# React

## React简介

起源：Facebook

由于react的设计思想极其独特，属于革命性创新，代码逻辑简单。

清楚两个概念：

Library（库）：小而巧；只是提供了一些api优点很方便的从一个库切换到另外一个库；但是代码机会不会改变

Framework（框架）：大而全；框架是提供了一整套的解决方案，切换框架比较困难

### 前端三大主流框架

三大框架一大抄

#### Angular.js

出来较早的前端框架，学习曲线比较陡

#### Vue.js

最火的（关注的人比较多）一门前端框架，中国人开发，对我们来说，文档友好

#### React.js

最流行（用的人比较多）的一门框架，因为他设计的很优秀

### 前端主流工具

#### NPM

NPM（Node Package Manager）是 Node.js 下的主流套件管理工具

NPM 主要是基于 [CommonJS](https://en.wikipedia.org/wiki/CommonJS) 的规范，通常必须搭配 Browserify 这样的工具才能在前端使用 NPM 的模组。

然而因 NPM 是基于 Nested Dependency Tree，不同的套件有可能会在引入依赖时会引入相同但不同版本的套件，造成档案大小过大的情形。这和另一个套件管理工具 [Bower](https://bower.io/) 专注在前端套件且使用 Flat Dependency Tree（让使用者决定相依的套件版本）是比较不同的地方。

#### ES6+

[ES6+](https://babeljs.io/blog/2015/06/07/react-on-es6-plus) 系指 ES6（ES2015）和 ES7 的联集，在 ES6+ 新的标准当中引入许多新的特性和功能，弥补了过去 JavaScript 被诟病的一些特性。由于未来 React 将以支援 ES6+ 为主，因此直接学习 ES6+ 用法是相对好的选择，本书的所有范例也将会以 ES6+ 撰写。

#### Babel

由于并非所有浏览器都支援 ES6+ 语法，所以透过 [Babel](https://babeljs.io/) 这个 JavaScript 编译器（可以想成是翻译机或是翻译蒟篛）可以让你的 ES6+ 、JSX 等程式码转换成浏览器可以看得懂的语法。通常会在资料夹的 root 位置加入 .babelrc 进行转译规则 preset 和引用外挂（plugin）的设定。

### Vue.js 和React.js对比

#### 组件化方面

1. 什么是模块化：从代码方面进行分析；把一些可以复用的代码，抽离为单个的模块；便于项目维护和开发
2. 什么是组件化：是从UI界面的角度进行分析的；把一些可以复用的UI元素，抽离为单个的组件；便于项目的维护和开发
3. 组件化的好处：随着项目的增大，手里的组件越来越多；很方便就能把现有的组件，拼接为一个完整的页面

创建组建的方式：Vue.component() /Vue.extends()/.vue.模板文件

1. Vue是如何实现组件化的：通过.vue文件，创建相应的组件

* Template 结构
* Script 行为
* Style 样式

1. react如何实现组件化：react中，一切都是以js来表现

#### 开发团队

React 由Facebook维护

Vue 由团队较小由作者团体维护

#### 社区方面

React诞生较早，社区强大；vue近两年才开始火，社区相对较小

#### 移动APP开发体验方面

Vue，结合weex这门技术，提供了迁移到移动端app开发的体验（没有重大成功案例）

React，结合reactNative，提供了无缝迁移到移动端APP的开发经验

### 为什么学习react

1. 基于js开发实现了组件化开发的思想
2. 团队实力强大
3. 社区强大
4. 提供了无缝切换ReactNative开发体验
5. 很多企业使用reactjs

### React核心概念

#### 虚拟dom

##### Dom本质：

浏览器中的概念，用js对象来表示页面上的元素，并提供dom对象的api

##### React中的虚拟dom

##### 虚拟dom目的

为了实现页面的高效更新

##### Dom和虚拟dom的区别

#### Diff算法

[..\面试题（vuejs+js+react+小程序+uni-app).doc](../面试题（vuejs+js+react+小程序+uni-app).doc)

##### Tree diff

新旧两颗dom树，逐层对比的过程，就是tree diff ； 当整个dom树逐层对比完毕，则所有需要被按需更新的元素，必然能够找到；

##### Component diff

在进行tree diff 的时候，每一层中，组件级对比叫做component diff；

如果对比前后，组件类型相同，则暂时认为此组件不需要被更新

如果对比前后，组件类型不相同，则需要移除久组件，创建新的组件，并追加到页面上

##### Element diff

组件类型相同时，进行元素级别对比，元素级别的对比叫做 element diff

### 生命周期钩子函数

#### 阶段划分钩子函数

##### 挂载阶段

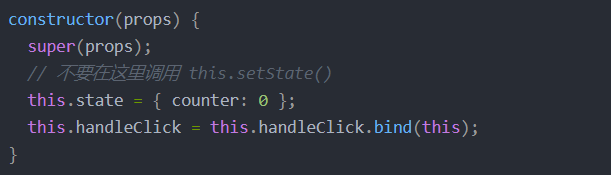
当组件实例被创建并插入 DOM 中时，其生命周期调用顺序如下

1. constructor() 只在实例化时执行一次

此函数有两个作用 ：1.管理应用程序业务逻辑所用到的 数据 2.改变自定义函数的this指向

只能在构造函数中直接为 this.state 赋值。如需在其他方法中赋值，你应使用 this.setState() 替代。

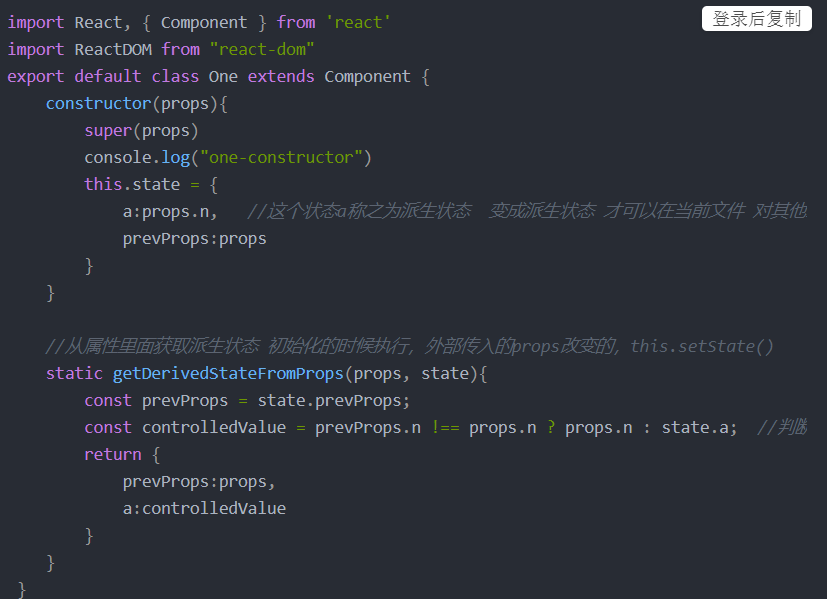
注意: React 组件在触发了 this.setState({}) 方法之后，会依次调用 shouldComponentUpdate, componentWillUpdate, render 和 componentDidUpdate 方法。



1. static getDerivedStateFromProps() 初始化 时执行 数据改变也执行

static getDerivedStateFromProps() 会在调用 render 方法之前调用，并且在初始挂载及后续更新时都会被调用。它应返回一个对象来更新 state，如果返回 null 则不更新任何内容。  
一般当我们需要对其它组件传递过来的数据 进行操作或改变时 使用 static

getDerivedStateFromProps()



1. render() 初始化 时执行 数据改变也执行
2. componentDidMount() 挂载完毕执行 一般在这里 进行ajax请求 只执行一次

注意:

下述生命周期方法即将过时，在新代码中应该避免使用它们：

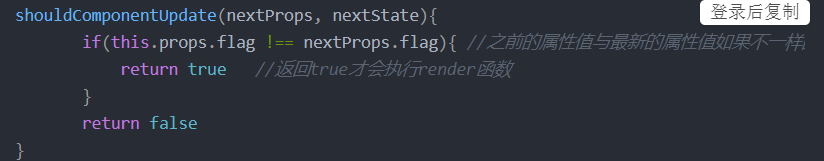
UNSAFE\_componentWillMount()

##### 更新阶段

当组件的 props 或 state 发生变化时会触发更新。组件更新的生命周期调用顺序如下：

1. static getDerivedStateFromProps()
2. shouldComponentUpdate() //此函数返回true时才会执行reder函数

shouldComponentUpdate() 发生在render函数之前 可以在挂载之前 做一些判断  
来决定是否 执行render函数 此函数返回true得时候才会执行render函数



PureComponent代表纯组件，纯组件内部是不能再去复写shouldComponentUpdate这个钩子函数了  
内部进行浅比较(进行值比较或者内部地址的比较)，进行性能的优化  
注意：如果是值比较的话，外部传入的属性与内部的属性不一样的话，才会进行render更新操作  
如果是引用地址比较的话，例如数组，之前的属性数组的地址与更改后的新的数组的地址一致，那么就不会进行render的更新操作

###### 性能提升

所谓的性能提升就是说白了，就是在某些场景下可以优化react的render的渲染次数。  
1.在普通组件里面可以使用shouldComponentUpdate钩子函数提升react性能。 （nextProps,nextState）  
在内部可以判断组件外部接受的最新属性与之前的属性是否一致，从而约束render刷新的时机。  
只要结果返回true，render就会立马执行渲染更新，返回false就代表render不会执行。  
2.可以使用PureComponent来优化性能。内部机制是通过浅比较去实现的。  
3.对于无状态组件的话，使用 React.memo(组件) 来进行性能提升

1. render()
2. getSnapshotBeforeUpdate() //在更新成真实dom之前执行
3. componentDidUpdate() //在更新成真实dom之后执行

注意:

下述方法即将过时，在新代码中应该避免使用它们：

UNSAFE\_componentWillUpdate()

UNSAFE\_componentWillReceiveProps()

##### 卸载阶段

当组件从 DOM 中移除时会调用如下方法：

​ componentWillUnmount()

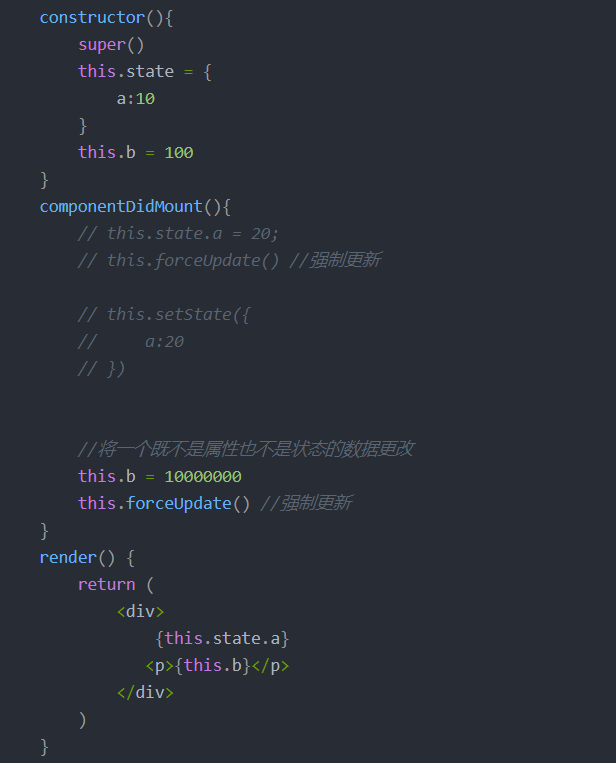
##### 错误处理

当渲染过程，生命周期，或子组件的构造函数中抛出错误时，会调用如下方法：

1. static getDerivedStateFromError()
2. componentDidCatch()

#### forceUpdate() 强制更新数据

vue中也有强制更新的方法 this.f o r c e U p d a t e ( ) 和 t h i s . forceUpdate() 和 this.*forceUpdate*()和*this*.set()  
当数据改变视图没有更新的时候 可以使用强制更新



#### 父组件

**import React , {Component} from 'react'**

**import Son from './Son'**

**class Father extends Component {**

**constructor(props){**

**// 可以绑定this 设置props 根据属性设置状态 给方法绑定this**

**super(props)**

**console.log('Father : constructor')**

**this.state = {**

**color : 'blue'**

**}**

**}**

**componentWillMount(){**

**// 修改state 不会触发update 可以做初始数据的获取**

**console.log('Father : componentWillMount')**

**// this.state = {**

**// color : 'blue'**

**// } // color值被改变 但是会报错，建议用setState**

**}**

**render(){**

**// 父组件在render过程中会触发子组件的初始化阶段，当所有的组件都转载完成后，父组件才会执行didmount**

**console.log('Father : render')**

**return (**

**<div>**

**<button className='Btn' style = {{color:this.state.color}}**

**onClick = {()=> this.changeColor('red')}**

**>变成红色</button>**

**<button onClick = {()=> this.changeColor('yellow')}>变成黄色</button>**

**{this.state.color !== 'yellow' && <Son color={this.state.color}/> }**

**</div>**

**)**

**}**

**changeColor(color){**

**this.setState({color})**

**}**

**componentDidMount(){**

**// 可以操作真实的DOM 初始化结束**

**console.log('Father : componentDidMount',document.querySelector('.Btn'))**

**}**

**// 当组件componentDidMount之后，就进入运行中阶段，此时只要数据(props,state)变化就会触发对应的生命周期钩子函数**

**shouldComponentUpdate(){**

**console.log("Father:shouldComponentUpdate");**

**console.log('Father:',arguments)**

**return true;**

**}**

**componentWillUpdate(){**

**console.log("Father:componentWillUpdate")**

**console.log('Father',arguments)**

**}**

**componentDidUpdate(){**

**console.log('Father:componentDidUpdate')**

**console.log('Father',arguments)**

**}**

**}**

**export default Father**

#### 子组件

import React , {Component} from 'react'

class Son extends Component {

constructor(props){

super(props)

this.state = {

style:{

width:'200px',

height:'200px',

backgroundColor:this.props.color

}

}

console.log('Children : constructor');

}

componentWillMount(){

console.log('Children : componentWillMount')

}

render(){

console.log('Childern : render')

return (

<div style={this.state.style}>

</div>

)

}

componentDidMount(){

console.log('Children : componentDidMount ')

}

// 当组件componentDidMount之后，就进入运行中阶段，此时只要数据(props,state)变化就会触发对应的生命周期钩子函数

// componentWillReceiveProps->shouldComponentUpdate->componentWillUpdate->render->componentDidUpdate

componentWillReceiveProps(props){

// 这个钩子函数中，新的属性还没挂载到this上，参数props是最新的props

console.log('Children : componentWillReceiveProps',arguments)

this.setState((prevState,props)=>{

console.log(prevState,props,'this.setState函数写法中接收的参数') // props 是最新的props

let newstyle = { ...prevState.style }

newstyle.backgroundColor = props.color

return {style:newstyle}

})

}

// 当属性或者状态变化后都会执行，需要返回true和false，默认返回true，控制是否执行下面的钩子函数（是否重新渲染）

shouldComponentUpdate(props, state){

// 此时，最新的属性和最新的状态依然没有挂载到this上，需要从函数中接收

console.log('Children:shouldComponentUpdate',arguments);

return true;

}

componentWillUpdate(props, state){

// 此时，最新的属性和最新的状态依然没有挂载到this上，需要从函数中接收 没什么作用

console.log('Children:componentWillUpdate',arguments)

}

componentDidUpdate (prevprops, prevstate) {

// 在这里可以得到重新渲染后的dom

if ( prevprops.color !== this.props.color ) {

// 此次组件rerender是因为color变化

}

console.log('Children:componentDidUpdate')

}

componentWillUnmount () {

// 当组件被销毁的时候会执行这个钩子函数

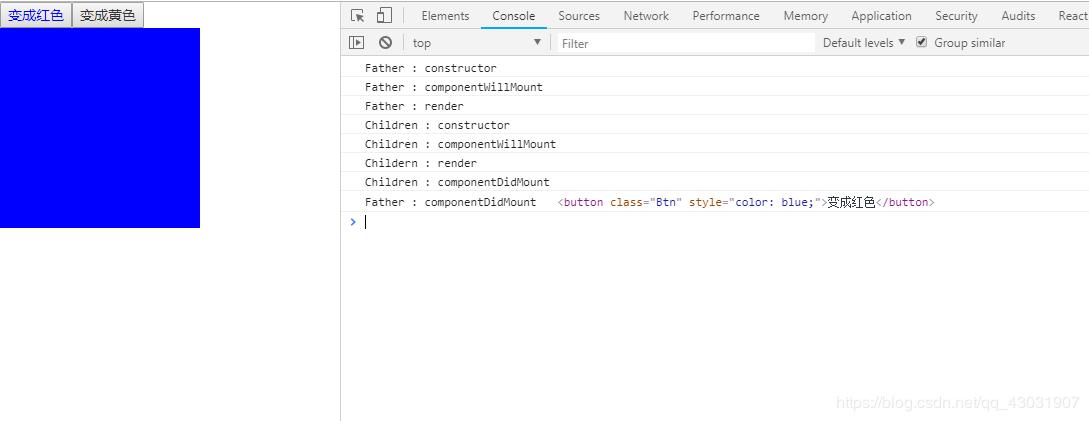
console.log('Children:componentWillUnmount')

}

}

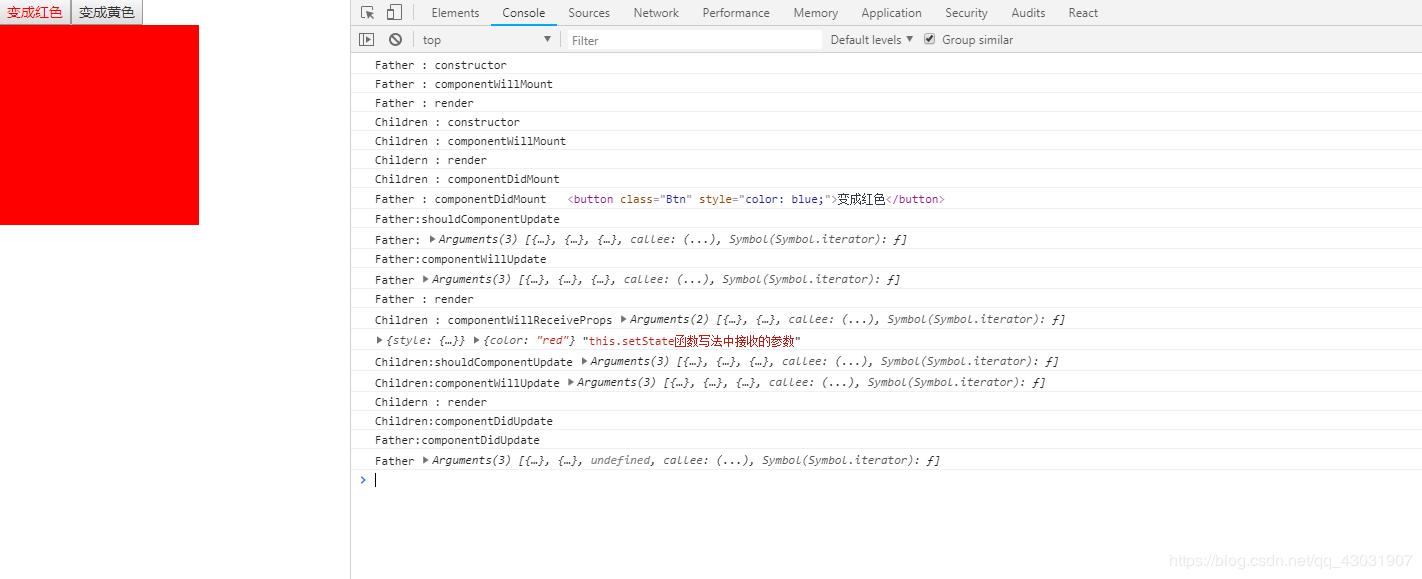
export default Son

#### 初始渲染



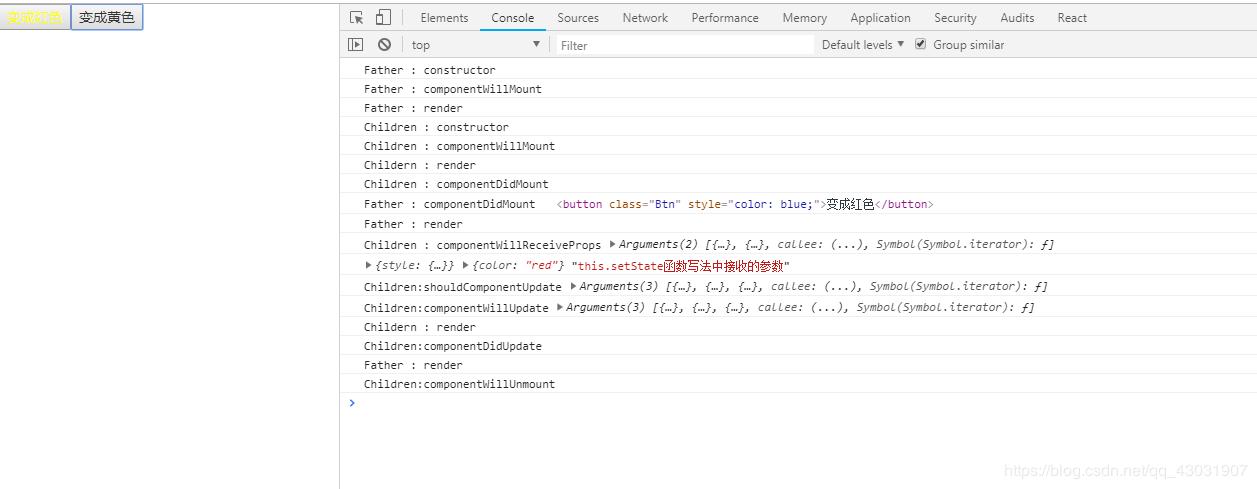
注意：父组件在render过程中会触发子组件的初始化阶段，当所有的子组件都装载（compoentDidMount）完成后，父组件才会执行自己的componentDidMount

#### 组件修改



当组件componentDidMount之后，就进入运行中阶段，此时只要数据(props,state)变化就会触发对应的生命周期钩子函数。  
会依次执行 componentWillReceiveProps->shouldComponentUpdate（scu）->componentWillUpdate->rende->componentDidUpdate

#### 组件销毁



当组件销毁后会执行组件的 componentWillMount 这个钩子函数。  
注意在React中组件销毁的时候连带dom一起消失的，所以没有销毁后的钩子

### 创建项目

#### 脚手架创建

安装脚手架

npm install -g create-react-app

使用脚手架创建项目：

create-react-app <项目名>

#### 使用webpack、babel、react来创建项目

首先创建项目文件夹

可以手动创建或者

mkdir my-app

然后进入项目文件夹目录

cd my-app

然后再项目文件夹目录下创建src文件夹用来存放代码

可以手动穿件文件夹或者

mkdir src

使用npm命令初始化项目

npm init -y

.安装webpack

首先需要安装webpack和webpack-cli，webpack-cli包含了webpack的众多指令

npm install webpack webpack-cli --save-dev

安装Babel

主要包含以下几个包

* @babel/core
* @babel/preset-env
* @babel/preset-react
* babel-loader

安装指令：

npm install @babel/core babel-loader @babel/preset-env @babel/preset-react --save-dev

安装完成后对babel配置

{

"presets": ["@babel/preset-env", "@babel/preset-react"]

}

Webpack配置

const path = require('path');

module.exports = {

entry: './src/main.js',

output: {

filename: 'bundle.js',

path: path.resolve(\_\_dirname, 'dist')

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: "babel-loader"

}

}

]

}

};

配置完成后在src目录下创建index.html/main.js/App.js等主要文件。

安装webpack相关html文件处理依赖：

需要安装html-webpack-plugin和html-loader

npm install html-webpack-plugin html-loader --save-dev

安装完后的配置

const path = require('path');

const HtmlWebPackPlugin = require("html-webpack-plugin");

module.exports = {

entry: './src/main.js',

output: {

filename: 'bundle.js',

path: path.resolve(\_\_dirname, 'dist')

},

module: {

rules: [

{

test: /\.js$/,

exclude: /node\_modules/,

use: {

loader: 'babel-loader'

}

},

{

test: /\.html$/,

use: {

loader: 'html-loader'

}

}

]

},

plugins: [

new HtmlWebPackPlugin({

titel: 'react app',

filename: 'index.html',

template: './src/index.html'

})

]

};

写代码前需要安装react和react-dom

npm install react react-dom --save

运行项目

需要安装webpack-dev-server用来启动本地服务器

npm install webpack-dev-server --save-dev

安装完成后在根目录创建配置package.json

"scripts": {

"start": "webpack-dev-server --open --mode development"

}

## 前端组件化