React原理解析02

资源

1. React中文网
2. React源码

课堂目标

1. 掌握虚拟dom、diff策略
2. 掌握fiber原理及实现

知识点

reconciliation协调

设计动力

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。

这个算法问题有一些通用的解决方案，即生成将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)

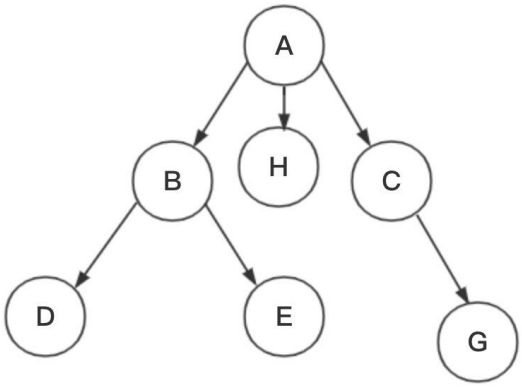
diff 策略

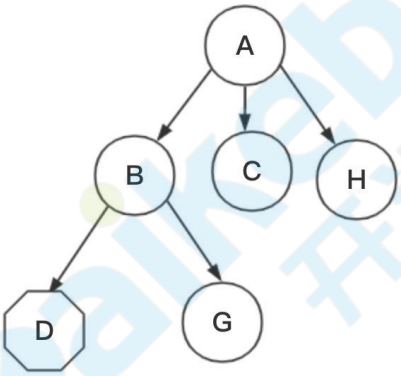
1. 同级比较，Web UI 中 DOM 节点跨层级的移动操作特别少，可以忽略不计。
2. 拥有不同类型的两个组件将会生成不同的树形结构。

   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；

保证key的稳定性例如不使用newDate()或者random()等函数去创造key值
key维持不可变





diff过程

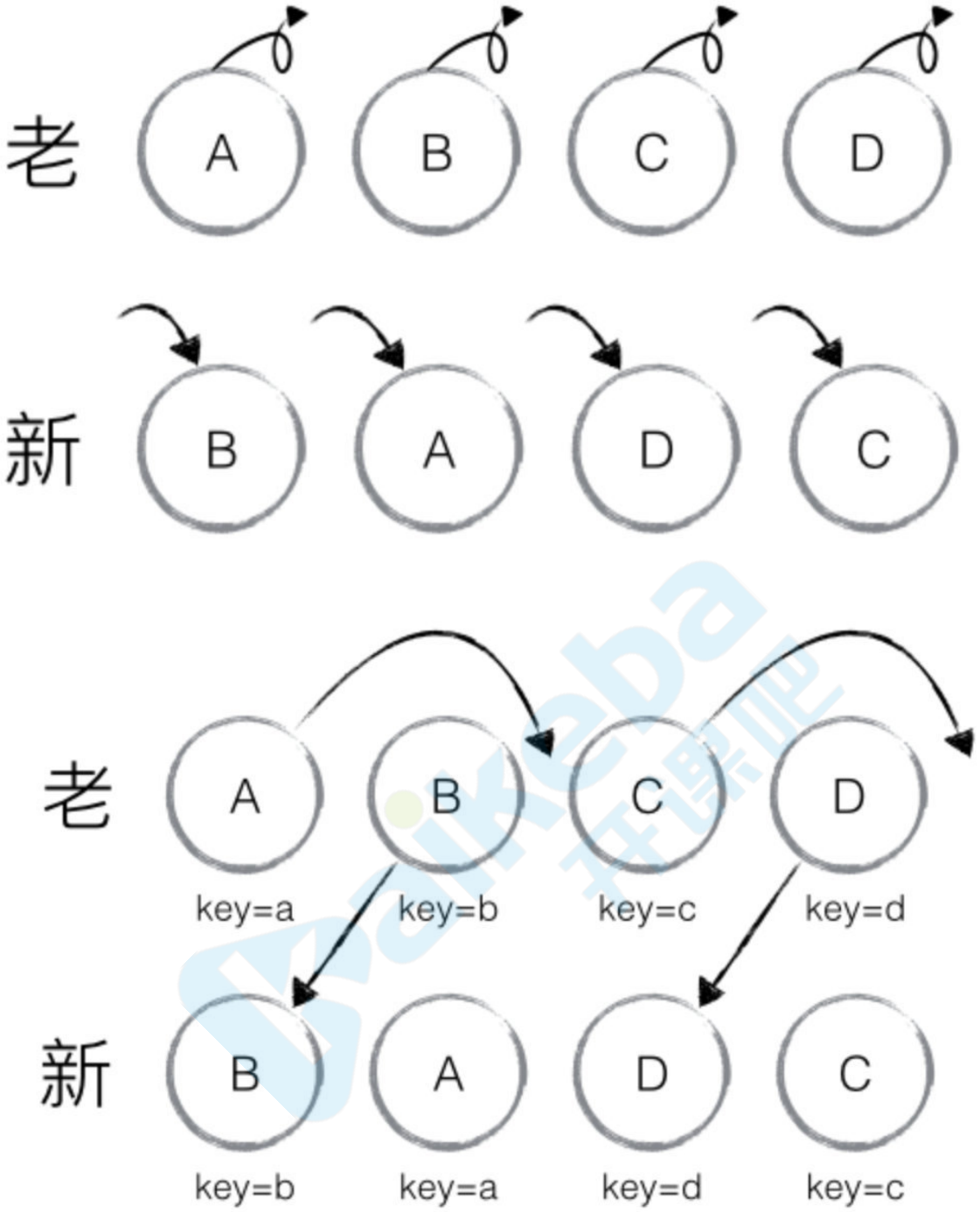
比对两个虚拟dom时会有三种操作：删除、替换和更新

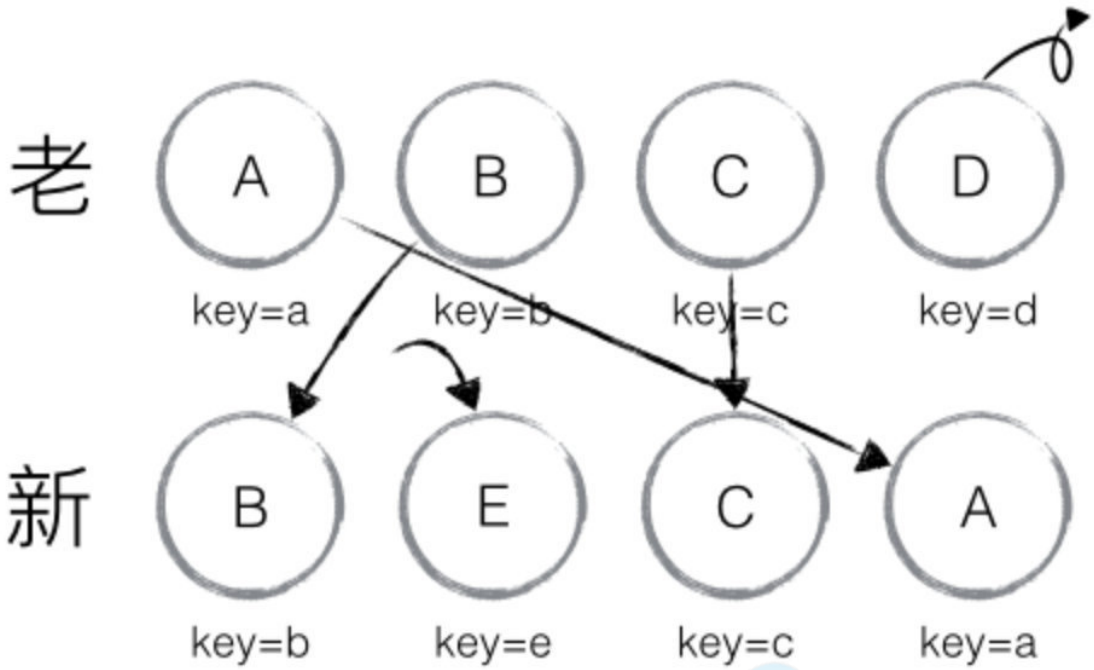
vnode是现在的虚拟dom，newVnode是新虚拟dom。

删除：newVnode不存在时

替换：vnode和newVnode类型不同或key不同时

更新：有相同类型和key但vnode和newVnode不同时


老
A
B
C
D
新
B
A
D
C
老
A
B
C
D
key=a
key=b
key=c
key=d
新
B
A
D
C
key=b
key=a
key=d
key=c



在实践中也证明这三个前提策略是合理且准确的，它保证了整体界面构建的性能。

fiber

为什么需要fiber

[React Conf 2017 Fiber介绍视频]

React的killer feature： virtual dom

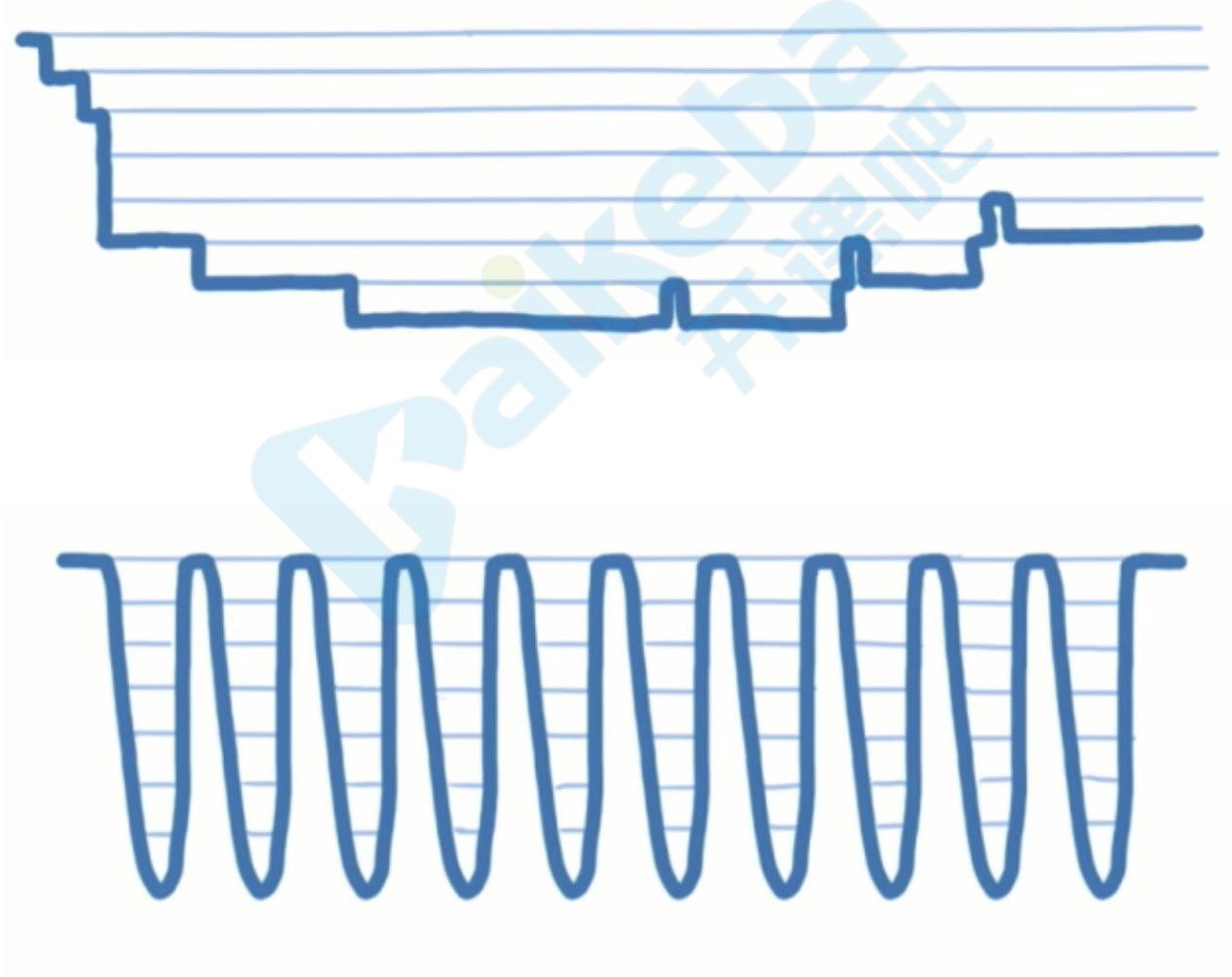
1. 为什么需要fiber

   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

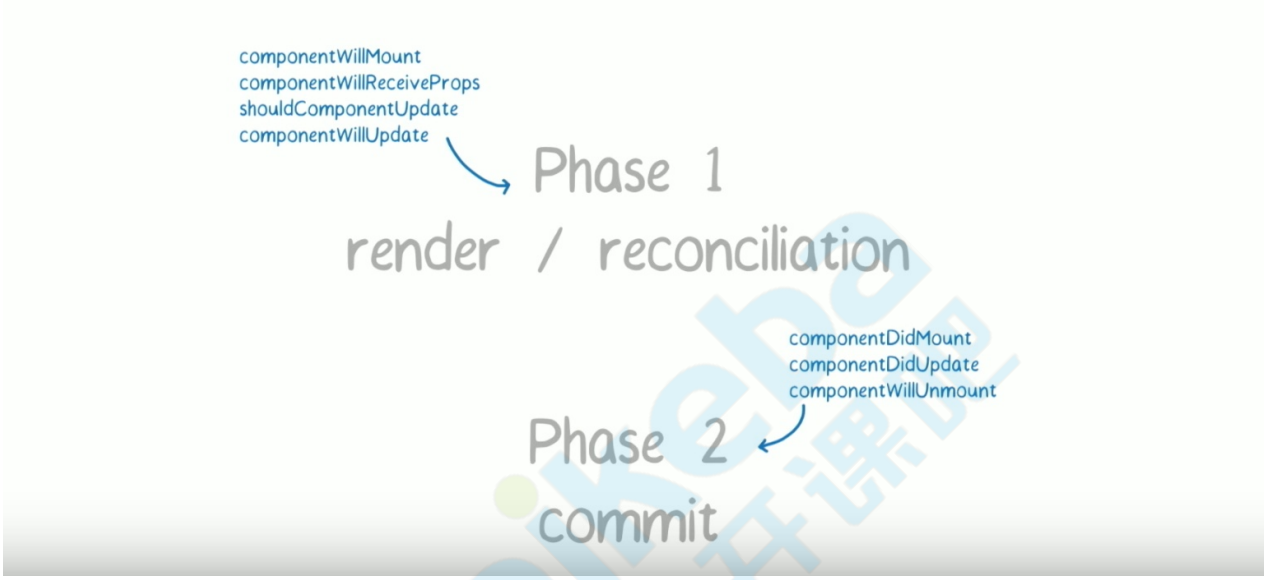
3. 增量渲染（把渲染任务拆分成块，匀到多帧）
4. 更新时能够暂停，终止，复用渲染任务
5. 给不同类型的更新赋予**优先级**
6. 并发方面新的基础能力
7. **更流畅**



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。



实现fiber

`window.requestIdleCallback(callback[, options])`

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。函数一般会按先进先调用的顺序执行，然而，如果回调函数指定了执行超时时间`timeout`，则有可能为了在超

时前执行函数而打乱执行顺序。

你可以在空闲回调函数中调用`requestIdleCallback()`，以便在下一次通过事件循环之前调度另一个回调。

`callback`

一个在事件循环空闲时即将被调用的函数的引用。函数会接收到一个名为 `IdleDeadline` 的参数，这个参数可以获取当前空闲时间以及回调是否在超时时间前已经执行的状态。

`options` 可选

包括可选的配置参数。具有如下属性：

- `timeout`：如果指定了timeout并具有一个正值，并且尚未通过超时毫秒数调用回调，那么回调会在下一次空闲时期被强制执行，尽管这样很可能会对性能造成负面影响。

react中requestIdleCallback的hack在react/packages/scheduler/src/forks/SchedulerHostConfig.default.js。

关于fiber

Fiber 是 React 16 中新的协调引擎。它的主要目的是使 Virtual DOM 可以进行增量式渲染。

一个更新过程可能被打断，所以React Fiber一个更新过程被分为两个阶段(Phase)：第一个阶段Reconciliation Phase和第二阶段Commit Phase。

./ReactDOM.js

```
import {PLACEMENT} from "./CONSTS";

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

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

// 根据vnode，创建一个node
function createNode(vnode) {
  const {type, props} = vnode;
  let node;
  if (type === "TEXT") {
```

```
    node = document.createTextNode("");
  } else if (type) {
    node = document.createElement(type);
  }
  updateNode(node, props);
  return node;
}

function
reconcilerChildren(workInProgressFiber,
children) {
  // 构建fiber结构
  // 这里的构建是按照顺序的，没有考虑移动位置等等
  // 更新 删除 新增
  let oldFiber = workInProgressFiber.base &&
workInProgressFiber.base.child;
  let prevSibling = null;
  for (let i = 0; i < children.length; i++) {
    let child = children[i];
    let newFiber = null;
    //  新元素存在
    newFiber = {
      type: child.type,
      props: child.props,
      node: null,
      base: null,
      parent: workInProgressFiber,
      effectTag: PLACEMENT
    };
```

```
    // todo 删除
    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;
  }
  // fiber结构构建 done
}

// 更新节点上属性，如className、nodeValue等
function updateNode(node, nextVal) {
  Object.keys(nextVal)
    .filter(k => k !== "children")
    .forEach(k => {
      if (k.slice(0, 2) === "on") {
        // 以on开头，就认为是一个事件，源码处理复杂一
些，
        let eventName =
k.slice(2).toLocaleLowerCase();
        node.addEventListener(eventName,
nextVal[k]);
      } else {
```

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

function updateHostComponent(fiber) {
  if (!fiber.node) {
    fiber.node = createNode(fiber);
  }
  const {children} = fiber.props;
  reconcilerChildren(fiber, children);
}

function performUnitOfWork(fiber) {
  // 1. 执行当前任务
  // 更新当前
  updateHostComponent(fiber);

  // 2.返回再下一个任务
  // 找下个任务的原则：先找子元素
  if (fiber.child) {
    return fiber.child;
  }
  // 如果没有子元素，寻找兄弟元素
  let nextFiber = fiber;
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling;
    }
```

```
    nextFiber = nextFiber.parent;
  }
}

// 调度diff或者是渲染任务
function workLoop(deadline) {
  // 有下一个任务，并且当前帧还没有结束
  while (nextUnitOfWork &&
deadline.timeRemaining() > 1) {
    nextUnitOfWork =
performUnitOfWork(nextUnitOfWork);
  }
  if (!nextUnitOfWork && wipRoot) {
    // 提交
    commitRoot();
  }
}

requestIdleCallback(workLoop);

function commitRoot() {
  commitWorker(wipRoot.child);
  currentRoot = wipRoot;
  wipRoot = null;
}

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

```
  }
  // 向上查找
  let parentNodeFiber = fiber.parent;
  while (!parentNodeFiber.node) {
    parentNodeFiber = parentNodeFiber.parent;
  }
  const parentNode = parentNodeFiber.node;
  if (fiber.effectTag === PLACEMENT &&
fiber.node !== null) {
    parentNode.appendChild(fiber.node);
  }
  commitWorker(fiber.child);
  commitWorker(fiber.sibling);
}

export default {
  render
};
```

实现函数组件与类组件

```
// import React from "react";
// import ReactDOM from "react-dom";
import React from "./kreact-test/";
import ReactDOM from "./kreact-test/ReactDOM";
import Component from "./kreact-
test/Component";
```

```
import "./index.css";

// 函数组件
function FunctionComponent(props) {
    return <div className="function
border">hello, {props.name}</div>;
}

// 类组件
class ClassComponent extends Component {
    render() {
        return <div className="class
border">hello, {this.props.name}</div>;
    }
}

const jsx = (
    <div className="app">
        <h1>hello, kkb</h1>
        <FunctionComponent name="function组件"
/>
        <ClassComponent name="class组件" />
        <p>全栈课学习</p>
        <a href="https://www.kaikeba.com/">跳
转</a>
    </div>
);
```

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

ReactDOM.js

```
function updateFunctionComponent(fiber) {
  const {type, props} = fiber;
  const children = [type(props)];
  reconcilerChildren(fiber, children);
}

function updateClassComponent(fiber) {
  const {type, props} = fiber;
  const cmp = new type(props); //实例化
  const children = [cmp.render()];
  reconcilerChildren(fiber, children);
}

function performUnitOfWork(fiber) {
  // 1. 执行当前任务
  // 更新当前
  const {type} = fiber;
  if (typeof type === "function") {
    type.isReactComponent
      ? updateClassComponent(fiber)
      : updateFunctionComponent(fiber);
  } else if (type) {
    updateHostComponent(fiber);
```

```
  } else {
    updateFragmentComponent(fiber);
  }

  // 2.返回再下一个任务
  // 找下个任务的原则：先找子元素
  if (fiber.child) {
    return fiber.child;
  }
  // 如果没有子元素，寻找兄弟元素
  let nextFiber = fiber;
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling;
    }
    nextFiber = nextFiber.parent;
  }
  // return
}
```

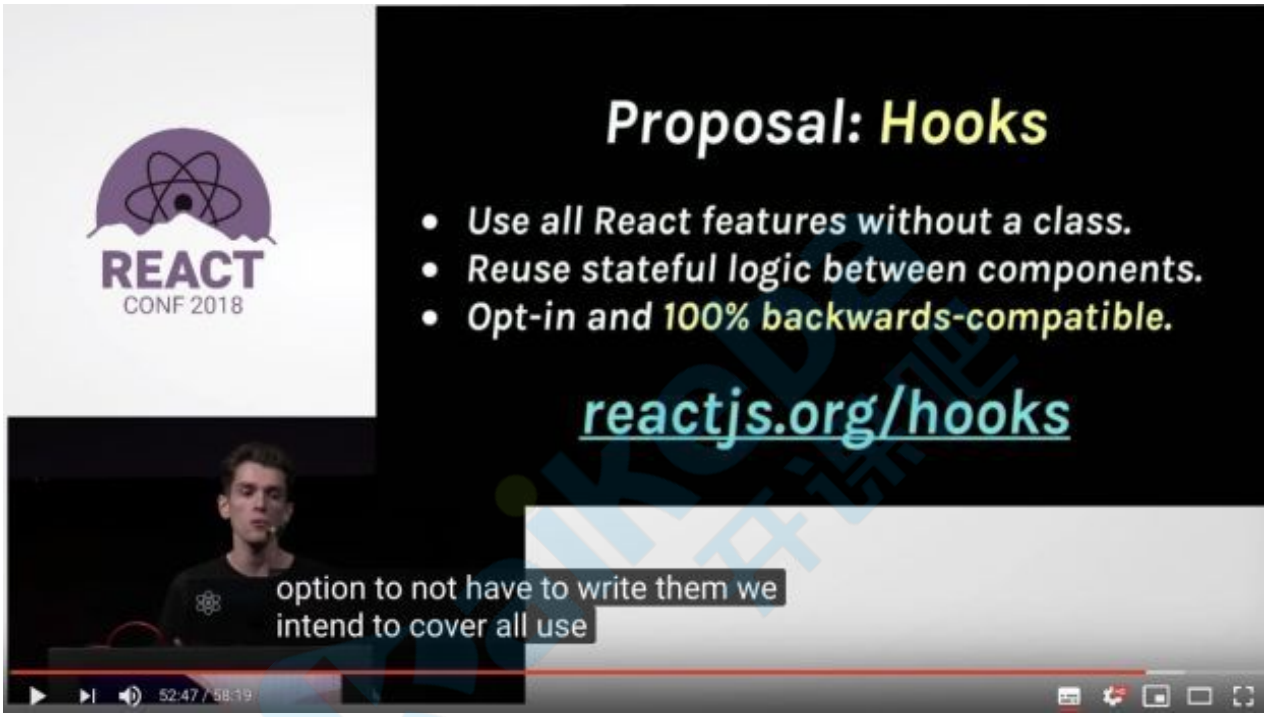
实现Fragment

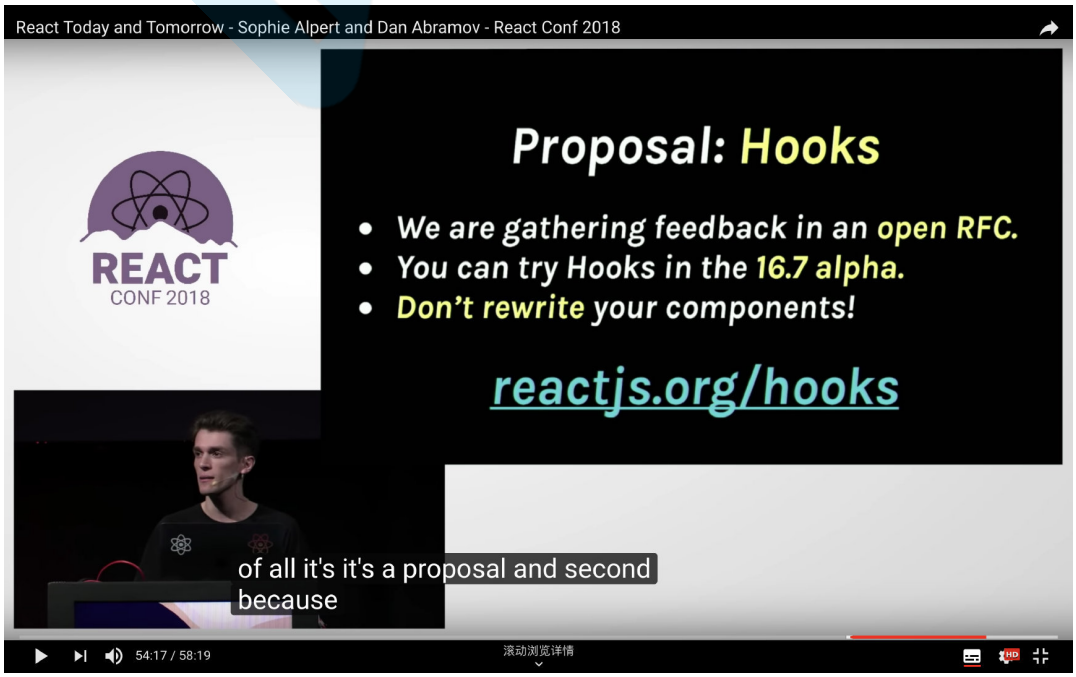
```
function updateFragmentComponent(fiber) {
  const {children} = fiber.props;
  reconcilerChildren(fiber, children);
}
```

Hooks原理

官网

hooks介绍视频：React Today and Tomorrow





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
3. 类似链表的实现原理

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

function FunComp(props) {
  const [data, setData] =
useState('initialState')

  function handleChange(e) {
    setData(e.target.value)
  }

  useEffect(() => {
    //subscribeToSomething()
    return () => {
      //unSubscribeToSomething()
    }
  })

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

```
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

```
// 正在工作的fiber
let wipFiber = null;
let hookIndex = null;
export function useState(init) {
  // 第一次进来用init赋值
  let 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;
  };
  wipFiber.hooks.push(hook);
  console.log("hook", hook); //sy-log
  hookIndex++;
  // 下一次进来就要更新了

  return [hook.state, setState];
}

function performUnitOfWork(fiber) {
  // 1. 执行当前任务
  // 更新当前
  const {type} = fiber;
  if (typeof type === "function") {
    type.isReactComponent
      ? updateClassComponent(fiber)
      : updateFunctionComponent(fiber);
  } else if (type) {
    updateHostComponent(fiber);
  } else {
    updateFragmentComponent(fiber);
```

```
  }

  // 2.返回再下一个任务
  // 找下个任务的原则：先找子元素
  if (fiber.child) {
    return fiber.child;
  }
  // 如果没有子元素，寻找兄弟元素
  let nextFiber = fiber;
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling;
    }
    nextFiber = nextFiber.parent;
  }
  // return
}

function commitWorker(fiber) {
  if (!fiber) {
    return;
  }
  // 向上查找
  let parentNodeFiber = fiber.parent;
  while (!parentNodeFiber.node) {
    parentNodeFiber = parentNodeFiber.parent;
  }
  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);
    parentNode.appendChild(fiber.node);
  }
  commitWorker(fiber.child);
  commitWorker(fiber.sibling);
}
function
reconcilerChildren(workInProgressFiber,
children) {
  // 构建fiber结构
  // 这里的构建是按照顺序的，没有考虑移动位置等等
  // 更新 删除 新增
  let oldFiber = workInProgressFiber.base &&
workInProgressFiber.base.child;
  let prevSibling = null;
  for (let i = 0; i < children.length; i++) {
    let child = children[i];
    let newFiber = null;
    const sameType = oldFiber && child &&
oldFiber.type === child.type;
    if (sameType) {
      // 类型一样
      // update
```

```
      newFiber = {
        type: oldFiber.type,
        props: child.props,
        node: oldFiber.node,
        base: oldFiber,
        parent: workInProgressFiber,
        effectTag: UPDATE
      };
    } else if (child) {
      //  新元素存在
      newFiber = {
        type: child.type,
        props: child.props,
        node: null,
        base: null,
        parent: workInProgressFiber,
        effectTag: PLACEMENT
      };
    }
    // todo 删除
    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;
  }
  // fiber结构构建 done
}
```

至此，整个ReactDOM文件在这里：

```
import {UPDATE, PLACEMENT} from "./CONSTS";

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

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

// 根据vnode，创建一个node
function createNode(vnode) {
```

```
  const {type, props} = vnode;
  let node;
  if (type === "TEXT") {
    node = document.createTextNode("");
  } else if (type) {
    node = document.createElement(type);
  }
  updateNode(node, {}, props);
  return node;
}

function
reconcilerChildren(workInProgressFiber,
children) {
  // 构建fiber结构
  // 这里的构建是按照顺序的，没有考虑移动位置等等
  // 更新 删除 新增
  let oldFiber = workInProgressFiber.base &&
workInProgressFiber.base.child;
  let prevSibling = null;
  for (let i = 0; i < children.length; i++) {
    let child = children[i];
    let newFiber = null;
    const sameType = oldFiber && child &&
oldFiber.type === child.type;
    if (sameType) {
      // 类型一样
      // update
      newFiber = {
```

```
        type: oldFiber.type,
        props: child.props,
        node: oldFiber.node,
        base: oldFiber,
        parent: workInProgressFiber,
        effectTag: UPDATE
      };
    } else if (child) {
      //  新元素存在
      newFiber = {
        type: child.type,
        props: child.props,
        node: null,
        base: null,
        parent: workInProgressFiber,
        effectTag: PLACEMENT
      };
    }
    // todo 删除
    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;
  }
  // fiber结构构建 done
}

// 更新节点上属性，如className、nodeValue等
function updateNode(node, preVal, nextVal) {
  Object.keys(nextVal)
    .filter(k => k !== "children")
    .forEach(k => {
      if (k.slice(0, 2) === "on") {
        // 以on开头，就认为是一个事件，源码处理复杂一
些，
        let eventName =
k.slice(2).toLocaleLowerCase();
        node.addEventListener(eventName,
nextVal[k]);
      } else {
        node[k] = nextVal[k];
      }
    });
}

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

```
  reconcilerChildren(fiber, children);
}

function updateClassComponent(fiber) {
  const {type, props} = fiber;
  const cmp = new type(props); //实例化
  const children = [cmp.render()];
  reconcilerChildren(fiber, children);
}

function updateHostComponent(fiber) {
  if (!fiber.node) {
    fiber.node = createNode(fiber);
  }
  const {children} = fiber.props;
  reconcilerChildren(fiber, children);
}

function updateFragmentComponent(fiber) {
  const {children} = fiber.props;
  reconcilerChildren(fiber, children);
}

function performUnitOfWork(fiber) {
  // 1. 执行当前任务
  // 更新当前
  const {type} = fiber;
  if (typeof type === "function") {
    type.isReactComponent
```

```
      ? updateClassComponent(fiber)
      : updateFunctionComponent(fiber);
  } else if (type) {
    updateHostComponent(fiber);
  } else {
    updateFragmentComponent(fiber);
  }

  // 2.返回再下一个任务
  // 找下个任务的原则：先找子元素
  if (fiber.child) {
    return fiber.child;
  }
  // 如果没有子元素，寻找兄弟元素
  let nextFiber = fiber;
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling;
    }
    nextFiber = nextFiber.parent;
  }
  // return
}

// 调度diff或者是渲染任务
function workLoop(deadline) {
  // 有下一个任务，并且当前帧还没有结束
  while (nextUnitOfWork &&
deadline.timeRemaining() > 1) {
```

```
    nextUnitOfWork =
performUnitOfWork(nextUnitOfWork);
  }
  if (!nextUnitOfWork && wipRoot) {
    // 提交
    commitRoot();
  }
  requestIdleCallback(workLoop);
}

requestIdleCallback(workLoop);

function commitRoot() {
  commitWorker(wipRoot.child);
  currentRoot = wipRoot;
  wipRoot = null;
}

function commitWorker(fiber) {
  if (!fiber) {
    return;
  }
  // 向上查找
  let parentNodeFiber = fiber.parent;
  while (!parentNodeFiber.node) {
    parentNodeFiber = parentNodeFiber.parent;
  }
  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);
    parentNode.appendChild(fiber.node);
  }
  commitWorker(fiber.child);
  commitWorker(fiber.sibling);
}

// 正在工作的fiber
let wipFiber = null;
let hookIndex = null;
export function useState(init) {
  // 第一次进来用init赋值
  let 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;
  };
  wipFiber.hooks.push(hook);
  hookIndex++;
  // 下一次进来就要更新了
  return [hook.state, setState];
}

export default {
  render
};
```

Hook API

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

- `useRef`
- `useImperativeHandle`
- `useLayoutEffect`
- `useDebugValue`

权衡

请谨记协调算法是一个实现细节。React 可以在每个 action 之后对整个应用进行重新渲染，得到的最终结果也会是一样的。在此情境下，重新渲染表示在所有组件内调用 `render` 方法，这不代表 React 会卸载或装载它们。React 只会基于以上提到的规则来决定如何进行差异的合并。

由于 React 依赖探索的算法，因此当以下假设没有得到满足，性能会有所损耗。

1. 该算法不会尝试匹配不同组件类型的子树。如果你发现你在两种不同类型的组件中切换，但输出非常相似的内容，建议把它们改成同一类型。在实践中，我们没有遇到这类问题。
2. Key 应该具有稳定，可预测，以及列表内唯一的特质。不稳定的 key（比如通过 `Math.random()` 生成的）会导致许多组件实例和 DOM 节点被不必要地重新创建，这可能导致性能下降和子组件中的状态丢失。

优化点：减少reflow

回顾

React原理解析02

- 资源
- 课堂目标
- 知识点
  - reconciliation协调
    - 设计动力
    - diffing算法
    - diff 策略
    - diff过程
      - 比对两个虚拟dom时会有三种操作：删除、替换和更新
  - fiber
    - 为什么需要fiber
    - 什么是fiber
  - 实现fiber
    - `window.requestIdleCallback(callback[, options])`
    - 关于fiber
    - 实现函数组件与类组件
    - 实现Fragment
  - Hooks原理
    - 实现useState
  - Hook API
  - 权衡
- 回顾
- 作业

作业

写至少一遍这个代码，加强理解。

下节课内容

源码查看及调试。

课程总结。