[选择题考点：](https://blog.csdn.net/Sabrina_cc/article/details/108608860?utm_medium=distribute.pc_relevant_download.none-task-blog-baidujs-1.nonecase&depth_1-utm_source=distribute.pc_relevant_download.none-task-blog-baidujs-1.nonecase)

1. 触发硬件加速的CSS属性

CSS的 animation、tranform、transition并不会自动开启GPU加速，而是通过浏览器的缓慢的软件渲染引擎来实现执行

使用 translate3d() rotate3d() scale3d()　这几个方法，我们就可以使用ＧＰＵ加速了。

如下几个css属性可以触发硬件加速：transform、opacity、filter、will-change：哪一个属性即将发生变化，进而进行优化。

2. JS模块规范

AMD, UMD, CMD, commonJS, ES6 module

参考链接：https://segmentfault.com/a/1190000012419990

3. 跨域的解决方案

通过jsonp跨域

document.domain + iframe跨域

location.hash + iframe

window.name + iframe跨域

postMessage跨域

跨域资源共享（CORS）

nginx代理跨域

nodejs中间件代理跨域

WebSocket协议跨域

参考链接: https://segmentfault.com/a/1190000011145364

4. 排序算法的时间复杂度与稳定性

参考链接: https://blog.csdn.net/pange1991/article/details/85460755

5. GBK和UTF8编码

6. 计算机网络各层协议

7. JS宏任务和微任务

宏任务（macrotask ）和微任务（microtask ）表示异步任务的两种分类。在挂起任务时，JS 引擎会将所有任务按照类别分到这两个队列中，首先在 macrotask 的队列（这个队列也被叫做 task queue）中取出第一个任务，执行完毕后取出 microtask 队列中的所有任务顺序执行；之后再取 macrotask 任务，周而复始，直至两个队列的任务都取完。同一次事件循环中 微任务永远在宏任务之前执行

参考链接：https://www.cnblogs.com/ckAng/p/11133643.html

8. HTTP/2

9. webpack优点

10. 防止XSS

11. 对称加密算法和非对称加密算法以及可逆性

12. 中断JS执行

13. 服务端渲染和客户端渲染的优缺点

（1）服务器端渲染

     每当有客户端请求页面了，服务器先在后端调用art-template，把指定的页面预先在后端渲染后，然后通过res.end把这个渲染完毕的完整页面，返回给客户端直接展示。

 优点：对SEO友好，因为我们经过服务器端渲染的页面，在网络中传输的时候，传输的是一个真实的页面。因此，爬虫客户端，当爬到我们的页面后，会分系我们给他提供的这个页面，此时，我们页面中的关键数据就会被爬虫给收录了。

缺点： 服务器端渲染，对服务器压力比较打，可以使用服务器端的页面缓存技术，减轻服务器的渲染压力；不适合前后端分离开发。

（2）客户端渲染

     每当用户要请求某个页面了， 第一步，用户需要先把这个页面请求到客户端，此时用户拿到的页面只是一个模板页面。 第二步，浏览器在解析模板页面的时候，会发起art-template的二次资源请求，同时要发送Ajax请求，去服务器获取数据 第三步，在客户端调用art-template 渲染HTML结构，并把渲染厚的htmlStr append 到页面指定的容器中;

缺点： 对SEO相当不友好

优点： 减轻了服务器端的渲染压力;同时，最大的好处就是：能够实现前后端分离开发;

14. 客户端浏览器HTTP请求

15. transform变形

[编程题:](https://blog.csdn.net/Sabrina_cc/article/details/108608860?utm_medium=distribute.pc_relevant_download.none-task-blog-baidujs-1.nonecase&depth_1-utm_source=distribute.pc_relevant_download.none-task-blog-baidujs-1.nonecase)

## 1. fizzBuzz

实现 fizzBuzz 函数，参数 num 与返回值的关系如下： 1、如果 num 能同时被 7 和 9 整除，返回字符串 fizzbuzz 2、如果 num 能被 7 整除，返回字符串 fizz 3、如果 num 能被 9 整除，返回字符串 buzz 4、如果参数为空或者不是 Number 类型，返回 false 5、其余情况，返回参数 num

// 处理输入

const input = readInt()

// 处理输出

const result = Solution(input)

print(result)

// 算法部分

function Solution(num) {

let result

if(num%7==0 && num%9==0){

result = "fizzbuzz"

} else if (num%7==0){

result = "fizz"

} else if(num%9 ==0){

result = "buzz"

} else if(num == null || num.isNaN()){

return false

} else {

return num

}

return result；

}

参考力扣代码: https://leetcode-cn.com/problems/fizz-buzz/

## 2. 推到多米诺

 一行中有 N 张多米诺骨牌，我们将每张多米诺骨牌垂直竖立。

在开始时，我们同时把一些多米诺骨牌向左或向右推。

每过一秒，倒向左边的多米诺骨牌会推动其左侧相邻的多米诺骨牌。

同样地，倒向右边的多米诺骨牌也会推动竖立在其右侧的相邻多米诺骨牌。

如果同时有多米诺骨牌落在一张垂直竖立的多米诺骨牌的两边，由于受力平衡， 该骨牌仍然保持不变。

就这个问题而言，我们会认为正在下降的多米诺骨牌不会对其它正在下降或已经下降的多米诺骨牌施加额外的力。

给定表示初始状态的字符串 "S" 。如果第 i 张多米诺骨牌被推向左边，则 S[i] = 'L'；如果第 i 张多米诺骨牌被推向右边，则 S[i] = 'R'；如果第 i 张多米诺骨牌没有被推动，则 S[i] = '.'。

返回表示最终状态的字符串。

// 处理输入

let input='', line;

while((line = read\_line()) != ''){

input += line;

}

// 处理输出

const result = Solution(input);

print(result);

// 算法部分

function Solution(s) {

let L = R = 0

let len = s.length

s = s.split('')

while(R < len){

if(s[R] === '.') R++

if(L === R) R++

if(s[R]==='L' && s[L]!=='R'){

while(L!==R){

s[L] = 'L'

L++

}

}else if (s[R]==='L' && s[L]==='R'){

let left =L,right = R

while(left < right){

s[left]='R'

left++

s[right]='L'

right--

}

L = R

}else if(s[R]==='R' && s[L]!=='R'){

L=R

} else if(s[R]==='R' && s[L]==='R'){

while(L!==R){

s[L]='R'

L++

}

} else if(R==len-1 &&s[L]=='R'){

while(L < R){

L++

s[L]='R'

}

}

}

return s.join('')

}

————————————————

版权声明：本文为CSDN博主「小白Rachel」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：https://blog.csdn.net/Sabrina\_cc/article/details/108608860