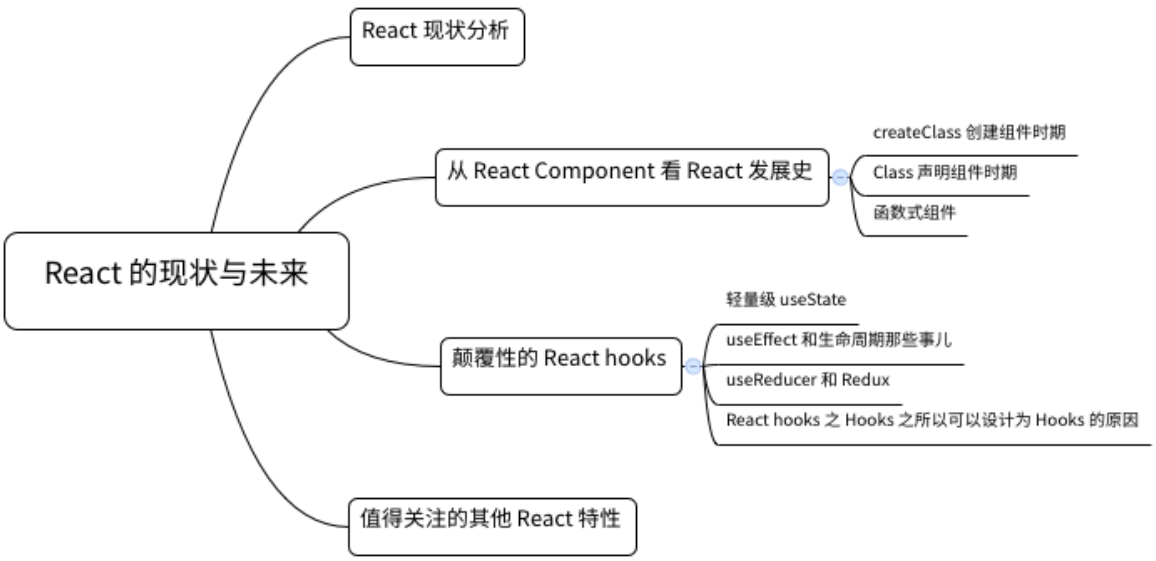
React 的现状与未来

React 自推出以来，一直进行着自身完善和演进。作为 React 开发者或前端开发者，有幸见证着一个伟大「框架」的成长，是非常幸运的。那么在这个过程中，我们应该学些什么？React 现在处于什么发展阶段？React 未来又将有哪些规划？

高级前端工程师不能只停留在使用框架上，我们自然要思考上述这些问题。这一讲我们就来聊聊 React 的现状与未来，如果不熟悉 React，也并不妨碍大家阅读。

相关知识点如下：



React 现状分析

React 经过几年的打磨，目前维持了一个稳定的迭代周期，并不断给开发者带来惊喜。其中难能可贵的是在 breaking changes 不多的前提下，仍在持续输出具

有变革精神的特性，保持着旺盛的生命力。不管是什么平台的调查，都显示 React 受众仍然最多，可以预见的是，React 未来仍将会统领前端发展。

关于 React 现状，我总结出以下几个特点：

开发模式已经定型，有利于开发者持续学习

仍然有强大的开发团队维护，不断带来改变，这些改变一方面使 React 更好，另一方面甚至推动了 JavaScript 语言的发展

社区生态强大，有一系列解决方案，数据状态管理、组件库、服务端渲染生态群百花齐放

在这些特点的背后，也有一些让开发者担忧的地方：

概念越来越多。一定程度上，新老概念并存，学习曲线激增

存在较多 unsafe_ 标记的 APIs，始终担忧彻底废弃相关 APIs 那一天的到来

新特性带来了较多「魔法」，也带来了一些困惑

当然，这些让开发者担忧的地方并不足以和 React 的强大相提并论，这些「问题」甚至在任何一个框架中都会存在。因此，我建议不管是工作需要，还是自身学习需要，前端开发者都可以使用并研究 React。

从 React Component 看 React 发展史

回顾 React 发展历史，很多 APIs 和特性的演进都很有意思，比如 refs、context，其中任何一点都值得单拎出来深入分析。但是我挑选了一个开发者一定会使用的 React Component 话题：从组件的创建和声明方式，我们来看一个框架的变革，并由此引出 React 目前最受关注的的 hooks 新特性。

React Component 的发展主要经历了三个阶段：

createClass 创建组件时期

ES class 声明组件时期

无状态（函数式）组件 + React hooks 时期

这一路，也是 React 从一个纯粹的视图层类库走向成熟完善的解决方案的过程。我们逐一来看。

createClass 创建组件时期

相信很多新的开发者都没有使用过 createClass API 创建组件，createClass 是一个函数，接受参数并返回组件实例，用起来并不复杂：

```
import React from 'react'

const component1 = React.createClass({
 propTypes: {
   foo: React.PropTypes.string
 },

 getDefaultProps() {
   return {
       foo: 'bar'
   }
 },


 getInitialState() {
     return {
     state1: 'lucas'
   }
 },

 handleClick() {

 },

 render() {
```

```
    return (




    )
  }
})
```

看起来很好理解，但是编写还是有些违背直觉。从 React 15.5 版本开始，官方就不再开始推荐，到了 React 16 版本，已经彻底废弃。

Class 声明组件时期

createClass 退出历史舞台的原因是被强势的 class 声明组件方式所取代。当时 ES6 正在如火如荼地发展，新增了 class 这一语法糖，React 团队很快赶时髦，支持了使用 class 声明组件的方式：

```
class Component1 extends React.Component {
 state = { name: 'Lucas' }

 handleClick = e => {
   console.log(e)
   this.setState({
     name: 'Messi',
   })
 }

 render() {
   return (



        {{this.state.name}}



   )
```

```
  }
}
```

代码非常直观清爽，但是 class 声明方式和早期的 createClass 相比，有非常重要的两点差别：

- React.createClass 支持在事件处理函数中自动绑定 this，而 class 声明的组件需要开发者手动绑定
- React.Component 不能使用 React mixins 来实现复用

这两个显著不同点决定了 React 生态社区发展的方向。

其中第一点不同，决定了 React 放弃了「多管闲事」地绑定 this，虽然这个行为在很多人看来毫无必要，很多类 React 框架都会帮助开发者对事件处理函数绑定 this，Vue 也是如此。

但是我们一般进行绑定 this 的方案多种多样，上述代码采用了 ES Next 的属性初始化方法，对 handleClick 进行了绑定。

第二点不同，决定了 React 实现复用的方式发展方向。首先肯定的是官方认为 mixin 是弊大于利的，已经被彻底放弃。那么社区跟进复用方案主要有两种：

- 高阶组件
- render prop 模式

其中高阶组件很好地体现了 React 函数式思想，是 React 精华之体现。而 render prop 目前也非常流行，并最终推动了 React 自身的发展：新的 context 特性，其 API 也变革为 render prop 模式，这是社区反哺 React 的例证：

```
{value => (

)}
```

但是使用 class 声明组件不是完美无懈可击的。React 官方团队认为，这种方式「已经背离了 React 的初衷」。我总结下来，class 声明组件的问题有以下两个。

> 带来了「面向生命周期编程」的困扰，随着逻辑变复杂，组件的生命周期函数随之变得很难维护和理解。我们想理清楚 componentDidMount、componentDidUpdate、componentWillUnmount、componentWillRecieveProps 这些钩子的逻辑并不困难。但是这些生命周期函数中的代码和 render 中的 state 以及 props 有什么关系？这些问题将会随着应用的复杂被无限放大，
>
> React 是函数式的，而 class 声明组件这种面向对象的行为显得不伦不类。

基于这两点，React 很快推出了函数式组件，或无状态组件（下面统称函数式组件，因为无状态组件在 hooks 特性下也会有状态）。

函数式组件

函数式组件非常简单，我们用函数定义一个组件，该函数接受 props 作为参数，只负责渲染：

```
const component = props =>

{ props.name }
```

这样的实现看上去棒极了，组件只负责接收数据并渲染，难得如此清爽和直接。然而它是完全无法取代 class 组件的，因为它不存在生命周期，完全的无状态让我们无法处理必要的逻辑。

因此，class 声明组件结合函数式组件的方案，类似容器组件结合木偶组件，成为现在的主流方式。

从 React component 的发展，我们能够管中窥豹：从中可以发现 React 绑定 this 的设计、React 实现复用的方案等一系列知识点，它无疑是 React 类库的主干。不过我们继续设想，能不能赋予函数式组件以类似生命周期的能力，完美解

决 class 组件的问题？这就是 React 近期带来的 React hooks 特性。请继续阅读。

颠覆性的 React hooks

说起 React hooks，想必大家已经了解了它出现的背景，那么它是如何解决问题的呢?

简单来说，它使得开发者可以按业务逻辑拆分代码，而不是生命周期。这样如果想实现复用，直接在任何组件中引入相关 hooks 即可。Hooks 把代码按照业务逻辑的相关性进行拆分，把同一业务的代码集中在一起，不同业务的代码独立开来，维护起来就清楚很多。

这里我们不会科普 hooks 的使用方案，因为官网上介绍的一定最好、最详尽，我们从原理和设计的角度来进行分析。

轻量级 useState

事实上，setState API 并没有什么问题，它也足够轻量，真正笨重的是 class 组件结合 setState。而使用 useState hook，使得函数式组件也具备了操作 state 的能力，且不需要引入生命周期函数。

useState 是一个函数，入参是 initialState；它返回一个数组，第一值是 state，第二个值是改变 state 的函数。

这里我来插播一个细节，为什么 useState 返回一个数组呢（其实返回的是 tuple，但是 JavaScript 还没有这个概念）？

```
let [name, setName]= useState('lucas')
```

如果返回的是一个对象是否更合适呢?

```
let { state: name, setState: setName } = useState('lucas')
```

这样表意更加清晰，而简单。也支持我们自动设置别名。事实上 React RFC 确实有相关讨论：">RFC: React Hooks 。

useState 其实很好实现：

```
const React = (function() {
 let stateValue

 return Object.assign(React, {
   useState(initialStateValue) {
     stateValue = stateValue || initialStateValue

     function setState(value) {
       stateValue = value
     }

     return [stateValue, setState]
   }
 })
})()
```

我们使用 stateValue 闭包变量储存 state，并提供修改 stateValue 的方法 setState，一并作为数组返回。

useEffect 和生命周期那些事儿

函数式组件通过 useState 具备了操控 state 的能力，修改 state 需要在适当的场景进行：class 声明的组件在组件生命周期中进行 state 更迭，那么在函数式组件中呢？我们需要用 useEffect 模拟生命周期，目前 useEffect 相当于 class Component 中的 componentDidMount、componentDidUpdate、componentWillUnmount 三个生命周期的综合。

也就是说：useEffect 声明的回调函数会在组件挂载、更新、卸载的时候执行。为了避免每次渲染都执行所有的 useEffect 回调，useEffect 提供了第二个参数，该参数是数组类型。只有在渲染时数组中的值发生了变化，才会执行该 useEffect 回调。如果传的是个空数组，也就是说并不依赖任何其它值，因此这样只会在组件第一次 Mount 后和 Unmount 前调用。

我们尝试实现 useEffect：

```
const React = (function() {
 let deps

 return Object.assign(React, {
   useEffect(callback, depsArray) {
     const shouldUpdate = !depsArray

     const depsChange = deps ? !deps.every((depItem,
index) => depItem === depsArray[index]) : true

     if (shouldUpdate || depsChange) {
         callback()

         deps = depsArray || []
     }
   }
 })
})()
```

我们看闭包变量 deps 存储前一刻 useEffect 的依赖数组值。在每次调用 useEffect 时，我们都会遍历 deps 数组和当前 depsArray 数组值，如果其中的任何一项有变化，depsChange 将为 true，进而执行 useEffect 的回调。

有读者可能会想到，那么生命周期 shouldComponentUpdate 如何模拟呢？事实上，我们不需要用 useEffect 来实现 shouldComponentUpdate。React 新特性中专门提供了 React.memo 来帮助开发者进行性能优化。另外，useEffect 是无法模拟 getSnapshotBeforeUpdate 和 componentDidCatch 这两个生命周期函数的。

上述两种实现都是简易版的，旨在剖析这两个 hooks 的工作原理，更多细节都没有实现。最重要的一点是：如果组件内多次调用 useState 或 useEffect，我们的实现为了区分每次 useState 调用之前不同的 state 值及 setter，需要额外使用一个数组来存储每次调用的配对值，比如：

```
const React = (function() {
 let hooks = []
```

```
 let currentHook = 0

 return Object.assign(React, {
   useState(initialStateValue) {
     hooks[currentHook] = hooks[currentHook] ||
initialStateValue

     function setState(value) {
       hooks[currentHook] = value
     }

     return [hooks[currentHook++], setState]
   },

   useEffect(callback, depsArray) {
     const shouldUpdate = !depsArray

     const depsChange = hooks[currentHook] ?
!hooks[currentHook].every((depItem, index) => depItem ===
depsArray[index]) : true

     if (shouldUpdate || depsChange) {
         callback()

         hooks[currentHook++] = depsArray || []
     }
   }

 })
})()
```

这也是为什么 hooks 只可以在顶层使用，不能写在循环体、条件渲染，或者嵌套 function 里。因为 React 内部实现需要按调用顺序来记录每个 useState 的调用，以做区分。

useReducer 和 Redux

我们知道，如果 State 的变化有比较复杂的状态流转，可以使用新的 hooks：useReducer 让应用更加 Redux 化，使得逻辑更加清晰。那么首先思考一个问题：到底是该用 useState 还是 useReducer 呢?

为此，我总结如下。

使用 useState 的情况：

- state 为基本类型（也要看情况）
- state 转换逻辑简单的场景
- state 转换只会在当前组件中出现，其他组件不需要感知这个 state
- 多个 useState hooks 之间的 state 并没有关联关系

使用 useReducer 的情况：

- state 为引用类型（也要看情况）
- state 转换逻辑比较复杂的场景
- 不同 state 之间存在较强的关联关系，应该作为一个 object，用一个 state 来表示的场景
- 如果需要更好的可维护性和可测试性

其实翻看 React 源码 useState 实现，useState 本质是 useReducer 的一个语法糖。

第二个问题：useReducer 是否代表着 React 内置了 Redux，我们就可以脱离 Redux 了呢？事实上，确实可以用简单的 React 代码，借助 context API 实现全局 Redux 或者局部 Redux：

store.js 文件：

```
import React from 'react'
const store = React.createContext(null)

export const initialState = {
   // ...
}

export const reducer = (state, action) => {
 switch (action.type) {
   // ...
 }
}
export default store
```

Provider 根组件挂载：

```
import React, { useReducer } from 'react'
import store, { reducer, initialState } from './store'

function App() {
 const [state, dispatch] = useReducer(reducer,
initialState)
 return (




 )
}
```

业务组件就可以直接使用：

```
import React, { useContext } from 'react'
import store from './store'
```

```
cosnt Child = props => {
 const { state, dispatch } = useContext(store)
 // ...
}
```

但是这样的行为尚不足以完全取代 Redux，我们这里不做展开。

React hooks 之 Hooks 之所以可以设计为 Hooks 的原因

我们现在了解了：

useState 让函数式组件能够使用 state

useEffect 让函数式组件可以模拟生命周期方法，并进行副作用操作

useReducer 让我们能够更清晰地处理状态数据

useContext 可以获取 context 值

那么为什么其他的一些 APIs，比如 React.memo 并没有成为一个 hook 呢？事实上 React 认为能够成为 hooks 的条件有两个特定：

composition：这个新特性需要具有组合能力，也就是说需要有复用价值，因为 hooks 的一大目标就是完成组件的复用。针对于此，开发者可以自定义 hooks，而不必官方束缚指定的 hooks，这样反倒可能会发成冲突；

debugging：hooks 一大特性就是能够调试，如果应用出现差错，我们能够从错误的 props 和 state 当中找到错误的组件或逻辑，能够具有这样调试功能的特性，才应该成为一个 hooks。

为此 Dan abramov 专门写了篇文章来讲述：Why Isn』t X a Hook?，这里我们不再赘述。

值得关注的其他 React 特性

我认为在众多新特性中，还有一个可能会对社区带来较大影响的是 React v16.6 发布的 React.Suspense 和 React.lazy。具体用法我们不再讲解，读者可自行补充基础知识。React.Suspens 给了 React 组件异步（中断）渲染的能力，打破了 React 组件之前「一鼓作气」渲染的格局。而 React.lazy 带来了延迟加载的能力，可以很好地取代社区上的一些轮子实现。

我们来看一个场景，React.Suspense 结合 React.lazy，实现代码分割和按需加载。

目前按需加载一般都采用 react-lodable，这个库稳定优雅且支持服务端渲染：

```
const Loading = ({ delay }) => {
 if (delay) {
   return
 }
 return null
}

export const AsyncComponent = Loadable({
 loader: () => import(/* webpackChunkName: "Component1" */
'./component1'),
 loading: Loading,
 delay: 500
})
```

这段代码定义了一个 Loading 组件，在请求返回之前进行渲染；delay 参数表示时间超过 500 毫秒才显示 Loading，防止闪烁 Loading 的出现。

如果换成 React.Suspense + React.lazy：

```
const Component = React.lazy(() => import(/*
webpackChunkName: "Component1" */ './component1'))

export const AsyncComponent = props => (
```

```
  }>


);
```

React.lazy 封装动态 import 的 React 组件，它要求 import() 必须返回一个会 Promise 对象，并且这个 Promise 对象会 resolve 为一个 ES 模块，模块中 export default 必须是一个合法的 React 组件。

React.Suspense 组件设置 fallback prop，当发现我们的 Component 是一个 Promise 类型时，且这个 Promise 没有被 resolved，那么就启用 fallback prop 所提供的组件，以便在我们等待网络返回结果时进行渲染。

我们可以结合 Error Boundary 特性，对于网络或者其他错误时，进行错误处理：

这样一来我们就实现了简单的 react-loadable 库。当然在 React.suspense 正式发布之前，我们当然可以自己手动实现一个 React.Suspense 组件，这里提供一个简单的版本，未考虑边界情况：

```
export class Suspense extends React.Component {
   state = {
     isLoading: false
   }

   componentDidCatch(error) {
     if (typeof error.then === 'function') {
       this.setState({ isLoading: true })
       error.then(() => {
         this.setState({ isLoading: false })
       })
     }
   }
```

```
    render() {
      const { children, fallback } = this.props
      const { isLoading } = this.state

      return isLoading ? fallback : children
    }
}
```

核心思路就是在首次渲染 Promise 出错时，使用 componentDidCatch 进行捕获，转而通过状态切换渲染 fallback 组件；在 Promise resolve 之后，通过状态切换渲染目标组件。

总结

这一讲我们梳理了 React 发展史上重要的里程碑，并展望了 React 未来发展。任何一门框架其实都免不了从问世到巅峰、再到逐步退出的过程。一个框架的兴衰，印证着技术潮流的更迭，作为开发者，合理分析框架发展背后的技术趋势，就非常重要了。

点击查看下一节

同构应用中你所忽略的细节