React原理解析03

下节课内容

课堂主题

深入掌握React

资源

1. React源码概览
2. React源码文件指引
3. Debug React源码

课堂目标

1. 掌握源码debug方式
2. 掌握fiber、hooks原理
3. 掌握组件常用优化方法

知识点

如何调试源码

步骤如下：

```
clone文件：git clone
https://github.com/bubucuo/DebugReact.git
安装：  npm install
启动：  npm start
```

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

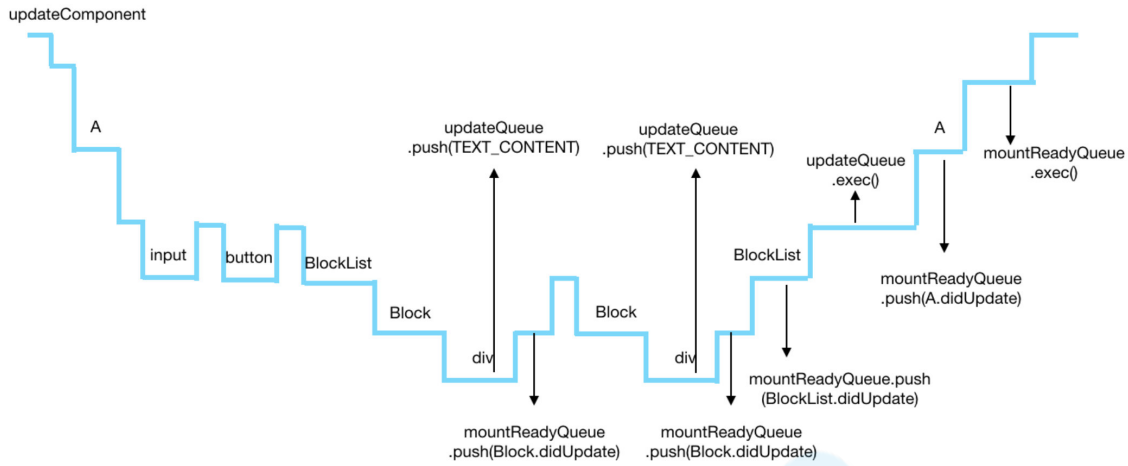
   对于大型项目，组件树会很大，这个时候递归遍历的成本就会很高，会造成主线程被持续占用，结果就是主线程上的布局、动画等周期性任务就无法立即得到处理，造成视觉上的卡顿，影响用户体验。

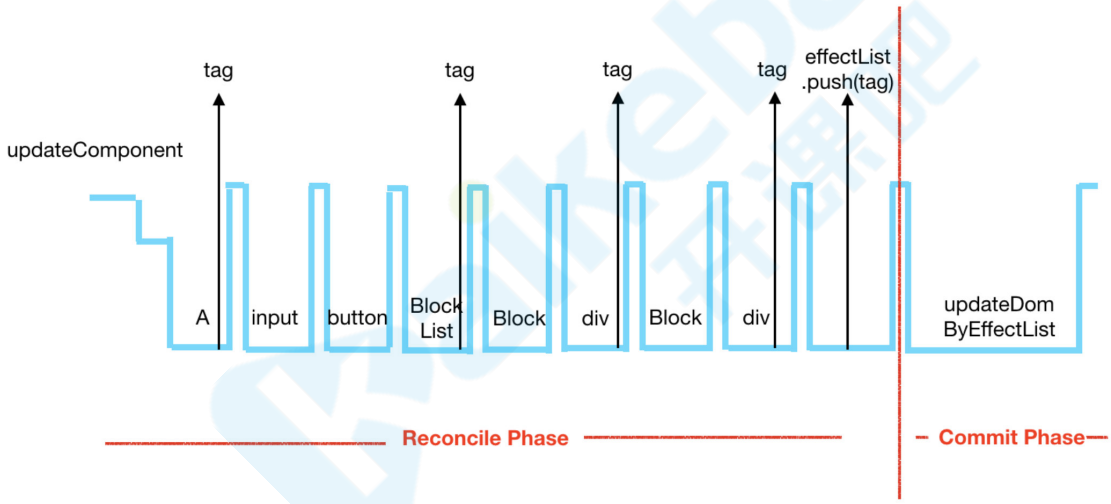
2. 任务分解的意义

   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。

fiber数据结构

查看react/packages/react-reconciler/src/ReactFiber.js - Fiber

tag: WorkTag

标记fiber的类型，类似我们之前写react核心api时候的vtype

WorkTag地址：/react/packages/shared/ReactWorkTags.js

key: null | string

当前子节点的唯一标识，ReactElement里面的key。

elementType: any,

element.type的值，保存的是协调期间当前child的身份。也就是我们调用`createElement`的第一个参数。

type: any

当前fiber关联的function或者class。

alternate: Fiber | null

fiber的一个合成版本。每一个更新过的fiber都会有一个alternate。

fiber的alternate是使用一个叫做`cloneFiber`的函数懒创建的(created lazily)。意思就是， 如果这个fiber的alternate存在，那么`cloneFiber`不会去创建一个新的对象，而是复用这个fiber的alternate，从而最大程度地减少任务分配(allocations)。

stateNode: any,

当前fiber的本地状态。

child: Fiber | null,

第一个子节点， 单向链表结构

sibling: Fiber | null,

下一个兄弟节点， 单向链表结构。兄弟节点的return指向的是同一个父节点。

index

下标

return: Fiber | null

是指程序(program)在处理完当前fiber之后应当返回的fiber，它也可被认为是一个parent fiber。

pendingProps 与 memoizedProps

概念上来说，props是指函数的参数。一个fiber的`pendingProps`被赋值在这个fiber执行(execution)的开始，`memoizedProps`则是在执行完成时被赋值。

当新传入的`pendingProps`等于`memoizedProps`的时候，这表示这个fiber 的上一个输出(output)可以被复用，从而避免了不必要的任务。

updateQueue: UpdateQueue | null,

用于state更新和回调的队列。

memoizedState: any

上一次渲染时候的state。

fiber reconciler

“fiber” reconciler 是一个新尝试，致力于解决 stack reconciler 中固有的问题，同时解决一些历史遗留问题。Fiber 从 React 16 开始变成了默认的 reconciler。

它的主要目标是：

- 能够把可中断的任务切片处理。
- 能够调整优先级，重置并复用任务。
- 能够在父元素与子元素之间交错处理，以支持 React 中的布局。
- 能够在 `render()` 中返回多个元素。
- 更好地支持错误边界。

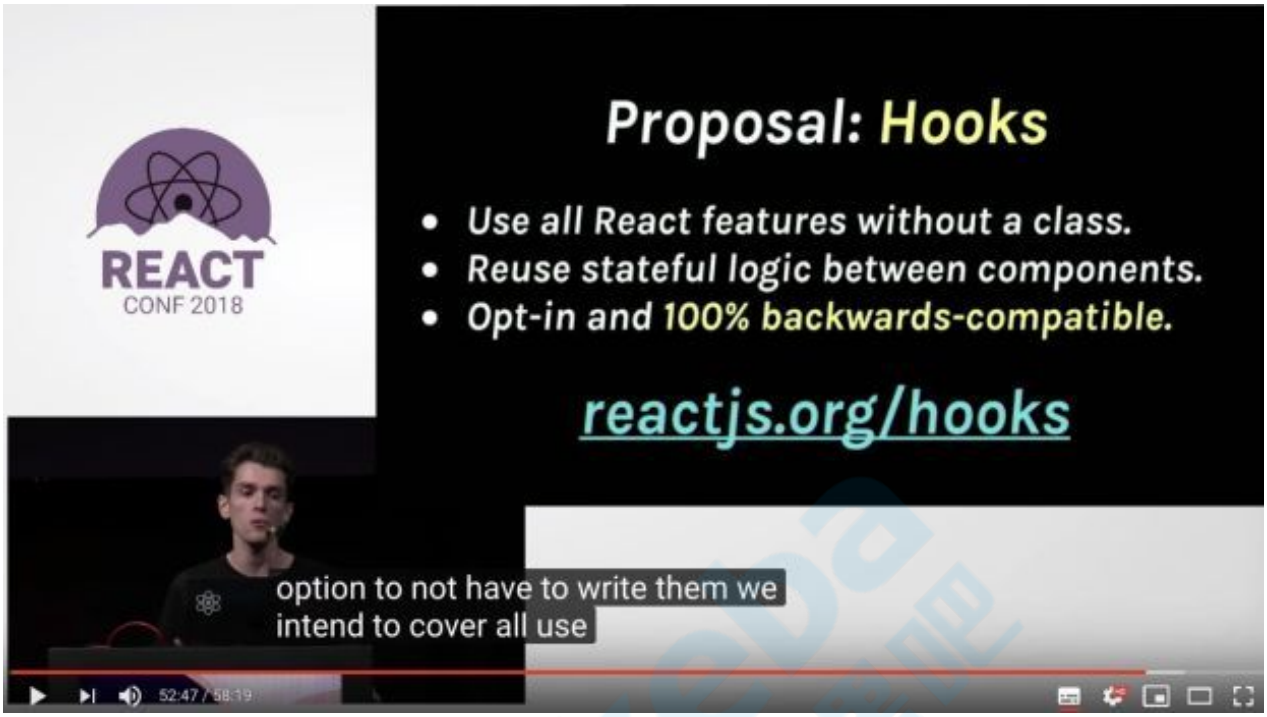
你可以在这里和这里，深入了解 React Fiber 架构。

源代码在 `packages/react-reconciler` 目录下。

Hooks

官网

hooks介绍视频：React Today and Tomorrow

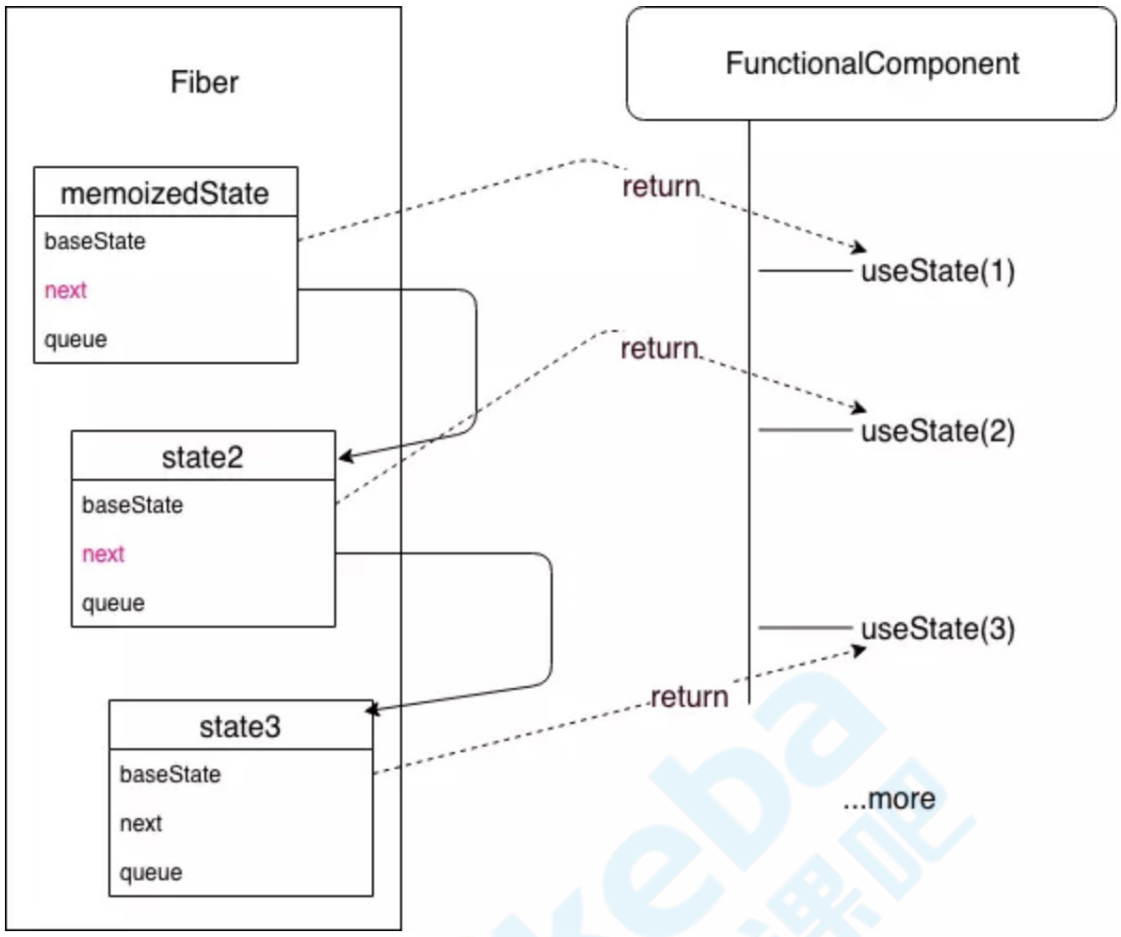


1. Hooks是什么？为了拥抱~~正能量~~函数式
2. Hooks带来的变革，让函数组件有了状态，可以替代class
3. 类似链表的实现原理

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

function FunComp(props) {
  const [data, setData] =
useState('initialState')

  function handleChange(e) {
    setData(e.target.value)
  }
```

```
  useEffect(() => {
    //subscribeToSomething()
    return () => {
      //unSubscribeToSomething()
    }
  })

  return (
    <input value={data} onChange={handleChange}
/>
  )
}
```

```
function FunctionalComponent () {
  const [state1, setState1] = useState(1)
  const [state2, setState2] = useState(2)
  const [state3, setState3] = useState(3)
}

hook1 => Fiber.memoizedState
state1 === hook1.memoizedState
hook1.next => hook2
state2 === hook2.memoizedState
hook2.next => hook3
state3 === hook2.memoizedState
```



事件系统

React 实现一个合成事件，这与渲染器无关，它适用于 React DOM 和 React Native。

深入研究事件系统代码的视频

树形组件设计与实现

设计思路

递归：自己调用自己

如计算f(n)=f(n-1)*n; n>0, f(1)=1

```
function foo(n) {
  return n===1 ? 1 : n*foo(n-1)
}
```

react中实现递归组件更加纯粹，就是组件递归渲染即可。假设我们的节点组件是TreeNode，它的render中只要发现当前节点拥有子节点就要继续渲染自己。节点的打开状态可以通过给组件一个open状态来维护。

实现

//TreeNode.js

```
import React, { Component } from "react";
import TreeNode from
"../../components/TreeNode";
//数据源
const treeData = {
  key: 0, //标识唯一性
  title: "全国", //节点名称显示
  children: [
    //子节点数组
    {
      key: 6,
      title: "北方区域",
      children: [
```

```
      {
        key: 1,
        title: "黑龙江省",
        children: [
          {
            key: 6,
            title: "哈尔滨",
          },
        ],
      },
      {
        key: 2,
        title: "北京",
      },
    ],
  },
  {
    key: 3,
    title: "南方区域",
    children: [
      {
        key: 4,
        title: "上海",
      },
      {
        key: 5,
        title: "深圳",
      },
    ],
```

```
    },
  ],
};
export default class TreePage extends Component
{
  render() {
    return (
      <div>
        <h1>TreePage</h1>
        <TreeNode data={treeData} />
      </div>
    );
  }
}
```

TreeNode.js

```
import React, { Component } from "react";
import classnames from "classnames";//先安装下
npm install classnames

export default class TreeNode extends Component
{
  constructor(props) {
    super(props);
    this.state = {
      expanded: false,
    };
```

```
  }
  handleExpanded = () => {
    this.setState({
      expanded: !this.state.expanded,
    });
  };
  render() {
    const { title, children } =
this.props.data;
    const { expanded } = this.state;
    const hasChildren = children &&
children.length > 0;
    return (
      <div>
        <div className="nodeInner" onClick=
{this.handleExpanded}>
          {hasChildren && (
            <i
              className={classnames("tri",
expanded ? "tri-open" : "tri-close")}
             ></i>
          )}
          <span>{title}</span>
        </div>
        {expanded && hasChildren && (
          <div className="children">
            {children.map(item => {
              return <TreeNode key={item.key}
data={item} />;
```

```
            })}
          </div>
        )}
      </div>
    );
  }
}
```

```
/* 树组件css */
.nodeInner {
  cursor: pointer;
}

.children {
  margin-left: 20px;
}

.tri {
  width: 20px;
  height: 20px;
  margin-right: 2px;
  padding-right: 4px;
}

.tri-close:after,
.tri-open:after {
  content: "";
  display: inline-block;
  width: 0;
```

```
  height: 0;
  border-top: 6px solid transparent;
  border-left: 8px solid black;
  border-bottom: 6px solid transparent;
}

.tri-open:after {
  transform: rotate(90deg);
}
```

常见组件优化技术

核心：减少不必要渲染，只渲染需要被渲染的组件。

定制组件的shouldComponentUpdate钩子

范例：通过shouldComponentUpdate优化组件

```
import React, { Component } from "react";

export default class CommentListPage extends
Component {
  constructor(props) {
    super(props);
    this.state = {
      commentList: [],
```

```
    };
  }
  componentDidMount() {
    this.timer = setInterval(() => {
      this.setState({
        commentList: [
          {
            id: 0,
            author: "小明",
            body: "这是小明写的文章",
          },
          {
            id: 1,
            author: "小红",
            body: "这是小红写的文章",
          },
        ],
      });
    }, 1000);
  }
  render() {
    const { commentList } = this.state;
    return (
      <div>
        <h1>CommentListPage</h1>
        {commentList.map(item => {
          return <Comment key={item.id} data=
{item} />;
        })}
```

```
      </div>
    );
  }
}

class Comment extends Component {
  shouldComponentUpdate(nextProps, nextState) {
    const { author, body } = this.props.data;
    const { author: newAuthor, body: newBody }
= nextProps.data;
    if (author === newAuthor && body ===
newBody) {
      return false; //如果不执行这里，将会多次
render
    }
    return true;
  }
  render() {
    const { author, body } = this.props.data;
    return (
      <div className="border">
        <p>{author}</p>
        <p>{body}</p>
      </div>
    );
  }
}
```

PureComponent

定制了shouldComponentUpdate后的Component

```
import React, { Component, PureComponent } from "react";

export default class PureComponentPage extends PureComponent {
  constructor(props) {
    super(props);
    this.state = {
      counter: 0,
      // obj: {
      //   num: 2,
      // },
    };
  }

  setCounter = () => {
    this.setState({
      counter: 100,
      // obj: {
      //   num: 200,
      // },
    });
  };

  render() {
```

```
    const { counter, obj } = this.state;
    console.log("render");
    return (
      <div>
        <h1>PuerComponentPage</h1>
        <div onClick={this.setCounter}>counter:
{counter}</div>
      </div>
    );
  }
}
```

缺点是必须要用class形式，而且要注意是**浅比较**,

路径：/react/packages/shared/shallowEqual.js

```
/**
 * Performs equality by iterating through keys on an object and returning false
 * when any key has values which are not strictly equal between the arguments.
 * Returns true when the values of all keys are strictly equal.
 */
function shallowEqual(objA: mixed, objB: mixed): boolean {
  if (Object.is(objA, objB)) {
    return true;
  }
  if (
    typeof objA !== 'object' ||
    objA === null ||
    typeof objB !== 'object' ||
    objB === null
  ) {
    return false;
  }
  const keysA = Object.keys(objA);
  const keysB = Object.keys(objB);
  if (keysA.length !== keysB.length) {
    return false;
  }
  for (let i = 0; i < keysA.length; i++) {
    if (
      !hasOwnProperty.call(objB, keysA[i]) ||
      !Object.is(objA[keysA[i]], objB[keysA[i]])
    ) {
      return false;
    }
  }
  return true;
}
```

React.memo

`React.memo(...)`是React v16.6引进来的新属性，是一个高阶组件。它的作用和`React.PureComponent`类似，是用来控制函数组件的重新渲染的。`React.memo(...)` 其实就是函数组件的`React.PureComponent`。

```jsx
import React, { Component, memo } from "react";

export default class ReactMemoPage extends
Component {
  constructor(props) {
    super(props);
    this.state = {
      date: new Date(),
      counter: 0,
    };
  }
  componentDidMount() {
    this.timer = setInterval(() => {
      this.setState({
        date: new Date(),
        //counter: this.state.counter + 1,
      });
    }, 1000);
  }
  componentWillUnmount() {
    clearInterval(this.timer);
  }
  render() {
    const { counter, date } = this.state;
    console.log("render", counter);
    return (
      <div>
        <h1>ReactMemoPage</h1>
        <p>{date.toLocaleTimeString()}</p>
```

```
        <MemoCounter counter={counter} />
      </div>
    );
  }
}

const MemoCounter = memo(props => {
  console.log("MemoCounter");
  return <div>{props.counter}</div>;
});
```

useMemo

把“创建”函数和依赖项数组作为参数传入 `useMemo`，它仅会在某个依赖项改变时才重新计算 memoized 值。这种优化有助于避免在每次渲染时都进行高开销的计算。

```
import React, { useState, useMemo } from
"react";

export default function ReactMemoPage(props) {
  const [counter, setCounter] = useState(0);
  const [val, setValue] = useState("");

  const expensive = useMemo(() => {
    console.log("compute");
    return counter * 100;
```

```
  }, [counter]);

  // const expensive = () => {
  //   console.log("compute");
  //   return counter * 100;
  // };

  return (
    <div>
      <p>{expensive}</p>
      <div>
        <button onClick={() =>
setCounter(counter + 1)}>counter</button>
        <input value={val} onChange={event =>
setValue(event.target.value)} />
      </div>
    </div>
  );
}
```

useCallback

把内联回调函数及依赖项数组作为参数传入 `useCallback`，它将返回该回调函数的 memoized 版本，该回调函数仅在某个依赖项改变时才会更新。当你把回调函数传递给经过优化的并使用引用相等性去避免非必要渲染（例如 `shouldComponentUpdate`）的子组件时，它将非常有用。

```
import React, { useState, useCallback } from
"react";

export default function ReactMemoPage() {
  const [val, setVal] = useState("");
  const [counter, setCounter] = useState(0);

  const addClick = useCallback(() => {
    setCounter(counter + 1);
  }, [counter]);

  // const addClick = () => {
  //   setCounter(counter + 1);
  // };
  console.log("----");
  return (
    <div>
      <input value={val} onChange={event =>
setVal(event.target.value)} />
      <button onClick={addClick}>{counter}
</button>
    </div>
  );
}
```

`useCallback(fn, deps)` 相当于 `useMemo(() => fn, deps)`。

> 注意

> 依赖项数组不会作为参数传给“创建”函数。虽然从概念上来说它表现为：所有“创建”函数中引用的值都应该出现在依赖项数组中。未来编译器会更加智能，届时自动创建数组将成为可能。

回顾

下节课内容

Redux-saga、项目实战（umi）