

## 资源

---

1. [react](#)
2. [create-react-app](#)

## 起步

---

1. 安装官方脚手架: `npm install -g create-react-app`
2. 创建项目: `create-react-app react-study`
3. 启动项目: `npm start`

## 文件结构

### 文件结构一览

— README.md	文档
— public	静态资源
— favicon.ico	
— index.html	
— manifest.json	
— src	源码
— App.css	
— App.js	根组件
— App.test.js	
— index.css	全局样式
— index.js	入口文件
— logo.svg	
— serviceWorker.js	pwa支持
— package.json	npm 依赖

### cra项目真容

使用 `npm run eject` 弹出项目真面目，会多出两个目录：

- ├─ config
  - ├─ env.js 处理.env环境变量配置文件
  - ├─ paths.js 提供各种路径
  - ├─ webpack.config.js webpack配置文件
  - └─ webpackDevServer.config.js 测试服务器配置文件
- ├─ scripts 启动、打包和测试脚本
  - ├─ build.js 打包脚本、
  - ├─ start.js 启动脚本
  - └─ test.js 测试脚本

env.js用来处理.env文件中配置的环境变量

```
// node运行环境: development、production、test等
const NODE_ENV = process.env.NODE_ENV;

// 要扫描的文件名数组
var dotenvFiles = [
  `${paths.dotenv}.${NODE_ENV}.local`, // .env.development.local
  `${paths.dotenv}.${NODE_ENV}`,      // .env.development
  NODE_ENV !== 'test' && `${paths.dotenv}.local`, // .env.local
  paths.dotenv, // .env
].filter(Boolean);

// 从.env*文件加载环境变量
dotenvFiles.forEach(dotenvFile => {
  if (fs.existsSync(dotenvFile)) {
    require('dotenv-expand')(
      require('dotenv').config({
        path: dotenvFile,
      })
    );
  }
});
```

实践一下，修改一下默认端口号，创建.env文件

```
PORT=8080
```

webpack.config.js 是webpack配置文件，开头的常量声明可以看出cra能够支持ts、sass及css模块化

```
// Check if TypeScript is setup
const useTypeScript = fs.existsSync(paths.appTsConfig);

// style files regexes
const cssRegex = /\.css$/;
const cssModuleRegex = /\.module\.css$/;
const sassRegex = /\.(scss|sass)$/;
const sassModuleRegex = /\.module\.scss$/;
```

## React和ReactDOM

删除src下面所有代码，新建index.js

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(<h1>React真帅</h1>, document.querySelector('#root'));
```

React负责逻辑控制，数据 -> VDOM

ReactDOM渲染实际DOM，VDOM -> DOM，如果换到移动端，就用别的库来渲染

React使用JSX来描述UI

入口文件定义，webpack.config.js

```
entry: [
  // webpackDevServer客户端，它实现开发时热更新功能
  isEnvDevelopment &&
  require.resolve('react-dev-utils/webpackHotDevClient'),
  // 应用程序入口: src/index
  paths.appIndexJs,
].filter(Boolean),
```

## JSX

JSX是一种JavaScript的语法扩展，其格式比较像模版语言，但事实上完全是在JavaScript内部实现的。

JSX可以很好地描述UI，能够有效提高开发效率

### 使用JSX

表达式{}的使用，index.js

```
const name = "react study";
const jsx = <h2>{name}</h2>;
```

函数也是合法表达式，index.js

```
const user = { firstName: "tom", lastName: "jerry" };
function formatName(user) {
  return user.firstName + " " + user.lastName;
}
const jsx = <h2>{formatName(user)}</h2>;
```

jsx是js对象，也是合法表达式，index.js

```
const greet = <p>hello, Jerry</p>
const jsx = <h2>{greet}</h2>;
```

条件语句可以基于上面结论实现，index.js

```
const showTitle = true;
const title = name ? <h2>{name}</h2> : null;
const jsx = (
  <div>
    {/* 条件语句 */}
    {title}
  </div>
);
```

数组会被作为一组子元素对待，数组中存放一组jsx可用于显示列表数据

```
const arr = [1,2,3].map(num => <li key={num}>{num}</li>)
const jsx = (
  <div>
    {/* 数组 */}
    <ul>{arr}</ul>
  </div>
);
```

属性的使用

```
import logo from "./logo.svg";

const jsx = (
  <div>
    {/* 属性：静态值用双引号，动态值用花括号；class、for等要特殊处理。 */}
    <img src={logo} style={{ width: 100 }} className="img" />
  </div>
);
```

css模块化，创建index.module.css，index.js

```
import style from "./index.module.css";
<img className={style.img} />
```

# 组件

## 组件的两种形式

### class组件

class组件通常**拥有状态和生命周期**，**继承于Component**，**实现render方法**

components/JsxTest.js

```
import React, { Component } from "react";
import logo from "../logo.svg";
import style from "../index.module.css";

export default class JsxTest extends Component {
  render() {
    const name = "react study";
    const user = { firstName: "tom", lastName: "jerry" };
    function formatName(user) {
      return user.firstName + " " + user.lastName;
    }
    const greet = <p>hello, Jerry</p>;
    const arr = [1, 2, 3].map(num => <li key={num}>{num}</li>);

    return (
      <div>
        { /* 条件语句 */ }
        {name ? <h2>{name}</h2> : null}
        { /* 函数也是表达式 */ }
        {formatName(user)}
        { /* jsx也是表达式 */ }
        {greet}
        { /* 数组 */ }
        <ul>{arr}</ul>
        { /* 属性 */ }
        <img src={logo} className={style.img} alt="" />
      </div>
    );
  }
}
```

### function组件

函数组件通常**无状态**，**仅关注内容展示**，**返回渲染结果即可**。

App.js

```
import React from "react";
import JsxTest from "../components/JsxTest";

function App() {
  return (
    <div>
      <JsxTest />
    </div>
  );
}

export default App;
```

## 组件状态管理

### 类组件中的状态管理

components/StateMgt.js

```
import React, { Component } from "react";

export default function StateMgt {
  return (
    <div>
      <Clock />
    </div>
  );
}
```

创建一个Clock组件

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    // 使用state属性维护状态，在构造函数中初始化状态
    this.state = { date: new Date() };
  }
  componentDidMount() {
    // 组件挂载时启动定时器每秒更新状态
    this.timerID = setInterval(() => {
      // 使用setState方法更新状态
      this.setState({
        date: new Date()
      });
    }, 1000);
  }
  componentWillUnmount() {
    // 组件卸载时停止定时器
  }
}
```

```

clearInterval(this.timerID);
}
render() {
  return <div>{this.state.date.toLocaleTimeString()}</div>;
}
}

```

## 拓展：setState特性讨论

- 用setState更新状态而不能直接修改

```
this.state.counter += 1; //错误的
```

- setState是批量执行的，因此对同一个状态执行多次只起一次作用，多个状态更新可以放在同一个setState中进行：

```

componentDidMount() {
  // 假如counter初始值为0，执行三次以后其结果是多少？
  this.setState({counter: this.state.counter + 1});
  this.setState({counter: this.state.counter + 1});
  this.setState({counter: this.state.counter + 1});
}

```

- setState通常是异步的，因此如果要获取到最新状态值有以下三种方式：

1. 传递函数给setState方法，

```

this.setState((state, props) => ({ counter: state.counter + 1})); // 1
this.setState((state, props) => ({ counter: state.counter + 1})); // 2
this.setState((state, props) => ({ counter: state.counter + 1})); // 3

```

2. 使用定时器：

```

setTimeout(() => {
  console.log(this.state.counter);
}, 0);

```

3. 原生事件中修改状态

```

componentDidMount(){
  document.body.addEventListener('click', this.changeValue, false)
}
changeValue = () => {
  this.setState({counter: this.state.counter+1})
  console.log(this.state.counter)
}

```

## 函数组件中的状态管理

```
import { useState, useEffect } from "react";

function ClockFunc() {
  // useState创建一个状态和修改该状态的函数
  const [date, setDate] = useState(new Date());
  // useEffect编写副作用代码
  useEffect(() => {
    // 启动定时器是我们的副作用代码
    const timerID = setInterval(() => {
      setDate(new Date());
    }, 1000);
    // 返回清理函数
    return () => clearInterval(timerID);
  }, []); // 参数2传递空数组使我们参数1函数仅执行一次
  return <div>{date.toLocaleTimeString()}</div>;
}
```

## 事件处理

React中使用onXX写法来监听事件。

范例：用户输入事件，创建EventHandle.js

```
import React, { Component } from "react";

export default class EventHandle extends Component {
  constructor(props) {
    super(props);

    this.state = {
      name: ""
    };

    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(e) {
    this.setState({ name: e.target.value });
  }
  render() {
    return (
      <div>
        {/* 使用箭头函数，不需要指定回调函数this，且便于传递参数 */}
        {/* <input
          type="text"
          value={this.state.name}
          onChange={e => this.handleChange(e)}
        /> */}
        {/* 直接指定回调函数，需要指定其this指向，或者将回调设置为箭头函数属性 */}
        <input
```



```

        type="text"
        value={this.state.name}
        onChange={this.handleChange}
      />
      <p>{this.state.name}</p>
    </div>
  );
}
}

```

## 组件通信

### Props属性传递

Props属性传递可用于父子组件相互通信

```

// index.js
ReactDOM.render(<App title="开课吧真不错" />, document.querySelector('#root'));

// App.js
<h2>{this.props.title}</h2>

```

如果父组件传递的是函数，则可以把子组件信息传入父组件，这个常称为状态提升，StateMgt.js

```

// StateMgt
<Clock change={this.onChange}/>

// Clock
this.timerID = setInterval(() => {
  this.setState({
    date: new Date()
  }, ()=>{
    // 每次状态更新就通知父组件
    this.props.change(this.state.date);
  });
}, 1000);

```

### context

跨层级组件之间通信

### redux

类似vuex，无明显关系的组件间通信

# 生命周期

React V16.3之前的生命周期

image-20190111173300397

范例：验证生命周期，创建Lifecycle.js

```
import React, { Component } from "react";
export default class Lifecycle extends Component {
  constructor(props) {
    super(props);
    // 常用于初始化状态
    console.log("1. 组件构造函数执行");
  }
  componentWillMount() {
    // 此时可以访问状态和属性，可进行api调用等
    console.log("2. 组件将要挂载");
  }
  componentDidMount() {
    // 组件已挂载，可进行状态更新操作
    console.log("3. 组件已挂载");
  }
  componentWillReceiveProps(nextProps, nextState) {
    // 父组件传递的属性有变化，做相应响应
    console.log("4. 将要接收属性传递");
  }
  shouldComponentUpdate(nextProps, nextState) {
    // 组件是否需要更新，需要返回布尔值结果，优化点
    console.log("5. 组件是否需要更新? ");
    return true;
  }
  componentWillUpdate() {
    // 组件将要更新，可做更新统计
    console.log("6. 组件将要更新");
  }
  componentDidUpdate() {
    // 组件更新
    console.log("7. 组件已更新");
  }
  componentWillUnmount() {
    // 组件将要卸载，可做清理工作
    console.log("8. 组件将要卸载");
  }
  render() {
    console.log("组件渲染");
    return <div>生命周期探究</div>;
  }
}
```

```
}  
}
```

激活更新阶段：App.js

```
class App extends Component {  
  state = { prop: "some content" };  
  componentDidMount() {  
    this.setState({ prop: "new content" });  
  }  
  render() {  
    return (  
      <div>  
        <Lifecycle prop={this.state.prop} />  
      </div>  
    );  
  }  
}
```

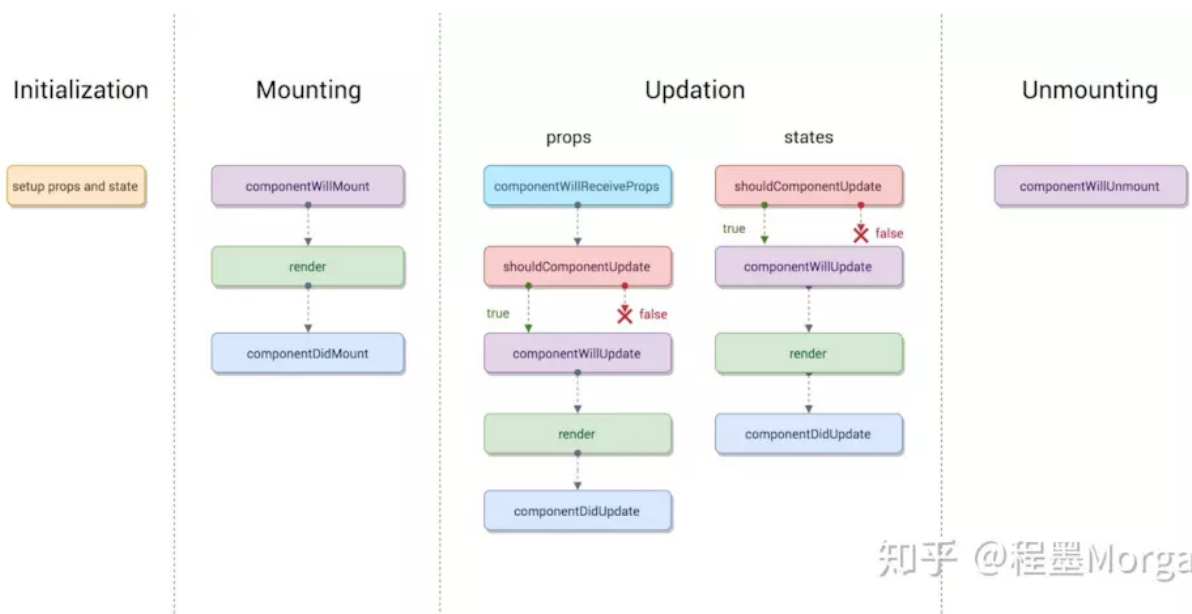
激活卸载阶段，App.js

```
class App extends Component {  
  state = { prop: "some content" };  
  componentDidMount() {  
    this.setState({ prop: "new content" });  
  
    setTimeout(() => {  
      this.setState({ prop: "" });  
    }, 2000);  
  }  
  render() {  
    return (  
      <div>  
        {this.state.prop && <Lifecycle prop={this.state.prop} />}  
      </div>  
    );  
  }  
}
```

## 生命周期探究

### React v16.0前的生命周期

大部分团队不见得会跟进升到16版本，所以16前的生命周期还是很有必要掌握的，何况16也是基于之前的修改



## 第一个是组件初始化(initialization)阶段

也就是以下代码中类的构造方法( `constructor()` ),`Test`类继承了`react Component`这个基类,也就继承这个`react`的基类,才能有`render()`,生命周期等方法可以使用,这也说明为什么 函数组件不能使用这些方法 的原因。

`super(props)` 用来调用基类的构造方法( `constructor()` ),也将父组件的`props`注入给子组件, 功子组件读取(组件中`props`只读不可变, `state`可变)。而 `constructor()` 用来做一些组件的初始化工作,如定义`this.state`的初始内容。

```
import React, { Component } from 'react';

class Test extends Component {
  constructor(props) {
    super(props);
  }
}
```

## 第二个是组件的挂载(Mounting)阶段

此阶段分为`componentWillMount`, `render`, `componentDidMount`三个时期。

- `componentWillMount`:

在组件挂载到`DOM`前调用,且只会被调用一次,在这边调用`this.setState`不会引起组件重新渲染,也可以把写在这边的内容提前到`constructor()`中,所以项目中很少用。

- `render`:

根据组件的`props`和`state`(无两者的重传递和重赋值,论值是否有变化,都可以引起组件重新`render`), `return`一个`React`元素(描述组件,即`UI`),不负责组件实际渲染工作,之后由`React`自身根据此元素去渲染出页面`DOM`。`render`是纯函数(`Pure function`: 函数的返回结果只依赖于它的参数;函数执行过程里面没有副作用),不能在里面执行`this.setState`,会有改变组件状态的副作用。

- `componentDidMount`:

组件挂载到DOM后调用，且只会被调用一次

### 第三个是组件的更新(update)阶段

在讲述此阶段前需要先明确下react组件更新机制。setState引起的state更新或父组件重新render引起的props更新，更新后的state和props相对之前无论是否有变化，都将引起子组件的重新render。详细可看[这篇文章](#)

#### 造成组件更新有两类（三种）情况：

- 1.父组件重新render

父组件重新render引起子组件重新render的情况有两种

a. 直接使用,每当父组件重新render导致的重传props，子组件将直接跟着重新渲染，无论props是否有变化。可通过shouldComponentUpdate方法优化。

```
class Child extends Component {
  shouldComponentUpdate(nextProps) { // 应该使用这个方法，否则无论props是否有变化都将会导致组件跟着重新渲染
    if(nextProps.someThings === this.props.someThings){
      return false
    }
  }
  render() {
    return <div>{this.props.someThings}</div>
  }
}
```

b.在componentWillReceiveProps方法中，将props转换成自己的state

```
class Child extends Component {
  constructor(props) {
    super(props);
    this.state = {
      someThings: props.someThings
    };
  }
  componentWillReceiveProps(nextProps) { // 父组件重传props时就会调用这个方法
    this.setState({someThings: nextProps.someThings});
  }
  render() {
    return <div>{this.state.someThings}</div>
  }
}
```

根据官网的描述

在该函数(componentWillReceiveProps)中调用 this.setState() 将不会引起第二次渲染。

是因为componentWillReceiveProps中判断props是否变化了，若变化了，this.setState将引起state变化，从而引起render，此时就没必要再做第二次因重传props引起的render了，不然重复做一样的渲染了。

- 2.组件本身调用setState，无论state有没有变化。可通过shouldComponentUpdate方法优化。

```

class Child extends Component {
  constructor(props) {
    super(props);
    this.state = {
      someThings:1
    }
  }
  shouldComponentUpdate(nextStates){ // 应该使用这个方法，否则无论state是否有变化都将会导致组件重新渲染
    if(nextStates.someThings === this.state.someThings){
      return false
    }
  }

  handleClick = () => { // 虽然调用了setState，但state并无变化
    const preSomeThings = this.state.someThings
    this.setState({
      someThings: preSomeThings
    })
  }

  render() {
    return <div onClick = {this.handleClick}>{this.state.someThings}</div>
  }
}

```

## 此阶段分为componentWillReceiveProps, shouldComponentUpdate, componentWillUpdate, render, componentDidUpdate

- componentWillMount(nextProps)

此方法只调用于props引起的组件更新过程中，参数nextProps是父组件传给当前组件的新props。但父组件render方法的调用不能保证重传给当前组件的props是有变化的，所以在此方法中根据nextProps和this.props来查明重传的props是否改变，以及如果改变了要执行啥，比如根据新的props调用this.setState出发当前组件的重新render

- shouldComponentUpdate(nextProps, nextState)

此方法通过比较nextProps, nextState及当前组件的this.props, this.state, 返回true时当前组件将继续执行更新过程，返回false则当前组件更新停止，以此可用来减少组件的不必要渲染，优化组件性能。

ps: 这边也可以看出，就算componentWillReceiveProps()中执行了this.setState，更新了state，但在render前（如shouldComponentUpdate, componentWillUpdate），this.state依然指向更新前的state，不然nextState及当前组件的this.state的对比就一直是true了。

- componentWillUpdate(nextProps, nextState)

此方法在调用render方法前执行，在这边可执行一些组件更新发生前的工作，一般较少用。

- render

render方法在上文讲过，这边只是重新调用。

- componentDidUpdate(prevProps, prevState)

此方法在组件更新后被调用，可以操作组件更新的DOM，prevProps和prevState这两个参数指的是组件更新前的props和state

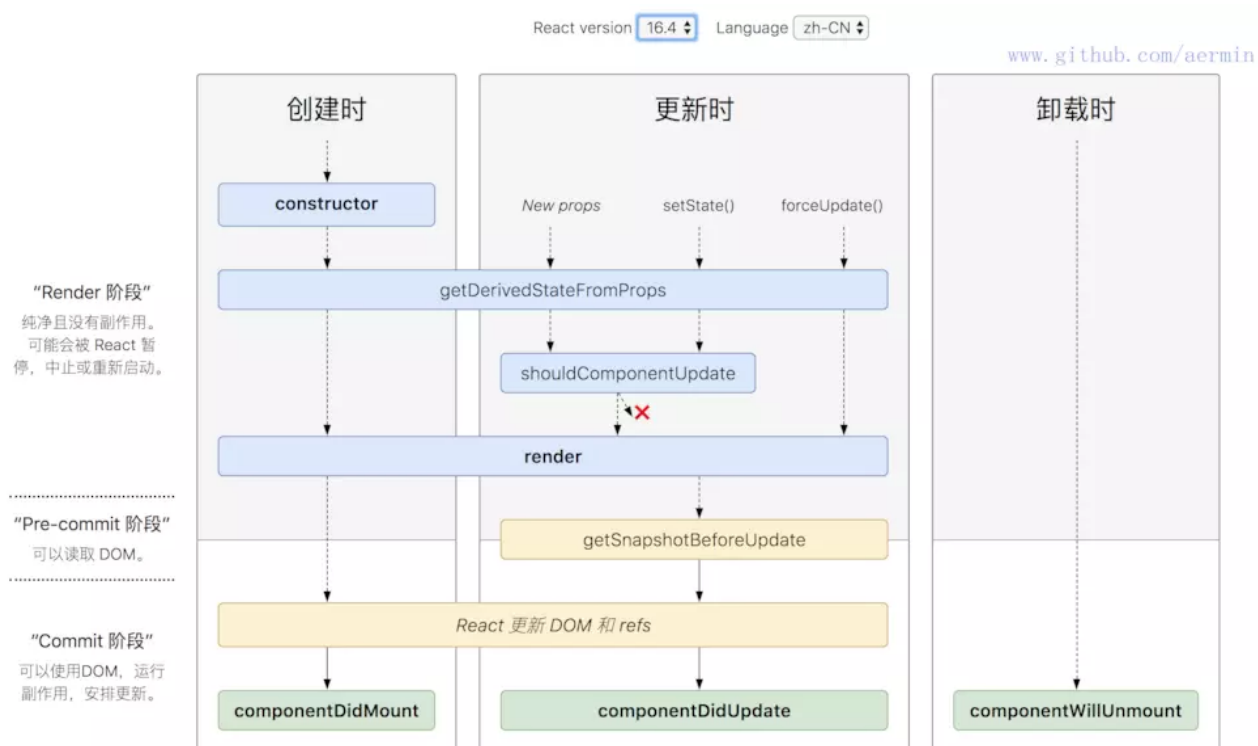
## 卸载阶段

此阶段只有一个生命周期方法：componentWillUnmount

- componentWillMount

此方法在组件被卸载前调用，可以在这里执行一些清理工作，比如清楚组件中使用的定时器，清楚componentDidMount中手动创建的DOM元素等，以避免引起内存泄漏。

## React v16.4 的生命周期



## 变更缘由

原来（React v16.0前）的生命周期在React v16推出的Fiber之后就不合适了，因为如果要开启async rendering，在render函数之前的所有函数，都有可能被执行多次。

原来（React v16.0前）的生命周期有哪些是在render前执行的呢？

- componentWillMount
- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate

如果开发者开了async rendering，而且又在以上这些render前执行的生命周期方法做AJAX请求的话，那AJAX将被无谓地多次调用。。。明显不是我们期望的结果。而且在componentWillMount里发起AJAX，不管多快得到结果也赶不上首次render，而且componentWillMount在服务器端渲染也会被调用到（当然，也许这是预期的结果），这样的IO操作放在componentDidMount里更合适。

禁止不能用比劝导开发者不要这样用的效果更好，所以除了shouldComponentUpdate，其他在render函数之前的所有函数（componentWillMount，componentWillReceiveProps，componentWillUpdate）都被getDerivedStateFromProps替代。

也就是用一个静态函数getDerivedStateFromProps来取代被deprecate的几个生命周期函数，就是强制开发者在render之前只做无副作用的操作，而且能做的操作局限在根据props和state决定新的state

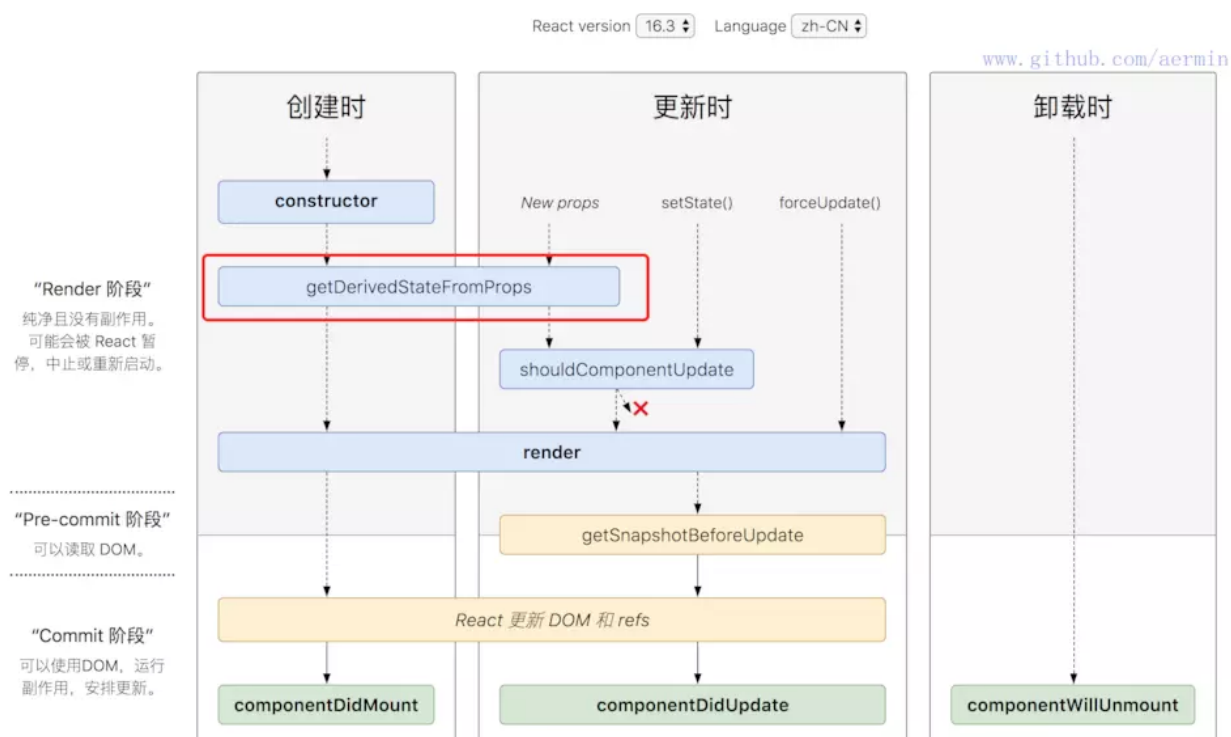
React v16.0刚推出的时候，是增加了一个componentDidCatch生命周期函数，这只是一个增量式修改，完全不影响原有生命周期函数；但是，到了React v16.3，大改动来了，引入了两个新的生命周期函数。

**新引入了两个新的生命周期函数：** `getDerivedStateFromProps`，`getSnapshotBeforeUpdate`

## getDerivedStateFromProps

**static getDerivedStateFromProps(props, state)** 在组件创建时和更新时的render方法之前调用，它应该返回一个对象来更新状态，或者返回null来不更新任何内容。

`getDerivedStateFromProps` 本来（React v16.3中）是只在创建和更新（由父组件引发部分），也就是不是由父组件引发，那么`getDerivedStateFromProps`也不会被调用，如自身`setState`引发或者`forceUpdate`引发。



React v16.3



这样的话理解起来有点乱，在React v16.4中改正了这一点，让`getDerivedStateFromProps`无论是Mounting还是Updating，也无论是因为什么引起的Updating，全部都会被调用，具体可看React v16.4 的生命周期图。

## getSnapshotBeforeUpdate

**getSnapshotBeforeUpdate()** 被调用于render之后，可以读取但无法使用DOM的时候。它使您的组件可以在可能更改之前从DOM捕获一些信息（例如滚动位置）。此生命周期返回的任何值都将作为参数传递给`componentDidUpdate()`。

官网给的例子：

```
class ScrollingList extends React.Component {
  constructor(props) {
    super(props);
    this.listRef = React.createRef();
  }

  getSnapshotBeforeUpdate(prevProps, prevState) {
    //我们是否要添加新的 items 到列表？
    // 捕捉滚动位置，以便我们可以稍后调整滚动。
    if (prevProps.list.length < this.props.list.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
  }

  componentDidUpdate(prevProps, prevState, snapshot) {
    //如果有snapshot值，我们已经添加了 新的items。
    // 调整滚动以至于这些新的items 不会将旧items推出视图。
    // （这边的snapshot是 getSnapshotBeforeUpdate方法的返回值）
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }

  render() {
    return (
      <div ref={this.listRef}>{/* ...contents... */}</div>
    );
  }
}
```