几道高级前端面试题解析

为什么 0.1 + 0.2 != 0.3，请详述理由

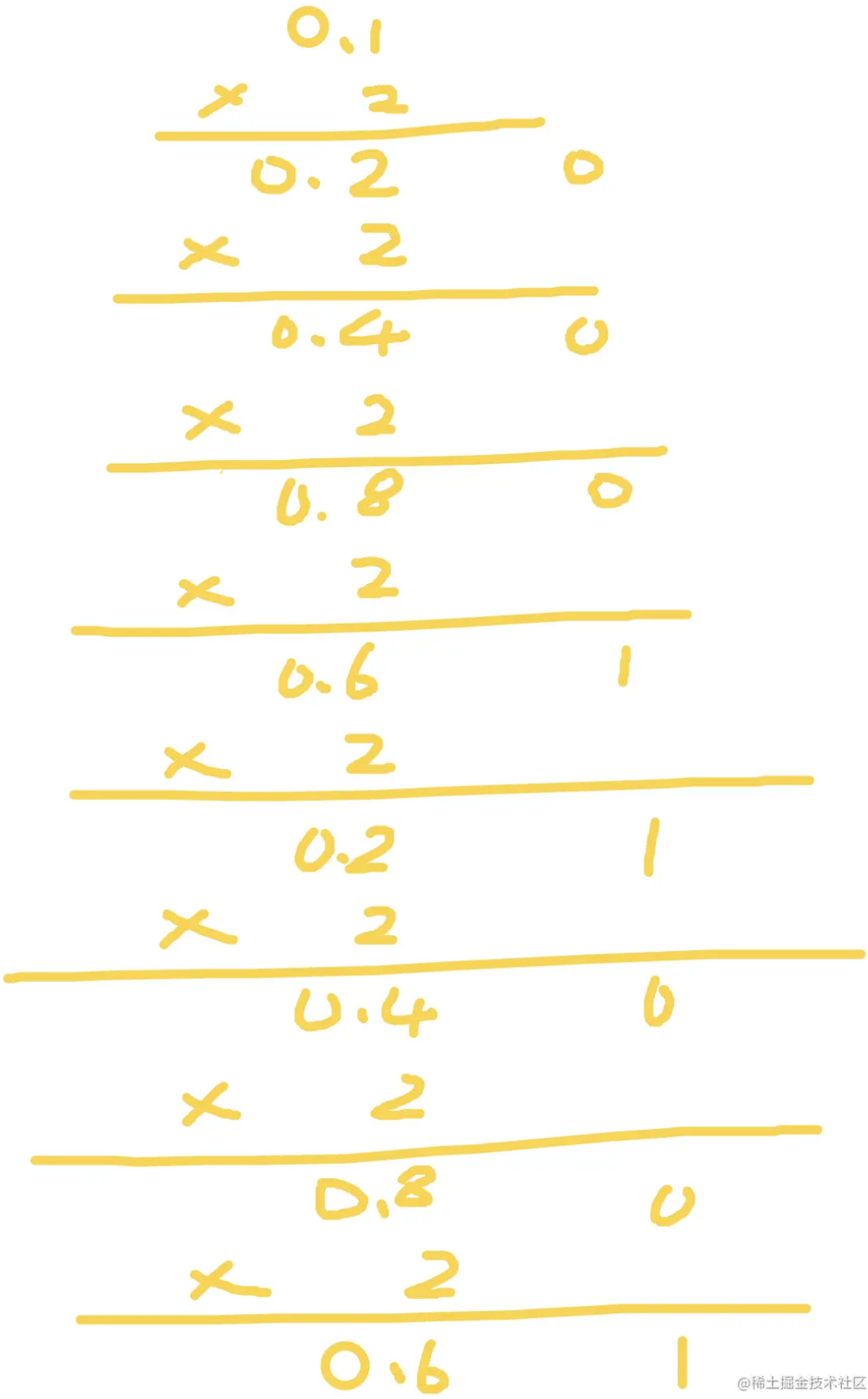
因为 JS 采用 IEEE 754 双精度版本（64位），并且只要采用 IEEE 754 的语言都有该问题。

我们都知道计算机表示十进制是采用二进制表示的，所以 `0.1` 在二进制表示为

ini 复制代码

```
// (0011) 表示循环
0.1 = 2^-4 * 1.10011(0011)
```

那么如何得到这个二进制的呢，我们可以来演算下



小数算二进制和整数不同。乘法计算时，只计算小数位，整数位用作每一位的二进制，并且得到的第一位为最高位。所以我们得出 `0.1 = 2^-4 * 1.10011(0011)` ，那么 `0.2` 的演算也基本如上所示，只需要去掉第一步乘法，所以得出 `0.2 = 2^-3 * 1.10011(0011)` 。

回来继续说 IEEE 754 双精度。六十四位中符号位占一位，整数位占十一位，其余五十二位都为小数位。因为 `0.1` 和 `0.2` 都是无限循环的二进制了，所以在小数位末尾处需要判断是否进位（就和十进制的四舍五入一样）。

所以 `2^-4 * 1.10011...001` 进位后就变成了 `2^-4 * 1.10011(0011 * 12次)010` 。那么把这两个二进制加起来会得出 `2^-2 * 1.0011(0011 * 11次)0100` ，这个值算成十进制就是 `0.30000000000000004`

下面说一下原生解决办法，如下代码所示

scss 复制代码

```
parseFloat((0.1 + 0.2).toFixed(10))
```

10 个 Ajax 同时发起请求，全部返回展示结果，并且至多允许三次失败，说出设计思路

这个问题相信很多人会第一时间想到 `Promise.all` ，但是这个函数有一个局限在于如果失败一次就返回了，直接这样实现会有点问题，需要变通下。以下是两种实现思路

javascript 复制代码

```
// 以下是不完整代码，着重于思路 非 Promise 写法
let successCount = 0
let errorCount = 0
let datas = []
ajax(url, (res) => {
     if (success) {
         success++
         if (success + errorCount === 10) {
             console.log(datas)
         } else {
             datas.push(res.data)
         }
     } else {
         errorCount++
         if (errorCount > 3) {
            // 失败次数大于3次就应该报错了
             throw Error('失败三次')
         }
     }
})
// Promise 写法
let errorCount = 0
let p = new Promise((resolve, reject) => {
    if (success) {
         resolve(res.data)
```

```
    } else {
        errorCount++
        if (errorCount > 3) {
           // 失败次数大于3次就应该报错了
           reject(error)
        } else {
            resolve(error)
        }
    }
})
Promise.all([p]).then(v => {
  console.log(v);
});
```

基于 Localstorage 设计一个 1M 的缓存系统，需要实现缓存淘汰机制

设计思路如下：

- 存储的每个对象需要添加两个属性：分别是过期时间和存储时间。
- 利用一个属性保存系统中目前所占空间大小，每次存储都增加该属性。当该属性值大于 1M 时，需要按照时间排序系统中的数据，删除一定量的数据保证能够存储下目前需要存储的数据。
- 每次取数据时，需要判断该缓存数据是否过期，如果过期就删除。

以下是代码实现，实现了思路，但是可能会存在 Bug，但是这种设计题一般是给出设计思路和部分代码，不会需要写出一个无问题的代码

kotlin 复制代码

```
class Store {
  constructor() {
    let store = localStorage.getItem('cache')
    if (!store) {
      store = {
        maxSize: 1024 * 1024,
        size: 0
      }
      this.store = store
    } else {
      this.store = JSON.parse(store)
    }
  }
  set(key, value, expire) {
    this.store[key] = {
      date: Date.now(),
      expire,
```

```
    value
  }
  let size = this.sizeOf(JSON.stringify(this.store[key]))
  if (this.store.maxSize < size + this.store.size) {
    console.log('超了-----------');
    var keys = Object.keys(this.store);
    // 时间排序
    keys = keys.sort((a, b) => {
      let item1 = this.store[a], item2 = this.store[b];
      return item2.date - item1.date;
    });
    while (size + this.store.size > this.store.maxSize) {
      let index = keys[keys.length - 1]
      this.store.size -= this.sizeOf(JSON.stringify(this.store[index]))
      delete this.store[index]
    }
  }
  this.store.size += size

  localStorage.setItem('cache', JSON.stringify(this.store))
}
get(key) {
  let d = this.store[key]
  if (!d) {
    console.log('找不到该属性');
    return
  }
  if (d.expire > Date.now) {
    console.log('过期删除');
    delete this.store[key]
    localStorage.setItem('cache', JSON.stringify(this.store))
  } else {
    return d.value
  }
}
sizeOf(str, charset) {
  var total = 0,
    charCode,
    i,
    len;
  charset = charset ? charset.toLowerCase() : '';
  if (charset === 'utf-16' || charset === 'utf16') {
    for (i = 0, len = str.length; i < len; i++) {
      charCode = str.charCodeAt(i);
      if (charCode <= 0xffff) {
        total += 2;
      } else {
        total += 4;
      }
```

```
      }
    } else {
      for (i = 0, len = str.length; i < len; i++) {
        charCode = str.charCodeAt(i);
        if (charCode <= 0x007f) {
          total += 1;
        } else if (charCode <= 0x07ff) {
          total += 2;
        } else if (charCode <= 0xffff) {
          total += 3;
        } else {
          total += 4;
        }
      }
    }
    return total;
  }
}
```

详细说明 Event loop

众所周知 JS 是门非阻塞单线程语言，因为在最初 JS 就是为了和浏览器交互而诞生的。如果 JS 是门多线程的语言话，我们在多个线程中处理 DOM 就可能会发生问题（一个线程中新加节点，另一个线程中删除节点），当然可以引入读写锁解决这个问题。

JS 在执行的过程中会产生执行环境，这些执行环境会被顺序的加入到执行栈中。如果遇到异步的代码，会被挂起并加入到 Task（有多种 task） 队列中。一旦执行栈为空，Event Loop 就会从 Task 队列中拿出需要执行的代码并放入执行栈中执行，所以本质上来说 JS 中的异步还是同步行为。

javascript 复制代码

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

console.log('script end');
```

以上代码虽然 `setTimeout` 延时为 0，其实还是异步。这是因为 HTML5 标准规定这个函数第二个参数不得小于 4 毫秒，不足会自动增加。所以 `setTimeout` 还是会在 `script end` 之后打印。

不同的任务源会被分配到不同的 Task 队列中，任务源可以分为 微任务（microtask） 和 宏任务（macrotask）。在 ES6 规范中，microtask 称为 `jobs`，macrotask 称为 `task`。

javascript 复制代码

```
console.log('script start');

setTimeout(function() {
  console.log('setTimeout');
}, 0);

new Promise((resolve) => {
    console.log('Promise')
    resolve()
}).then(function() {
  console.log('promise1');
}).then(function() {
  console.log('promise2');
});

console.log('script end');
// script start => Promise => script end => promise1 => promise2 => setTimeout
```

以上代码虽然 `setTimeout` 写在 `Promise` 之前，但是因为 `Promise` 属于微任务而 `setTimeout` 属于宏任务，所以会有以上的打印。

微任务包括 `process.nextTick` ， `promise` ， `Object.observe` ， `MutationObserver`

宏任务包括 `script` ， `setTimeout` ， `setInterval` ， `setImmediate` ， `I/O` ， `UI rendering`

很多人有个误区，认为微任务快于宏任务，其实是错误的。因为宏任务中包括了 `script` ，浏览器会先执行一个宏任务，接下来有异步代码的话就先执行微任务。

所以正确的一次 Event loop 顺序是这样的

1. 执行同步代码，这属于宏任务
2. 执行栈为空，查询是否有微任务需要执行
3. 执行所有微任务
4. 必要的话渲染 UI
5. 然后开始下一轮 Event loop，执行宏任务中的异步代码

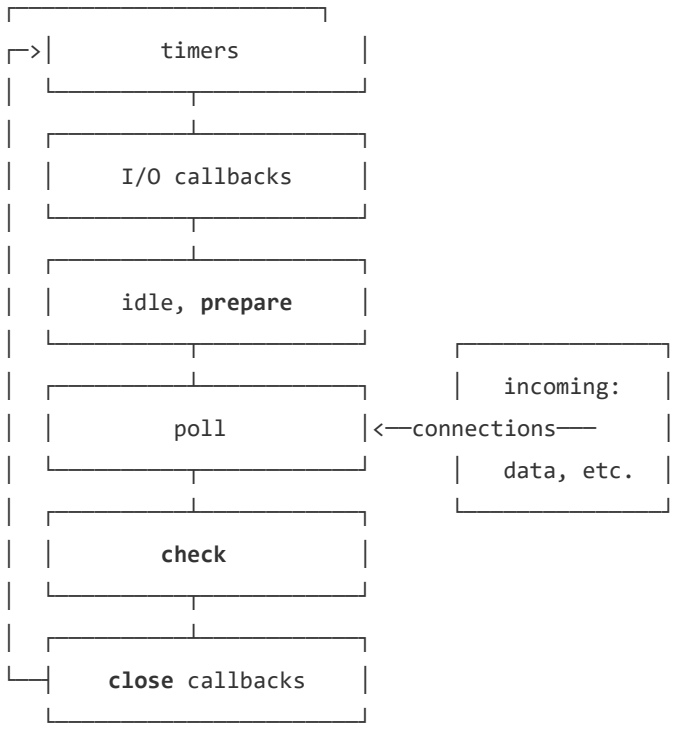
通过上述的 Event loop 顺序可知，如果宏任务中的异步代码有大量的计算并且需要操作 DOM 的话，为了更快的 界面响应，我们可以把操作 DOM 放入微任务中。

Node 中的 Event loop

Node 中的 Event loop 和浏览器中的不相同。

Node 的 Event loop 分为6个阶段，它们会按照顺序反复运行

sql 复制代码



timer

timers 阶段会执行 `setTimeout` 和 `setInterval`

一个 `timer` 指定的时间并不是准确时间，而是在达到这个时间后尽快执行回调，可能会因为系统正在执行别的事务而延迟。

下限的时间有一个范围： `[1, 2147483647]` ，如果设定的时间不在这个范围，将被设置为1。

I/O

I/O 阶段会执行除了 close 事件，定时器和 `setImmediate` 的回调

idle, prepare

idle, prepare 阶段内部实现

poll

poll 阶段很重要，这一阶段中，系统会做两件事情

1. 执行到点的定时器
2. 执行 poll 队列中的事件

并且当 poll 中没有定时器的情况下，会发现以下两件事情

- 如果 poll 队列不为空，会遍历回调队列并同步执行，直到队列为空或者系统限制
- 如果 poll 队列为空，会有两件事发生
  - 如果有 `setImmediate` 需要执行，poll 阶段会停止并且进入到 check 阶段执行 `setImmediate`
  - 如果没有 `setImmediate` 需要执行，会等待回调被加入到队列中并立即执行回调

如果有别的定时器需要被执行，会回到 timer 阶段执行回调。

check

check 阶段执行 `setImmediate`

close callbacks

close callbacks 阶段执行 close 事件

并且在 Node 中，有些情况下的定时器执行顺序是随机的

```javascript 复制代码
setTimeout(() => {
    console.log('setTimeout');
}, 0);
setImmediate(() => {
    console.log('setImmediate');
})
// 这里可能会输出 setTimeout，setImmediate
// 可能也会相反的输出，这取决于性能
// 因为可能进入 event loop 用了不到 1 毫秒，这时候会执行 setImmediate
// 否则会执行 setTimeout
```

当然在这种情况下，执行顺序是相同的

javascript 复制代码

```
var fs = require('fs')

fs.readFile(__filename, () => {
    setTimeout(() => {
        console.log('timeout');
    }, 0);
    setImmediate(() => {
        console.log('immediate');
    });
});
// 因为 readFile 的回调在 poll 中执行
// 发现有 setImmediate ，所以会立即跳到 check 阶段执行回调
// 再去 timer 阶段执行 setTimeout
// 所以以上输出一定是 setImmediate，setTimeout
```

上面介绍的都是 macrotask 的执行情况，microtask 会在以上每个阶段完成后立即执行。

javascript 复制代码

```
setTimeout(()=>{
    console.log('timer1')

    Promise.resolve().then(function() {
        console.log('promise1')
    })
}, 0)

setTimeout(()=>{
    console.log('timer2')

    Promise.resolve().then(function() {
        console.log('promise2')
    })
}, 0)

// 以上代码在浏览器和 node 中打印情况是不同的
// 浏览器中打印 timer1, promise1, timer2, promise2
// node 中打印 timer1, timer2, promise1, promise2
```

Node 中的 `process.nextTick` 会先于其他 microtask 执行。

javascript 复制代码

```
setTimeout(() => {
  console.log("timer1");

  Promise.resolve().then(function() {
    console.log("promise1");
  });
```

```
}, 0);

process.nextTick(() => {
  console.log("nextTick");
});
// nextTick, timer1, promise1
```