React原理剖析

课堂主题

深入理解React原理

资源

1. React中文网
2. React源码

课堂目标

1. 手写createElement/Component/render三个核心api
2. 深入理解setState原理
3. 深入理解虚拟dom

知识点

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,
  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,

  unstable_ConcurrentMode: REACT_CONCURRENT_MODE_TYPE,
  unstable_Profiler: REACT_PROFILER_TYPE,
```

```
  __SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED:
ReactSharedInternals,
};

// Note: some APIs are added with feature flags.
// Make sure that stable builds for open source
// don't modify the React object to avoid deopts.
// Also let's not expose their names in stable builds.

if (enableStableConcurrentModeAPIs) {
  React.ConcurrentMode = REACT_CONCURRENT_MODE_TYPE;
  React.Profiler = REACT_PROFILER_TYPE;
  React.unstable_ConcurrentMode = undefined;
  React.unstable_Profiler = undefined;
}

export default React;
```

核心精简后：

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

react-dom 主要是render逻辑

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

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. 原理：babel-loader会预编译JSX为React.createElement(...)

4. 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：

REACT 编辑器 显示JSX

```
class App extends React.Component {
  render() {
    return (
      <div>
        Hello {this.props.name}, I am {2 + 2} years old
      </div>
    )
  }
}

ReactDOM.render(
  <App name="React" />,
  mountNode
)
```

JSX预处理后：

REACT 编辑器 显示JSX

```
class App extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name,
      ", I am ",
      2 + 2,
      " years old"
    )
  }
}

ReactDOM.render(React.createElement(App, { name: "React" }),
mountNode)
```

使用自定义组件的情况:

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

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

class ClassCmp extends Component {
  render() {
    return <div>name: {this.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>
);

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

build后

```
function FuncCmp(props) {
  return React.createElement(
    "div",
    null,
    "name: ",
    props.name
  );
}
class ClassCmp extends React.Component {
  render() {
```

```
    return React.createElement(
      "div",
      null,
      "name: ",
      this.props.name
    );
  }
}
let jsx = React.createElement(
  "div",
  null,
  "    ",
  React.createElement(
    "div",
    { className: "border" },
    "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('root'));
```

实现三大接口：React.createElement, React.Component, ReactDom.render

CreateElement

将传入的节点定义转换为vdom。

src/index.js

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

import React from "./kreact/";
import ReactDOM from "./kreact/ReactDOM";

import "./index.css";

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

class ClassCmp extends React.Component {
  constructor(props) {
    super(props);
    this.state = { counter: 0 };
  }
  clickHandle = () => {
    console.log("clickHandle");
  };
  render() {
    const { counter } = this.state;
    return (
      <div>
        name: {this.props.name}
        <p>counter: {counter}</p>
        <button onCclick={this.clickHandle}>点击</button>
        {[0, 1, 2].map(item => {
          return <FuncCmp name={"我是function组件" + item}
key={item} />;
        })}
      </div>
    );
  }
}

let jsx = (
```

```
  <div>
    <div className="border">我是内容</div>
    <FuncCmp name="我是function组件" />
    <ClassCmp name="我是class组件" />
  </div>
);
ReactDOM.render(jsx, document.getElementById("root"));
// 0. 文本 1. 原生的 2. function组件 3 class组件
```

- 创建./src/kkreact/index.js，它需要包含createElement方法

```
import { Component } from "./Component";

function createElement(type, props, ...children) {
  props.children = children;
  //判断组件类型
  let vtype;
  if (typeof type === "string") {
    // 原生标签
    vtype = 1;
  } else if (typeof type === "function") {
    // 类组件 函数组件
    vtype = type.isReactComponent ? 3 : 2;
  }
  return {
    vtype,
    type,
    props,
  };
}

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

```
export default React;
```

- 修改index.js实际引入kreact，测试

```
import React from "./kreact/";
```

> 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类

render

- 创建react-dom.js
- 需要实现一个render函数，能够将vdom渲染出来，这里先打印vdom结构

```
import { mount } from "./virtual-dom";

function render(vnode, container) {
  //vnode->node
  mount(vnode, container);
  // container.appendChild(node)
  console.log("render", vnode);
}

export default {
  render,
};
```

实现Component

要实现class组件，需要添加Component类，kreact.js

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

> 浅层封装，setState现在只是一个占位符

组件类型判断

传递给createElememnt的组件有三种组件类型，1： dom组件， 2. class组件， 3. 函数组件，使用vtype属性标识

转换vdom为真实dom

```
export function mount(vnode, container) {
  const { vtype } = vnode;

  //创建文本节点
  if (!vtype) {
    mountText(vnode, container);
  }

  // 创建原生节点
  if (vtype === 1) {
    mountHtml(vnode, container);
  }

  // 创建函数组件
  if (vtype === 2) {
    mountFunc(vnode, container);
  }
```

```
  //创建类组件
  if (vtype === 3) {
    mountClass(vnode, container);
  }
}

//创建文本节点
function mountText(vnode, container) {
  const textNode = document.createTextNode(vnode);
  container.appendChild(textNode);
}

// 创建原生节点
function mountHtml(vnode, container) {
  const { type, props } = vnode;
  const node = document.createElement(type);
  const { children, ...rest } = props;
  Object.keys(rest).map(item => {
    if (item === "className") {
      node.setAttribute("class", rest[item]);
    } else if (item.slice(0, 2) === "on") {
      node.addEventListener("click", rest[item]);
    }
  });
  children.map(item => {
    if (Array.isArray(item)) {
      item.map(c => {
        mount(c, node);
      });
    } else {
      mount(item, node);
    }
  });

  container.appendChild(node);
}
```

```
// 创建函数组件
function mountFunc(vnode, container) {
  const { type, props } = vnode;
  const node = type(props);
  mount(node, container);
}

//创建类组件
function mountClass(vnode, container) {
  const { type, props } = vnode;
  const cmp = new type(props);
  const node = cmp.render();
  mount(node, container);
}
```

执行渲染，kreact-dom.js

```
import { mount } from "./virtual-dom";

function render(vnode, container) {
  //vnode->node
  mount(vnode, container);
  // container.appendChild(node)
  console.log("render", vnode);
}

export default {
  render,
};
```

总结：

1. webpack+babel编译时，替换JSX为React.createElement(type,props,...children)

2. 所有React.createElement()执行结束后得到一个JS对象即vdom，它能够完整描述dom结构
3. ReactDOM.render(vdom, container)可以将vdom转换为dom并追加到container中
4. 实际上，转换过程需要经过一个diff过程，比对出实际更新补丁操作dom

setState

class组件的特点，就是拥有特殊状态并且可以通过setState更新状态并重新渲染视图，是React中最重要的api。

问题：

1. setState为什么是异步

```
// 批量
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 2 });
console.log("counter", this.state);
//数组传递，覆盖原来的state
this.setState([{ counter: 100, msg: "omg" }], () => {
    console.log("clickHandle", this.state);
});
//回调
this.setState({ counter: this.state.counter + 1 },()=>{});
this.setState(nextState => {
   console.log("next", nextState);
});

// 异步
this.setState({ counter: this.state.counter + 1 });
console.log("counter", this.state);//0

// 不异步
setTimeout(()=>{
```

```
    setState({foo: 'bar'})
},1000)
// 原生事件
dom.addEventListener('click',()=>{
    setState({foo: 'bar'})
})
```

setState并没有直接操作去渲染，而是执行了一个异步的updater队列 我们使用一个类来专门管理，./kkreact/Component.js

```
export let updateQueue = {
  updaters: [],
  isPending: false,
  add(updater) {
    _.addItem(this.updaters, updater)
  },
  batchUpdate() {
    if (this.isPending) {
      return
    }
    this.isPending = true
    /*
     each updater.update may add new updater to updateQueue
     clear them with a loop
     event bubbles from bottom-level to top-level
     reverse the updater order can merge some props and
state and reduce the refresh times
     see Updater.update method below to know why
    */
    let { updaters } = this
    let updater
    while (updater = updaters.pop()) {
      updater.updateComponent()
    }
    this.isPending = false
  }
}
```

```
function Updater(instance) {
  this.instance = instance
  this.pendingStates = []
  this.pendingCallbacks = []
  this.isPending = false
  this.nextProps = this.nextContext = null
  this.clearCallbacks = this.clearCallbacks.bind(this)
}

Updater.prototype = {
  emitUpdate(nextProps, nextContext) {
    this.nextProps = nextProps
    this.nextContext = nextContext
    // receive nextProps!! should update immediately
    nextProps || !updateQueue.isPending
    ? this.updateComponent()
    : updateQueue.add(this)
  },
  updateComponent() {
    let { instance, pendingStates, nextProps, nextContext }
= this
    if (nextProps || pendingStates.length > 0) {
      nextProps = nextProps || instance.props
      nextContext = nextContext || instance.context
      this.nextProps = this.nextContext = null
      // merge the nextProps and nextState and update by
one time
      shouldUpdate(instance, nextProps, this.getState(),
nextContext, this.clearCallbacks)
    }
  },
  addState(nextState) {
    if (nextState) {
      _.addItem(this.pendingStates, nextState)
      if (!this.isPending) {
        this.emitUpdate()
```

```
      }
    }
  },
  replaceState(nextState) {
    let { pendingStates } = this
    pendingStates.pop()
    // push special params to point out should replace
state
    _.addItem(pendingStates, [nextState])
  },
  getState() {
    let { instance, pendingStates } = this
    let { state, props } = instance
    if (pendingStates.length) {
      state = _.extend({}, state)
      pendingStates.forEach(nextState => {
        let isReplace = _.isArr(nextState)
        if (isReplace) {
          nextState = nextState[0]
        }
        if (_.isFn(nextState)) {
          nextState = nextState.call(instance, state,
props)
        }
        // replace state
        if (isReplace) {
          state = _.extend({}, nextState)
        } else {
          _.extend(state, nextState)
        }
      })
      pendingStates.length = 0
    }
    return state
  },
  clearCallbacks() {
    let { pendingCallbacks, instance } = this
```

```
    if (pendingCallbacks.length > 0) {
      this.pendingCallbacks = []
      pendingCallbacks.forEach(callback =>
callback.call(instance))
    }
  },
  addCallback(callback) {
    if (_.isFn(callback)) {
      _.addItem(this.pendingCallbacks, callback)
    }
  }
}
```

2. 为什么 setState只有在React合成事件和生命周期数中是异步的，在原生事件和setTimeout、setInterval、addEventListener中都是同步的?

原生事件绑定不会通过合成事件的方式处理，自然也不会进入更新事务的处理流程。setTimeout也一样，在setTimeout回调执行时已经完成了原更新组件流程，也不会再进入异步更新流程，其结果自然就是是同步的了。

setState总结：

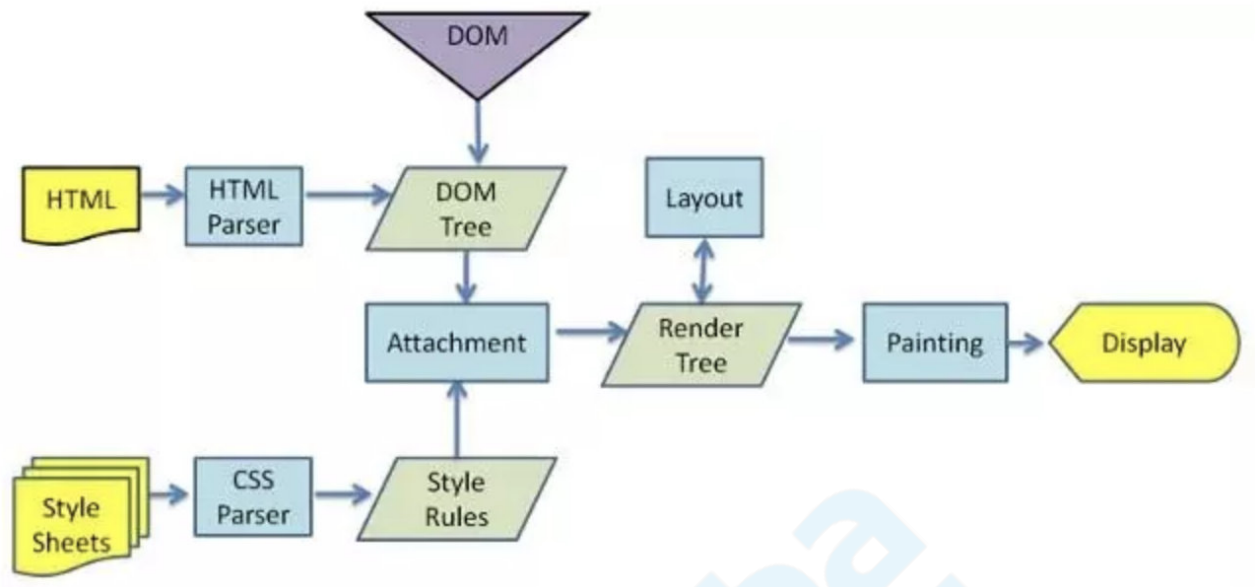
1. setState()执行时，updater会将partialState添加到它维护的pendingStates中，等到
2. updateComponent负责合并pendingStates中所有state变成一个state
3. forceUpdate执行新旧vdom比对-diff以及实际更新操作

虚拟dom

常见问题：react virtual dom是什么？说一下diff算法?

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 = str + key + " "
}
console.log(str)
```

```
align title lang translate dir dataset hidden tabIndex accessKey draggable spellcheck contentEditable isContentEditable offsetParent offsetTop offsetLeft  VM1696:
offsetWidth offsetHeight style innerText outerText webkitdropzone 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 onshow onstalled onsubmit onsuspend ontimeupdate ontoggle
onvolumechange onwaiting click focus blur onautocomplete onautocompleteerror namespaceURI prefix localName tagName id className classList attributes innerHTML outerHTML
shadowRoot scrollTop scrollLeft scrollWidth scrollHeight clientTop clientLeft clientWidth clientHeight onbeforecopy onbeforecut onbeforepaste oncopy oncut onpaste
onsearch onselectstart onwheel onwebkitfullscreenchange onwebkitfullscreenerror previousElementSibling nextElementSibling children firstElementChild lastElementChild
childElementCount hasAttributes getAttribute getAttributeNS setAttribute setAttributeNS removeAttribute removeAttributeNS hasAttribute hasAttributeNS getAttributeNode
getAttributeNodeNS setAttributeNode setAttributeNodeNS removeAttributeNode closest matches getElementsByTagName getElementsByTagNameNS getElementsByClassName
insertAdjacentHTML createShadowRoot getDestinationInsertionPoints requestPointerLock getClientRects getBoundingClientRect scrollIntoView insertAdjacentElement
insertAdjacentText scrollIntoViewIfNeeded webkitMatchesSelector animate remove webkitRequestFullScreen webkitRequestFullscreen querySelector querySelectorAll prepend
append before after replaceWith nodeType nodeName baseURI ownerDocument parentNode parentElement childNodes firstChild lastChild previousSibling nextSibling nodeValue
textContent hasChildNodes normalize cloneNode isEqualNode compareDocumentPosition contains lookupPrefix lookupNamespaceURI isDefaultNamespace insertBefore appendChild
replaceChild removeChild isSameNode 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 addEventListener removeEventListener dispatchEvent
```

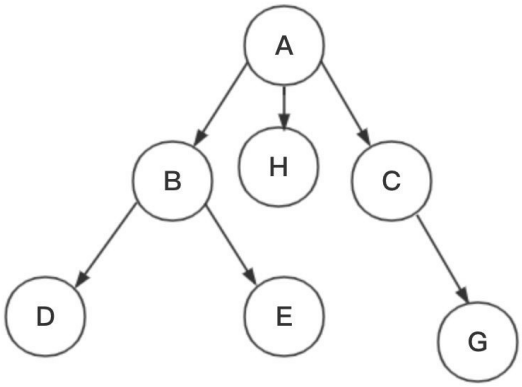
why? DOM操作很慢，轻微的操作都可能导致页面重新排版，非常耗性能。相对于DOM对象，js对象处理起来更快，而且更简单。通过diff算法对比新旧vdom之间的差异，可以批量的、最小化的执行dom操作，从而提高性能。

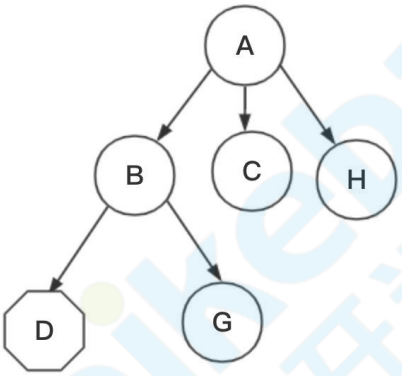
where? react中用JSX语法描述视图，通过babel-loader转译后它们变为React.createElement(...)形式，该函数将生成vdom来描述真实dom。将来如果状态变化，vdom将作出相应变化，再通过diff算法对比新老vdom区别从而做出最终dom操作。

how?

diff算法

算法复杂度O(n)

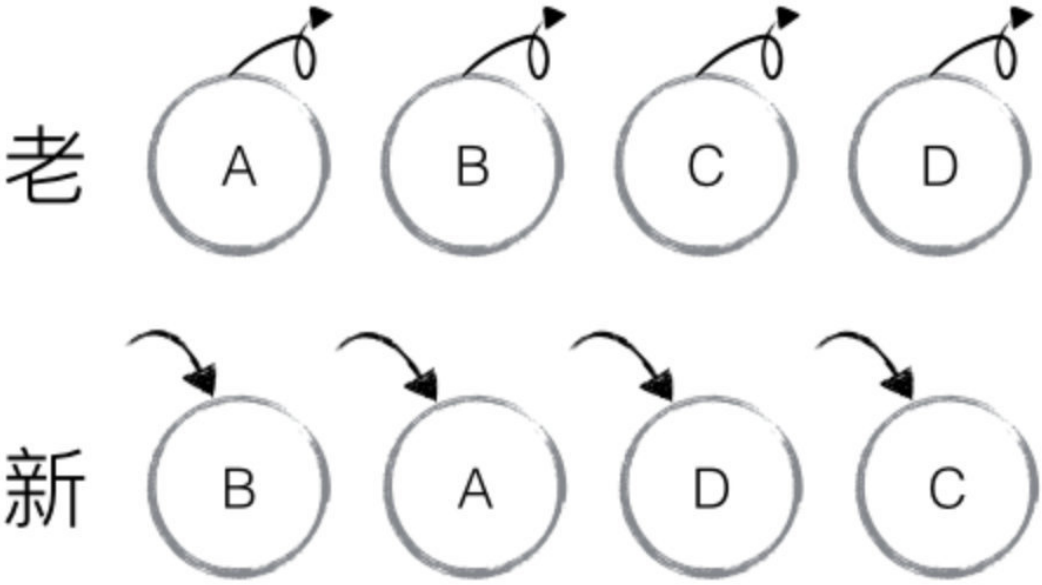


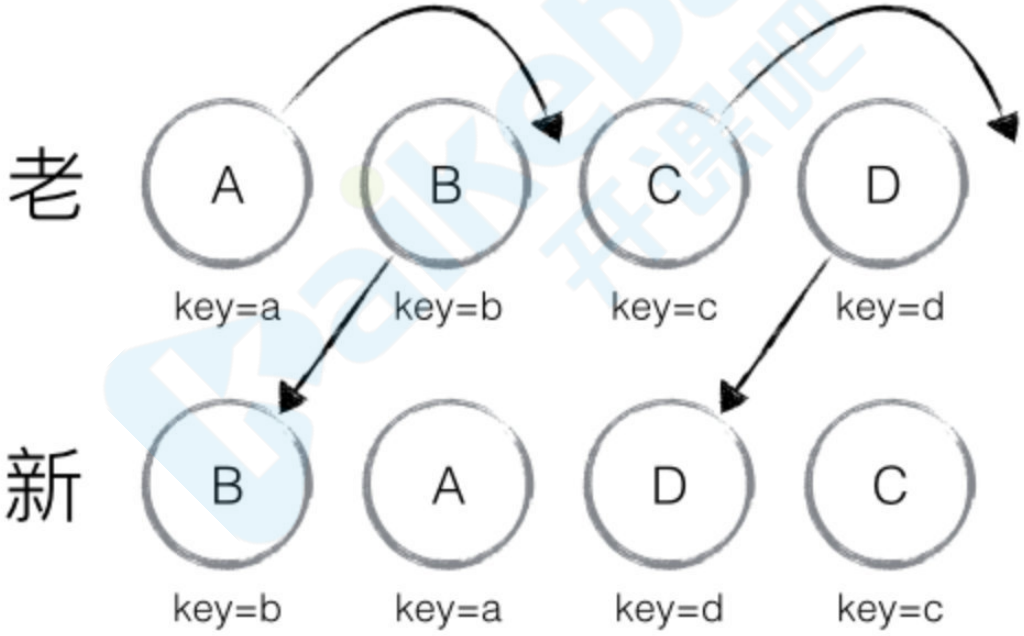


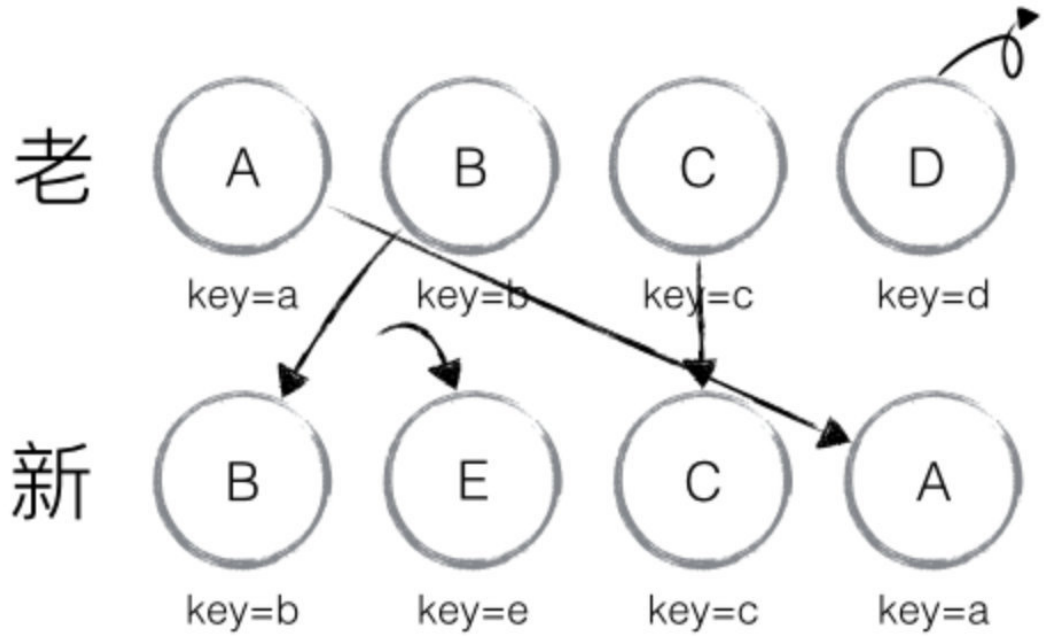
diff 策略

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；


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。

diff过程

比对两个虚拟dom时会有三种操作：删除、替换和更新

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

```
export function compareTwoVnodes(vnode, newVnode, node,
parentContext) {
    let newNode = node
    if (newVnode == null) {
        // remove
        destroyVnode(vnode, node)
        node.parentNode.removeChild(node)
    } else if (vnode.type !== newVnode.type || vnode.key
!== newVnode.key) {
```

```
        // replace
        destroyVnode(vnode, node)
        newNode = initVnode(newVnode, parentContext)
        node.parentNode.replaceChild(newNode, node)
    } else if (vnode !== newVnode || parentContext) {
        // same type and same key -> update
        newNode = updateVnode(vnode, newVnode, node,
parentContext)
    }
    return newNode
}
```

更新操作

根据组件类型执行不同更新操作

```
function updateVnode(vnode, newVnode, node, parentContext)
{
    let { vtype } = vnode
  //更新class类型组件
    if (vtype === VCOMPONENT) {
        return updateVcomponent(vnode, newVnode, node,
parentContext)
    }
  //更新函数类型组件
    if (vtype === VSTATELESS) {
        return updateVstateless(vnode, newVnode, node,
parentContext)
    }

    // ignore VCOMMENT and other vtypes
    if (vtype !== VELEMENT) {
        return node
    }

    // 更新元素
```

```
    let oldHtml = vnode.props[HTML_KEY] &&
vnode.props[HTML_KEY].__html
    if (oldHtml != null) {
        // 设置了innerHTML时先更新当前元素在初始化innerHTML
        updateVelem(vnode, newVnode, node, parentContext)
        initVchildren(newVnode, node, parentContext)
    } else {
        // 正常更新：先更新子元素，在更新当前元素
        updateVChildren(vnode, newVnode, node,
parentContext)
        updateVelem(vnode, newVnode, node, parentContext)
    }
    return node
}
```

patch过程

虚拟dom比对最终要转换为对应patch操作

属性更新

```
function updateVelem(velem, newVelem, node) {
    let isCustomComponent = velem.type.indexOf('-') >= 0 ||
velem.props.is != null
    _.patchProps(node, velem.props, newVelem.props,
isCustomComponent)
    return node
}
```

子元素更新

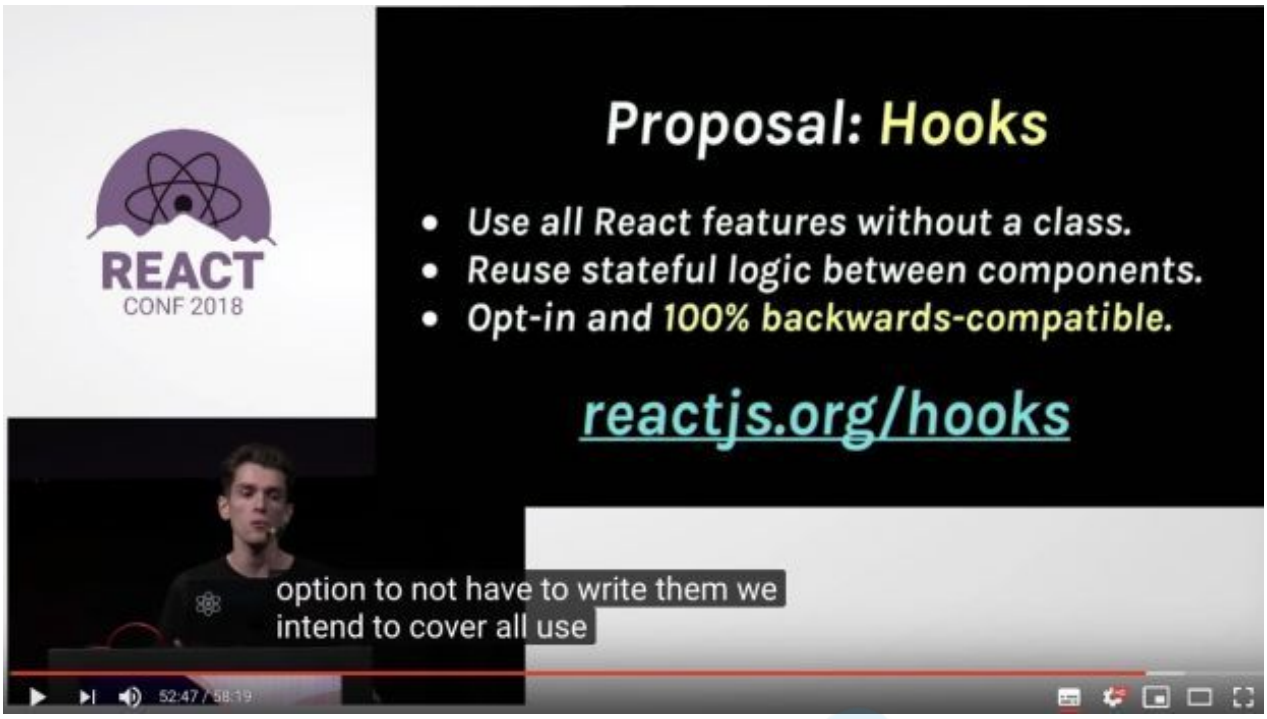
```
function updateVChildren(vnode, newVnode, node,
parentContext) {
    // 更新children，产出三个patch数组
    let patches = {
        removes: [],
        updates: [],
        creates: [],
    }
    diffVchildren(patches, vnode, newVnode, node,
parentContext)
    _.flatEach(patches.removes, applyDestroy)
    _.flatEach(patches.updates, applyUpdate)
    _.flatEach(patches.creates, applyCreate)
}
```

扩展点

Hooks

官网

hooks介绍视频：React Today and Tomorrow

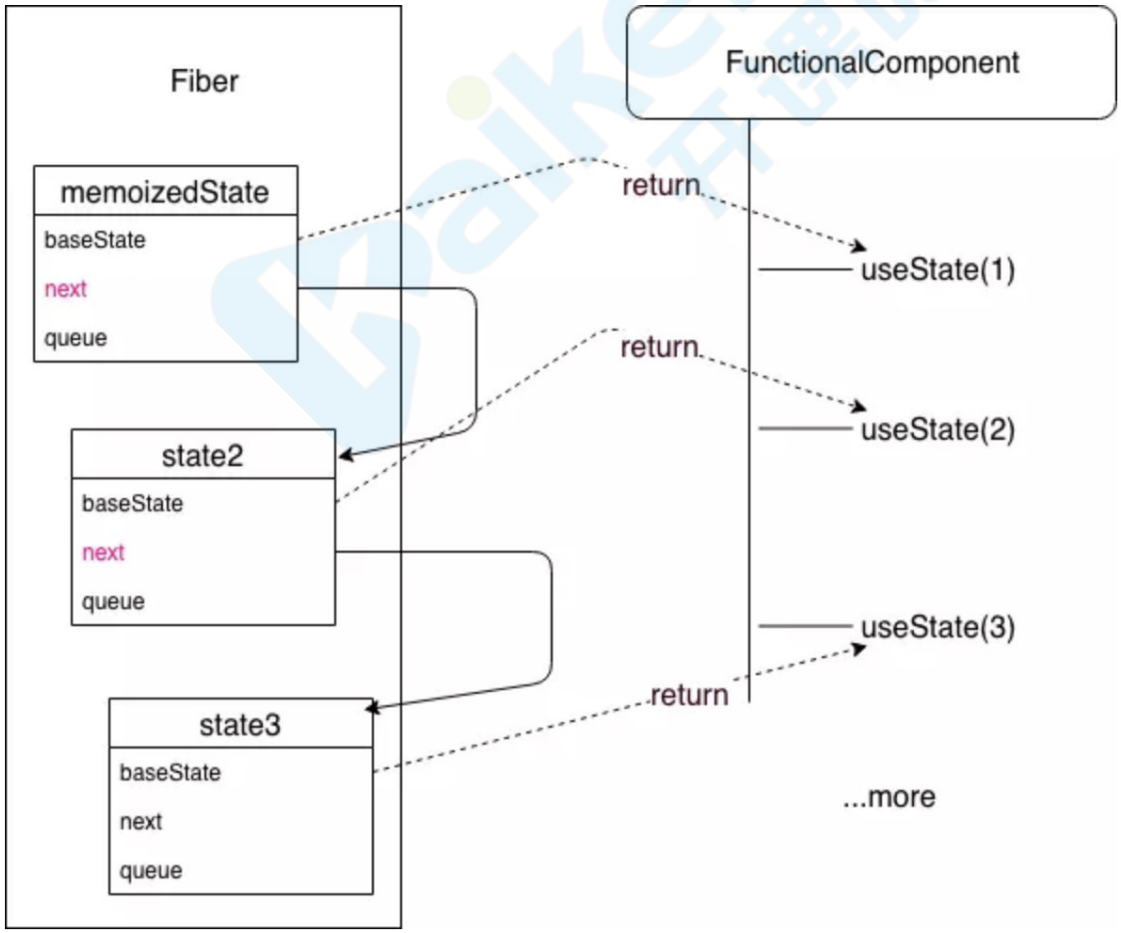


1. Hooks是什么？为了拥抱~~王能量~~函数式
2. Hooks带来的变革，让函数组件有了状态，可以替代class
3. 类似链表的实现原理

```
import React, { useState, useEffect } from 'react'

function FunComp(props) {
  const [data, setData] = useState('initialState')

  function handleChange(e) {
    setData(e.target.value)
  }

  useEffect(() => {
    subscribeToSomething()

    return () => {
      unSubscribeToSomething()
    }
  })

  return (
    <input value={data} onChange={handleChange} />
```

```
  )
}
```

```
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
```



Fibter

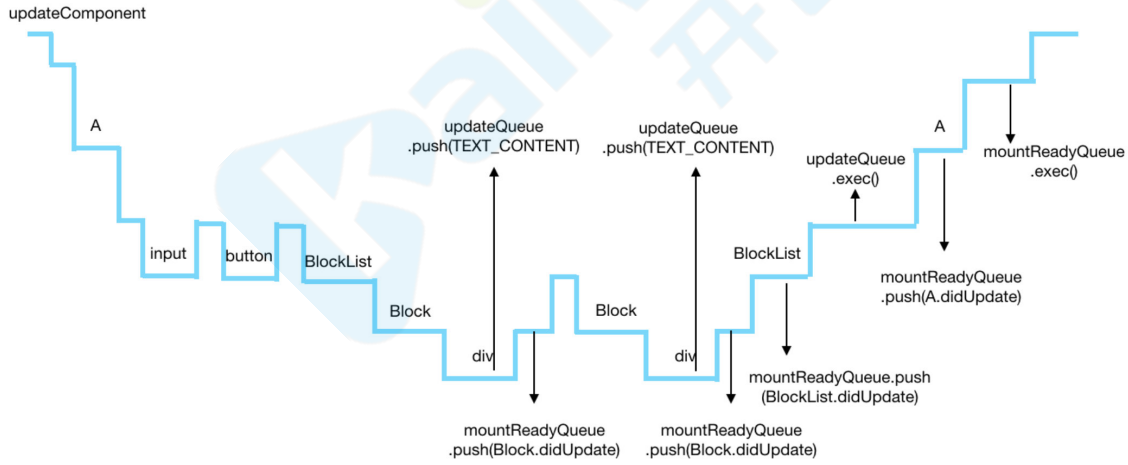
1. 为什么需要fiber

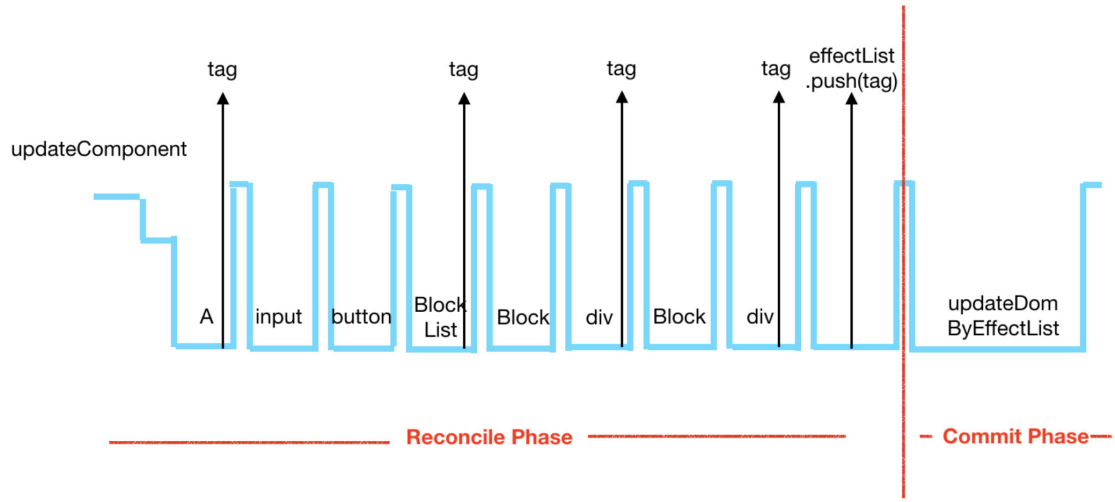
   对于大型项目，组件树会很大，这个时候递归遍历的成本就会很高，会造成主线程被持续占用，结果就是主线程上的布局、动画等周期性任务就无法立即得到处理，造成视觉上的卡顿，影响用户体验。

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**





why React

Mev-Rael Aug 22, 2016

Executive Product Leader & Mentor for High-End Influencers and Brands @ mevrael.com

Describe in 2 sentences what real world problem React is solving?

15 Reply

Ben Alpert Aug 31, 2016

React core team at Facebook

React tries to make it easy to build complicated apps by making it easy to split up your app into many simple, independent components. This means that when you change code in one place, other parts of your app don't break. It's hard to explain this benefit clearly when convincing people to use React but anecdotally, people find that their React apps are more stable and easier to debug compared to alternatives.

React is part of the "product infrastructure" group at Facebook, and our goal is to make it easier for product engineers to build great products. That could mean making it easier to get started, easier to build a high-quality app that feels good, or easier to debug an app once it's written. We also don't have top-down directives telling Facebook engineers what libraries or frameworks they need to use, so we focus on developer experience and developer happiness in order to get people at Facebook to use React. In the long run, our hope is that we can eliminate complexity and make it easier to get started with building UIs if you're new to programming.

总结

去实践中学习React