2021大厂前端秋季招聘面试（社招+校招）核心面试点(二)

#2021#

一、贪心算法

贪心算法其实可以认为是dp问题的一个特例, 除了动态规划的各种特征外, 贪心算法还需要满足"贪心选择性质", 当然效率也比动态规划要高。

- 贪心选择性质：每一步走做出一个局部最优的选择，最终的结果就是全局最优。

同学们肯定一眼就看出问题来了, 并不是所有问题都是这样的, 很多问题局部最优并不能保证全局最优, 只有小部分问题具有这种特质。

比如你前面堆满了金条, 你只能拿5根, 怎么保证拿到的价值最大？ 答案当然是：每次都拿剩下的金条中最重的那根, 那么最后你拿到的一定是最有价值的。

比如斗地主, 对方出了一个3, 你手上有345678还有一个2, 按照贪心选择, 这时候你应该出4了, 实际上咱们会尝试出2, 然后345678起飞~

区间调度问题

来看一个经典的贪心算法问题 Interval Scheduling 区间调度.

有许多[start, end]的闭区间, 请设计一个算法, 算出这些区间中, 最多有几个互不相交的区间。

```
function intervalSchedule(intvs: number[][]) {}

// 比如intvs = [[1,3], [2,4], [3,6]]
// 这些区间最多有两个区间互不相交, 即 [1,3], [3,6], intervalSchedule函数此时应该返回2
```

这个问题在现实中其实有很多应用场景, 比如你今天有好几个活动可以参加, 每个活动区间用[start, end]表示开始和结束时间, 请问你今天最多能参加几个活动?

贪心求解

大家可以先想一想，有什么思路?

1. 可以每次选择可选区间中开始最早的那个?

不行, 因为可能有的区间开始很早, 结束很晚, 比如[0,10], 使我们错过了很多短区间比如[1,2], [2,3]

2. 可以每次选择可选区间中最短的那个?

   不行, 直接看上面这个例子1,3], [2,4], [3,6, 这样的话会选择出 [1, 3], [2, 4], 并不能保证他们不相交

   正确思路

1. 从可选区间intvs里, 选择一个end最小的区间x
2. 把所有与x相交的区间从intvs中剔除
3. 重复1,2, 直到intvs为空, 之前选出的各种区间x， 就是我们要求的结果

   把整个思路转换成代码的话, 因为我们要选出end最小的区间, 所以我们可以先对区间根据end升序排序.

1. 选出end最小的区间

   由于我们已经排过序了, 所以直接选择第一个区间即可

2. 剔除与x相交的区间

   这一步就没第一步那么简单了, 这里建议大家画个图看看

- 代码如下

  看一下区间调度.js

区间调度算法的应用

1. 无重叠区间

   给定一个区间的集合，找到需要移除区间的最小数量，使剩余区间互不重叠。

   注意:

   可以认为区间的终点总是大于它的起点。

区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠。
示例 1:

输入: [[1,2], [2,3], [3,4], [1,3]]

输出: 1

解释: 移除 [1,3] 后，剩下的区间没有重叠。
示例 2:

输入: [[1,2], [1,2], [1,2]]

输出: 2

解释: 你需要移除两个 [1,2] 来使剩下的区间没有重叠。
示例 3:

输入: [[1,2], [2,3]]

输出: 0

解释: 你不需要移除任何区间，因为它们已经是无重叠的了。

- 解答

刚才咱们已经找到了最多有几个互不相交的区间数n, 那么总数减去n就可以了~

咱们来从头写一遍, 就当是复习了

```
var eraseOverlapIntervals = function(intervals) {
    if (intervals.length === 0) {
        return 0;
    }
    let sortArray = intervals.sort((a,b) => a[1] - b[1]);

    let count = 1;

    let xEnd = sortArray[0][1];
```

```
    for (let item of intervals) {
        // 注意，这里题目说了区间 [1,2] 和 [2,3] 的边界相互“接触”，但没有相互重叠，所以应该是item[0] >= xEnd
        if (item[0] >= xEnd) {
            xEnd = item[1];
            count++;
        }
    }

    return intervals.length - count;
};
```

2. 用最少的箭头射爆气球

在二维空间中有许多球形的气球。对于每个气球，提供的输入是水平方向上，气球直径的开始和结束坐标。由于它是水平的，所以y坐标并不重要，因此只要知道开始和结束的x坐标就足够了。开始坐标总是小于结束坐标。平面内最多存在104个气球。

一支弓箭可以沿着x轴从不同点完全垂直地射出。在坐标x处射出一支箭，若有一个气球的直径的开始和结束坐标为 xstart，xend， 且满足 xstart ≤ x ≤ xend，则该气球会被引爆。可以射出的弓箭的数量没有限制。 弓箭一旦被射出之后，可以无限地前进。我们想找到使得所有气球全部被引爆，所需的弓箭的最小数量。

Example:

输入:
10,16], [2,8], [1,6], [7,12

输出:
2

解释:
对于该样例，我们可以在x = 6（射爆[2,8],[1,6]两个气球）和 x = 11（射爆另外两个气球）。

- 解答

这个问题和区间调度算法又是非常的类似, 大家稍微转换一下思路即可。

如果最多有n个不重叠的空间, 那么就至少需要n个箭头穿透所有空间，所以我们要求的其实就是最多有几个不重叠的空间。

来看一下这张图

但是这个题里的描述, 边界重叠后, 箭头是可以一起射爆的, 所以两个区间的边界重叠也算是区间重叠。

二、防抖和节流?

函数防抖（debounce）：当持续触发事件时，一定时间段内没有再触发事件，事件处理函数才会执行一次，如果设定的时间到来之前，又一次触发了事件，就重新开始延时。如下图，持续触发scroll事件时，并不执行handle函数，当1000毫秒内没有触发scroll事件时，才会延时触发scroll事件。 debounce.webp

函数节流（throttle）：当持续触发事件时，保证一定时间段内只调用一次事件处理函数。节流通俗解释就比如我们水龙头放水，阀门一打开，水哗哗的往下流，秉着勤俭节约的优良传统美德，我们要把水龙头关小点，最好是如我们心意按照一定规律在某个时间间隔内一滴一滴的往下滴。如下图，持续触发scroll事件时，并不立即执行handle函数，每隔1000毫秒才会执行一次handle函数。

分别适合用在什么场景?

节流：resize scroll

防抖：input输入

来写一个节流? 还有其他方式实现吗?

1. 时间戳

```
function throttle(fn, interval) {
    let last = 0

    return function () {
        let now = +new Date()

        if (now - last >= interval) {
            last = now;
            fn.apply(this, arguments);
        }
```

```
    }
}
```

2. 计时器

```
function throttle(func, delay) {
    let timer = null;
    return function () {
        let context = this;
        let args = arguments;
        if (!timer) {
            timer = setTimeout(function () {
                func.apply(context, args);
                timer = null;
            }, delay);
        }
    }
}
```

这是写的首节流还是尾节流?

能写一个更完美的吗?

三、事件循环

1. 事件循环存在的意义是什么?
2. 事件循环的基本概念?
3. 浏览器和node环境的事件循环有什么区别?
4. 来看一下这几个代码片段的输出结果是什么?

为什么有事件循环?

单线程:

JavaScript的主要用途是与用户互动，以及操作DOM。如果它是多线程的会有很多复杂的问题要处理，比如有两个线程同时操作DOM，一个线程删除了当前的DOM节点，一个线程是要操作当前的DOM阶段，最后以哪个线程的操作为准？为了避免这种，所以JS是单线程的。即使H5提出了web worker标准，它有很多限制，受主线程控制，是主线程的子线程。

非阻塞：通过 event loop 实现。

宏任务和微任务

宏任务和微任务

为什么要引入微任务，只有一种类型的任务不行么?

页面渲染事件，各种IO的完成事件等随时被添加到任务队列中，一直会保持先进先出的原则执行，我们不能准确地控制这些事件被添加到任务队列中的位置。但是这个时候突然有高优先级的任务需要尽快执行，那么一种类型的任务就不合适了，所以引入了微任务队列。

浏览器里的事件循环

关于微任务和宏任务在浏览器的执行顺序是这样的：

1. 执行全局Script同步代码，这些同步代码有一些是同步语句，有一些是异步语句（比如setTimeout等）；
2. 全局Script代码执行完毕后，调用栈Stack会清空；
3. 从微队列microtask queue中取出位于队首的回调任务，放入调用栈Stack中执行，执行完后microtask queue长度减1；
4. 继续取出位于队首的任务，放入调用栈Stack中执行，以此类推，直到直到把microtask queue中的所有任务都执行完毕。注意，如果在执行microtask的过程中，又产生了microtask，那么会加入到队列的末尾，也会在这个周期被调用执行；
5. microtask queue中的所有任务都执行完毕，此时microtask queue为空队列，调用栈Stack也为空；
6. 取出宏队列macrotask queue中位于队首的任务，放入Stack中执行；
7. 执行完毕后，调用栈Stack为空；
8. 重复第3-7个步骤；
9. 重复第3-7个步骤；

……

可以看一下这个网站, 理解一些调用栈和队列的概念. http://latentflip.com/loupe

常见的 task（宏任务） 比如：setTimeout、setInterval、script（整体代码）、I/O 操作、UI 渲染等。

常见的 micro-task 比如: new Promise().then(回调)、MutationObserver(html5新特性) 等。

宏任务队列里一次循环是要执行所有任务, 还是只执行一个?

宏队列macrotask一次只从队列中取一个任务执行，执行完后就去执行微任务队列中的任务；

微任务队列里一次循环是要执行所有任务, 还是只执行一个?

微任务队列中所有的任务都会被依次取出来执行，知道microtask queue为空；

Node里的事件循环

大体的task（宏任务）执行顺序是这样的：

timers定时器：本阶段执行已经安排的 setTimeout() 和 setInterval() 的回调函数。
pending callbacks待定回调：执行延迟到下一个循环迭代的 I/O 回调。
idle, prepare：仅系统内部使用。
poll 轮询：检索新的 I/O 事件;执行与 I/O 相关的回调（几乎所有情况下，除了关闭的回调函数，它们由计时器和 setImmediate() 排定的之外），其余情况 node 将在此处阻塞。
check 检测：setImmediate() 回调函数在这里执行。
close callbacks 关闭的回调函数：一些准备关闭的回调函数，如：socket.on('close', ...)。

微任务和宏任务在Node的执行顺序

Node 10以前：
执行完一个阶段的所有任务
执行完nextTick队列里面的内容
然后执行完微任务队列的内容

Node 11以后：
和浏览器的行为统一了，都是每执行一个宏任务就执行完微任务队列。

代码输出顺序题

1. 第一题

```
async function async1() {
    console.log('async1 start');
    await async2();
    console.log('async1 end');
}
async function async2() {
    console.log('async2');
}
console.log('script start');
setTimeout(function () {
    console.log('setTimeout');
}, 0)
async1();
```

```
new Promise(function (resolve) {
    console.log('promise1');
    resolve();
}).then(function () {
    console.log('promise2');
});
console.log('script end');
```

2. 第二题

```
console.log('start');
setTimeout(() => {
    console.log('children2');
    Promise.resolve().then(() => {
        console.log('children3');
    })
}, 0);

new Promise(function (resolve, reject) {
    console.log('children4');
    setTimeout(function () {
        console.log('children5');
        resolve('children6')
    }, 0)
}).then((res) => {
    console.log('children7');
    setTimeout(() => {
        console.log(res);
    }, 0)
})
```

3. 第三题

```
const p = function () {
    return new Promise((resolve, reject) => {
```

```
        const p1 = new Promise((resolve, reject) => {
            setTimeout(() => {
                resolve(1)
            }, 0)
            resolve(2)
        })
        p1.then((res) => {
            console.log(res);
        })
        console.log(3);
        resolve(4);
    })
}


p().then((res) => {
    console.log(res);
})
console.log('end');
```

常见代码题

一、如何实现setTimeout?

通过一些现有的api, 尝试实现setTimeout?

即实现如下两个函数

```
/**
 * 清除定时器
 * @param {Number} timerId 定时器Id
 */
const mClearTimeout = (timerId) => {

}
```

```
/**
 * 设置定时器
 * @param {Function} fn 回调函数
 * @param {Number} timeout 延迟时间
 * @param  {...any} args 回调函数的参数
 */
const mSetTimeout = (fn, timeout, ...args) => {

}
```

二、用setTimeout实现setInterval

期望利用原生的setTimeout来模拟实现setInterval, 即实现如下函数

```
/**
 * 使用setTimeout模拟实现setInterval
 * @param {Function} fn 回调函数
 * @param {*} delay 延迟时间
 * @param  {...any} args 回调函数的参数
 */
function mockSetInterval(fn, delay, ...args) {

}
```

三、sleep的多种实现

实现sleep, 能够实现阻塞主流程一段时间, 尝试多种方式实现

```
/**
 * @param {Number} delay
 */
function sleep(delay) {

}
```

四、实现红绿灯

要求使用一个div实现红绿灯效果, 把一个圆形 div 按照绿色 3 秒，黄色 1 秒，红色 2 秒循环改变背景色。

Tips: 同学们可以回去尝试使用 setTimeout嵌套/promise链式调用 分别实现一下