你以为你懂 React 吗?

这一讲我们来重点解析一下 React，说 React 是前端中最受瞩目的框架其实并不夸张。React 推出之后，很快风靡业界，并且 React 倡导的多种思想也对其他框架（比如 Vue）有着广泛影响。

据我观察，很多 React 开发者停留在「会使用」的阶段，而并没有在细节之处把握 React 精髓；我们可能对各种生命周期「了如指掌」，可能对「React 虚拟 DOM diff 算法」「对答如流」，可能对「单向数据流」「如数家珍」，可是你真的了解 React 吗?

同时，现在 React 面试越来越「范式化」，除了实际动手写码的题目之外，其他相关面试题毫无新意，很容易被应试，很难考察开发者的 React 功底。

对此，下面挑选出 React 中一些「不为人知」却又非常重要的点，为大家进行解析。在此过程中，通过剖析实现，读者可以更好、更深入地理解 React，掌握了这些内容，有可能在某些方向上比你的 React 面试官更有深度。

相关知识点如下：

神奇的 JSX

其实 React 的「专利发明」并不多，比如虚拟 DOM、组件化思想并不是 Facebook 原创。但 JSX 是 React 真正的创造，我认为这是 React 最「伟大」的发明。

JSX 是 React 的骨骼，它搭起了 React 应用的组件，是整个项目的组件框架基础。

「不就是 HTML in JS」吗？有什么神奇之处呢？请继续阅读。

JSX 就是丑陋的模版

直观上看，JSX 是将 HTML 直接嵌入在了 JS 代码里面，这是刚开始接触 React 时，很多人最不能接受的设定。因为前端开发者被「表现和逻辑层分离」这种思想「洗脑」太久了：表现和逻辑耦合在一起，在某种程度上是一种混乱和困扰。

但是从现在发展来看，JSX 完全符合「真香定律」：JSX 让前端实现真正意义上的组件化成为了可能。

可能有读者认为 JSX 很简单，但是你真的理解它了吗？试着回答这么几个问题：

如何在 JSX 中调试代码?

为什么 JSX 中不能直接使用 if...else

在回答这些问题之前，先来看看 JSX 是如何实现条件渲染的。

JSX 多种姿势实现条件渲染

很常见的一个场景：渲染一个列表。但是需要满足：列表为空数组时，显示空文案「Sorry，the list is empty」。同时列表数据可能通过网络获取，存在列表没有初始值为 null 的情况。

JSX 实现这种条件渲染最简洁的手段就是三目运算符：

```
const list = ({list}) => {
 const isNull = !list
 const isEmpty = !isNull && !list.length
```

```
 return (


   {
     isNull
     ? null
     : (
         isEmpty
         ?
Sorry, the list is empty


         :

             {
               list.map(item => )
             }


       )
   }


 )
}
```

但是我们多加了几个状态：加上出现错误时，正在加载时的逻辑，三目运算符嵌套地狱可能就要出现了：

```
const list = ({isLoading, list. error}) => {
 return (


   {
```

```
      condition1
      ?
      : (
          condition2
          ?
          : (
            condition3
            ?
            :
          )
        )
    }


 )
}
```

如何破解这种嵌套呢？我们常用的手段是抽离出 render function：

```
const getListContent = (isLoading, list, error) => {
   console.log(list)
   console.log(isLoading)
   console.log(error)
  // ...
  return ...
}

const list = ({isLoading, list, error}) => {
 return (


      {
        getListContent(isLoading, list, error)
      }
```

```
 )
}
```

甚至使用 IIFE：

```
const list = ({isLoading, list, error}) => {
 return (



     {
       (() => {
         console.log(list)
         console.log(isLoading)
         console.log(error)

         if (error) {
           return Something is wrong!
         }
         if (!error && isLoading) {
           return Loading...
         }
         if (!error && !isLoading && !list.length) {
           return
Sorry, the list is empty



         }
         if (!error && !isLoading && list.length > 0) {
           return

             {
               list.map(item => )
             }
```

```
          }
        })()
      }


  )
}
```

这样一来就可以使用 console.log 进行简单调试了，也可以使用 if...else 进行条件渲染。

再回到问题的本源：「为什么不能直接在 JSX 中使用 if...else，只能借用函数逻辑实现呢」？实际上，我们都知道 JSX 会被编译为 React.createElement。直白来说，React.createElement 的底层逻辑是无法运行 JavaScript 代码的，而它只能渲染一个结果。因此 JSX 中除了 JS 表达式，不能直接写 JavaScript 语法。准确来讲，JSX 只是函数调用和表达式的语法糖。

React 程序员天天都在使用 JSX，但并不是所有人都明白其背后原理的。

JSX 的强大和灵活

虽然 JSX 只是函数调用和表达式的语法糖，但是 JSX 仍然具有强大而灵活的能力。React 组件复用最流行的方式都是在 JSX 能力基础之上的，比如 HoC，比如 render prop 模式：

```
class WindowWidth extends React.Component {
 constructor() {
   super()
   this.state = {
     width: 0
   }
 }

 componentDidMount() {
   this.setState(
     {
```

```
        width: window.innerWidth
      },
      window.addEventListener('resize', ({target}) => {
        this.setState({
          width: target.innerWidth
        })
      })
    )
  }

  render() {
    return this.props.children(this.state.width)
  }
}



  {
    width => (width > 800 ?

show
: null)
  }
```

甚至，我们还可以让 JSX 具有 Vue template 的能力：

```
render() {
  const visible = true

  return (




       content
```

```
  )
}

render() {
 const list = [1, 2, 3, 4]

 return (




        {item}




  )
}
```

因为 JSX 总要进行一步编译，在这个编译过程中我们借助 AST（抽象语法树）对 v-if、v-for 进行处理即可。

你真的了解异步的 this.setState 吗?

绝大多数 React 开发者都知道 this.setState 是异步执行的，但是我会说「你这个结论是错误的！」，那么 this.setState 到底是异步执行还是同步执行？这是一个问题……

this.setState 全是异步执行吗?

this.setSate 这个 API，官方描述为：

setState() does not always immediately update the component. It may batch or defer the update until later. This makes reading this.state right after calling setState() a potential pitfall.

既然用词是 may，那么说明 this.setState 一定不全是异步执行，也不全是同步执行的。所谓的「延迟更新」并不是针对所有情况。

实际上， React 控制的事件处理过程，setState 不会同步更新 this.state。而在 React 控制之外的情况， setState 会同步更新 this.state。

什么是 React 控制内外呢？举个例子：

```
onClick() {
 this.setState({
   count: this.state.count + 1
 })
}

componentDidMount() {
 document.querySelectorAll('#btn-raw')
   .addEventListener('click', this.onClick)
}

render() {
 return (

      click out React

      click in React

 )
}
```

id 为 btn-raw 的 button 上绑定的事件，是在 componentDidMount 方法中通过 addEventListener 完成的，这是脱离于 React 事件之外的，因此它是同步更新的。反之，代码中第二个 button 所绑定的事件处理函数对应的 setState 是异步更新的。

这样的设计也不难理解，通过「延迟更新」，可以达到更好的性能。

this.setState promise 化

官方提供了这种处理异步更新的方法。其中之一就是 setState 接受第二个参数，作为状态更新后的回调。但这无疑又带来了我们熟悉的 callback hell 问题。

举一个场景，我们在开发一个 tabel，这个 table 类似 excel，当用户敲下回车键时，需要将光标移动到下一行，这是一个 setState 操作，然后马上进行聚焦，这又是一个 setState 操作。如果当前行就是最后一行，那用户敲下回车时，需要先创建一个新行，这是第一个 setState 操作，同时将光标移动到新的「最后一行」，这是第二个 setState 操作；在这个新行中进行聚焦，这是第三个 setState 操作。这些 setState 操作依赖于前一个 setState 的完成。

面对这种场景，如果我们不想出现回调地狱的场景。常见的处理方式是利用生命周期方法，在 componentDidUpdate 中进行相关操作。第一次 setState 进行完后，在其触发的 componentDidUpdate 中进行第二次 setState，依此类推。

但是这样存在的问题也很明显：逻辑过于分散。生命周期方法中有很多很难维护的「莫名其妙操作」，出现「面向生命周期编程」的情况。

回到刚才问题，解决回调地狱其实是我们前端工程师的拿手好戏了，最直接的方案就是将 setState Promise 化：

```
const setStatePromise = (me, state) => {
 new Promise(resolve => {
   me.setState(state, () => {
     resolve()
   })
```

```
  })
}
```

这只是 patch 做法，如果修改 React 源码的话，也不困难：

原生事件 VS React 合成事件

对 React 熟悉的读者可能知道：

> React 中的事件机制并不是原生的那一套，事件没有绑定在原生 DOM 上，大多数事件绑定在 document 上（除了少数不会冒泡到 document 的事件，如 video 等)
>
> 同时，触发的事件也是对原生事件的包装，并不是原生 event
>
> 出于性能因素考虑，合成事件（syntheticEvent）是被池化的。这意味着合成事件对象将会被重用，在调用事件回调之后所有属性将会被废弃。这样做可以大大节省内存，而不会频繁的创建和销毁事件对象。

这样的事件系统设计，无疑性能更加友好，但同时也带来了几个潜在现象。

现象 1：异步访问事件对象

我们不能以异步的方式访问合成事件对象：

```
function handleClick(e) {
 console.log(e)

 setTimeout(() => {
   console.log(e)
 }, 0)
}
```

上述代码第二个 console.log 总将会输出 undefined。

为此 React 也贴心的为我们准备了持久化合成事件的方法：

```
function handleClick(e) {
 console.log(e)

 e.persist()

 setTimeout(() => {
   console.log(e)
 }, 0)
}
```

现象 2：如何阻止冒泡

在 React 中，直接使用 e.stopPropagation 不能阻止原生事件冒泡，因为事件早已经冒泡到了 document 上，React 此时才能够处理事件句柄。

如代码：

```
componentDidMount() {
 document.addEventListener('click', () => {
   console.log('document click')
 })
}

handleClick = e => {
 console.log('div click')
 e.stopPropagation()
}

render() {
 return (



     click
```

```
  )
}
```

执行后会打印出 div click，之后是 document click。e.stopPropagation 是没有用的。

但是 React 的合成事件还给使用原生事件留了一个口子，通过合成事件上的 nativeEvent 属性，我们还是可以访问原生事件。原生事件上的 stopImmediatePropagation 方法：除了能做到像 stopPropagation 一样阻止事件向父级冒泡之外，也能阻止当前元素剩余的、同类型事件的执行（第一个 click 触发时，调用 e.stopImmediatePropagtion 阻止当前元素第二个 click 事件的触发）。因此这一段代码：

```
componentDidMount() {
 document.addEventListener('click', () => {
   console.log('document click')
 })
}

handleClick = e => {
 console.log('div click')
 e.nativeEvent.stopImmediatePropagation()
}

render() {
 return (



     click



 )
}
```

只会打印出 div click。

请不要再背诵 Diff 算法了

很多开发者在面试中能「背诵」出 React DOM diff 算法的方式，熟悉那著名的「三个假设」（不了解的读者可先自行学习），可是你真的懂 Diff 算法吗？如果我是面试官，我问几个简单的问题，你是否还能招架得住？

我们通过一个侧面来剖析 Diff 算法的细节。

Element diff 的那些事儿

我们都知道 React 把对比两个树的时间复杂度从 On 立方降低到大 On，三个假设也都老生常谈了。但是关于兄弟列表的 diff 细节，React 叫做 element diff，我们可以展开一下。

React 三个假设在对比 element 时，存在短板，于是需要开发者给每一个 element 通过提供 key ，这样 react 可以准确地发现新旧集合中的节点中相同节点，对于相同节点无需进行节点删除和创建，只需要将旧集合中节点的位置进行移动，更新为新集合中节点的位置。

组件 1234，变为 2143，此时 React 给出的 diff 结果为 2，4 不做任何操作；1，3 进行移动操作即可。

也就是元素在旧集合中的位置，相比新集合中的位置更靠后的话，那么它就不需要移动。当然这种 diff 听上去就并非完美无缺的。

我们来看这么一种情况：

实际只需对 4 执行移动操作，然而由于 4 在旧集合中的位置是最大的，导致其他节点全部移动，移动到 4 节点后面。

这无疑是很愚蠢的，性能较差。针对这种情况，官方建议：

「在开发过程中，尽量减少类似将最后一个节点移动到列表首部的操作。」

实际上很多类 React 类库（Inferno.js，Preact.js）都有了更优的 element diff 移动策略。

有 key 就一定「性能最优」吗?

刚才提到，在进行 element diff 时：由于 key 的存在，react 可以准确地判断出该节点在新集合中是否存在，这极大地提高了 element diff 效率。

但是加了 key 一定要比没加 key 的性能更高吗?

我们来看这个场景，集合 [1,2,3,4] 渲染成 4 组数字，注意仅仅是数字这么简单：

```
1

2

3

4
```

当它变为 [2，1，4，5]：删除了 3，增加了 5，按照之前的算法，我们把 1 放到 2 后面，删除 3，再新增 5。整个操作移动了一次 dom 节点，删除和新增一共 2 处节点。

由于 dom 节点的移动操作开销是比较昂贵的，其实对于这种简单的 node text 更改情况，我们不需要再进行类似的 element diff 过程，只需要更改 dom.textContent 即可。

```
const startTime = performance.now()

$('#1').textContent = 2
```

```
$('#2').textContent = 1
$('#3').textContent = 4
$('#4').textContent = 5

console.log('time consumed:' performance.now() -
startTime)
```

这么看，也许没有 key 的情况下要比有 key 的性能更好。

总结

这一讲，我们聚焦 React 当中那些「不为人知」的设计细节，这些设计细节却从不同角度体现了 React 的理念和思想。仔细想来，也许「我之前理解的 React 还是很肤浅」！实际上，任何一个类库或者框架，我们都不能停留在初级使用上，而更应该从使用的经验出发，深入细节，这样才能更好地理解框架，也能更快地自我提升。

点击查看下一节

揭秘 React 真谛：组件设计