前端中的算法

资源

开始学习

数据结构与算法

数据结构和算法是教我们写出可以高效运行的代码。

前端中的数据结构与算法

全排序（数组、链表、二叉树、堆）

偏排序（数组、链表、堆）

查找与搜索（二叉树、平衡二叉树（包括红黑树）、哈希表）

动态规划（数组、链表、堆、二叉树）

React中的算法与数据结构：深度优先搜索、递归、**动态规划**、散列表、 数组、链表、二叉树、**堆**、栈等

Vue中的算法与数据结构：**动态规划**、递归、**二分查找**、散列表、LRU(最近最少使用) 、数组等

怎么学习新技术

###

虚拟DOM

React 本身只是一个 DOM 的抽象层，使用组件构建虚拟 DOM。

常见问题：react virtual dom是什么？说一下diff算法?

Virtual DOM 及内核

什么是 Virtual DOM?

Virtual DOM 是一种编程概念。在这个概念里， UI 以一种理想化的，或者说"虚拟的"表现形式被保存于内存中，并通过如 ReactDOM 等类库使之与"真实的" DOM 同步。这一过程叫做协调。

这种方式赋予了 React 声明式的 API：您告诉 React 希望让 UI 是什么状态，React 就确保 DOM 匹配该状态。这使您可以从属性操作、事件处理和手动 DOM 更新这些在构建应用程序时必要的操作中解放出来。

与其将 "Virtual DOM" 视为一种技术，不如说它是一种模式，人们提到它时经常是要表达不同的东西。在 React 的世界里，术语 "Virtual DOM" 通常与 React 元素关联在一起，因为它们都是代表了用户界面的对象。而 React 也使用一个名为 "fibers" 的内部对象来存放组件树的附加信息。上述二者也被认为是 React 中 "Virtual DOM" 实现的一部分。

Shadow DOM 和 Virtual DOM 是一回事吗?

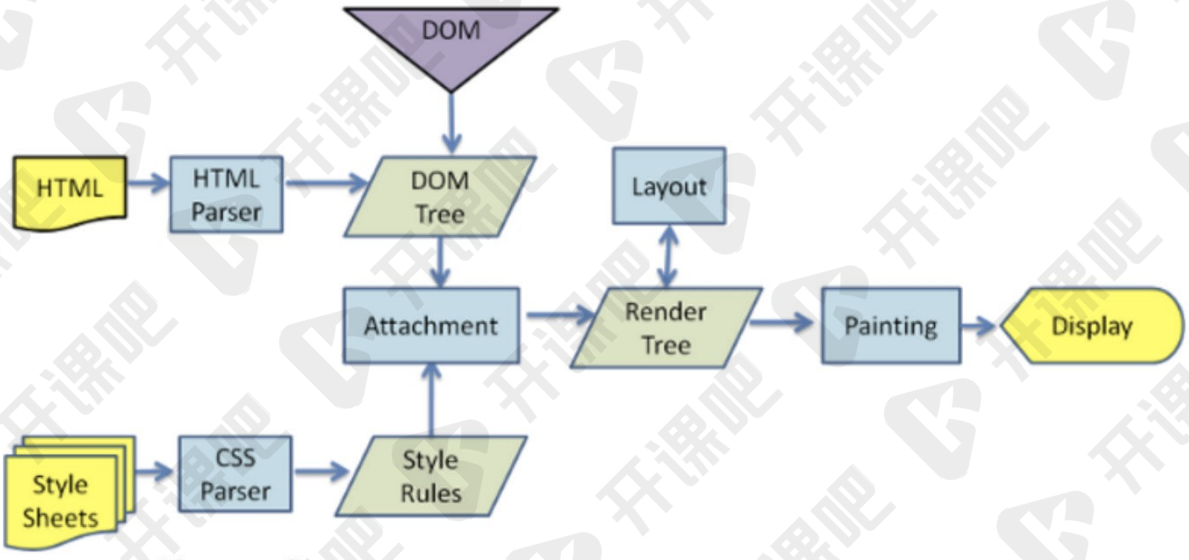
不，他们不一样。Shadow DOM 是一种浏览器技术，主要用于在 web 组件中封装变量和 CSS。Virtual DOM 则是一种由 Javascript 类库基于浏览器 API 实现的概念。

什么是 "React 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。了解更多.

what? 用 JavaScript 对象表示 DOM 信息和结构，当状态变更的时候，重新渲染这个 JavaScript 的对象结构。这个 JavaScript 对象称为virtual dom；

传统dom渲染流程




```
var div = document.createElement('div');
var str = '';
for(var key  in div){
    str += ' ' + key ;
}
console.log(str);
```

```
 align title lang translate dir hidden accessKey draggable spellcheck autocapitalize  VM23717:6
contentEditable isContentEditable inputMode offsetParent offsetTop offsetLeft offsetWidth
offsetHeight style innerText outerText oncopy oncut onpaste onabort onblur oncancel oncanplay
oncanplaythrough onchange onclick onclose oncontextmenu oncuechange ondblclick ondrag ondragend
ondragenter ondragleave ondragover ondragstart ondrop ondurationchange onemptied onended onerror
onfocus oninput oninvalid onkeydown onkeypress onkeyup onload onloadeddata onloadedmetadata
onloadstart onmousedown onmouseenter onmouseleave onmousemove onmouseout onmouseover onmouseup
onmousewheel onpause onplay onplaying onprogress onratechange onreset onresize onscroll onseeked
onseeking onselect onstalled onsubmit onsuspend ontimeupdate ontoggle onvolumechange onwaiting
onwheel onauxclick ongotpointercapture onlostpointercapture onpointerdown onpointermove
onpointerup onpointercancel onpointerover onpointerout onpointerenter onpointerleave
onselectstart onselectionchange dataset nonce tabIndex click focus blur enterKeyHint onformdata
onpointerrawupdate attachInternals namespaceURI prefix localName tagName id className classList
slot part attributes shadowRoot assignedSlot innerHTML outerHTML scrollTop scrollLeft
scrollWidth scrollHeight clientTop clientLeft clientWidth clientHeight attributeStyleMap
onbeforecopy onbeforecut onbeforepaste onsearch previousElementSibling nextElementSibling
children firstElementChild lastElementChild childElementCount onfullscreenchange
onfullscreenerror onwebkitfullscreenchange onwebkitfullscreenerror setPointerCapture
releasePointerCapture hasPointerCapture hasAttributes getAttributeNames getAttribute
getAttributeNS setAttribute setAttributeNS removeAttribute removeAttributeNS hasAttribute
hasAttributeNS toggleAttribute getAttributeNode getAttributeNodeNS setAttributeNode
setAttributeNodeNS removeAttributeNode closest matches webkitMatchesSelector attachShadow
getElementsByTagName getElementsByTagNameNS getElementsByClassName insertAdjacentElement
insertAdjacentText insertAdjacentHTML requestPointerLock getClientRects getBoundingClientRect
scrollIntoView scroll scrollTo scrollBy scrollIntoViewIfNeeded animate computedStyleMap before
after replaceWith remove prepend append querySelector querySelectorAll requestFullscreen
webkitRequestFullScreen webkitRequestFullscreen createShadowRoot getDestinationInsertionPoints
elementTiming ELEMENT_NODE ATTRIBUTE_NODE TEXT_NODE CDATA_SECTION_NODE ENTITY_REFERENCE_NODE
ENTITY_NODE PROCESSING_INSTRUCTION_NODE COMMENT_NODE DOCUMENT_NODE DOCUMENT_TYPE_NODE
DOCUMENT_FRAGMENT_NODE NOTATION_NODE DOCUMENT_POSITION_DISCONNECTED DOCUMENT_POSITION_PRECEDING
DOCUMENT_POSITION_FOLLOWING DOCUMENT_POSITION_CONTAINS DOCUMENT_POSITION_CONTAINED_BY
DOCUMENT_POSITION_IMPLEMENTATION_SPECIFIC nodeType nodeName baseURI isConnected ownerDocument
parentNode parentElement childNodes firstChild lastChild previousSibling nextSibling nodeValue
textContent hasChildNodes getRootNode normalize cloneNode isEqualNode isSameNode
compareDocumentPosition contains lookupPrefix lookupNamespaceURI isDefaultNamespace insertBefore
appendChild replaceChild removeChild addEventListener removeEventListener dispatchEvent
```

why? DOM操作很慢，轻微的操作都可能导致页面重新排版，非常耗性能。相对于DOM对象，js对象处理起来更快，而且更简单。通过diff算法对比新旧vdom之间的差异，可以批量的、最小化的执行dom操作，从而提升用户体验。

where? React中用JSX语法描述视图(View)，~~通过babel-loader转译后它们变为React.createElement(...)形式~~，该函数将生成vdom来描述真实dom。将来如果状态变化，vdom将作出相应变化，~~再~~通过diff算法对比新老vdom区别从而做出最终dom操作。

how?

JSX

在线尝试

1. 什么是JSX

   语法糖

   React 使用 JSX 来替代常规的 JavaScript。

   JSX 是一个看起来很像 XML 的 JavaScript 语法扩展。

2. 为什么需要JSX
   - 开发效率：使用 JSX 编写模板简单快速。
   - 执行效率：JSX编译为 JavaScript 代码后进行了优化，执行更快。
   - 类型安全：在编译过程中就能发现错误。
3. 与vue的异同：
   - react中虚拟dom+jsx的设计一开始就有，vue则是演进过程中才出现的
   - jsx本来就是js扩展，转义过程简单直接的多；vue把template编译为render函数的过程需要复杂的编译器转换字符串-ast-js函数字符串

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。

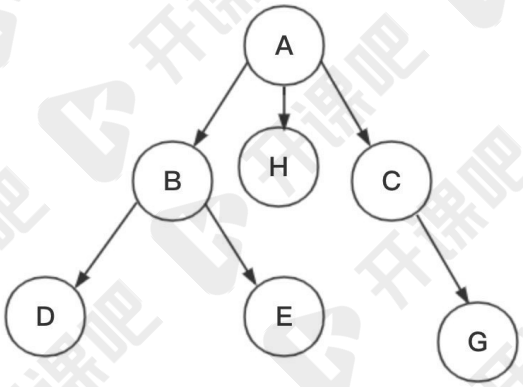
diffing算法

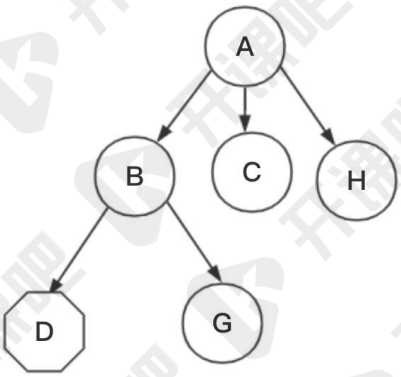
算法复杂度O(n)

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

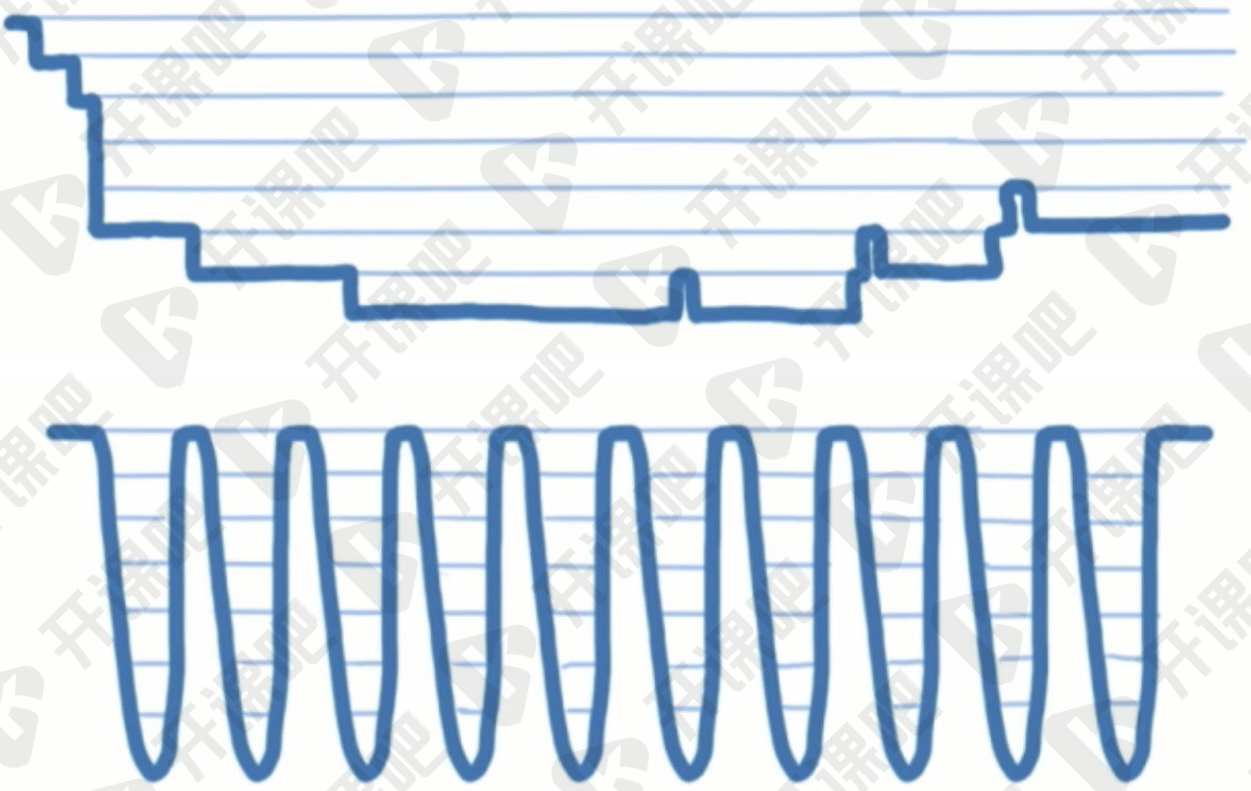
1. 为什么需要fiber

   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

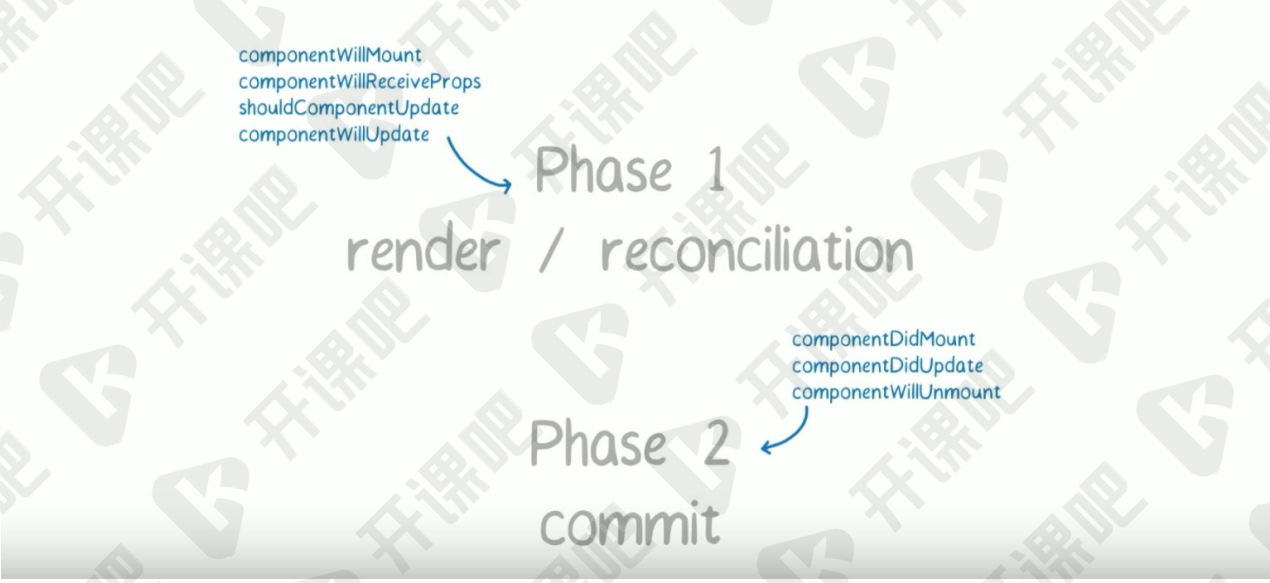
3. 增量渲染（把渲染任务拆分成块，匀到多帧）
4. 更新时能够暂停，终止，复用渲染任务
5. 给不同类型的更新赋予**优先级**
6. 并发方面新的基础能力
7. **更流畅**



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务，每个组件可以一个或者多个。



组件类型

- 文本节点
- HTML标签节点
- 函数组件
- 类组件
- 等等

src/react/packages/react-reconciler/src/ReactWorkTags.js

```
export const FunctionComponent = 0;
export const ClassComponent = 1;
export const IndeterminateComponent = 2; // Before we know whether it is
function or class
export const HostRoot = 3; // Root of a host tree. Could be nested inside
another node.
export const HostPortal = 4; // A subtree. Could be an entry point to a
different renderer.
export const HostComponent = 5;
export const HostText = 6;
```

```
export const Fragment = 7;
export const Mode = 8;
export const ContextConsumer = 9;
export const ContextProvider = 10;
export const ForwardRef = 11;
export const Profiler = 12;
export const SuspenseComponent = 13;
export const MemoComponent = 14;
export const SimpleMemoComponent = 15;
export const LazyComponent = 16;
export const IncompleteClassComponent = 17;
export const DehydratedFragment = 18;
export const SuspenseListComponent = 19;
export const ScopeComponent = 21;
export const OffscreenComponent = 22;
export const LegacyHiddenComponent = 23;
export const CacheComponent = 24;
```

Debug React

https://github.com/bubucuo/DebugReact

关注公众号bubucuo，回复“3”获取Debug React压缩包

fiber结构

图：https://www.processon.com/view/link/5dea52b9e4b079080a22a846

生成fiber

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

export default function createFiber(vnode, returnFiber) {
  const fiber = {
    type: vnode.type,
    key: vnode.key,
    props: vnode.props,
    stateNode: null, // 原生标签时候指dom节点，类组件时候指的是实例
    child: null, // 第一个子fiber
    sibling: null, // 下一个兄弟fiber
    return: returnFiber, // 父fiber
    // 标记节点是什么类型的
    flags: Placement,
    // 老节点
```

```
    alternate: null,
    deletions: null, // 要删除子节点 null或者[]
    index: null, //当前层级下的下标，从0开始
  };

  return fiber;
}
```

执行任务

原则：深度优先遍历（王朝的故事）

```
let wip = null;

function performUnitOfWork() {
  // todo 1. 执行当前任务wip
  // 判断wip是什么类型的组件
  const { type } = wip;
  if (isStr(type)) {
    // 原生标签
    updateHostComponent(wip);
  } else if (isFn(type)) {
    type.prototype.isReactComponent
      ? updateClassComponent(wip)
      : updateFunctionComponent(wip);
  }
  // todo 2. 更新wip
  // 深度优先遍历（王朝的故事）
  if (wip.child) {
    wip = wip.child;
    return;
  }
  let next = wip;
  while (next) {
    if (next.sibling) {
      wip = next.sibling;
      return;
    }
    next = next.return;
  }
  wip = null;
}
```

工具函数

flags定义为二进制，而不是字符串或者单个数字，一方面原因是因为二进制单个数字具有唯一性、某个范围内的组合同样具有唯一性，另一方原因在于简洁方便、且速度快。

```
// ! flags
export const NoFlags = /*                      */ 0b0000000000000000000000;

export const Placement = /*                    */ 0b0000000000000000000010; //
2
export const Update = /*                       */ 0b0000000000000000000100; //
4
export const Deletion = /*                     */ 0b0000000000000000001000; //
8

export function isStr(s) {
  return typeof s === "string";
}

export function isStringOrNumber(s) {
  return typeof s === "string" || typeof s === "number";
}

export function isFn(fn) {
  return typeof fn === "function";
}

export function isArray(arr) {
  return Array.isArray(arr);
}
```

位运算

fiber的flags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.flags为132，那这个时候，workInProgress.effectTag & Update 和workInProgress.flags & 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.flags |= 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。

ReactDOM.createRoot替换ReactDOM.render

React18中将会使用最新的ReactDOM.createRoot作为根渲染函数，ReactDOM.render作为兼容，依然会存在，但是会成为遗留模式，开发环境下会出现warning。

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

实现ReactDOM.createRoot

```
import createFiber from "./createFiber";
// work in progress; 当前正在工作中的
import {scheduleUpdateOnFiber} from "./ReactFiberWorkLoop";

function ReactDOMRoot(internalRoot) {
  this._internalRoot = internalRoot;
}

ReactDOMRoot.prototype.render = function(children) {
  const root = this._internalRoot;

  updateContainer(children, root);
};

function createRoot(container) {
  const root = {
    containerInfo: container,
  };

  return new ReactDOMRoot(root);
}

function updateContainer(element, container) {
  const {containerInfo} = container;
  const fiber = createFiber(element, {
    type: containerInfo.nodeName.toLowerCase(),
    stateNode: containerInfo,
  });
  scheduleUpdateOnFiber(fiber);
}

// function render(element, container) {
//   updateContainer(element, {containerInfo: container});
// }

export default {
  // render,
```

```
  createRoot,
};
```