

你不知道的 React






Presentation slides for developers

Press Space for next page →



React由什么组成?

React 用于构建 Web 和原生交互界面的库

-  **心智模型** - 描述UI, 副作用, 渲染时机, 交互行为, 状态管理, 代数效应, $u=f(s)$?
-  **设计原则** - 离谱的React, React 一点都不 reactivity!
-  **UI Runtime** - 包结构, 不同概念的数据结构, Fiber, hooks, call tree, 异步还是同步
-  **Concurrent Mode** - 渲染中断, 时间切片, lane, 状态恢复
-  **React Compiler** - v8工程师都顶不住

参考资料 [React 官网](#) - [overreact](#) - [jsr.dev](#) - [图解react](#)

心智模型

React框架自身的概念和实践理念，似乎已经超过了前端应有的控制范围，从16版本后，其内核复杂度飞升，心智负担重，备受诟病。

名称解释

组件 /

component

React构建的基本单元

副作用 /

effect

外部数据行为和React结合的过程

代数效应

分离业务和数据执行关系，代码编写只关注业务逻辑



尤雨溪

@yuxiyou

一个框架挖下许多艰深复杂的坑，然后不填这些坑，而是靠文档去解释如何绕开这些坑。用户看了不但不质疑为什么这些坑有存在的必要，反而击节赞叹文档写得太好了。

虽然 Dan 的文档写得确实不错... 但这心态真不是被框架 PUA 了吗？



Sixian



@nowworkforsixian · 1月3日

React 的新版官方文档太棒了，把许多坑都直接列了出来，并用心地附上了例子解释这样为什么错，会导致什么样的问题，非常直观。我一直觉得掌握一个东西最有效的方法就是先 break it，知道它为什么坏，就能了解当前的设计到底是在解决什么问题。

...

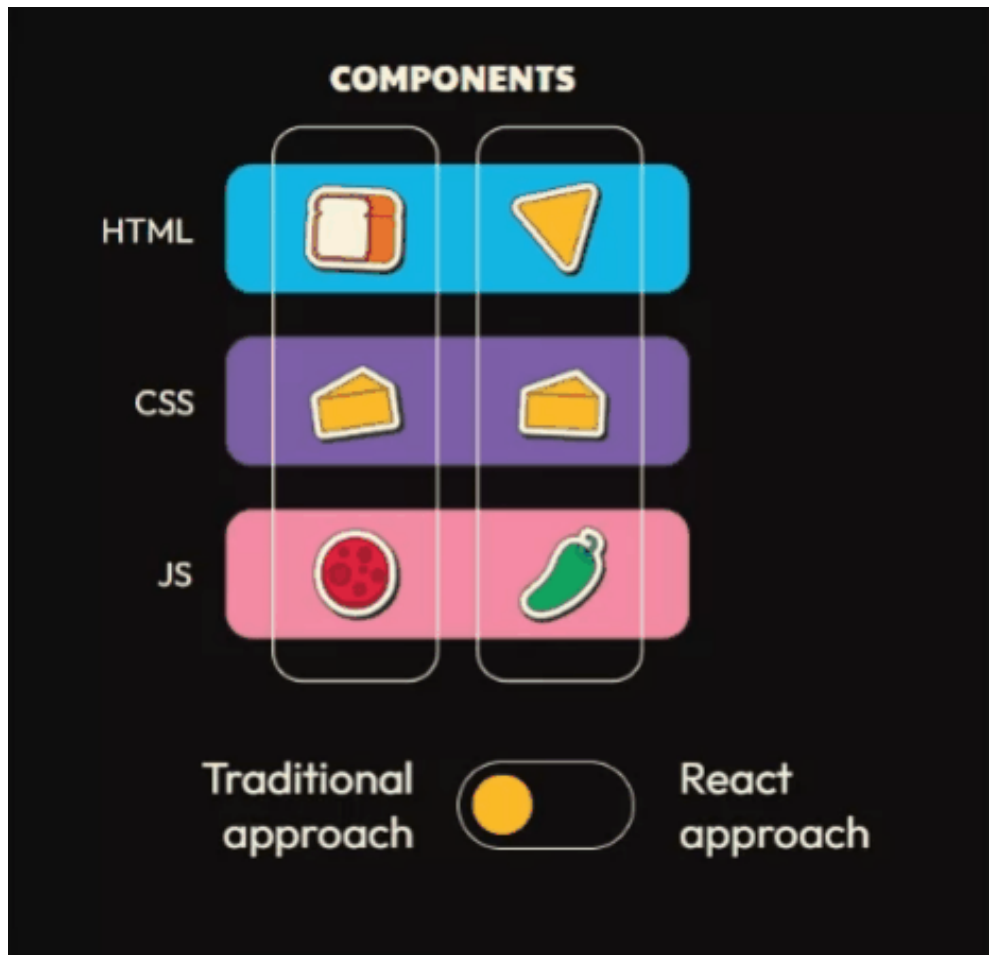
[显示更多](#)

基本示例

使用React构建web应用

```
1 // 步骤1 JSX描述组件和属性
2 const Button = (props) => {
3   const {text} = props
4   return <div>{text}</div>
5 }
6
7 ReactDOM.render(
8   // { type: 'Button', props: { text: 'click' } }
9   <Button text="click" />,
10  document.getElementById('container')
11 );
```

以上代码中，使用 jsx 简单的构造了一个组件并将其渲染到页面上，jsx 就是语法糖，将 css js html 混合组合，并通过props实现父子组件之前的通信。



状态管理和副作用

React有一套自己的状态管理机制，Component的本质是函数，通过 `useState` api能够将组件之间的状态实现统一管理。`useEffect`能够将外部数据和React框架相结合，从而改变React状态。

demo

```
function App() {  
  const [count, setCount] = useState(0);  
  
  return (  
    <div className="App">  
      <h2>You clicked {count} times!</h2>  
      <button onClick={() => {  
        setCount(count - 1)  
      }}>Decrement</button>  
      <button onClick={() => {  
        setCount(count + 1)  
      }}>Increment</button>  
    </div>  
  );  
}
```

demo

```
const ResourceList = ({ sendResource }) => {  
  const [resourceList, setResourceList] = useState([]);  
  const fetchData = async () => {  
    const resp = await axios.get(  
      `https://jsonplaceholder.typicode.com/${sendResource}`  
    );  
    setResourceList(resp.data);  
  };  
  
  useEffect(() => {  
    fetchData();  
  }, [sendResource]);  
  
  return <ul>  
    {resourceList.map((item) => (  
      <li>{item.title}</li>  
    ))}  
  </ul>
```

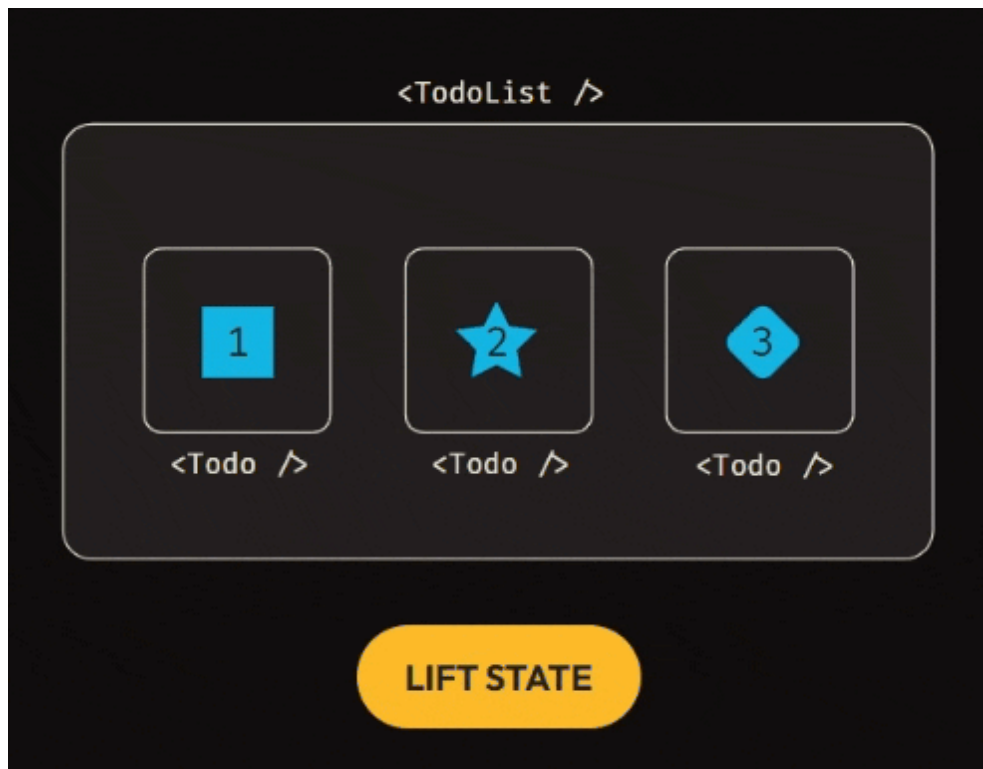
组件通信

React使用的是单向数据流，组件只会监听数据的变化，当数据发生变化时它们会使用接收到的新值，而不是去修改已有的值。当组件的更新机制触发后，它们只是使用新值进行重新渲染而已。

因此，如果父子或者兄弟组件要做到双向通信，需要子组件的状态提升到父组件，将相关 state 从这两个组件上移除，并把 state 放到它们的公共父级，再通过 props 将 state 传递给这两个组件。这被称为“状态提升”。

Props：由父组件传递给子组件的只读属性，不能在子组件中修改

State：组件内部维护的可变状态，可以通过 `setState()` 方法进行修改。



渲染

当React渲染一个组件时，React会创建组件的快照，其中包含了React在特定时间点需要更新视图所需的一切。Props、state、事件处理程序以及UI描述（基于这些props和state）都被捕获在这个快照中。随后React接收UI的描述并使用它来更新视图。React 将重新渲染拥有该状态的组件以及其所有子组件 - 无论这些子组件是否接受任何 props

设计原则

UI Before API

以贴合用户使用习惯作为框架设计的基本目的

拥抱复杂度

将复杂度吸收到框架内部，为框架使用者提供简单直接的概念

局部优先

干净的组件

统一路线

框架设计符合一个理论一个路线

UI Runtime

基础包结构

react

react 基础包, 只提供定义 react 组件(ReactElement)的必要函数

react-dom

react 渲染器之一, 是 react 与 web 平台连接的桥梁(可以在浏览器和 nodejs 环境中使用), 将 react-reconciler中的运行结果输出到 web 界面上

react-

reconciler

react 得以运行的核心包(综合协调react-dom,react,scheduler各包之间的调用与配合). 管理 react 应用状态的输入和结果的输出. 将输入信号最终转换成输出信号传递给渲染器.

scheduler

调度机制的核心实现, 控制由react-reconciler送入的回调函数的执行时机, 在concurrent模式下可以实现任务分片. 在编写react应用的代码时, 同样几乎不会直接用到此包提供的 api.

image

核心对象和函数

Fiber 对象 一个Fiber对象代表一个即将渲染或者已经渲染的组件 React fiber执行demo

Hook 对象 Hook用于function组件中,能够保持function组件的状态 `useState,useEffect`

Task 对象 用于内部scheduler包进行任务调度

current 对象-当前渲染在页面上实际ui的 fiber树结构

workInProgress 对象-fiber树内存中工作备份, 下一帧即将替换当前 current 的fiber树结构

`scheduleUpdateOnFiber()`: 告知React开始渲染的位置, 每次rerender都会触发 image

`ensureRootIsScheduled()`: 确保根节点有任务被调度, 同时处理重复的`useState`调用 (合并和消除)

`scheduleCallback()`: 创建任务并确定优先级, 放入任务堆中等待 schedule调度

`workLoop()`:任务循环, 不断的执行

`performConcurrentWorkOnRoot()`: 实际调度任务执行的区域 (render阶段, 可中断)

`commitRoot()`:提交fiber变动, 替换当前current对象 (浏览器开始执行渲染阶段, 不可中断) rerender-demo
rerender

concurrent mode 可中断渲染

并发渲染，中断可恢复，优先级调度，自动批处理，以上所说的所有结构和对象，都是为这个目标而服务

为什么要把一个ui框架的复杂度提升到这个地步：一切都为 ui before api 快速响应

我们日常使用 App，浏览网页时，有两类场景会制约快速响应：

当遇到大计算量的操作或者设备性能不足使页面掉帧，导致卡顿。

发送网络请求后，由于需要等待数据返回才能进一步操作导致不能快速响应。

这两类场景可以概括为：

CPU 的瓶颈

IO 的瓶颈

React对应的解决方案就是

CPU:时间切片，中断恢复机制

IO:模糊异步同步界限，suspense----可中断demo — 未开启demo

中断

老的React在任务更新上更类似于栈结构，通过递归进行，所以更新一旦开始，中途就无法中断。当层级很深时，递归更新时间超过了 16ms，用户交互就会卡顿。

而新的React架构，整体是由树和链表组成，UI更新时涉及到 状态管理和视图渲染，其中的数据变化由fiber结构中的 hooks 接管，而hooks就是链表结构，更新的复杂度为 $O(1)$ ，至于视图渲染，React并不会及时渲染视图变化而是在 render 阶段标记每一个节点的变更内容，统一在commit阶段处理。

而是否中断有一套自己的判断机制，每个任务都有对应的执行时长，当有高优先级任务插入，进行中断，如果低优先级任务超时，低优先级任务变为同步任务直接执行，其次如果任务执行时间过长，也会被中断将cpu控制权交还

中断示意图

中断步骤详解

状态恢复，中断容易恢复难，hooks 是一个链表，不同的更新可能存在前后依赖关系，如果高优先级中断低优先级，很有可能出现数据失真。

React Compiler

没错，在19版本的 React 马上就要有自己的 compiler了！

经过之前的介绍，我们知道 React 不会每次都只处理需要更新的节点，而是通过标记和遍历整个树结构来找到需要处理的节点，而如何实现最大可能的节点复用，主要还是依靠开发者自己的手动判断，特别是涉及到一些状态和依赖关系极其复杂的组件时，会通过调用 `useMemo` 和 `useCallback` 进行手动标记。

为了更好的炫技，不！是服务开发者，React 团队实现了自己的 Compiler！

编译器利用其对 JavaScript 和 React 规则的了解，自动对组件和钩子中的值或值组进行记忆化。如果它检测到规则的破坏，它将自动跳过那些组件或钩子，并继续安全地编译其他代码。

demo