

# 查探fiber的背景与动机与实现

---

## 查探fiber的背景与动机与实现

资源

课堂目标

知识点

fiber结构

实现fiber

```
window.requestIdleCallback(callback[, options])
```

实现fiber代码

Hook简介

没有破坏性改动

Hook解决了什么问题

在组件之间复用状态逻辑很难

复杂组件变得难以理解

难以理解的 class

如何调试源码

回顾

## 资源

---

1. 课堂代码课堂地址: <https://github.com/bubucuo/React-Training-Camp/tree/date-20201023>
2. [React Hook简介](#)
3. [React源码](#)

## 课堂目标

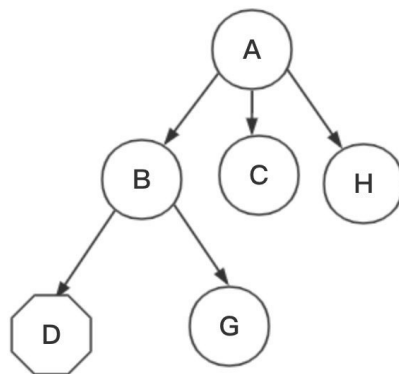
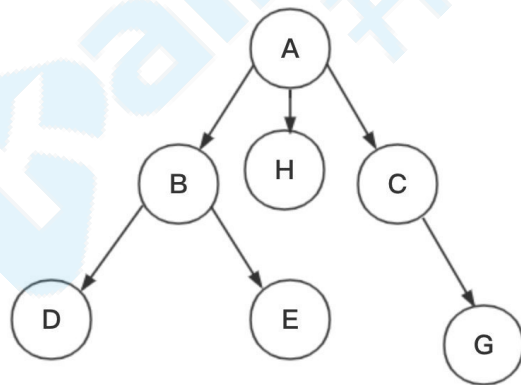
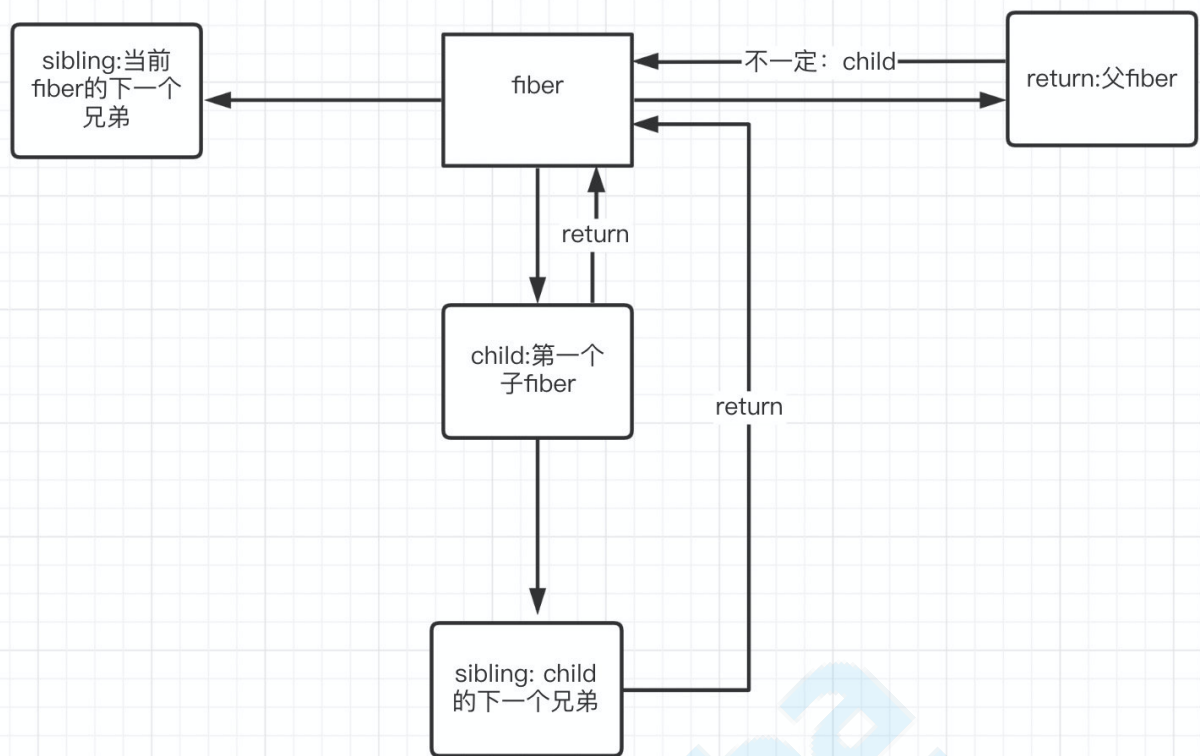
---

2. 理解fiber架构
3. 手写fiber架构的kreact
4. 理解hook解决的问题

## 知识点

---

### fiber结构



## 实现fiber

**`window.requestIdleCallback(callback[, options])`**

**`window.requestIdleCallback()`**方法将在浏览器的空闲时段内调用的函数排队。这使开发者能够在主事件循环上执行后台和低优先级工作，而不会影响延迟关键事件，如动画和输入响应。函数一般会按先进先调用的顺序执行，然而，如果回调函数指定了执行超时时间 `timeout`，则有可能为了在超时前执行函数而打乱执行顺序。

你可以在空闲回调函数中调用 `requestIdleCallback()`，以便在下一次通过事件循环之前调度另一个回调。

`callback`

一个在事件循环空闲时即将被调用的函数的引用。函数会接收到一个名为 `IdleDeadline` 的参数，这个参数可以获取当前空闲时间以及回调是否在超时时间前已经执行的状态。

`options` 可选

包括可选的配置参数。具有如下属性：

- `timeout`：如果指定了`timeout`并具有一个正值，并且尚未通过超时毫秒数调用回调，那么回调会在下一次空闲时期被强制执行，尽管这样很可能会对性能造成负面影响。

react中`requestIdleCallback`的hack在  
`react/packages/scheduler/src/forks/SchedulerHostConfig.default.js`。

## 实现fiber代码

Fiber 是 React 16 中新的协调引擎。它的主要目的是使 Virtual DOM 可以进行增量式渲染。

一个更新过程可能被打断，所以React Fiber一个更新过程被分为两个阶段(Phase)：第一个阶段 Reconciliation Phase和第二阶段Commit Phase。

```
import React from './kreact/';
import ReactDOM, {useState} from './kreact/react-dom';
import Component from './kreact/Component';

import './index.css';

class ClassComponent extends Component {
  static defaultProps = {
    color: 'pink'
  };
  render() {
    return (
      <div className="border">
        <div className={this.props.color}>color </div>
      </div>
    );
  }
}
```

```

        {this.props.name}
      </div>
    );
  }
}

function FunctionComponent({name}) {
  const [count, setCount] = useState(0);

  return (
    <div className="border">
      {name}
      <button onClick={() => setCount(count + 1)}> {count}: count add</button>
      <div className="border">
        {count % 2 ? (
          <button onClick={() => console.log("omg")}>click</button>
        ) : (
          <div>omg</div>
        )}
      </div>
    </div>
  );
}

const jsx = (
  <div className="box border">
    <p>全栈学习</p>
    <a href="https://zh-hans.reactjs.org/">React学习</a>
    <FunctionComponent name="函数组件" />
    <ClassComponent name="class组件" />

    <>
      <h1>文本1</h1>
      <h2>文本2</h2>
    </>

  </div>
);

ReactDOM.render(jsx, document.getElementById("root"));

```

./react-dom.js

```

import {TEXT} from "../const";

// 下一个单元任务 数据类型就是fiber
let nextUnitOfWork = null;

```

开课吧 web全栈架构师

```

// work in progress 正在工作当中的
// 正在工作当中的fiber root
let wipRoot = null;

// ! fiber数据结构
// type 标记fiber的类型
// key 标记当前层级下的唯一性
// props fiber属性
// base 上一次更新的fiber节点
// child 第一个子节点
// sibling 下一个兄弟节点
// return 父节点
// stateNode 真实dom节点
// !

// todo vnode、vnode 虚拟DOM节点
// todo node 真实DOM节点
function render(vnode, container) {
  // // 1. vnode->node
  // const node = createNode(vnode);
  // // 2. container.appendChild(node);
  // container.appendChild(node);

  wipRoot = {
    stateNode: container,
    props: {
      children: [vnode]
    }
  };

  nextUnitOfWork = wipRoot;
}

// 创建真实DOM节点
function createNode(vnode) {
  let node;

  const {type, props} = vnode;
  if (type === TEXT) {
    node = document.createTextNode("");
  } else if (typeof type === "string") {
    node = document.createElement(type);
  } else if (typeof type === "function") {
    node = type.prototype.isReactComponent
      ? updateClassComponent(vnode)
      : updateFunctionComponent(vnode);
  } else {
    // * 这个地方是个简写，源码当中没有生成Fragment，直接协调的子节点
    node = document.createDocumentFragment();
  }
}

```

```

    }
    // reconcileChildren(props.children, node);
    updateNode(node, props);
    return node;
}

function updateNode(node, nextVal) {
    Object.keys(nextVal)
        .filter(k => k !== "children")
        .forEach(k => {
            // node.setAttributes(k, nextVal[k]);
            node[k] = nextVal[k];
        });
}

// 返回真实dom节点
// 执行函数
function updateFunctionComponent(fiber) {
    const {type, props} = fiber;
    let children = [type(props)];
    reconcileChildren(fiber, children);
}

// 返回真实dom节点
// 先实例化, 再执行render函数
function updateClassComponent(fiber) {
    const {type, props} = fiber;
    let cmp = new type(props);
    let vnode = cmp.render();
    const children = [vnode];
    reconcileChildren(fiber, children);
}

// 原生标签节点更新
function updateHostComponent(fiber) {
    // 判断真实dom节点是否存在, 不存在的话, 去创建
    if (!fiber.stateNode) {
        fiber.stateNode = createNode(fiber);
    }

    const {children} = fiber.props;
    reconcileChildren(fiber, children);
    // console.log("fiber", fiber); //sy-log
}

function reconcileChildren(workInProgress, children) {
    // 记录上一个的哥哥fiber
    let prevSibling = null;
    for (let i = 0; i < children.length; i++) {

```

```

    let child = children[i];

    // ! 这里只是新增
    let newFiber = {
      type: child.type,
      props: child.props,
      stateNode: null,
      base: null,
      // child:
      // sibling:
      return: workInProgress
    };

    if (i === 0) {
      workInProgress.child = newFiber;
    } else {
      prevSibling.sibling = newFiber;
    }
    prevSibling = newFiber;
  }
}

function performUnitOfWork(fiber) {
  // 任务1: 执行更新fiber
  const {type} = fiber;
  if (typeof type === "function") {
    type.prototype.isReactComponent
      ? updateClassComponent(fiber)
      : updateFunctionComponent(fiber);
  } else {
    updateHostComponent(fiber);
  }

  // 任务2: 返回下一个要更新的fiber
  // 顺序是 子节点、兄弟、爸爸或者祖先的兄弟
  // 什么都没了, 就更新完成了
  if (fiber.child) {
    return fiber.child;
  }
  let nextFiber = fiber;
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling;
    }
    nextFiber = nextFiber.return;
  }
}

function workLoop(deadline) {

```

```

// 当前有需要更新的fiber, 并且浏览器有空闲时间
while (nextUnitOfWork && deadline.timeRemaining() > 1) {
  // 执行更新fiber, 并且返回下一个要更新的fiber
  nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
}

if (!nextUnitOfWork && wipRoot) {
  // 把fiber更新到根节点中, 其实就是把vnode->node
  commitRoot();
}

requestIdleCallback(workLoop);
}

requestIdleCallback(workLoop);

function commitRoot() {
  commitWorker(wipRoot.child);
  wipRoot = null;
}

function commitWorker(fiber) {
  if (!fiber) {
    return;
  }

  // ! 找到fiber.stateNode的父或者祖先DOM节点parentNode
  let parentNodeFiber = fiber.return;
  while (!parentNodeFiber.stateNode) {
    parentNodeFiber = parentNodeFiber.return;
  }
  let parentNode = parentNodeFiber.stateNode;

  // 新增
  if (fiber.stateNode) {
    parentNode.appendChild(fiber.stateNode);
  }

  commitWorker(fiber.child);
  commitWorker(fiber.sibling);
}

export default {render};

```

const.js



```
export const TEXT = "TEXT";

export const PLACEMENT = "PLACEMENT";
export const UPDATE = "UPDATE";
export const DELETION = "DELETION";
```

## Hook简介

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

1. Hooks是什么？为了拥抱正函数式
2. Hooks带来的变革，让函数组件有了状态和其他的React特性，可以替代class

```
import React, { useState } from 'react';

function Example() {
  // 声明一个新的叫做 "count" 的 state 变量
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

## 没有破坏性改动

在我们继续之前，请记住 Hook 是：

- **完全可选的。** 你无需重写任何已有代码就可以在一些组件中尝试 Hook。但是如果你不想，你不必现在就去学习或使用 Hook。
- **100% 向后兼容的。** Hook 不包含任何破坏性改动。
- **现在可用。** Hook 已发布于 v16.8.0。

没有计划从 React 中移除 class。

Hook 不会影响你对 React 概念的理解。恰恰相反，Hook 为已知的 React 概念提供了更直接的 API：props, state, context, refs 以及生命周期。稍后我们将看到，Hook 还提供了一种更强大的方式来组合他们。

# Hook解决了什么问题

Hook 解决了我们五年来编写和维护成千上万的组件时遇到的各种各样看起来不相关的问题。无论你正在学习 React，或每天使用，或者更愿尝试另一个和 React 有相似组件模型的框架，你都可能对这些问题似曾相识。

## 在组件之间复用状态逻辑很难

React 没有提供将可复用性行为“附加”到组件的途径（例如，把组件连接到 store）。如果你使用过 React 一段时间，你也许会熟悉一些解决此类问题的方案，比如 [render props](#) 和 [高阶组件](#)。但是这类方案需要重新组织你的组件结构，这可能会很麻烦，使你的代码难以理解。如果你在 React DevTools 中观察过 React 应用，你会发现由 providers，consumers，高阶组件，render props 等其他抽象层组成的组件会形成“嵌套地狱”。尽管我们可以在 [DevTools 过滤掉它们](#)，但这说明了一个更深层次的问题：React 需要为共享状态逻辑提供更好的原生途径。

你可以使用 Hook 从组件中提取状态逻辑，使得这些逻辑可以单独测试并复用。**Hook 使你在无需修改组件结构的情况下复用状态逻辑**。这使得在组件间或社区内共享 Hook 变得更加便捷。

具体将在[自定义 Hook](#) 中对此展开更多讨论。

## 复杂组件变得难以理解

我们经常维护一些组件，组件起初很简单，但是逐渐会被状态逻辑和副作用充斥。每个生命周期常常包含一些不相关的逻辑。例如，组件常常在 `componentDidMount` 和 `componentDidUpdate` 中获取数据。但是，同一个 `componentDidMount` 中可能也包含很多其它的逻辑，如设置事件监听，而之后需在 `componentWillUnmount` 中清除。相互关联且需要对照修改的代码被进行了拆分，而完全不相关的代码却在同一个方法中组合在一起。如此很容易产生 bug，并且导致逻辑不一致。

在多数情况下，不可能将组件拆分为更小的粒度，因为状态逻辑无处不在。这也给测试带来了一定挑战。同时，这也是很多人将 React 与状态管理库结合使用的原因之一。但是，这往往会引入了很多抽象概念，需要你在不同的文件之间来回切换，使得复用变得更加困难。

为了解决这个问题，**Hook 将组件中相互关联的部分拆分成更小的函数（比如设置订阅或请求数据），而并非强制按照生命周期划分**。你还可以使用 reducer 来管理组件的内部状态，使其更加可预测。

我们将在[使用 Effect Hook](#) 中对此展开更多讨论。

## 难以理解的 class

除了代码复用和代码管理会遇到困难外，我们还发现 class 是学习 React 的一大屏障。你必须去理解 JavaScript 中 `this` 的工作方式，这与其他语言存在巨大差异。还不能忘记绑定事件处理器。没有稳定的[语法提案](#)，这些代码非常冗余。大家可以很好地理解 props，state 和自顶向下的数据流，但对 class 却一筹莫展。即便在有经验的 React 开发者之间，对于函数组件与 class 组件的差异也存在分歧，甚至还要区分两种组件的使用场景。

另外，React 已经发布五年了，我们希望它能在下一个五年也与时俱进。就像 [Svelte](#)，[Angular](#)，[Glimmer](#) 等其它的库展示的那样，组件[预编译](#)会带来巨大的潜力。尤其是在它不局限于模板的时候。最近，我们一直在使用 [Prepack](#) 来试验 [component folding](#)，也取得了初步成效。但是我们发现使用 class 组件会无意中鼓励开发者使用一些让优化措施无效的方案。class 也给目前的工具带来了一些问题。例如，class 不能很好的压缩，并且会使热重载出现不稳定的情况。因此，我们想提供一个使代码更易于优化的 API。

为了解决这些问题，**Hook** 使你在非 class 的情况下可以使用更多的 **React** 特性。从概念上讲，React 组件一直更像是函数。而 Hook 则拥抱了函数，同时也没有牺牲 React 的精神原则。Hook 提供了问题的解决方案，无需学习复杂的函数式或响应式编程技术。

## 如何调试源码

步骤如下：

```
clone文件: git clone https://github.com/bubucuo/React-Training-Camp.git
安装: yarn
启动: yarn start
```

方便查看逻辑，去webpack里把dev设置为false，参考上面的git地址。

## 回顾

### 查探fiber的背景与动机与实现

资源

课堂目标

知识点

fiber结构

实现fiber

```
window.requestIdleCallback(callback[, options])
```

实现fiber代码

Hook简介

没有破坏性改动

Hook解决了什么问题

在组件之间复用状态逻辑很难

复杂组件变得难以理解

难以理解的 class

如何调试源码

回顾

