前端框架_React知识点精讲

Fiber 机制

React 是一个用于**构建用户界面**的 JavaScript 库。

> 它的**核心**是**跟踪组件状态的变化**并将更新的状态投射到屏幕上。
>
> 在 React 中，我们把这个过程称为{调和| Reconciliation}。我们调用 setState 方法，框架会检查{状态|state}或{属性|props}是否发生了变化，并在用户界面上**重新**显示一个组件。

从渲染方法返回的{不可变|immutable}的**React元素树**通常被称为{虚拟DOM| Virtual DOM} 。这个术语有助于在早期向人们解释React，但它也造成了混乱，在React文档中已不再使用。在这篇文章中，我将坚持称它为{React元素树| Tree of React elements}。

> 除了**React元素树**，该框架有一棵**内部实例树**（组件、DOM节点等），**用来保持状态**。
>
> 从**16版**开始，React推出了一个新的**内部实例树的实现**，以及管理它的算法，代号为 Fiber。

在**调和**过程中还有其他操作，如**调用生命周期方法**或更新ref。**所有这些操作在 Fiber 架构中都被统称为** {工作| Work}。

工作的类型通常取决于React元素的类型。例如，

- 对于一个*类组件*， React 需要创建一个实例，
- 而对于一个函数组件，它不需要这样做。

如你所知，我们在 React 中有许多种类的元素。

- 类组件(React.Component)
- 函数组件
- 宿主组件(DOM节点)
- Portals (将子节点渲染成存在于父组件的DOM层次之外的DOM节点)

React 元素的类型是由 createElement 函数的第一个参数定义的。这个函数一般在render方法中使用，**用于创建一个元素**。而在React开发中，我们一般都使用JSX语法来定义元素(而JSX是createElement的语法糖)，**JSX 标签的第一部分决定了React元素的类型**。例如，

- 以大写字母开头表示JSX标签是指一个**React组件**
  - <ClickCounter>
- 以小写字母开头表示**宿主组件**或者**自定义组件**
  - <button>
  - <p-test>

从 {React 元素| React Element} 到 {Fiber 节点| Fiber Node}

> React 中的**每个组件都是一个UI表示**

这里是我们的 ClickCounter 组件的模板。

html 复制代码

```
<button key="1" onClick={this.onClick}>
  更新数字
</button>
<span key="2">
  {this.state.count}
</span>
```

{React 元素| React Element}

> 一旦模板通过{JSX编译器| JSX Compiler}，你最终会得到**一堆React元素**。-->这就是真正从 React 组件的渲染方法中返回的东西，**而不是HTML**。

如果不需要使用 JSX语法，可以使用React.createElement。 render方法中对React.createElement的调用将**创建**这样的两个数据结构

javascript 复制代码

```
[
  {
    $$typeof: Symbol(react.element),
    type: 'button',
    key: "1",
    props: {
      children: '更新数字',
      onClick: () => { ... }
    }
  },
  {
    $$typeof: Symbol(react.element),
    type: 'span',
    key: "2",
    props: {
      children: 0
    }
  }
]
```

你可以看到 React 给这些对象添加了$$typeof属性，可以**标识它们是React元素**。然后还有**描述元素的属性** type、 key 和 props,这些值取自你传递给React.createElement函数的内容。

{Fiber 节点| Fiber Node}

> 在**调和过程**中，从render方法返回的**每个React元素的数据**都被合并到Fiber树中。
>
> 与React元素不同， fiber **不会在每次渲染时重新创建**。这些是{可变的数据结构| mutable data structures}，持有组件状态和 DOM信息

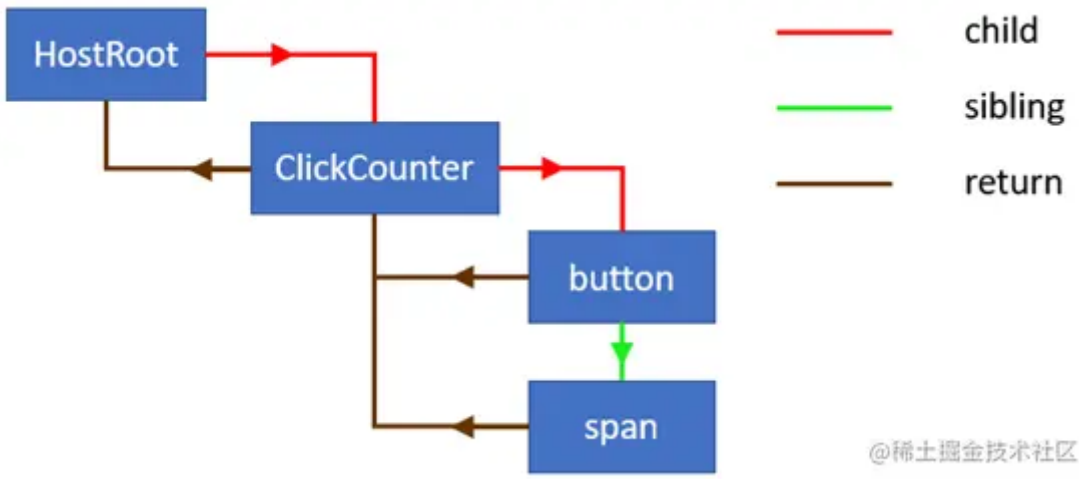
每个React元素都被转换为相应类型的Fiber节点，描述需要完成的工作。

> 可以把fiber看作是一个**数据结构**，它代表了一些要做的工作，或者说，**一个工作单位**。
>
> Fiber的架构还提供了一种方便的方式来**跟踪、安排、暂停和中止**工作。

当一个**React元素**第一次被转换成一个**Fiber节点**时， React 使用该元素的数据在 createFiberFromTypeAndProps 函数中创建一个fiber。

> 在随后的更新中， React **重用** Fiber节点，只是**使用来自相应 React元素 的数据更新必要的属性**。如果相应的React元素不再从渲染方法中返回， React可能还需要根据关键 props在层次结构中移动节点或删除它。

因为**React为每个React元素创建了一个 fiber 节点**，由于我们有一个由元素组成的 element 树，所以我们也将有一个由fiber节点组成的fiber树。在我们的示例应用程序中，它看起来像这样。



所有的**Fiber节点**都是通过child、 sibling和return属性构建成**链表**连接起来的。

Current Tree 和 workInProgress Tree

> **在第一次渲染之后，React 最终会有一个 Fiber 树，它反映了用来渲染 UI 的应用程序的状态**。这个树通常被称为{当前树| Current Tree}。
>
> 当React开始**状态更新**时，它建立了一个所谓的{workInProgress 树| workInProgress Tree}，反映了**未来**将被刷新到屏幕上的状态。

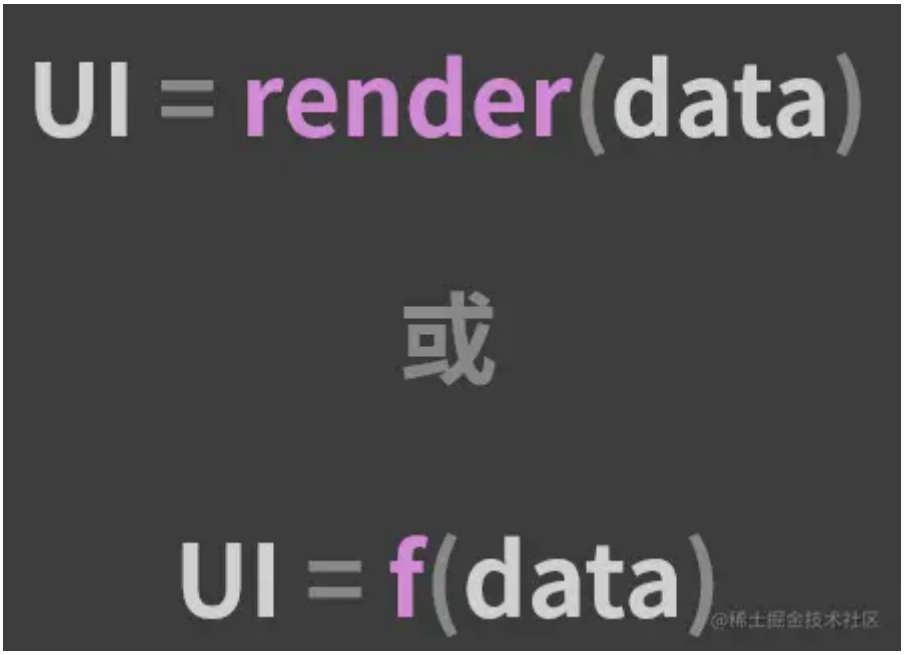
所有的工作都在workInProgress树的 fiber 上进行。当React穿过current树时，对于每个**现有**的fiber节点，它创建一个**备用节点**，构成 workInProgress树。这个节点是**使用render方法返回的React元素的数据创建**的。一旦更新处理完毕，所有相关的工作都完成了， React 就会有一个*备用的树*，准备刷新到屏幕上。**一旦这个workInProgress树被渲染到屏幕上，它就成为current树**。

React 的核心原则之一是**一致性**。 React 总是**一次性地更新所有DOM--它不会显示部分结果**。 workInProgress树作为一个用户不可见的{草稿|draft}，这样 React 可以**先处理所有的组件，然后将它们的变化刷新到屏幕上**。

> 每个fiber节点通过alternate属性保存着对**另一棵树**上的对应节点的引用。
> current树的一个节点指向workInProgress树的节点，反之亦然。

{副作用| Side-effects}

可以把React中的**组件看作是一个使用state和props来计算UI表现的函数**。



每一个操作，如**DOM的突变**或**调用生命周期方法**，都应该被视为一个**副作用**，或者简单地说，是一个{效果|effect}。

> 从React组件中执行过**数据获取**、**事件订阅**或**手动改变DOM**。我们称这些操作为"副作用"（或简称"效果"），因为它们会影响其他组件，而且不能在渲染过程中进行。

你可以看到大多数state和props的更新都会导致副作用的产生。由于**应用效果是一种工作类型**，fiber节点是一种方便的机制，除了更新之外，还可以**跟踪效果**。

> 每个fiber节点都可以有与之相关的效果。它们被编码在 effectTag 字段中。

所以**Fiber中的效果基本上定义了更新处理后需要对实例进行的操作**。

- 对于宿主组件（DOM元素），工作包括*添加*、*更新*或*删除*元素。
- 对于类组件，React 可能需要**更新Refs**并调用 componentDidMount 和 componentDidUpdate 生命周期方法。

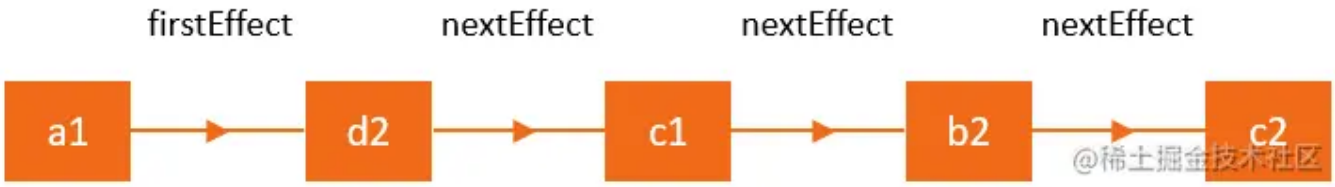
{效果清单| Effects list}

React处理更新的速度非常快，为了达到这种性能水平，它采用了一些有趣的技术。其中之一是**建立一个带有副作用的 fiber节点 的线性列表**，以便快速迭代。**迭代线性列表要比树形快得多**，而且不需要在没有副作用的节点上花费时间。

> 这个列表的目的是**标记有DOM更新或其他与之相关的副作用的节点**。这个**列表是 workInProgress 树的一个子集**，并且使用 nextEffect 属性链接，而不是current和 workInProgress 树中使用的 child 属性。

把 React 应用想象成一棵圣诞树，用 "圣诞灯 "把所有有效果的节点绑在一起。

当访问这些节点时， React 使用 firstEffect 指针来计算**列表的开始位置**，用 nextEffect将拥有效果的节点连接起来。 所以上图可以表示为这样的一个线性列表。



Fiber-Node的数据结构

现在让我们来看看为 ClickCounter 组件创建的fiber节点的结构。

javascript 复制代码

```
{
    stateNode: new ClickCounter,
    type: ClickCounter,
    alternate: null,
    key: null,
    updateQueue: null,
    memoizedState: {count: 0},
    pendingProps: {},
    memoizedProps: {},
    tag: 1,
    effectTag: 0,
    nextEffect: null
}
```

1. stateNode
    - 保存对与fiber节点相关的组件、DOM节点或其他React元素类型的类实例的**引用**
2. type
    - 定义了与该fiber相关的**函数或类**。
3. **tag**
    - 定义了fiber的类型。
      定义在调和算法中被用来确定需要做什么工作。
4. updateQueue
    - **状态更新、回调和DOM更新的队列**
5. memoizedState
    - **用于创建输出的fiber的state**
    - 当处理更新时，它反映了**当前屏幕上**呈现的状态。
6. memoizedProps
    - 在**上一次渲染过程**中用于创建输出的 fiber 的 props。
7. pendingProps
    - 从React元素的**新数据**中更新的props，需要应用于子组件或DOM元素。
8. key
    - 用于在一组子item中**唯一标识**子项的字段。

渲染算法

> React的工作主要分两个阶段进行：{渲染| Render}和{提交| Commit}。

在render阶段， React 通过 setState 或 React.render对预定的组件进行更新，并找出UI中需要更新的内容。

- 如果是**初次渲染**， React 为render方法返回的每个元素创建一个**新的** fiber节点。
- 在接下来的**更新中**，现有 React元素 的fiber被**重新使用和更新**。

该阶段的结果是**一棵标有副作用的fiber节点树**。这些效果描述了在接下来的**提交阶段**需要做的工作。在commit阶段， React **遍历标有效果的 fiber树，并将效果应用于实例**。它遍历 effect列表，执行DOM更新和其他用户可见的变化。

重要的是， render阶段的工作可以**异步进行**。 React 可以根据**可用的时间**来处理一个或多个fiber节点，然后停下来，把**已经完成的工作储存起来，并将处理 fiber 的操作**{暂停|yield}。然后从上次离开的地方继续。但有时，可能需要丢弃已完成的工作并从头开始。针对在这个阶段执行的工作的暂停操作**不会导致任何用户可见的UI变化**，如DOM更新。相比之下，接下来的**提交阶段总是同步的**。这是因为在这个阶段进行的工作会导致用户可见的变化，例如DOM更新。这就是为什么React需要一次性完成这些工作。

调用生命周期的方法是React执行的一种工作类型。有些方法是在render阶段调用的，有些是在commit阶段调用的。下面是在**render阶段工作时调用的生命周期的列表**。

- [UNSAFE_]componentWillMount（废弃）
- [UNSAFE_]componentWillReceiveProps（废弃）
- static getDerivedStateFromProps
- shouldComponentUpdate
- [UNSAFE_]componentWillUpdate（废弃）
- render

正如你所看到的，从16.3版本开始，一些在渲染阶段执行的传统生命周期方法被标记为 UNSAFE。它们现在在文档中被称为**遗留生命周期**。它们将在未来的16.x版本中被废弃。

我们来简单解释下，为什么会有生命周期会被遗弃。

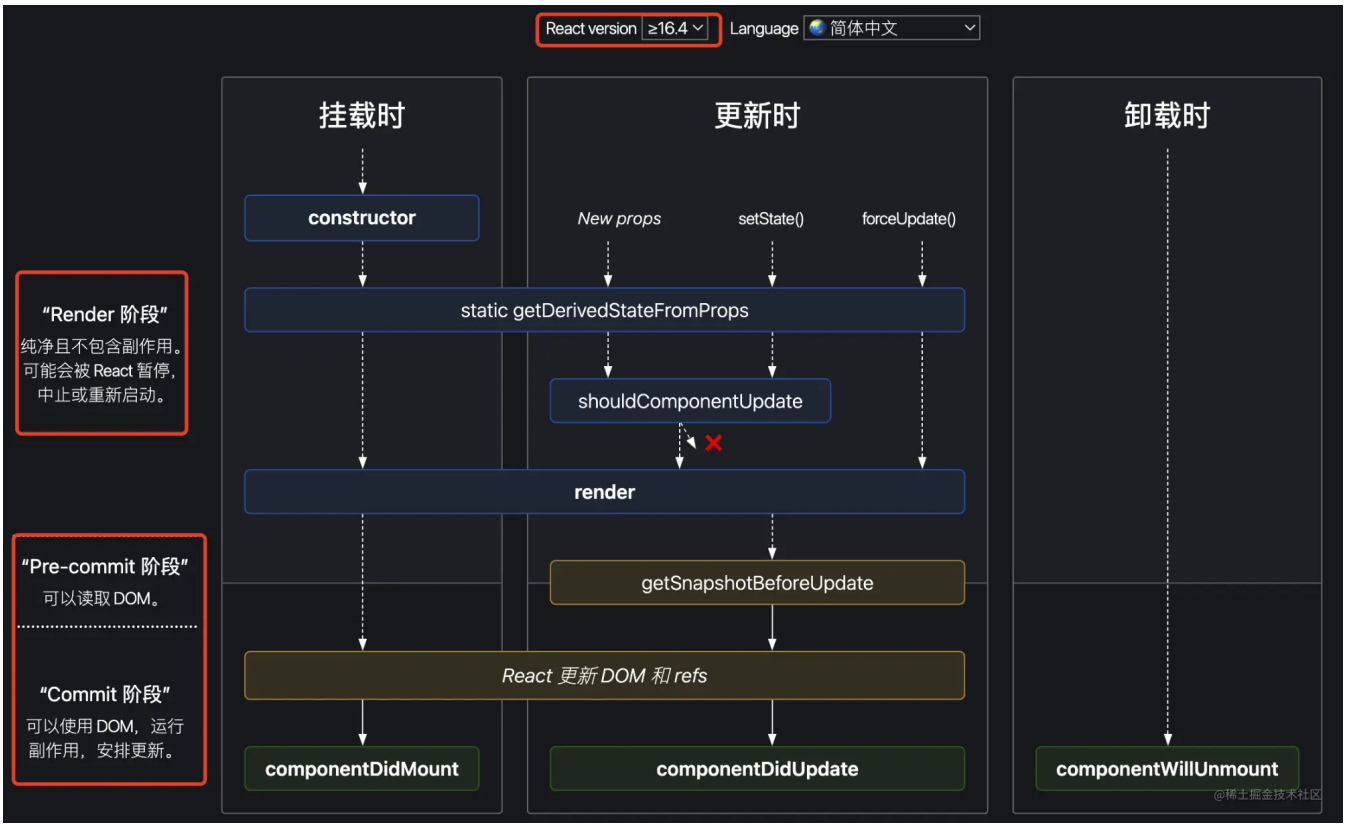
由于render阶段不会产生像DOM更新那样的副作用，React可以*异步处理组件的更新*（甚至有可能在多个线程中进行）。然而，标有 UNSAFE 的生命周期经常被滥用。开发者倾向于将有副作用的代码放在这些方法中，这可能会**给新的异步渲染方法带来问题**。

下面是在commit阶段执行的生命周期方法的列表。

- getSnapshotBeforeUpdate
- componentDidMount
- componentDidUpdate
- componentWillUnmount

因为这些方法在**同步提交阶段执行**，它们可能包含副作用并触及DOM。

这里我们贴一个针对react-16.4+版本的类组件的生命周期方法。



Render 阶段

> **调和算法**总是使用 renderRoot 函数从最上面的 HostRoot fiber节点开始。然而，React会跳过已经处理过的fiber节点，直到**找到工作未完成的节点**。

例如，如果你在组件树的深处调用 setState，React会从顶部开始，但迅速*跳过*父节点，直到它到达调用了setState方法的组件。

workLoop 主要流程

> **所有**fiber节点都在 workLoop 中被处理

下面是该循环的同步部分的实现。

javascript 复制代码

```
function workLoop(isYieldy) {
  if (!isYieldy) {
    while (nextUnitOfWork !== null) {
      nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
    }
  } else {...}
}
```

在上面的代码中， nextUnitOfWork 持有对来自 workInProgress 树的fiber节点的引用，该节点有一些工作要做。 当 React 遍历 Fiber 树时，它**使用这个变量来了解是否还有其他未完成工作的 Fiber 节点**。 处理current fiber后，该变量将包含对树中下一个fiber节点的引用或为空。

有 4 个主要函数用于遍历树并启动或完成工作：

1. performUnitOfWork
2. beginWork
3. completeUnitOfWork
4. completeWork

当 React 沿着树向下移动时。**它先完成孩子节点的处理，再转向其父节点**

从代码实现中可以看出， performUnitOfWork 和 completeUnitOfWork 都主要用于**迭代**，而**主要操作发生在 beginWork 和 completeWork 函数中**。

Commit 阶段

该阶段从函数 completeRoot 开始。 这是 React 更新 DOM 并调用**变动前后**生命周期方法的地方。

当 React 进入这个阶段时，它**有 2 棵树**。

- **第一个树**代表当前在屏幕上呈现的状态。
- **第二个树**是在**render阶段**构建了一个{备用树| alternate tree}。
    - 它在源代码中称为 finishedWork 或 workInProgress，表示**需要**在屏幕上反映的状态。
    - 该备用树通过child指针和sibling指针进行各个节点的连接。

还有一个**效果列表**——来自finishedWork树的节点**子集**，通过 nextEffect 指针链接。 请记住，**效果列表是render阶段的结果**。 渲染的重点是确定哪些节点需要插入、更新或删除，哪些组件需要调用其生命周期方法。 这就是效果列表告诉我们。 **它正是在commit阶段需要处理的节点集**。

在commit阶段运行的主要函数是 commitRoot。 在指定的fiber上执行更新操作。

以下是运行上述步骤的函数的要点：

javascript 复制代码

```
function commitRoot(root, finishedWork) {
  commitBeforeMutationLifecycles()
  commitAllHostEffects();
  root.current = finishedWork;
  commitAllLifeCycles();
}
```

> 这些**子函数中的每一个都实现了一个循环**，该循环遍历效果列表并检查效果的类型。当它找到与函数目的相关的效果时，它会应用它。

---

1. 突变前的生命周期
    - 对于类组件，此效果意味着调用 getSnapshotBeforeUpdate 生命周期方法。

2. DOM更新
    - commitAllHostEffects 是 React 执行 DOM 更新的函数。
3. 突变后的生命周期方法
    - commitAllLifecycles 是 React 调用**所有剩余生命周期方法** componentDidUpdate 和 componentDidMount 的函数。

Fiber 调和器

{Fiber 调和器| Fiber Reconciler}成为 React 16+ 版本的**默认调和器**，它完全重写了 React 原有的调和算法，以解决 React 中一些长期存在的问题。

这一变化使 React 摆脱了{同步堆栈调节器| Synchronous Stack Reconciler}的限制。以前，你可以添加或删除组件，但**必须等调用堆栈为空，而且任务不能被中断**。

使用新的调节器,也**确保最重要的更新尽快发生**。(更新存在优先级)

{堆栈调和器| Stack Reconciler}

> 为什么这被称为 "堆栈 "调节器？这个名字来自于 "堆栈 "数据结构，它是一个**后进先出**的机制。

我们从最熟悉的 ReactDOM.render(<App />, document.getElementById('root')) 语法开始探索。

ReactDOM 模块将 <App/ > 传递给调和器，但这里有两个问题:

- <App /> 指的是什么?
- 什么是调和器?

让我们来一一解答这些问题。

<App /> 指的是什么?

<App /> 是一个React元素。根据 React博客描述，“元素是一个描述*组件实例*或*DOM节点*及其所需属性的**普通对象**“。

换句话说，元素**不是实际的DOM节点或组件实例**；它们是一种向 React 描述它们是*什么类型的元素*，它们*拥有什么属性*，以及*它们的孩子是谁*的信息组织方式。

> React 元素在早期的React介绍文档中，有另外一个家喻户晓的名字：**{虚拟DOM| Virtual-DOM}**
> 只不过，V-Dom在理解上在某些场景下会产生歧义，所以逐渐被React 元素所替代

React {调和算法| Reconciliation}

该算法使得 React 更容易解析和遍历应用，用以建立对应的*DOM树*。**实际的渲染工作会在遍历完成后发生**。

当 React 遇到一个类或一个函数组件时，它会基于元素的props来渲染UI视图。

例如，如果<App>组件渲染了以下内容，那么 React 会遍历<Form>和<Button>组件,它们想根据相应的 props 渲染成什么。

```html
<Form>
  <Button>
    Submit
  </Button>
</Form>
```

React 会**重复这个过程**，直到它掌握了页面上与每个组件所对应的DOM元素的相关渲染信息。

> 这种通过**递归元素树**，以掌握React应用的组件树的DOM元素的过程，被称为**调和**。

在调和结束时，React 知道DOM树的结果，像 react-dom 或 react-native 这些**渲染器**渲染更新DOM节点所需的**最小变化集**。这意味着，当你调用 ReactDOM.render() 或

setState()时， React 就会执行调和处理。

在 setState 的情况下，它执行了一个遍历，并通过**将新的树与渲染的树进行比较**来确定树中的变化。然后，它将这些变化应用到**当前树**上。

递归操作

在上文介绍**堆栈调和器**中得知，在进行调和处理时，会执行**递归操作**，而递归操作和**调用栈**有很大的关系，进而我们可以得出，递归和**堆栈**也有千丝万缕的联系。

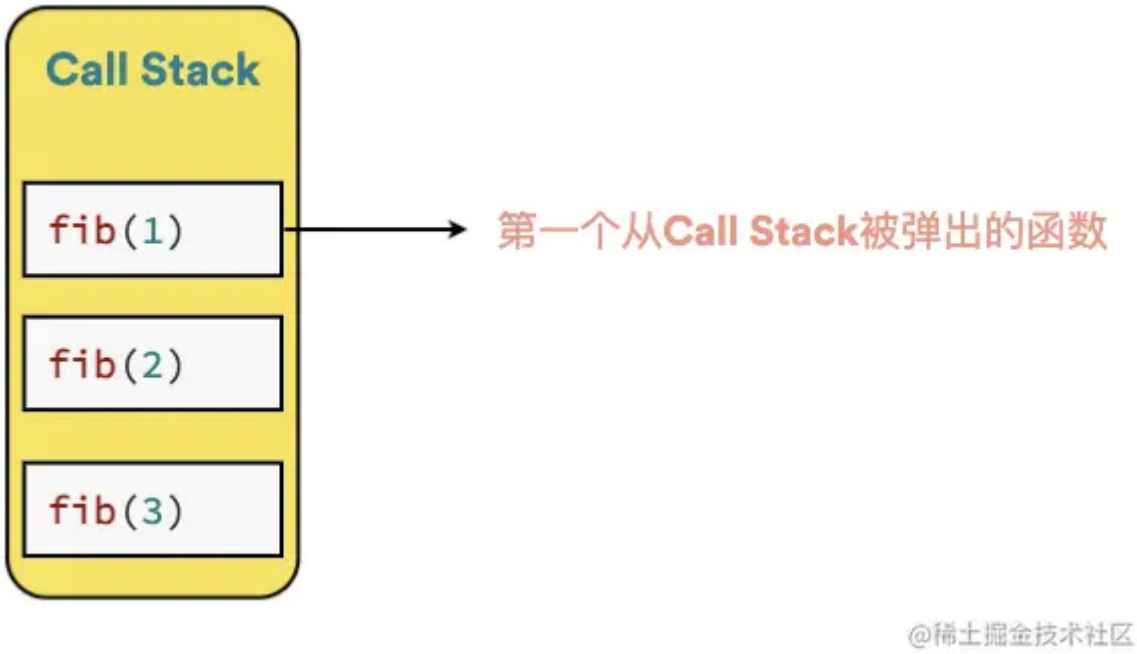
用一个简单的例子，看看在**调用栈**中会发生什么。

javascript 复制代码

```
function fib(n) {
  if (n < 2){
    return n
  }
  return fib(n - 1) + fib (n - 2)
}

fib(3)
```

我们可以看到，调用堆栈将对fib()的每一次调用都**推入堆栈**，直到弹出fib(1)（第一个返回的函数调用）。



我们刚才看到的调和算法是一个**纯粹的递归算法**。*一个更新会导致整个子树立即重新渲染*。虽然这很好用，但这也有一些局限性。

> 在用户界面中，**没有必要让每个更新都立即显示**；
> 事实上，这样做可能会造成浪费，导致**帧数下降并降低用户体验**。

另外，不同类型的更新**有不同的优先级**--动画更新必须比数据存储的更新完成得快。

页面{丢帧| dropped frames} 问题

{帧率| Frame Rate}

> **帧率**是指连续图像出现在显示器上的**频率**。
> 我们在电脑屏幕上看到的一切都**由屏幕上播放的图像或帧组成，其速度在眼睛看来是瞬间的**。

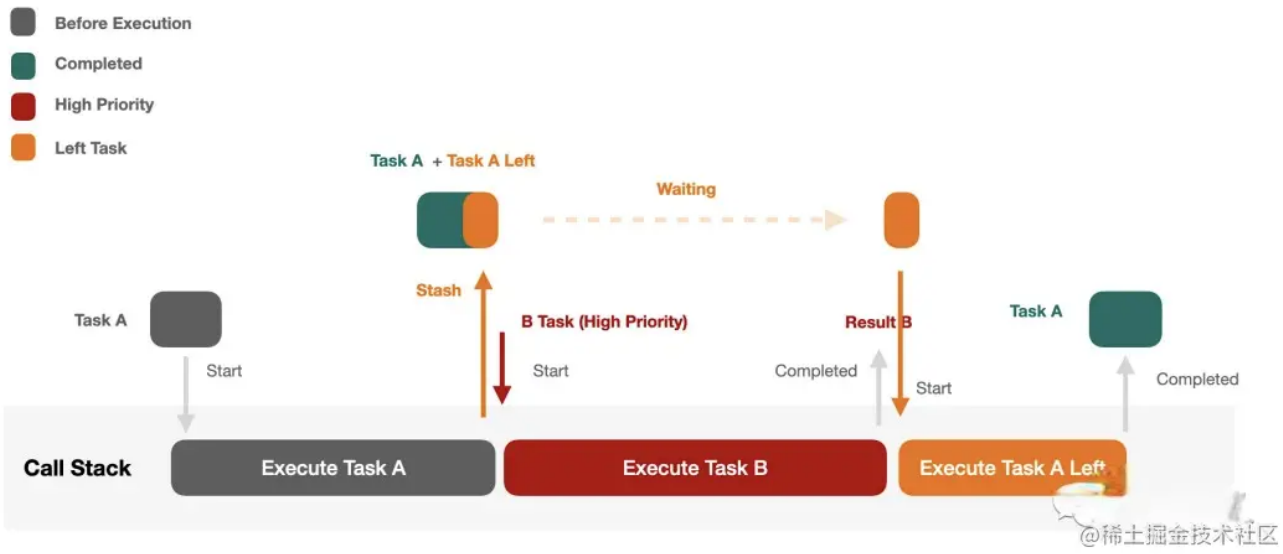
通常情况下，为了画面流畅和即时，视频的播放速度必须达到**每秒30帧**（FPS）左右；任何更高的速度都能带来更好的体验。

现在大多数设备都是以60FPS刷新屏幕，1/60=16.67ms，这意味着**每16ms就有一个新的帧显示**。这个数字很重要，因为如果 React渲染器在屏幕上渲染的时间**超过**16ms，**浏览器就会丢弃该帧**。

然而，在现实中，浏览器要做一些**内部工作**，所以你的所有工作**必须在10ms内完成**。当你不能满足这个预算时，*帧率就会下降*，**内容就会在屏幕上抖动**。这通常被称为 jank，它对用户的体验有负面影响。

如果每次有更新时，React 调和算法都会遍历整个App树，并重新渲染，**如果**遍历的时间超过16ms，就会**掉帧**。

这也是许多人希望更新按**优先级分类**，而不是盲目地把每个更新都传给**调和器**。另外，许多人希望能够**暂停并在下一帧恢复工作**。这样一来，React可以更好地控制与16ms渲染预算的工作。



这导致React团队重写了调和算法，它被称为Fiber。那么，让我们来看看Fiber是如何解决这个问题的。

React Fiber 如何工作的

总结一下实现Fiber所需要的功能

- 为不同类型的工作分配**优先权**

- **暂停工作**，以后再来处理
- 如果不再需要，就放弃工作
- **重复使用**以前完成的工作

实现这样的事情的挑战之一是 JavaScript 引擎的**工作方式**和**语言中缺乏线程**。为了理解这一点，让我们简单地探讨一下 JavaScript 引擎如何处理执行上下文。

JavaScript的{执行堆栈| Execution Stack}

每当你在 JavaScript 中写一个函数， JavaScript 引擎就会创建一个*函数执行上下文*。

每次 JavaScript 引擎启动时，它都会创建一个**全局执行上下文**，以保存全局对象；例如，浏览器中的window对象和Node.js中的global对象。 JavaScript 使用一个堆栈数据结构来处理这两个上下文，也被称为**执行堆栈**。

因此，当存在如下代码时， JavaScript 引擎首先创建一个全局执行上下文，并将其推入执行栈。

javascript 复制代码

```
function a() {
  console.log("i am a")
  b()
}

function b() {
  console.log("i am b")
}

a()
```

然后，它为 a() 函数创建一个函数执行上下文。由于b()是在a()中调用的，它为b()创建了另一个函数执行上下文，并将其推入堆栈。

当b()函数返回时，引擎销毁了b()的上下文。当我们退出a()函数时， a()的上下文被销毁。执行过程中的堆栈看起来像这样。

但是，当浏览器发出像HTTP请求这样的**异步事件**时会发生什么？ JavaScript 引擎是储存执行栈并处理异步事件，还是等待事件完成?

JavaScript 引擎在这里做了一些不同的事情：在**执行堆栈的底部**， JavaScript 引擎有一个**队列数据结构**，也被称为{事件队列| Event Queue}。事件队列**处理异步调用**。

JavaScript 引擎通过*等待执行栈清空来处理队列中的项目*。所以，每次执行栈清空时，JavaScript 引擎都会检查事件队列，从队列中弹出项目，并处理事件。

> 值得注意的是，只有当**执行栈为空**或者**执行栈中唯一的项目是全局执行上下文**时，JavaScript 引擎才会检查事件队列。

虽然我们称它们为异步事件，但这里有一个微妙的区别：**事件在到达队列时是异步的，但在实际处理时，它们并不是真正的异步**。

回到我们的堆栈调节器，当 React 遍历树时，它在执行堆栈中这样做。所以，当更新发生时，它们会在事件队列中进行**排队**。只有当执行栈清空时，更新才被处理。

这正是Fiber解决的问题，它重新实现了**具有智能功能的堆栈**--例如，暂停、恢复和中止。

> Fiber是对堆栈的**重新实现**，专门用于React组件。
> 可以把一个Fiber看成是一个**虚拟的堆栈框架**。

重新实现堆栈的**好处**是，你可以把**堆栈帧保留在内存中**，并随时随地执行它们。

简单地说， Fiber代表了**一个有自己的虚拟堆栈的工作单位**。在以前的调和算法的实现中， React 创建了一棵对象树（React元素），这些对象是**不可变**的，并递归地遍历该树。

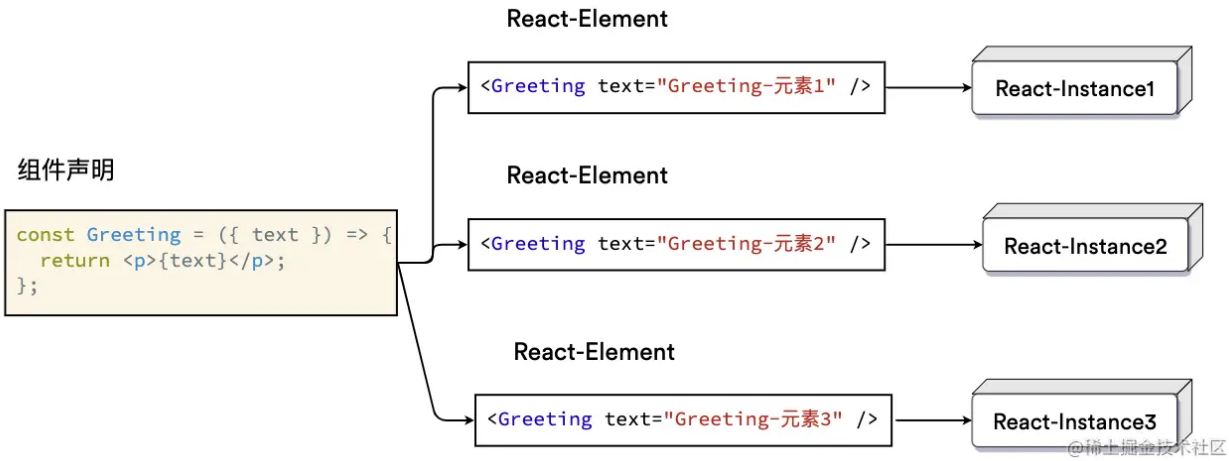
在当前的实现中， React 创建了**一棵可变的Fiber节点树**。 Fiber节点有效地持有组件的 state、 props和它所渲染的DOM元素。

而且，由于fiber节点可变的， React **不需要为更新而重新创建每个节点；它可以简单地克隆并在有更新时更新节点**。

在fiber树的情况下， React 并不执行递归遍历。相反，它创建了一个**单链的列表**，（Effect-List）并执行了一个**父级优先**、**深度优先**的遍历。

React 元素 VS 组件

1. React组件被**声明一次**
2. 但组件可以作为JSX中的React元素被**多次使用**
3. 当元素被使用时，它就成为该组件的**一个实例**，挂载在React的组件树中



4.

> React-Component是一个组件的**一次性声明**，但它可以作为JSX中的React-Element使用一次或多次。
> 也就是说 React-Component 和 React-Element 是**1对多**的关系

React-全局状态管理

全局状态管理库需要解决的问题

1. 从组件树的**任何地方**读取存储的状态
    - 允许开发者将他们的状态**持久化在内存中**
    - 当涉及到实际**状态存储**时，有两种主要方法
        1. **由React自身维护**
        2. **将数据存储在React外部**，然后以**单例**的形式存储
2. 写入存储状态的能力
    - 应该提供一个直观的API来读取和写入存储的数据。
    - React是依靠**数据引用相等**和**不可变的更新操作**来判断是否触发重新渲染
    - Redux 遵循这种模式，要求**所有的状态更新都以不可变的方式进行**
        - 一个弊端就是你必须写*大量的模板*，以满足那些早已习惯数据可随时变更的人进行数据更新
    - 在一些**后-redux**的全局状态管理解决方案中还有其他一些库，如Valtio，也允许开发者使用可变风格的API。
3. 提供**优化渲染**的机制
    - 随着数据量的增加，当状态发生变化时的**调和过程**是一件耗时操
    - 优化这一过程是状态管理库需要解决的最大挑战之一。通常有两种主要的方法。
        1. 允许开发者**手动优化**这个过程
        2. 为开发者**自动处理**

           Valtio:它在JS引擎下使用Proxy来自动跟踪事物的更新，并自动管理一个组件何时应该重新渲染。
4. 提供**优化内存使用**的机制
    1. 利用React **生命周期**来存储状态意味着更容易利用组件卸载时的**自动垃圾收集**。
    2. 对于像Redux这样提倡**单一全局存储模式**的库，你需要对其中的存储的数据进行**手动回收**。
5. 与**并发模式的兼容性**
    - **并发模式**允许React在**渲染过程中 "暂停 "并切换优先级**。

- 对于状态管理库来说，如果在渲染过程中读取的值发生了变化，那么两个组件就有可能从外部存储中读取不同的值。这就是所谓的 **数据撕裂**
- React 团队为库创建者(Redux/Mobx)创建了 useSyncExternalStore hook来解决这个问题

6. 数据的**持久化**
7. **上下文丢失**问题
   - 这是将多个 react渲染器 混合在一起的应用程序的一个问题
   - 有一个同时利用 react-dom 和 react-three-fiber 库的应用程序。在这种情况下，React 无法调和两个独立的上下文。
8. **props失效**问题
9. **孤儿**问题
   - 这指的是 Redux 的一个老问题，在这个问题上，如果子组件先被挂载，并在父组件之前和Redux建立关联，那么如果在父组件被挂载之前更新状态，就会造成不一致的情况。

状态管理生态系统的发展史

1. Redux的最初崛起
   - Redux 是 Flux 模式的**最早实现之一**，得到了广泛的采用。
   - 它提倡使用**单一存储**，部分灵感来自**Elm架构**。
   - 随着时间的推移， Redux 在一些特定的领域，变现不尽人意，导致它不再受到青睐
     1. 小型应用程序中的问题：
        - 从组件树中的**任何地方**访问存储的状态，以避免在多个层次上对数据和函数进行**逐层向下传递**。
        - 对于那些组件层级简单、没有什么交互性的简单应用来说，这往往是**矫枉过正**
     2. 大型应用程序中的问题

- 随着时间的推移，我们较小的应用程序发展成为较大的应用程序
- 在实践中，一个前端应用程序有许多**不同类型的状态**。每种类型都有属于各自的子问题。 大致可以分为4类
    1. **本地**UI状态
    2. **远程**服务器缓存状态
    3. url状态
    4. **全局**共享状态
- Redux 倾向于**吸纳所有的状态**，不管它是什么类型，因为它提倡单一的存储。这通常会**导致将所有的东西存储在一个大的单体存储中**。

2. 不再强调Redux的作用
    - 很多Web应用都是CRUD（create，read，update 和 delete）风格的应用，主要目的是**将前端与远程状态数据同步**。
    - 值得花时间解决的主要问题是**远程服务器缓存**的一系列问题
    - 这些问题包括如何 *获取*、*缓存*和与 *服务器状态同步*。
3. 偏向React-Hook的实现方式
    - 随着hook的出现。一时间，开发应用管理状态的方式又从Redux这样的重度抽象摇身一变为利用新的hook API的原生上下文

解决远程状态管理问题的专用库的崛起

对于大多数CRUD风格的Web应用来说，**本地状态**结合专门的**远程状态管理库**能解决所有状态都杂糅在一起的问题。

这个趋势中的一些例子库包括

1. React query、
2. SWR、
3. Apollo
4. Relay

全局状态管理库和模式的新浪潮

自下而上模式的崛起

我们可以看到以前的状态管理解决方案，如Redux，设计理念是状态 **自上而下**流动。它**倾向于在组件树的顶端吸走所有的状态**。状态被维护在组件树的高处，下面的组件通过选择器拉取他们需要的状态。

在新的组件构建理念中，一种**自下而上**的观点对构建具有组合模式的应用具有很好的指导作用。

而hook就是这种理念的践行者，即把 *可组合的部件*放在一起形成一个更大的整体。

> 通过hook，我们可以从具有巨大全局存储的**单体状态管理**转变为向自下而上的 **微状态管理**，通过hook消费更小的状态片。

像Recoil和Jotai这样的流行库以其 **原子状态**的概念体现了这种自下而上的理念。

> **原子是一个最小但完整的状态单位**。它们是小块的状态，可以连接在一起形成新的**衍生状态**。最终形成了一个应用状态图。

这个模型允许你自下而上地建立起**状态图**。并通过仅使图中已更新的原子失效来优化渲染。

现代库如何解决状态管理的核心问题

下面是每个库为解决状态管理的每个核心问题所采取的不同方法的简化总结。

从子树的任何地方读取存储状态

库	更新时机	API示例
React-Redux	嵌入到React运行时	useSelector(state => state.foo)
Recoil	嵌入到React运行时	const todos = atom({ key: 'todos', default: [] }) const todoList = useRecoilValue(todos)
Jotai	嵌入到React运行时	const countAtom = atom(0) const [count, setCount] = useAtom(countAtom)
Valtio	JS引擎维护	const state = proxy({ count: 0 }) const snap = useSnapshot(state) state.count++

写入和更新存储状态的能力

库	API更新类型
React-Redux	更新不可变
Recoil	更新不可变
Jotai	更新不可变
Zustand	更新不可变
Valtio	更新可变

运行时性能重新渲染的优化

- **手动优化**通常意味着创建订阅特定状态的选择器函数（Selector）。
    - 这样做的**好处**是，消费者可以**精细地控制**如何订阅和优化订阅该状态的组件将如何重新渲染。

    - **缺点**是这是一个手动的过程，可能容易出错，而且人们可能会说这需要不必要的开销，不应该成为API的一部分。

- **自动优化**是指库对这个过程进行优化，只重新渲染必要的东西，自动地，为你作为一个消费者。
    - 这里的**优点**当然是易于使用，而且消费者能够专注于开发功能，而不需要担心手动优化。
    - 这样做的**缺点**是，作为消费者，优化过程是一个**黑盒**，如果没有手动优化的方式，有些特定场景会让人很抓狂。

库	描述
React-Redux	利用特定选择器函数，**手动优化**
Recoil	通过订阅原子的**半手动方式**
Jotai	通过订阅原子的**半手动方式**
Zustand	利用特定选择器函数，**手动优化**
Valtio	通过 Proxy 快照进行**自动**优化

内存优化

内存优化往往只在非常大的应用程序上才会出现问题。

与大型单体存储相比，较小的独立存储的好处是，当所有订阅的组件卸载时，它们可以自动收集垃圾。而大型单体存储如果没有适当的内存管理，则更容易出现内存泄漏。

库	描述
React-Redux	**手动**管理
Recoil	0.3.0版本后- **自动**管理
Jotai	**自动**管理 - atoms作为键存储在WeakMap中
Zustand	**半自动**--API可用来帮助手动取消订阅的组件
Valtio	**半自动**--订阅组件卸载时收集的垃圾

构建面向未来的前端架构

组件思维

> React 是最流行的**基于组件**的前端框架。

在React官网文档中有一篇Thinking in react，它阐述了在以 **React方式**构建前端应用程序时如何思考的心智模型。

它所阐述的主要原则，指导你在构建一个组件时需要考虑哪些方面。

- **组件的责任是什么**？好的组件API设计自然遵循{单一责任原则|Single Responsibility Principle}，这对{组合模式|Composition Patterns}很重要。
- 什么是其**状态的最小，但完整**的表示？我们的想法是，从**最小但完整**的状态开始，你可以从中推导出变化。
- **状态应该住在哪里**？一般来说，如果一个状态可以被变成一个组件的本地状态，优先将其设置为组件本地state。**组件内部对全局状态的依赖越多，它们的可重用性就越低**。

> **一个组件最好只做一件事**。如果它最终成长起来，它应该被分解成更小的子组件。

{自上而下|Top down} 与 {自下而上|Bottom up}

组件是React等现代框架的**核心抽象单位**。有两种主要的方式来考虑创建它们。

> 你可以**自上而下**或**自下而上**地构建。
>
> - 在比较简单的项目中，**自上而下**比较容易
> - 而在比较大的项目中，**自下而上**比较容易

自上而下的构建组件

自上而下通常是最直观和最直接的方法。这也是从事功能开发的开发人员在构建组件时最常见的心智模式。

自上而下的方法是什么样子的？ 当开始页面结构设计时，常见的建议是："在用户界面周围画上方框，这些将成为你的组件"。

这构成了我们最终创建的*顶层组件*的基础。采用这种方法，我们通常以创建一个*粗略的组件*来开始构建页面。

对于较小的项目，这种方法能够简单快速的构建页面。**但是**，针对大型项目来讲，这种自上而下的数据流向就会出现问题。

自上而下模式的弊端

这里有一个比较常见的场景。在一个正在快速迭代的项目中。你已经通过*画方框*的方式来界定出你组件的范围并将其交付到页面中。但是，新需求出现了，需要你针对导航组件进行修改。

在其对现有组件的抽象思路和API有一个简单了解前提下，需求继任者在需求变更的*裹挟*下，在开始coding之前，它可能会有如下的心理路程。

- 思考这是否是**正确的抽象**。如果不是，在处理新需求的过程中，就可以通过*代码重构*对其进行改造处理。

- 增加一个额外的属性。在一个简单的条件后面添加新的功能(React中的条件渲染)，只需要判定特定的属性，来处理新增需求的变更。它的好处就是，快。没错，就是快。

在规模的加持下，每次较小的决定都会导致我们的组件变得更加复杂。当组件变的臃肿&复杂的时候，我们已经违背了React中构建组件的基本原则之一 -- 简单性(一个组件最好只做一件事)

需求的不断变更，事情变得愈发不可控制。

一开始是一个相对简单的组件，有一个简单的API，但在几个快速迭代的过程中，很快就会发展成其他东西。

基于此时的现状，下一个需要使用或改编这个组件的开发者或团队要面对的是**一个需要复杂配置的单体组件，而且很可能根本没有相关使用文档**。

在这一点上，一个常见的情况是考虑扔掉一切，**从头开始重写这个组件**。

{单体组件|Monolithic Components}的健康增长

> 除了第一次，一切都应该自上而下地构建

正如我们所看到的，**单体组件是那些试图做太多事情的组件**。它们通过props接收过多的数据或配置选项，管理过多的状态，并输出过多的用户界面。

它们通常从简单的组件开始，通过需求的不断变更和新增，随着时间的推移，最终做了太多的事情。

一开始只是一个简单的组件，在几个迭代过程并追加新功能后，就会变成一个单体组件。

当这种情况发生在多个组件上时，并且多人同时在同一个代码库中开发，代码很快就会变得更难改变，页面也会变的更慢。

以下是单体组件可能导致 *性能问题* 或者 *代码臃肿* 的原因。

过早的抽象化

这是另外一个导致单体组件出现的原因。 这与作为软件开发者早期被灌输的一些常见*开发模式*有关。特别是对 {DRY|Don't Repeat Yourself}的原则。

阻碍跨团队的代码重用

如果它是一个"全有或全无"的单体组件，那么就很难复用现有的逻辑。与重构或者直接修改别人组件或者库的方式相比，在你自己的组件中重新实现相关逻辑或者利用*条件*判断来进行逻辑复用，显的更加安全。 但是，如果此处变更涉及多个组件，那就需要对多个组件进行相同的处理。

增加包的大小

单体组件阻止了这些努力的发生，因为你*必须把所有的东西作为一个大块的组件来加载*。

如果独立的组件的话，这些组件就可被优化，并且只在用户**真正需要**的时候加载。**消费者只需支付他们实际使用的性能价格**。

运行时性能不佳

像React这样的框架，有一个简单的state->UI的功能模型，是令人难以置信的生产力。但是，为了查看虚拟DOM中的变化而进行的**调和操作**在页面规模比较大的情况下是很昂贵的。**单体组件很难保证在状态发生变化时只重新渲染最少的东西**。

在单体组件和一般的自上而下的方法中，找到这种分割是很困难的，容易出错，而且常常导致过度使用memo()。

自下而上的构建组件

与自上而下的方法相比，自下而上的方法往往不那么直观，而且最初可能会比较慢。

关于事情应该被分解到什么程度，没有一个正确的答案。管理这个问题的关键是使用{单一责任原则|Single Responsibility Principle}作为指导准则。

自下而上的心智模式与自上而下有什么不同？

总的复杂性分布在许多较小的单一{责任组件|Responsibility Components}中，而不是一个单一的单体组件。

自下而上的方法的最终结果是直观的。它需要更多的前期努力，因为更简单的API的复杂性被封装在各个组件中。但这也使得它成为一种能够进行页面自由组装的优势。

与我们自上而下的方法相比，好处很多。

1. 使用该组件的不同团队只需对他们**实际导入和使用的组件**进行维护
2. 可以很容易地用**代码分割**和**异步加载**那些对用户来说不是优先显示的元素
3. **渲染性能更好，更容易管理**，因为只有因更新而改变的子树需要重新渲染
4. 从代码结构的角度来看，它也更具有**可扩展性**，因为每个组件都可以被单独处理和优化。

自下而上*最初比较慢*，但从长远来看会更快，因为它的扩展性更强。你可以更容易地避免仓促的抽象，这是防止单体组件泛滥的最好方法。

假设我们在组装现有的页面，在采用自下而上的构建方式下，时间和精力往往耗费在**零件组装**上。但是从后期的可维护性来讲，这是一个值得做的事。

自下而上方法的力量在于，你的页面构建以**我可以将哪些简单的基础原件组合在一起以实现我想要的东西**为前提，而不是一开始就考虑到某个特定的抽象。

> 敏捷软件开发最重要的经验之一是**迭代的价值**

自下而上的方法可以让你在长期内更好地进行迭代。

避免单体组件的策略

1. 平衡单一责任与DRY的关系
2. {控制反转|Inversion of Control}
    - 理解这一原则的一个简单例子是callback和promise之间的区别。

        - 对于callback，你不一定知道这个函数会去哪里，会被调用多少次，或者用什么调用。
        - promise将控制权转回给消费者，所以你可以开始组成你的逻辑，假装value已经在那里了。
    - 通过children暴露slot(槽)，或者通过renderProps来保持消费者对内容的控制权
3. 组件扩展
    - 理想情况下，你的组件只做一件事。
    - 因此，在预制抽象的情况下，消费者根据它们需要实现的操作，用他们自己的功能对其进行**包装扩展**。
4. 利用storybook驱动的发展
    1. 根据组件的实际作用为其命名
    2. 避免包含实施细节的props名称
    3. 谨慎对待通过props进行的配置
    4. 避免在渲染方法中定义组件
        - 在一个组件中拥有 **辅助**组件是很常见的。这些组件最终会在每次渲染时被重新加载，并可能导致一些奇怪的错误。
        - 此外，有多个内部的renderX、 renderY方法往往是一种不好的举措。这些通常是一个组件变得单一化的标志，这些都是需要被进行分解处理的点。
