# 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：** 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。
js 是单线程的，其实说的是js 的主线程是单线程的，主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，主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，就产生历史事件循环和消息队列机制。消息队列是一种先进先出的数据结构。当产生任务的时候，会以此将任务推入消息队列，在主线程空闲的时候，依次取出任务并执行，这就是事件循环。
在消息队列中的任务是宏任务，每个宏任务都会管理一个微任务队列。主线程每次从消息队列中取出一个宏任务执行，在执行任务过程中，如果产生微任务，就把微任务放入微任务队列中，在宏任务执行结束之前，依次执行微任务队列中的队列。
实际上宏任务队列还包括普通宏任务队列和延迟任务队列，普通任务队列是先进先出的数据结构，而延迟任务队列则是hashmap 类型的结构，每次主线程会取普通队列中的第一个任务或者是延迟队列中的最老的到时任务执行

# 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:

```
const fp = require('lodash/fp')
const {Maybe, Container} = require('./support')
let maybe = Maybe.of([5,6,1])
let ex1 = () => {
    return fp.map(fp.add(1))
}
console.log(maybe.map(ex1())._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:

```
const fp = require('lodash/fp')
const {Maybe, Container} = require('./support')
let xs = Container.of(['do', 'ray', 'me', 'fa', 'so', 'la', 'ti', 'do'])
let ex2 = ()=>{
    return fp.first
}
console.log(xs.map(ex2())._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：

```
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 = () => {
    return fp.first
}
console.log(safeProp('name')(user).map(ex3())._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)
  }
}
```

# 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**