

React核心入门

React核心入门

课堂目标

资源

起步

文件结构

文件结构一览

React和ReactDOM

JSX

使用JSX

组件

组件的两种形式

class组件

function组件

组件状态管理

类组件中的状态管理

函数组件中的状态管理

事件处理

组件通信

props属性传递

context

redux

生命周期

变更缘由

新引入了两个新的生命周期函

数： `getDerivedStateFromProps`， `getSnapshotBeforeUpdate`

`getDerivedStateFromProps`

`getSnapshotBeforeUpdate`

验证生命周期

后续

回顾

课堂目标

1. create-react-app使用
2. 掌握组件使用
3. 掌握JSX语法
4. 掌握setState
5. 理解事件处理、组件生命周期
6. 掌握组件通信各种方式

资源

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

起步

1. 安装官方脚手架: `npm install -g create-react-app`
2. 创建项目: `create-react-app lesson1` 或者不执行步骤1 直接执行 ``npx create-react-app lesson1``
3. 打开lesson1: `cd lesson1`
4. 启动项目: `npm start`
5. 暴露配置项: `npm run eject`

如果出现错误类似 `/babel-preset-react-app/node_modules/@babel/runtime/helpers/slicedToArray' at webpackMissingModule` , 就 **`npm add @babel/runtime`**

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

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';

// 这里怎么没有出现React字眼?
// JSX => React.createElement(...)
ReactDOM.render(<h1>Hello React</h1>,
document.querySelector('#root'));
```

React负责逻辑控制，数据 -> VDOM

ReactDOM渲染实际DOM，VDOM -> DOM

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，体验[JSX](#)

JSX实质就是React.createElement的调用，最终的结果是React“元素”（JavaScript对象）。

```
const jsx = <h2>react study</h2>;  
ReactDOM.render(jsx,  
  document.getElementById('root'));
```

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} />
```

或者npm install sass -D

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

更多css modules规则[参考](#)

组件

组件是抽象的独立功能模块，react应用程序由组件构建而成。

组件的两种形式

组件有两种形式：**function**组件和**class**组件。

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，创建pages/ClassComponent.js

提取前面jsx相关代码至pages/ClassComponent.js

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

export default class ClassComponent 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}
      </div>
    );
  }
}
```

```

    { /* 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>
  );
}
}

```

创建并指定src/App.js为根组件

```

import React, { Component } from 'react';
import ClassComponent from
'./pages/ClassComponent';

class App extends Component {
  render() {
    return (
      <div>
        <ClassComponent/>
      </div>
    );
  }
}

```

```
export default App;
```

index.js中使用App组件

```
import App from "../App";

ReactDOM.render(<App />,
  document.getElementById("root"));
```

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。

改造App.js

```
import React from "react";
import FuncCmp from "../pages/FuncCmp";

function App() {
  return (
    <div className="App">
      <ClassCmp />
    </div>
  );
}

export default App;
```

从React16.8开始引入了hooks，函数组件也能够拥有状态，后面组件状态管理部分讨论

组件状态管理

如果组件中数据会变化，并影响页面内容，则组件需要拥有状态（state）并维护状态。

类组件中的状态管理

class组件使用state和setState维护状态

创建一个Clock

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

export default class ClassComponent 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() {
    // 组件挂载之后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);
  }
}
```

```
    });  
    }, 1000);  
}  
componentWillUnmount() {  
    // 组件卸载前停止定时器  
    clearInterval(this.timerID);  
}  
componentDidUpdate() {  
    console.log("componentDidUpdate");  
}  
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 +  
    2});  
}
```

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

2. 使用定时器:

```
setTimeout(() => {  
  this.changeValue();  
  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)
}
```

总结： `setState`只有在合成事件和生命周期函数中是异步的，在原生事件如`addEventListener`和`setTimeout`、`setInterval`中都是同步的。

拓展：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，其结果自然就是同步的了。

函数组件中的状态管理

函数组件通过hooks api维护状态


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

export default function FuncCmp() {
  const [date, setDate] = useState(new Date());
  useEffect(() => {
    const timeId = setInterval(() => {
      setDate(new Date());
    }, 1000);
    return () => clearInterval(timeId);
  });

  return <div>{date.toLocaleTimeString()}</div>;
}
```

提示: 如果你熟悉 React class 的生命周期函数, 你可以把 `useEffect` Hook 看做

`componentDidMount`, `componentDidUpdate` 和 `componentWillUnmount` 这三个函数的组合。

hooks api后面课程会继续深入讲解。

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Search.js

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

```
export default class Search extends Component {
  constructor(props) {
    super(props);
    this.state = { name: "" };
    // this.change = this.change.bind(this);
  }
  btn = () => {
    //使用箭头函数，不需要指定回调函数this，且便于传递参数
    console.log("btn");
  };
  change = e => {
    let value = e.target.value;
    this.setState({
      name: value,
    });
    console.log("name", this.state.name);
  };
  render() {
    const { name } = this.state;
    return (
      <div>
        <button onClick={this.btn}>按钮</button>
        <input
          type="text"
          placeholder="请输入"
          name={name}
          onChange={this.change}
        />
      </div>
    );
  }
}
```

```
        </div>
      );
    }
  }
}
```

事件回调函数注意绑定this指向，常见三种方法：

1. 构造函数中绑定并覆盖：this.change = this.change.bind(this)
2. 方法定义为箭头函数：change = ()=>{}
3. 事件中定义为箭头函数：onChange={()=>this.change()}

react里遵循单项数据流，没有双向绑定，输入框要设置value和onChange，称为受控组件

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

主要用于组件库开发中，后面组件化内容中详细介绍

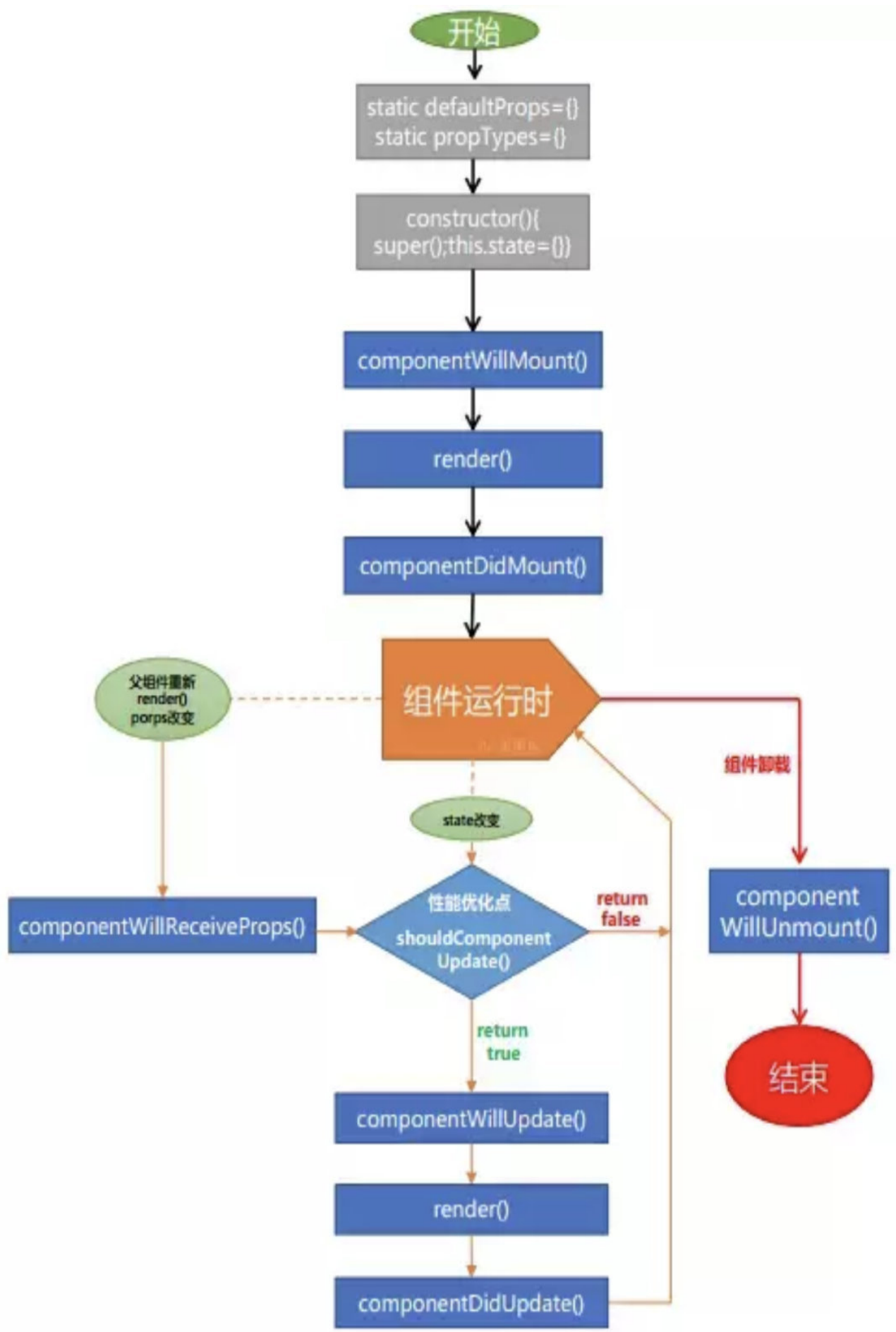
redux

类似vuex，无明显关系的组件间通信

后面全家桶部分详细介绍

生命周期

React V16.3之前的生命周期



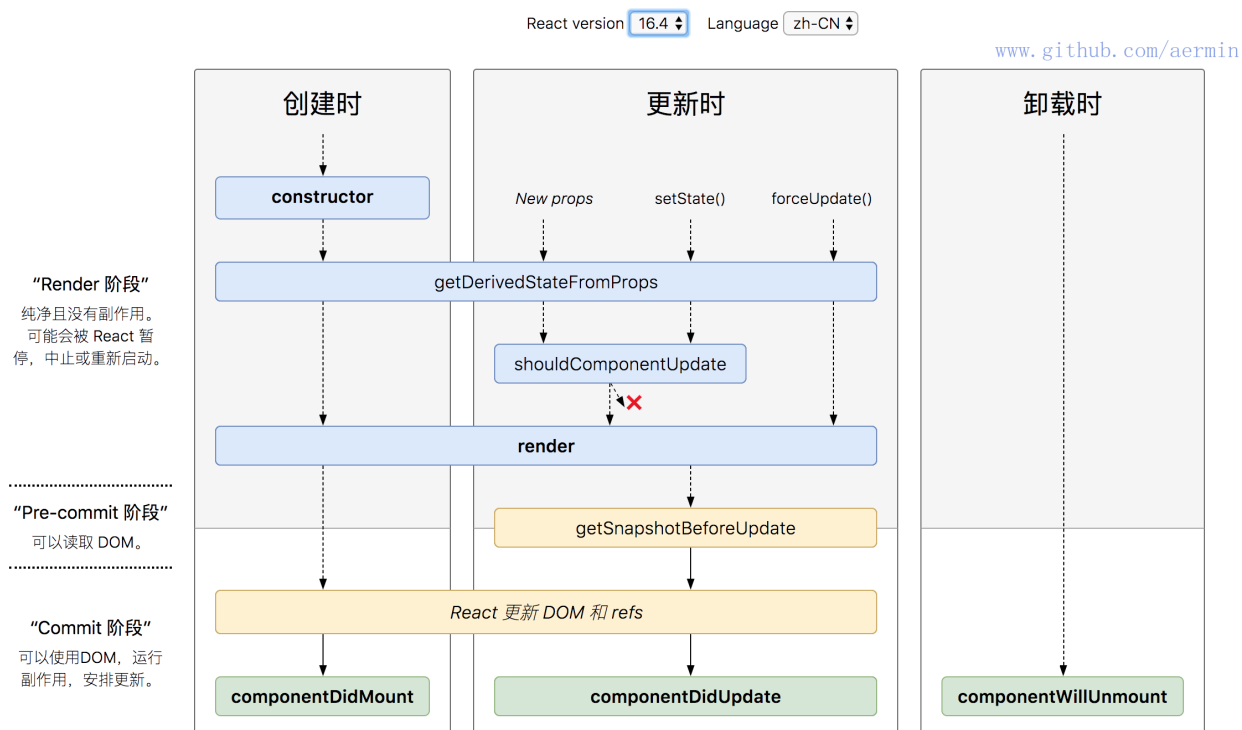
V16.4之后的生命周期：

V17可能会废弃的三个生命周期函数用
getDerivedStateFromProps替代，目前使用的话加上
UNSAFE_：

- componentWillMount
- componentWillReceiveProps
- componentWillUpdate

引入两个新的生命周期函数：

- static getDerivedStateFromProps
- getSnapshotBeforeUpdate



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`

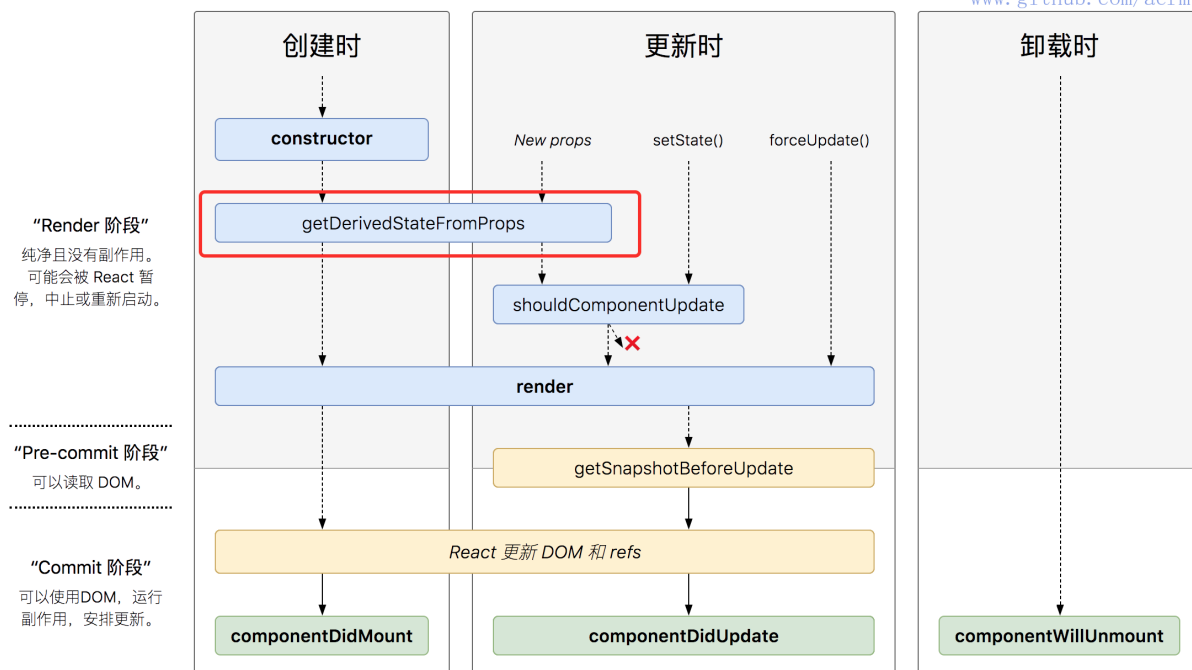
```
static getDerivedStateFromProps(props, state)
```

`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。

请注意，不管原因是什么，都会在每次渲染前触发此方法。这与 `UNSAFE_componentWillReceiveProps` 形成对比，后者仅在父组件重新渲染时触发，而不是在内部调用 `setState` 时。

[链接](#)

React v16.3 的生命周期图



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。

React v16.4后的`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。

`getSnapshotBeforeUpdate`

```
getSnapshotBeforeUpdate(prevProps, prevState)
```

在render之后，在componentDidUpdate之前。

`getSnapshotBeforeUpdate()` 在最近一次渲染输出（提交到 DOM 节点）之前调用。它使得组件能在发生更改之前从 DOM 中捕获一些信息（例如，滚动位置）。此生命周期的任何返回值将作为参数传递给 `componentDidUpdate()`。

此用法并不常见，但它可能出现在 UI 处理中，如需要以特殊方式处理滚动位置的聊天线程等。

应返回 snapshot 的值（或 `null`）。

[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>
    );
}
```

在上述示例中，重点是从 `getSnapshotBeforeUpdate` 读取 `scrollHeight` 属性，因为“render”阶段生命周期（如 `render`）和“commit”阶段生命周期（如 `getSnapshotBeforeUpdate` 和 `componentDidUpdate`）之间可能存在延迟。

验证生命周期

范例：创建Lifecycle.js

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

v17可能会废弃的三个生命周期函数用

getDerivedStateFromProps替代，目前使用的话加上

UNSAFE_：

- componentWillMount
- componentWillReceiveProps
- componentWillUpdate

```
*/
```

```
export default class LifeCycle extends Component  
{
```

```
  constructor(props) {  
    super(props);  
    this.state = {  
      counter: 0,  
    };  
    console.log("constructor",  
this.state.counter);  
  }
```

```
  static getDerivedStateFromProps(props, state) {  
    // 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。

```
    const { counter } = state;
```

```
    console.log("getDerivedStateFromProps",
counter);
    return counter < 8 ? null : { counter: 0 };
}
getSnapshotBeforeUpdate(prevProps, prevState) {
    const { counter } = prevState;
    console.log("getSnapshotBeforeUpdate",
counter);
    return null;
}
/* UNSAFE_componentWillMount() {
    //不推荐, 将会被废弃
    console.log("componentWillMount",
this.state.counter);
} */
componentDidMount() {
    console.log("componentDidMount",
this.state.counter);
}
componentWillUnmount() {
    //组件卸载之前
    console.log("componentWillUnmount",
this.state.counter);
}
/* UNSAFE_componentWillUpdate() {
    //不推荐, 将会被废弃
    console.log("componentWillUpdate",
this.state.counter);
} */
componentDidUpdate() {
```

```
    console.log("componentDidUpdate",
this.state.counter);
}

shouldComponentUpdate(nextProps, nextState) {
    const { counter } = this.state;
    console.log("shouldComponentUpdate", counter,
nextState.counter);
    return counter !== 5;
}

setCounter = () => {
    this.setState({
        counter: this.state.counter + 1,
    });
};

render() {
    const { counter } = this.state;
    console.log("render", this.state);
    return (
        <div>
            <h1>我是LifeCycle页面</h1>
            <p>{counter}</p>
            <button onClick={this.setCounter}>改变
counter</button>
            { /* {!!(counter % 2) && <Foo /> } */ }
            <Foo counter={counter} />
        </div>
    );
}
```

```

    }
}

class Foo extends Component {
  UNSAFE_componentWillReceiveProps(nextProps) {
    //不推荐，将会被废弃
    // UNSAFE_componentWillReceiveProps() 会在已挂
    载的组件接收新的 props 之前被调用
    console.log("Foo componentWillReceiveProps");
  }
  componentWillUnmount() {
    //组件卸载之前
    console.log(" Foo componentWillUnmount");
  }
  render() {
    return (
      <div
        style={{ border: "solid 1px black",
margin: "10px", padding: "10px" }}
      >
        我是Foo组件
        <div>Foo counter: {this.props.counter}
      </div>
    </div>
    );
  }
}

```


后续

组件

回顾

React核心入门

- 课堂目标

- 资源

- 起步

 - 文件结构

 - 文件结构一览

 - React和ReactDOM

 - JSX

 - 使用JSX

 - 组件

 - 组件的两种形式

 - class组件

 - function组件

 - 组件状态管理

 - 类组件中的状态管理

 - 函数组件中的状态管理

 - 事件处理

 - 组件通信

 - props属性传递

 - context

 - redux

生命周期

变更缘由

新引入了两个新的生命周期函数：

`getDerivedStateFromProps`, `getSnapshotBeforeUpdate`

`getDerivedStateFromProps`

`getSnapshotBeforeUpdate`

验证生命周期

后续

回顾