



下载APP



31 | 虚拟DOM (下) : 想看懂虚拟DOM算法, 先刷个算法题

2021-12-31 大圣

《玩转Vue 3全家桶》

课程介绍 >

**讲述：大圣**

时长 09:42 大小 8.89M



你好，我是大圣。上一讲我们仔细分析了 Vue 中虚拟 DOM 如何执行的，整体流程就是树形结构的 diff 计算，但是在 diff 的计算过程中，如何高效计算虚拟 DOM 属性的变化，以及如何更新数组的子元素，需要一些算法知识的补充。

给你提前划个重点，今天我们将讲到如何使用位运算来实现 Vue 中的按需更新，让静态的节点可以越过虚拟 DOM 的计算逻辑，并且使用计算最长递增子序列的方式，来实现队伍的高效排序。我们会剖析 Vue 框架源码，结合对应的 LeetCode 题，帮助你掌握算法的核心原理和实现。



位运算

前面也复习了，在执行 diff 之前，要根据需要判断每个虚拟 DOM 节点有哪些属性需要计算，因为无论响应式数据怎么变化，静态的属性和节点都不会发生变化。

所以我们看每个节点 diff 的时候会做什么，在 renderer.ts 代码文件中就可以看到代码，主要就是通过虚拟 DOM 节点的 patchFlag 树形判断是否需要更新节点。

方法就是使用 & 操作符来判断操作的类型，比如 patchFlag & PatchFlags.CLASS 来判断当前元素的 class 是否需要计算 diff；shapeFlag & ShapeFlags.ELEMENT 来判断当前虚拟 DOM 是 HTML 元素还是 Component 组件。这个“&”其实就是位运算的按位与。

[复制代码](#)

```
1 // class
2 // this flag is matched when the element has dynamic class bindings.
3 if (patchFlag & PatchFlags.CLASS) {
4   if (oldProps.class !== newProps.class) {
5     hostPatchProp(el, 'class', null, newProps.class, isSVG)
6   }
7 }
8
9 // style
10 // this flag is matched when the element has dynamic style bindings
11 if (patchFlag & PatchFlags.STYLE) {
12   hostPatchProp(el, 'style', oldProps.style, newProps.style, isSVG)
13 }
14 if (shapeFlag & ShapeFlags.ELEMENT) {
15   processElement(
16     n1,
17     n2,
18     container,
19     anchor,
20     parentComponent,
21     parentSuspense,
22     isSVG,
23     slotScopeIds,
24     optimized
25   )
26 } else if (shapeFlag & ShapeFlags.COMPONENT) {
27   processComponent(
28     n1,
29     n2,
30     container,
31     anchor,
32     parentComponent,
33     parentSuspense,
34     isSVG,
35     slotScopeIds,
```

```
36     optimized
37   )
38 }
```

上面的代码中 `&` 就是按位与的操作符，这其实是二进制上的计算符号，所以我们首先要了解一下什么是二进制。

我们日常使用的数字都是十进制数字，比如数字 13 就是 $1 \times 10 + 3$ 的运算结果，每个位置都是代表 10 的 n 次方。13 也可以使用二进制表达，因为二进制每个位置只能是 0 和 1 两个数字，每个位置代表的是 2 的 n 次方，13 在二进制里是 1101，就是 $1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$ 。

而在 JavaScript 中我们可以很方便地使用 `toString(2)` 的方式，把十进制数字转换成二进制。运算的概念很简单，就是在二进制上的“与”和“或”运算：

[复制代码](#)

```
1  (13).toString(2) // 1101
2
3  0 & 0 // 0
4  0 & 1 // 0
5  1 & 0 // 0
6  1 & 1 // 1
7
8  0 | 0 // 0
9  0 | 1 // 1
10 1 | 0 // 1
11 1 | 1 // 1
12
13 1 << 2 // 1左移动两位，就是100 就是1*2平方 = 4
```

二进制中，我们每个位置只能是 0 或者 1 这两个值，`&` 和 `|` 的概念和 JavaScript 中的 `&&` 和 `||` 保持一致。两个二进制的 `&` 运算就是只有两个二进制位置都是 1 的时候，结果是 1，其余情况运算结果都是 0；`|` 是按位置进行“或”运算，只有两个二进制位置都是 0 的时候，结果是 0，其余情况运算结果都是 1；并且，还可以通过左移 `<<` 和右移 `>>` 操作符，实现乘以 2 和除以 2 的效果。

由于**这些都是在二进制上的计算**，运算的性能通常会比字符串和数字的计算性能要好，这也是很多框架内部使用位运算的原因。

这么说估计你不是很理解，我们结合一个 LeetCode 题看看为什么说二进制的位运算性能更好。

为什么位运算性能更好

我们来做一下 LeetCode231 题，题目描述很简单，判断数字 n 是不是 2 的幂次方，也就是说，判断数字 n 是不是 2 的整次方，比如 2、4、8。我们可以很轻松地写出 JavaScript 的解答， n 一直除以 2，如果有余数就是 false，否则就是 true：

[复制代码](#)

```
1 var isPowerOfTwo = function(n) {
2     if(n === 1) return true
3     while( n > 2 ){
4         n = n / 2
5         if(n % 2 !== 0) return false
6     }
7     return n===2
8
9 };
```

不过上面的解答我们可以用位运算来优化。

先来分析一下 2 的幂次方的特点。

2 的幂次方就是数字 1 左移动若干次，其余位置全部都是 0，所以 $n-1$ 就是最高位变成 0，其余位置都变成 1，就像十进制里的 $10000-1 = 9999$ 。这样， **n 和 $n-1$ 每个二进制位的数字都不一样，我们可以很轻松地用按位“与”来判断这个题的答案**，如果 $n \& n-1$ 是 0 的话，数字 n 就符合 2 的整次幂的特点：

[复制代码](#)

```
1 16
2 10000
3 16-1 = 15
4 01111
5 16&15 == 0
6
7 var isPowerOfTwo = function(n) {
8     return n>0 && (n & (n - 1)) === 0
9 };
```


所以我们使用位运算提高了代码的整体性能。

如何运用位运算

好，搞清楚为什么用位运算，我们回来看 diff 判断，如何根据位运算的特点，设计出权限的组合认证方案。

比如 Vue 中的动态属性，有文本、class、style、props 几个属性，我们可以使用二进制中的一个位置来表示权限，看下面的代码，**我们使用左移的方式分别在四个二进制上标记了 1，代表四种不同的权限，使用按位或的方式去实现权限授予。**

比如，一个节点如果 TEXT 和 STYLE 都需要修改，我们只需要使用 | 运算符就可以得到 flag1 的权限表示，这就是为什么 Vue 3 中针对虚拟 DOM 类型以及虚拟 DOM 需要动态计算 diff 的树形都做了标记，你可以在 [Vue 3 的源码](#)中看到下面的配置：

 复制代码

```
1  const PatchFlags = {
2    TEXT:1,          // 0001
3    CLASS: 1<<1,    // 0010
4    STYLE:1<<2,     // 0100
5    PROPS:1<<3      // 1000
6  }
7
8  const flag1 = PatchFlags.TEXT | PatchFlags.STYLE // 0101
9
10 // 权限校验
11
12 flag1 & PatchFlags.TEXT // 有权限，结果大于1
13 flag1 & PatchFlags.CLASS //没有权限 是0
```

最长递增子系列

然后就到了今天的重点：我们虚拟 DOM 计算 diff 中的算法了。

上一讲我们详细介绍了在虚拟 diff 计算中，如果新老子元素都是数组的时候，需要先做首尾的预判，如果新的子元素和老的子元素在预判完毕后，未处理的元素依然是数组，那么就需要对两个数组计算 diff，最终找到最短的操作路径，能够让老的子元素通过尽可能少的操作，更新成为新的子元素。

Vue 3 借鉴了 infero 的算法逻辑，就像操场上需要按照个头从低到高站好一样，我们采用的思路是先寻找一个现有队列中由低到高的队列，让这个队列尽可能的长，它们的相对位置不需要变化，而其他元素进行插入和移动位置，这样就可以做到尽可能少的操作 DOM。

所以如何寻找这个最长递增的序列呢？这就是今天的重点算法知识了，我们看 [LeetCode 第 300 题](#)，题目描述如下，需要在数组中找到最长底层的自序列长度：

[复制代码](#)

```
1 给你一个整数数组 nums，找到其中最严格递增子序列的长度。
2
3 子序列是由数组派生而来的序列，删除（或不删除）数组中的元素而不改变其余元素的顺序。
4 例如，[3,6,2,7] 是数组 [0,3,1,6,2,2,7] 的子序列。
5
6 =
7 输入：nums = [10,9,2,5,3,7,101,18]
8 输出：4
9 解释：最长递增子序列是 [2,3,7,101]，因此长度为 4。
```

首先我们可以使用动态规划的思路，通过每一步的递推，使用 dp 数组，记录出每一步操作的最优解，最后得到全局最优解。

在这个例子中，我们可以把 dp[i] 定义成 nums[0] 到 nums[i] 这个区间内，数组的最长递增子序列的长度，并且 dp 数组的初始值设为 1。

从左边向右递推，如果 $\text{nums}[i+1] > \text{nums}[i]$ ， $\text{dp}[i+1]$ 就等于 $\text{dp}[i] + 1$ ；如果 $\text{nums}[i+1] < \text{nums}[i]$ ，就什么都不需要干，这样我们在遍历的过程中，就能根据数组当前位置之前的最长递增子序列长度推导出 i+1 位置的最长递增子序列长度。

所以可以得到如下解法：

[复制代码](#)

```
1 /**
2  * @param {number[]} nums
3  * @return {number}
4  */
5 const lengthOfLIS = function(nums) {
6     let n = nums.length;
7     if (n == 0) {
```



```
8         return 0;
9     }
10    let dp = new Array(n).fill(1);
11    for (let i = 0; i < n; i++) {
12        for (let j = 0; j < i; j++) {
13            if (nums[j] < nums[i]) {
14                dp[i] = Math.max(dp[i], dp[j] + 1);
15            }
16        }
17    }
18    return Math.max(...dp)
19 }
```

由于我们需要两层循环，所以这个解法的时间复杂度是 n 的平方，这个解法其实已经不错了，但是还有更优秀的解法，也就是 Vue 3 中用到的算法：贪心 + 二分。

贪心 + 二分

我们再看一下这个题，贪心的思路就是在寻找最长递增的序列，所以，[1,3]要比[1,5]好，也就是说，在这个上升的序列中，我们要让上升速度尽可能变得慢，这样才有可能让后面的元素尽可能也递增。

我们可以创建一个 arr 数组，用来保存这种策略下的最长递增子序列。

如果当前遍历的 $nums[i]$ 大于 arr 的最后一个元素，也就是大于 arr 的最大值时，我们把 $nums[i]$ 追加到后面即可，否则我们就在 arr 中**寻找一个第一个大于 $num[i]$ 的数字并替换它**。因为 arr 是递增的数列，所以在寻找插入位置的时候，我们可以使用二分查找的方式，把整个算法的复杂度变成 $O(n \lg n)$ 。

下面的代码就是贪心 + 二分的解法，我们可以得到正确的最长递增子序列的长度：

```
1  /**
2   * @param {number[]} nums
3   * @return {number}
4   */
5  const lengthOfLIS = function(nums) {
6      let len = nums.length
7      if (len <= 1) {
8          return len
9      }
10 }
```

[复制代码](#)

```
11     let arr = [nums[0]]
12     for (let i = 0; i < len; i++) {
13         // nums[i] 大于 arr 尾元素时，直接追加到后面，递增序列长度+1
14         if (nums[i] > arr[arr.length - 1]) {
15             arr.push(nums[i])
16         } else {
17             // 否则，查找递增子序列中第一个大于numsp[i]的元素 替换它
18             // 递增序列，可以使用二分查找
19             let left = 0
20             let right = arr.length - 1
21             while (left < right) {
22                 let mid = (left + right) >> 1
23                 if (arr[mid] < nums[i]) {
24                     left = mid + 1
25                 } else {
26                     right = mid
27                 }
28             }
29             arr[left] = nums[i]
30         }
31     }
32     return arr.length
    }
```

但是贪心 + 二分的这种解法，现在只能得到最长递增子序列的长度，但是最后得到的 arr 并不一定是最长递增子序列，因为我们移动的 num[i]位置可能会不正确，只是得到的数组长度是正确的，所以我们需要对这个算法改造一下，把整个数组复制一份之后，最后也能得到正确的最长递增子序列。

具体代码怎么写呢？我们来到 Vue 3 的 renderer.ts 文件中，函数 `getSequence` 就是用来生成最长递增子序列，看下面的代码：

[复制代码](#)

```
1 // https://en.wikipedia.org/wiki/Longest_increasing_subsequence
2 function getSequence(arr: number[]): number[] {
3     const p = arr.slice() //赋值一份arr
4     const result = [0]
5     let i, j, u, v, c
6     const len = arr.length
7     for (i = 0; i < len; i++) {
8         const arrI = arr[i]
9         if (arrI !== 0) {
10             j = result[result.length - 1]
11             if (arr[j] < arrI) {
12                 p[i] = j // 存储在result最后一个索引的值
13                 result.push(i)
```



```
14         continue
15     }
16     u = 0
17     v = result.length - 1
18     // 二分查找, 查找比arrI小的节点, 更新result的值
19     while (u < v) {
20         c = (u + v) >> 1
21         if (arr[result[c]] < arrI) {
22             u = c + 1
23         } else {
24             v = c
25         }
26     }
27     if (arrI < arr[result[u]]) {
28         if (u > 0) {
29             p[i] = result[u - 1]
30         }
31         result[u] = i
32     }
33 }
34 }
35 u = result.length
36 v = result[u - 1]
37 // 查找数组p 找到最终的索引
38 while (u-- > 0) {
39     result[u] = v
40     v = p[v]
41 }
42 return result
43 }
```

这段代码就是 Vue 3 里的实现, result 存储的就是长度是 i 的递增子序列最小末位置的索引, 最后计算出最长递增子序列。

我们得到 increasingNewIndexSequence 队列后, 再去遍历数组进行 patch 操作就可以实现完整的 diff 流程了:

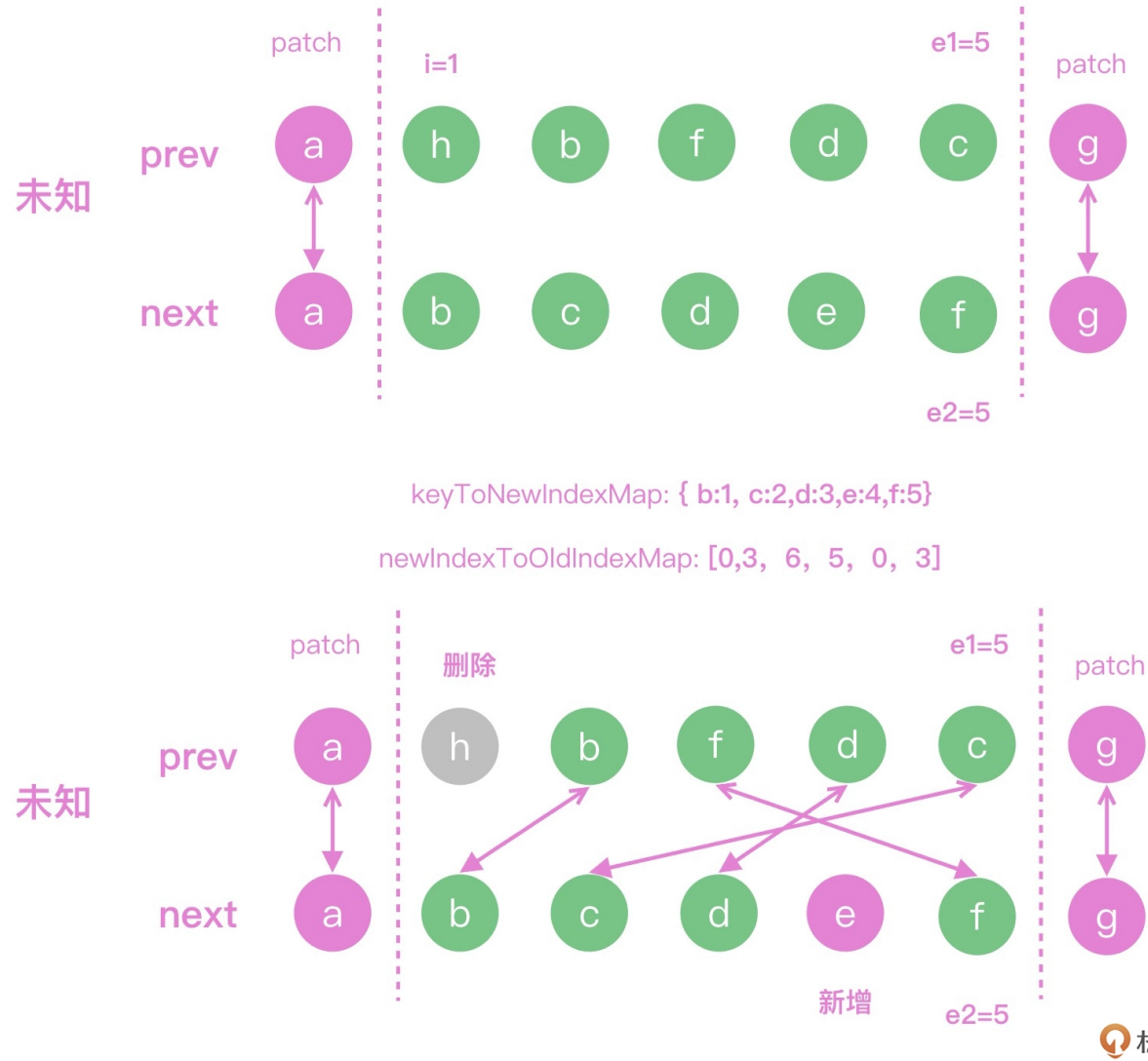
[复制代码](#)

```
1     for (i = toBePatched - 1; i >= 0; i--) {
2         const nextIndex = s2 + i
3         const nextChild = c2[nextIndex] as VNode
4         const anchor =
5             nextIndex + 1 < l2 ? (c2[nextIndex + 1] as VNode).el : parentAnchor
6         if (newIndexToOldIndexMap[i] === 0) {
7             // mount new
8             patch(
9                 null,
```

```
10         nextChild,  
11         container,  
12         anchor,  
13         parentComponent,  
14         parentSuspense,  
15         isSVG,  
16         slotScopeIds,  
17         optimized  
18     )  
19 } else if (moved) {  
20     // move if:  
21     // There is no stable subsequence (e.g. a reverse)  
22     // OR current node is not among the stable sequence  
23     if (j < 0 || i !== increasingNewIndexSequence[j]) {  
24         move(nextChild, container, anchor, MoveType.REORDER)  
25     } else {  
26         j--  
27     }  
28 }  
29 }
```

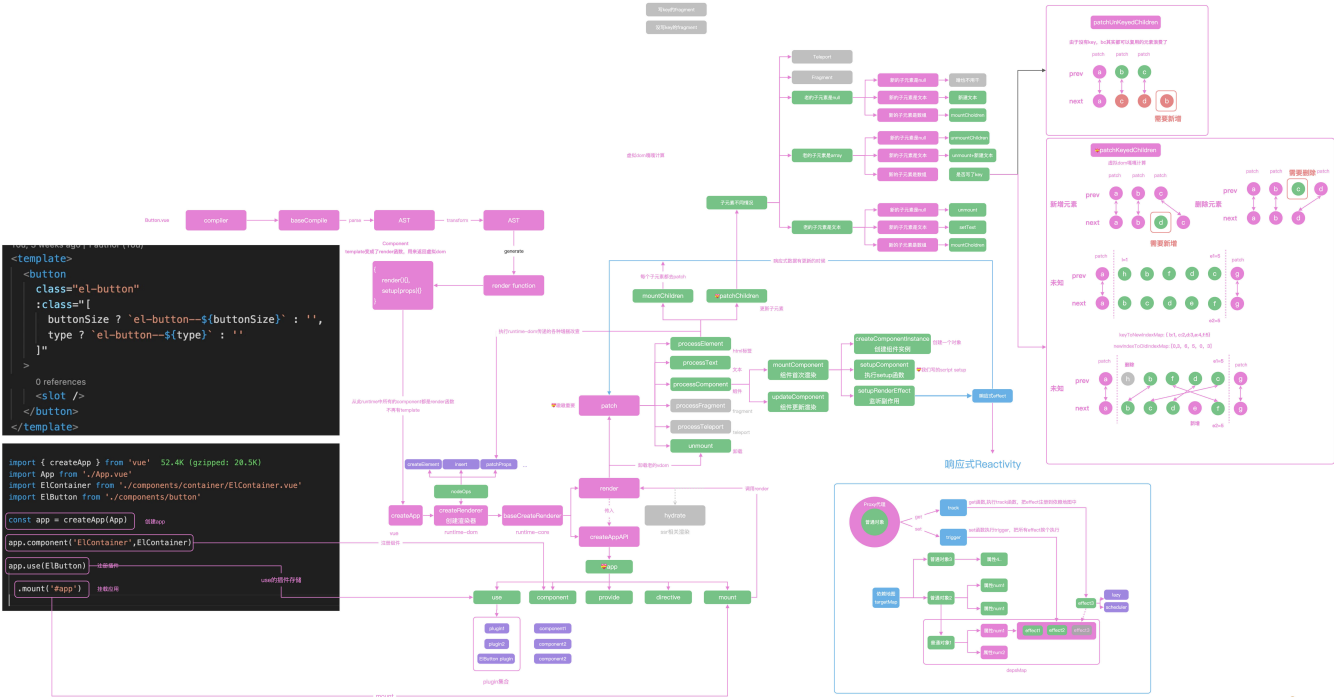
上面代码的思路，我们用下图演示。做完双端对比之后，a 和 g 已经计算出可以直接复用 DOM，剩下的队列中我们需要把 hbfdc 更新成 abdef。

首先我们需要使用 **keyToNewIndexMap** 存储新节点中每个 key 对应的索引，比如下图中 key 是 c 的元素的索引就是 2；然后计算出 **newIndexOldIndexMap** 存储这个 key 在老的子元素中的位置，我们可以根据 c 的索引是 2，在 **newIndexOldIndexMap** 中查询到在老的子元素的位置是 6，关于 **newIndexOldIndexMap** 的具体逻辑你可以在上面的代码中看到：



总结

今天的内容到这就结束了，对照着 Vue 执行全景图，我们回顾一下讲到的知识点。



首先我们分析了 Vue 3 中虚拟 DOM diff 中的静态标记功能，标记后通过位运算，可以快速判断出一个节点的类型是 HTML 标签还是 Vue 组件，然后去执行不同的操作方法；在节点更新的流程中，也可以通过位运算的方式确定需要更新的范围。

位运算就是通过二进制上的与和或运算，能够高效地进行权限的判断，我们在工作中如果涉及权限的判断，也可以借鉴类似的思路，Linux 中的读写权限也是通过位运算的方式来实现的。

然后我们剖析了 Vue 的虚拟 DOM 中最为复杂的最长递增子序列算法，通过对 LeetCode 第 300 的题分析掌握了动态规划和贪心 + 二分的解法。

掌握算法思想之后，我们再回到 Vue3 的源码中分析代码的实现逻辑，patchKeyedChildren 的核心逻辑就是在进行双端对比后，对无法预判的序列计算出最长递增子序列之后，我们通过编译数组，对其余的元素进行 patch 或者 move 的操作，完整实现了虚拟 DOM 的 diff。

学到这里相信你已经完全搞懂了虚拟 DOM 的执行，以及关键的 diff 操作思路，可以体会到 Vue 中极致的优化理念，使用位运算对 Vue 中的动态属性和节点进行标记，实现高效判断；对于两个数组的 diff 计算使用了最长递增子序列算法实现，优化了 diff 的时间复杂度。这也是为什么我一直建议刚入行的前端工程师要好好学习算法的主要原因。

思考题

最后给你留个思考题, 你现在的项目中有哪些地方能用到位运算的地方呢? 欢迎在评论去留言分享你的想法, 我们下一讲再见。

分享给需要的人, Ta订阅后你可得 **20元** 现金奖励

 生成海报并分享

 赞 1  提建议

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 30 | 虚拟DOM (上) : 如何通过虚拟DOM更新页面?

下一篇 32 | 编译原理 (上) : 手写一个迷你Vue 3 Compiler的入门原理

更多课程推荐

陈天 · Rust 编程第一课

实战驱动, 快速上手 Rust

陈天

Tubi TV 研发副总裁



涨价倒计时 

今日订阅 **¥89**, 1月12日涨价至**¥199**

精选留言 (1)

写留言



费城的二鹏
2021-12-31

思考题
后端用户权限的设计，采用了位运算，前端也需要位运算解析

