JS 异步处理机制

李京 @ 36氪

异步是怎么来的?

简要介绍为什么会有异步以及我们为什么关心异步问题

单线程 & 异步？

单线程 vs. 多线程

JavaScript 运行在 JavaScript 引擎（JavaScript Engine）中，并且是单线程的。

同步 vs. 异步

通俗理解

- 同步：函数返回就能立即得到预期结果
- 异步：函数返回不能立即得到预期结果

非阻塞

JavaScript 处理 I/O 通常由事件或者回调函数实现。例如等待 ajax 返回时，仍然接收用户输入等。

运行时概念

调用栈 # Call Stack

对象堆 # Heap

任务队列 # Task Queue

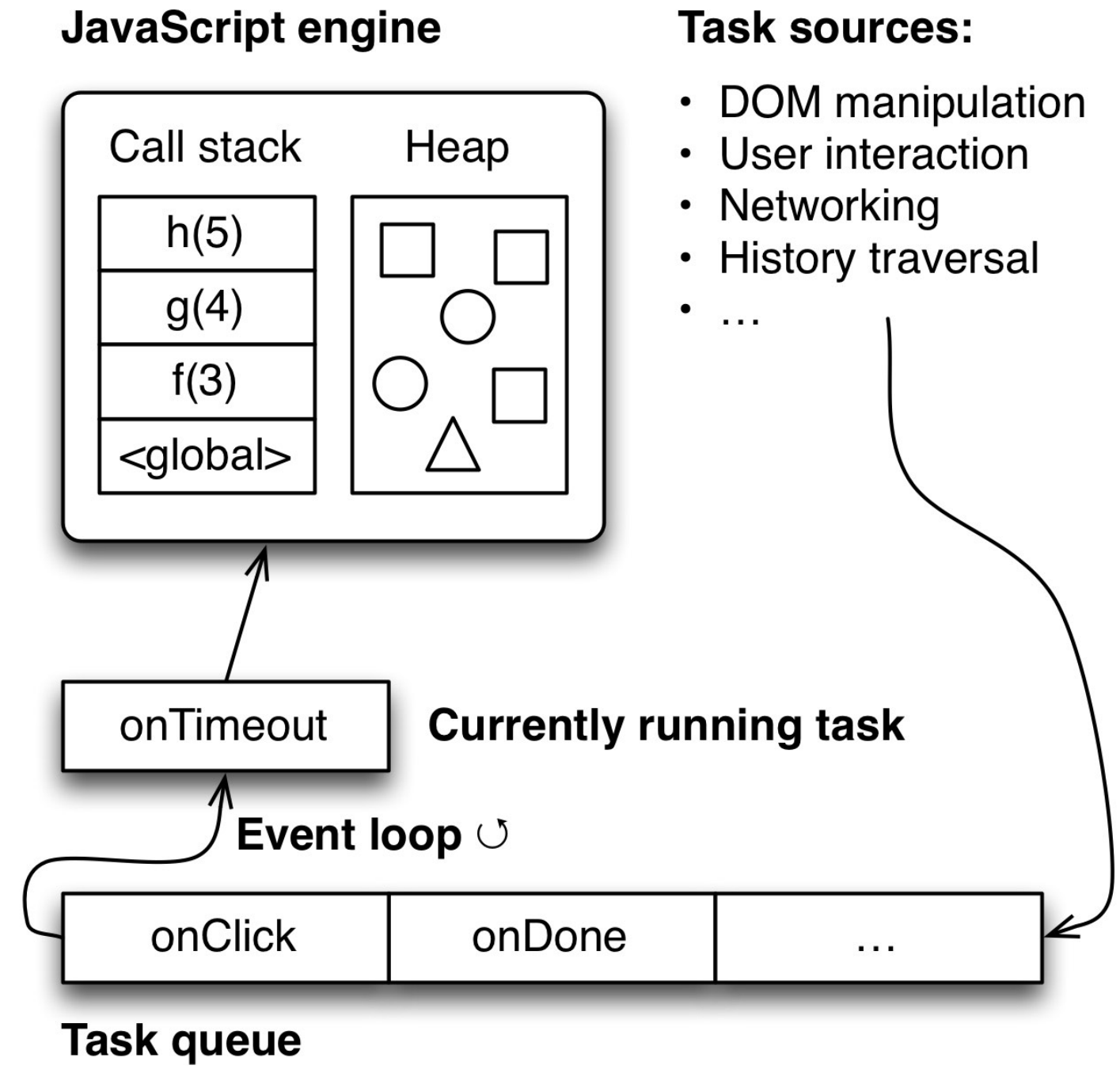
WEB APIS

通常分为两类：I/O 函数 和 计时函数

DOM 事件

Ajax

setTimeout 等


JavaScript engine
Task sources:
• DOM manipulation
• User interaction
• Networking
• History traversal
• …
Call stack
Heap
h(5)
g(4)
f(3)
<global>
onTimeout
Currently running task
Event loop
onClick
onDone
…
Task queue

调用栈

是指由调用函数形成的 frames 栈

MDN 链接

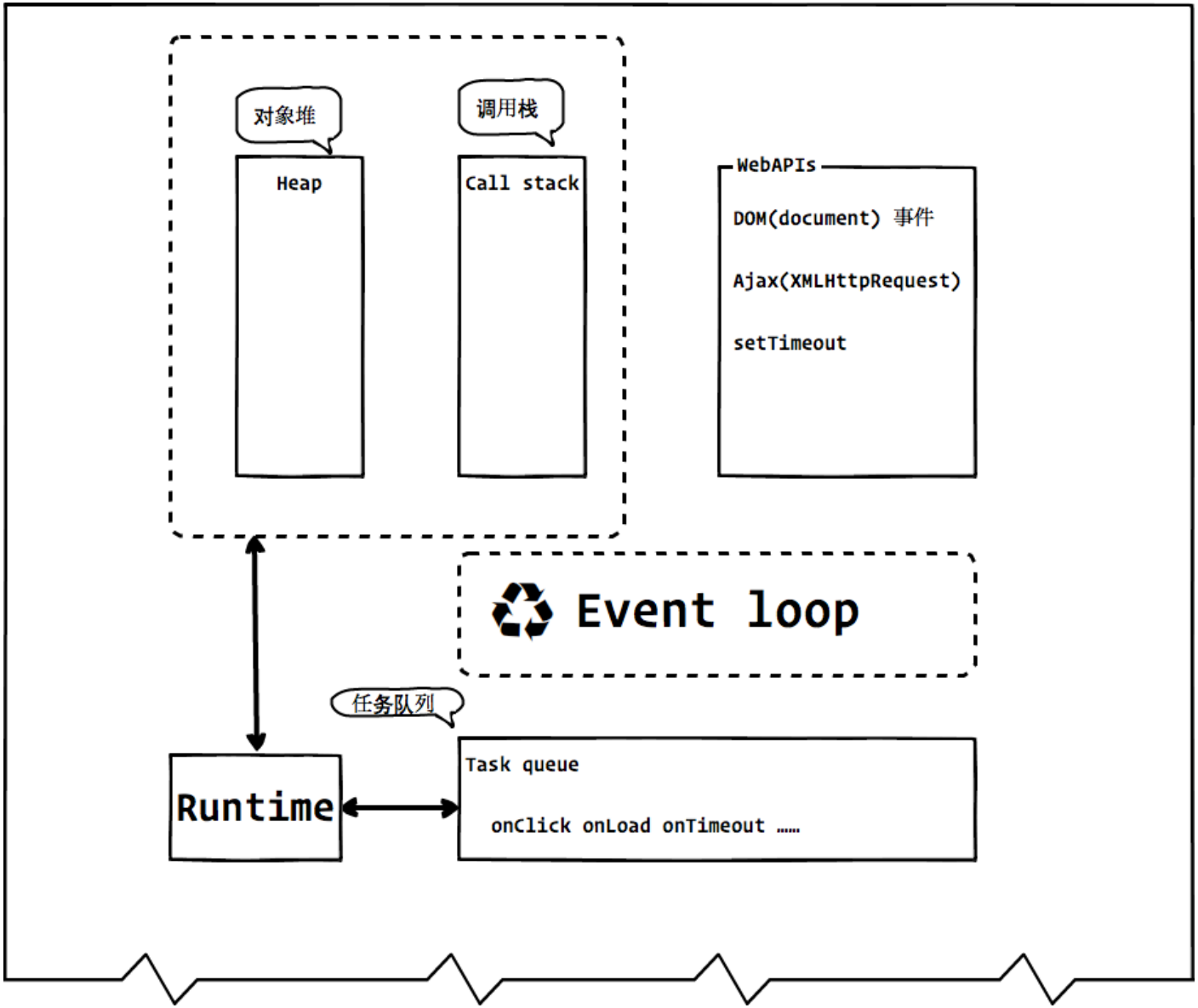
对象堆

对象被分配在一个堆中，一个用以表示一个内存中大的未被组织的区域

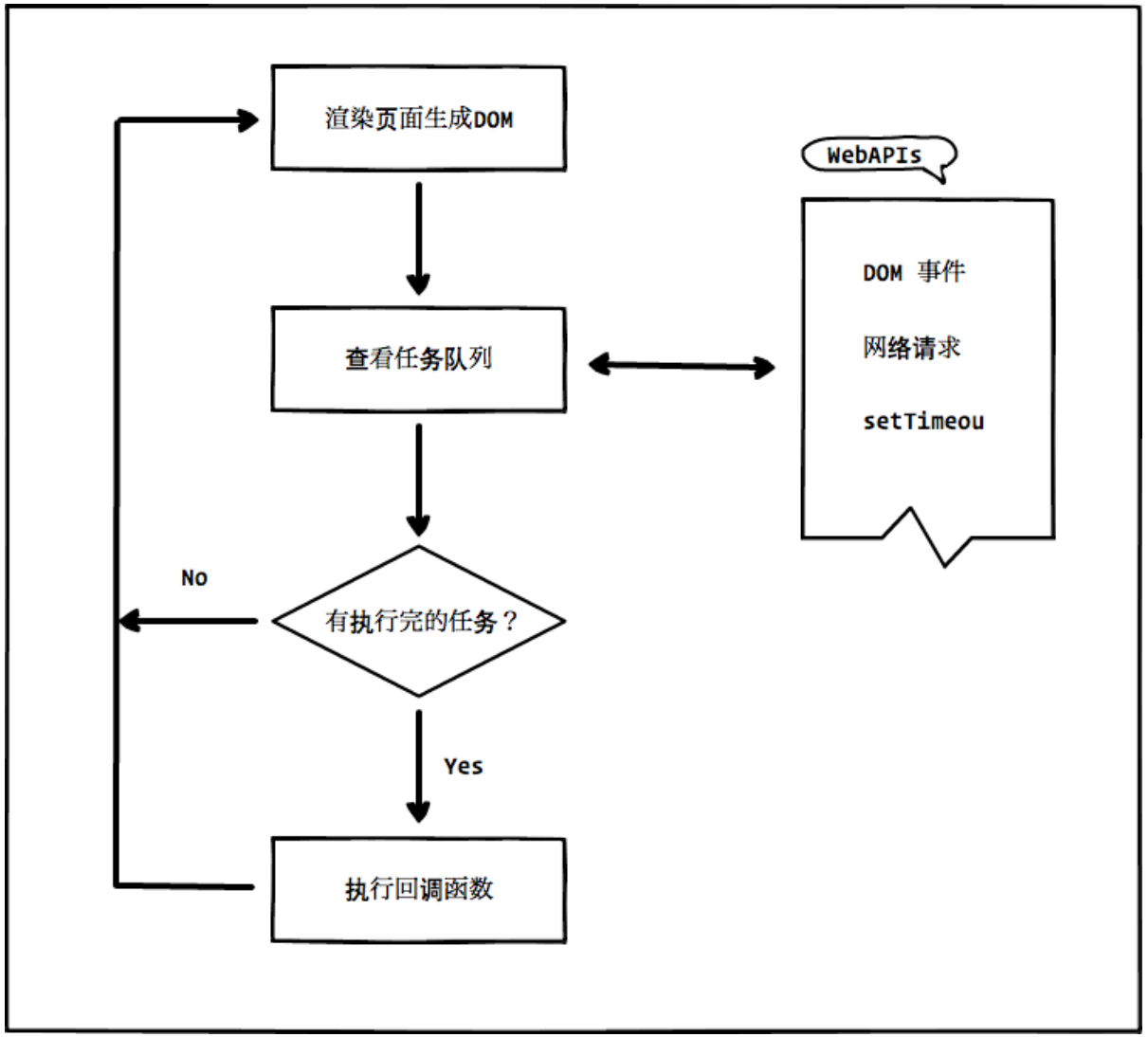
任务队列

JavaScript 运行时包含一个待处理任务的队列。每一条任务都与一个函数相关联

我的理解


对象堆
Heap
调用栈
Call stack
WebAPIs
DOM(document) 事件
Ajax(XMLHttpRequest)
setTimeout
Event loop
任务队列
Runtime
Task queue
onClick onLoad onTimeout ……

大致流程


渲染页面生成DOM
查看任务队列
有执行完的任务？
No
Yes
执行回调函数
WebAPIs
DOM 事件
网络请求
setTimeou

分解一下大致流程：

- 主线程执行任务，形成调用栈；
- 被触发的异步事件后，如果异步事件有相应的回调函数，该函数被推入任务队列中；
- 调用栈栈清空后，读取任务队列中的任务并执行；
- 重复以上。

异步编程方法

- 回调函数（Callback）
- Promise
- ES6 Generator
- async / await

回调函数

```
function queryAPI(url, sucCallback, errCallback) {
    var xhr, result;
    xhr = new XMLHttpRequest();
    xhr.open('GET', url, true); // 第三个参数为 async：意味着是否执行异步操作
    xhr.onload = function(e) {
       if (xhr.status === 200) {
            sucCallback(JSON.parse(xhr.responseText));
        } else {
            errCallback(new Error(xhr.statusText));
        }
    };
    xhr.onerror = function () {
        errCallback(new Error(xhr.statusText));
    };

    xhr.send();
```

好处

- 好理解
- 容易控制

坏处

- 不优雅
- 异常处理复杂
- 回调地狱

PROMISE

```
function queryAPI(url) {
    return new Promise(function(resolve, reject) {
        var xhr, result;
        xhr = new XMLHttpRequest();
        xhr.open('GET', url, true); // 第三个参数为 async：意味着是否执行异步操作
        xhr.onload = function () {
            if (xhr.status === 200) {
                resolve(JSON.parse(xhr.responseText));
            } else {
                reject(new Error(xhr.statusText));
            }
        };
        xhr.onerror = function () {
            reject(new Error(xhr.statusText));
        };
```

PROMISE

A promise represents the eventual result of an asynchronous operation. —— Promises A+

PROMISE 对象

Promise 对象是一个返回值的代理。

异步方法返回一个包含了原返回值的 Promise 对象。

它允许你为异步操作的成功返回值或者失败信息指定处理方法。

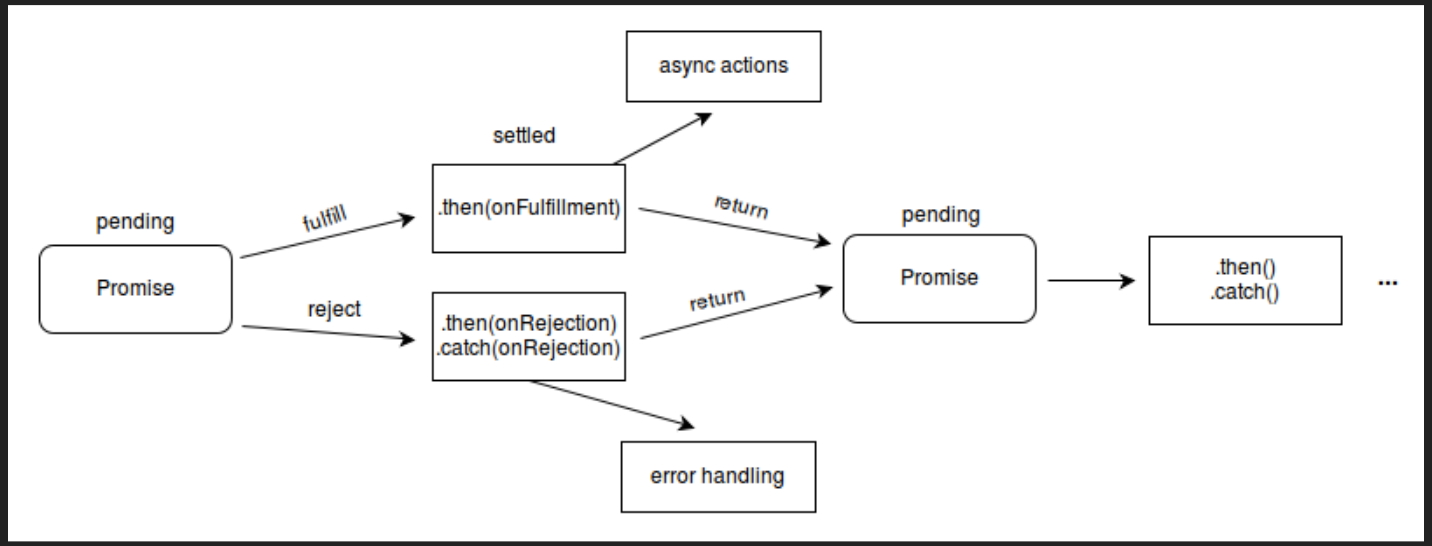
PROMISE 状态

- pending： 初始状态
- fulfilled： 成功地操作
- rejected： 失败的操作
- settled： fulfilled(成功) 或 rejected(失败)。

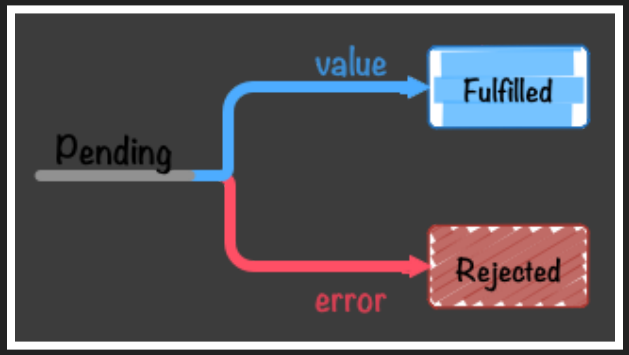
```
var promise = new Promise(function(resolve, reject) {
    // 异步处理
    // 处理结束后、调用resolve 或 reject
    // resolve 时：onFulfilled 被调用
    // reject 时：onRejected 被调用
});

promise.then(onFulfilled, onRejected)
```

PROMISE 状态转换



- 当 Promise 对象的状态发生转换时，promise.then 绑定的方法被调用。
- 因为Promise.prototype.then和 Promise.prototype.catch方法返回 promises对象, 所以它们可以被链式调用。



常用库

- q
- bluebird
- when
- ...

ES6 GENERATOR

在 ES6 中定义一个生成器函数很简单，在 function 后跟上「*」即可：

```
function* foo1() { };
function *foo2() { };
function * foo3() { };

foo1.toString(); // "function* foo1() { }"
foo2.toString(); // "function* foo2() { }"
foo3.toString(); // "function* foo3() { }"
foo1.constructor; // function GeneratorFunction() { [native code] }
```

调用一个生成器函数并不马上执行它的主体，而是返回一个这个生成器函数的迭代器（iterator）对象。

生成器函数通常和 yield 关键字同时使用。函数执行到每个 yield 时都会中断并返回 yield 的右值（通过 next 方法返回对象中的 value 字段）。下次调用 next，函数会从 yield 的下一个语句继续执行。等到整个函数执行完，next 方法返回的 done 字段会变成 true。

```
function* numGenerator() {
    var i = 0;
    console.info('Generator function start');
    while(i < 3) {
        console.info('Yield start');
        yield i++;
        console.info('Yield end');
    }
    console.info('Generator function end');
}

var result = numGenerator();
```

NEXT

next也可以接受一个任意参数，该参数将作为上一个yield的返回值。

```
function* generateNaturalNumber() {
    var i = 0;
    while(i <= 100) {
        var j = yield i;
        j && (i = j);
        i++;
    }
}
```

YIELD*

yield* 将执行权托管给另一个迭代器。

```
function* anotherGenerator(i) {
  yield i + 1;
  yield i + 2;
  yield i + 3;
}

function* generator(i){
  yield i;
  yield* anotherGenerator(i);
  yield i + 10;
}

var result = generator(10);
```

使用CO进行异步流程控制

```
var co = require('co');

co(function *(){
  // resolve multiple promises in parallel
  var a = Promise.resolve(1);
  var b = Promise.resolve(2);
  var c = Promise.resolve(3);
  var res = yield [a, b, c];
  console.log(res);
}).catch(onerror)

function onerror(err) {
  console.error(err.stack);
}
```

ASYNC / AWAIT

async 函数就是 Generator 函数的语法糖

基本规则

- async 表示这是一个async函数，await只能用在这个函数里面；
- await 表示在这里等待promise返回结果了，再继续执行；
- await 后面跟着的应该是一个promise对象。

举例

```
var sleep = function (time) {
    return new Promise(function (resolve, reject) {
        setTimeout(function () {
            resolve();
        }, time);
    })
};

var start = async function () {
    // 在这里使用起来就像同步代码那样直观
    console.log('start');
    await sleep(3000);
    console.log('end');
};

start();
```

结束 & THANKS