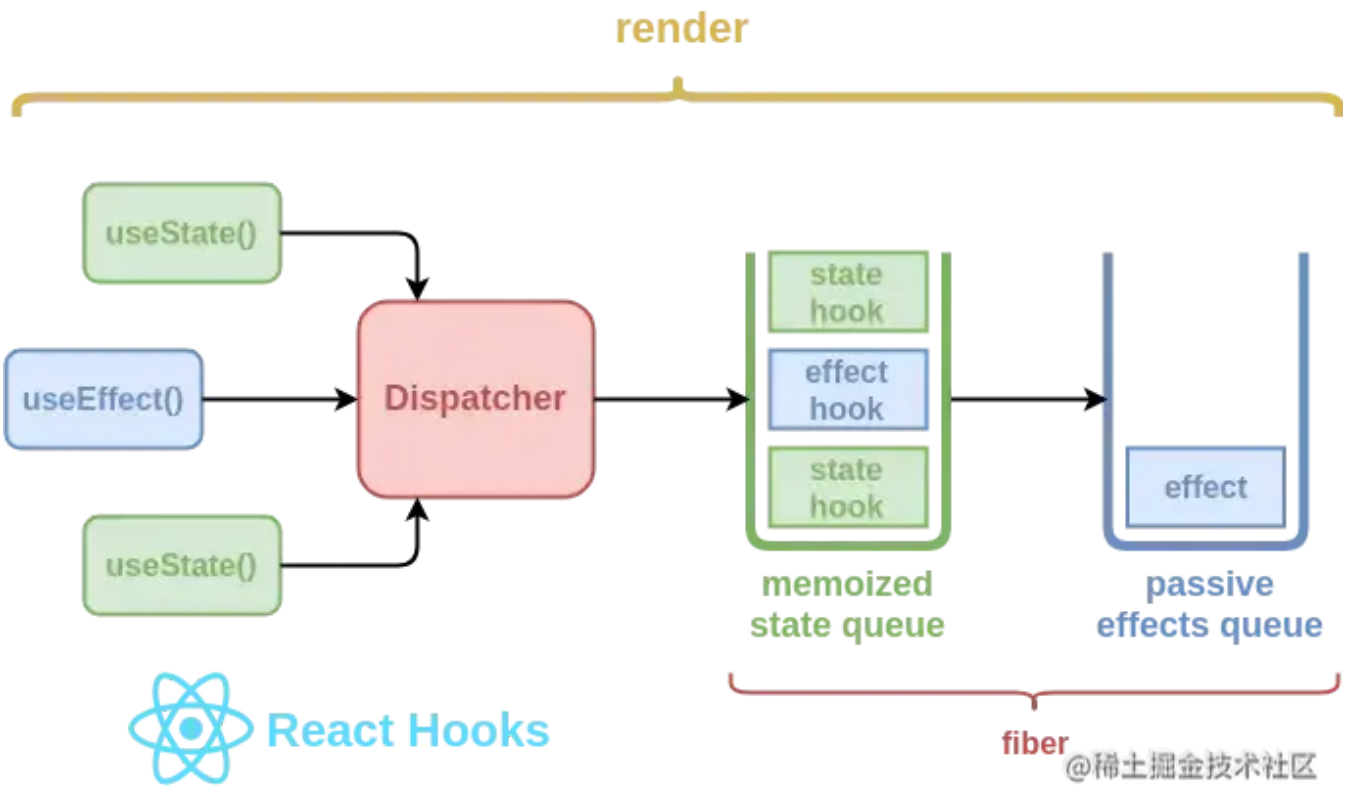
react源码中的hooks

今天，让我们一起深入探究 React Hook 的实现方法，以便更好的理解它。但是，它的各种神奇特性的不足是，一旦出现问题，调试非常困难，这是由于它的背后是由复杂的堆栈追踪（stack trace）支持的。因此，通过深入学习 React 的新特性：hook 系统，我们就能比较快地解决遇到的问题，甚至可以直接杜绝问题的发生。

> **在开始讲解之前，我先声明我不是 React 的开发者或者维护者，所以我的理解可能也并不是完全正确。我确实非常深入地研究过了 React 的 hook 系统的实现，但是无论如何我仍无法保证这就是 React 实际的工作方式。话虽如此，我还是会用 React 源代码中的证据和引用来支持我的文章，使我的论点尽可能坚实。**



React hook 系统概要示意图

我们先来了解 hook 的运行机制，并要确保它一定在 React 的作用域内使用，因为如果 hook 不在正确的上下文中被调用，它就是毫无意义的，这一点你或许已经知道了。

Dispatcher

dispatcher 是一个包含了 hook 函数的共享对象。基于 ReactDOM 的渲染状态，它将会被动态的分配或者清理，并且它能够确保用户不可在 React 组件之外获取 hook（详见源码）。

在切换到正确的 Dispatcher 以渲染根组件之前，我们通过一个名为 `enableHooks` 的标志来启用/禁用 hook。在技术上来说，这就意味着我们可以在运行时开启或关闭 hook。React 16.6.X 版本中也有对此的实验性实现，但它实际上处于禁用状态（详见源码）

当我们完成渲染工作后，我们将 dispatcher 置空并禁止用户在 ReactDOM 的渲染周期之外使用 hook。这个机制能够保证用户不会做什么蠢事（详见源码）。

dispatcher 在每次 hook 的调用中都会被函数 `resolveDispatcher()` 解析。正如我之前所说，在 React 的渲染周期之外，这些都无意义了，React 将会打印出警告信息：**“hook 只能在函数组件内部调用”**（详见源码）。

```csharp 复制代码
let currentDispatcher
const dispatcherWithoutHooks = { /* ... */ }
const dispatcherWithHooks = { /* ... */ }

function resolveDispatcher() {
  if (currentDispatcher) return currentDispatcher  throw Error("Hooks can't be called")}function use
  const dispatcher = resolveDispatcher()
  return dispatcher.useXXX(...args)
}

function renderRoot() {
  currentDispatcher = enableHooks ? dispatcherWithHooks : dispatcherWithoutHooks  performWork()  cur
}
```

dispatcher 实现方式概览。

现在我们简单了解了 dispatcher 的封装机制，下面继续回到本文的核心 —— hook。下面我想先给你介绍一个新的概念：

hook 队列

在 React 后台，hook 被表示为以调用顺序连接起来的节点。这样做原因是 hook 并不能简单的被创建然后丢弃。它们有一套特有的机制，也正是这些机制让它们成为 hook。一个 hook 会有数个属性，在继续学习之前，我希望你能牢记于心：

- 它的初始状态会在初次渲染的时候被创建。
- 它的状态可以在运行时更新。
- React 可以在后续渲染中记住 hook 的状态。
- React 能根据调用顺序提供给你正确的状态。
- React 知道当前 hook 属于哪个 fiber。

另外，我们也需要重新思考看待组件状态的方式。目前，我们只把它看作一个简单的对象：

```css
{
  foo: 'foo',
  bar: 'bar',
  baz: 'baz',
}
```

旧视角理解 React 的状态

但是当处理 hook 的时候，状态需要被看作是一个队列，每个节点都表示一个状态模型：

```yaml
{
  memoizedState: 'foo',
  next: {
    memoizedState: 'bar',
    next: {
      memoizedState: 'bar',
      next: null
    }
  }
}
```

新视角理解 React 的状态

单个 hook 节点的结构可以在源码中查看。你将会发现，hook 还有一些附加的属性，但是弄明白 hook 是如何运行的关键在于它的 `memoizedState` 和 `next` 属性。其他的属性会被 `useReducer()` hook 使用，可以缓存发送过的 action 和一些基本的状态，这样在某些情况下，reduction 过程还可以作为后备被重复一次：

- `baseState` —— 传递给 reducer 的状态对象。
- `baseUpdate` —— 最近一次创建 `baseState` 的已发送的 action。
- `queue` —— 已发送 action 组成的队列，等待传入 reducer。

不幸的是，我还没有完全掌握 reducer 的 hook，因为我没办法复现它任何的边缘情况，所以讲述这部分就很困难。我只能说，reducer 的实现和其他部分相比显得很不一致，甚至它自己源码中的注解都声明“不确定这些是否是所需要的语义”；所以我怎么可能确定呢？！

所以我们还是回到对 hook 的讨论，在每个函数组件调用前，一个名为 `prepareHooks()` 的函数将先被调用，在这个函数中，当前 fiber 和 fiber 的 hook 队列中的第一个 hook 节点将被保存在全局变量中。这样，我们无论何时调用 hook 函数（`useXXX()`），它都能知道运行上下文。

```javascript 复制代码
let currentlyRenderingFiber
let workInProgressQueue
let currentHook

// 源代码：https://github.com/facebook/react/tree/5f06576f51ece88d846d01abd2ddd575827c6127/react-reco
function prepareHooks(recentFiber) {
  currentlyRenderingFiber = workInProgressFiber
  currentHook = recentFiber.memoizedState
}

// 源代码：https://github.com/facebook/react/tree/5f06576f51ece88d846d01abd2ddd575827c6127/react-reco
function finishHooks() {
  currentlyRenderingFiber.memoizedState = workInProgressHook
  currentlyRenderingFiber = null
  workInProgressHook = null
  currentHook = null
}

// 源代码：https://github.com/facebook/react/tree/5f06576f51ece88d846d01abd2ddd575827c6127/react-reco
function resolveCurrentlyRenderingFiber() {
  if (currentlyRenderingFiber) return currentlyRenderingFiber
  throw Error("Hooks can't be called")
}
// 源代码：https://github.com/facebook/react/tree/5f06576f51ece88d846d01abd2ddd575827c6127/react-reco
function createWorkInProgressHook() {
  workInProgressHook = currentHook ? cloneHook(currentHook) : createNewHook()
  currentHook = currentHook.next
  workInProgressHook
}

function useXXX() {
  const fiber = resolveCurrentlyRenderingFiber()
  const hook = createWorkInProgressHook()
  // ...
}

function updateFunctionComponent(recentFiber, workInProgressFiber, Component, props) {
```

```
  prepareHooks(recentFiber, workInProgressFiber)
  Component(props)
  finishHooks()
}
```

相关参考视频讲解：进入学习

hook 队列实现的概览。

一旦更新完成，一个名为 `finishHooks()` 的函数将会被调用，在这个函数中，hook 队列中第一个节点的引用将会被保存在已渲染 fiber 的 `memoizedState` 属性中。这就意味着，hook 队列和它的状态可以在外部定位到。

javascript 复制代码

```
const ChildComponent = () => {
  useState('foo')
  useState('bar')
  useState('baz')

  return null
}

const ParentComponent = () => {
  const childFiberRef = useRef()

  useEffect(() => {
    let hookNode = childFiberRef.current.memoizedState

    assert(hookNode.memoizedState, 'foo')
    hookNode = hooksNode.next
    assert(hookNode.memoizedState, 'bar')
    hookNode = hooksNode.next
    assert(hookNode.memoizedState, 'baz')
  })

  return (
    <ChildComponent ref={childFiberRef} />
  )
}
```

从外部读取某一组件记忆的状态

下面我们来分类讨论 hook，首先从使用最广泛的开始 —— state hook：

State hook

你一定会觉得很吃惊：`useState` hook 在后台使用了 `useReducer`，并且它将 `useReducer` 作为预定义的 reducer（详见源码）。这意味着，`useState` 返回的结果实际上已经是 reducer 状态，同时也是一个 action dispatcher。请看，如下是 state hook 使用的 reducer 处理器：

```javascript 复制代码
function basicStateReducer(state, action) {
  return typeof action === 'function' ? action(state) : action;
}
```

state hook 的 reducer，又名基础状态 reducer。

所以正如你想象的那样，我们可以直接将新的状态传入 action dispatcher；但是你看到了吗？！我们也可以传入 **action 函数**给 dispatcher，**这个 action 函数可以接收旧的状态并返回新的。**（在本篇文章写就时，这种方法并没有记录在 React 官方文档中，很遗憾的是，它其实非常有用！）这意味着，当你向组件树发送状态设置器的时候，你可以修改父级组件的状态，同时不用将它作为另一个属性传入，例如：

```javascript 复制代码
const ParentComponent = () => {
  const [name, setName] = useState()

  return (
    <ChildComponent toUpperCase={setName} />
  )
}

const ChildComponent = (props) => {
  useEffect(() => {
    props.toUpperCase((state) => state.toUpperCase())
  }, [true])

  return null
}
```

根据旧状态返回新状态。

最后，effect hook —— 它对于组件的生命周期影响很大，那么它是如何工作的呢：

effect hook

effect hook 和其他 hook 的行为有一些区别，并且它有一个附加的逻辑层，这点我在后文将会解释。在我分析源码之前，首先我希望你牢记 effect hook 的一些属性：

- 它们在渲染时被创建，但是在浏览器绘制**后**运行。
- 如果给出了销毁指令，它们将在下一次绘制前被销毁。
- 它们会按照定义的顺序被运行。

> **注意，我使用了“绘制”而不是“渲染”。它们是不同的，在最近的 React 会议中，我看到很多发言者错误的使用了这两个词！甚至在官方 React 文档中，也有写“在渲染生效于屏幕之后”，其实这个过程更像是“绘制”。渲染函数只是创建了 fiber 节点，但是并没有绘制任何内容。**

于是就应该有另一个队列来保存这些 effect hook，并且还要能够在绘制后被定位到。通常来说，应该是 fiber 保存包含了 effect 节点的队列。每个 effect 节点都是一个不同的类型，并能在适当的状态下被定位到：

- 在修改之前调用 `getSnapshotBeforeUpdate()` 实例（详见源码）。
- 运行所有插入、更新、删除和 ref 的卸载（详见源码）。
- 运行所有生命周期函数和 ref 回调函数。生命周期函数会在一个独立的通道中运行，所以整个组件树中所有的替换、更新、删除都会被调用。这个过程还会触发任何特定于渲染器的初始 effect hook（详见源码）。
- `useEffect()` hook 调度的 effect —— 也被称为“被动 effect”，它基于这部分代码（也许我们要开始在 React 社区内使用这个术语了？！）。

hook effect 将会被保存在 fiber 一个称为 `updateQueue` 的属性上，每个 effect 节点都有如下的结构（详见源码）：

- `tag` —— 一个二进制数字，它控制了 effect 节点的行为（后文我将详细说明）。
- `create` —— 绘制**之后**运行的回调函数。
- `destroy` —— 它是 `create()` 返回的回调函数，将会在初始渲染**前**运行。
- `inputs` —— 一个集合，该集合中的值将会决定一个 effect 节点是否应该被销毁或者重新创建。
- `next` —— 它指向下一个定义在函数组件中的 effect 节点。

除了 `tag` 属性，其他的属性都很简明易懂。如果你对 hook 很了解，你应该知道，React 提供了一些特殊的 effect hook：比如 `useMutationEffect()` 和 `useLayoutEffect()`。这两个 effect hook 内部都使用了 `useEffect()`，实际上这就意味着它们创建了 effect hook，但是却使用了不同的 tag 属性值。

这个 tag 属性值是由二进制的值组合而成（详见源码）：

```javascript 复制代码
const NoEffect = /*              */ 0b00000000;
const UnmountSnapshot = /*       */ 0b00000010;
const UnmountMutation = /*       */ 0b00000100;
const MountMutation = /*         */ 0b00001000;
const UnmountLayout = /*         */ 0b00010000;
const MountLayout = /*           */ 0b00100000;
const MountPassive = /*          */ 0b01000000;
const UnmountPassive = /*        */ 0b10000000;
```

React 支持的 hook effect 类型

这些二进制值中最常用的情景是使用管道符号（`|`）连接，将比特相加到单个某值上。然后我们就可以使用符号（`&`）检查某个 tag 属性是否能触发一个特定的行为。如果结果是非零的，就表示可以。

```javascript 复制代码
const effectTag = MountPassive | UnmountPassive
assert(effectTag, 0b11000000)
assert(effectTag & MountPassive, 0b10000000)
```

如何使用 React 的二进制设计模式的示例

这里是 React 支持的 hook effect，以及它们的 tag 属性（详见源码）：

- Default effect—— `UnmountPassive | MountPassive`.
- Mutation effect—— `UnmountSnapshot | MountMutation`.
- Layout effect—— `UnmountMutation | MountLayout`.

以及这里是 React 如何检查行为触发的（详见源码）：

```javascript 复制代码
if ((effect.tag & unmountTag) !== NoHookEffect) {
  // Unmount
}
if ((effect.tag & mountTag) !== NoHookEffect) {
```

```
  // Mount
}
```

React 源码节选

所以，基于我们刚才学习的关于 effect hook 的知识，我们可以实际操作，从外部向 fiber 插入一些 effect：

javascript 复制代码

```
function injectEffect(fiber) {
  const lastEffect = fiber.updateQueue.lastEffect

  const destroyEffect = () => {
    console.log('on destroy')
  }

  const createEffect = () => {
    console.log('on create')

    return destroy
  }

  const injectedEffect = {
    tag: 0b11000000,
    next: lastEffect.next,
    create: createEffect,
    destroy: destroyEffect,
    inputs: [createEffect],
  }

  lastEffect.next = injectedEffect
}

const ParentComponent = (
  <ChildComponent ref={injectEffect} />
)
```

这就是 hooks 了！阅读本文你最大的收获是什么？你将如何把新学到的知识应用于 React 应用中？希望看到你留下有趣的评论！