

React原理解析02

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

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

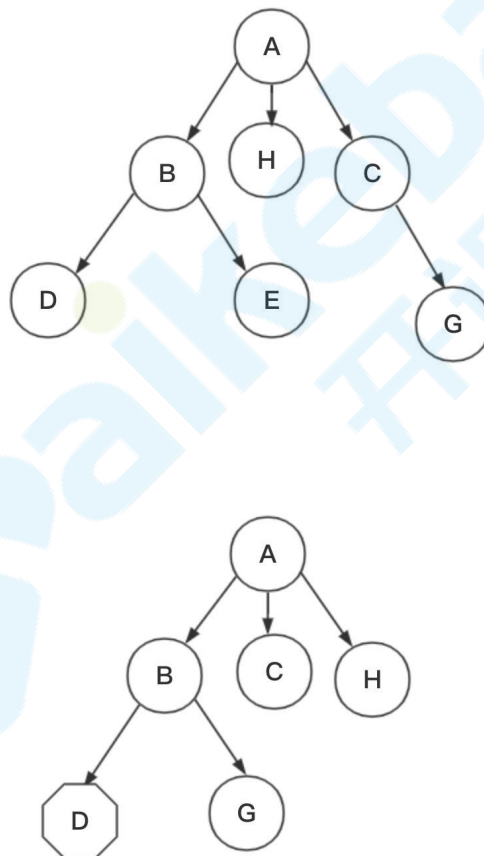
算法复杂度 $O(n)$

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

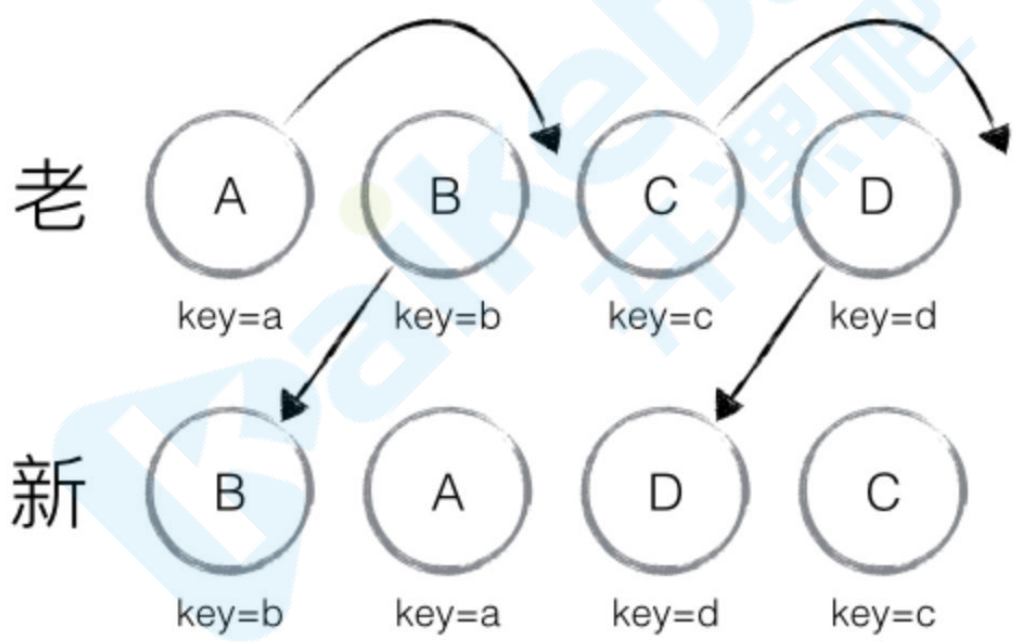
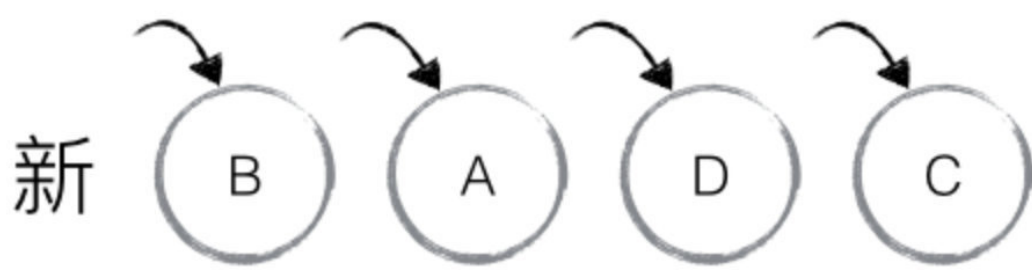
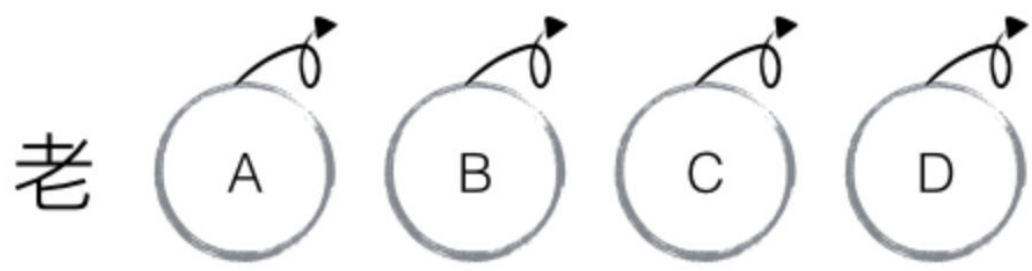
比对两个虚拟dom时会有三种操作：删除、替换和更新

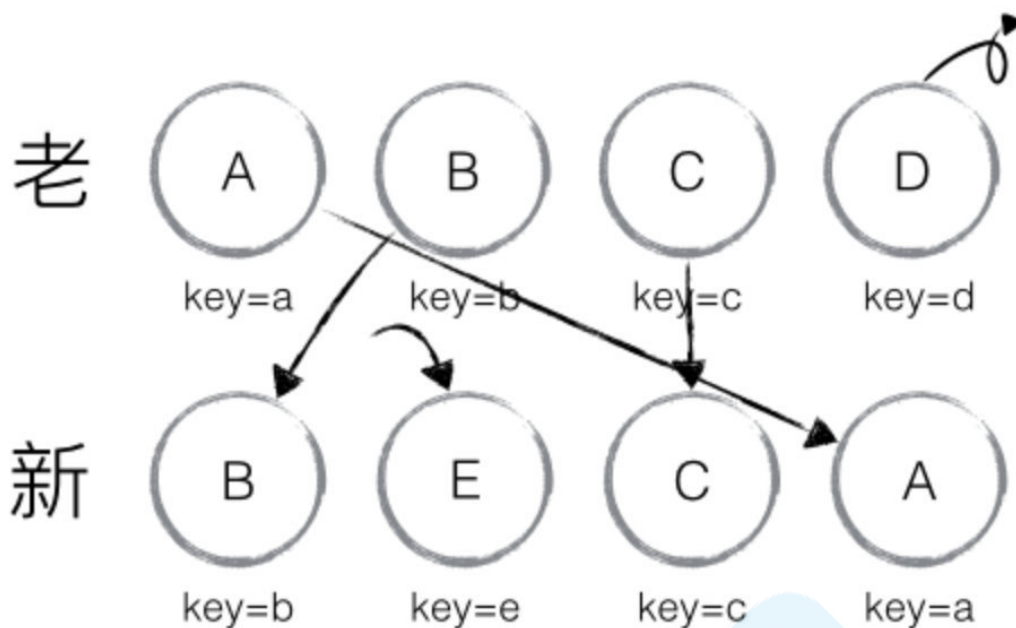
vnode是现在的虚拟dom，newVnode是新虚拟dom。

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





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

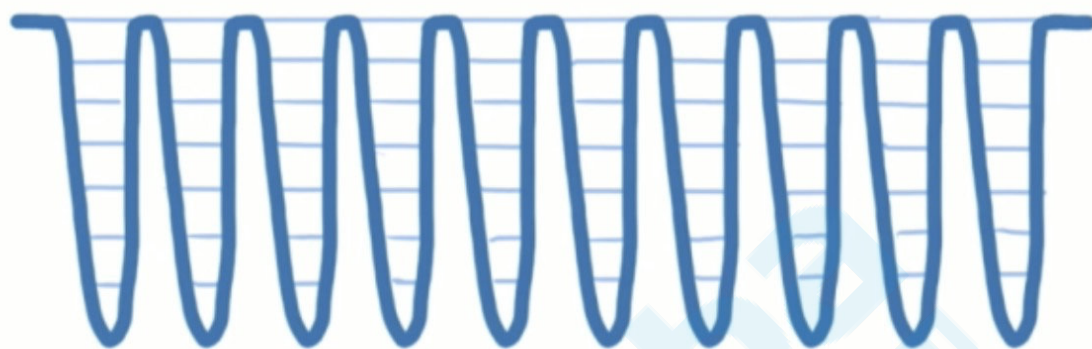
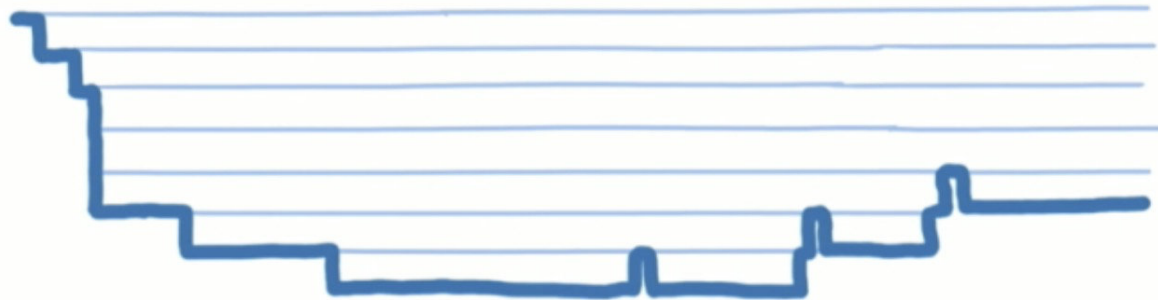
3. 增量渲染（把渲染任务拆分成块，匀到多帧）

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務，每个组件可以一个或者多个。

componentWillMount
componentWillReceiveProps
shouldComponentUpdate
componentWillUpdate

Phase 1

render / reconciliation

componentDidMount
componentDidUpdate
componentWillUnmount

Phase 2

commit

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

import "../index.css";

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

function FunctionComponent(props) {
  return <div className="border">{props.name}</div>;
}

const jsx = (
  <div className="border">
    <p>全栈</p>
    <a href="https://www.kaikeba.com/">开课吧</a>
    <FunctionComponent name="函数组件" />
    <ClassComponent name="类组件" />
  </div>
);

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

web全栈架构师

./react-dom.js

```
// vnode 虚拟dom节点
// node 真实dom节点

// container node是node节点
function render(vnode, container) {
  wipRoot = {
    stateNode: container,
    props: {children: vnode}
  };

  nextUnitOfWork = wipRoot;
}

function createNode(workInProgress) {
  let node = null;
  // todo vnode->node

  const {type, props} = workInProgress;

  if (typeof type === "string") {
    // 原生标签
    node = document.createElement(type);
  }

  updateNode(node, props);

  return node;
}

//原生标签节点处理
function updateHostComponent(workInProgress) {
  if (!workInProgress.stateNode) {
    workInProgress.stateNode = createNode(workInProgress);
  }

  reconcileChildren(workInProgress, workInProgress.props.children);

  console.log("workInProgress", workInProgress); //sy-log
}

// 函数组件
// 执行函数
function updateFunctionComponent(workInProgress) {
  const {type, props} = workInProgress;

  const children = type(props);

  reconcileChildren(workInProgress, children);
}

// 类组件
// 先实例化 再执行render函数
function updateClassComponent(vnode) {

```

```

const {type, props} = vnode;
const instance = new type(props);
const vvnode = instance.render();
const node = createNode(vvnode);
return node;
}

// 更新属性
// todo 加一下属性的具体处理 比如style
function updateNode(node, nextVal) {
  Object.keys(nextVal).forEach(k => {
    if (k === "children") {
      if (typeof nextVal.children === "string") {
        const textNode = document.createTextNode(nextVal.children);
        node.appendChild(textNode);
      }
    } else {
      node[k] = nextVal[k];
    }
  });
}

function reconcileChildren(workInProgress, children) {
  if (
    !(workInProgress.props && typeof workInProgress.props.children !== "string")
  ) {
    return;
  }

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

  let previousNewFiber = null;
  for (let index = 0; index < newChildren.length; index++) {
    const child = newChildren[index];

    let newFiber = null;
    newFiber = {
      type: child.type, //
      props: child.props,
      child: null,
      sibling: null,
      return: workInProgress,
      stateNode: null
    };

    // 构建fiber结构
    if (index === 0) {
      workInProgress.child = newFiber;
    } else {
      previousNewFiber.sibling = newFiber;
    }

    previousNewFiber = newFiber;
  }
}

// fiber 结构
// child 第一个子节点

```



```

// sibling 下一个兄弟
// return 爸爸
// stateNode dom节点
function performUnitOfWork(workInProgress) {
  // * step1: 执行当前fiber (治理王朝)
  // todo 执行
  const {type} = workInProgress;
  if (typeof type === "function") {
    type.prototype.isReactComponent
      ? updateClassComponent(workInProgress)
      : updateFunctionComponent(workInProgress);
  } else {
    // 原生标签
    updateHostComponent(workInProgress);
  }

  // * step2: 返回下一个fiber (王朝的故事)
  // 原则: 先找子节点
  if (workInProgress.child) {
    return workInProgress.child;
  }
  // 没有子节点, 传给兄弟,
  // 没有兄弟,
  let nextFiber = workInProgress;
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling;
    }
    nextFiber = nextFiber.return;
  }
}

// 下一个fiber任务
let nextUnitOfWork = null;
// wip work in progress 正在进行当中的, 数据结构fiber root
let wipRoot = null;

function workLoop(IdleDeadline) {
  while (nextUnitOfWork && IdleDeadline.timeRemaining() > 1) {
    // 执行当前fiber, 返回下一个fiber
    nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
  }

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

requestIdleCallback(workLoop);

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

function commotWorker(workInProgress) {
  if (!workInProgress) {

```

```

    return;
  }
  // step1: commit workInProgress
  let parentNodeFiber = workInProgress.return;
  while (!parentNodeFiber.stateNode) {
    parentNodeFiber = parentNodeFiber.return;
  }

  // 父（祖先）dom节点
  const parentNode = parentNodeFiber.stateNode;
  if (workInProgress.stateNode) {
    parentNode.appendChild(workInProgress.stateNode);
  }

  // step2: commit workInProgress.child
  commitWorker(workInProgress.child);
  // step1: commit workInProgress.sibling
  commitWorker(workInProgress.sibling);
}

export default {render};

```

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，提交updateClassComponent代码截图。

下节课内容

Hook原理。

