

# 如何实现渲染十万条数据？

## 前言

在我的项目中，有时需要展示上万条甚至十万条数据，而如何在保证性能的前提下高效地渲染这些数据成为了我面临的难题。在经过一段时间的摸索和实践后，我积累了一些经验和技巧，现在我将这些分享给大家。

## 不进行任何操作（一次性渲染）

这种方法核心理念是尽可能减少对DOM的操作，特别是在处理大量数据时。DOM操作通常是相对昂贵的，因为它们会触发浏览器的回流和重绘。

回流指的是浏览器根据最新的 DOM 树计算元素的几何属性，然后再进行布局，而重绘则是将元素的像素信息绘制到屏幕上。

我们用两个输出语句来看看 `js代码运行时间` 和 `总渲染时间`，把它们进行对比：

- 在 JS 的 `Event Loop` 中，当JS引擎所管理的执行栈中的事件以及所有微任务事件全部执行完后，才会触发渲染线程对页面进行渲染
- 第一个 `console.log` 的触发时间是在页面进行渲染之前，此时得到的间隔时间为JS运行所需要的时间
- 第二个 `console.log` 是放到 `setTimeout` 中的，它的触发时间是在渲染完成，在下一次 `Event Loop` 中执行的

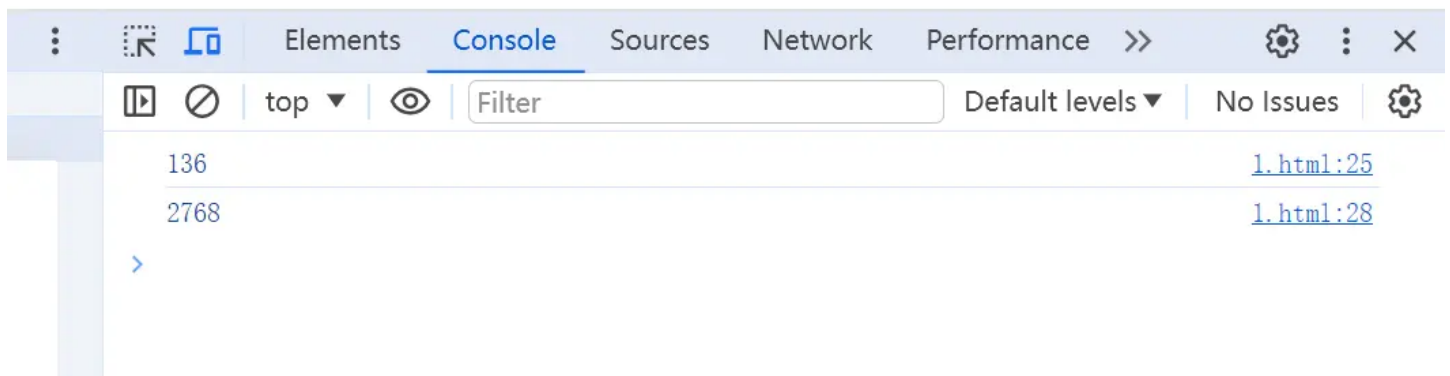
```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Document</title>
7 </head>
8 <body>
9   <!-- 创建一个空的无序列表，用于存放随机数字 -->
10  <ul id="container"></ul>
11
12  <script>
13    // 获取代码执行前的时间戳
14    let now = Date.now();
15
16    // 定义要生成的随机数字的总数
```

```

17     const total = 100000;
18
19     // 获取页面中的<ul>元素
20     let ul = document.getElementById('container');
21
22     // 循环生成指定数量的随机数字，并添加到<ul>元素中作为<li>元素
23     for (let i = 0; i < total; i++) {
24         // 创建<li>元素
25         let li = document.createElement('li');
26         // 将<li>元素的文本内容设置为一个0到total之间的随机整数
27         li.innerText = ~~(Math.random() * total);
28         // 将<li>元素添加到<ul>元素中
29         ul.appendChild(li);
30     }
31
32     // 计算执行代码所花费的时间，并输出到控制台
33     console.log(Date.now() - now);
34
35     // 设置一个延时，以便在执行完所有代码后再次输出时间
36     setTimeout(() => {
37         console.log(Date.now() - now);
38     });
39 </script>
40
41 </body>
42 </html>
43

```

对比时间我们可以看见，渲染的时间也太长了，所以我们得想想办法，有没有什么方法可以优化一下渲染的过程



## setTimeout（使用定时器）

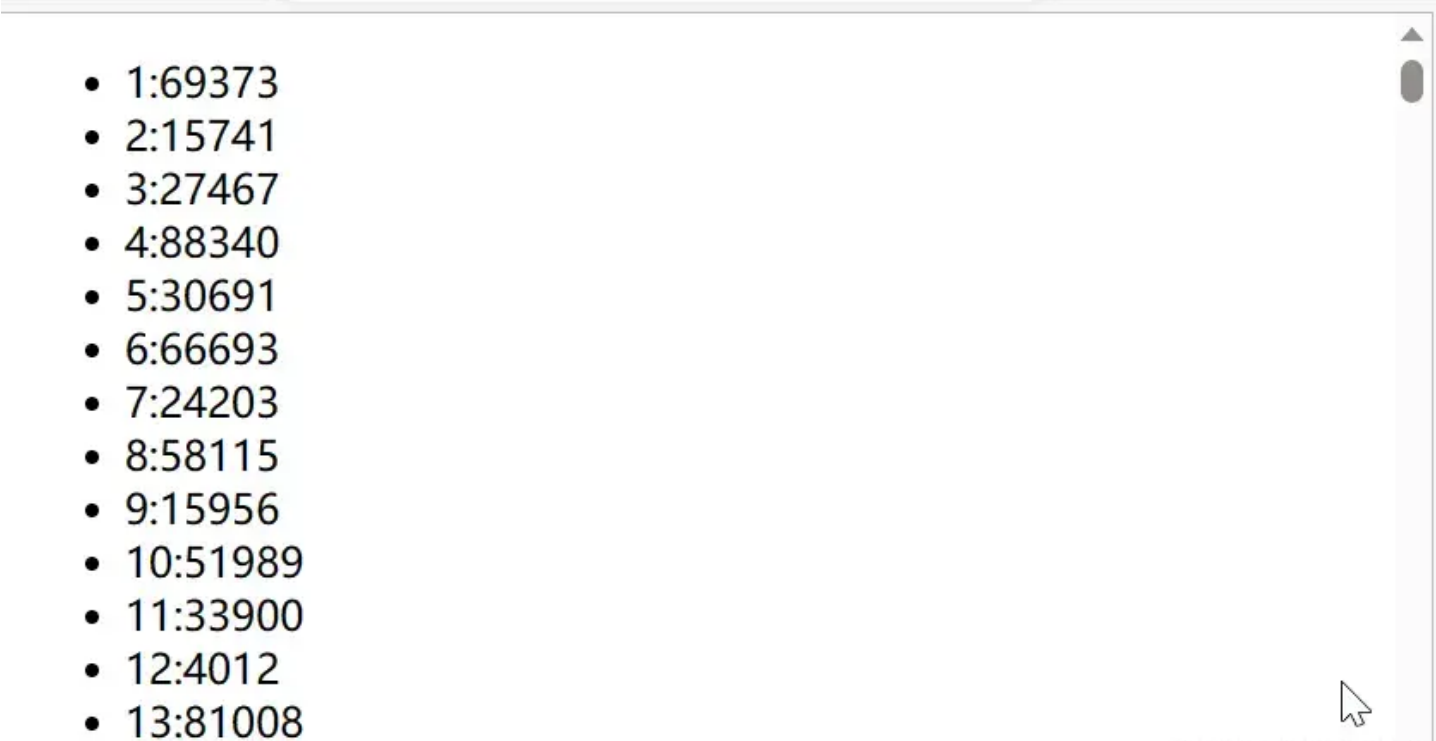
setTimeout 的工作原理是将任务放入事件队列中，等待当前执行栈清空后执行。这种方法可以用来延迟执行某些耗时的操作，以允许浏览器在操作之间进行页面重绘。

我们根据上面代码改编一下：

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Document</title>
7 </head>
8 <body>
9
10   <!-- 空的无序列表，用于容纳动态生成的列表项 -->
11   <ul id="container"></ul>
12
13   <script>
14
15     // 总共需要生成的列表项数量
16     const total = 1000000
17     // 获取ul元素
18     let ul = document.getElementById('container')
19     // 每次添加的列表项数量
20     let once = 20
21     // 计算总共需要执行多少次循环
22     let page = total / once
23
24     // 定义递归函数用于分片加载列表项
25     function loop(curTotal) {
26       // 如果当前已加载的列表项数量小于等于0，则结束递归
27       if (curTotal <= 0) return
28
29       // 计算当前分片应该加载的列表项数量
30       let pageCount = Math.min(curTotal, once)
31
32       // 设置一个异步任务，模拟分片加载效果
33       setTimeout(() => {
34         // 循环生成列表项并添加到ul元素中
35         for (let i = 0; i < pageCount; i++) {
36           let li = document.createElement('li')
37           // 生成随机数作为列表项的文本内容
38           li.innerText = ~~(Math.random() * total)
39           ul.appendChild(li)
40         }
41         // 继续加载下一个分片
42         loop(curTotal - pageCount)
43       }, 0)
44     }
```

```
45      // 开始加载列表项
46      loop(total)
47
48  </script>
49 </body>
50 </html>
```

现在数据有了，但是我发现如果快速往下滑动的时候，会有类似 **闪屏** 的效果

- 
- 1:69373
  - 2:15741
  - 3:27467
  - 4:88340
  - 5:30691
  - 6:66693
  - 7:24203
  - 8:58115
  - 9:15956
  - 10:51989
  - 11:33900
  - 12:4012
  - 13:81008

这是因为原因：

- **setTimeout的执行时间是不确定的**，setTimeout任务被放入事件队列中，只有在主线程执行完毕后才会检查并执行事件队列中的任务。
- **屏幕刷新频率受分辨率和屏幕尺寸影响**，而setTimeout只能设置一个固定的时间间隔，这个时间不一定和屏幕刷新时间相同。

## requestAnimationFrame + fragment (时间分片)

requestAnimationFrame 是一种**在浏览器重绘之前执行操作的方法**，通常用于动画和其他需要高性能的场景。它可以**保证回调函数在每次页面重绘之前执行**，从而避免了由于执行操作而导致的页面抖动问题。

更新后的代码如下：

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
```

```
4     <meta charset="UTF-8">
5     <meta name="viewport" content="width=device-width, initial-scale=1.0">
6     <title>Document</title>
7 </head>
8
9 <body>
10     <ul id="container"></ul> <!-- 页面中的一个空的无序列表，用于存放随机数字 -->
11
12     <script>
13
14         // 定义总共要生成的随机数字的数量
15         const total = 10000;
16
17         // 获取页面中的列表元素
18         let ul = document.getElementById('container');
19
20         // 定义每次生成的随机数字数量
21         let once = 20;
22
23         // 计算总共需要执行多少次生成随机数字的操作
24         let page = total / once;
25
26         // 定义一个递归函数，用于生成随机数字并添加到列表中
27         function loop(curTotal) {
28             // 如果已经生成了所有的随机数字，则结束递归
29             if (curTotal <= 0) return;
30
31             // 计算本次要生成的随机数字数量，不能超过剩余的总数量
32             let pageCount = Math.min(curTotal, once);
33
34             // 使用 requestAnimationFrame 在下一次页面重绘前执行生成随机数字的操作
35             window.requestAnimationFrame(() => {
36                 // 创建一个虚拟的文档片段，用于提高性能
37                 let fragment = document.createDocumentFragment();
38
39                 // 循环生成随机数字并添加到文档片段中
40                 for (let i = 0; i < pageCount; i++) {
41                     let li = document.createElement('li');
42                     li.innerText = ~~(Math.random() * total); // 生成一个随机数字
并设置为列表项的文本内容
43                     fragment.appendChild(li); // 将列表项添加到文档片段中
44                 }
45
46                 // 将文档片段中的所有列表项一次性添加到页面中的列表中
47                 ul.appendChild(fragment);
48
49                 // 继续递归调用 loop 函数，生成剩余数量的随机数字
```

```
50         loop(curTotal - pageCount);
51     });
52 }
53
54 // 初始调用 loop 函数，开始生成随机数字
55 loop(total);
56
57 </script>
58
59 </body>
60
61 </html>
62
```

## 补充

`requestAnimationFrame` (RAF) 和 `setTimeout` 都是用于控制 JavaScript 代码执行时间的方法，但它们有一些关键的区别：

### 1. 执行时机：

- `setTimeout`：将任务放入事件队列，并在指定的延迟时间后执行任务。但是，由于 JavaScript 是单线程的，如果主线程忙于执行其他任务，`setTimeout` 中的任务可能会被延迟执行。
- `requestAnimationFrame`：在浏览器下一次重绘之前执行任务。它会与浏览器的渲染循环同步，以确保在页面重新绘制时执行任务，通常是每秒 60 次（60Hz）。

### 2. 性能优化：

- `setTimeout`：尽管可以设置延迟时间，但执行时间并不是可靠的。这可能导致性能问题，尤其是在需要频繁执行的动画或渲染任务中。
- `requestAnimationFrame`：浏览器会自动优化执行时间，通常能够更好地与页面渲染协调，避免因执行任务而导致的卡顿或闪烁。

### 3. 用途：

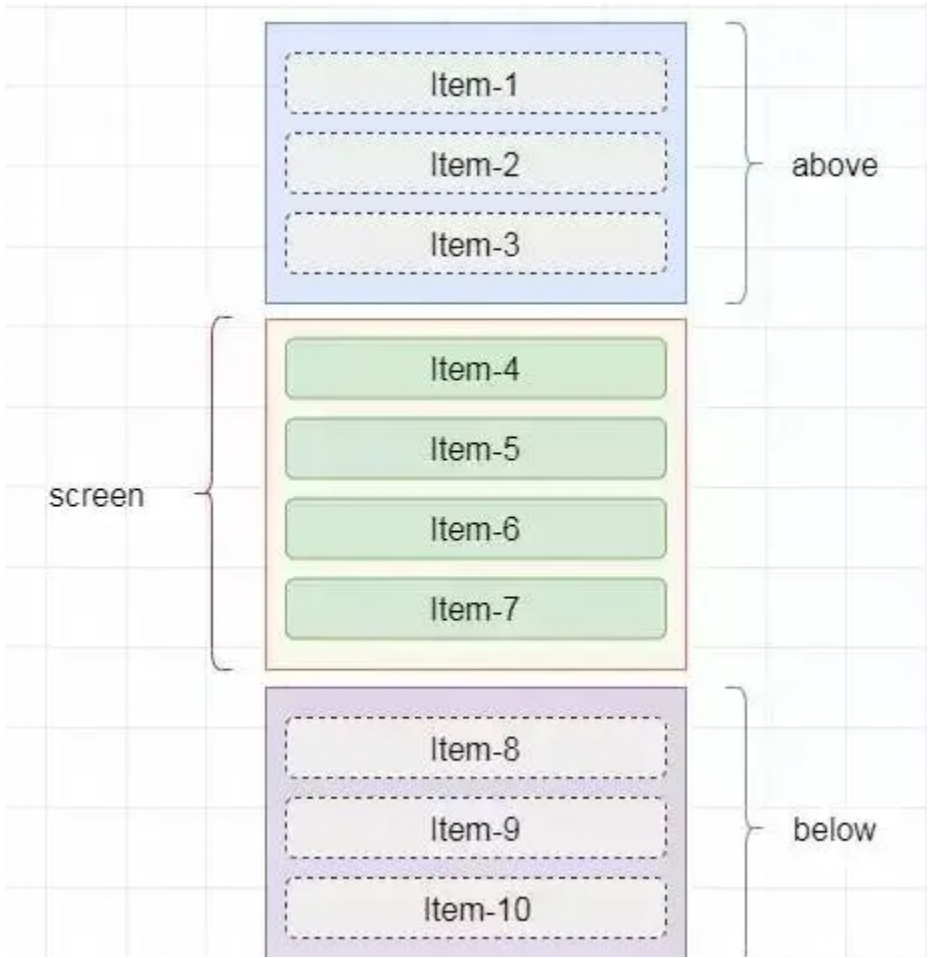
- `setTimeout`：适用于一般性的延迟执行任务，例如定时器功能或简单的异步操作。
- `requestAnimationFrame`：主要用于执行与页面渲染相关的任务，如动画或需要频繁更新的 UI 效果。它可以确保在每次页面重绘时都进行优化的任务执行，以提供更流畅的用户体验。

## 虚拟列表

虚拟列表是一种专门为处理大量数据而设计的优化方案。它的核心思想是**只渲染当前可视区域内的数据，而不是一次性渲染所有数据**。通过动态计算可视区域内需要展示的数据，可以有效地减少渲染时

间和资源消耗，提高页面的响应速度和流畅度。

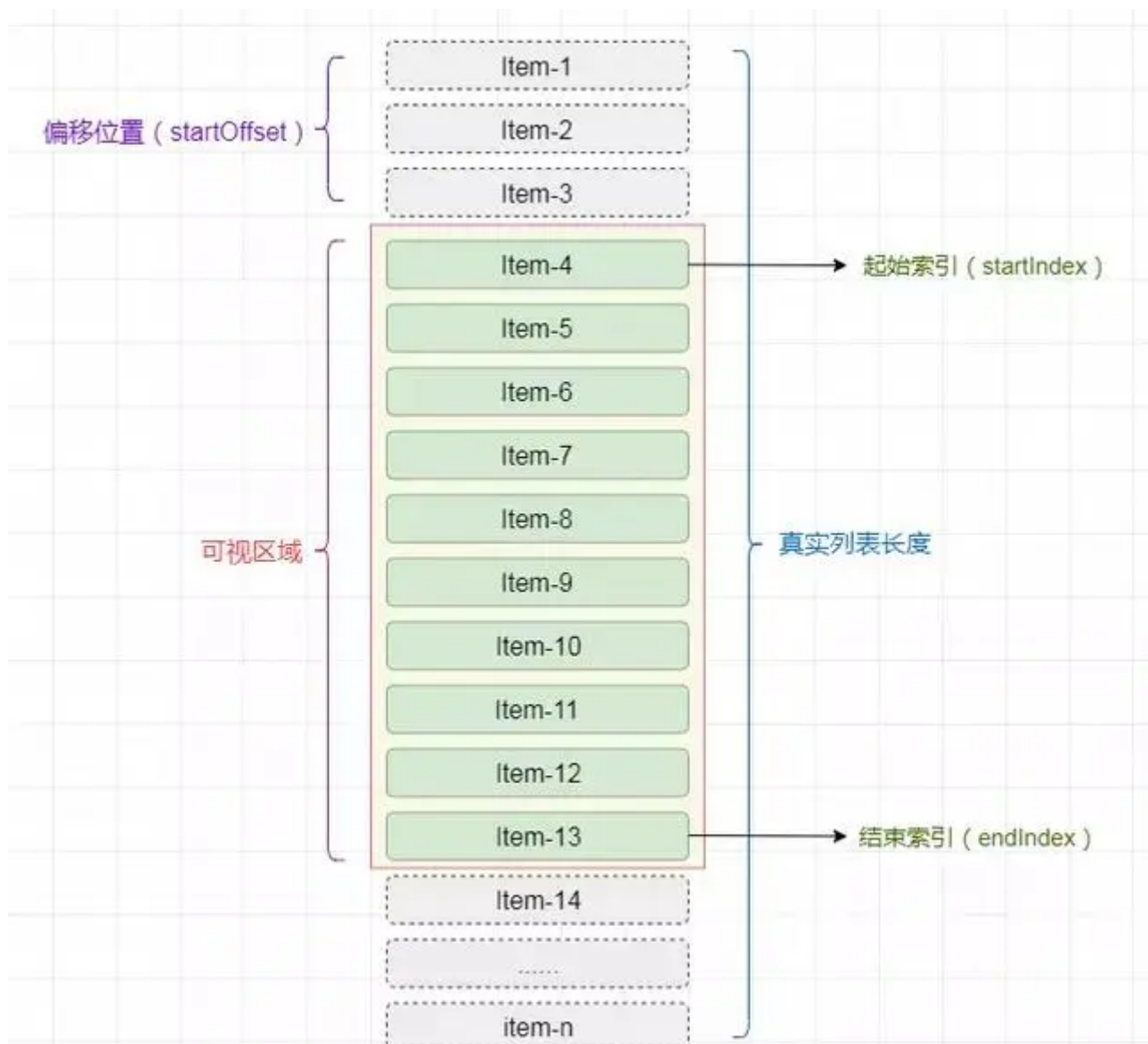
如图：



虚拟列表的实现通常包括以下几个步骤：

- **计算可视区域**：获取可视区域的高度，并根据滚动位置计算可视区域内的起始和结束索引。
- **动态渲染**：根据计算得到的起始和结束索引，从数据源中获取相应的数据进行渲染。
- **滚动优化**：监听滚动事件，当滚动位置发生变化时，重新计算可视区域并更新渲染的数据。

实现示意图：



```
1 <!DOCTYPE html>
2 <html lang="en">
3
4 <head>
5   <meta charset="UTF-8">
6   <meta name="viewport" content="width=device-width, initial-scale=1.0">
7   <title>Document</title>
8   <!-- 导入 Vue.js -->
9   <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
10  <style>
11    * {
12      margin: 0;
13      padding: 0;
14    }
15
16    .v-scroll {
17      width: 300px;
18      height: 400px;
19      border: 1px solid #000;
```



```

20         margin: 100px 0 0 100px;
21         overflow-y: scroll;
22     }
23
24     li {
25         list-style: none;
26         padding-left: 20px;
27         height: 40px;
28         line-height: 40px;
29         border-bottom: 1px solid #d5cece;
30         box-sizing: border-box;
31     }
32 </style>
33 </head>
34
35 <body>
36     <div id="app">
37         <!-- 滚动容器 -->
38         <div class="v-scroll" ref="scrollBox" @scroll="doScroll">
39             <ul>
40                 <!-- 使用 v-for 渲染列表项 -->
41                 <li v-for="(item, index) in currentList">{{index + 1}} --
42                 {{item}}</li>
43             </ul>
44         </div>
45
46     <script>
47         // 导入 Vue.js 的相关模块
48         const { createApp, ref, onMounted, computed } = Vue
49
50
51         // 创建 Vue 应用
52         createApp({
53             setup() {
54                 // 创建对所有数据的引用
55                 const allList = ref([]) // 所有的数据
56
57                 // 模拟接口请求，获取数据
58                 const getAllList = (count) => { // 接口请求
59                     for (let i = 0; i < count; i++) {
60                         allList.value.push(`我是列表${allList.value.length + 1}项`
61                     )
62                 }
63                 getAllList(300)
64

```

```

65 // -----
66
67 // 定义变量和函数
68 const boxHeight = ref(0) // 可视区域高度
69 const itemHeight = ref(40) // 每一项的高度
70 const scrollBox = ref(null) // 可视区域容器
71
72 // 当组件挂载完成后，获取滚动容器的高度
73 onMounted(() => {
74     boxHeight.value = scrollBox.value.clientHeight
75 })
76
77 // 计算可视区域内显示的列表项数量
78 const itemNum = computed(() => {
79     return Math.floor(boxHeight.value / itemHeight.value) + 2
80 })
81
82 // 可视区域内的第一项的索引
83 const startIndex = ref(0)
84
85 // 页面滚动事件处理函数
86 const doScroll = () => {
87     // 获取当前滚动位置的索引
88     const index = Math.floor(scrollBox.value.scrollTop /
itemHeight.value)
89     // 如果滚动位置没有改变，则不做处理
90     if (index === startIndex.value) return
91     // 更新可视区域内的第一项索引
92     startIndex.value = index;
93 }
94
95 // 计算可视区域内的最后一项下标
96 const endIndex = computed(() => {
97     let index = startIndex.value + itemNum.value * 2
98     if (!allList.value[index]) {
99         index = allList.value.length - 1
100     }
101     return index
102 })
103
104 // 计算当前应该渲染的列表项数据
105 const currentList = computed(() => {
106     let index = 0
107     if (startIndex.value <= itemNum.value) {
108         index = 0
109     } else {
110         index = startIndex.value - itemNum.value

```

```
110         }
111         return allList.value.slice(index, endIndex.value + 1)
112     })
113
114     // 返回组件中需要使用的数据和方法
115     return {
116         allList,
117         currentList,
118         boxHeight,
119         itemHeight,
120         scrollBox,
121         doScroll
122     }
123 }
124 }).mount('#app')
125 </script>
126
127 </body>
128 </html>
```

总的来说，这段代码实现了一个简单的虚拟滚动列表功能。