React技术框架探秘

用友网络科技股份有限公司

营销云-公共开发部

方奇功 | 2018年 10 月 18 日

Contents

四、同构与性能优化

引言

```
function tick() {
  document.getElementById('container').innerHTML =
    '<div><h1>Hello, world!</h1><h2>' + new Date().toLocaleTimeString() + '</h2></div>';
}
```

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>{new Date().toLocaleTimeString()}</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('container')
  );
}
```

首先需要开发思维的改变

上面两个函数实现有什么异同点？

关于JSX

```
const title = <h1 className="title">Hello, world!</h1>;
```

- jsx本质上是语法糖
- babel转换

```
const title = React.createElement(
    'h1',
    { className: 'title' },
    'Hello, world!'
);
```

React.createElement

定义

```
createElement( tag, attrs, child1, child2, child3 );
```

第一个参数：DOM节点的标签名，它的值可能是div，h1，span等
第二个参数：是一个对象，里面包含了所有的属性，可能包含了className，id等
从第三个参数开始，就是它的子节点

实现

```
function createElement( tag, attrs, ...children ) {
    return {
        tag,
        attrs,
        children
    }
}
```

虚拟DOM

```
// 将上文定义的createElement方法放到对象React中
const React = {
    createElement
}

const element = (
    <div>
        hello<span>world!</span>
    </div>
);
console.log( element );
```

React.createElement
何时被执行?

```
▼{tag: "div", attrs: null, children: Array(2)}
   attrs: null
 ▼children: Array(2)
    0: "hello"
  ▼1:
      attrs: null
    ▶children: ["world!"]
      tag: "span"
    ▶__proto__: Object
    length: 2
  ▶__proto__: Array(0)
   tag: "div"
 ▶__proto__: Object
```

createElement方法返回的对象记录了这个DOM节点所有的信息

这个记录信息的对象称之为**虚拟DOM**

详见test1示例

ReactDOM.render

```
ReactDOM.render(
    <h1>Hello, world!</h1>,
    document.getElementById('root')
);
```

```
ReactDOM.render(
    React.createElement( 'h1', null, 'Hello, world!' ),
    document.getElementById('root')
);
```

第一个参数：虚拟DOM

第二个参数：挂载的目标DOM

将虚拟DOM渲染成真实的DOM

详见test2示例

render实现

```
function render( vnode, container ) {

    // 当vnode为字符串时，渲染结果是一段文本
    if ( typeof vnode === 'string' ) {
        const textNode = document.createTextNode( vnode );
        return container.appendChild( textNode );
    }

    const dom = document.createElement( vnode.tag );

    if ( vnode.attrs ) {
        Object.keys( vnode.attrs ).forEach( key => {
            const value = vnode.attrs[ key ];
             setAttribute( dom, key, value );    // 设置属性
        } );
    }

    vnode.children.forEach( child => render( child, dom ) );    // 递归渲染子节点

    return container.appendChild( dom );    // 将渲染结果挂载到真正的DOM上
}
```

渲染和更新

详见test3示例

```
function tick() {
    const element = (
        <div>
            <h1>Hello, world!</h1>
            <h2>It is {new Date().toLocaleTimeString()}.</h2>
        </div>
      );
    ReactDOM.render(
        element,
        document.getElementById( 'root' )
    );
}

setInterval( tick, 1000 );
```

Hello, world!

It is 下午8:49:20.

Elements Console Sources Network

```
<!DOCTYPE html>
<html lang="en">
▶<head>…</head>
▼<body>
  ▶<div id="root">…</div> == $0
    <script src="/dist/771f8c8….js"></script>
    <div id="cli_dialog_div"></div>
  </body>
</html>
```

Question

JS模块化开发规范?

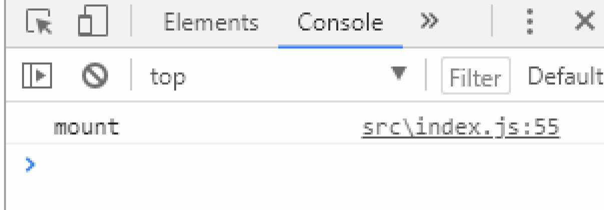
```
import React from 'react';    // 下面的代码没有用到React对象，为什么也要将其import进来
import ReactDOM from 'react-dom';

ReactDOM.render( <App />, document.getElementById( 'editor' ) );
```

四、同构与性能优化

组件和生命周期示例

```
class Counter extends React.Component {
    constructor( props ) {
        super( props );
        this.state = {
            num: 0
        }
    }

    componentWillUpdate() {
        console.log( 'update' );
    }

    componentWillMount() {
        console.log( 'mount' );
    }

    onClick() {
        this.setState( { num: this.state.num + 1 } );
    }

    render() {
        return (
            <div onClick={ () => this.onClick() }>
                <h1>number: {this.state.num}</h1>
                <button>add</button>
            </div>
        );
    }
}

ReactDOM.render(
    <Counter />,
    document.getElementById( 'root' )
);
```

详见test4示例



组件定义方式

- 函数

```
function Welcome( props ) {
    return <h1>Hello, {props.name}</h1>;
}
```

- 类

```
class Welcome extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}</h1>;
    }
}
```

函数定义可以看做是类定义的一种简单形式

组件基类React.Component

- state：私有状态
- props：传入数据

```
// React.Component
class Component {
    constructor( props = {} ) {
        this.state = {};
        this.props = props;
    }
}
```

- setState：修改数据，重新渲染

```
import { renderComponent } from '../react-dom/render'
class Component {
    constructor( props = {} ) {
        // ...
    }

    setState( stateChange ) {
        // 将修改合并到state
        Object.assign( this.state, stateChange );
        renderComponent( this );
    }
}
```

this.state.attr = ‘xxx’

vs

this.setState({attr: ‘xxx’})

render改进

```
function _render( vnode, container ) {

    // ...

    if ( typeof vnode.tag === 'function' ) {

        const component = createComponent( vnode.tag, vnode.attrs );

        setComponentProps( component, vnode.attrs );

        return component.base;
    }

    // ...
}
```

Children小结

Children现在看到有三种类型：

1、String

2、原生DOM节点的element

3、React components – 自定义组件的element

不单单是Object类型

4、false, null, undefined, number

5、数组 – 使用map方法的时候

createComponent – 创建组件

```
// 创建组件
function createComponent( component, props ) {

    let inst;
    // 如果是类定义组件，则直接返回实例
    if ( component.prototype && component.prototype.render ) {
        inst = new component( props );
    // 如果是函数定义组件，则将其扩展为类定义组件
    } else {
        inst = new Component( props );
        inst.constructor = component;
        inst.render = function() {
            return this.constructor( props );
        }
    }

    return inst;
}
```

setComponentProps – 更新props

```
// set props
function setComponentProps( component, props ) {

    if ( !component.base ) {
        if ( component.componentWillMount ) component.componentWillMount();
    } else if ( component.componentWillReceiveProps ) {
        component.componentWillReceiveProps( props );
    }

    component.props = props;

    renderComponent( component );

}
```

renderComponent – 渲染组件

- setState
- setComponentProps

```
export function renderComponent( component ) {

    let base;

    const renderer = component.render();

    if ( component.base && component.componentWillUpdate ) {
        component.componentWillUpdate();
    }

    base = _render( renderer );

    if ( component.base ) {
        if ( component.componentDidUpdate ) component.componentDidUpdate();
    } else if ( component.componentDidMount ) {
        component.componentDidMount();
    }

    if ( component.base && component.base.parentNode ) {
        component.base.parentNode.replaceChild( base, component.base );
    }

    component.base = base;
    base._component = component;

}
```

React组件生命周期总结

组件的生命周期分成三个状态：

- Mounting：已插入真实 DOM
- Updating：正在被重新渲染
- Unmounting：已移出真实 DOM

React为每个状态都提供了两种处理函数：will和did

- componentWillMount()
- componentDidMount()
- componentWillUpdate(object nextProps, object nextState)
- componentDidUpdate(object prevProps, object prevState)
- componentWillUnmount()

React还提供两种特殊状态的处理函数

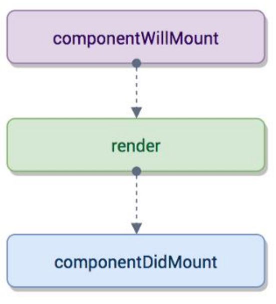
- componentWillReceiveProps(object nextProps)：已加载组件收到新的参数时调用
- shouldComponentUpdate(object nextProps, object nextState)：组件判断是否重新渲染时调用

React组件生命周期总结

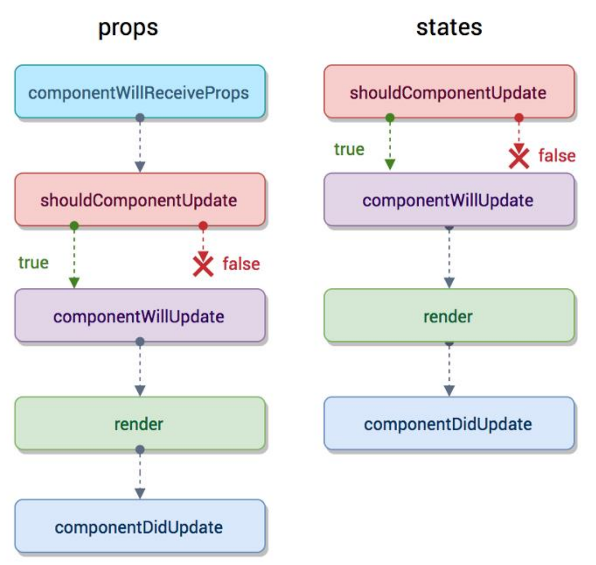
Initialization

setup props and state

Mounting



Updation



Unmounting

componentWillUnmount

React组件运行机制模拟

Example → <div>Hello World!</div>

- element是否为Object
  - 是
    - 原生DOM — ReactDOMComponent
    - 自定义类 — ReactCompositeComponentWrapper
  - 否
    - string, number — ReactDOMTextComponent
    - null, false — ReactDOMEmptyComponent

首次渲染

ReactCompositeComponentWrapper

React.render
（<Example />，
container）

mountComponent:

1、实例化Example，得到instance对象
2、renderedElement = Instance.render();
3、初始化renderedElement，得到child
4、child.mountComponent(container)

生命周期函数在哪被调用？

componentDidMount

```
{
  type: function Example,
  props: {
    children: null
  }
}
```

<div>Hello World!</div>

```
{
  type: 'div',
  props: {
    children: 'Hello World'
  }
}
```

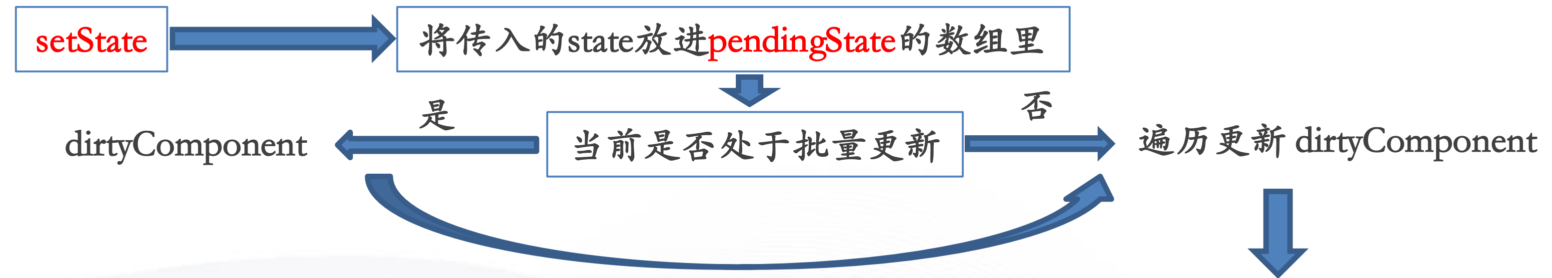
ReactDOMComponent

mountComponent:

根据element来生成对应的真实DOM节点

递归调用　生命周期

渲染更新



ReactCompositeComponentWrapper

shou

生命周期函数在哪被调用?

componentWillUpdate

componentDidUpdate

updateComponent

1、计算出nextState
2、render()得到nextRenderElement
3、与prevElement 进行Diff 比较,更新节点

ReactDOMComponent

updateComponent

用于直接操作更新浏览器DOM元素

生命周期

SCU – 运行效率优化点

- shouldComponentUpdate(nextProps, nextState)
  - 通过判断下一个状态与当前状态的差异，决定是否渲染当前组件
  - 能够很大程度上减少不必要的组件渲染，**提高运行效率**
  - 详见test7示例

四、同构与性能优化

DOM渲染过程

HMTL

HTML 解析器
分词 + 解析

DOM
树

DOM
+CSS

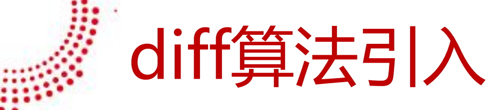
渲染树

渲染

CSS
样式

CSS
解析器

CSS
规则



diff算法引入

我们目前的做法是每次更新都重新渲染整个组件甚至是整个应用，这样的做法在页面复杂时将会暴露出性能上的问题，DOM操作非常昂贵，而为了减少DOM操作，React又做了哪些事？

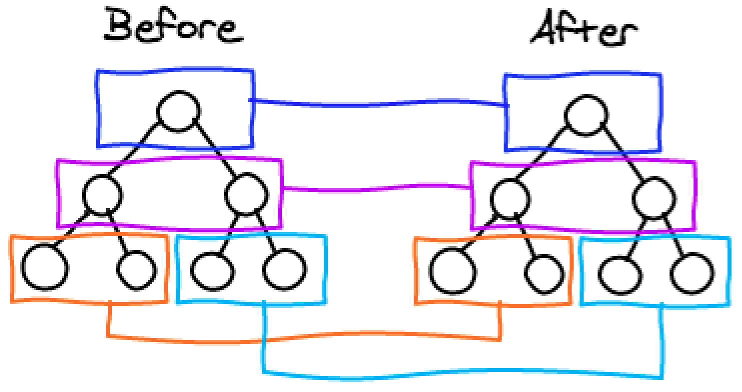
为了减少DOM更新，我们需要找渲染前后真正变化的部分，只更新这一部分DOM。而对比变化，找出需要更新部分的算法我们称之为**diff算法**。

diff算法原则

- 对比真实DOM和虚拟DOM，直接更新真实DOM

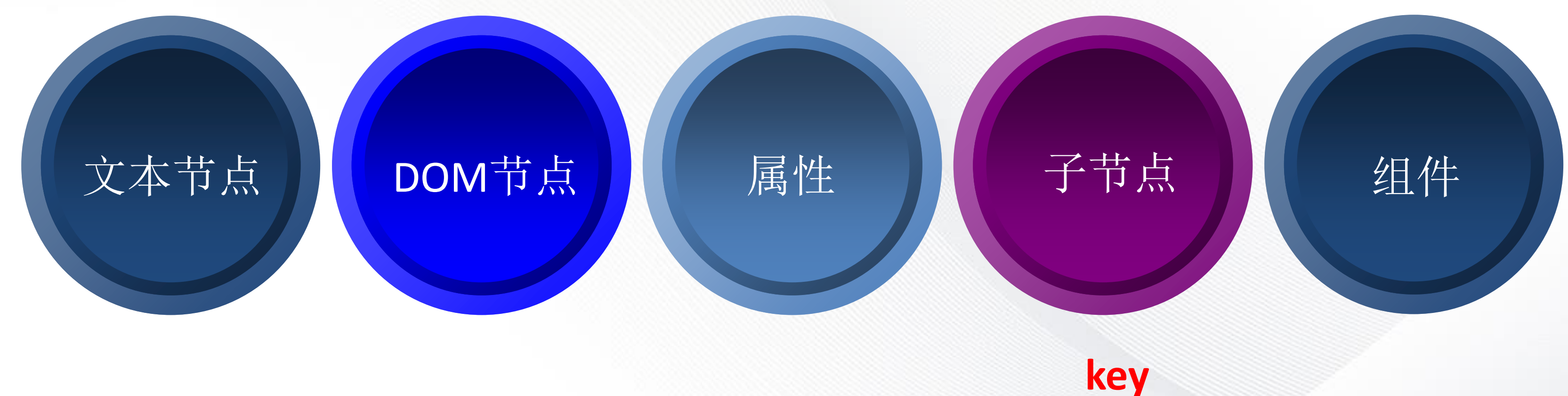
```
/**
 * @param {HTMLElement} dom 真实DOM
 * @param {vnode} vnode 虚拟DOM
 * @returns {HTMLElement} 更新后的DOM
 */
function diff( dom, vnode ) {
    // ...
}
```

- 只对比同一层级的变化



diff算法实现

diff.js



有无diff运行结果对比

localhost:1234

count: 1

add

Elements Console Sources Network

```
<!DOCTYPE html>
<html lang="en">
▶<head>…</head>
▼<body>
  ▼<div id="root">
    ▼<div>
      ▼<h1>
          "count: "
          "1"
        </h1>
        <button>add</button> == $0
      </div>
    </div>
    <script src="/dist/771f8c8….js"></script>
    <div id="cli_dialog_div"></div>
  </body>
</html>
```

localhost:1234

count: 1

add

Elements Console Sources Network

```
<!DOCTYPE html>
<html lang="en">
▶<head>…</head>
▼<body> == $0
  ▼<div id="root">
    ▼<div>
      ▼<h1>
          "count: "
          "1"
        </h1>
        <button>add</button>
      </div>
    </div>
    <script src="/dist/771f8c8….js"></script>
    <div id="cli_dialog_div"></div>
  </body>
</html>
```

详见test5示例

Fiber架构思想简介 – 两个阶段

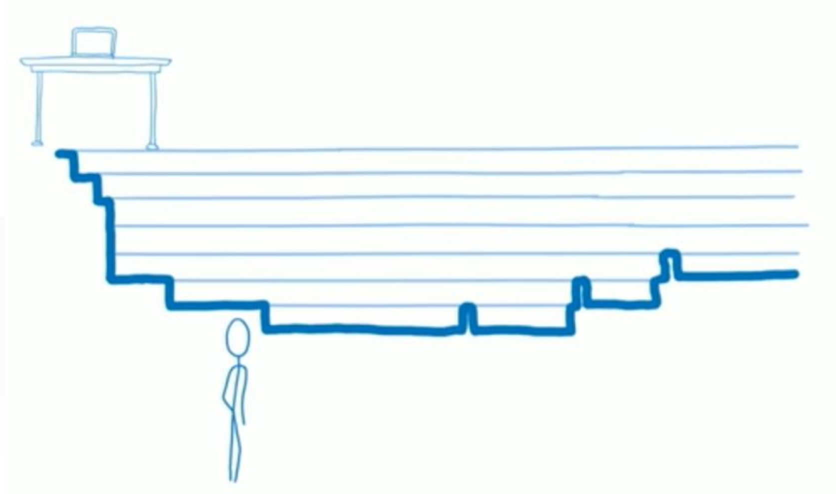
- render/reconciliation 通过构造workInProgress tree得出change（可中断）
- commit 应用这些DOM change（不可中断）

```
// 第1阶段 render/reconciliation
componentWillMount
componentWillReceiveProps
shouldComponentUpdate
componentWillUpdate

// 第2阶段 commit
componentDidMount
componentDidUpdate
componentWillUnmount
```

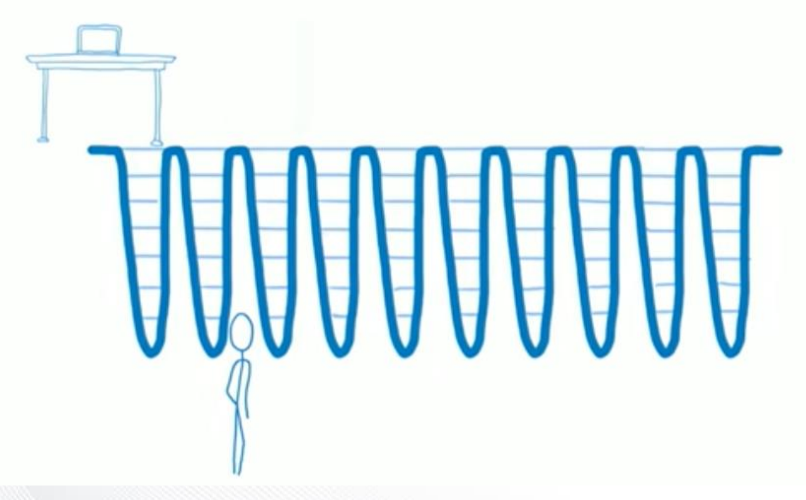
引入目的：可以使得一些更高优先级的任务，如用户的操作能够优先执行，提高用户的体验，至少用户不会感觉到卡顿~

温馨提示：第二阶段一口气做完（同步执行，不能喊停）的，这个阶段的实际工作量是比较大的，所以尽量不要在后3个生命周期函数里干重活儿

reconciler对比



Stack reconciler



Fiber reconciler

Fiber架构图

Mounting | **Updating** | **Unmounting**

New props | *setState()* | *forceUpdate()*

“Render Phase”

Pure and has no side effects. May be paused, aborted or restarted by React.

constructor

getDerivedStateFromProps

shouldComponentUpdate

render

“Pre-Commit Phase”

Can read the DOM.

getSnapshotBeforeUpdate

“Commit Phase”

Can work with DOM, run side effects, schedule updates.

React updates DOM and refs

componentDidMount

componentDidUpdate

componentWillUnmount

新的问题引入

每次调用setState后会立即调用renderComponent重新渲染组件
现实中我们可能会在极短的时间内多次调用setState

```
onClick() {
    for ( let i = 0; i < 100; i++ ) {
        this.setState( { num: this.state.num + 1 } );
    }
}
```

异步的setState

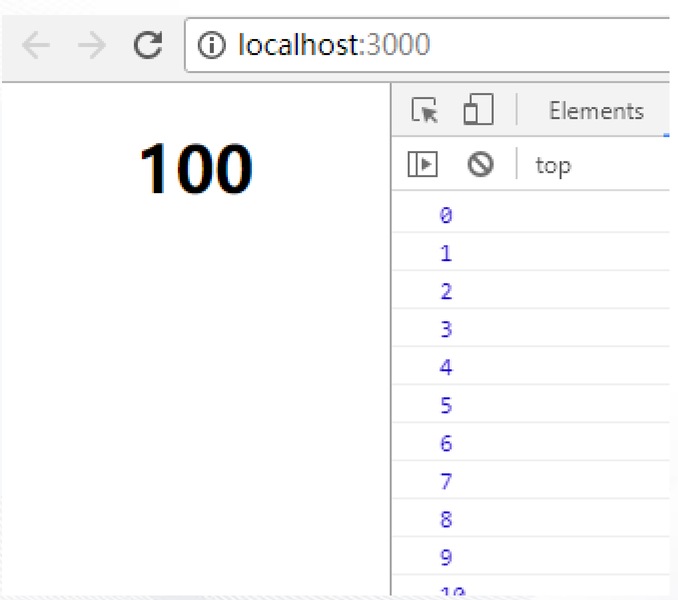
```
class App extends Component {
    constructor() {
        super();
        this.state = {
            num: 0
        }
    }
    componentDidMount() {
        for ( let i = 0; i < 100; i++ ) {
            this.setState( { num: this.state.num + 1 } );
            console.log( this.state.num );    // 会输出什么?
        }
    }
    render() {
        return (
            <div className="App">
                <h1>{ this.state.num }</h1>
            </div>
        );
    }
}
```


这是React的优化手段，但是显然它也会在导致一些不符合直觉的问题。

异步的setState

针对上面这种情况，React给出了一种解决方案：setState接收的参数还可以是一个函数，在这个函数中可以拿先前的状态，并通过这个函数的返回值得到下一个状态。

```
componentDidMount() {
    for ( let i = 0; i < 100; i++ ) {
        this.setState( prevState => {
            console.log( prevState.num );
            return {
                num: prevState.num + 1
            }
        } );
    }
}
```



详见test6示例

setState队列

- 入队：将更新添加进队列

```
const queue = [];
function enqueueSetState( stateChange, component ) {
    queue.push( {
        stateChange,
        component
    } );
}
```

- setState方法改进

```
setState( stateChange ) {
    enqueueSetState( stateChange, this );
}
```

清空队列

```
function flush() {
    let item;
    // 遍历
    while( item = setStateQueue.shift() ) {

        const { stateChange, component } = item;

        // 如果没有prevState，则将当前的state作为初始的prevState
        if ( !component.prevState ) {
            component.prevState = Object.assign( {}, component.state );
        }

        // 如果stateChange是一个方法，也就是setState的第二种形式
        if ( typeof stateChange === 'function' ) {
            Object.assign( component.state, stateChange( component.prevState, component.props ) );
        } else {
            // 如果stateChange是一个对象，则直接合并到setState中
            Object.assign( component.state, stateChange );
        }

        component.prevState = component.state;

    }
}
```

setState队列改进 - 渲染组件

渲染组件能不能在遍历队列时进行？

```
const queue = [];
const renderQueue = [];
function enqueueSetState( stateChange, component ) {
    queue.push( {
        stateChange,
        component
    } );
    // 如果renderQueue里没有当前组件，则添加到队列中
    if ( !renderQueue.some( item => item === component ) ) {
        renderQueue.push( component );
    }
}
```

```
function flush() {
    let item, component;
    while( item = queue.shift() ) {
        // ...
    }
    // 渲染每一个组件
    while( component = renderQueue.shift() ) {
        renderComponent( component );
    }

}
```

延迟执行

```
setTimeout( () => {
    console.log( 2 );
}, 0 );
Promise.resolve().then( () => console.log( 1 ) );
console.log( 3 );
```

原理详见阮一峰的文章：《JavaScript 运行机制详解：再谈Event Loop》

```
function defer( fn ) {
    return Promise.resolve().then( fn );
}
```

```
function enqueueSetState( stateChange, component ) {
    // 如果queue的长度是0，也就是在上次flush执行之后第一次往队列里添加
    if ( queue.length === 0 ) {
        defer( flush );
    }
    queue.push( {
        stateChange,
        component
    } );
    if ( !renderQueue.some( item => item === component ) ) {
        renderQueue.push( component );
    }
}
```

用友云
yonyou cloud

目录
Contents

一、设计原理简介

二、组件和生命周期

三、渲染过程推究

四、同构与性能优化

同构实现首屏直出

- 同构：同一套组件，浏览器端和服务器端都可以渲染
- 服务器端：react-dom/server
  - renderToString
  - renderToStaticMarkup
- 浏览器端：react-dom
  - render

详见test9示例

表格组件大数据下性能对比

- FixedDataTable
  - 扩展性强
  - 大数据下运行效率高
  - 对使用者要求比较高
- AntdTable
  - 使用方便
  - 大数据下运行效率低
  - 扩展性差

大数据下运行效率有很明显的差异，**源于dom操作管理不同**
详见test8示例

性能监测工具

- Performance Tools: react-addons-perf
  - `start()`
  - `stop()`
  - `getLastMeasurements()`
  - `printInclusive()`
  - `printExclusive()`
  - `printWasted()`
  - `printOperations()`
  - `printDOM()`

(index)	Owner > Component	Inclusive render time (ms)	Instance count
0	"Route > Home"	82.4	1
1	"Home > CourseItem"	37.3	770
2	"Home > RecommendCourse"	5.3	1

▸ Array(3)

(index)	Owner > Component	Inclusive wasted time (ms)	Instance count
0	"Home > CourseItem"	37.3	770
1	"Home > RecommendCourse"	5.3	1

http://git.yonyou.com/fangqg/testweb