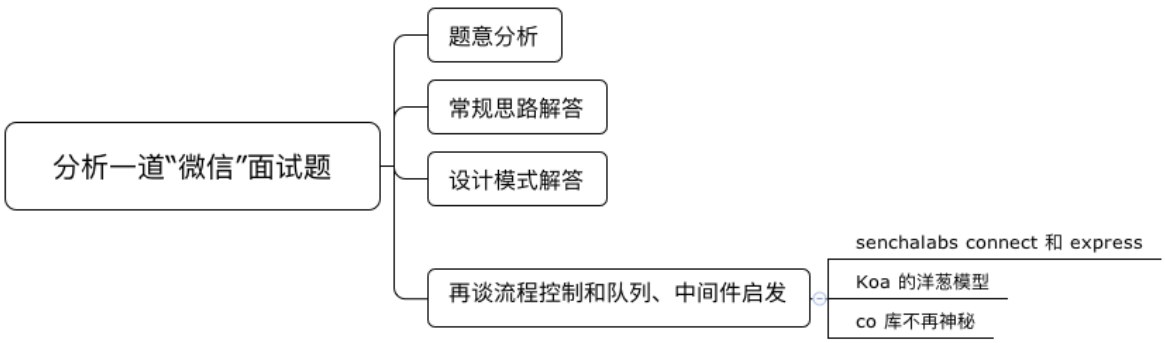
分析一道「微信」面试题

前一段时间，一道疑似「微信」招聘的面试题出现，可能有不少读者已经了解过了。这道题乍一看挺难，但是细细分析却还算简单，我们甚至可以用多种手段解题，用不同思想来给出答案。

网上零零碎碎的有一些解答，但是缺乏全面梳理。我认为通过这道题，有必要将前端多重知识点「融会贯通」，在这里和大家分享。

本讲知识点如下：



题意分析

我们来先看看题目：

实现一个 LazyMan，按照以下方式调用时，得到相关输出：

```
LazyMan("Hank")
// Hi! This is Hank!
```

```
LazyMan("Hank").sleep(10).eat("dinner")
// Hi! This is Hank!
// 等待 10 秒..
// Wake up after 10
// Eat dinner~

LazyMan("Hank").eat("dinner").eat("supper")
// Hi This is Hank!
// Eat dinner~
// Eat supper~

LazyMan("Hank").sleepFirst(5).eat("supper")
// 等待 5 秒
// Wake up after 5
// Hi This is Hank!
// Eat supper
```

当面试者拿到这道题目的时候，乍看题干可能会有点慌张。其实很多面试失败是「自己吓唬自己」，在平时放松状态下写代码，也许解题不在话下。

下面我们就从接到题目开始，剖析应该如何进行分析：

可以把 LazyMan 理解为一个构造函数，在调用时输出参数内容

LazyMan 支持链式调用

链式调用过程提供了以下几个方法：sleepFirst、eat、sleep

其中 eat 方法输出参数相关内容：Eat + 参数

sleep 方法比较特殊，链式调用将暂停一定时间后继续执行，看到这里也许应该想到 setTimeout

sleepFirst 最为特殊，这个任务或者这个方法的 **优先级最高**；调用 sleepFirst 之后，链式调用将暂停一定时间后继续执行。请再次观察题干，尤其是最后

一个 demo，sleepFirst 的输出优先级最高，调用后先等待 5 秒输出 Wake up after 5，再输出 Hi This is Hank!

我们应该如何解这个题目呢，从拿到需求开始进行分析：

先从最简单的，我们可以封装一些基础方法，比如 log 输出、封装 setTimeout 等

因为 LazyMan 要实现一系列调用，**且调用并不是顺序执行的，比如如果 sleepFirst 出现在调用链时，优先执行；同时任务并不是全部都同步执行的**，因此**我们应该实现一个任务队列**，这个队列将调度执行各个任务

因此每次调用 LazyMan 或链式执行时，我们应该将相关调用方法加入到（push）任务队列中，储存起来，后续统一被调度

在写入任务队列时，如果当前的方法为 sleepFirst，那么需要将该方法放到队列的最头处，这应该是一个 unshift 方法

这么一分析，这道题就「非常简单」了。

我们来试图解剖一下这道题目的考察点：

面向对象思想与设计，包括类的使用等

对象方法链式调用的理解和设计

小部分设计模式的设计

因为存在「重复逻辑」，考察代码的解耦和抽象能力

逻辑的清晰程度以及其他编程思维

常规思路解答

基于以上思路，我们给出较为常规的答案，其中代码已经加上了必要的注释：

```
class LazyManGenerator {
 constructor(name) {
   this.taskArray = []

   // 初始化时任务
   const task = () => {
     console.log(`Hi! This is ${name}`)
     // 执行完初始化时任务后，继续执行下一个任务
     this.next()
   }

   // 将初始化任务放入任务队列中
   this.taskArray.push(task)

   setTimeout(() => {
     this.next()
   }, 0)
 }

 next() {
     // 取出下一个任务并执行
   const task = this.taskArray.shift()
   task && task()
 }

 sleep(time) {
   this.sleepTask(time, false)
   // return this 保持链式调用
   return this
 }

 sleepFirst(time) {
   this.sleepTask(time, true)
   return this
 }

 sleepTask(time, prior) {
```

```
    const task = () => {
      setTimeout(() => {
        console.log(`Wake up after ${time}`)
        this.next()
      }, time * 1000)
    }

    if (prior) {
      this.taskArray.unshift(task)
    } else {
      this.taskArray.push(task)
    }
  }

  eat(name) {
    const task = () => {
      console.log(`Eat ${name}`)
      this.next()
    }

    this.taskArray.push(task)
    return this
  }
}

function LazyMan(name) {
  return new LazyManGenerator(name)
}
```

简单分析一下：

LazyMan 方法返回一个 LazyManGenerator 构造函数的实例

在 LazyManGenerator constructor 当中，我们维护了 taskArray 用来存储任务，同时将初始化任务放到 taskArray 当中

还是在 LazyManGenerator constructor 中，将任务的逐个执行即 next 调用放在 setTimeout 中，这样就能够保证在开始执行任务时，taskArray 数组已经填满了任务

我们来看看 next 方法，取出 taskArray 数组中的首项，进行执行

eat 方法将 eat task 放到 taskArray 数组中，注意 eat task 方法需要调用 this.next() 显式调用「下一个任务」；同时返回 this，完成链式调用

sleep 和 sleepFirst 都调用了 sleepTask，不同在于第二个参数：sleepTask 第二个参数表示是否优先执行，如果 prior 为 true，则使用 unshift 将任务插到 taskArray 开头

这个解法最容易想到，也相对来说容易，主要是面向过程。关键点在于对于 setTimeout 任务队列的准确理解以及 return this 实现链式调用的方式。

事实上，sleepTask 应该作为 LazyManGenerator 类的私有属性出现，因为 ES class 暂时 private 属性没有被广泛实现，这里不再追求实现。

设计模式解答

关于这道题目的解答，网上最流行的是一种发布订阅模式的方案。相关代码出处：lazyMan。

但是其实仔细看其实现，也是上一环节中常规解法的变种。虽然说是发布订阅模式，但是其实仍然是 next 思想执行下一个任务的思路，该实现 publish 和 subscribe 方法分别是完成执行任务和注册任务逻辑。我认为这样的代码实现有一点「过度设计」之嫌，更像是往发布订阅模式上去靠，整体流程不够自然。

当然读者仍可参考，并有自己的思考，这里我不再更多分析。

再谈流程控制和队列、中间件启发

这道题目我们给出解法并不算完，更重要也更有价值的是思考、延伸。微信题目较好地考察了候选者的流程控制能力，而流程控制在前端开发者面前也非常重要。

我们看上述代码中的 next 函数，它负责找出 stack 中的下一个函数并执行：

```
next() {
   // 取出下一个任务并执行
   const task = this.taskArray.shift()
   task && task()
}
```

NodeJS 中 connect 类库，以及其他框架的中间件设计也都离不开类似思想的 next。比如生成器自动执行函数 co、redux、koa 也通过不同的实现，可以让 next 在多个函数之间执行完后面的函数再折回来执行 next，较为巧妙。我们具体来看一下。

senchalabs connect 和 express

具体场景：在 Node 环境中，有 parseBody、checkIdInDatabase 等相关中间件，他们组成了 middlewares 数组：

```
const middlewares = [
 function middleware1(req, res, next) {
   parseBody(req, function(err, body) {
     if (err) return next(err);
     req.body = body;
     next();
   });
 },
 function middleware2(req, res, next) {
   checkIdInDatabase(req.body.id, function(err, rows) {
     if (err) return next(err);
     res.dbResult = rows;
     next();
   });
 },
 function middleware3(req, res, next) {
   if (res.dbResult && res.dbResult.length > 0) {
     res.end('true');
   }
```

```
    else {
      res.end('false');
    }
    next();
  }
]
```

当一个请求打开时，我们需要链式调用各个中间件：

```
const requestHandler = (req, res) => {
  let i = 0

  function next(err) {
    if (err) {
      return res.end('error:', err.toString())
    }

    if (i < middlewares.length) {
      middlewares[i++](req, res, next)
    } else {
      return
    }
  }

  // 初始执行第一个中间件
  next()
}
```

基本思路和面试题解法一致：

- 将所有中间件（任务处理函数）储存在一个 list 中
- 循环依次调用中间件（任务处理函数）

senchalabs/connect 这个库做了很好的封装，是 express 等框架设计实现的原始模型。这里我们简单分析一下 senchalabs/connect 这个库的实现。

用法：

首先使用 createServer 方法创建 app 实例，

```
const app = createServer()
```

对应源码：

```
function createServer() {
 function app(req, res, next){ app.handle(req, res, next);
}
 merge(app, proto);
 merge(app, EventEmitter.prototype);
 app.route = '/';
 app.stack = [];
 return app;
}
```

我们看 app 实例「继承」了 EventEmitter 类，实现事件发布订阅，同时 stack 数组来维护各个中间件任务。

接着使用 app.use 来添加中间件：

```
app.use('/api', function(req, res, next) {//...})
```

源码实现：

```
proto.use = function use(route, fn) {
 var handle = fn;
 var path = route;

 // default route to '/'
 if (typeof route !== 'string') {
   handle = route;
   path = '/';
 }
```

```
  // wrap sub-apps
  if (typeof handle.handle === 'function') {
    var server = handle;
    server.route = path;
    handle = function (req, res, next) {
      server.handle(req, res, next);
    };
  }

  // wrap vanilla http.Servers
  if (handle instanceof http.Server) {
    handle = handle.listeners('request')[0];
  }

  // strip trailing slash
  if (path[path.length - 1] === '/') {
    path = path.slice(0, -1);
  }

  // add the middleware
  debug('use %s %s', path || '/', handle.name ||
'anonymous');
  this.stack.push({ route: path, handle: handle });

  return this;
};
```

通过 if...else 逻辑区分出三种不同的 fn 类型：

fn 是一个普通的 function(req,res[,next]){} 函数

fn 是一个普通的 httpServer

fn 是一个普通的是另一个 connect 的 app 对象（sub app 特性）

对于这三种类型，分别转换为 function(req, res, next) {} 的形式，具体我们不再分析。最重要的执行过程是：

```
this.stack.push({ route: path, handle: handle })
```

以及返回：

```
return this
```

以上就完成了中间件即任务的注册，我们有：

```
app.stack = [function1, function2, function3, ...];
```

接下来看看任务的调度和执行。使用方法：

```
app.handle(req, res, out)
```

handle 源码实现：

```
proto.handle = function handle(req, res, out) {
 var index = 0;
 var protohost = getProtohost(req.url) || '';
 var removed = '';
 var slashAdded = false;
 var stack = this.stack;

 // final function handler
 var done = out || finalhandler(req, res, {
   env: env,
   onerror: logerror
 });

 // store the original URL
 req.originalUrl = req.originalUrl || req.url;

 function next(err) {
```

```
    // ...
  }

  next();
};
```

源码导读：out 参数是关于 sub app 的特性，这个特性可以暂时忽略，我们暂时不关心。handle 实现我们并不陌生，它构建 next 函数，并触发第一个 next 执行。

next 实现：

```
function next(err) {
    if (slashAdded) {
      req.url = req.url.substr(1);
      slashAdded = false;
    }

    if (removed.length !== 0) {
      req.url = protohost + removed +
req.url.substr(protohost.length);
      removed = '';
    }

    // next callback
    var layer = stack[index++];

    // all done
    if (!layer) {
      defer(done, err);
      return;
    }

    // route data
    var path = parseUrl(req).pathname || '/';
    var route = layer.route;
```

```
  // skip this layer if the route doesn't match
  if (path.toLowerCase().substr(0, route.length) !==
route.toLowerCase()) {
    return next(err);
  }

  // skip if route match does not border "/", ".", or end
  var c = path.length > route.length &&
path[route.length];
  if (c && c !== '/' && c !== '.') {
    return next(err);
  }

  // trim off the part of the url that matches the route
  if (route.length !== 0 && route !== '/') {
    removed = route;
    req.url = protohost + req.url.substr(protohost.length
+ removed.length);

    // ensure leading slash
    if (!protohost && req.url[0] !== '/') {
      req.url = '/' + req.url;
      slashAdded = true;
    }
  }

  // call the layer handle
  call(layer.handle, route, err, req, res, next);
}
```

源码导读：

取出下一个中间件

```
var layer = stack[index++]
```

如果当前请求路由和 handler 不匹配，则跳过：

```
if (path.toLowerCase().substr(0, route.length) !==
route.toLowerCase()) {
 return next(err);
}
```

若匹配，则执行 call 函数，call 函数实现：

```
function call(handle, route, err, req, res, next) {
 var arity = handle.length;
 var error = err;
 var hasError = Boolean(err);

 debug('%s %s : %s', handle.name || '', route,
req.originalUrl);

 try {
   if (hasError && arity === 4) {
     // error-handling middleware
     handle(err, req, res, next);
     return;
   } else if (!hasError && arity < 4) {
     // request-handling middleware
     handle(req, res, next);
     return;
   }
 } catch (e) {
   // replace the error
   error = e;
 }

 // continue
 next(error);
}
```

注意：我们使用了 try...catch 包裹逻辑，这是很必要的容错思维，这样第三方中间件的执行如果出错，不至于打挂我们的应用。

较为巧妙的一点是：function(err, req, res, next){} 形式为错误处理函数，function(req, res, next){} 为正常的业务逻辑处理函数。因此通过 Function.length 来判断当前 handler 是否为容错函数，来做到参数的传入。

call 函数是 next 函数的核心，它是一个执行者，并在最后的逻辑中继续执行 next 函数，完成中间件的顺序调用。

NodeJS 的框架 express，实际就是 senchalabs connect 的升级版，通过对 connect 源码的学习，我们应该更加清楚流程的调度和控制，再去看 express 就轻而易举了。

Senchalabs connect 用流程控制库的回调函数及中间件的思想来解耦回调逻辑；Koa 则是用 generator 方法解决回调问题（最新版使用 async/await）。事实上，也可以用事件、Promise 的方式实现，下一环节，我们就分析 Koa 的洋葱模型。

Koa 的洋葱模型

对 Koa 中间的洋葱模型的分析文章上不少，著名的洋葱圈图示我也不在自己画了，具体使用不再介绍，不了解的读者请先自行学习。

我想先谈一下面向切面编程（AOP），在 JavaScript 语言为例，一个简单的示例：

```
Function.prorotype.before = function (fn) {
 const self = this
 return function (...args) {
   console.log('')
   let res = fn.call(this)
   if (res) {
     self.apply(this, args)
   }
 }
```

```
}

Function.prototype.after = function (fn) {
 const self = this
 return function (...args) {
   let res = self.apply(this, args)
   if (res) {
     fn.call(this)
   }
 }
}
```

这样的代码实现，是我们能够在执行某个函数 fn 之前，先执行某段逻辑；在某个函数 fn 之后，再去执行另一段逻辑。其实是一种简单中间件流程控制的体现。不过这样的 AOP 有一个问题：无法实现异步模式。

那么如何实现 Koa 的异步中间件模式呢？即某个中间件执行到一半，交出执行权，之后再回来继续执行。我们直接看源码分析，这段源码实现了 Koa 洋葱模型中间件：

```
function compose(middleware) {
 return function *(next) {(
   if (!next) next = noop();

   var i = middleware.length;

   while (i--) {
     next = middleware[i].call(this, next);
     console.log('isGenerator:', (typeof next.next ===
'function' && typeof next.throw === 'function')); // true
   }

   return yield *next;
 }
}

function *noop(){}
```

其中，一个中间件的写法类似：

```
app.use(function *(next){
 var start = new Date;
 yield next;
 var ms = new Date - start;
 this.set('X-Response-Time', ms + 'ms');
});
```

这是一个很简单的记录 response time 的中间件，中间件跳转的信号是 yield next。

较新版本的 Koa 已经改用 async/await 实现，思路也是完全一样的，当然看上去更加优雅：

```
function compose (middleware) {
 if (!Array.isArray(middleware)) throw new
TypeError('Middleware stack must be an array!')
 for (const fn of middleware) {
   if (typeof fn !== 'function') throw new
TypeError('Middleware must be composed of functions!')
 }

 return function (context, next) {
   let index = -1
   return dispatch(0)

   function dispatch (i) {
     if (i <= index) return Promise.reject(new
Error('next() called multiple times'))
     index = i
     let fn = middleware[i]
     if (i === middleware.length) {
       fn = next
     }
     if (!fn) return Promise.resolve()
```

```
      try {
        return Promise.resolve(fn(context, function next () {
          return dispatch(i + 1)
        }))
      } catch (err) {
        return Promise.reject(err)
      }
    }
  }
}
```

我们来重点解读一下这个版本的实现：

compose 传入的 middleware 参数必须是数组，否则抛出错误

middleware 数组的每一个元素必须是函数，否则抛出错误

compose 返回一个函数，保存对 middleware 的引用

compose 返回函数的第一个参数是 context，所有中间件的第一个参数就是传入的 context

compose 返回函数的第二个参数是 next 函数，next 是实现洋葱模型的关键

index 记录当前运行到第几个中间件

执行第一个中间件函数：return dispatch(0)

dispatch 函数中，参数 i 如果小于等于 index，说明一个中间件中执行了多次 next，我们进行报错，由此可见一个中间件函数内部不允许多次调用 next 函数

取出中间件函数 fn = middleware[i]

如果 i === middleware.length，说明执行到了圆心，将 next 赋值给 fn

因为 async 需要后面是 Promise，我们包一层 Promise

next 函数是固定的，它可以执行下一个中间件函数

```
function next () {
 return dispatch(i + 1)
}
```

如果读者不好理解，可以参考应用示例：

```
async function middleware1(ctx, next) {
 console.log('1')
 await next()
 console.log('2')
};

async function middleware2(ctx, next) {
 console.log('3')
 await next()
 console.log('4')
};
```

如果读者还是难以理解，我给出一个简版逻辑：

```
function compose (middleware) {
 return dispatch(0)
 function dispatch(i) {
   fn = middleware[i]
   if(!fn) return
   return fn(() => dispatch(i + 1))
 }
}
```

co 库不再神秘

说到流程控制，也少不了大名鼎鼎的 co 库。co 函数库是 TJ 大神基于 ES6 generator 的异步解决方案，因此这里需要读者熟练掌握 ES6 generator。目前虽然 co 库可能不再「流行」，但是了解其实现，模拟类似场景也是非常有必要的。

我们这里不解读其源码，而是实现一个类似的自动执行 generator 的方案：

```
const runGenerator = generatorFunc => {
 const it = generatorFunc()
 iterate(it)

 function iterate (it) {
   step()

   function step(arg, isError) {
     const {value, done} = isError ? it.throw(arg) :
it.next(arg)

     let response

     if (!done) {
       if (typeof value === 'function') {
         response = value()
       } else {
         response = value
       }

       Promise.resolve(response).then(step, err =>
step(err, true))
     }
   }
 }
}
```

代码解读：

runGenerator 函数接受一个生成器函数 generatorFunc

运行 generatorFunc 得到结果，并通过 iterate 函数，迭代该生成器结果

iterate 函数中执行 step 函数，step 函数的第一个参数 arg 是上一个 yield 右表达式的「求出的值」，即下面对应的 response

这里需要考虑 response 的求值过程，它通过 value 计算得来，value 是 yield 右侧的值，它有这么几种情况：

yield new Promise()，value 是一个 promise 实例，那么 response 就是该 Promise 实例 resolve 后的值

yield () => {return value}，value 是一个函数，那么 response 就是执行该函数后的返回值

yield value，value 是一个普通值，那么 response 就是该值

我们最终统一利用 Promise.resolve 的特性，对 response 进行处理，并递归（迭代）调用 step

同时利用 step 函数 arg 参数，赋值给上一个 yield 的左表达式值，并返回下一个 yield 右表达式的值

执行代码：

```
function* gen1() {
 yield console.log(1)
 yield console.log(2)
 yield console.log(3)
}

runGenerator(gen1)
```

或者：

```
function* gen2() {
 var value1 = yield Promise.resolve('promise')
```

```
 console.log(value1)

 var value2 = yield () => Promise.resolve('thunk')
 console.log(value2)

 var value3 = yield 2
 console.log(value3)
}

runGenerator(gen2);
```

最后还是附上 co 的实现：

```
function co(gen) { // co 接受一个 generator 函数
   var ctx = this
   var args = slice.call(arguments, 1)

   return new Promise(function(resolve, reject) { // co 返回一个 Promise 对象
       if(typeof gen === 'function') gen = gen.apply(ctx, args) // gen 为 generator 函数，执行该函数
       if(!gen || typeof gen.next !== 'function') return resolve(gen) // 不是则返回并更新 Promise 状态为 resolve

       onFulfilled() // 将 generator 函数的 next 方法包装成 onFulfilled，主要是为了能够捕获抛出的异常

       /**
         * @param {Mixed} res
         * @return {Promise}
         * @api private
        */
       function onFulfilled(res) {
           var ret;
           try {
               ret = gen.next(res)
           } catch (err) {
```

```
            return reject(err)
        }
        next(ret)
    }

    /**
      * @param {Error} err
      * @return {Promise}
      * @api private
     */
    function onRejected(err) {
        var ret
        try {
            ret = gen.throw(err)
        } catch (err) {
            return reject(err)
        }
        next(ret)
    }

    /**
      * Get the next value in the generator,
     * return a promise.
     *
     * @param {Object} ret
     * @return {Promise}
     * @api private
     */
    function next(ret) {
        if(ret.done) return resolve(ret.value)
        var value = toPromise.call(ctx, ret.value) //
if (isGeneratorFunction(obj) || isGenerator(obj)) return
co.call(this, obj);
        if(value && isPromise(value)) return
value.then(onFulfilled, onRejected)
        return onRejected(new TypeError('You may only
yield a function, promise, generator, but the following
```

```
object was passed: ' + String(ret.value) + '"'))
        }
    })
}
```

如果读者对于以上内容理解有困难，那么我建议还是从 generator 等最基本的概念切入，不必心急，慢慢反复体会。

总结

这道「著名」的「微信」面试题，绝不只是网上分析的几行代码答案那么简单，本讲我们从这道题目出发，分析了几种解决方案。更重要的是，在解决方案的基础上，我们重点剖析了 JavaScript 处理任务流程、控制触发逻辑的方方面面。也许在小型传统页面应用中，这样「相对复杂」的处理场景并不多见，但是在大型项目、富交互项目、后端 NodeJS 中非常重要，尤其是中间件思想、洋葱模型是非常典型的编程思路，希望读者能认真体会。

最后我们分析了 generator 以及 Koa 中间件实现原理，也许读者在平时基础业务开发中接触不到这些知识，但是请想一想 redux-saga 的实现、中间件的编写，其实都是这些内容运用体现。进阶即是如此，如果不掌握好这些「难啃」的知识，那么永远无法写出优秀的框架和解决方案。

点击查看下一节

离不开的网络基础