# [译]JavaScript异步编程的演化

原文 <https://blog.risingstack.com/asynchronous-javascript/>

## 异步函数经过很长的发展旅程，终于即将来临。不久以前我们还在使用calllback回调，直到 Promise/A+ 规范提出，然后是genertor函数，现在是asyc函数。

现在我们回头看一看这些年异步js怎么形成的。

## Callbacks

这一切都始与回调。

## 异步的JavaScript

异步编程，就像我们现在所了解的JavaScript，函数是第一类公民(http://blog.leapoahead.com/2015/09/19/function-as-first-class-citizen/) ：函数可以作为参数被传递到其他的函数。这是callback函数怎么产生的：如果你听过一个函数作为一个参数传递到另外一个函数，在一个函数之内你可以调用它，当这个函数完成。也可以没有返回值的时候，仅仅使用这个参数调用其他的函数。

Something.save(function(err) {

if (err) {

//error handling return;

}

console.log('success');});

在nodejs的核心部分这些被称为error-first回调- 在npm上大多数核心模块也是用这种方式。

Callbacks方式的问题：

如果使用不当很容易的形成回调地狱和面条式代码。

很容易忽视错误处理。

不能使用return 声明返回值, 也不能使用 throw 关键字。

因为这些情况很容易的发生，JavaScript社区开始寻找使异步JavaScript开发更加容易的解决方案。

其中一个答案是 async 模块。如果你使用的很多的 callbacks，你应该知道如果使用异步函数来映射数组，让异步变为同步将会多么复杂。这时感谢”[Caolan McMahon](https://twitter.com/caolan)” async 模块的出现。

在async之内，你可以很容易的做这些，像这样：

async.map([1, 2, 3], AsyncSquaringLibrary.square,

function(err, result){

// result will be [1, 4, 9]});

它仍然是不是很容易读和写 - 因此Promises来临.

## Promises

当前JavaScript Promises规范需要追溯到2012年和可以在ES6中使用- 然而Promises不是被发明在JavaScript社区，术语来自 [Daniel P. Friedman](https://en.wikipedia.org/wiki/Daniel_P._Friedman) 在1976年.

一个Promise代表一个异步操作最后的结果。

以前的回调可以用Promise这样写：

Something.save()

.then(function() {

console.log('success');

})

.catch(function() {

//error handling })

你可能注意到Promises也可以使用在运用callbacks的地方。Then和catch回调被注册后将被调用当异步操作完成或者它没有被实现的时候。Promsie另一个好的特性是它可以被链式调用:

saveSomething()

.then(updateOtherthing)

.then(deleteStuff)

.then(logResults);

在我们运用Promise的时候，在环境中没有实现Promise的时候你也许必须使用 polyfills。在这种情况下一个好的选择是运使用[bluebird](https://github.com/petkaantonov/bluebird).这些库提供了比原生Promise更多的功能-甚至在这些情况下限制你使用 **Promises/A+** 规范的特性。

但是为什么你不应该使用这个语法糖了？看下Promise：[The Extension Problem](http://blog.getify.com/promises-part-4/).获取更多的Promise说明参阅[The Extension Problem](http://blog.getify.com/promises-part-4/).

你也许会问：在大多数库仅仅只导出了callback接口我应该怎样使用Promise。

这是很容易的，你仅仅需要使用Promise在原始的callback函数包一层。

function saveToTheDb(value) {

return new Promise(function(resolve, reject) {

db.values.insert(value, function(err, user) { // remember error first ;) if (err) {

return reject(err); // don't forget to return here }

resolve(user);

})

}}

一些库/框架支持两种方式，同时提供一个callback和一个Promise接口。如果你在封装一个库，同时支持这两种这是一个好的实践。你可以很容易这样做，像这样子：

function foo(cb) {

if (cb) {

return cb();

}

return new Promise(function (resolve, reject) {

});}

或者更简单，你可以开始只提供一个Promise接口并且提供一个向后兼容的工具如 [callbackify](https://www.npmjs.com/package/callbackify)。[callbackify](https://www.npmjs.com/package/callbackify)主要做一些相同的代码【片段展示，但是用一种更通用的方式。

## Generators / yield

JavaScript Generator 是一个相对较新的概念，他们被介绍在ES6中。

当你执行函数的时候，你可以在任何地方暂停执行，可以计算一些东西和做一些东西，然后返回继续做他，甚至有值并且继续执行，这时很好的特性。

enerator函数是这样得。我们调用一个generator函数的时候，它还没开始执行的时候，我们将必须手动迭代它。

function\* foo () {

var index = 0;

while (index < 2) {

yield index++;

}}var bar = foo();

console.log(bar.next()); // { value: 0, done: false } console.log(bar.next()); // { value: 1, done: false } console.log(bar.next()); // { value: undefined, done: true }

如果你想更容易的运用generators在写异步JavaScript，你可以运用co。

co(function\* (){

yield Something.save();}).then(function() {

// success}).catch(function(err) {

//error handling});

你也许会问：在并行的时候发生了什么操作呢？答案比你想象的简单（答案就是Promise.all）

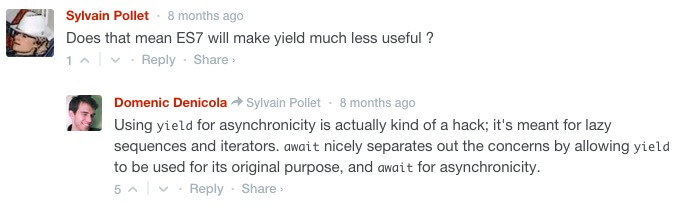
yield [Something.save(), Otherthing.save()];

## Async / await

Async 函数被说明在ES7-并且当前仅仅可以在像babel的

转换器中使用。（免责：现在我们讨论的是async关键字，而不是async包）

总之：运用async关键字，我们可以做我们使用co和generators 组合hacking的任何事。



async函数使用Promise-这是为什么async函数将使用一个Promise返回。

因此如果我们想在以前的例子上做相同的事，我们必须像下面一样重写我们的片段：

async function save(Something) {

try {

await Something.save()

} catch (ex) {

//error handling }

console.log('success');}

正如你使用async函数所看到的那样，你必须将async关键字放在函数声明之前。然后，你就可以使用await关键字在你刚刚创建的异步函数里面。

使用async函数同步执行是和yield 方式非常相似的-除了Promise.all，你必须调用它。

koa早已支持async函数，因此你现在可以尝试运用它和babel。

import koa from koa; let app = koa();

app.experimental = true;

app.use(async function (){

this.body = await Promise.resolve('Hello Reader!')})

app.listen(3000);

## 进一步阅读

现在，大多数新项目中我们使用 [Hapi with generators](https://blog.risingstack.com/hapi-on-steroids-using-generator-functions-with-hapi/)-和koa一起。