React 的未来（1）： 拥抱异步渲染

在这一节中，我们会展望一下 React 的未来，不过，在向前看之前，我们要回顾一下 React 的历史。

React 简史

React 最初是 Facebook 的一个内部项目，投入实际产品是在 2011 年，用于 Facebook 的时间线界面，随后，Facebook 在其收购的 Instagram 中也使用了这种技术，2013 年 5 月，Facebook 决定将 React 开源。

有很长一段时间，React 一直是以 0.x.x 的模式来增长版本，大版本是 0，只有小版本和补丁版本迭代。直到 2016 年 4 月，React 经过这么长时间产品环境的磨练，已经足够成熟，所以版本一下子从 v0.14.8 跳到了 v15.0.0。其实这种版本改变的模式对开发者并没有什么直接影响，但是反映了 Facebook 致力于 React 开发的决心。

到了 v16.0.0 的时候，React 有一个重大改变——核心代码被重写，引入了叫 Fiber 这个全新的架构。这个架构使得 React 用异步渲染成为可能，但要注意，这个改变只是让异步渲染（async rendering）成为“可能”，React 却并没有在 v16 发布的时候立刻开启这种“可能”，也就是说，React 在 v16 发布之后依然使用的是同步渲染。不过，虽然异步渲染没有立刻采用，Fiber 架构还是打开了通向新世界的大门，React v16 一系列新功能几乎都是基于 Fiber 架构。

要面向 React 未来，我们首先要理解这个异步渲染的概念。

同步渲染的问题

长期以来，React 一直用的是同步渲染，这样对 React 实现非常直观方便，但是会带来性能问题。

假设有一个超大的 React 组件树结构，有 1000 个组件，每个组件平均使用 1 毫秒，那么，要做一次完整的渲染就要花费 1000 毫秒也就是 1 秒钟，然而 JavaScript 运行环境是单线程的，也就是说，React 用同步渲染方式，渲染最根部组件的时候，会同步引发渲染子组件，再同步渲染子组件的子组件……最后完成整个组件树。在这 1 秒钟内，同步渲染霸占 JavaScript 唯一的线程，其他的操作什么都做不了，在这 1 秒钟内，如果用户要点击什么按钮，或者在某个输入框里面按键，都不会看到立即的界面反应，这也就是俗话说的“卡顿”。

在同步渲染下，要解决“卡顿”的问题，只能是尽量缩小组件树的大小，以此缩短渲染时间，但是，应用的规模总是在增大的，不是说缩小就能缩小的，虽然我们利用定义 shouldComponentUpdate 的方法可以减少不必要的渲染，但是这也无法从根本上解决大量同步渲染带来的“卡顿”问题。

异步渲染：两阶段渲染

React Fiber 引入了异步渲染，有了异步渲染之后，React 组件的渲染过程是分时间片的，不是一口气从头到尾把子组件全部渲染完，而是每个时间片渲染一点，然后每个时间片的间隔都可去看看有没有更紧急的任务（比如用户按键），如果有，就去处理紧急任务，如果没有那就继续照常渲染。

根据 React Fiber 的设计，一个组件的渲染被分为两个阶段：第一个阶段（也叫做 render 阶段）是可以被 React 打断的，一旦被打断，这阶段所做的所有事情都被废弃，当 React 处理完紧急的事情回来，依然会重新渲染这个组件，这时候第一阶段的工作会重做一遍；第二个阶段叫做 commit 阶段，一旦开始就不能中断，也就是说第二个阶段的工作会稳稳当当地做到这个组件的渲染结束。

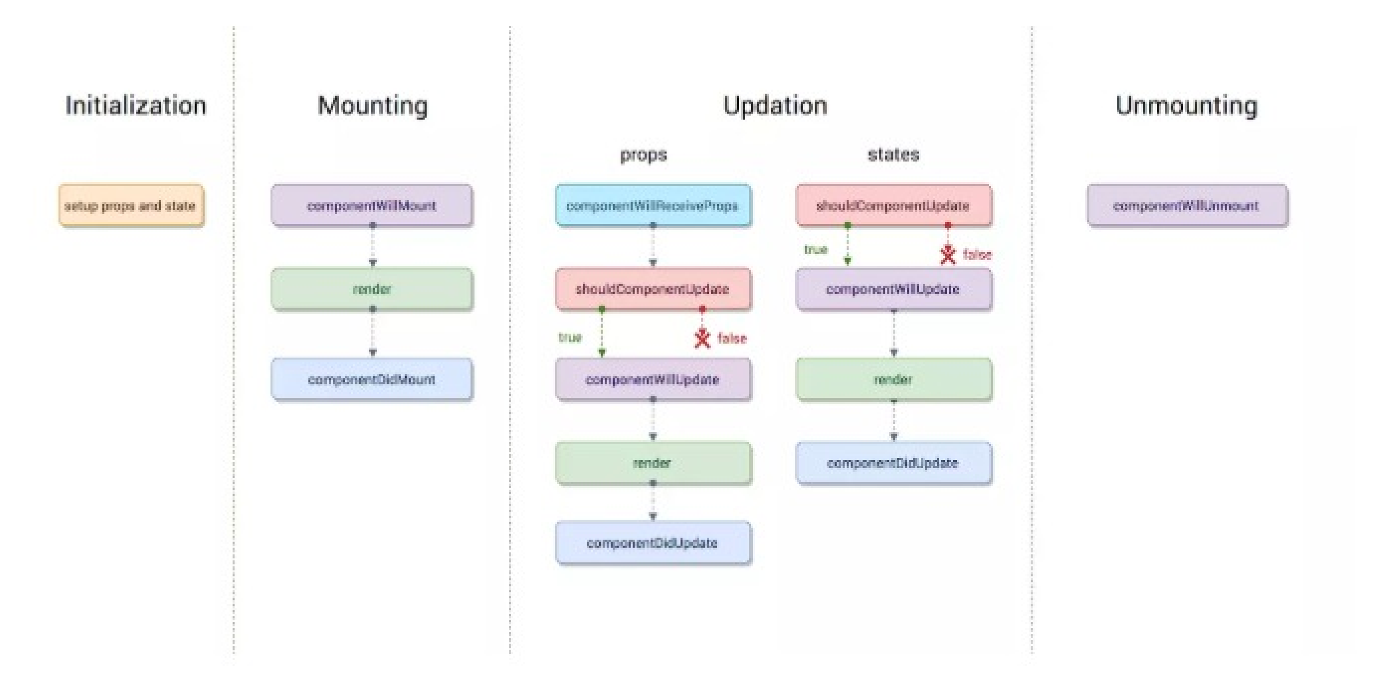
两个阶段的分界点，就是 `render` 函数。render 函数之前的所有生命周期函数（包括 render)都属于第一阶段，之后的都属于第二阶段。

开启异步渲染，虽然我们获得了更好的感知性能，但是考虑到第一阶段的的生命周期函数可能会被重复调用，不得不对历史代码做一些调整。

在 React v16.3 之前，render 之前的生命周期函数（也就是第一阶段生命周期函数）包括这些：

- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate
- componentWillMount
- render

下图是 React v16.3 之前的完整的生命周期函数图：



React 官方告诫开发者，虽然目前所有的代码都可以照常使用，但是未来版本中会废弃掉，为了将来，使用 React 的程序应该快点去掉这些在第一阶段生命函数中有副作用的功能。不得不说 React 真的很够意思，提前这么久告诉大家这个事情，让大家有足够的时间去修改自己的代码。

一个典型的错误用例，也是我被问到做多的问题之一：为什么不在 componentWillMount 里去做 AJAX？componentWillMount 可是比 componentDidMount 更早调用啊，更早调用意味着更早返回结果，那样性能不是更高吗？

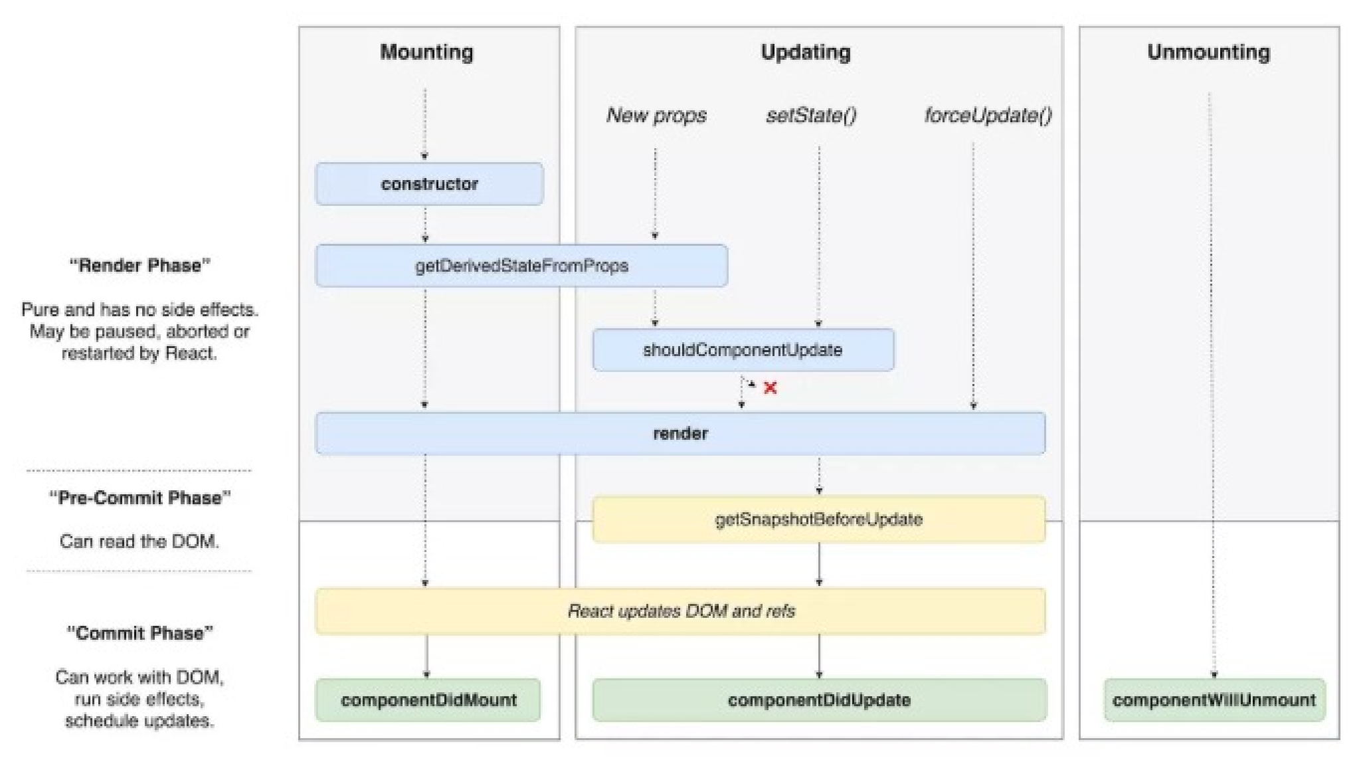
首先，一个组件的 componentWillMount 比 componentDidMount 也早调用不了几微秒，性能没啥提高；而且，等到异步渲染开启的时候，componentWillMount 就可能被中途打断，中断之后渲染又要重做一遍，想一想，在 componentWillMount 中做 AJAX 调用，代码里看到只有调用一次，但是实际上可能调用 N 多次，这明显不合适。相反，若把 AJAX 放在 componentDidMount，因为 componentDidMount 在第二阶段，所以绝对不会多次重复调用，这才是 AJAX 合适的位置（当然，React 未来有更好的办法，在下一小节 Suspense 中可以讲到）。

getDerivedStateFromProps

到了 React v16.3，React 干脆引入了一个新的生命周期函数 getDerivedStateFromProps，这个生命周期函数是一个 static 函数，在里面根本不能通过 `this` 访问到当前组件，输入只能通过参数，对组件渲染的影响只能通过返回值。没错，getDerivedStateFromProps 应该是一个纯函数，React 就是通过要求这种纯函数，强制开发者们必须适应异步渲染。

```
static getDerivedStateFromProps(nextProps, prevState) {
  //根据nextProps和prevState计算出预期的状态改变，返回结果会被送给setState
}
```

到了 React v16.3，React 生命周期函数全图如下：



注意，上图中并包含全部React生命周期函数，在React v16发布时，还增加了一个 `componentDidCatch`，当异常发生时，一个可以捕捉到异常的 `componentDidCatch` 就排上用场了。不过，很快React觉着这还不够，在v16.6.0又推出了一个新的捕捉异常的生命周期函数 `getDerivedStateFromError`。

如果异常发生在第一阶段（render阶段），React就会调用 `getDerivedStateFromError`，如果异常发生在第二阶段（commit阶段），React会调用 `componentDidCatch`。这个区别也体现出两个阶段的区分对待。

适应异步渲染的组件原则

明白了异步渲染的来龙去脉之后，开发者就应该明白，现在写代码必须要为未来的某一次 React 版本升级做好准备，当 React 开启异步渲染的时候，你的代码应该做到在 render 之前最多只能这些函数被调用：

- 构造函数
- getDerivedStateFromProps
- shouldComponentUpdate

幸存的这些第一阶段函数，除了构造函数，其余两个全都必须是纯函数，也就是不应该做任何有副作用的操作。

实际上，如果之前你的用法规范，除了 shouldComponentUpdate 不怎么使用第一阶段生命周期函数，你还会发现不怎么需要改动代码，比如 componentWillMount 中的代码移到构造函数中就可以了。但是如果用法错乱，比如滥用componentWillReceiveProps，那就不得不具体情况具体分析，从而决定这些代码移到什么位置。

开发者中一个普遍的误区，就是总想把任务往前提，提到靠前的生命周期函数去，就像我前面说过的在 componentWillMount 中做 AJAX。正确的做法是根据各函数的语义来放置代码，并不是越往前越好。

小结

这一小节我们介绍了 React v16 引入的“异步渲染”，读者通过本节阅读应该能够理解：

1. 异步渲染的意义；
2. 理解两阶段渲染；
3. 为了应对异步渲染，我们开发 React 组件需要对生命周期函数做哪些调整。

留言

评论将在后台进行审核，审核通过后对所有人可见

Artyhacker 前端 @ 北京某小国企

每次回顾或重构代码,componentWillReceiveProps总是可以通过更优化的方式给干掉. 事实上使用这个函数多是一开始使用了redux,后来图简单又混用了state,同时state和redux的store还有数据关联,这本来就是很混乱的代码.

▲ 0 收起评论 18天前

Artyhacker 前端 @ 北京某小国企

对了,最普遍的应该是props和state的数据产生了复杂的关联,也是可以通过更合理的设计,明确地区分开容器和视图(也就是前面的聪明和傻瓜组件)来避免这种数据关联.

18天前

评论审核通过后显示 评论

yadong 前端开发 @ 知乎

“但是如果用法错乱，比如滥用componentWillReceiveProps”，为什么不可以用componentWillReceiveProps，现在代码中大量用到了这个生命周期函数

▲ 0 收起评论 25天前

xiari 高级前端工程师 @ 拼多多

因为这种时候通常状态能提升或只维持在当前state,根本就用不上componentWillReceiveProps 或者 getDerivedStateFromProps。 这点官网的Blog里也有提及。

20天前

yadong 前端开发 @ 知乎

回复 xiari：使用redux的话state大部分均存储在store当中的，当store中数据发生变更（最简单的是fetch数据发生变更），在不卸载组件的前提下，我们基本都是用‘componentWillReceiveProps’来进行控制的

20天前

王巍达 前端 @ JD

回复 yadong：你的意思是你们把fetch的数据放到store中然后组件内依然有state接收了store中的数据吗？不然只有this.props.data根本不需要这个生命周期刷新数据呀

18天前

评论审核通过后显示 评论

ITSheng

作为生命周期函数而言，如果不能有副作用，又不需要返回值「shouldComponentUpdate和getDerivedStateFromProps除外」，那这个函数还有啥意义吗

▲ 0 收起评论 28天前

程墨 Hulu

有哪个生命周期函数既不能有副作用也没有返回值吗？

27天前

评论审核通过后显示 评论

brownz web前端开发

一个组件的 componentWillMount 比 componentDidMount 也早调用不了几微秒，性能没啥提高

有个疑问，如果这个组件非常庞大，那相应render的耗时应该会较长吧。那这时componentWillMount 会比 componentDidMount 提前多久调用，还会是微秒级别吗

▲ 0 收起评论 1月前

程墨 Hulu

唉，我不想再讨论这个问题了，如果谁想去用componentWillMount去调用AJAX，那就去做吧，后果自负。

1月前

Catherine酱

根本没有必要放到will里面取获取数据，按照规范放到did里面对于一般的场景再性能上看不出区别，而且省得后期react升级还需要去改动代码

1月前

评论审核通过后显示 评论