpdate(expirationTime, suspenseConfig);
222       update.tag = ReplaceState;
223       update.payload = payload;
224       if (callback !== undefined && callback !== null) {
225         update.callback = callback;
226       }
227       enqueueUpdate(fiber, update);
228       scheduleUpdateOnFiber(fiber, expirationTime);
229     },
```

协调

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。

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。

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。

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。

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);
  }
```

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

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，修改课上给出的例子，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. 协调
2. 事件系统
3. setState、forceUpdate、render的具体更新流程
4. 组件常见优化技术