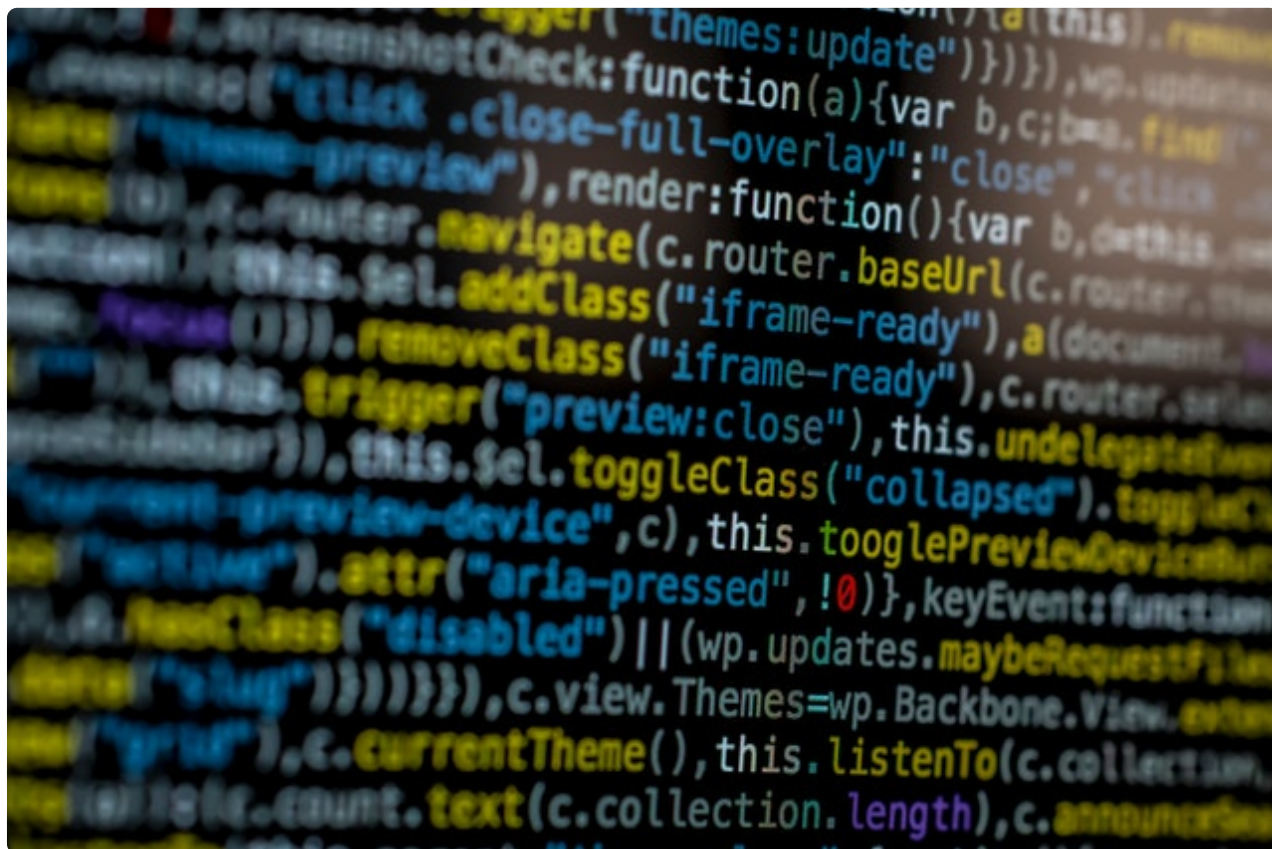


20 React 应用程序首次渲染时内部运行流程概述

更新时间：2020-09-11 09:56:11



“读书给人以快乐、给人以光彩、给人以才干。——培根”

前言

前面几章先后介绍了如何理解 **React** 世界的基础理论体系，包括如何深入理解组件与元素的设计理念，**React Fiber** 架构的本质以及 **React** 任务调度体系。本章的主要内容是基于这些基础理论体系探索 **React** 应用程序首次渲染时其内部运行流程。

在第三章第三节已经提到 **React** 应用程序首次渲染流程的阶段划分，见下图 5.1.1。

React 应用程序首次渲染时内部运行流程

prerender 阶段：基建工作—构建 fiberRoot 对象

render 阶段1：解析工作单元，构建 workInProgress 对象树（将元素转化为 Fiber 结点）

render 阶段2：完成工作单元解析，收集副作用

commit 阶段：将副作用信息更新到屏幕

图 5.1.1 React 应用程序首次渲染流程

我们将 React 应用程序首次渲染流程分为三个阶段，分别是 **prerender**、**render** 和 **commit**。本章将依次介绍应用程序首次渲染时在 React 这三个阶段所做的工作。

注：官方对 React 应用程序运行阶段的划分并没有给出明确的界限。为了方便理解我们将应用程序的整个渲染过程做了拆分。但是这种拆分不希望给读者产生固定的思维模式，建议通过实际的运行调试来分析其整个渲染过程。

以一个 Demo 为研究主线

我们设计了一个点击计数的简单应用，在这个应用中通过点击按钮实现统计按钮点击次数的功能，见代码示例 5.1.1。本章以及第六章将都以该应用为研究基础，研究应用程序的首次渲染和更新渲染时其内部执行机制。

```
class UpdateCounter extends Component{
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
      text: '点击计数'
    };
    // 知道这里为什么要用bind吗？
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({
      count: this.state.count + 1
    });
  }

  render() {
    return (
      <div className="wrap-box">
        <button key="1" onClick={this.handleClick}>{this.state.text}</button>
        <span key="2" id="spanText" className="span-text">{this.state.count}</span>
      </div>
    )
  }
}
```

代码示例 5.1.1 描述的是点击计数的组件 `UpdateCounter`，通过执行 `ReactDOM.render(<UpdateCounter name="Taylor" />, container)` 组件被渲染到屏幕，其最终效果见下图 5.1.2。

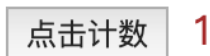


图 5.1.2 点击计数应用页面效果

`container` 指向了一个 DOM 元素，不需要我们深入的研究。我们需要仔细分析一下 `<UpdateCounter name="Taylor" />`，它是基于 JSX 语法的「元素」，当程序开始执行后，它在内存中的状态是什么样的呢？见代码示例 5.1.2。

```
{
  $$typeof: Symbol(react.element),
  key: null,
  props: {name: "Taylor"},
  ref: null,
  // type 指向了组件的构造函数
  type: f UpdateCounter(props),
  // ...
}
```

代码示例 5.1.2 点击计数组件对应的元素

现在我们想一想，**React** 如何才能将上面这个元素的内容映射到屏幕上面呢？

class 组件元素的两个重要属性 `props` 和 `type`

`props` 属性存储了外部传入组件内部的一些数据，光有数据是不行的，因为页面需要真实的 DOM 结构。那么，怎么生成页面的结构呢？

`type` 属性指向了组件的构造函数，**React** 可以通过执行 `new element.type()` 获得组件的实例 `instance`，然后通过执行 `instance.render()` 获得该组件内部的元素结构，见代码示例 5.1.3。

```

{
  $$typeof: Symbol(react.element),
  key: null,
  ref: null,
  type: "div",
  props: {
    className: "wrap-box",
    children: [
      {
        $$typeof: Symbol(react.element),
        key: "1",
        props: {children: "点击计数", onClick: f},
        ref: null,
        type: "button",
        // ...
      },
      {
        $$typeof: Symbol(react.element),
        key: "2",
        props: {id: "spanText", className: "span-text", children: 0},
        ref: null,
        type: "span",
        // ...
      }
    ],
  },
  // ...
}

```

代码示例 5.1.2 点击计数组件内部返回的元素

这些元素的结构描述了真实的 DOM 结构，但是仅仅知道元素的结构是远远不够的，**React** 会将每一个元素结点都转换成 **Fiber** 结点，在后面文章会有详细介绍哦。

小结

本节简单介绍了用于研究 **React** 应用程序运行时的 **Demo**，包括 **Demo** 的代码示例以及在运行时其自身的元素结构和内部返回的元素结构，这些元素结构最终会映射成真实的 DOM。在下一节将会介绍 **React** 应用程序首次渲染过程中在 **prerender** 阶段所做的工作。

}

