React 核心概念_2

> 阅读本文前置条件，对 React 相关概念有一定的了解：例如 state / props是什么；简单使用过 hooks；使用过 React 生命周期。

1. 数据流

> 这里只讲 React 自身的数据管理，状态管理器例如 redux，mobx等会在后面的章节介绍

（1）为什么有 state 和 props

> **因为React 组件可以大致分为两类：**

- Fool Component（容器组件）：自身的 UI 层由自身的 state 控制
- Smart Component（展示组件）：自身的 UI 层不仅由自身的 state 控制，还可以被传入的 props 控制

携程前端团队说的一句话我十分赞成：“容器组件负责处理复杂的业务逻辑以及数据，展示组件负责处理UI层”。

所以在容器组件中，对于 state 的管理和处理就十分重要了，它的职责就是把 state 管理得当，**并通过不同的方式对 state 进行处理后 变成 props** 传入每个对应的展示组件。

（2）强调单向数据流

单向数据流的一大好处就是所有的状态改变(mutation)可追溯。

举个例子，父组件维护了一个状态，假设子组件可随意更改父组件甚至祖宗组件的状态，那各个组件的状态改变就会变得难以追溯，父组件的状态也可能被子组件意外修改而不可察觉。而单向数据流保证了父组件的状态不会被子组件意外修改，如果要修改，只能通过在子组件中dispatch一个action来

对全局状态修改，全局状态在通过props分发给子组件；又或是调用父组件的方法；又或是发事件，这些操作是肉眼可见且可控的。不至于造成状态总被意外修改而导致难以维护的情况

> 为什么前端react，vue框架都是单向数据流? - 知乎

（3）Props 使用哲学

- **永远不要尝试去手动修改传入的 props，永远！**
- **派生状态应该被谨慎地使用，即一般避免这种情况：**

> 把 props 赋值给 state，且每次 props 更改时，state 也更改

React 官网如此说明：

> 派生状态会导致代码冗余，并使组件难以维护（**因为如果不同再将新的 props 更新到相应的 state 上去。这样做一来会破坏 state 数据的单一数据源，导致组件状态变得不可预测**）。
>
> 确保你已熟悉这些简单的替代方案：
>
> - 如果你需要**执行副作用**（例如，数据提取或动画）以响应 props 中的更改，请改用 `componentDidUpdate`。
> - 如果只想在 **prop 更改时重新计算某些数据**，请使用 memoization helper 代替。
> - 如果你想**在 prop 更改时“重置”某些 state**，请考虑使组件完全受控或使用 `key` 使组件完全不受控 代替

（4）State 使用哲学

- **不要直接修改 state**

> 1. 不会生效
> 2. **在未来某个时候，如果通过 setState 改变了 state ， 那么这次未通过 setState 去改变的 state 将会同样生效（不可知的 bug）。**

- **setState 不会立即生效。出于性能考虑，React 可能会把多个 setState 调用合并成一个调用。**

> 所以你不要依赖他们的值来更新下一个状态。如果必要，请让 setState 接收一个函数，而不是一个对象。这个函数用上一个 state 作为第一个参数，将此次更新被应用时的 props 作为第二个参数。

- 不要在 **componentWillUpdate（已废弃）** 中使用 setState，会触发无限循环

---

2. 生命周期

> Hooks 之前，React 有着传统意义上的生命周期
>
> 引入 Fiber 之后，有些生命周期已经不太适合用在新的架构上，于是React 官方为了避免开发者习惯性地在一些已经废弃的生命函数书写代码，降低心智负担，废弃了三个生命周期：
>
> **componentWillMount，componentWillUpdate，componentWillReceiveProps**
>
> 但是 hooks 之后，个人理解是没有所谓的生命周期了，函数式组件受数据驱动更新的思想代替了 class 组件的生命周期函数
>
> 好文欣赏：React v16.3之后的组件生命周期函数

（1）React v16 之前的生命周期

- **初始化阶段**
  a. **constructor() 函数，将父组件的props注入给子组件，并做一些初始化 state 的功能。**

- **挂载阶段**
  a. componentWillMount
  b. render
  c. componentDIdMount

- 更新阶段
  a. componentWillReceiveProps（**这个函数不会被 shouldComponentUpdate 影响到，只要父组件渲染，就执行这个函数，而本身组件 update，并不会触发这个函数**）
  b. shouldComponentUpdate

c. componentWillUpdate
d. render
e. componentDidUpdate

- 卸载
  a. componentWillUnmount

（2）React v16 之后删除的生命周期

> 原来的生命周期在React v16推出的 Fiber 架构之后就不合适了，因为如果要开启 async rendering，render函数之前的所有函数，都有可能被执行多次。
>
> 如果是 async render，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。

所以 **render 前执行的生命周期方法在新的架构下需要好好考虑以下生命函数存在的必要：**

- componentWillMount
- componentWillReceiveProps(nextProps)
- shouldComponentUpdate(nextProps, nextState)
- componentWillUpdate

由于 shouldComponentUpdate 不会有对 state / props 有副作用，所以 shouldComponentUpdate 这个函数可以得到保留。

（3）React v16 之后新增的生命周期

> 为了替代被废弃的生命周期函数：**componentWillMount，componentWillUpdate，componentWillReceiveProps；React 官方提供了一个新的 Api：**
>
> `static getDerivedStateFromProps(nextProps, preState)`

它可以返回一个对象来更新 state，或者返回 null 来表示不需要任何 state 的更新。

设计理念：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。

注意点：getDerivedStateFromProps 函数 和之前的 componentWillReceiveProps 不一样，无论是Mounting还是 Updating，也无论是因为什么引起的 Updating，都会调用这个函数，而且能做的操作局限在根据 props 和 state 决定新的 state。

（4）React v16 之后的生命周期总览

1. 初始化阶段
   a. constructor

2. 挂载阶段
   a. static getDerivedStateFromProps
   b. render
   c. componentDidMount

3. 更新阶段
   a. static getDerivedStateFromProps
   b. shouldComponentUpdate
   c. render
   d. getSnapshotBeforeUpdate
   e. componentDidUpdate

4. 卸载阶段
   a. componentWillUnmount

（5）React hooks 生命周期

> 本来函数式编程的思想决定了 **函数组件** 的本质是函数，没有 state 的概念的，因此**不存在生命周期**一说。
>
> 但是 React 引入 hooks 概念后，函数组件拥有了自己的一套数据流管理机制，而它所谓的生命周期，笔者认为更像提供一种受数据驱动而改变 UI 的策略，和传统意义上的生命周期不太一样。

- 挂载阶段
  a. useState 初始化数据

- 挂载 / 更新阶段
  a. getDerivedStateFromProps，可以通过直接比较进行模拟

JavaScript

```
function Demo(props) {
    const [prevRow, setPrevRow] = useState('');
    if (props.row !== prevRow) {
        setIsScrollingDown(prevRow !== null && row > prevRow);
        setPrevRow(row);
    }
}
```

  b. shouldComponentUpdate，可以用 React.memo代替，一些规则可以传入第二个参数
  c. render，和之前一样
  d. componentDidMount, componentDidUpdate，用 useEffect 配合依赖项

- 卸载阶段
  a. componentWillUnmount， useEffect 中返回的函数

（6）React hooks 之前和之后的版本对比

Hooks 之前 **Hooks之后**

	A	B
1	constructor	useState
2	getDerivedStateFromProps	useState 里面 update 函数
3	shouldComponentUpdate	useMemo
4	render	函数本身
5	componentDidMount	useEffect
6	componentDidUpdate	useEffect
7	componentWillUnmount	useEffect 里面返回的函数
8	componentDidCatch	无
9	getDerivedStateFromError	无

---

3. 对于派生状态使用的思考

你可能不需要派生状态

---

4. 事件系统

（1）设计动机

> React 想实现一个全浏览器的框架， 为了实现这种目标就需要提供全浏览器一致性的事件系统，以此抹平不同浏览器的差异

HTML

```
<button onClick={handleClick}>
  Activate Lasers
</button>
复制代码
```

我们已经知道这个 onClick 只是一个合成事件而不是原生事件， 那这段时间究竟发生了什么？ 原生事件和合成事件是如何对应起来的?

上面的代码看起来很简洁，实际上 React 事件系统工作机制比起上面要复杂的多，脏活累活全都在底层处理了， 简直框架劳模。其工作原理大体上分为两个阶段

- 事件绑定
- 事件触发

（2）如何绑定事件

> React 既然提供了合成事件，就需要知道合成事件与原生事件是如何对应起来的。React 在内部有一个插件：EventPlugin，事件插件可以认为是 React 将不同的合成事件处理函数封装成了一个模块，每个模块只处理自己对应的合成事件。

一个 plugin 就是一个对象， 这个对象包含了下面两个属性:

JavaScript

```
{
  eventTypes, // 一个数组，包含了所有合成事件相关的信息，包括其对应的原生事件关系
  extractEvents: // 一个函数，当原生事件触发时执行这个函数
}
```

- **事件绑定阶段**

> 1. React 执行 diff 操作，标记出哪些 **DOM 类型** 的节点需要添加或者更新
> 2. 当检测到需要创建一个节点或者更新一个节点时， 使用 **registrationNameModule** 查看一个 prop 是不是一个事件类型，如果是则执行下一步

3. 通过 **registrationNameDependencies** 检查这个 React 事件依赖了哪些**原生事件类型**

JavaScript

```
// registrationNameDependencies
{
    onBlur: ['blur'],
    onClick: ['click'],
    onClickCapture: ['click'],
    onChange: ['blur', 'change', 'click', 'focus', 'input', 'keydown', 'keyup', 'selectionchange'],
    onMouseEnter: ['mouseout', 'mouseover'],
    onMouseLeave: ['mouseout', 'mouseover'],
    ...
}
```

4. 检查这些一个或多个原生事件类型有没有注册过，如果有则忽略
5. 如果这个原生事件类型没有注册过，则注册这个原生事件到 `document` 上，回调为React提供的 dispatchEvent 函数

由👆上面我们可以得出一个结论:

- 我们将所有事件类型都注册到 document 上，类似于全局给我们做了一次事件委托
- 同一个类型的事件 React 只会绑定一次原生事件，例如无论我们写了多少个 onClick， 最终反应在 DOM 事件上只会有一个 listener。**React 只是会确保这个原生事件能够被它自己捕捉到，后续由 React 来派发我们的事件回调，当我们页面发生较大的切换时候，React 可以什么都不做，避免了添加监听器和去除监听器的传统操作，浪费性能**

（3）事件触发

事件触发可以大概分为下面几点（简单来说）

1. 触发事件，执行 dispatchEvent
2. 通过原生事件类型决定使用哪个合成事件类型
3. 如果对象池里有这个类型的实例，则取出这个实例，覆盖其属性，作为本次派发的事件对象（事件对象复用），若没有则新建一个实例（单例模式）
4. 找到一条触发链，即当前 DOM 到 最近的 React 组件这个路径，这个链就是我们要触发合成事件的链，(**只包含原生类型组件**)

> 5. 正反两个方向遍历这条链（捕获/冒泡），触发所有的 onClickCapture / onClick 事件

由👆上面，我们可以得出以下结论

- React 的冒泡和捕获并不是真正 DOM 级别的冒泡和捕获
- 事件只针对原生组件生效，自定义组件不会触发
- 当所有事件处理函数被调用之后，其所有属性都会被置空（事件池清空）。例如，以下代码是无效的（事件池爬）：

JavaScript

```
function handleChange(e) {
  // This won't work because the event object gets reused.
  setTimeout(() => {
    console.log(e.target.value); // Too late!
  }, 100);
}
```

（4）React v17 改变

- 支持了原生捕获事件的支持
- onScroll 事件不再进行事件冒泡
- 取消事件池，好耶！
- 事件委托挂载在 rootNode 上，而不是 document 上（官方解释是为了方便不同版本的 React 组件嵌套时事件处理不会出现问题，不是很懂
