---

link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=267 sentences=600, words=4484

1.请说一下你对 React 的理解？#

- 官方文档 (https://zh-hans.reactjs.org/)
- 题目分析 本题属于概念题
- 解题思路
    - 是什么? 一句话说直达本质
    - 能干什么? 用途和应用场景
    - 如何干的? 核心的工作原理
    - 干的怎么样? 优缺点

1.1 React 是什么？#

- React是一个用于构建用户界面的 JavaScript 库

1.2 React 能干什么？#

- 可以通过组件化的方式构建 构建快速响应的大型 Web应用程序

1.3 React 如何干的？#

1.3.1 声明式

- 声明式 使用声明式的编写用户界面,代码可行方便调试
- 声明式渲染和命令式渲染
    - 命令式渲染 命令我们的程序去做什么,程序就会跟着你的命令去一步一步执行
    - 声明式渲染 我们只需要告诉程序我们想要什么效果，其他的交给程序来做

```
let root = document.getElementById("root");

ReactDOM.render(<h1 onClick={() => console.log("hello")}>helloh1>, root);

let h1 = document.createElement("h1");
h1.innerHTML = "hello";
h1.addEventListener("click", () => console.log("hello"));
root.appendChild(h1);
```

1.3.2 组件化

- 组件化 把页面拆分为一个个组件，方便视图的拆分和复用，还可以做到高内聚和低耦合

1.3.3 一次学习，随处编写

- 可以使用 React 开发 Web、Android、IOS、VR 和命令行程序
- ReactNative 使用 React 来创建 Android 和 iOS 的原生应用
- React 360 是一个创建 3D 和 VR 用户交互的框架

1.4 React 干的怎么样？#

1.4.1 优点

- 开发团队和社区强大
- 一次学习，随处编写
- API 比较简洁

1.4.2 缺点

- 没有官方系统解决方案，选型成本高
- 过于灵活，不容易写出高质量的应用

1.5 其它扩展

- JSX 实现声明式
- 虚拟 DOM 可以实现跨平台
- React 使用的设计模式
- 自己 React 大型架构经验

2.为什么 React 会引入 JSX?

- 题目分析 方案选型，考察知识广度
- 解题思路
    - 解释概念?
    - 想实现什么目的?
    - 有哪些可选方案，为什么这个方案最好
    - JSX 的工作原理?

2.1 JSX 是什么

- jsx (https://zh-hans.reactjs.org/docs/introducing-jsx.html)
- JSX 是一个 JavaScript的语法扩展,JSX 可以很好地描述 UI 应该呈现出它应有交互的本质形式
- JSX 其实是 React.createElement的语法糖

2.2 React 想实现什么目的？#

- 需要实现声明式
- 代码结构需要非常清晰和简洁，可读性强
- 结构、样式和事件等能够实现高内聚低耦合,方便重用和组合
- 不想引入新的的概念和语法,只写 JavaScript

2.3 为什么 JSX 最好

2.3.1 模板

- Vue.js 使用了基于 HTML 的模板语法，允许开发者声明式地将 DOM 绑定至底层 Vue 实例的数据
- 引入太多概念，比如 Angular 就引入了控制器、作用域、服务等概念

```
增加 1
```

2.4 JSX 工作原理

- babeljs (https://www.babeljs.cn/repl)
- astexplorer (https://astexplorer.net)

2.4.1 安装

```
npm install @babel/core @babel/plugin-syntax-jsx @babel/plugin-transform-react-jsx @babel/types --save
```

2.4.2 AST抽象语法树

- 抽象语法树（Abstract Syntax Tree，AST）是源代码语法结构的一种抽象表示。它以树状的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构

2.4.3 babel 工作流

2.4.4 旧转换

```
const babel = require("@babel/core");
const sourceCode = `hello`;
const result = babel.transform(sourceCode, {
  plugins: [["@babel/plugin-transform-react-jsx", { runtime: "classic" }]],
});
console.log(result.code);
```

2.4.5 新转换

```
const babel = require("@babel/core");
const sourceCode = `hello`;
const result = babel.transform(sourceCode, {
  plugins: [["@babel/plugin-transform-react-jsx", { runtime: "automatic" }]],
});
console.log(result.code);
```

3. 请说一下你对 Virtual DOM 的理解？#

- 题目分析 本题属于概念题
- 解题思路
    - 是什么? 一句话说直达本质
    - 能干什么? 用途和应用场景
    - 如何干的? 核心的工作原理
    - 干的怎么样? 优缺点

3.1 创建项目

3.1.1 安装

```
npm install @babel/core @babel/plugin-proposal-class-properties @babel/plugin-proposal-decorators @babel/preset-env  @babel/preset-react  babel-loader html-webpack-plugin webpack webpack-cli  webpack-dev-server --save-dev
```

```
npm install react@experimental react-dom@experimental --save
```

3.1.2 webpack.config.js

```
const path = require("path");
const HtmlWebpackPlugin = require("html-webpack-plugin");
module.exports = {
  mode: "development",
  devtool: "source-map",
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "dist"),
    filename: "[name].js",
    publicPath: "/",
  },
  module: {
    rules: [
      {
        test: /\.js?$/,
        use: {
          loader: "babel-loader",
          options: {
            presets: [["@babel/preset-env"], "@babel/preset-react"],
            plugins: [
              ["@babel/plugin-proposal-decorators", { legacy: true }],
              ["@babel/plugin-proposal-class-properties", { loose: true }],
            ],
          },
        },
        exclude: /node_modules/,
      },
    ],
  },
  plugins: [new HtmlWebpackPlugin({ template: "./public/index.html" })],
};
```

3.1.3 package.json

```
{
  "scripts": {
    "start": "webpack serve"
  }
}
```

3.2 实现虚拟 DOM

- React.createElement 函数所返回的就是一个虚拟 DOM
- 虚拟 DOM 就是一个描述真实 DOM 的纯 JS 对象

3.2.1 src\index.js

src\index.js

```
import React from "./react";
let virtualDOM = (
  <div id="A1" key="A1">
    <div id="B1" key="B1">
      B1
    div>
    <div id="B2" key="B2">
      B2
    div>
  div>
);
console.log(virtualDOM);
```

3.2.2 src\react.js

- src\react.js (https://gitee.com/mirrors/react/blob/v17.0.1/packages/react/src/React.js#L101)

```
import { createElement } from "./ReactElement";
const React = {
  createElement,
};
export default React;
```

3.2.3 src\ReactSymbols.js

- src\ReactSymbols.js (//https://gitee.com/mirrors/react/blob/v17.0.1/packages/shared/ReactSymbols.js#L39)

```
const symbolFor = Symbol.for;
export let REACT_ELEMENT_TYPE = symbolFor("react.element");
```

3.2.4 ReactElement.js

src\ReactElement.js

- src\ReactElement.js (https://gitee.com/mirrors/react/blob/v17.0.1/packages/react/src/ReactElement.js#L348)

```

import { REACT_ELEMENT_TYPE } from "./ReactSymbols";
const RESERVED_PROPS = {
  key: true,
  ref: true,
  _store: true,
  __self: true,
  __source: true,
};
export function createElement(type, config, children) {
  const props = {};
  let key = null;
  if (config != null) {
    key = config.key;
  }
  for (let propName in config) {
    if (!RESERVED_PROPS.hasOwnProperty(propName)) {
      props[propName] = config[propName];
    }
  }
  const childrenLength = arguments.length - 2;
  if (childrenLength === 1) {
    props.children = children;
  } else if (childrenLength > 1) {
    const childArray = Array(childrenLength);
    for (let i = 0; i < childrenLength; i++) {
      childArray[i] = arguments[i + 2];
    }
    props.children = childArray;
  }

  const element = {
    $typeof: REACT_ELEMENT_TYPE,
    type,
    key,
    props,
  };
  return element;
}
```

3.3 优缺点

3.3.1 优点

- 处理了浏览器兼容性问题，避免用户操作真实 DOM，那么又麻烦又容易出错
- 内容经过了 XSS 处理，可以防范 XSS 攻击
- 容易实现跨平台开发 Android、iOS、VR 应用
- 更新的时候可以实现差异化更新，减少更新 DOM 的操作

3.3.2 缺点

- 虚拟 DOM 需要消耗额外的内存
- 首次渲染其实并不一定会更快

4. 函数组件和类组件的相同点和不同点？#

- 组件 & Props (https://zh-hans.reactjs.org/docs/components-and-props.html)
- 组合 vs 继承 (https://zh-hans.reactjs.org/docs/composition-vs-inheritance.html)
- 组件允许你将 UI 拆分为独立可复用的代码片段，并对每个片段进行独立构思
- 题目分析 本题属于差异题
- 解题思路
    - 相同点
    - 不同点

4.1 实现组件

4.1 实现函数组件

4.1.1 src\index.js

```
import React from './react';
let virtualDOM = (

    B1
    B2

)
+function FunctionComponent(){
+  return virtualDOM;
+}
+let functionVirtualDOM = ;
console.log(functionVirtualDOM);
```

4.2 实现类组件

4.2.1 src\index.js

```
import React from './react';
let virtualDOM = (

    B1
    B2

)
+class ClassComponent extends React.Component{
+  render(){
+      return virtualDOM;
+  }
+}
+let functionVirtualDOM = ;
console.log(functionVirtualDOM);
```

4.2.2 ReactBaseClasses.js

- src\ReactBaseClasses.js ([src\ReactBaseClasses.js](https://gitee.com/mirrors/react/blob/v17.0.1/packages/react/src/ReactBaseClasses.js#L20)>)

```
export function Component(props) {
  this.props = props;
}

Component.prototype.isReactComponent = {};
```

4.2.3 src\react.js

src\react.js

```
import {createElement} from './ReactElement';
+import {Component} from './ReactBaseClasses';
const React = {
    createElement,
+   Component
};
export default React;
```

4.3 相同点和不同点

4.3.1 相同点

- 它们都可以接收属性并且返回 React 元素

4.3.2 不同点

- 编程思想不同: 类组件需要创建实例，是基于面向对象的方式编程，而函数式组件不需要创建实例，接收输入，返回输出，是基于函数式编程的思路来编写的
- 内存占用：类组件需要创建并保存实例，会占用一定内存，函数组件不需要创建实例，可以节约内存占用
- 捕获特性：函数组件具有值捕获特性
- 可测试性: 函数式组件更方便编写单元测试
- 状态: 类组件有自己的实例，可以定义状态，而且可以修改状态更新组件，函数式组件以前没有状态，现在可以使用 useState 使用状态
- 生命周期: 类组件有自己完整的生命周期，可以在生命周期内编写逻辑，函数组件以前没有生命周期，现在可以使用 useEffect 实现类似生命周期的功能
- 逻辑复用: 类组件可以通过继承实现逻辑的复用，但官方推荐组件优于继承，函数组件可以通过自定义 Hooks 实现逻辑的复用
- 跳过更新: 类组件可以通过 `shouldComponentUpdate`和 `PureComponent`来跳过更新，而函数式组件可以使用 `React.memo`来跳过更新
- 发展前景: 未来函数式组件将会成为主流，因为它可以更好的屏蔽 this 问题、规范和复用逻辑、更好的适合时间分片和并发渲染

4.3.2.1 捕获特性

```
import React from "react";
import ReactDOM from "react-dom";
class ClassComponent extends React.Component {
  state = { number: 0 };
  handleClick = () => {
    setTimeout(() => console.log(this.state.number), 3000);
    this.setState({ number: this.state.number + 1 });
  };
  render() {
    return (
      <div>
        <p>{this.state.number}p>
        <button onClick={this.handleClick}>+button>
      div>
    );
  }
}
function FunctionComponent() {
  let [number, setNumber] = React.useState(0);
  let handleClick = () => {
    setTimeout(() => console.log(number), 3000);
    setNumber(number + 1);
  };
  return (
    <div>
      <p>{number}p>
      <button onClick={handleClick}>+button>
    div>
  );
}

let virtualDOM = <FunctionComponent />;
ReactDOM.render(virtualDOM, document.getElementById("root"));
```

4.3.2.2 跳过更新

```
class PureComponent extends Component {
  shouldComponentUpdate(newProps, nextState) {
    return (
      !shallowEqual(this.props, newProps) ||
      !shallowEqual(this.state, nextState)
    );
  }
}
function shallowEqual(obj1, obj2) {
  if (obj1 === obj2) {
    return true;
  }
  if (
    typeof obj1 != "object" ||
    obj1 === null ||
    typeof obj2 != "object" ||
    obj2 === null
  ) {
    return false;
  }
  let keys1 = Object.keys(obj1);
  let keys2 = Object.keys(obj2);
  if (keys1.length != keys2.length) {
    return false;
  }
  for (let key of keys1) {
    if (!obj2.hasOwnProperty(key) || obj1[key] !== obj2[key]) {
      return false;
    }
  }
  return true;
}
```

5. 请说一下 React 中的渲染流程

- 题目分析 本题属于原理题
- 解题思路
  - 宏观的设计理念
  - 关键原理清晰描述，抽象和具象相结合
  - 结合工程实践和工作成果

5.1 设计理念

- 跨平台渲染=>虚拟 DOM
- 快速响应=>异步可中断+增量更新

5.2 性能瓶颈

- JS 任务执行时间过长
  - 浏览器刷新频率为 60Hz,大概 16.6 毫秒渲染一次，而 JS 线程和渲染线程是互斥的，所以如果 JS 线程执行任务时间超过 16.6ms 的话，就会导致掉帧，导致卡顿，解决方案就是 React 利用空闲的时间进行更新，不影响渲染进行的渲染
  - 把一个耗时任务切分成一个个小任务，分布在每一帧里的方式就叫时间切片

5.3 案例

- concurrent-mode (https://zh-hans.reactjs.org/docs/concurrent-mode-intro.html)
- concurrent-mode-adoption (https://zh-hans.reactjs.org/docs/concurrent-mode-adoption.html)

```
import React from "react";
import ReactDOM from "react-dom";
class App extends React.Component {
  state = { list: new Array(10000).fill(0) };
  add = () => {
    this.setState({ list: [...this.state.list, 1] });
  };
  render() {
    return (
      <ul>
        <input />
        <button onClick={this.add}>addbutton>
        {this.state.list.map((item, index) => (
          <li key={index}>{item}li>
        ))}
      ul>
    );
  }
}
let root = document.getElementById("root");

ReactDOM.unstable_createRoot(root).render(<App />);
```

5.4 屏幕刷新率

- 目前大多数设备的屏幕刷新率为 60 次/秒
- 浏览器渲染动画或页面的每一帧的速率也需要跟设备屏幕的刷新率保持一致
- 页面是一帧一帧绘制出来的，当每秒绘制的帧数（FPS）达到 60 时，页面是流畅的,小于这个值时，用户会感觉到卡顿
- 每个帧的预算时间是 16.66 毫秒 (1 秒/60)
- 1s 60 帧，所以每一帧分到的时间是 1000/60 ≈ 16 ms,所以我们书写代码时力求不让一帧的工作量超过 16ms

5.5 帧

- 每个帧的开头包括样式计算、布局和绘制
- JavaScript 执行 Javascript 引擎和页面渲染引擎在同一个渲染线程,GUI 渲染和 Javascript 执行两者是互斥的
- 如果某个任务执行时间过长，浏览器会推迟渲染

5.6 requestIdleCallback

- 我们希望快速响应用户，让用户觉得够快，不能阻塞用户的交互
- requestIdleCallback 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
- 正常帧任务完成后没超过 16 ms,说明时间有富余，此时就会执行 `requestIdleCallback` 里注册的任务

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  head>
  <body>
    <script>
      function sleep(d) {
        for (var t = Date.now(); Date.now() - t const works = [
        () => {
          console.log("第1个任务开始");
          sleep(20);
          console.log("第1个任务结束");
        },
        () => {
          console.log("第2个任务开始");
          sleep(20);
          console.log("第2个任务结束");
        },
        () => {
          console.log("第3个任务开始");
          sleep(20);
          console.log("第3个任务结束");
        },
      ];

      requestIdleCallback(workLoop);
      function workLoop(deadline) {
        console.log("本帧剩余时间", parseInt(deadline.timeRemaining()));
        while (deadline.timeRemaining() > 1 && works.length > 0) {
          performUnitOfWork();
        }
        if (works.length > 0) {
          console.log(
            `只剩下${parseInt(
              deadline.timeRemaining()
            )}ms,时间片到了等待下次空闲时间的调度`
          );
          requestIdleCallback(workLoop);
        }
      }
      function performUnitOfWork() {
        works.shift()();
      }
    script>
  body>
html>
```

5.7 React16+的渲染流程

- scheduler (https://gitee.com/mirrors/react/tree/v17.0.1/packages/scheduler) 选择高优先级的任务进入 reconciler
- reconciler (https://gitee.com/mirrors/react/tree/v17.0.1/packages/react-reconciler) 计算变更的内容
- react-dom (https://gitee.com/mirrors/react/tree/v17.0.1/packages/react-dom) 把变更的内容渲染到页面上

5.7.1 index.js

```
import React from "./react";
import ReactDOM from "./react-dom";
let style = { border: "3px solid red", margin: "5px" };
let virtualDOM = (
  <div id="A1" key="A1" style={style}>
    A1
  div>
);
let root = document.getElementById("root");
ReactDOM.render(virtualDOM, root);
```

5.7.5 fiber

- 我们可以通过某些调度策略合理分配 CPU 资源，从而提高用户的响应速度
- 通过 Fiber 架构，让自己的调和过程变成可被中断。 适时地让出 CPU 执行权，除了可以让浏览器及时地响应用户的交互

5.7.5.1 Fiber是一个执行单元

- Fiber 是一个执行单元,每次执行完一个执行单元, React 就会检查现在还剩多少时间，如果没有时间就将控制权让出去

5.7.5.2 Fiber是一种数据结构

- React 目前的做法是使用链表, 每个 VirtualDOM 节点内部表示为一个 Fiber
- 从顶点开始遍历
- 如果有第一个儿子，先遍历第一个儿子
- 如果没有第一个儿子，标志着此节点遍历完成
- 如果有弟弟遍历弟弟
- 如果有没有下一个弟弟，返回父节点标识完成父节点遍历，如果有叔叔遍历叔叔
- 没有父节点遍历结束

5.8 实现渲染

5.8.1 定义 JSX

```
let style = { border: "1px solid red", color: "red", margin: "5px" };
let A = {
  type: "div",
  key: "A",
  props: {
    style,
    children: [
      "A文本",
      { type: "div", key: "B1", props: { style, children: "B1文本" } },
      { type: "div", key: "B2", props: { style, children: "B2文本" } },
    ],
  },
};
```

5.8.2.workLoop

```
let style = {border:'1px solid red',color:'red',margin:'5px'};
let A = {
  type: 'div',
  key: 'A',
  props: {
      style,
      children: [
          'A文本',
          { type: 'div', key: 'B1', props: { style,children: 'B1文本' } },
          { type: 'div', key: 'B2', props: { style,children: 'B2文本' } }
      ]
  }
}
+let workInProgress;
+const TAG_ROOT = 'TAG_ROOT';
+function workLoop() {
+  while (workInProgress) {
+      workInProgress = performUnitOfWork(workInProgress);
+  }
+}
+let rootFiber = {
+  tag: TAG_ROOT,
+  key: 'ROOT',
+  stateNode: document.getElementById('root'),
+  props: { children: [A] }
+}
+function performUnitOfWork(fiber) {
+  console.log(fiber.key);
+}
+workInProgress=rootFiber;
+workLoop();
```

5.8.3.beginWork

```
let style = {border:'1px solid red',color:'red',margin:'5px'};
let A = {
  type: 'div',
  key: 'A',
  props: {
      style,
      children: [
          'A文本',
          { type: 'div', key: 'B1', props: { style,children: 'B1文本' } },
          { type: 'div', key: 'B2', props: { style,children: 'B2文本' } }
      ]
  }
}
let workInProgress;
const TAG_ROOT = 'TAG_ROOT';
const TAG_TEXT = 'TAG_TEXT';
const TAG_HOST = 'TAG_HOST';
function workLoop() {
  while (workInProgress) {
      workInProgress = performUnitOfWork(workInProgress);
  }
}
let rootFiber = {
  tag: TAG_ROOT,
  key: 'ROOT',
  stateNode: document.getElementById('root'),
  props: { children: [A] }
}

workInProgress=rootFiber;
workLoop();

function performUnitOfWork(fiber) {
+  beginWork(fiber);
+  if (fiber.child) {//如果子节点就返回第一个子节点
+      return fiber.child;
+  }
+  while (fiber) {//如果没有子节点说明当前节点已经完成了渲染工作
+      if (fiber.sibling) {//如果它有弟弟就返回弟弟
+          return fiber.sibling;
+      }
+      fiber = fiber.return;//如果没有弟弟让爸爸完成，然后找叔叔
+  }
}
+/**
+ * 根据当前的fiber和子JSX构建子fiber树
+ * @param {*} fiber
+ * @returns
+ */
+function beginWork(fiber) {
+  console.log('beginWork', fiber.key);
+  let nextChildren = fiber.props.children;
+  if(typeof nextChildren === 'string'){
+    nextChildren=null;
+  }
+  return reconcileChildren(fiber,nextChildren);
+}
+
+function reconcileChildren(returnFiber, nextChildren) {
+  let firstChild = null;
+  let previousNewFiber = null;
+  let newChildren=[];
+  if(Array.isArray(nextChildren)){
+    newChildren = nextChildren;
+  }else if(!!nextChildren){
+    newChildren=[nextChildren];
+  }
+  for (let newIdx = 0; newIdx < newChildren.length; newIdx++) {
+      let newFiber = createFiber(newChildren[newIdx]);
+      newFiber.return = returnFiber;
+      if (!previousNewFiber) {
+        firstChild = newFiber;
+      } else {
+          previousNewFiber.sibling = newFiber;
+      }
+      previousNewFiber = newFiber;
+  }
+  returnFiber.child = firstChild;
+  return firstChild;
+}
+function createFiber(element) {
+  if (typeof element === 'string') {
+      return { tag: TAG_TEXT, type: element.type, key: element, props: element };
+  } else {
+      return { tag: TAG_HOST, type: element.type, key: element.key, props: element.props };
+  }
+}
```

5.8.4. completeUnitOfWork

```
+import {setInitialProperties} from './utils';
let style = {border:'1px solid red',color:'red',margin:'5px'};
let A = {
  type: 'div',
  key: 'A',
  props: {
      style,
      children: [
          'A文本',
          { type: 'div', key: 'B1', props: { style,children: 'B1文本' } },
          { type: 'div', key: 'B2', props: { style,children: 'B2文本' } }
      ]
  }
```

```
}
let workInProgress;
const TAG_ROOT = 'TAG_ROOT';
const TAG_TEXT = 'TAG_TEXT';
const TAG_HOST = 'TAG_HOST';
+const Placement = 'Placement';

function workLoop() {
  while (workInProgress) {
      workInProgress = performUnitOfWork(workInProgress);
  }
}
let rootFiber = {
  tag: TAG_ROOT,
  key: 'ROOT',
  stateNode: document.getElementById('root'),
  props: { children: [A] }
}

workInProgress=rootFiber;
workLoop();

function performUnitOfWork(fiber) {
  beginWork(fiber);
  if (fiber.child) {//如果子节点就返回第一个子节点
      return fiber.child;
  }
  while (fiber) {//如果没有子节点说明当前节点已经完成了渲染工作
+     completeUnitOfWork(fiber);//可以结束此fiber的渲染了
      if (fiber.sibling) {//如果它有弟弟就返回弟弟
          return fiber.sibling;
      }
      fiber = fiber.return;//如果没有弟弟让爸爸完成，然后找叔叔
  }
}
+function completeUnitOfWork(workInProgress) {
+  console.log('completeUnitOfWork', workInProgress.key);
+  let stateNode;
+  switch (workInProgress.tag) {
+    case TAG_HOST:
+      stateNode=createStateNode(workInProgress);
+      setInitialProperties(stateNode, workInProgress.props);
+      break;
+    case TAG_TEXT:
+      createStateNode(workInProgress);
+      break;
+  }
+  makeEffectList(workInProgress);
+}
+function createStateNode(fiber){
+  if (fiber.tag === TAG_TEXT) {
+      let stateNode = document.createTextNode(fiber.props);
+      fiber.stateNode = stateNode;
+  } else if (fiber.tag === TAG_HOST) {
+      let stateNode = document.createElement(fiber.type);
+      if (typeof fiber.props.children === 'string') {
+          stateNode.appendChild(document.createTextNode(fiber.props.children));
+      }
+      fiber.stateNode = stateNode;
+  }
+  return fiber.stateNode;
+}
+function makeEffectList(completedWork){
+  const returnFiber = completedWork.return;
+  if (returnFiber) {
+    if (!returnFiber.firstEffect) {//父亲为空就指向儿子的子链表
+      returnFiber.firstEffect = completedWork.firstEffect;
+    }
+    if (completedWork.lastEffect) {//父亲非空就父亲老尾下一个指向儿子子链表头,父亲尾指出儿子子链表头
+      if (returnFiber.lastEffect) {
+          returnFiber.lastEffect.nextEffect = completedWork.firstEffect;
+      }
+      returnFiber.lastEffect = completedWork.lastEffect;//父亲的尾指向自己的尾
+    }
+    if (completedWork.flags) {
+          if (returnFiber.lastEffect) {//如果父亲有尾，尾巴下一个指向自己
+              returnFiber.lastEffect.nextEffect = completedWork;
+          } else {//如果父亲没有尾，父亲的头毛都指向自己
+              returnFiber.firstEffect = completedWork;
+          }
+          returnFiber.lastEffect = completedWork;
+    }
+  }
}
/**
 * 根据当前的fiber和子JSX构建子fiber树
 * @param {*} fiber
 * @returns
 */
function beginWork(fiber) {
  console.log('beginWork', fiber.key);
  let nextChildren = fiber.props.children;
  if(typeof nextChildren
    nextChildren=null;
  }
  return reconcileChildren(fiber,nextChildren);
}

function reconcileChildren(returnFiber, nextChildren) {
  let firstChild = null;
  let previousNewFiber = null;
  let newChildren=[];
  if(Array.isArray(nextChildren)){
    newChildren = nextChildren;
```

```
  }else if(!!nextChildren){
    newChildren=[nextChildren];
  }
  for (let newIdx = 0; newIdx < newChildren.length; newIdx++) {
      let newFiber = createFiber(newChildren[newIdx]);
      newFiber.return = returnFiber;
      if (!previousNewFiber) {
        firstChild = newFiber;
      } else {
          previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
  }
  returnFiber.child = firstChild;
  return firstChild;
}
function createFiber(element) {
  if (typeof element
      return { tag: TAG_TEXT, type: element.type, key: element, props: element };
  } else {
      return { tag: TAG_HOST, type: element.type, key: element.key, props: element.props };
  }
}
```

5.8.5 commitRoot

```
import {setInitialProperties} from './utils';
let style = {border:'1px solid red',color:'red',margin:'5px'};
let A = {
  type: 'div',
  key: 'A',
  props: {
      style,
      children: [
          'A文本',
          { type: 'div', key: 'B1', props: { style,children: 'B1文本' } },
          { type: 'div', key: 'B2', props: { style,children: 'B2文本' } }
      ]
  }
}
let workInProgress;
const TAG_ROOT = 'TAG_ROOT';
const TAG_TEXT = 'TAG_TEXT';
const TAG_HOST = 'TAG_HOST';
const Placement = 'Placement';

function workLoop() {
  while (workInProgress) {
      workInProgress = performUnitOfWork(workInProgress);
  }
+ commitRoot(rootFiber);
}
+function commitRoot(rootFiber){
+  let currentEffect = rootFiber.firstEffect;
+  while(currentEffect){
+    let flags = currentEffect.flags;
+    switch (flags) {
+      case Placement:
+        commitPlacement(currentEffect);
+        break;
+    }
+    currentEffect=currentEffect.nextEffect;
+  }
+}
function commitPlacement(currentFiber) {
  let parent = currentFiber.return.stateNode;
  parent.appendChild(currentFiber.stateNode);
}
let rootFiber = {
  tag: TAG_ROOT,
  key: 'ROOT',
  stateNode: document.getElementById('root'),
  props: { children: [A] }
}

workInProgress=rootFiber;
workLoop();

function performUnitOfWork(fiber) {
  beginWork(fiber);
  if (fiber.child) {//如果子节点就返回第一个子节点
      return fiber.child;
  }
  while (fiber) {//如果没有子节点说明当前节点已经完成了渲染工作
      completeUnitOfWork(fiber);//可以结束此fiber的渲染了
      if (fiber.sibling) {//如果它有弟弟就返回弟弟
          return fiber.sibling;
      }
      fiber = fiber.return;//如果没有弟弟让爸爸完成，然后找叔叔
  }
}
function completeUnitOfWork(workInProgress) {
  console.log('completeUnitOfWork', workInProgress.key);
  let stateNode;
  switch (workInProgress.tag) {
    case TAG_HOST:
      stateNode=createStateNode(workInProgress);
      setInitialProperties(stateNode, workInProgress.props);
      break;
    case TAG_TEXT:
      createStateNode(workInProgress);
      break;
  }
```

```
  makeEffectList(workInProgress);
}
function createStateNode(fiber){
  if (fiber.tag
      let stateNode = document.createTextNode(fiber.props);
      fiber.stateNode = stateNode;
  } else if (fiber.tag
      let stateNode = document.createElement(fiber.type);
      if (typeof fiber.props.children
          stateNode.appendChild(document.createTextNode(fiber.props.children));
      }
      fiber.stateNode = stateNode;
  }
  return fiber.stateNode;
}
function makeEffectList(completedWork){
  const returnFiber = completedWork.return;
  if (returnFiber) {
    if (!returnFiber.firstEffect) {//父亲为空就指向儿子的子链表
      returnFiber.firstEffect = completedWork.firstEffect;
    }
    if (completedWork.lastEffect) {//父亲非空就父亲老尾下一个指向儿子子链表头,父亲尾指出儿子子链表头
      if (returnFiber.lastEffect) {
          returnFiber.lastEffect.nextEffect = completedWork.firstEffect;
      }
      returnFiber.lastEffect = completedWork.lastEffect;//父亲的尾指向自己的尾
    }
    if (completedWork.flags) {
          if (returnFiber.lastEffect) {//如果父亲有尾，尾巴下一个指向自己
              returnFiber.lastEffect.nextEffect = completedWork;
          } else {//如果父亲没有尾，父亲的头毛都指向自己
              returnFiber.firstEffect = completedWork;
          }
          returnFiber.lastEffect = completedWork;
    }
  }
}
/**
 * 根据当前的fiber和子JSX构建子fiber树
 * @param {*} fiber
 * @returns
 */
function beginWork(fiber) {
  console.log('beginWork', fiber.key);
  let nextChildren = fiber.props.children;
  if(typeof nextChildren
    nextChildren=null;
  }
  return reconcileChildren(fiber,nextChildren);
}

function reconcileChildren(returnFiber, nextChildren) {
  let firstChild = null;
  let previousNewFiber = null;
  let newChildren=[];
  if(Array.isArray(nextChildren)){
    newChildren = nextChildren;
  }else if(!!nextChildren){
    newChildren=[nextChildren];
  }
  for (let newIdx = 0; newIdx < newChildren.length; newIdx++) {
      let newFiber = createFiber(newChildren[newIdx]);
      newFiber.return = returnFiber;
      newFiber.flags = Placement;
      if (!previousNewFiber) {
        firstChild = newFiber;
      } else {
          previousNewFiber.sibling = newFiber;
      }
      previousNewFiber = newFiber;
  }
  returnFiber.child = firstChild;
  return firstChild;
}
function createFiber(element) {
  if (typeof element
      return { tag: TAG_TEXT, type: element.type, key: element, props: element };
  } else {
      return { tag: TAG_HOST, type: element.type, key: element.key, props: element.props };
  }
}
```

6. 请说一下 React 中有 DOM-DIFF 算法？#

- 在 React17+中 DOM-DIFF 就是根据老的 fiber 树和最新的 JSX 对比生成新的 fiber 树的过程

6.1 React 优化原则

- 只对同级节点进行对比，如果 DOM 节点跨层级移动，则 React 不会复用
- 不同类型的元素会产出不同的结构 ，会销毁老结构，创建新结构
- 可以通过 key 标识移动的元素

6.2 单节点

- 如果新的节点只有一个的话

- type 不同

```
h1

/*************/

h2
```

- key 不同

```
h1

/*************/

h2
```

- type 和 key 都相同

```
h1

/*************/

h1-new
```

- key 相同但是 type 不同，直接删除所有老节点

```
h1
h2

/*************/

p
```

- key 不同，删除当前老节点，接着对比下一个节点

```
h1
h2

/*************/

h2
```

6.3 多节点

- 如果新的节点有多个节点的话
- 节点有可能更新、删除、新增
- 多节点的时候会经历二轮遍历
- 第一轮遍历主要是处理节点的更新,更新包括属性和类型的更新
- 第二轮遍历主要处理节点的新增、删除和移动
- 移动时的原则是尽量少量的移动，如果必须有一个要动，新地位高的不动，新地位低的动
- 一一对比，都可复用，只需更新

```
A
B
C
D

/*************/

A-new
B-new
C-new
D-new
```

- 一一对比，key 相同，type 不同，删除老的，添新的

```
A
B
C
D

/*************/

A-new
B-new
C-new
D-new
```

- key 不同退出第一轮循环

```
A
B
C
D

/*************/

A-new
C-new
D-new
B-new
```

移动

```
import * as React from "react";
import * as ReactDOM from "react-dom";
let oldStyle = { border: "3px solid red", margin: "5px" };
let newStyle = { border: "3px solid green", margin: "5px" };
let root = document.getElementById("root");
let oldVDOM = (
  <ul>
    <li key="A" style={oldStyle}>
      A
    li>
    <li key="B" style={oldStyle}>
      B
    li>
    <li key="C" style={oldStyle}>
      C
    li>
    <li key="D" style={oldStyle}>
      D
    li>
    <li key="E" style={oldStyle}>
      E
    li>
    <li key="F" style={oldStyle}>
      F
    li>
  ul>
);
ReactDOM.render(oldVDOM, root);
setTimeout(() => {
  let newVDOM = (
    <ul>
      <li key="A" style={newStyle}>
        A-new
      li>
      <li key="C" style={newStyle}>
        C-new
      li>
      <li key="E" style={newStyle}>
        E-new
      li>
      <li key="B" style={newStyle}>
        B-new
      li>
      <li key="G" style={newStyle}>
        G
      li>
    ul>
  );
  ReactDOM.render(newVDOM, root);
}, 1000);
```

7. 请说一下你对 React 合成事件的理解？

7.1 事件工作流

- 事件捕获
- 事件目标
- 事件冒泡
- 事件委托
- 先绑定先执行

7.2 事件差异

类型 原生事件 合成事件 命名方式 全小写 小驼峰命名 事件处理函数 字符串 函数对象 阻止默认行为 返回 false event.preventDefault()

```
const handleClick = (event)=>{event.preventDefault();}
// 原生事件
Button

//合成事件
Button
```

7.3 合成事件

- React 把事件委托到 document 对象上
- 当真实 DOM 元素触发事件,先处理原生事件，然后会冒泡到 document 对象后,再处理 React 事件
- React 事件绑定的时刻是在 reconciliation 阶段,会在原生事件的绑定前执行
- 目的和优势
    - 进行浏览器兼容,React 采用的是顶层事件代理机制，能够保证冒泡一致性
    - 事件对象可能会被频繁创建和回收，因此 React 引入事件池,在事件池中获取或释放事件对象(React17 中被废弃)

7.3.1 React17 以前

7.3.1.1 使用

```
import * as React from "react";
import * as ReactDOM from "react-dom";
class App extends React.Component {
  parentRef = React.createRef();
  childRef = React.createRef();
  componentDidMount() {
    this.parentRef.current.addEventListener(
      "click",
      () => {
        console.log("父元素原生捕获");
      },
      true
    );
    this.parentRef.current.addEventListener("click", () => {
      console.log("父元素原生冒泡");
    });
    this.childRef.current.addEventListener(
      "click",
      () => {
        console.log("子元素原生捕获");
      },
      true
    );
    this.childRef.current.addEventListener("click", () => {
      console.log("子元素原生冒泡");
    });
    document.addEventListener(
      "click",
      () => {
        console.log("document捕获");
      },
      true
    );
    document.addEventListener("click", () => {
      console.log("document冒泡");
    });
  }
  parentBubble = () => {
    console.log("父元素React事件冒泡");
  };
  childBubble = () => {
    console.log("子元素React事件冒泡");
  };
  parentCapture = () => {
    console.log("父元素React事件捕获");
  };
  childCapture = () => {
    console.log("子元素React事件捕获");
  };
  render() {
    return (
      <div
        ref={this.parentRef}
        onClick={this.parentBubble}
        onClickCapture={this.parentCapture}
      >
        <p
          ref={this.childRef}
          onClick={this.childBubble}
          onClickCapture={this.childCapture}
        >
          事件执行顺序
        p>
      div>
    );
  }
}
ReactDOM.render(<App />, document.getElementById("root"));
```

7.3.1.2 实现

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>eventtitle>
  head>
  <body>
    <div id="parent">
      <p id="child">事件执行顺序p>
    div>
    <script>
      document.addEventListener("click", dispatchEvent);
      function dispatchEvent(event, isCapture) {
        let paths = [];
        let current = event.target;
        while (current) {
          paths.push(current);
          current = current.parentNode;
        }
        for (let i = paths.length - 1; i >= 0; i--) {
          let eventHandler = paths[i].onClickCapture;
          eventHandler && eventHandler();
        }
        for (let i = 0; i < paths.length; i++) {
          let eventHandler = paths[i].onClick;
          eventHandler && eventHandler();
        }
      }
      let parent = document.getElementById("parent");
      let child = document.getElementById("child");
      parent.addEventListener(
        "click",
        () => {
          console.log("父元素原生捕获");
        },
        true
      );
      parent.addEventListener("click", () => {
        console.log("父元素原生冒泡");
      });
      child.addEventListener(
        "click",
        () => {
          console.log("子元素原生捕获");
        },
        true
      );
      child.addEventListener("click", () => {
        console.log("子元素原生冒泡");
      });
      document.addEventListener(
        "click",
        () => {
          console.log("document捕获");
        },
        true
      );
      document.addEventListener("click", () => {
        console.log("document冒泡");
      });
      parent.onClick = () => {
        console.log("父元素React事件冒泡");
      };
      parent.onClickCapture = () => {
        console.log("父元素React事件捕获");
      };
      child.onClick = () => {
        console.log("子元素React事件冒泡");
      };
      child.onClickCapture = () => {
        console.log("子元素React事件捕获");
      };

    script>
  body>
html>
```

7.3.2 React17 以后

7.3.2.1 使用

```
import * as React from "react";
import * as ReactDOM from "react-dom";
class App extends React.Component {
  parentRef = React.createRef();
  childRef = React.createRef();
  componentDidMount() {
    this.parentRef.current.addEventListener(
      "click",
      () => {
        console.log("父元素原生捕获");
      },
      true
    );
    this.parentRef.current.addEventListener("click", () => {
      console.log("父元素原生冒泡");
    });
    this.childRef.current.addEventListener(
      "click",
      () => {
        console.log("子元素原生捕获");
      },
      true
    );
    this.childRef.current.addEventListener("click", () => {
      console.log("子元素原生冒泡");
    });
    document.addEventListener(
      "click",
      () => {
        console.log("document原生捕获");
      },
      true
    );
    document.addEventListener("click", () => {
      console.log("document原生冒泡");
    });
  }
  parentBubble = () => {
    console.log("父元素React事件冒泡");
  };
  childBubble = () => {
    console.log("子元素React事件冒泡");
  };
  parentCapture = () => {
    console.log("父元素React事件捕获");
  };
  childCapture = () => {
    console.log("子元素React事件捕获");
  };
  render() {
    return (
      <div
        ref={this.parentRef}
        onClick={this.parentBubble}
        onClickCapture={this.parentCapture}
      >
        <p
          ref={this.childRef}
          onClick={this.childBubble}
          onClickCapture={this.childCapture}
        >
          事件执行顺序
        p>
      div>
    );
  }
}
ReactDOM.render(<App />, document.getElementById("root"));
```

7.3.2.2 实现

```
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>eventtitle>
  head>
  <body>
    <div id="root">
      <div id="parent">
        <p id="child">事件执行顺序p>
      div>
    div>
    <script>
      let root = document.getElementById("root");
      let parent = document.getElementById("parent");
      let child = document.getElementById("child");

      root.addEventListener(
        "click",
        (event) => dispatchEvent(event, true),
        true
      );
      root.addEventListener("click", (event) => dispatchEvent(event, false));
      function dispatchEvent(event, isCapture) {
        let paths = [];
        let current = event.target;
        while (current) {
          paths.push(current);
          current = current.parentNode;
        }
        if (isCapture) {
          for (let i = paths.length - 1; i >= 0; i--) {
            let eventHandler = paths[i].onClickCapture;
            eventHandler && eventHandler();
          }
        } else {
          for (let i = 0; i < paths.length; i++) {
            let eventHandler = paths[i].onClick;
            eventHandler && eventHandler();
          }
        }
      }

      parent.addEventListener(
        "click",
        () => {
          console.log("父元素原生捕获");
        },
        true
      );
      parent.addEventListener("click", () => {
        console.log("父元素原生冒泡");
      });
      child.addEventListener(
        "click",
        () => {
          console.log("子元素原生捕获");
        },
        true
      );
      child.addEventListener("click", () => {
        console.log("子元素原生冒泡");
      });
      document.addEventListener(
        "click",
        () => {
          console.log("document原生捕获");
        },
        true
      );
      document.addEventListener("click", () => {
        console.log("document原生冒泡");
      });
      parent.onClick = () => {
        console.log("父元素React事件冒泡");
      };
      parent.onClickCapture = () => {
        console.log("父元素React事件捕获");
      };
      child.onClick = () => {
        console.log("子元素React事件冒泡");
      };
      child.onClickCapture = () => {
        console.log("子元素React事件捕获");
      };
    script>
  body>
html>
```

7.4 事件系统变更

- 更改事件委托
    - 首先第一个修改点就是更改了事件委托绑定节点，在 16 版本中，React 都会把事件绑定到页面的 document 元素上，这在多个 React 版本共存的情况下就会虽然某个节点上的函数调用了 `event.stopPropagation()`,但还是会导致另外一个 React 版本上绑定的事件没有被阻止触发，所以在 17 版本中会把事件绑定到 render 函数的节点上
- 去除事件池
    - 17 版本中移除了事件对象池，这是因为 React 在旧浏览器中重用了不同事件的事件对象，以提高性能，并将所有事件字段在它们之前设置为 null。在 React 16 及更早版本中，使用者必须调用 `event.persist()` 才能正确的使用该事件，或者正确读取需要的属性

7.5 案例

React16

```
import * as React from "react";
import * as ReactDOM from "react-dom";
class Dialog extends React.Component {
  state = { show: false };
  componentDidMount() {
    document.addEventListener("click", () => {
      this.setState({ show: false });
    });
  }
  handleButtonClick = (event) => {

    event.nativeEvent.stopImmediatePropagation();
    this.setState({ show: true });
  };

  render() {
    return (
      <div>
        <button onClick={this.handleButtonClick}>显示button>
        {this.state.show && (
          <div
            onClick={(event) => event.nativeEvent.stopImmediatePropagation()}
          >
            Modal
          div>
        )}
      div>
    );
  }
}
ReactDOM.render(<Dialog />, document.getElementById("root"));
```

React17

```
import * as React from 'react';
import * as ReactDOM from 'react-dom';
class Dialog extends React.Component{
  state = {show: false};
  componentDidMount() {
    document.addEventListener("click",  () => {
      this.setState({show: false});
    });
  }
  handleButtonClick = (event) => {
+   event.stopPropagation();
-   //event.nativeEvent.stopImmediatePropagation();
    this.setState({show: true});
  };

  render() {
    return (

        显示
        {this.state.show && (
+          event.stopPropagation()}>
            Modal

        )}

    );
  }
}
ReactDOM.render(, document.getElementById('root'));
```