

ECMAScript 6 入门

作者：阮一峰

授权：署名-非商用许可证



目录

- 0.前言
- 1.ECMA Script 6简介
- 2.let 和 const 命令
- 3.变量的解构赋值
- 4.字符串的扩展
- 5.字符串的新增方法
- 6.正则的扩展
- 7.数值的扩展
- 8.函数的扩展
- 9.数组的扩展
- 10.对象的扩展
- 11.对象的新增方法
- 12.Symbol
- 13.Set 和 Map 数据结构
- 14.Proxy
- 15.Reflect
- 16.Promise 对象
- 17.Iterator 和 for...of 循环
- 18.Generator 函数的语法
- 19.Generator 函数的异步应用
- 20.async 函数
- 21.Class 的基本语法
- 22.Class 的继承
- 23.Module 的语法
- 24.Module 的加载实现
- 25.编程风格
- 26.读懂规格
- 27.异步遍历器
- 28.ArrayBuffer
- 29.最新提案
- 30.Decorator
- 31.参考链接

其他

- 源码
- 修订历史
- 反馈意见

Generator 函数的异步应用

- 1.传统方法
- 2.基本概念
- 3.Generator 函数

《ES6 实战教程》 深入学习一线大厂必备 ES6 技能。VIP 教程限时免费领取。 [← 立即查看](#)

异步编程对 JavaScript 语言太重要。JavaScript 语言的执行环境是“单线程”的，如果没有异步编程，根本没法用，非卡死不可。本章主要介绍 Generator 函数如何完成异步操作。

1. 传统方法

ES6 诞生以前，异步编程的方法，大概有下面四种。

- 回调函数
- 事件监听
- 发布/订阅
- Promise 对象

Generator 函数将 JavaScript 异步编程带入了一个全新的阶段。

2. 基本概念

异步

所谓"异步"，简单说就是一个任务不是连续完成的，可以理解成该任务被人为分成两段，先执行第一段，然后转而执行其他任务，等做好了准备，再回过头执行第二段。

比如，有一个任务是读取文件进行处理，任务的第一段是向操作系统发出请求，要求读取文件。然后，程序执行其他任务，等到操作系统返回文件，再接着执行任务的第二段（处理文件）。这种不连续的执行，就叫做异步。

相应地，连续的执行就叫做同步。由于是连续执行，不能插入其他任务，所以操作系统从硬盘读取文件的这段时间，程序只能干等着。

回调函数

JavaScript 语言对异步编程的实现，就是回调函数。所谓回调函数，就是把任务的第二段单独写在一个函数里面，等到重新执行这个任务的时候，就直接调用这个函数。回调函数的英语名字 `callback`，直译过来就是"重新调用"。

读取文件进行处理，是这样写的。

```
fs.readFile('/etc/passwd', 'utf-8', function (err, data) {  
  if (err) throw err;  
  console.log(data);  
});
```

上面代码中，`readFile` 函数的第三个参数，就是回调函数，也就是任务的第二段。等到操作系统返回了 `/etc/passwd` 这个文件以后，回调函数才会执行。

一个有趣的问题是，为什么 Node 约定，回调函数的第一个参数，必须是错误对象 `err`（如果没有错误，该参数就是 `null`）？

原因是执行分成两段，第一段执行完以后，任务所在的上下文环境就已经结束了。在这以后抛出的错误，原来的上下文环境已经无法捕捉，只能当作参数，传入第二段。

Promise

回调函数本身并没有问题，它的问题出现在多个回调函数嵌套。假定读取 `A` 文件之后，再读取 `B` 文件，代码如下。

```
fs.readFile(fileA, 'utf-8', function (err, data) {
  fs.readFile(fileB, 'utf-8', function (err, data) {
    // ...
  });
});
```

不难想象，如果依次读取两个以上的文件，就会出现多重嵌套。代码不是纵向发展，而是横向发展，很快就会乱成一团，无法管理。因为多个异步操作形成了强耦合，只要有一个操作需要修改，它的上层回调函数和下层回调函数，可能都要跟着修改。这种情况就称为“回调函数地狱”（callback hell）。

Promise 对象就是为了解决这个问题而提出的。它不是新的语法功能，而是一种新的写法，允许将回调函数的嵌套，改成链式调用。采用 Promise，连续读取多个文件，写法如下。

```
var readFile = require('fs-readfile-promise');

readFile(fileA)
  .then(function (data) {
    console.log(data.toString());
  })
  .then(function () {
    return readFile(fileB);
  })
  .then(function (data) {
    console.log(data.toString());
  })
  .catch(function (err) {
    console.log(err);
  });
```

上面代码中，我使用了 `fs-readfile-promise` 模块，它的作用就是返回一个 Promise 版本的 `readFile` 函数。Promise 提供 `then` 方法加载回调函数，`catch` 方法捕捉执行过程中抛出的错误。

可以看到，Promise 的写法只是回调函数的改进，使用 `then` 方法以后，异步任务的两段执行看得更清楚了，除此以外，并无新意。

Promise 的最大问题是代码冗余，原来的任务被 Promise 包装了一下，不管什么操作，一眼看去都是一堆 `then`，原来的语义变得很不清楚。

那么，有没有更好的写法呢？

协程

传统的编程语言，早有异步编程的解决方案（其实是多任务的解决方案）。其中有一种叫做"协程"（coroutine），意思是多个线程互相协作，完成异步任务。

协程有点像函数，又有点像线程。它的运行流程大致如下。

- 第一步，协程 A 开始执行。
- 第二步，协程 A 执行到一半，进入暂停，执行权转移到协程 B。
- 第三步，（一段时间后）协程 B 交还执行权。
- 第四步，协程 A 恢复执行。

上面流程的协程 A，就是异步任务，因为它分成两段（或多段）执行。

举例来说，读取文件的协程写法如下。

```
function* asyncJob() {  
  // ...其他代码  
  var f = yield readFile(fileA);  
  // ...其他代码  
}
```

上面代码的函数 `asyncJob` 是一个协程，它的奥妙就在其中的 `yield` 命令。它表示执行到此处，执行权将交给其他协程。也就是说，`yield` 命令是异步两个阶段的分界线。

协程遇到 `yield` 命令就暂停，等到执行权返回，再从暂停的地方继续往后执行。它的最大优点，就是代码的写法非常像同步操作，如果去除 `yield` 命令，简直一模一样。

协程的 Generator 函数实现

Generator 函数是协程在 ES6 的实现，最大特点就是可以交出函数的执行权（即暂停执行）。

整个 Generator 函数就是一个封装的异步任务，或者说是异步任务的容器。异步操作需要暂停的地方，都用 `yield` 语句注明。Generator 函数的执行方法如下。

```
function* gen(x) {  
  var y = yield x + 2;  
  return y;  
}  
  
var g = gen(1);  
g.next() // { value: 3, done: false }  
g.next() // { value: undefined, done: true }
```

上面代码中，调用 Generator 函数，会返回一个内部指针（即遍历器）`g`。这是 Generator 函数不同于普通函数的另一个地方，即执行它不会返回结果，返回的是指针对象。调用指针 `g` 的 `next` 方法，会移动内部指针（即执行异步任务的第一段），指向第一个遇到的 `yield` 语句，上例是执行到 `x + 2` 为止。

换言之，`next` 方法的作用是分阶段执行 Generator 函数。每次调用 `next` 方法，会返回一个对象，表示当前阶段的信息（`value` 属性和 `done` 属性）。`value` 属性是 `yield` 语句后面表达式的值，表示当前阶段的值；`done` 属性是一个布尔值，表示 Generator 函数是否执行完毕，即是否还有下一个阶段。

Generator 函数的数据交换和错误处理

Generator 函数可以暂停执行和恢复执行，这是它能封装异步任务的根本原因。除此之外，它还有两个特性，使它可以作为异步编程的完整解决方案：函数体内外的数据交换和错误处理机制。

`next` 返回值的 `value` 属性，是 Generator 函数向外输出数据；`next` 方法还可以接受参数，向 Generator 函数体内输入数据。

```
function* gen(x) {
  var y = yield x + 2;
  return y;
}

var g = gen(1);
g.next() // { value: 3, done: false }
g.next(2) // { value: 2, done: true }
```

上面代码中，第一个 `next` 方法的 `value` 属性，返回表达式 `x + 2` 的值 3。第二个 `next` 方法带有参数 2，这个参数可以传入 Generator 函数，作为上个阶段异步任务的返回结果，被函数体内的变量 `y` 接收。因此，这一步的 `value` 属性，返回的就是 2（变量 `y` 的值）。

Generator 函数内部还可以部署错误处理代码，捕获函数体外抛出的错误。

```
function* gen(x) {
  try {
    var y = yield x + 2;
  } catch (e) {
    console.log(e);
  }
  return y;
}

var g = gen(1);
g.next();
g.throw('出错了');
// 出错了
```

上面代码的最后一行，Generator 函数体外，使用指针对象的 `throw` 方法抛出的错误，可以被函数体内的 `try...catch` 代码块捕获。这意味着，出错的代码与处理错误的代码，实现了时间和空间上的分离，这对于异步编程无疑是很重要的。

异步任务的封装

下面看看如何使用 Generator 函数，执行一个真实的异步任务。

```
var fetch = require('node-fetch');

function* gen() {
  var url = 'https://api.github.com/users/github';
  var result = yield fetch(url);
  console.log(result.bio);
}
```

上面代码中，Generator 函数封装了一个异步操作，该操作先读取一个远程接口，然后从 JSON 格式的数据解析信息。就像前面说过的，这段代码非常像同步操作，除了加上了 `yield` 命令。

执行这段代码的方法如下。

```
var g = gen();
var result = g.next();

result.value.then(function(data) {
  return data.json();
}).then(function(data) {
  g.next(data);
});
```

上面代码中，首先执行 Generator 函数，获取遍历器对象，然后使用 `next` 方法（第二行），执行异步任务的第一阶段。由于 `Fetch` 模块返回的是一个 Promise 对象，因此要用 `then` 方法调用下一个 `next` 方法。

可以看到，虽然 Generator 函数将异步操作表示得很简洁，但是流程管理却不方便（即何时执行第一阶段、何时执行第二阶段）。

4. Thunk 函数

Thunk 函数是自动执行 Generator 函数的一种方法。

参数的求值策略

Thunk 函数早在上个世纪 60 年代就诞生了。

那时，编程语言刚刚起步，计算机学家还在研究，编译器怎么写比较好。一个争论的焦点是"求值策略"，即函数的参数到底应该何时求值。

```
var x = 1;

function f(m) {
  return m * 2;
}

f(x + 5)
```

上面代码先定义函数 `f`，然后向它传入表达式 `x + 5`。请问，这个表达式应该何时求值？

一种意见是"传值调用"（call by value），即在进入函数体之前，就计算 `x + 5` 的值（等于 6），再将这个值传入函数 `f`。C 语言就采用这种策略。

```
f(x + 5)
// 传值调用时，等同于
f(6)
```

另一种意见是"传名调用"（call by name），即直接将表达式 `x + 5` 传入函数体，只在用到它的时候求值。Haskell 语言采用这种策略。

```
f(x + 5)
// 传名调用时，等同于
(x + 5) * 2
```

传值调用和传名调用，哪一种比较好？

回答是各有利弊。传值调用比较简单，但是对参数求值的时候，实际上还没用到这个参数，有可能造成性能损失。

```
function f(a, b) {  
    return b;  
}  
  
f(3 * x * x - 2 * x - 1, x);
```

上面代码中，函数 `f` 的第一个参数是一个复杂的表达式，但是函数体内根本没用到。对这个参数求值，实际上是不必要的。因此，有一些计算机学家倾向于“传名调用”，即只在执行时求值。

Thunk 函数的含义

编译器的“传名调用”实现，往往是将参数放到一个临时函数之中，再将这个临时函数传入函数体。这个临时函数就叫做 Thunk 函数。

```
function f(m) {  
    return m * 2;  
}  
  
f(x + 5);  
  
// 等同于  
  
var thunk = function () {  
    return x + 5;  
};  
  
function f(thunk) {  
    return thunk() * 2;  
}
```

上面代码中，函数 `f` 的参数 `x + 5` 被一个函数替换了。凡是用到原参数的地方，对 `Thunk` 函数求值即可。

这就是 Thunk 函数的定义，它是“传名调用”的一种实现策略，用来替换某个表达式。

JavaScript 语言的 Thunk 函数

JavaScript 语言是传值调用，它的 Thunk 函数含义有所不同。在 JavaScript 语言中，Thunk 函数替换的不是表达式，而是多参数函数，将其替换成一个只接受回调函数作为参数的单参数函数。

```
// 正常版本的readFile（多参数版本）  
fs.readFile(fileName, callback);  
  
// Thunk版本的readFile（单参数版本）  
var Thunk = function (fileName) {  
    return function (callback) {  
        return fs.readFile(fileName, callback);  
    };  
};  
  
var readFileThunk = Thunk(fileName);  
readFileThunk(callback);
```

上面代码中，`fs` 模块的 `readFile` 方法是一个多参数函数，两个参数分别为文件名和回调函数。经过转换器处理，它变成了一个单参数函数，只接受回调函数作为参数。这个单参数版本，就叫做

任何函数，只要参数有回调函数，就能写成 Thunk 函数的形式。下面是一个简单的 Thunk 函数转换器。

```
// ES5版本
var Thunk = function(fn) {
  return function () {
    var args = Array.prototype.slice.call(arguments);
    return function (callback) {
      args.push(callback);
      return fn.apply(this, args);
    }
  };
};

// ES6版本
const Thunk = function(fn) {
  return function (...args) {
    return function (callback) {
      return fn.call(this, ...args, callback);
    }
  };
};
```

使用上面的转换器，生成 `fs.readFile` 的 Thunk 函数。

```
var readFileThunk = Thunk(fs.readFile);
readFileThunk(fileA)(callback);
```

下面是另一个完整的例子。

```
function f(a, cb) {
  cb(a);
}
const ft = Thunk(f);

ft(1)(console.log) // 1
```

Thunkify 模块

生产环境的转换器，建议使用 Thunkify 模块。

首先是安装。

```
$ npm install thunkify
```

使用方式如下。

```
var thunkify = require('thunkify');
var fs = require('fs');

var read = thunkify(fs.readFile);
read('package.json')(function(err, str) {
  // ...
});
```

Thunkify 的源码与上一节那个简单的转换器非常像。


```
function thunkify(fn) {
  return function() {
    var args = new Array(arguments.length);
    var ctx = this;

    for (var i = 0; i < args.length; ++i) {
      args[i] = arguments[i];
    }

    return function (done) {
      var called;

      args.push(function () {
        if (called) return;
        called = true;
        done.apply(null, arguments);
      });

      try {
        fn.apply(ctx, args);
      } catch (err) {
        done(err);
      }
    }
  };
}
```

它的源码主要多了一个检查机制，变量 `called` 确保回调函数只运行一次。这样的设计与下文的 `Generator` 函数相关。请看下面的例子。

```
function f(a, b, callback){
  var sum = a + b;
  callback(sum);
  callback(sum);
}

var ft = thunkify(f);
var print = console.log.bind(console);
ft(1, 2)(print);
// 3
```

上面代码中，由于 `thunkify` 只允许回调函数执行一次，所以只输出一行结果。

Generator 函数的流程管理

你可能会问，`Thunk` 函数有什么用？回答是以前确实没什么用，但是 `ES6` 有了 `Generator` 函数，`Thunk` 函数现在可以用于 `Generator` 函数的自动流程管理。

`Generator` 函数可以自动执行。

```
function* gen() {
  // ...
}

var g = gen();
var res = g.next();

while(!res.done){
  console.log(res.value);
```

```
    res = g.next();  
  }  
}
```

上面代码中，Generator 函数 `gen` 会自动执行完所有步骤。

但是，这不适合异步操作。如果必须保证前一步执行完，才能执行后一步，上面的自动执行就不可行。这时，Thunk 函数就能派上用处。以读取文件为例。下面的 Generator 函数封装了两个异步操作。

```
var fs = require('fs');  
var thunkify = require('thunkify');  
var readFileThunk = thunkify(fs.readFile);  
  
var gen = function* () {  
  var r1 = yield readFileThunk('/etc/fstab');  
  console.log(r1.toString());  
  var r2 = yield readFileThunk('/etc/shells');  
  console.log(r2.toString());  
};
```

上面代码中，`yield` 命令用于将程序的执行权移出 Generator 函数，那么就需要一种方法，将执行权再交还给 Generator 函数。

这种方法就是 Thunk 函数，因为它可以在回调函数里，将执行权交还给 Generator 函数。为了便于理解，我们先看如何手动执行上面这个 Generator 函数。

```
var g = gen();  
  
var r1 = g.next();  
r1.value(function (err, data) {  
  if (err) throw err;  
  var r2 = g.next(data);  
  r2.value(function (err, data) {  
    if (err) throw err;  
    g.next(data);  
  });  
});
```

上面代码中，变量 `g` 是 Generator 函数的内部指针，表示目前执行到哪一步。`next` 方法负责将指针移动到下一步，并返回该步的信息（`value` 属性和 `done` 属性）。

仔细查看上面的代码，可以发现 Generator 函数的执行过程，其实是将同一个回调函数，反复传入 `next` 方法的 `value` 属性。这使得我们可以用递归来自动完成这个过程。

Thunk 函数的自动流程管理

Thunk 函数真正的威力，在于可以自动执行 Generator 函数。下面就是一个基于 Thunk 函数的 Generator 执行器。

```
function run(fn) {  
  var gen = fn();  
  
  function next(err, data) {  
    var result = gen.next(data);  
    if (result.done) return;  
    result.value(next);  
  }  
  
  next();  
}
```

```
function* g() {  
  // ...  
}  
  
run(g);
```

上面代码的 `run` 函数，就是一个 Generator 函数的自动执行器。内部的 `next` 函数就是 Thunk 的回调函数。`next` 函数先将指针移到 Generator 函数的下一步（`gen.next` 方法），然后判断 Generator 函数是否结束（`result.done` 属性），如果没结束，就将 `next` 函数再传入 Thunk 函数（`result.value` 属性），否则就直接退出。

有了这个执行器，执行 Generator 函数方便多了。不管内部有多少个异步操作，直接把 Generator 函数传入 `run` 函数即可。当然，前提是每一个异步操作，都要是 Thunk 函数，也就是说，跟在 `yield` 命令后面的必须是 Thunk 函数。

```
var g = function* () {  
  var f1 = yield readFileThunk('fileA');  
  var f2 = yield readFileThunk('fileB');  
  // ...  
  var fn = yield readFileThunk('fileN');  
};  
  
run(g);
```

上面代码中，函数 `g` 封装了 `n` 个异步的读取文件操作，只要执行 `run` 函数，这些操作就会自动完成。这样一来，异步操作不仅可以写得像同步操作，而且一行代码就可以执行。

Thunk 函数并不是 Generator 函数自动执行的唯一方案。因为自动执行的关键是，必须有一种机制，自动控制 Generator 函数的流程，接收和交还程序的执行权。回调函数可以做到这一点，Promise 对象也可以做到这一点。

5. co 模块

基本用法

`co` 模块是著名程序员 TJ Holowaychuk 于 2013 年 6 月发布的一个小工具，用于 Generator 函数的自动执行。

下面是一个 Generator 函数，用于依次读取两个文件。

```
var gen = function* () {  
  var f1 = yield readFile('/etc/fstab');  
  var f2 = yield readFile('/etc/shells');  
  console.log(f1.toString());  
  console.log(f2.toString());  
};
```

`co` 模块可以让你不用编写 Generator 函数的执行器。

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

上面代码中，Generator 函数只要传入 `co` 函数，就会自动执行。

`co` 函数返回一个 `Promise` 对象，因此可以用 `then` 方法[处理返回的Promise对象](#)。[上一章](#) [下一章](#)

```
co(gen).then(function () {
  console.log('Generator 函数执行完成');
});
```

上面代码中，等到 Generator 函数执行结束，就会输出一行提示。

co 模块的原理

为什么 co 可以自动执行 Generator 函数？

前面说过，Generator 就是一个异步操作的容器。它的自动执行需要一种机制，当异步操作有了结果，能够自动交回执行权。

两种方法可以做到这一点。

- （1）回调函数。将异步操作包装成 Thunk 函数，在回调函数里面交回执行权。
- （2）Promise 对象。将异步操作包装成 Promise 对象，用 then 方法交回执行权。

co 模块其实就是将两种自动执行器（Thunk 函数和 Promise 对象），包装成一个模块。使用 co 的前提条件是，Generator 函数的 yield 命令后面，只能是 Thunk 函数或 Promise 对象。如果数组或对象的成员，全部都是 Promise 对象，也可以使用 co，详见后文的例子。

上一节已经介绍了基于 Thunk 函数的自动执行器。下面来看，基于 Promise 对象的自动执行器。这是理解 co 模块必须的。

基于 Promise 对象的自动执行

还是沿用上面的例子。首先，把 fs 模块的 readFile 方法包装成一个 Promise 对象。

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

var readFile = function (fileName) {
  return new Promise(function (resolve, reject) {
    fs.readFile(fileName, function(error, data) {
      if (error) return reject(error);
      resolve(data);
    });
  });
};

var gen = function* () {
  var f1 = yield readFile('/etc/fstab');
  var f2 = yield readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

然后，手动执行上面的 Generator 函数。

```
var g = gen();

g.next().value.then(function(data) {
  g.next(data).value.then(function(data) {
    g.next(data);
```

```
});  
});
```

手动执行其实就是用 `then` 方法，层层添加回调函数。理解了这一点，就可以写出一个自动执行器。

```
function run(gen) {  
  var g = gen();  
  
  function next(data) {  
    var result = g.next(data);  
    if (result.done) return result.value;  
    result.value.then(function(data) {  
      next(data);  
    });  
  }  
  
  next();  
}  
  
run(gen);
```

上面代码中，只要 Generator 函数还没执行到最后一步，`next` 函数就调用自身，以此实现自动执行。

co 模块的源码

`co` 就是上面那个自动执行器的扩展，它的源码只有几十行，非常简单。

首先，`co` 函数接受 Generator 函数作为参数，返回一个 Promise 对象。

```
function co(gen) {  
  var ctx = this;  
  
  return new Promise(function(resolve, reject) {  
  });  
}
```

在返回的 Promise 对象里面，`co` 先检查参数 `gen` 是否为 Generator 函数。如果是，就执行该函数，得到一个内部指针对象；如果不是就返回，并将 Promise 对象的状态改为 `resolved`。

```
function co(gen) {  
  var ctx = this;  
  
  return new Promise(function(resolve, reject) {  
    if (typeof gen === 'function') gen = gen.call(ctx);  
    if (!gen || typeof gen.next !== 'function') return resolve(gen);  
  });  
}
```

接着，`co` 将 Generator 函数的内部指针对象的 `next` 方法，包装成 `onFulfilled` 函数。这主要是为了能够捕捉抛出的错误。

```
function co(gen) {  
  var ctx = this;  
  
  return new Promise(function(resolve, reject) {  
    if (typeof gen === 'function') gen = gen.call(ctx);  
    if (!gen || typeof gen.next !== 'function' 上一章 re 下一章 ;
```

```

    onFulfilled();
    function onFulfilled(res) {
        var ret;
        try {
            ret = gen.next(res);
        } catch (e) {
            return reject(e);
        }
        next(ret);
    }
});
}

```

最后，就是关键的 `next` 函数，它会反复调用自身。

```

function next(ret) {
    if (ret.done) return resolve(ret.value);
    var value = toPromise.call(ctx, ret.value);
    if (value && isPromise(value)) return value.then(onFulfilled, onRejected);
    return onRejected(
        new TypeError(
            'You may only yield a function, promise, generator, array, or object, '
            + 'but the following object was passed: "'
            + String(ret.value)
            + '"')
    );
};
}

```

上面代码中，`next` 函数的内部代码，一共只有四行命令。

第一行，检查当前是否为 Generator 函数的最后一步，如果是就返回。

第二行，确保每一步的返回值，是 Promise 对象。

第三行，使用 `then` 方法，为返回值加上回调函数，然后通过 `onFulfilled` 函数再次调用 `next` 函数。

第四行，在参数不符合要求的情况下（参数非 Thunk 函数和 Promise 对象），将 Promise 对象的状态改为 `rejected`，从而终止执行。

处理并发的异步操作

`co` 支持并发的异步操作，即允许某些操作同时进行，等到它们全部完成，才进行下一步。

这时，要把并发的操作都放在数组或对象里面，跟在 `yield` 语句后面。

```

// 数组的写法
co(function* () {
    var res = yield [
        Promise.resolve(1),
        Promise.resolve(2)
    ];
    console.log(res);
}).catch(onerror);

```

```

// 对象的写法
co(function* () {

```

```

var res = yield {
  1: Promise.resolve(1),
  2: Promise.resolve(2),
};
console.log(res);
}).catch(onerror);

```

下面是另一个例子。

```

co(function* () {
  var values = [n1, n2, n3];
  yield values.map(somethingAsync);
});

function* somethingAsync(x) {
  // do something async
  return y
}

```

上面的代码允许并发三个 `somethingAsync` 异步操作，等到它们全部完成，才会进行下一步。

实例：处理 Stream

Node 提供 Stream 模式读写数据，特点是一次只处理数据的一部分，数据分成一块块依次处理，就好像“数据流”一样。这对于处理大规模数据非常有利。Stream 模式使用 EventEmitter API，会释放三个事件。

- `data` 事件：下一块数据块已经准备好了。
- `end` 事件：整个“数据流”处理完了。
- `error` 事件：发生错误。

使用 `Promise.race()` 函数，可以判断这三个事件之中哪一个最先发生，只有当 `data` 事件最先发生时，才进入下一个数据块的处理。从而，我们可以通过一个 `while` 循环，完成所有数据的读取。

```

const co = require('co');
const fs = require('fs');

const stream = fs.createReadStream('./les_miserables.txt');
let valjeanCount = 0;

co(function*() {
  while(true) {
    const res = yield Promise.race([
      new Promise(resolve => stream.once('data', resolve)),
      new Promise(resolve => stream.once('end', resolve)),
      new Promise((resolve, reject) => stream.once('error', reject))
    ]);
    if (!res) {
      break;
    }
    stream.removeAllListeners('data');
    stream.removeAllListeners('end');
    stream.removeAllListeners('error');
    valjeanCount += (res.toString().match(/valjean/ig) || []).length;
  }
  console.log('count:', valjeanCount); // count: 1120
});

```

上面代码采用 Stream 模式读取《悲惨世界》的文本文件，对于每个数据块都使用 `stream.once` 方法，在 `data`、`end`、`error` 三个事件上添加一次性回调函数。变量 `res` 只有在 `data` 事件发生时才有值，然后累加每个数据块之中 `valjean` 这个词出现的次数。

留言