React原理解析02

资源

1. React中文网
2. React源码

课堂目标

1. 掌握MessageChannel
2. 掌握React调度策略
3. 掌握hook原理

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。

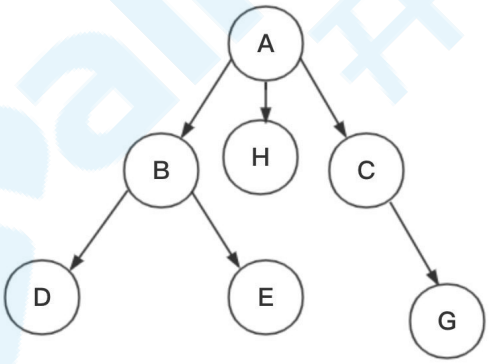
diffing算法

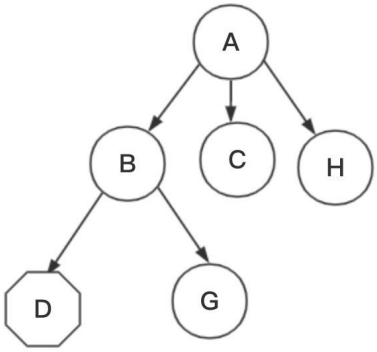
算法复杂度O(n)

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。

前置知识点

Performance.now()

https://developer.mozilla.org/zh-CN/docs/Web/API/Performance/now

和JavaScript中其他可用的时间类函数（比如`Date.now`）不同的是，`window.performance.now()`返回的时间戳没有被限制在一毫秒的精确度内，相反，它们以浮点数的形式表示时间，精度最高可达微秒级。

另外一个不同点是，`window.performance.now()`是以一个恒定的速率慢慢增加的，它不会受到系统时间的影响（系统时钟可能会被手动调整或被NTP等软件篡改）。另外，`performance.timing.navigationStart + performance.now()`约等于`Date.now()`。

MessageChannel

https://developer.mozilla.org/zh-CN/docs/Web/API/MessageChannel

Channel Messaging API的**`MessageChannel`**接口允许我们创建一个新的消息通道，并通过它的两个`MessagePort`属性发送数据。

> **Note:** 此特性在 Web Worker 中可用。

属性

`MessageChannel.port1` 只读

返回channel的port1。

`MessageChannel.port2` 只读

返回channel的port2。

构造函数

`MessageChannel()`

返回一个带有两个MessagePort属性的MessageChannel新对象。

示例：

```
const channel = new MessageChannel();
const {port1, port2} = channel;
port1.onmessage = function(msgEvent) {
  console.log("port1 收到消息: " + msgEvent.data); //sy-log
  port1.postMessage("port2 请相应");
};

port2.onmessage = function(msgEvent) {
  console.log("port2 收到消息: ", msgEvent.data); //sy-log
};

port2.postMessage("port1 请相应");
```

调度

React下有个包叫scheduler，它用于处理浏览器环境中的任务调度，现在只用于了React内部，但是据计划是要做成通用库的。现在开放的公共API还没有完成，还处于开发阶段。

DebugReact > src > react > packages > scheduler > src > forks > SchedulerDOM.js > ...

```
54   // Max 31 bit integer. The max integer size in V8 for 32-bit systems.
55   // Math.pow(2, 30) - 1
56   // 0b111111111111111111111111111111
57   var maxSigned31BitInt = 1073741823;
58
59   // Times out immediately
60   var IMMEDIATE_PRIORITY_TIMEOUT = -1;
61   // Eventually times out
62   var USER_BLOCKING_PRIORITY_TIMEOUT = 250;
63   var NORMAL_PRIORITY_TIMEOUT = 5000;
64   var LOW_PRIORITY_TIMEOUT = 10000;
65   // Never times out
66   var IDLE_PRIORITY_TIMEOUT = maxSigned31BitInt;
67
68   // Tasks are stored on a min heap
69   var taskQueue = [];
70   var timerQueue = [];
71
72   // Incrementing id counter. Used to maintain insertion order.
73   var taskIdCounter = 1;
74
75   // Pausing the scheduler is useful for debugging.
76   var isSchedulerPaused = false;
77
78   var currentTask = null;
79   var currentPriorityLevel = NormalPriority;
80
81   // This is set while performing work, to prevent re-entrancy.
82   var isPerformingWork = false;
83
84   var isHostCallbackScheduled = false;
85   var isHostTimeoutScheduled = false;
86
87   // Capture local references to native APIs, in case a polyfill overrides
88   const setTimeout = window.setTimeout;
89   const clearTimeout = window.clearTimeout;
90   const setImmediate = window.setImmediate; // IE and Node.js + jsdom
```

实现scheduler

```
import {isFn} from "./utils";

const taskQueue = [];
const timerQueue = [];

let deadline = 0;
const threshold = 5;

export function scheduleCallback(callback) {
  const newTask = {callback};
  taskQueue.push(newTask);
  schedule(flushWork);
}
```

```
export function schedule(callback) {
  timerQueue.push(callback);
  postMessage();
}

const postMessage = () => {
  const {port1, port2} = new MessageChannel();
  port1.onmessage = () => {
    let tem = timerQueue.splice(0, timerQueue.length);
    tem.forEach((c) => c());
  };
  port2.postMessage(null);
};

function flushWork() {
  deadline = getCurrentTime() + threshold;
  let currentTask = taskQueue[0];
  while (currentTask && !shouldYield()) {
    const {callback} = currentTask;
    callback();
    taskQueue.shift();
    currentTask = taskQueue[0];
  }
}

export function shouldYield() {
  return getCurrentTime() >= deadline;
}

export function getCurrentTime() {
  return performance.now();
}
```

Hooks原理

```
function FunctionComponent(props) {
  const [count, setCount] = useState(0);
  const [count2, setCount2] = useReducer((x) => x + 1, 0);

  return (
    <div className="border">
      <p>{props.name}</p>
      <p>{count}</p>
      <button
        onClick={() => {
          setCount(count + 1);
        }}>
        click
      </button>

      <p>{count2}</p>
      <button
        onClick={() => {
          setCount2(count2 + 1);
```

```
        }}>
        click
      </button>
    </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useReducer

```
import {scheduleUpdateOnFiber} from "./ReactFiberWorkLoop";

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

export function renderHooks(wip) {
  currentlyRenderingFiber = wip;
  currentlyRenderingFiber.memoizedState = null;
  workInProgressHook = null;
}

// fiber(memoizedState)->hook0(next)->hook1(next)->hook2(next)->null
// workInProgressHook=hook2 当前的hook
function updateWorkInProgressHook() {
  let hook;
  // todo
  const current = currentlyRenderingFiber.alternate;
  if (current) {
    // 不是初次渲染，是更新，意味着可以在老hook基础上更新
    currentlyRenderingFiber.memoizedState = current.memoizedState;
    if (workInProgressHook) {
      // 不是第一个hook
      hook = workInProgressHook = workInProgressHook.next;
    } else {
      // 是第一个hook
      hook = workInProgressHook = current.memoizedState;
    }
  } else {
    // 是初次渲染，需要初始化hook
```

```
    hook = {
      memoizedState: null, //状态值
      next: null, // 指向下一个hook或者null
    };
    if (workInProgressHook) {
      // 不是第一个hook
      workInProgressHook = workInProgressHook.next = hook;
    } else {
      // 是第一个hook
      workInProgressHook = currentlyRenderingFiber.memoizedState = hook;
    }
  }

  return hook;
}

export function useReducer(reducer, initialState) {
  /**
   * memoizedState 状态值
   * next 指向下一个hook
   */
  const hook = updateWorkInProgressHook();

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

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

  return [hook.memoizedState, dispatch];
}
```

fiber.js

```
import {Placement} from "./utils";
/**
 * fiber （vnode）
 * type 类型
 * key 标记当前层级下的唯一性
 * props 属性值
 * child 第一个子节点（fiber）
 * return 父节点（fiber）
 * alternate 老节点
 * sibling 下一个兄弟节点（fiber）
 * flags 标记当前节点类型（比如插入、更新、删除等）
 * stateNode 原生标签时候，指向dom节点，（类组件时候指向实例）
 */

export function createFiber(vnode, returnFiber) {
  const newFiber = {
    type: vnode.type,
```

```
    key: vnode.key,
    props: vnode.props,
    child: null,
    sibling: null,
    return: returnFiber,
    alternate: null,
    flags: Placement,
    stateNode: null,
  };
  return newFiber;
}
```

更新ReactFiberWorkLoop.js

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

// wip work in progress 当前正在工作中的fiber
// 根节点更新
let wipRoot = null;
// 下一个要更新的fiber节点
let nextUnitOfWork = null;

export function scheduleUpdateOnFiber(fiber) {
  fiber.alternate = {...fiber};
  wipRoot = fiber;
  wipRoot.sibling = null;
  nextUnitOfWork = wipRoot;
  scheduleCallback(workLoop);
}

function performUnitOfwWork(wip) {
  // 1. 更新自己
  // 判断节点类型，因为不同的节点更新方式不一样
  const {type} = wip;
  if (isFn(type)) {
    //类组件或者函数组件
    // todo 区分函数组件和类组件
    updateFunctionComponent(wip);
  } else if (isStr(type)) {
    // 原生标签
    updateHostComponent(wip);
  } else {
    updateFragmentComponent(wip);
  }

  // 2. 返回下一个要更新的任务
  // 深度优先遍历（王朝的故事）
```

```
  if (wip.child) {
    return wip.child;
  }
  while (wip) {
    if (wip.sibling) {
      return wip.sibling;
    }
    wip = wip.return;
  }
  return null;
}
function workLoop() {
  while (nextUnitOfWork && !shouldYield()) {
    // 有要更新的fiber任务，并且浏览器有空闲时间
    nextUnitOfWork = performUnitOfwWork(nextUnitOfWork);
  }

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

// requestIdleCallback(workLoop);

// commit
function commitRoot() {
  // 初次渲染
  commitWorker(wipRoot.child);
}

function getParentNode(fiber) {
  let tem = {...fiber};
  while (tem) {
    if (tem.return.stateNode) {
      return tem.return.stateNode;
    }
    tem = tem.return;
  }

  return null;
}

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

  const {stateNode, flags} = wip;
  // parentNode是wip的父或者祖先dom节点
  const parentNode = getParentNode(wip);
  // 1. commit self
  // todo deletion update
  // placement
  if (stateNode && flags & Placement) {
    parentNode.appendChild(wip.stateNode);
  }
```

```
  if (stateNode && flags & Update) {
    //复用vnode和node，
    updateNode(stateNode, wip.alternate.props, wip.props);
  }

  // 2. commit child
  commitWorker(wip.child);
  // 3. commit sibling
  commitWorker(wip.sibling);
}
```

ReactFiberReconciler.js

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

export function updateHostComponent(wip) {
  if (!wip.stateNode) {
    wip.stateNode = document.createElement(wip.type);
    updateNode(wip.stateNode, {}, wip.props);
  }

  reconcileChildren(wip, wip.props.children);
}

//函数组件
export function updateFunctionComponent(wip) {
  renderHooks(wip);
  const {type, props} = wip;
  const child = type(props);
  reconcileChildren(wip, child);
}

export function updateFragmentComponent(wip) {
  reconcileChildren(wip, wip.props.children);
}

function reconcileChildren(returnFiber, children) {
  if (isStringOrNumber(children)) {
    return;
  }

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

  let previousNewFiber = null;
  let oldFiber = returnFiber.alternate && returnFiber.alternate.child;
  for (let i = 0; i < newChildren.length; i++) {
    const child = newChildren[i];
    const newFiber = createFiber(child, returnFiber);
    const same = sameNode(child, oldFiber);
    if (same) {
      // 更新
      Object.assign(newFiber, {
        alternate: oldFiber,
```

```
        stateNode: oldFiber.stateNode,
        flags: Update,
      });
    }
    if (oldFiber) {
      oldFiber = oldFiber.sibling;
    }
    if (previousNewFiber === null) {
      returnFiber.child = newFiber;
    } else {
      previousNewFiber.sibling = newFiber;
    }
    previousNewFiber = newFiber;
  }
}

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

react.js

```
export {useReducer} from "./hooks";
```

属性更新

```
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];
      }
    });
}
```

回顾

作业

1. 补全代码，实现useState。

下节课内容

hook