React原理解析02

课堂主题

深入理解React原理

资源

1. React源码
2. React源码概览
3. flow文档
4. debug react
5. kkreact文件指引
6. React源码文件指引

课堂目标

1. 深入理解setState原理

2. 深入理解虚拟dom、diff

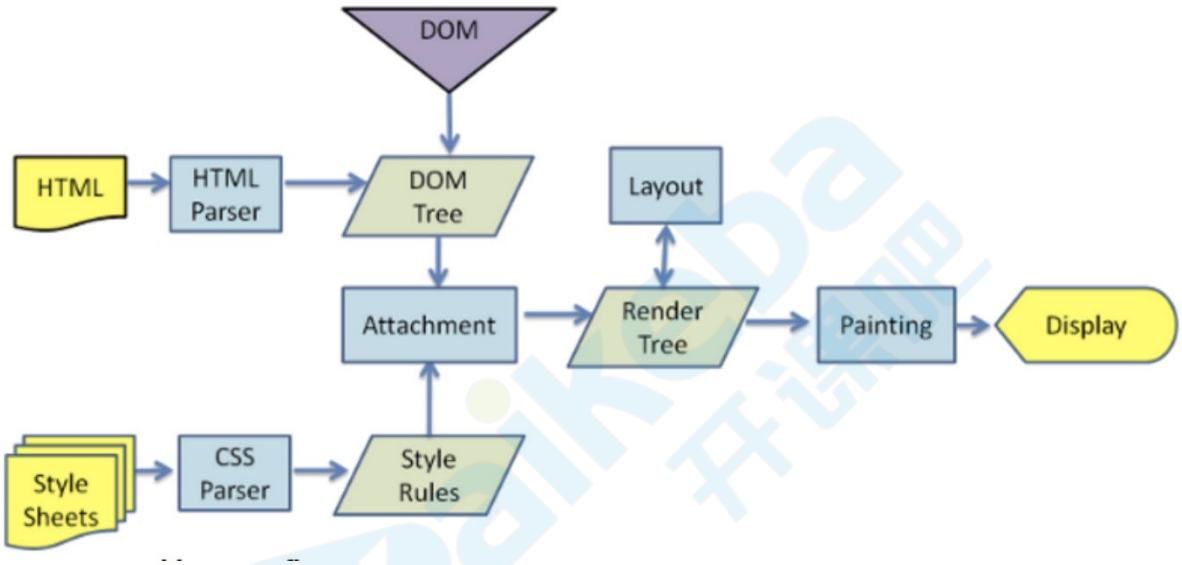
知识点

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 += ' ' + 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?

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。

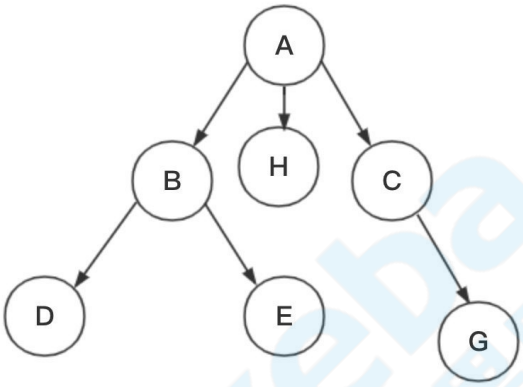
如果在 React 中使用了该算法，那么展示 1000 个元素所需要执行的计算量将在十亿的量级范围。这个开销实在是太过高昂。于是 React 在以下两个假设的基础之上提出了一套 O(n) 的启发式算法：

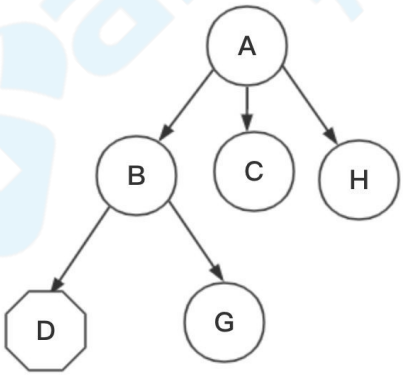
1. 两个不同类型的元素会产生出不同的树；
2. 开发者可以通过 `key` prop 来暗示哪些子元素在不同的渲染下能保持稳定；

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)

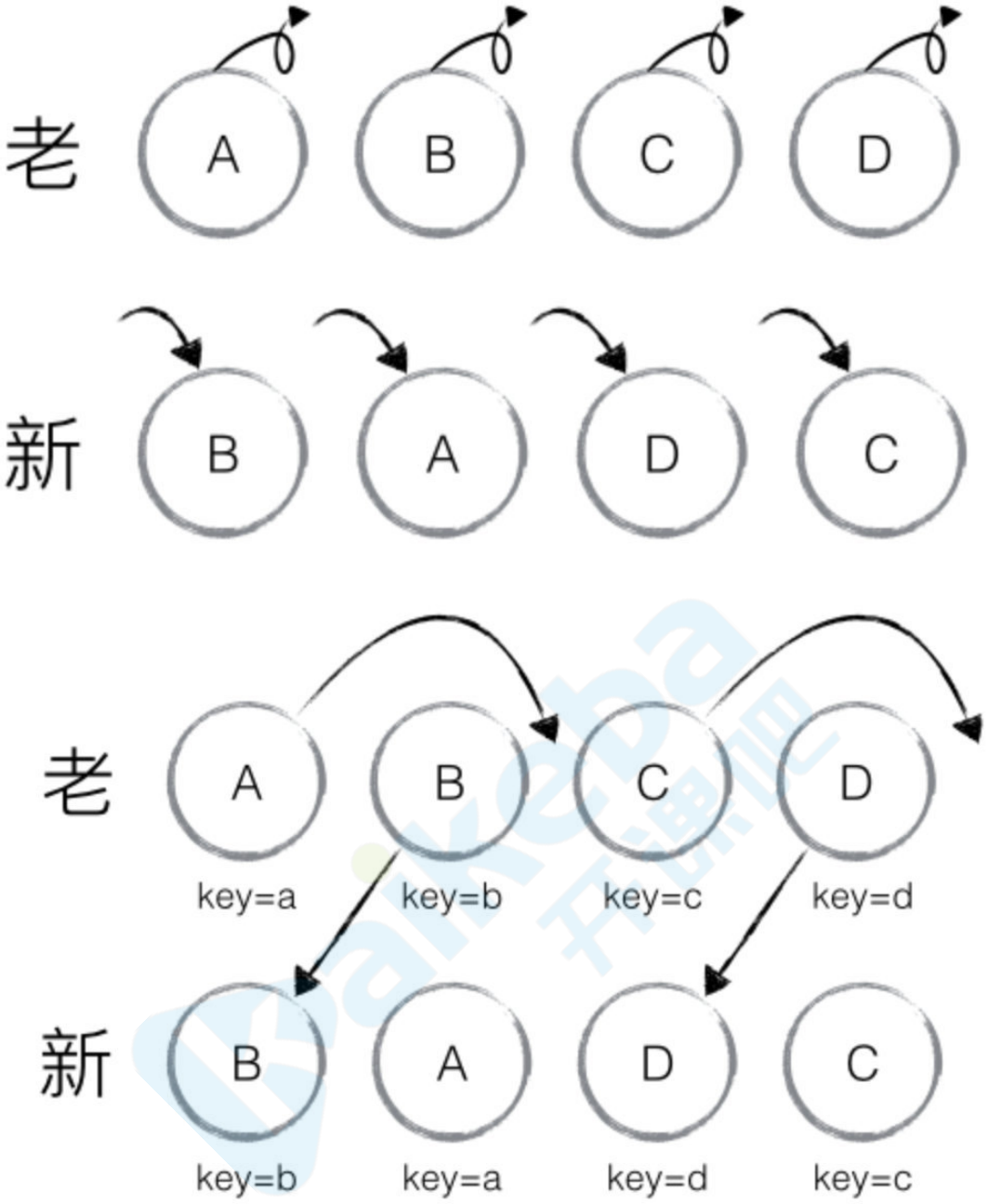


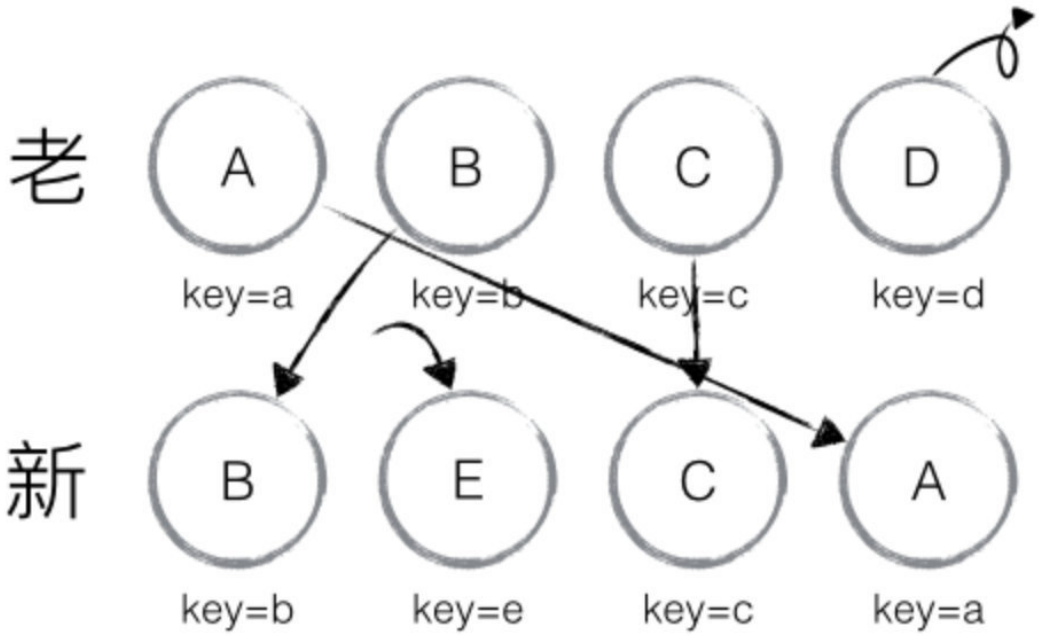


diff 策略

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);//在vdom中删除
        node.parentNode.removeChild(node)//dom中删除
    } else if (vnode.type !== newVnode.type || vnode.key !== newVnode.key) {
        // replace
        destroyVnode(vnode, node);//在vdom中删除
        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)
}
```

权衡

请谨记协调算法是一个实现细节。React 可以在每个 action 之后对整个应用进行重新渲染，得到的最终结果也会是一样的。在此情境下，重新渲染表示在所有组件内调用 `render` 方法，这不代表 React 会卸载或装载它们。React 只会基于以上提到的规则来决定如何进行差异的合并。

由于 React 依赖探索的算法，因此当以下假设没有得到满足，性能会有所损耗。

1. 该算法不会尝试匹配不同组件类型的子树。如果你发现你在两种不同类型的组件中切换，但输出非常相似的内容，建议把它们改成同一类型。在实践中，我们没有遇到这类问题。
2. Key 应该具有稳定，可预测，以及列表内唯一的特质。不稳定的 key（比如通过 `Math.random()` 生成的）会导致许多组件实例和 DOM 节点被不必要地重新创建，这可能导致性能下降和子组件中的状态丢失。

优化点：减少reflow

setState

`setState()` 会对一个组件的 `state` 对象安排一次更新。当 state 改变了，该组件就会重新渲染。

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异步

```
// 批量
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 2 });
console.log("counter", 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 React 不同步地更新 this.state?

   在开始重新渲染之前，React 会有意地进行“等待”，直到所有在组件的事件处理函数内调用的 `setState()` 完成之后。这样可以通过避免不必要的重新渲染来提升性能。

3. 为什么setState是异步的?

   这里的异步指的是多个state会合成到一起进行批量更新。

   Dan的回答

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

setState 什么时候是异步的?

目前，在事件处理函数内部的 `setState` 是异步的。

例如，如果 `Parent` 和 `Child` 在同一个 click 事件中都调用了 `setState` ，这样就可以确保 `Child` 不会被重新渲染两次。取而代之的是，React 会将该 state “冲洗” 到浏览器事件结束的时候，再统一地进行更新。这种机制可以在大型应用中得到很好的性能提升。

这只是一个实现的细节，所以请不要直接依赖于这种机制。在以后的版本当中，React 会在更多的情况下静默地使用 state 的批更新机制。

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。

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

flow

flow 是一个针对 JavaScript 代码的静态类型检测器。Flow 由 Facebook 开发，经常与 React 一起使用。flow 通过特殊的类型语法为变量、函数、以及 React 组件提供注解，帮助你尽早地发现错误。你可以阅读 introduction to Flow 来了解它的基础知识。

React将 flow引入源码，用于类型检查。在许可证头部的注释中，标记为 `@flow` 注释的文件是已经经过类型检查的。

Flow is a static type checker for your JavaScript code. It does a lot of work to make you more productive. Making you code faster, smarter, more confidently, and to a bigger scale.

Flow checks your code for errors through **static type annotations.** These *types* allow you to tell Flow how you want your code to work, and Flow will make sure it does work that way.

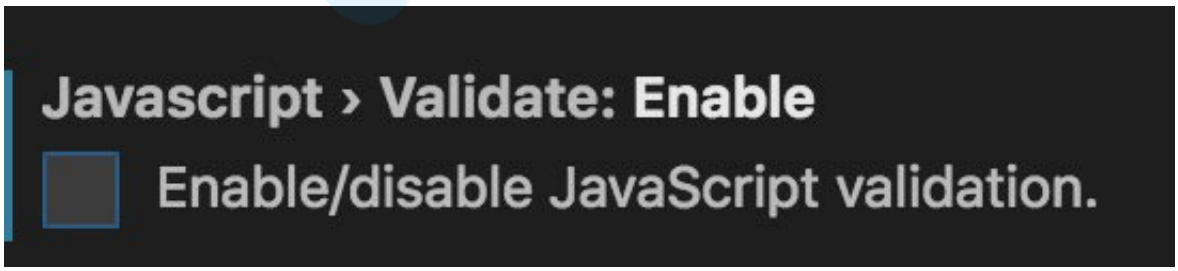
```
1  // @flow
2  function square(n: number): number {
3    return n * n;
4  }
5
6  square("2"); // Error!
```

Because Flow understands JavaScript so well, it doesn't need many of these types. You should only ever have to do a minimal amount of work to describe your code to Flow and it will *infer* the rest. A lot of the time, Flow can understand your code without any types at all.

```
1  // @flow
2  function square(n) {
3    return n * n; // Error!
4  }
5
6  square("2");
```

You can also adopt Flow incrementally and easily remove it at anytime, so you can try Flow out on any codebase and see how you like it.

阅读源码的时候，如果总有错误提示`'types' can only be used in a .ts file. 8010`，在扩展中找到TypeScript，配置"javascript.validate.enable": false。



回顾

- 虚拟dom
- 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
    - 更新操作
    - patch过程
  - 权衡
- setState
  - setState总结：
- flow

回顾