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。

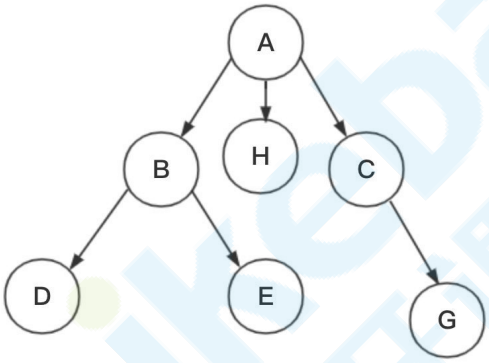
diffing算法

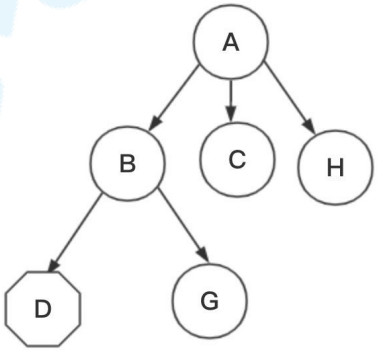
算法复杂度O(n)

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；





diff过程

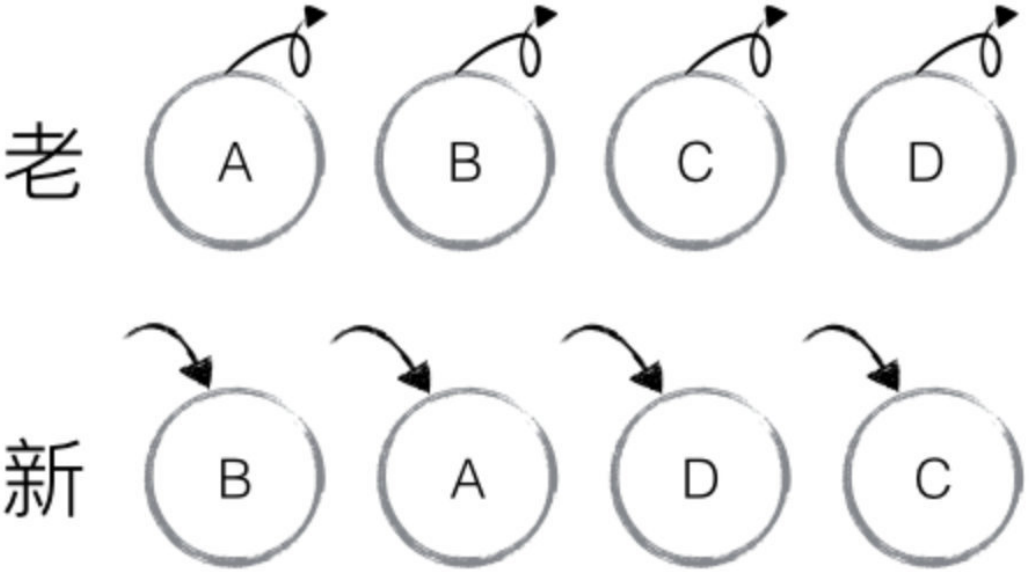
比对两个虚拟dom时会有三种操作：删除、替换和更新

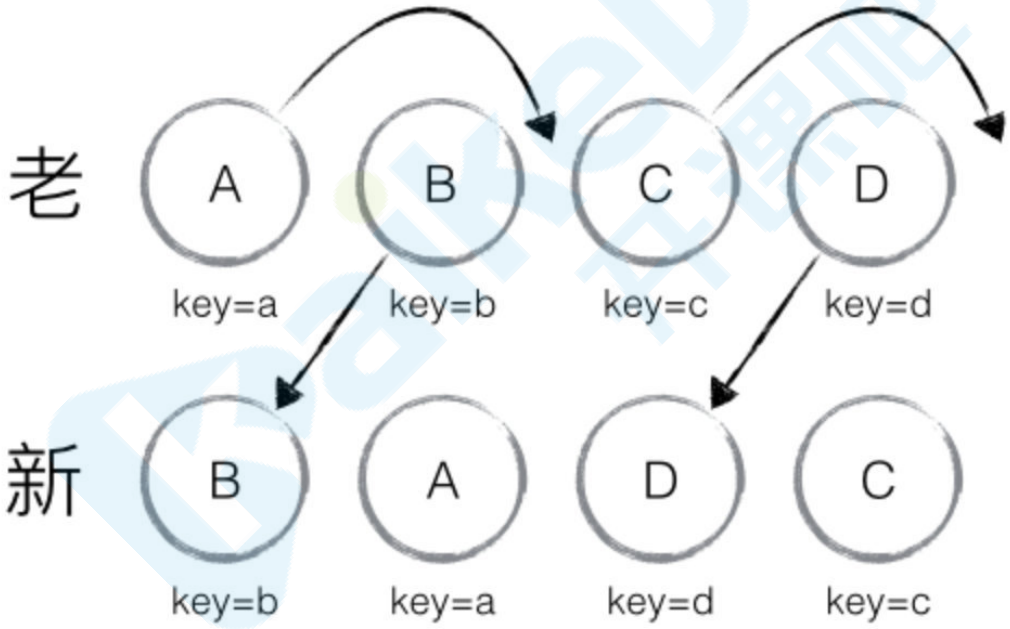
vnode是现在的虚拟dom，newVnode是新虚拟dom。

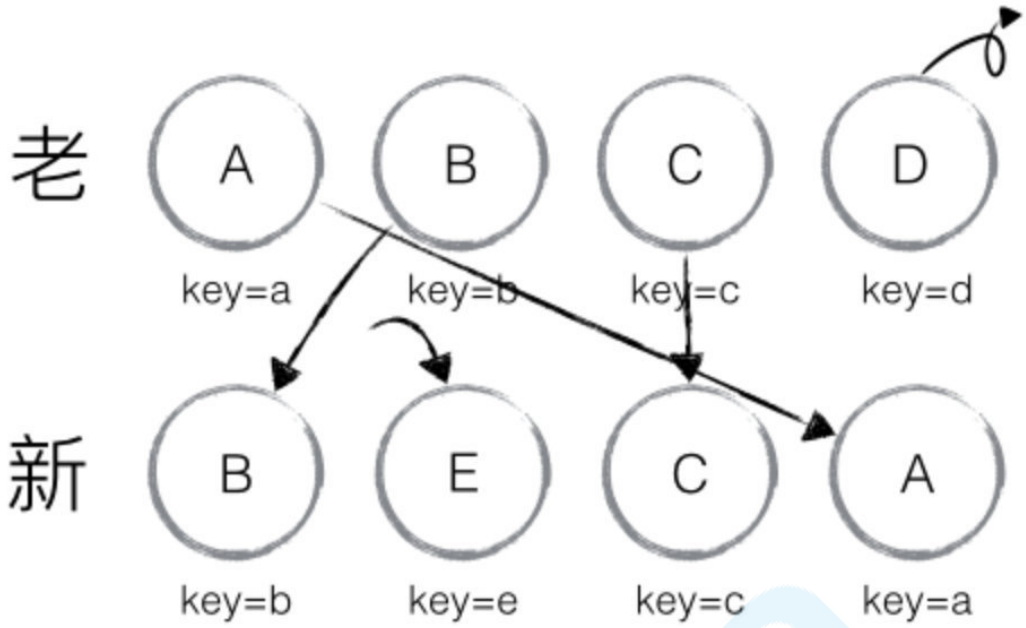
删除：newVnode不存在时

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

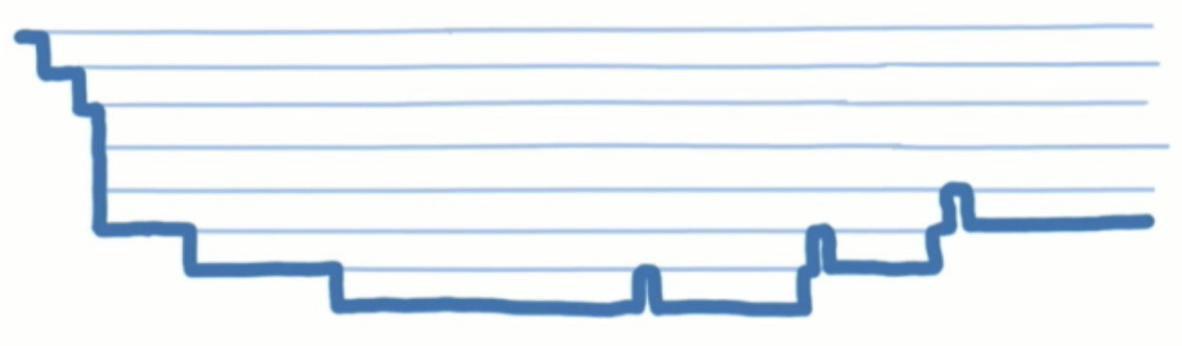
1. 为什么需要fiber

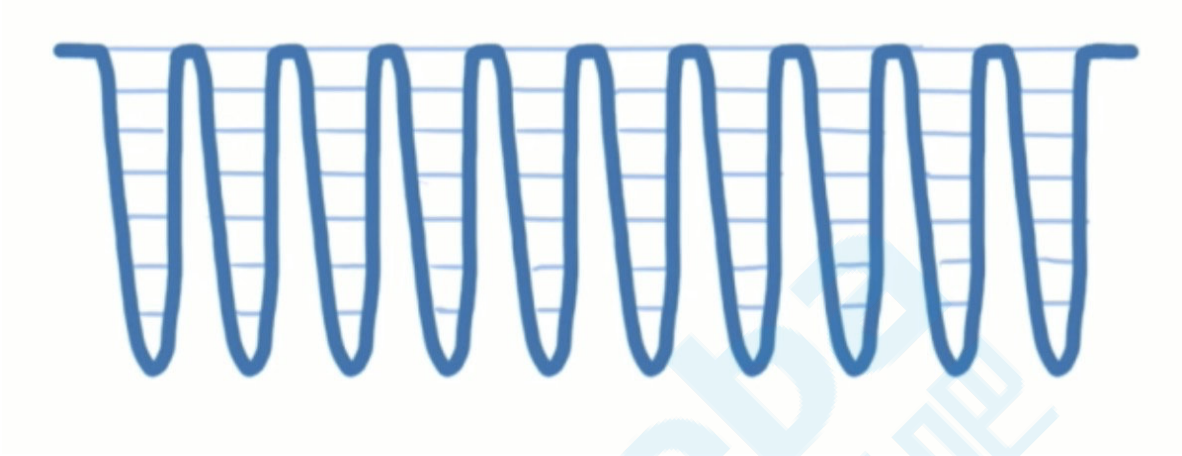
   对于大型项目，组件树会很大，这个时候递归遍历的成本就会很高，会造成主线程被持续占用，结果就是主线程上的布局、动画等周期性任务就无法立即得到处理，造成视觉上的卡顿，影响用户体验。

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**

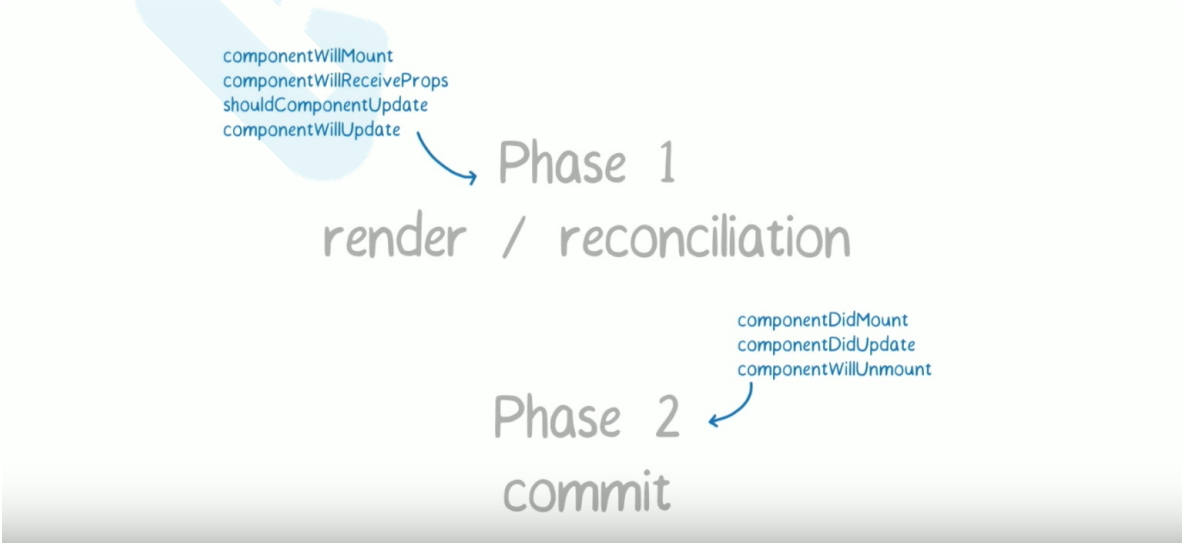




什么是fiber

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。

实现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。

```
import React from "./kreact/";
import ReactDOM, {useState} from "./kreact/react-dom";
import Component from "./kreact/Component";

import "./index.css";

class ClassComponent extends Component {
  static defaultProps = {
    color: "pink"
  };
  render() {
    return (
      <div className="border">
        <div className={this.props.color}>color </div>
        {this.props.name}
      </div>
    );
  }
}

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>
  );
}

const jsx = (
  <div className="box border">
    <p>全栈学习</p>
    <a href="https://zh-hans.reactjs.org/">React学习</a>
    <FunctionComponent name="函数组件" />
    <ClassComponent name="class组件" />

    <>
      <h1>文本1</h1>
      <h2>文本2</h2>
    </>

  </div>
);

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

./react-dom.js

```
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 = [];
}

// vnode->node
// 生成node节点
function createNode(vnode) {
  const {type, props} = vnode;
  let node = null;
  if (type === TEXT) {
    node = document.createTextNode("");
  } else if (typeof type === "string") {
    node = document.createElement(type);
  }
  updateNode(node, {}, props);
  return node;
}

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;
  }
}

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];
      }
    });
}

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

function updateClassComponent(fiber) {
  // 略。。。
}

function performUnitOfWork(fiber) {
  // 1. 执行当前任务
  // 执行当前任务
```

```
  const {type} = fiber;
  if (typeof type === "function") {
    type.isReactComponent
      ? updateClassComponent(fiber)
      : updateFunctionComponent(fiber);
  } else {
    // 原生标签
    updateHostComponent(fiber);
  }

  // 2、 返回下一个任务
  // 原则就是：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.return;
  }
}

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

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) {
    commitRoot();
  }

  requestIdleCallback(workLoop);
}

requestIdleCallback(workLoop);

// ! 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);
  }
}

// !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];
}

export default {
  render
```

```
};
```

const.js

```
export const TEXT = "TEXT";

export const PLACEMENT = "PLACEMENT";
export const UPDATE = "UPDATE";
export const DELETION = "DELETION";
```

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])`
    - 实现fiber
- 回顾
- 作业
- 下节课内容

作业

1. 实现class组件渲染，完成updateClassComponent。

下节课内容

Hook原理。