## 0 前言

**记录遇到的问题及个人答案。2023年2月一共面试了六家，最后去了发 offer 最快的一家。2024年回顾时改了下原文档，并添加个人解答。**

## 1 面试公司

### 1.1 路刻 2023.02.01

#### 1.1.1 回流重绘

html 文档节点以流的形式渲染在网页中。

**回流：**当文档流中的某个文档节点的位置和大小发生变化时，会引发文档流的变化，这就需要重新计算得到文档流中所有节点的位置和大小。

**重绘：**（文档节点的位置和大小计算完成后，所有节点的几何位置就确定了。）根据节点的特性（html 标签和 css 样式）计算绝对像素值。（用于屏幕展示）

**减少回流次数：**

1. 减少或合并修改操作；
2. 对频繁变化的节点，使用动画以及脱离文档流；
3. 获取节点位置大小信息的方法会引发回流，不要频繁掉用。

**减少重绘次数：**

1. 减少回流的次数；
2. 合并 css 样式的修改操作。

#### 1.1.2 CSSOM

浏览器渲染机制步骤：

1. 渲染引擎解析 html 生成 DOM 树，解析 CSS 生成 CSSOM 树；
2. 结合 DOM 树和 CSSOM 树，生成渲染树；
3. 根据渲染树回流和重绘，得到节点的绝对像素；
4. 显示像素。

其中，CSSOM 包含了所有具有 css 样式信息（内联样式、外部样式、继承样式）的节点。

#### 1.1.3 地址栏URL敲回车

完成的过程非常复杂，下面从不同粒度来说明整个过程。

**大致过程：**

1. 浏览器通过url找到服务器；
2. 下载相关资源，html、css、js、图像等；
3. 浏览器通过 html 和 css 渲染页面，并执行 js 脚本。

**更详细一点的过程：**

1. 浏览器判断 url 格式正确性；
2. 通过域名解析系统找到 url 对应的 ip 地址；
3. 访问该 ip 服务器的端口，下载 html 文件；
4. 解析 html 文件内部链接，下载相关资源；
5. 渲染引擎解析 html 文件生成 DOM 树，解析 css 文件生成 CSSOM 树，合并得到渲染树，再进行回流与重绘，完成这一帧页面的渲染；
6. JS执行引擎，会执行在解析 html过程中遇到的 script 标签中的 js 脚本。

**更详情的步骤细节：**

1. 浏览器程序判断 url 格式正确性，如果正确向下执行；
2. 浏览器根据缓存策略，判断是否发起新的资源请求，如果需要则执行下一步；
3. 根据本地路由缓存查找 url 对应 ip，若无，向局域网DNS查找，若无，不断向上层网络供应商DNS查找，直至找到；
4. 浏览器访问 ip 端口，请求资源下载；
5. 服务器受到请求，由本机提供服务，或者通过负载均衡策略，由其他机器提供服务；
6. 提供服务的机器，可能会对单页应用进行服务端渲染，返回处理过的html文档；
7. 前端如果得到服务端处理过的html文档，直接渲染显示，以减少前端页面白屏时间；
8. 前端如果没有得到处理过的 html 文档，则由前端按照传统步骤自行处理，过程如下：
9. 解析 html 中的资源链接，根据缓存策略，向本地缓存或服务器获取相关资源；
10. 解析 html 时，先构建一个 DOM 树，其中包含页面中所有元素的信息；
11. 遇到 script 时，会暂停解析 html 文档，并执行脚本中的代码；
12. 遇到 css 时，会构建 CSSOM 树，包含了所有具有 css 样式信息（内联样式、外部样式、继承样式）的节点；
13. 每一帧（60 Hz 时为16.7 ms ）渲染时，通过DOM树和CSSOM树生成的渲染树回流重绘，得到渲染之后的页面；
14. js脚本执行完成之后，一般会添加交互事件并发起 API 数据请求，最后引发也买呢的回流与重绘。

**更进一步的细节：**

浏览器主要分为三个部分：渲染引擎、js执行引擎、其他应用程序。

......

#### 1.1.4 react服务端渲染

前端渲染的过程，在得到html文件时就已经开始了，而css文件、js文件的加载和解析是异步的。

单页应用使用js在客户端生成html来呈现内容，用户需要等待js解析执行完成才能看到页面，这就使得首屏加载时间变长。

为了解决这个问题，在服务端可以用node解析url请求的页面的js脚本，构建完成html文件后再返回给前端。

#### 1.1.5 计时器

1、定时器是浏览器提供的功能，创建的定时器会给到 webapis 执行；

2、等待时间到达后，将回调函数转入任务队列中的宏任务队列；

3、等待当前调用栈任务执行完成、再等待微任务队列的任务执行完成；

4、排队执行宏任务队列，直到该回调函数入栈执行完成。

#### 1.1.6 取消事件监听

只能移出添加了具名函数作为回调函数的事件监听；

一般在添加事件监听前移出一下，防止重复添加；最后离开页面时，移出一下。

#### 1.1.7 引入css性能差异

四种引入方式：

1. 内联样式

难以复用，增大html文件的体积，但是可以减少下载css文件的次数。

1. style标签

比内联样式好维护一点点。

1. @import引入

基于css的规则串行下载外部css文件。

1. link引入

基于html的规则并行下载外部css文件。

#### 1.1.8 css性能提高

**从文件下载的角度：**

1. 减少http请求次数：合并css文件、合并icon图片；
2. 减少下载的文件体积：抽取公共样式、文件压缩；
3. 并行下载优于串行下载：使用html中的link标签引入外部样式，快于使用css中的@import规则引入外部样式。

**从浏览器解析css代码的角度：**

1. 减少选择器的查找时间：减少的嵌套层数、id选择器不用被嵌套、减少通配符的使用；
2. 减少无用的css代码：功能重复的代码、未生效的代码；
3. 减少使用开销大的属性：比如 filter 有条件时可平替为 color，css表达式也尽量改用其他方式实现；
4. 控制纵轴使用和图层的层数。

**从回流与重绘的角度：**

1. 合并修改次数：用class定义样式，通过切换class来切换样式；
2. 经常改动的元素脱离文档流：为动画元素使用绝对定位。

#### 1.1.9 useEffect返回值

useEffect两个参数，第一个是正常触发执行的回调函数，第二个是触发回调函数执行的依赖数组。

回调函数的返回值是一个函数，在该组件卸载时执行，或者，在重新渲染时，再次调用当前effect前会执行上一个effect的返回函数。

### 1.2 快手2023.2.6

#### 1.2.1 服务端渲染

服务端渲染常用于单页面应用中，能够通过请求的 url 去获取到比较完整的 html 文档发回给前端。这样能够减少前端的获取数据和处理数据的时间消耗。

#### 1.2.2 http2.0

http2.0 提供了更高的性能：

1. 请求和响应可以在同一个链接中并行发出；
2. 数据帧和首部压缩，减少无效数据的传输；
3. 服务器推送功能。

#### 1.2.3 实现 get(obj={a:{b:{c:2}}},path='a.b[c]',defVa=3)

1. const getTarget = (obj = { a: { b: { c: 2 } } }, path = 'a.b[c]', defVal = 3) => {
2. *// 1*
3. *// return eval(`obj.${path.replaceAll('[', '.').replaceAll(']', '')}`)*
4. *// 2*
5. const stack = []
6. let currentObj = obj
7. for (let chart of ('.' + path).split('')) {
8. switch (chart) {
9. case '.':
10. case '[':
11. stack.push(chart)
12. break;
13. case ']':
14. currentObj = currentObj[stack.pop()]
15. stack.pop()
16. break;
17. default:
18. if (stack.at(-1) === '.') {
19. currentObj = currentObj[chart]
20. stack.pop()
21. } else {
22. stack.push(chart)
23. }
24. break;
25. }
26. }
27. return currentObj
28. }
29. console.log(getTarget());

#### 1.2.4 es6功能

1、更好的变量作用域控制，let 和 const；

2、更简洁的字符串拼接，字面量；

3、函数的单步控制，迭代器与生成器、Promise、以及 es7 的async 与 await；

4、模块化管理，import 和 export；

5、便于操作，数组、对象的新的内置函数；

6、便于赋值和传参，结构赋值和拓展运算符；

其他功能在业务中不是很常见了。

#### 1.2.5 useEffect、useMemo

1、副作用函数

useEffect(()=>{

// code

return ()=>{

// when unmount or reload

}

},[

// dependence variable

])

2、保持对象引用不变

const [y] = useMemo(()=>{return y},[x])

#### 1.2.6 set、map

set 提供了具有元素唯一性的集合。

map 提供更高效和使用范围更广阔的字典。

#### 1.2.7 推算

1. new Promise(() => {
2. console.log(1);
3. throw error
4. console.log(2);
5. }).then(() => {
6. console.log(3);
7. }, () => {
8. console.log(4);
9. }).catch(() => {
10. console.log(5);
11. }).then(() => {
12. console.log(6);
13. })
14. // console 1 4 6

#### 1.2.8 inner-join

sql 表内联：两表分别有a、b条记录，则内联后得到 m 条记录，m 为 a、b 相关条件的交集。

#### 1.2.9 webpack

打包工具，主要提供的作用有：

1. 对源代码进行转换、压缩、追踪，对其他资源对模块化管理；
2. 便于开发的热更新功能；

#### 1.2.10 nodejs框架

NodeJS 是一个开源的、独立于平台的 JavaScript 运行时环境，它能够在服务器端（以及 Web 浏览器之外）执行 JavaScript 代码。

nodejs 框架是用于更快地构建 Web 应用程序服务端的不同模块、预先编写的代码、库和其他组件的集合。

#### 1.2.11 http缓存

http 请求的结果可能缓存在内存中、磁盘中，能够减少向服务器发出的请求次数。

#### 1.2.12 防抖节流

为了减少用户重复操作带来的性能损失，需要对用户操作的响应事件进行优化。

防抖：只有最后一次操作有效。

节流：一段时间内，只有一次操作有效。

#### 1.2.13 垂直居中

除了性能差的定位方式，还有比较推荐的布局方式。

比如 display: flex ，只需要对 item 设置即可。

#### 1.2.14 组件库设计思路

1、调查业务需求和发展方向，确定技术栈；

2、调研已有的组件库，找到最合适的底层库；

3、设计组件库层级、代码结构、项目组织方式；

4、开发、调试、发版。

#### 1.2.15 单元测试

对于前端开发来说，最小可测试单元可以是一个函数、一个类、一个组件，或者一个模块。

编写测试用例，使用工具（如 jest）自动进行测试。

### 1.3 字节外包2023.2.7

#### 1.3.1 webpack优化

webpack优化是指通过各种手段提高webpack打包效率和打包结果性能的过程。

对源代码和其他资源进行转换、压缩、分包，减少http请求次数和体积。

#### 1.3.2 es6功能

略。同 1.2.4 项。

#### 1.3.3 set、map

略。同 1.2.6 项。

#### 1.3.4 服务器端渲染

略。同 1.2.1 项。

#### 1.3.5 http缓存

略。同 1.2.11 项。

#### 1.3.6 事件循环

1. async function async1() {
2. console.log('async1 start')*//2*
3. await async2()
4. console.log('async1 end')*//6*
5. }
6. async function async2() {
7. console.log('async2')*//3*
8. }
9. console.log('script start')*//1*
10. setTimeout(function () {
11. console.log('setTimeout')*//9*
12. }, 0)
13. async1();
14. new Promise(function (resolve) {
15. console.log('promise1')*//4*
16. resolve();
17. }).then(function () {
18. console.log('promise2')*//7*
19. }).then(function () {
20. console.log('promise3')*//8*
21. })
22. console.log('script end')*//5*

#### 1.3.7 数组扁平化函数实现

1. type TestArr = number | number[] | TestArr[]
2. const flat = (arr: TestArr): number[] => {
3. const res: number[] = []
4. const recur = (arr: TestArr, res: number[]): void => {
5. if (typeof arr === 'number') {
6. res.push(arr)
7. return
8. }
9. for (let i = 0; i < arr.length; i++) {
10. recur(arr[i], res)
11. }
12. }
13. recur(arr, res)
14. return res
15. }
16. console.log(flat([1, 2, [3, 4], [5, [6, 7]]]));

#### 1.3.8 this绑定

1. var a = 1
2. function get() {
3. const a = 2
4. return () => { console.log(a); }
5. }
6. const obj = {
7. a: 3
8. }
9. const getA = get();
10. getA()*//2*
11. obj.getA = get();
12. obj.getA()*//2*
13. const getBind = get.bind(obj)()
14. getBind()*//2*

#### 1.3.9 useState执行

1. <!DOCTYPE html>
2. <html>
3. <head>
4. <meta charset="utf-8" />
5. <title>Hello React!</title>
6. <script src="https://unpkg.com/react@^16/umd/react.production.min.js"></script>
7. <script src="https://unpkg.com/react-dom@16.13.0/umd/react-dom.production.min.js"></script>
8. <script src="https://unpkg.com/babel-standalone@6.26.0/babel.js"></script>
9. </head>
10. <body>
11. <div id="root"></div>
12. <script type="text/babel">
13. class App extends React.Component {
14. constructor(props) {
15. super(props);
16. this.state = { A: 0 };
17. }
18. click1 = () => {
19. this.setState(prevState => ({
20. A: prevState.A + 1
21. }));
22. console.log('a1---', this.state.A);
23. this.setState(prevState => ({
24. A: prevState.A + 1
25. }));
26. console.log('a2---', this.state.A);
27. this.setState(prevState => ({
28. A: prevState.A + 1
29. }));
30. console.log('a3---', this.state.A);
31. *//  setState 是异步更新的*
32. }
33. render() {
34. return (
35. <button onClick={this.click1}>A: {this.state.A}</button>
36. )
37. }
38. }
39. ReactDOM.render(<App />, document.getElementById('root'))
40. </script>
41. </body>
42. </html>

#### 1.3.10 数组set去重

console.log([...new Set([1, 2, 3, 3, 4, 1, 5])]);

#### 1.3.11 跨域

浏览器获取资源的服务器的通信协议、主机域名、端口可能和 js 脚本中访问的资源目标的通信协议、主机域名、端口不一致，这样浏览器认为存在安全隐患。

同时服务器对来自其他机器的 js 脚本的资源请求也是存疑的。

为了确认是否安全，浏览器在一个正式请求发送之前，先发一个空数据的请求，询问服务器的跨域可访问性。

如果是服务器认可的请求来源，则允许通过；否则拒绝请求，浏览器报跨域警告。

### 1.4 益科2023.02.14

#### 1.4.1 数组去重

1. const arr = [1, '1', 0, '0', 1, 0, '1', '0'];
2. *// newArr内容：[1, '1', 0, '0']*
3. console.log('newArr: ', Array.from(new Set(arr)));

#### 1.4.2 实现方法 getIds

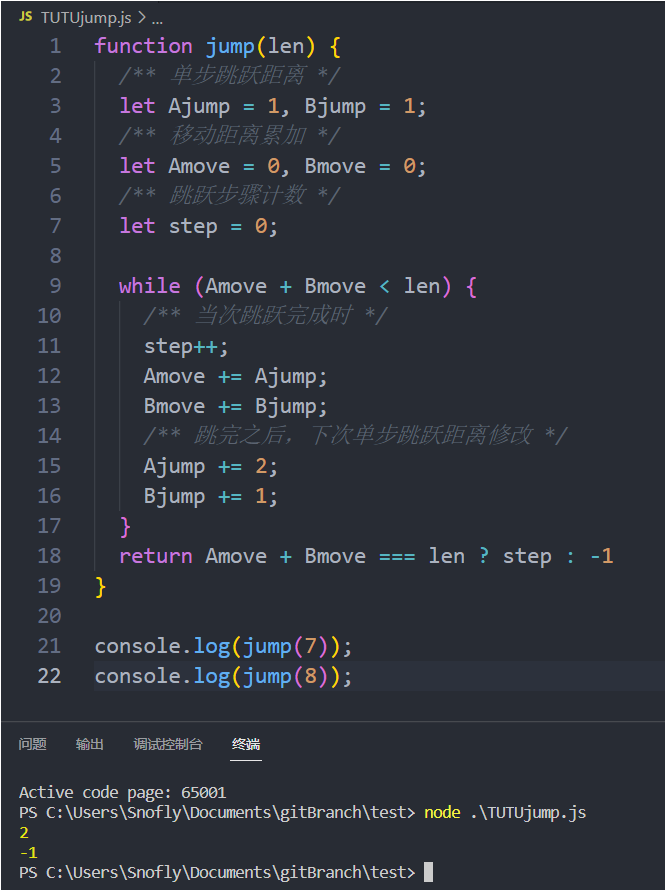
1. // 从数组 customers 中查找 所有年龄（age）小于 30 的人并将其 customerId 重新组成一个新的数组返回，达到如下代码执行效果：
2. const customers = [
3. { name: 'Jack', age: 23, customerId: '001' },
4. { name: 'Jam', age: 15, customerId: '002' },
5. { name: 'Lilei', age: 36, customerId: '003' },
6. { name: 'Hanmei', age: 47, customerId: '004' },
7. { name: 'Dav', age: 29, customerId: '005' },
8. ];
9. // ids内容：['001','002','005']
10. console.log('ids: ',
11. customers.filter(i => i.age < 30).map(i => i.customerId)
12. )

#### 1.4.3 实现方法 isSimilar

1. *// 用于对比两个字符串是否相似，相似返回 true ，不相似返回 false，达到如下代码的执行结果：字符串包含的字符种类以及每种字符的个数均相等即为相似。*
2. const isSimilar = function (str) {
3. if (str.length !== this.length) return false
4. const map = new Map()
5. for (const c of this) {
6. map.set(c, (map.get(c) || 0) + 1)
7. }
8. for (const c of str) {
9. if (!map.get(c)) return false
10. map.set(c, map.get(c) - 1)
11. }
12. return true
13. }
14. String.prototype.isSimilar = isSimilar;
15. // test case
16. const str1 = 'ababc';
17. const str2 = 'babac';
18. const str3 = 'abccc';
19. const str4 = 'abcccw';
20. *// const str5 = 'cab c';*
21. const str5 = 'babcc';
22. const str6 = 'abccc';
23. console.log(str1.isSimilar(str2));  *// 输出 true*
24. console.log(str2.isSimilar(str1));  *// 输出 true*
25. console.log(str1.isSimilar(str3));  *// 输出 false*
26. console.log(str1.isSimilar(str4));  *// 输出 false*
27. console.log(str5.isSimilar(str6));  *// 输出 false*

### 1.5 水木羽林2023.02.20

#### 1.5.1 笔试 相遇问题



#### 1.5.2 笔试 排序



#### 1.5.3 面试URL回车

略。同 1.1.3 项。

### 1.6 奥本未来2023.02.20

#### 1.6.1 笔试1

一、阅读下面的源码，回答问题

 1    var a = [];

 2    for (var i = 0; i < 10; i++) {

 3        a.push(

 4            function() {

 5                console.log(i);

 6            }

 7        );

 8    }

 9

10    for (var i = 0; i < 10; i++) {

11        a[i]();

12    }

**这段代码的输出是什么？解释一下为什么会有这样的输出。**

输出：0123456789。

解释：因为var将变量i提升到全局作用域，11行执行函数时，访问的i是被10行修改了的全局变量。

我在下面改写一段代码来解释执行过程：

var a = [];

var i;

for (i = 0; i < 10; i++) {

  a.push(

    function () {

      console.log(i);

    }

  );

}

*/\* 数组元素是功能相同的函数, a[0] 和 a[9] 都一样\*/*

for (i = 0; i < 10; i++) {

  a[0]();

}

*/\*\* 变化i，可以看到实际上6行中访问的i就是全局变量i \*/*

for (i = 0; i < 10; i += 2) {

  a[0]();

}

**如果把10-12行的循环替换为：a.forEach(f => f()); 结果会有什么变化，为什么？**

输出：十个10。

解释：因为forEach遍历的时候，没有修改全局变量i，于是log时的i始终等于10。

**如果想要只修改第4-6行，实现输出结果和第2题的输出一致，可以怎么做？**

思路：将log放到异步队列中，当它开始执行时，变量i已经赋值为10了。

代码：

1. var a = [];
2. for (var i = 0; i < 10; i++) {
3. a.push(
4. function () {
5. setTimeout(() => {
6. console.log(i);
7. }, 0);
8. }
9. );
10. }
11. for (var i = 0; i < 10; i++) {a[i]();}

#### 1.6.2 笔试2

JSX文件可以在JS文件中写类似HTML的标签。

**浏览器引擎本身是不支持JSX的，我们编写的JSX文件为什么可以在浏览器中运行？**

答：因为JSX文件会先被转化为JS代码，再给浏览器执行。这个转换过程是打包的时候进行的。

#### 1.6.3 笔试3

一个React程序的执行流程大概是什么样的？从JSX中编写的标签到文档中的一个DOM元素，中间会有哪些转换步骤？

流程：

触发更新

render阶段：计算更新会造成的副作用

commit阶段：在宿主环境执行副作用

通过修改组件 state 触发 fiber tree 更新，在 render 阶段，React 会计算 DOM 的更新，并将所有需要更新的 fiber 整理成一个 effect list，在 commit 阶段中， React 会遍历 effect list 执行所有的副作用，期间会执行更新相关的生命周期、挂载 DOM 等等。

JSX到DMO的转换：

1）Babel 会把 JSX 转译成一个名为 React.createElement() 函数调用。

2）React.createElement() 会预先执行一些检查，然后创建称为 React 元素的对象。

3）React 读取这些对象，然后使用它们来构建 DOM 以及保持随时更新：根据 react element 生成 fiber tree，再根据 fiber tree 生成 dom tree。

#### 1.6.4 面试

##### 1.6.4.1 题目：原生js实现拖动。

1. <html>
2. <head>
3. <style>
4. .box {
5. width: 50px;
6. height: 50px;
7. border: 1px solid black;
8. position: absolute;
9. }
10. </style>
11. </head>
12. <body>
13. <div class="box" id="box">
14. </div>
15. <script>
16. const boxNode = document.getElementById('box')
17. function setPosition(x, y) {
18. boxNode.style.top = x
19. boxNode.style.left = y
20. }
21. let isCatch = false
22. function changeCatchState() {
23. isCatch = !isCatch
24. console.log(isCatch);
25. }
26. function moveMouse(e) {
27. if (isCatch && e.toElement === document.body) {
28. setPosition(e.offsetY, e.offsetX)
29. }
30. }
31. window.removeEventListener('click', changeCatchState)
32. window.removeEventListener('mousemove', moveMouse)
33. window.addEventListener('click', changeCatchState)
34. window.addEventListener('mousemove', moveMouse)
35. </script>
36. </body>
37. </html>

##### 1.6.4.2 useEffect 无第二参数，空数组，有值数组，值有ref的数组。

1、无第二参数——每次渲染都执行；

2、空数组——只在第一次渲染的时候执行；

3、有值数组——第一次渲染时执行，并在依赖项变化时重新执行；

4、依赖项为 ref —— 无效。

5、依赖项为 ref.current —— 可以改变，但不主动出发重新渲染。

##### 1.6.4.3 虚拟DOM。

虚拟 DOM 的主要作用是提高渲染性能。当需要更新 UI 时，框架会先更新虚拟 DOM，然后将其与真实 DOM 进行比较，只更新那些发生变化的部分。

##### 1.6.4.4 函数组件和hooks。

以函数返回值的形式创建组件，由于函数本身的数据是一次性的，所以需要 hooks 保存和管理状态。

##### 1.6.4.5 useState 同步。

“在 React 中，useState 的 setState 方法默认是异步的。这意味着，在调用 setState 之后，组件不会立即重新渲染。相反，React 会将更新批量处理并稍后进行一次性渲染。

在某些情况下，您可能希望在调用 setState 之后立即重新渲染组件。例如，如果您正在使用 useState 来跟踪用户输入，您可能希望在用户输入每次更改时立即更新 UI。”

1. const MyState = () => {
2. const [count, setCount] = useState(0)
3. console.log('render', count);
4. function handleClick() {
5. flushSync(() => {setCount(p => p + 1)})
6. console.log('count:', count);
7. flushSync(() => {setCount(p => p + 1)})
8. }
9. return  <Button onClick={handleClick}>{count}</Button>
10. }

##### 1.6.4.6 兄弟组件通信。

1、间接通信：

store/context、props.state、缓存、服务端

2、直接通信：postmessage、websocket

##### 1.6.4.7 组件key的作用。

准确对比虚拟 DOM 和实际 DOM 树（或更旧一点对虚拟 DOM）之间的差异，减少无用的 DOM 操作。

对部分 dom 只需要替换属性，对部分 dom 需要整体替换，对部分 dom 需要删除，对部分 dom 需要新增。

##### 1.6.4.8 在if中使用useState。

在 React 中，在 if 语句中定义 useState 会直接报错，这是因为 React 使用 单链表 来管理 Hooks。

在 React 组件渲染过程中，会执行以下步骤：

1、初始化所有 Hooks

2、调用 render 函数生成虚拟 DOM

3、将虚拟 DOM 转换为真实 DOM

在初始化 Hooks 阶段，React 会创建一个单链表来存储所有 Hooks。每个 Hook 在链表中都包含一个节点，该节点包含 Hook 的值和更新函数。

当组件重新渲染时，React 会遍历链表并更新每个 Hook 的值。

##### 1.6.4.9 伪元素的使用。

伪元素是一个附加至选择器末的关键词，允许你对被选择元素的特定部分修改样式。