结束语 | 聊聊 React 17，谈谈学习前端框架的心法

2020/12/30 修言

经过 23 个课时的学习，从基础到原理再到生产实践，相信此时此刻的你已经对 React 建立起了一个系统且深入的认知。前不久，React 17 版本正式面世，但 React 17 到底意味着什么，或许有不少同学还是一头雾水。因此，在专栏的最后一讲，我们先一起聊聊面向未来的 React——React 17 版本带来的改变。

聊完面向未来的 React，还要和你聊聊面向未来的学习——授人以鱼不如授人以渔，面对前端框架的更迭，技术的日新月异，我们应该如何确保自己能够保持住对新事物的敏锐度和理解能力呢？在本文的最后，我将对前端框架的学习方法论做一个提取，希望能帮助你建立良好的学习习惯。

React 17 带来了哪些改变

我们先来看看 React 官方是如何介绍 React 17 的：

> React v17 的发布非比寻常，因为它没有增加任何面向开发者的新特性。但是，**这个版本会使得 React 自身的升级变得更加容易**。
> 值得特别说明的是，React v17 作为后续版本的“基石”，它让不同版本的 React 相互嵌套变得更加容易。
> —— React 官方

React 17 中没有新特性，这是由它的定位决定的。React 17 的定位是**后续 18、19 等更新版本的“基石”**，它是一个“承上启下”的版本，用官方的说法来说，“**React v17 开启了 React 渐进式升级的新篇章**”。

所谓“渐进式升级”，是相对于“一次性升级”来说的。日后我们需要将项目从 React 17 迁移至 18、19 等更新版本时，不需要一口气把整个应用升级到新版本，而是可以部分升级，比如说我们完全可以在 React 18 中安全地引入 React 17 版本的某个组件。而在 React 17 之前，这样做将会伴随着不可用的风险，彼时我们但凡要升级 React 版本，就必须一次性将整个应用迁移至目标版本。

“渐进式升级”意味着更大的选择余地，它将在未来为大量的 React 老版本项目留出喘息的空间，确保开发者们不必为了兼容多版本而徒增烦恼。

没有新特性，不代表没有变化，更不代表没有东西可以学了。事实上，React 17 中仍然有不少值得我们关注的用户侧改变，个人认为最重要的是以下三点：

- 新的 JSX 转换逻辑
- 事件系统重构
- Lane 模型的引入

除此之外，React 17 中还有一些细节层面的变化，比如调整了 useEffect 钩子中清理副作用的时机，强化了组件返回 undefined 的错误校验等，这些也很有趣，大家课下可以多摸索摸索。

重构 JSX 转换逻辑

在过去，如果我们在 React 项目中写入下面这样的代码：

复制代码

```
function MyComponent() {
  return <p>这是我的组件</p>
}
```

React 是会报错的，原因是 React 中对 JSX 代码的转换依赖的是 React.createElement 这个函数。因此但凡我们在代码中包含了 JSX，那么就必须在文件中引入 React，像下面这样：

复制代码

```
import React from 'react';
function MyComponent() {
  return <p>这是我的组件</p>
}
```

而 React 17 则允许我们在不引入 React 的情况下直接使用 JSX。这是因为在 React 17 中，编译器会自动帮我们引入 JSX 的解析器，也就是说像下面这样一段逻辑：

复制代码

```
function MyComponent() {
  return <p>这是我的组件</p>
}
```

会被编译器转换成这个样子：

复制代码

```
import {jsx as _jsx} from 'react/jsx-runtime';
function MyComponent() {
  return _jsx('p', { children: '这是我的组件' });
}
```

react/jsx-runtime 中的 JSX 解析器将取代 React.createElement 完成 JSX 的编译工作，这个过程对开发者而言是自动化、无感知的。因此，新的 JSX 转换逻辑带来的最显著的改变就是**降低了开发者的学习成本**。

为什么说降低了学习成本呢？仔细回忆一下，当你入门 React 时，是不是很容易因为漏掉对 React 的引入从而引发 JSX 相关的问题？当这些问题发生的时候，作为新手难免会懵逼。直到你有一天成了进阶选手，理解了 JSX 和 React.createElement 之间的关系，你才恍然大悟，从此便永远地记住了“有 JSX 就必须有 React 引入”这条真理，于是你每次用 JSX 的时候都要手动 import 一下 React，虽极为不便，但也没有什么办法。

但如果你学习的是 React 17，那么从一开始你就不会感知到 React 需要和 JSX 一同引入这件事情，后续更不必去纠结背后的许多个为什么，因为 React 17 已经为你打点好了一切。

新的 JSX 转换逻辑告诉了我们一个道理——框架，并不是越复杂越难学才越牛。好的框架，甚至说好的“轮子”，追求的一定都是简单和稳定。

react/jsx-runtime 中的 JSX 解析器看上去似乎在调用姿势上和 React.createElement 区别不大，那么它是否只是 React.createElement 换了个马甲呢？当然不是，它在内部实现了 React.createElement 无法做到的性能优化和简化。在一定情况下，它可能会略微改善编译输出内容的大小。

事件系统重构

事件系统在 React 17 中的重构要从以下两个方面来看：

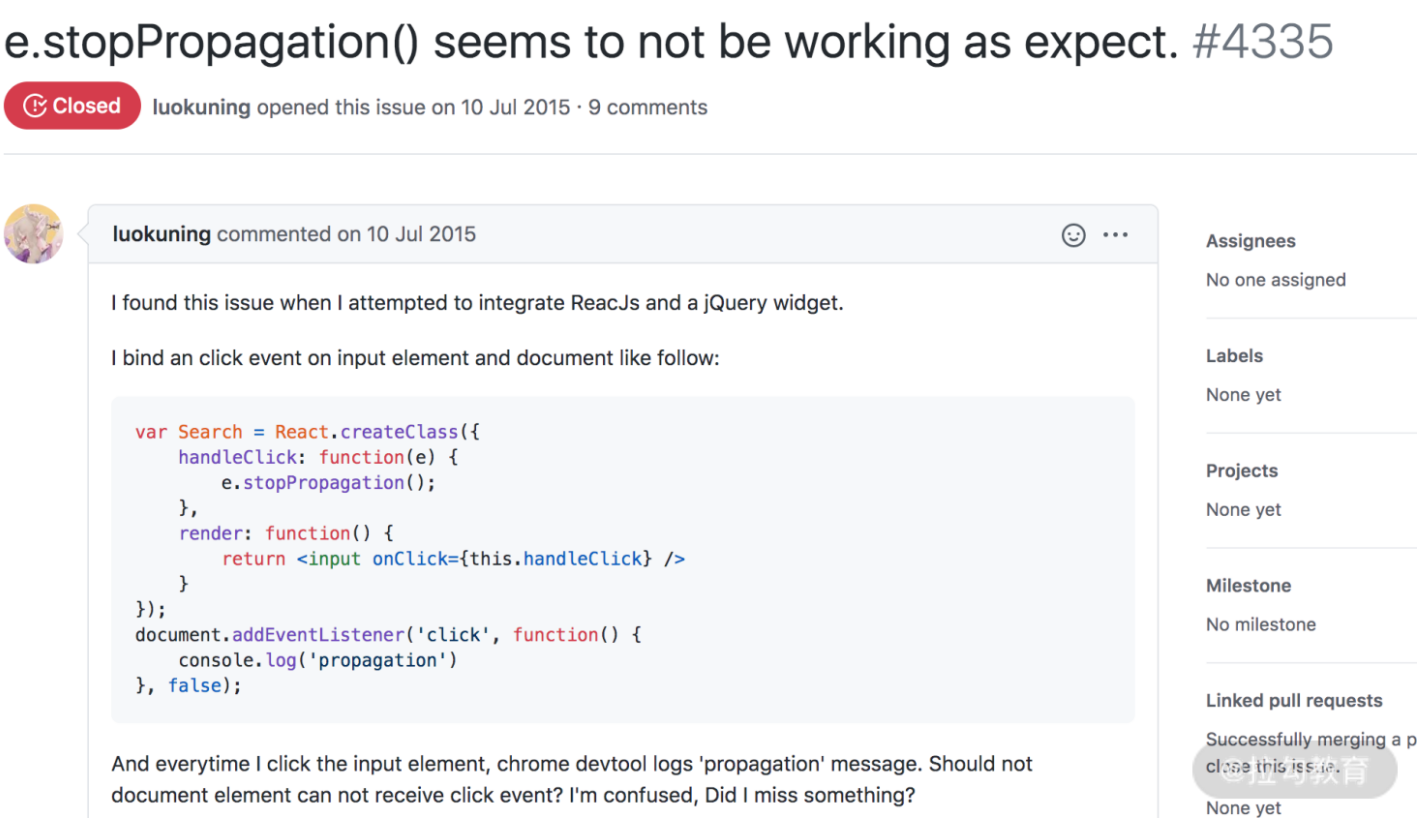
- **卸掉历史包袱**
- **拥抱新的潮流**

1. 卸掉历史包袱：放弃利用 document 来做事件的中心化管控

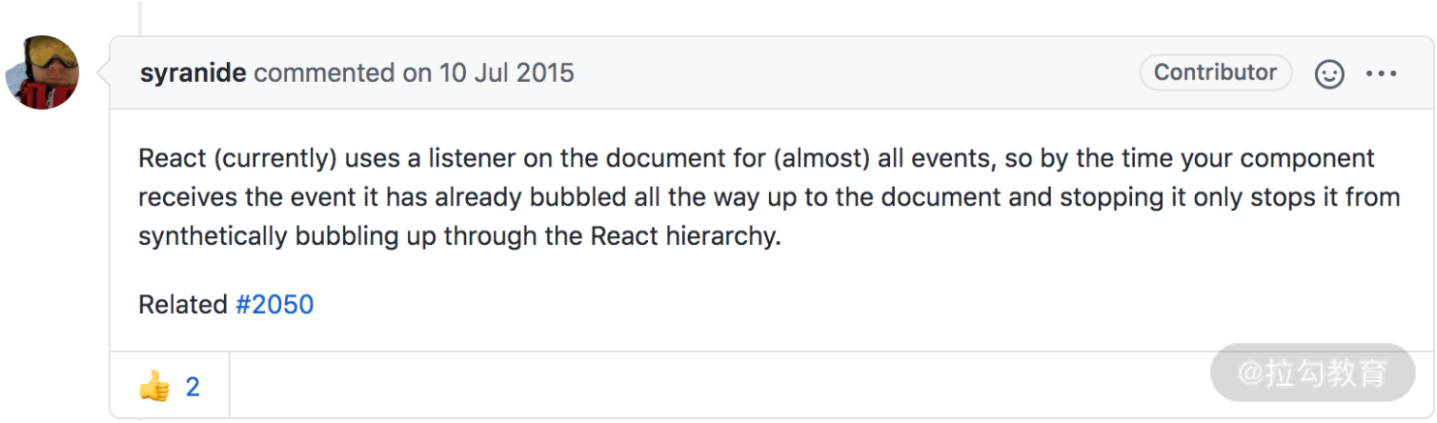
在本专栏的第 17 讲，相信你已经对 React 16.13.x 版本中的事件系统实现原理有了深入的了解。当时我们反复强调过，React 会**通过将所有事件冒泡到 document 来实现对事件的中心化管控**。

这样的做法虽然看上去已经足够巧妙，但仍然有它不聪明的地方——document 是整个文档树的根节点，操作 document 带来的影响范围实在是太大了，这将会使事情变得更加不可控。相关的 Bug 中最

出名的莫过于下图这个 GitHub issue 了：



提问者说他试图在 input 这个元素的 React 事件函数中阻止冒泡，但事实并没有如他所愿——每次点击 input 的时候，事件还是会被冒泡到 document 上去。对此，他得到的回复是这样的：



原因和大家想的一样：由于 React 依赖将 DOM 事件冒泡到 document 上来实现对所有事件的中心化管控。图中源代码中，作者在 handleClick 这个 React 事件函数中阻止了冒泡，这只能够保证该事件对应的合成事件在 React 事件体系下的冒泡被阻止了（也就是说 React 不会为这个合成事件模拟冒泡效果，关于“模拟冒泡”这个概念，第 17 讲中有源码级的原理分析，趁机拐回去复习一下吧^_^），并不能够阻止原生 DOM 事件的冒泡。因此我们安装在 document 上的事件监听器一定会被触发。

且不说 document 中心化管控这个设定给开发者带来了多大的限制，单看设计理念，就多少能够预感到其中的风险：document 是一个全局的概念，而组件只是全局的一个部分。由 React 组件引入的 React 事件系统，理论上看和组件绑在一起是最合适的，不应该将影响范围扩大到全局。

在 React 17 中，React 团队终于正面解决了这个问题：事件的中心化管控不会再全部依赖 document，管控相关的逻辑被转移到了每个 React 组件自己的容器 DOM 节点中。比如说我们在 ID 为 root 的 DOM 节点下挂载了一个 React 组件，像下面代码这样：

```
1. const rootElement = document.getElementById("root");
2. ReactDOM.render(<App />, rootElement);
```

复制代码

那么事件管控相关的逻辑就会被安装到 root 节点上去。这样一来， React 组件就能够自己玩自己的，再也无法对全局的事件流构成威胁了。

2. 拥抱新的潮流：放弃事件池

在 React 17 之前，合成事件对象会被放进一个叫作“事件池”的地方统一管理。这样做的目的是能够实现事件对象的复用，进而提高性能：每当事件处理函数执行完毕后，其对应的合成事件对象内部的所有属性都会被置空，意在为下一次被复用做准备。这也就意味着事件逻辑一旦执行完毕，我们就拿不到事件对象了，React 官方给出的这个例子就很能说明问题，请看下面这个代码：

```
1. function handleChange(e) {
2.   // This won't work because the event object gets reused.
3.   setTimeout(() => {
4.     console.log(e.target.value); // Too late!
5.   }, 100);
6. }
```

复制代码

异步执行的 setTimeout 回调会在 handleChange 这个事件处理函数执行完毕后执行，因此它拿不到想要的那个事件对象 e。

屏幕前面因为拿不到心仪的事件对象而写出过 Bug 的同学，请在心里默默地扣个 1——我相信如果文章有弹幕，此时一定能看到满屏的 1。事件池的这个设计，虽然利好了性能，却整懵了用户。很多人只有在写过无数 Bug 之后，才会后知后觉地发现，要想拿到目标事件对象，必须显式地告诉 React——我永远需要它，也就是调用 e.persist() 函数，像下面这样：

```
1. function handleChange(e) {
2.   // Prevents React from resetting its properties:
3.   e.persist();
```

复制代码

```
  setTimeout(() => {
    console.log(e.target.value); // Works
  }, 100);
}
```

在旧版本中，React 这样做在很大程度上是希望能能够对部分性能一般的老浏览器进行向下兼容。

但随着时代的发展，如今市面上的浏览器虽不能说是性能绝佳，但基本上也不会因为事件池里的对象多几个少几个就给你表演内存泄漏。因此，React 17 拥抱了新时代的潮流，重新在研发体验和向下兼容性能之间做了选择，这一次，它选择了前者——**放弃事件池，为每一个合成事件创建新的对象**。

因此在 React 17 中，我们不需要 e.persist()，也可以随时随地访问我们想要的事件对象。

Lane 模型的引入

如果你曾经尝试持续较长一段时间地关注并阅读 React 源码，会发现 React 16 中几乎每一个小版本的更新，都要动一动 Fiber Reconciler 相关的逻辑。这一点曾经让我感到很头痛，一度陷入“学不完了”的惆怅中。虽然变更仅仅发生在编码层面，并不至于影响核心思想，但这仍然会对源码学习者造成不小的压力。

为了避免你陷入类似的困境，本专栏涉及 Fiber 源码的几讲（第 13 ~ 16 讲），其中源码全部引用自 React 17——直接学最新的，是当下最保险的方式，这意味着我们捕获到的是 React 团队截至目前的“最佳思路”。

而在第 13 ~ 16 讲的分析中，我们其实就已经在源码的各个角落见过“Lane”这个概念，也分析过一些和 Lane 相关的一些函数。初学 React 源码的同学由此可能会很自然地认为：优先级就应该是用 Lane 来处理的。但事实上，React 16 中处理优先级采用的是 expirationTime 模型。

expirationTime 模型使用 expirationTime（一个时间长度） 来描述任务的优先级；而 Lane 模型则使用二进制数来表示任务的优先级：

> lane 模型通过将不同优先级赋值给一个位，通过 31 位的位运算来操作优先级。

Lane 模型提供了一个新的优先级排序的思路，相对于 expirationTime 来说，它对优先级的处理会更细腻，能够覆盖更多的边界条件。

如何深入了解一个前端框架

作为团队自研前端框架方向的负责人，我在实际工作中需要调研和深扒的框架类型可能会比大家想象的多得多。那么面对一个陌生的前端框架，我们应该怎样做才能够高效且平稳地完成从“小工”到“专家”的蜕变呢?

这个问题其实是没有标准答案的，它和每个人的学习习惯、学习效率甚至元认知能力都有关系。但我想总有一些具体到行为上的规律是可以复用的。今天我想和你分享的，就是一部分我在团队的新人包括实习生同学身上验证过的、可执行度较高的学习经验，希望能够对你日后的生涯道路有所帮助。

不要小看官方文档

在实际的读者调研中，我发现很多同学对 React 官方文档不够重视。大家习惯于在入门阶段借助文档完成“快速上手”，却忽视了文档所能够提供给我们的一些更有价值的信息——比如框架的设计思想、源码分层及一些对特殊功能点的介绍。

在专栏的更新过程中，我会在引用官方文档的地方标注出处，这促使了一部分同学去阅读一部分的文档内容，这是一件好事情。React 文档在前端框架文档中属于相当优秀的范本，如果你懂得利用文档，会发现它不只是一个 API 手册或是入门教程，而是一套成体系的官方教学。

如果专栏中的一些文档的摘要引用使你受用，不妨尝试去阅读一下完整的原文。在日常的源码阅读包括生产实践中，如果遇到了 React 相关的问题，请不要急于去阅读参差不齐的社区文章——先问问 React 文档试试看吧，或许你能收获的会比你想象中要多。

调用栈就是你的学习地图

若你的学习层次已经超越了阅读官方文档这个阶段，接下来可能会想要了解框架到底是如何运行的。此时你已经掌握了框架的设计理念和基本特性，也有了一些简单项目的实践经验，但或许还并不具备从头挑战源码的知识储备和心理准备。这时，在阅读源码之前，框架的函数调用栈将会给你指明许多方向性的问题。

比如当你想要了解 Hooks，那么就可以尝试去观察不同 Hooks 调用所触发的函数调用栈，从中找出出镜率最高的那些函数，它们大概率暗示着 Hooks 源码的主流程。事件系统、render 过程之类的也是同理。观察调用栈，寻找共性，然后点对点去阅读关键函数的源码，这将大大降低我们阅读源码的难度。

如何阅读源码

当你理解了一部分核心功能的源码逻辑之后，难免会对整个框架的运行机制产生好奇。这时候直接从入口文件出发去阅读所有的源码，仍然是一个不太明智的选择。

在整体阅读源码之前，我们最好去复习一下框架官方对框架架构设计、源码分层相关的介绍——这些信息未必会全部暴露在文档里，但借助搜索引擎，我们总能找到一些线索——比如框架作者/官方团队的博文，其内容的权威度基本和文档持平。

在理解了整个框架项目的架构分层之后，我们阅读源码的姿势就可以多样化一些了：可以尝试分层阅读，一次搞清楚一个大问题，最后再把整个思路按照架构分层的逻辑组合起来；也可以继续借助调用栈，通过观察一个完整的执行流程（比如 React 的首屏渲染过程）中所涉及的函数，自行将每个层次的逻辑对号入座，然后再向下拆分，我个人采用的就是这种办法。

总结

行文至此，我们对 React 世界的探索就要告一段落了。

通过对本专栏的学习，希望大家最终收获不仅仅是对 React 框架的通透理解，更有一套健康的、可持续的前端学习的方法论。

经过几年的角逐和沉淀，主流前端框架的稳定性逐渐在强化，三大框架都向着 Web Components 的标准演化，一切并非倾向于无序，而是变得越来越确定、越来越可预测。因此大家大可不必迷信舆论中对“学不完了”这种焦虑的渲染，更不必去盲目追逐新的花样。在当下的这种趋势下，很多时候若你能深入地吃透一个优秀的框架，就能够迅速地积累许多可以复用的理解经验，这将为你学习其他的新知识创造极大的加速度。

所以说，不要急，慢慢来，或许会比较快。

主流框架趋于稳定，而前端世界却仍在野蛮生长。近年来，Serverless、低代码、智能化、跨平台等新的概念层出不穷，每一个都足以将前端推入一个崭新的时代。面对层出不穷的技术热点，我们除了要保持冷静的头脑，还需要保持飞快的脚步，这一切的一切，都需要你主动去构建并持续地迭代属于自己的、行之有效的学习习惯。如果本专栏在这个方面能够帮到你，将是我最大的幸事。