简答题

谈谈你是如何理解js异步编程的，eventloop,消息队列都是做什么的，什么是宏任务，什么是微任务?

答:
1）js 异步编程的主要特点就是代码的执行顺序不是代码的编码顺序，并且能实现类似于多线程并发执行的效果，而无需阻塞线程。
2）所谓事件循环，其实是指渲染进程中的js引擎线程的执行循环。js 是单线程的，其实说的是jjs引擎线程是单线程的，js引擎线程在某个时间点只能执行一个任务，如果在执行任务的过程中，有其他任务产生，比如输入监听，js引擎线程就没办法接收并执行监听回调函数了，为了解决这个问题， 而产生的事件循环和消息队列。
3）消息队列中的包括宏任务队列和微任务队列，chromiun中还有延时队列（延时队列是一个哈希结构，里面的任务也是宏任务），每个宏任务会关联一个微任务队列
4）宏任务，包括主代码块和宏任务队列中的任务。宏任务队列中的任务，是在js引擎线程执行宏任务的过程中，其他线程（事件触发线程，定时触发器线程，异步http请求线程等）将任务推入宏任务队列中的。
5）微任务，是在执行宏任务/微任务过程中产生的，由js引擎线程推入微任务队列中
6）事件循环的过程是：

- js引擎线程执行一个宏任务
- 在当前宏任务执行结束之前，js引擎线程开始依次执行其关联的微任务队列中的微任务
- 当清空微任务队列中的任务时，宏任务结束，此时如果发现需要渲染，将控制权交给GUI渲染线程
- 渲染完毕，js引擎线程接管控制权，从宏任务队列中取出一个宏任务，并开始下一轮循环

7)需要注意的是，微任务的执行时间是会算在宏任务的执行时间里面的,,js引擎线程和GUI渲染线程是互斥的

代码题

一、 将下面的代码用promise的方式改进

```
setTimeout(function () {
  var a = 'hello'
  setTimeout(function () {
    var b = 'lagou'
    setTimeout(function () {
      var c = 'I ♥ U'
      console.log(a + b + c)
    }, 10)
  }, 10)
}, 10)
```

答：

```
function doIt(){
    var a = await new Promise(function(resolve){
        setTimeout(function(){
            resolve('hello')
        },10)

    })
    var b = await new Promise(function(resolve){
        setTimeout(function(){
            resolve('logou')
        },10)

    })
    var c = await new Promise(function(resolve){
        setTimeout(function(){
            resolve('i love u')
        },10)

    })
    console.log(a + b + c)
}
doIt()
```

二、基于一下代码完成下面的四个练习

```
const fp = require('lodash/fp')
// 数据
// horsepower 马力, dollar_value 价格,
in_stock 库存
const cars = [
  { name: 'Ferrari FF', horsepower: 660,
dollar_value: 700000, in_stock: true },
  { name: 'Spyker C12 Zagato',
horsepower: 650, dollar_value: 648000,
in_stock: false },
  { name: 'Jaguar XKR-S', horsepower:
550, dollar_value: 132000, in_stock:
false },
  { name: 'Audi R8', horsepower: 525,
dollar_value: 114200, in_stock: false },
  { name: 'Aston Martin One-77',
horsepower: 750, dollar_value: 1850000,
in_stock: true },
  { name: 'Pagani Huayra', horsepower:
700, dollar_value: 1300000, in_stock:
false },
]
```

练习1：使用组合函数fp.flowRight()重新实现下面这个函数

```
let isLastInStock = function (cars) {
  // 获取最后一条数据
  let last_car = fp.last(cars)
  // 获取最后一条数据的 in_stock 属性值
  return fp.prop('in_stock', last_car)
}
```

答：

```
let isLastInStock = fp.flowRight(fp.prop('in_stock'), fp.last)
```

练习2：使用fp.flowRight(), fp.prop()和fp.first()获取第一个car的name

答：

```
let isFirstInStock = fp.flowRight(fp.prop('in_stock'), fp.first)
```

练习3：使用帮助函数 _average 重构 averageDollarValue,使用函数组合的方式实现

```
let _average = function (xs) {
  return fp.reduce(fp.add, 0, xs) /
xs.length
} // <- 无须改动
let averageDollarValue = function (cars)
{
  let dollar_values = fp.map(function
(car) {
    return car.dollar_value
  }, cars)
  return _average(dollar_values)
}
```

答：

```
let averageDollarValue = fp.flowRight(_average, fp.map(function(car){
    return car.dollar_value
}))
```

练习4：使用flowRight 写一个sanitizeNames()函数，返回一个下划线连接的小写字符串，把数组中的name转换为这种形式：例如sanitizeNames(["hello World"])=>["hello_world"]

```
let _underscore = fp.replace(/\W+/g,
'_') // <-- 无须改动，并在 sanitizeNames
中使用它
```

答：

```
let sanitizeNames = fp.map(fp.flowRight(_underscore, fp.toLower))
```

三、基于下面提供的代码，完成后续的四个练习

```
// support.js
class Container {
  static of(value) {
    return new Container(value)
  }
  constructor(value) {
    this._value = value
  }
  map(fn) {
    return Container.of(fn(this._value))
  }
}
class Maybe {
  static of(x) {
    return new Maybe(x)
  }
  isNothing() {
    return this._value === null ||
this._value === undefined
  }
  constructor(x) {
    this._value = x
  }
  map(fn) {
    return this.isNothing() ? this :
Maybe.of(fn(this._value))
  }
}
module.exports = { Maybe, Container }
```

练习1：使用fp.add(x,y) 和 fp.map(f,x)创建一个能让functor里的值增加的函数
ex1

```
// app.js
const fp = require('lodash/fp')
const { Maybe, Container } =
require('./support')
let maybe = Maybe.of([5, 6, 1])
let ex1 = () => {
  // 你需要实现的函数。。。
}
```

答:

```
let ex1 = () => {
    return maybe.map(fp.map(fp.add(1))).__value
}
```

练习2：实现一个函数ex2,能够使用fp.first获取列表的第一个元素

```
// app.js
const fp = require('lodash/fp')
const { Maybe, Container } =
require('./support')
let xs = Container.of(['do', 'ray',
'me', 'fa', 'so', 'la', 'ti', 'do'])
let ex2 = () => {
  // 你需要实现的函数。。。
}
```

答:

```
let ex2 = ()=>{
    return xs.map(fp.first).__value
}
```

练习3：实现一个函数ex3,使用safeProp和fp.first找到user的名字的首字母

```
// app.js
const fp = require('lodash/fp')
const { Maybe, Container } =
require('./support')
let safeProp = fp.curry(function (x, o)
{
  return Maybe.of(o[x])
})
let user = { id: 2, name: 'Albert' }
let ex3 = () => {
  // 你需要实现的函数。。。
}
```

答:

```
let ex3 = () => {
    return safeProp('name')(user).map(fp.first)._value
}
```

练习4：使用Maybe重写ex4, 不要有if语句

```
// app.js
const fp = require('lodash/fp')
const { Maybe, Container } =
require('./support')
let ex4 = function (n) {
  if (n) {
    return parseInt(n)
  }
}
```

答:

```
let ex4 = function(n){
    return Maybe.of(n).map(parseInt)._value
}
```

三、手写实现MyPromise 源码

要求：尽可能还原Promise 中的每一个API，并通过注释的放肆描述思路和原理

答：

```
/* 先设置出promise的三种状态*/
var PENDING = 'PENDING'
var FULFILLED = 'FULFILLED'
var REJECTED = 'REJECTED'
function MyPromise(fn) {
    this.status = PENDING; // 初始状态为pending
    this.value = null; // 初始化value
    this.reason = null; // 初始化reason

    // 构造函数里面添加两个数组存储成功和失败的回调
    this.onFulfilledCallbacks = [];
    this.onRejectedCallbacks = [];

    // resolve方法参数是value
    //在resolve 函数时不需要关心value的类型，一切类型判断放到then中进行
    function resolve(value) {
        var run = function () {
            if (this.status == PENDING) {
                // 只有pending 状态才会转变为其他状态，这里需要判断，避免同时执行了resolve和reject
                this.status = FULFILLED;
                this.value = value;
                // 当resolve 是异步执行时，then方法会先执行，
                // then方法中的onFulFilled回调就需要存在队列中，等待resolve改变时执行
                this.onFulfilledCallbacks.forEach(callback => {
                    callback(value);
                });
            }
        };
        // then 中的回调函数延迟执行
        setTimeout(run.bind(this), 0);
    }

    // reject方法参数是reason，解析同上
    function reject(reason) {
        var run = function () {
            if (this.status == PENDING) {
                this.status = REJECTED;
                this.reason = reason;
                this.onRejectedCallbacks.forEach(callback => {
                    callback(reason);
                });
            }
        };
```

```
        // then 中的回调函数延迟执行
        setTimeout(run.bind(this), 0);
    }

    try {
        fn(resolve.bind(this), reject.bind(this));
    } catch (error) {
        reject(error);
    }
}
function deepResolvePromise(promise, x, resolve, reject) {
    // 如果 promise 和 x 指向同一对象，以 TypeError 为据因拒绝执行 promise
    // 这是为了防止死循环
    if (promise === x) {
        return reject(new TypeError('Chaining cycle detected for promise #<Promise>'));
    }

    // 如果 x 为 Promise ，则使 promise 接受 x 的状态
    if (x instanceof MyPromise) {
        // 如果 x 处于等待态， promise 需保持为等待态直至 x 被执行或拒绝
        if (x.status === PENDING) {
            x.then(function (y) {
                deepResolvePromise(promise, y, resolve, reject);
            }, reject);
        } else if (x.status === FULFILLED) {
            // 如果 x 处于执行态，用相同的值执行 promise
            resolve(x.value);
        } else if (x.status === REJECTED) {
            // 如果 x 处于拒绝态，用相同的据因拒绝 promise
            reject(x.reason);
        }
    }
    // 如果 x 为对象或者函数
    else if (x && (typeof x === 'object' || typeof x === 'function')) {

        try {
            // 把 x.then 赋值给 then
            var then = x.then;
        } catch (error) {
            // 如果取 x.then 的值时抛出错误 e ，则以 e 为据因拒绝 promise
            return reject(error);
        }

        // 如果 then 是函数
        if (typeof then === 'function') {
            var called = false;
            // 将 x 作为函数的作用域 this 调用之
```

```
            // 传递两个回调函数作为参数，第一个参数叫做 resolvePromise ，第二个参数叫做 rejectPromise
            try {
                then.call(
                    x,
                    // 如果 resolvePromise 以值 y 为参数被调用，则运行 [[Resolve]](promise, y)
                    function (y) {
                        // 如果 resolvePromise 和 rejectPromise 均被调用，
                        // 或者被同一参数调用了多次，则优先采用首次调用并忽略剩下的调用
                        // 实现这条需要前面加一个变量called
                        if (called) return;
                        called = true;
                        deepResolvePromise(promise, y, resolve, reject);
                    },
                    // 如果 rejectPromise 以据因 r 为参数被调用，则以据因 r 拒绝 promise
                    function (r) {
                        if (called) return;
                        called = true;
                        reject(r);
                    });
            } catch (error) {
                // 如果调用 then 方法抛出了异常 e：
                // 如果 resolvePromise 或 rejectPromise 已经被调用，则忽略之
                if (called) return;

                // 否则以 e 为据因拒绝 promise
                reject(error);
            }
        } else {
            // 如果 then 不是函数，以 x 为参数执行 promise
            resolve(x);
        }
    } else {
        // 如果 x 不为对象或者函数，以 x 为参数执行 promise
        resolve(x);
    }
}
MyPromise.prototype.then = function (onFulfilled, onRejected) {
    var that = this;   // 保存一下this

    // then 方法一定返回一个promis，所以这里需要return promise
    // 返回的promise 不能等于 onFulfilled 回调函数返回的值，否则会抛出TypeError的错误，所以，需要记录下promise
    var promise2 = new MyPromise(function (resolvePromise, rejectPromise) {
```

```
133         // 执行then方法时，需要判断当前promise的状态，如果时fullfilled/rejected 直接调用回调函数函数
134         // 如果pendding状态，需要推入队列，等待状态改变时执行
135         // 如果onFulfilled不是函数，给一个默认函数，返回value
136 
137         switch (that.status) {
138 
139             case PENDING:
140                 that.onFulfilledCallbacks.push(function () {
141                     // 这里 的try catch 一定要放在function 内部，因为是需要捕获异步调用时的错误
142                     try {
143                         if (typeof onFulfilled !== 'function') {
144                             // 如果onFulfilled不是函数，直接把当前值传到下一个then回调
145                             resolvePromise(that.value);
146                         } else {
147                             // 如果onFulfilled是函数，需要判断返回的值，并进行深度递归（promise深度promise的清空）
148                             // 因为它需要告诉then返回的promise它状态改变了
149                             var x = onFulfilled(that.value);
150                             deepResolvePromise(promise2, x, resolvePromise, rejectPromise);
151                         }
152                     } catch (error) {
153                         rejectPromise(error);
154                     }
155                 });
156                 that.onRejectedCallbacks.push(function () {
157                     try {
158                         if (typeof onRejected !== 'function') {
159                             rejectPromise(that.reason);
160                         } else {
161                             var x = onRejected(that.reason);
162                             deepResolvePromise(promise2, x, resolvePromise, rejectPromise);
163                         }
164                     } catch (error) {
165                         rejectPromise(error);
166                     }
167                 });
168                 break;
169             case FULFILLED:
170                 setTimeout(function () {
171                     try {
172                         if (typeof onFulfilled !== 'function') {
173                             resolvePromise(that.value);
174                         } else {
175                             var x = onFulfilled(that.value);
```

```
                            deepResolvePromise(promise2, x,
resolvePromise, rejectPromise);
                        }
                    } catch (error) {
                        rejectPromise(error);
                    }
                }, 0);

                break;
            case REJECTED:
                setTimeout(function () {
                    try {
                        if (typeof onRejected !== 'function') {
                            rejectPromise(that.reason);
                        } else {
                            var x = onRejected(that.reason);
                            deepResolvePromise(promise2, x,
resolvePromise, rejectPromise);
                        }
                    } catch (error) {
                        rejectPromise(error);
                    }
                }, 0);

                break;

        }


    });

    return promise2;

}
MyPromise.resolve = function (val) {
    if (val instanceof MyPromise) {
        return val;
    }

    return new MyPromise(function (resolve) {
        resolve(val);
    });
}

MyPromise.reject = function (reason) {
    return new MyPromise(function (resolve, reject) {
        reject(reason);
    });
}
```

```
223
224 MyPromise.all = function (promiseList) {
225     var resPromise = new MyPromise(function (resolve, reject) {
226         var count = 0;
227         var result = [];
228         var length = promiseList.length;
229
230         if (length === 0) {
231             return resolve(result);
232         }
233
234         promiseList.forEach(function (promise, index) {
235             MyPromise.resolve(promise).then(function (value) {
236                 count++;
237                 result[index] = value;
238                 if (count === length) {
239                     resolve(result);
240                 }
241             }, function (reason) {
242                 reject(reason);
243             });
244         });
245     });
246
247     return resPromise;
248 }
249
250 MyPromise.race = function (promiseList) {
251     var resPromise = new MyPromise(function (resolve, reject) {
252         var length = promiseList.length;
253
254         if (length === 0) {
255             return resolve();
256         } else {
257             for (var i = 0; i < length; i++) {
258                 MyPromise.resolve(promiseList[i]).then(function (value) {
259                     return resolve(value);
260                 }, function (reason) {
261                     return reject(reason);
262                 });
263             }
264         }
265     });
266
267     return resPromise;
268 }
269
270 MyPromise.prototype.catch = function (onRejected) {
271     return this.then(null, onRejected);
```

```
}
// finally , 无论什么状态都会实现fn方法，并且，如果是fulffile，返回都promis也是fulfill，reject，最后也会抛出错误
MyPromise.prototype.finally = function (fn) {
    return this.then(function (value) {
        return MyPromise.resolve(fn()).then(function () {
            return value;
        });
    }, function (error) {
        return MyPromise.resolve(fn()).then(function () {
            throw error
        });
    });
}


module.exports = MyPromise
```