查探React Fiber的背景与具体实现

查探React Fiber的背景与具体实现
- 资源
- 课堂目标
- 知识点
  - fiber
    - 为什么需要fiber
    - 什么是fiber
  - 实现fiber
    - `window.requestIdleCallback(callback[, options])`
    - 实现fiber
  - 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
- 回顾

资源

2. React中文网
3. React源码
4. fiber结构图：https://www.processon.com/view/link/5dea52b9e4b079080a22a846

课堂目标

4. 了解fiber架构
5. 深入掌握React工作原理
6. 掌握fiber

知识点

fiber

为什么需要fiber

[React Conf 2017 Fiber介绍视频]

React的killer feature： virtual dom

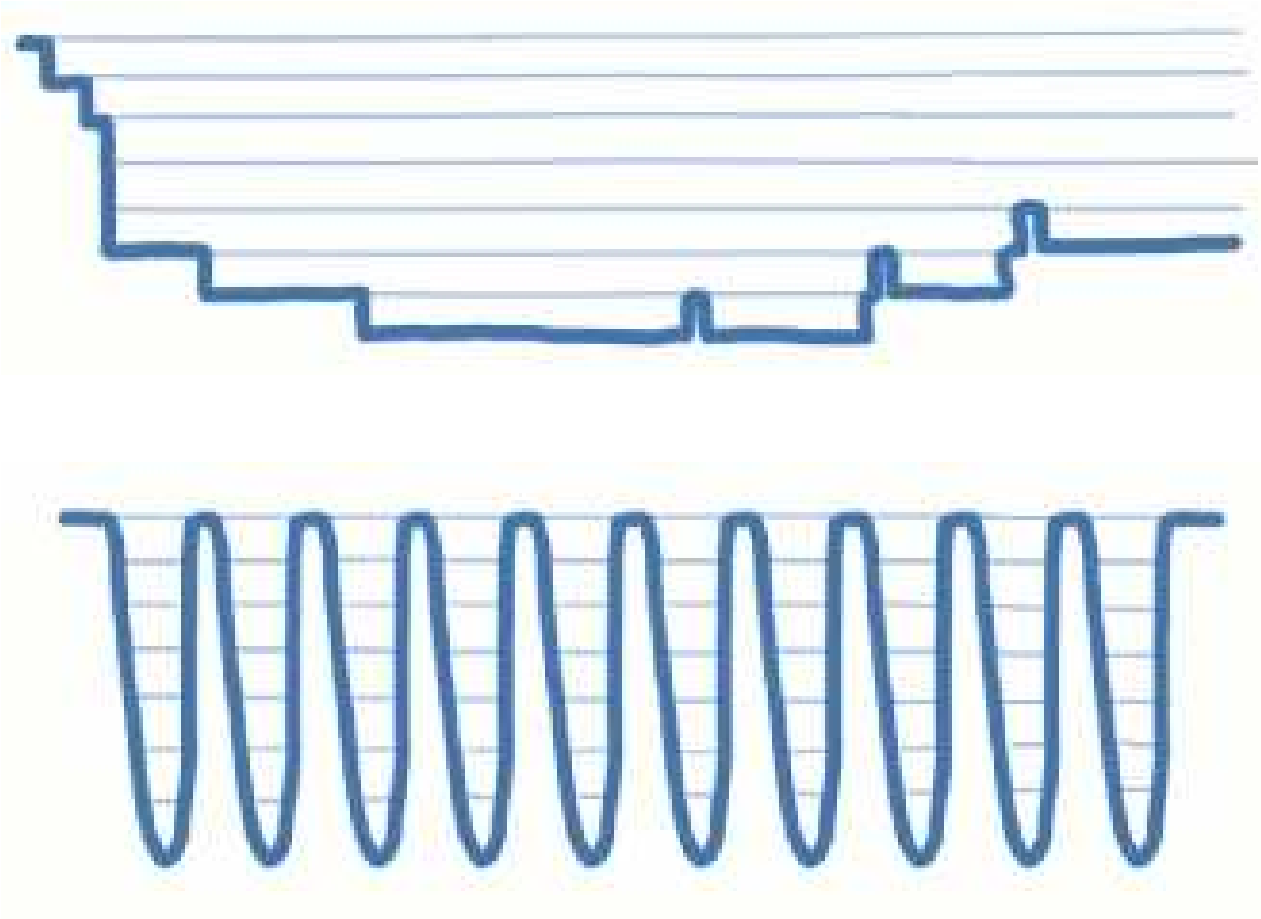
1. 为什么需要fiber

   对于大型项目，组件树会很大，这个时候递归遍历的成本就会很高，会造成主线程被持续占用，结果就是主线程上的布局、动画等周期性任务就无法立即得到处理，造成视觉上的卡顿，影响用户体验。

2. 任务分解的意义

   解决上面的问题

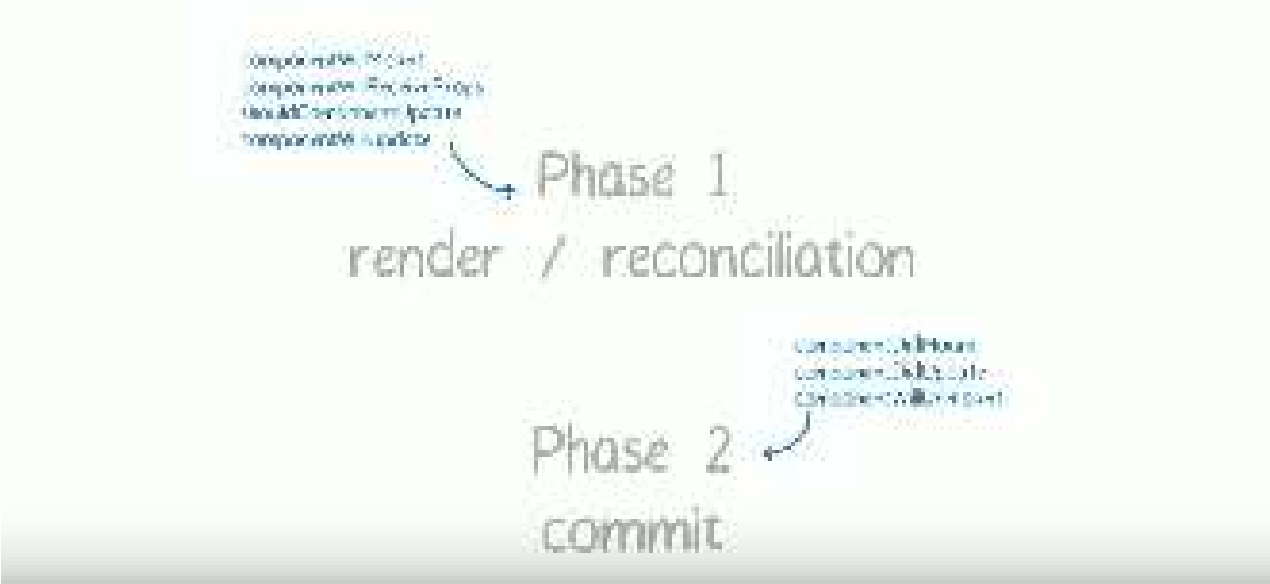
3. 增量渲染（把渲染任务拆分成块，匀到多帧）
4. 更新时能够暂停，终止，复用渲染任务
5. 给不同类型的更新赋予**优先级**
6. 并发方面新的基础能力
7. **更流畅**



什么是fiber

A Fiber is work on a Component that needs to be done or was done. There can be more than one per component.

fiber是指组件上将要完成或者已经完成的任务，每个组件可以一个或者多个。



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

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 ReactDOM from "react-dom";
import ReactDOM from "./kreact/react-dom";
import Component from "./kreact/Component";
import "./index.css";

function FunctionComponent(props) {
  return (
    <div className="border">
      <p>{props.name}</p>
    </div>
  );
}

class ClassComponent extends Component {
  render() {
    return (
      <div className="border">
        <p>{this.props.name}</p>
      </div>
    );
  }
}

function FragmentComponent(props) {
  return (
    <>
      <h1>111</h1>
      <h1>222</h1>
    </>
  );
}

const jsx = (
  <section className="border">
    <h1>慢慢慢</h1>
    <h1>全栈</h1>
    <a href="https://www.kaikeba.com/">kkb</a>
    <FunctionComponent name="函数组件" />
    <ClassComponent name="类组件" />
    <FragmentComponent />
```

```
  </section>
);

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

./react-dom.js

```
// ! 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
// !

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

function render(vnode, container) {
  wipRoot = {
    type: "div",
    props: {children: {...vnode}},
    stateNode: container
  };
  nextUnitOfWork = wipRoot;
}

function isStringOrNumber(sth) {
  return typeof sth === "string" || typeof sth === "number";
}

function createNode(workInProgress) {
  const {type, props} = workInProgress;

  const node = document.createElement(type);
  updateNode(node, props);

  return node;
}

function updateNode(node, props) {
  Object.keys(props).forEach(k => {
    if (k === "children") {
```

```
      if (isStringOrNumber(props[k])) {
        node.textContent = props[k];
      }
    } else {
      node[k] = props[k];
    }
  });
}

function updateHostComponent(workInProgress) {
  if (!workInProgress.stateNode) {
    workInProgress.stateNode = createNode(workInProgress);
  }

  reconcileChildren(workInProgress, workInProgress.props.children);

  console.log("work", workInProgress); //sy-log
}

// 函数组件
// 执行函数
function updateFunctionComponent(workInProgress) {
  const {type, props} = workInProgress;
  const child = type(props);
  reconcileChildren(workInProgress, child);
}

// 类组件
// 先实例化 再执行render函数
function updateClassComponent(workInProgress) {
  const {type, props} = workInProgress;
  const instance = new type(props);
  const child = instance.render();
  reconcileChildren(workInProgress, child);
}

function updateTextComponent(vnode) {
  const node = document.createTextNode(vnode);
  return node;
}

function updateFragmentComponent(workInProgress) {
  reconcileChildren(workInProgress, workInProgress.props.children);
}

function reconcileChildren(workInProgress, children) {
  if (isStringOrNumber(children)) {
    return;
  }
```

```
  const newChildren = Array.isArray(children) ? children : [children];
  let previousNewFiber = null;
  for (let i = 0; i < newChildren.length; i++) {
    let child = newChildren[i];

    let newFiber = {
      type: child.type,
      props: {...child.props},
      stateNode: null,
      child: null,
      sibling: null,
      return: workInProgress
    };

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

let nextUnitOfWork = null;

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

  // 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 (workInProgress.child) {
    return workInProgress.child;
  }
  let nextFiber = workInProgress;
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling;
```

```
    }
    nextFiber = nextFiber.return;
  }
}

function workLoop(IdleDeadline) {
  while (nextUnitOfWork && IdleDeadline.timeRemaining() > 1) {
    nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
  }
  // commit

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

requestIdleCallback(workLoop);

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

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

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

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

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

export default {render};
```

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~~王能量~~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。

回顾

查探React Fiber的背景与具体实现

- 资源
- 课堂目标
- 知识点
  - fiber
    - 为什么需要fiber
    - 什么是fiber
  - 实现fiber
    - `window.requestIdleCallback(callback[, options])`
    - 实现fiber
  - 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
- 回顾