React原理解析01

React原理解析01
- 资源
- 课堂目标
- 知识点
  - 虚拟dom
  - JSX
  - React核心api
  - ReactDOM
    - `render()`
  - 实现React.createElement, ReactDom.render, Component
    - CreateElement
    - ReactDOM.render
    - Component

资源

1. React中文网
2. React源码

课堂目标

1. 深入掌握虚拟dom
2. 掌握createElement、render、Component三个基础核心api

知识点

React 本身只是一个 DOM 的抽象层，使用组件构建虚拟 DOM。

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?

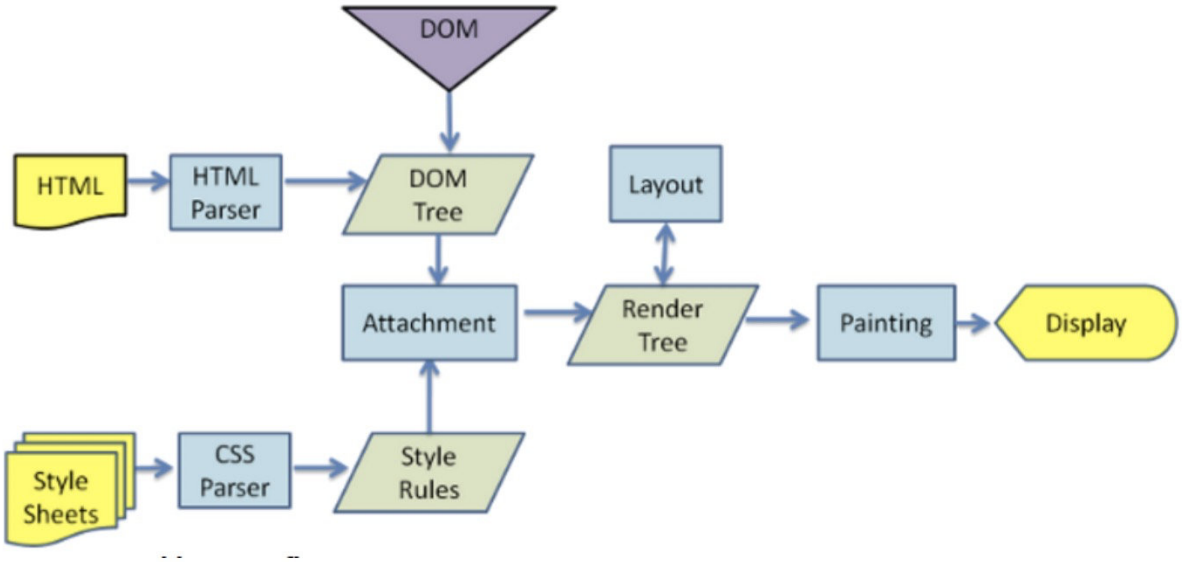
不，他们不一样。Shadow DOM 是一种浏览器技术，主要用于在 web 组件中封装变量和 CSS。Virtual DOM 则是一种由 Javascript 类库基于浏览器 API 实现的概念。

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，从而提高性能。

where? React中用JSX语法描述视图，~~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. React 16原理：babel-loader会预编译JSX为React.createElement(...)
4. React 17原理：React 17中的 JSX 转换**不会将 JSX 转换为 React.createElement**，而是自动从 React 的 package 中引入新的入口函数并调用。另外此次升级不会改变 JSX 语法，旧的 JSX 转换也将继续工作。
5. 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

JSX预处理前：

LIVE JSX EDITOR ☑ JSX?

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div class="app">
        Hello {this.props.name}
      </div>
    );
  }
}

ReactDOM.render(
  <HelloMessage name="Taylor" />,
  document.getElementById('hello-example')
);
```

JSX预处理后：

LIVE JSX EDITOR JSX?

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      { "class": "app" },
      "Hello ",
      this.props.name
    );
  }
}

ReactDOM.render(React.createElement(HelloMessage, { name:
"Taylor" }), document.getElementById('hello-example'));
```

使用自定义组件的情况:

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";

class ClassCmp extends React.Component {
  render() {
    return (
      <div className='app'>
        Hello {this.props.name}
      </div>
    );
  }
}

function FuncCmp(props) {
  return <div>name: {props.name}</div>;
}

const jsx = (
  <div>
    <p>我是内容</p>
    <FuncCmp name="我是function组件" />
    <ClassCmp name="我是class组件" />
  </div>
);
```

```
ReactDOM.render(
  jsx,
  document.getElementById('hello-example')
);
```

build后

```
class ClassCmp extends React.Component {
  render() {
    return React.createElement(
      "div",
      { "class": "app" },
      "Hello ",
      this.props.name
    );
  }
}

function FuncCmp(props) {
  return React.createElement(
    "div",
    null,
    "name: ",
    props.name
  );
}

const jsx = React.createElement(
  "div",
  null,
  React.createElement(
    "p",
    null,
    "我是内容"
  ),
  React.createElement(FuncCmp, { name: "我是function组件" }),
  React.createElement(ClassCmp, { name: "我是class组件" })
);

ReactDOM.render(jsx, document.getElementById('hello-example'));
```

React核心api

react

```
const React = {
  Children: {
    map,
    forEach,
    count,
    toArray,
    only,
  },

  createRef,
  Component,
  PureComponent,

  createContext,
  forwardRef,
  lazy,
  memo,

  useCallback,
  useContext,
  useEffect,
  useImperativeHandle,
  useDebugValue,
  useLayoutEffect,
  useMemo,
  useReducer,
  useRef,
  useState,

  Fragment: REACT_FRAGMENT_TYPE,
  Profiler: REACT_PROFILER_TYPE,
  StrictMode: REACT_STRICT_MODE_TYPE,
  Suspense: REACT_SUSPENSE_TYPE,

  createElement: __DEV__ ? createElementWithValidation : createElement,
  cloneElement: __DEV__ ? cloneElementWithValidation : cloneElement,
  createFactory: __DEV__ ? createFactoryWithValidation : createFactory,
  isValidElement: isValidElement,

  version: ReactVersion,

  __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED: ReactSharedInternals,
};
```

核心精简后：

```
const React = {
    createElement,
    Component
}
```

最核心的api：

React.createElement：创建虚拟DOM

React.Component：实现自定义组件

ReactDOM.render：渲染真实DOM

ReactDOM

`render()`

```
ReactDOM.render(element, container[, callback])
```

当首次调用时，容器节点里的所有 DOM 元素都会被替换，后续的调用则会使用 React 的 DOM 差分算法（DOM diffing algorithm）进行高效的更新。

如果提供了可选的回调函数，该回调将在组件被渲染或更新之后被执行。

实现React.createElement, ReactDom.render, Component

CreateElement

将传入的节点定义转换为vdom。

注意节点类型：

- 文本节点
- HTML标签节点
- function组件
- class组件
- fragment
- 其他如portal等节点

DebugReact > src > react > packages > shared > 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 FundamentalComponent = 20;
export const ScopeComponent = 21;
export const Block = 22;
```

src/index.js

```
// 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">
        <p>{this.props.name}</p>
      </div>
    );
  }
}

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

    {/* <ul>
      {[1, 2, 3].map(item => (
        <React.Fragment key={item}>
          <li>111</li>
          <li>222</li>
        </React.Fragment>
      ))}
    </ul> */}

    {/* <>
      <h1>111</h1>
      <h1>222</h1>
    </> */}
  </div>
);

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

// 文本标签 done
// 原生标签 done
// 函数组件 done
// 类组件 done
// Fragment
// 逻辑组件 Provider Consumer
```

createElement方法

```
import {TEXT} from "./const";

function createElement(type, config, ...children) {
  if (config) {
    delete config.__self;
    delete config.__source;
  }
  // ! 源码中做了详细处理，比如过滤掉key、ref等
  const props = {
    ...config,
    children: children.map(child =>
      typeof child === "object" ? child : createTextNode(child)
```

```
    )
  };

  return {
    type,
    props
  };
}

function createTextNode(text) {
  return {
    type: TEXT,
    props: {
      children: [],
      nodeValue: text
    }
  };
}

export default {
  createElement,
};
```

- 修改index.js实际引入kkreact，测试

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

> createElement被调用时会传入标签类型type，标签属性props及若干子元素children

> index.js中从未使用React类或者其任何接口，为何需要导入它?
>
> JSX编译后实际调用React.createElement方法，所以只要出现JSX的文件中都需要导入React

ReactDOM.render

```
// vnode 虚拟dom节点
// node 真实dom节点

// container  node是node节点
function render(vnode, container) {
  console.log("vnode", vnode, container); //sy-log

  // step1 : vnode->node
  const node = createNode(vnode);
  //step2: container.appnedChild(node)
  container.appendChild(node);
}
```

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

  const {type} = vnode;
  if (typeof type === "string") {
    // 原生标签
    node = updateHostComponent(vnode);
  } else if (typeof type === "function") {
    // 函数组件或者类组件
    node = type.prototype.isReactComponent
      ? updateClassComponent(vnode)
      : updateFunctionComponent(vnode);
  }

  return node;
}

//原生标签节点处理
function updateHostComponent(vnode) {
  const {type, props} = vnode;
  let node = document.createElement(type);

  if (typeof props.children === "string") {
    let childText = document.createTextNode(props.children);
    node.appendChild(childText);
  } else {
    reconcileChildren(props.children, node);
  }

  updateNode(node, props);
  return node;
}

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

  const vvnode = type(props);

  const node = createNode(vvnode);
  return node;
}

// 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)
    .filter(k => k !== "children")
    .forEach(k => {
      node[k] = nextVal[k];
    });
}

// vnode->node ,插入到dom节点里
function reconcileChildren(children, node) {
  if (Array.isArray(children)) {
    for (let index = 0; index < children.length; index++) {
      const child = children[index];
      render(child, node);
    }
  } else {
    render(children, node);
  }
}

export default {render};
```

Component

```
class Component {
  static isReactComponent = {};
  constructor(props) {
    this.props = props;
  }
}

// function Component(props) {
//   this.props = props;
// }

// Component.prototype.isReactComponent = {};

export default Component;
```