React 核心概念_1

1. jsx

A. 为什么使用 jsx，与 vue 的 template 相比有什么好处吗

- template: 模版语法，直观，有大量的内置指令来降低 js 使用成本，但是灵活度低
- jsx: 灵活，逻辑性强

> 组件可以分为两种：视图层组件或者是逻辑层组件，template 语法更符合前者的设计理念，而 jsx 符合后者的设计理念，vue 官方认为前者组件用的比较多，所以推崇 template

B. jsx 设计理念

React 创建的元素分为两类：**Dom元素和组件元素**

我们知道 web 网页是由一个又一个 html 标签嵌套而形成的，当我们想用 js 去表达元素的时候，第一反应就是 json 格式（天然可以用来表示 html 标签嵌套）

例如：

JavaScript

```
{
  type: 'div',
  props: {
    className: 'div-demo',
    children: [{
      type: 'button',
      props: {
        children: '点击按钮就送屠龙宝刀'
      }
    }]
  }
}
```

就类似于下面的 html 标签

HTML

```
<div class="div-demo">
  <button>点击按钮就送屠龙宝刀</button>
</div>
```

而这种方式天然地支持组件封装：

JavaScript

```
  const Button = ({type, text}) => {
    return {
      type: 'button',
      props: {
        className: `btn-{type}`,
        children: text,
      }
    }
  }
```

> 不过当一个组件有很多嵌套子组件的时候，可能这种方式写起来就比较复杂了，所以类 html 的 jsx 完美地解决了这一点：

JavaScript

```
// 这样写组件他不香吗
export default () => {
  return (
    <div>
      <button></button>
      <span></span>
    </div>
  )
}
```

C. jsx 一些概念

首先明确一点，React 不强制要求使用 jsx，**jsx 其实是一种语法糖，每个 JSX 元素只是调用 React.createElement(component, props, ...children) 配合 babel 的语法糖**。因此，使用 JSX 可以完成的任何事情都可以通过纯 JavaScript 完成，如下所示：

JavaScript

```javascript
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);

// 等同于下面

class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

ReactDOM.render(
  React.createElement(Hello, {toWhat: 'World'}, null),
  document.getElementById('root')
);
```

D. React.createElement 实现

返回一个类似 html 嵌套的 json 对象

JavaScript

```
function createElement(type, props, ...children) {
  return {
    type,
    props: {
      ...props,
      children: children.map((item) => {
        // 这个递归就很灵性哈哈哈
        typeof item === 'object' ? item : createTextElement(item)
      })
    }
  }
}
```

E. jsx 的 props 属性命名

1. 因为 JSX 语法上更接近 JavaScript 而不是 HTML，所以 React DOM 使用 camelCase（小驼峰命名）来定义属性的名称，而不使用 HTML 属性名称的命名约定，例如 class 改为 className，tabindex 改为 tabIndex
2. 如果往原生 dom 中添加自定义属性，**要以 data- 为开头**，不然 React 是不会渲染的

JavaScript

```
<div a="demo"></div> // 错误❌
<div data-a="demo"></div> //对
```

2. render

A. render 函数是什么，有啥用

渲染函数，将虚拟 DOM 渲染，向真实 DOM 添加内容，处理更新和删除。

B. 简单实现 render 中添加内容功能

JavaScript

```javascript
function render(element, container) {
  // 这个 element 就是 React.createElement 返回的 JSON 对象，我们平时写代码是用的是 jsx 语法糖
  // 此外我们还需要处理文本元素，如果元素类型是TEXT_ELEMENT我们创建文本节点而不是常规节点
  const dom =
    element.type == 'TEXT_ELEMENT'
      ? document.createTextNode('')
      : document.createElement(element.type)

  // 将元素prop分配给节点
  const isProperty = (key) => key !== 'children'
  Object.keys(element.props)
    .filter(isProperty)
    .forEach((name) => {
      dom[name] = element.props[name]
    })

  // 递归地为每个子节点做同样的事情，请注意这个递归
  element.props.children.forEach((child) => {
    render(child, dom)
  })

  // 将新节点附加到容器中
  container.appendChild(element)
}
```

> 不过我们可以发现一个问题：一旦开始渲染，就不会停止（递归），直到我们渲染了完整的元素树。如果元素树很大，则它可能会阻塞主线程太长时间。而且，如果浏览器需要执行高优先级的操作（例如处理用户输入或保持动画流畅），则它必须等到渲染完成为止。

C. 可中断的 render（react fiber）

要想避免因为深度递归而导致的渲染时间太长的问题，我们可以将把工作分成几个小单元，在完成每个单元后，如果需要执行其他高优先级操作，我们让浏览器中断渲染。

想要实现这一点的话，React 从v15升级到v16后重构了整个架构，**下面可以来讨论下这部分前置知识**

D. React v15

React v15架构分为下面两个

- Reconciler（协调器）—— 负责找出变化的组件
- Renderer（渲染器）—— 负责将变化的组件渲染到页面上

> 简单来说，每次更改状态触发更新时，Reconciler 会去对比找出本次更新中变化的虚拟DOM，然后通知 Renderer 把变化的组件渲染在当前宿主环境

缺点：递归更新

> 由于递归执行，所以更新一旦开始，中途就无法中断。当层级很深时，递归更新时间超过了17ms（60帧一秒，一帧16.7ms），用户交互就会卡顿

E. React v16

React16架构可以分为三层：

- Scheduler（调度器）—— 调度任务的优先级，高优任务优先进入Reconciler
- Reconciler（协调器）—— 负责找出变化的组件
- Renderer（渲染器）—— 负责将变化的组件渲染到页面上

> 可以看到，相较于React15，React16中新增了Scheduler（调度器）

调度器就是用当前是否有剩余时间作为任务中断的标准，当浏览器有剩余时间时通知渲染器。

其实部分浏览器已经实现了这个API，这就是requestIdleCallback。但是由于以下因素，React放弃使用：

- 浏览器兼容性
- requestIdleCallback is called only 20 times per second - Chrome on my 6x2 core Linux machine, it's not really useful for UI work. https://github.com/facebook/react/issues/13206#issuecomment-418923831

所以基于此，React 自己设计了调度器，除了在空闲时触发回调的功能外，Scheduler 还提供了多种调度优先级供任务设置。

React v16更新工作从递归变成了可以中断的循环过程。每次循环都会调用 shouldYield 判断当前是否有剩余时间，如下所示：

JavaScript

```
/** @noinline */
function workLoopConcurrent() {
  // Perform work until Scheduler asks us to yield
  while (workInProgress !== null &amp;amp;&amp;amp; !shouldYield()) {
    // 注意这个 performUnitOfWork，下面会讲到
    workInProgress = performUnitOfWork(workInProgress);
  }
}
```

而当Scheduler 将任务交给 Reconciler 后，Reconciler 会为变化的虚拟 DOM 打上代表增/删/更新的标记

整个 Scheduler 与 Reconciler 的工作都在内存中进行。只有当所有组件都完成 Reconciler 的工作，才会统一交给 Renderer，这样做的好处是当遇到中断时，不会更新页面上的 DOM，所以即使反复中断，用户也不会看见更新不完全的 DOM。

> 记住一个核心思想，递归变为可控的循环

F. PerformUnitOfWork（后面的知识学有余力可以看）

render 函数开始于 performUnitOfWork 方法的调用

JavaScript

```
// 可中断的遍历
function workLoopConcurrent() {
  while (workInProgress !== null &amp;amp;&amp;amp; !shouldYield()) {
    // workInProgress 代表当前已创建的 workInProgress fiber
    // performUnitOfWork 方法会创建下一个 Fiber 节点并赋值给 workInProgress，并将 workInProgress 与已创建的 Fiber 节点连接起来构成 Fiber 树
    workInProgress = performUnitOfWork(workInProgress);
  }
}
```