异步操作和Async函数

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。

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

ES6将JavaScript异步编程带入了一个全新的阶段，ES7的`Async`函数更是提出了异步编程的终极解决方案。

---

1. 基本概念

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

异步

所谓"异步"，简单说就是一个任务分成两段，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。它的英语名字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', function (err, data) {
  if (err) throw err;
  console.log(data);
});
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

上面代码中，readFile函数的第二个参数，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.js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，在这两段之间抛出的错误，程序无法捕捉，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, function (err, data) {
  fs.readFile(fileB, function (err, data) {
    // ...
  });
});
```

不难想象，如果依次读取多个文件，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。这种情况就称为"回调函数噩梦"（callback hell）。

Promise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');
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
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？

---

2. Generator函数

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

Generator函数的概念

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。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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方法返回值的value属性，是Generator函数向外输出数据；next方法还可以接受参数，这是向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 {
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
    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

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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）。

---

3. Thunk函数

参数的求值策略

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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;
};
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
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，将其替换成单参数的版本，且只接受回调函数作为参数。

```
// 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 readFileThunk = Thunk(fileName);
readFileThunk(callback);

var Thunk = function (fileName){
  return function (callback){
    return fs.readFile(fileName, callback);
  };
};
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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Thunk函数。

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
var 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);
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

下面是另一个完整的例子。

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

let log = console.log.bind(console);
ft(1)(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){
  // ...
});
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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){
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
  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();
}
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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 readFile = thunkify(fs.readFile);

var gen = function* (){
  var r1 = yield readFile('/etc/fstab');
  console.log(r1.toString());
  var r2 = yield readFile('/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){
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
    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() {
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
  // ...
}

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 readFile('fileA');
  var f2 = yield readFile('fileB');
  // ...
  var fn = yield readFile('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

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

以做到这一点，Promise 对象也可以做到这一点。

---

4. 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。

比如，有一个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。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

co函数返回一个Promise对象，因此可以用then方法添加回调函数。

```
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。

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。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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);
  });
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
});
```

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。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
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') return resolve(gen);
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
    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, onRejec
  return onRejected(new TypeError('You may only yield a function, promi
    + '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。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
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。

---

5. async函数

含义

ES7提供了`async`函数，使得异步操作变得更加方便。`async`函数是什么？一句话，`async`函数就是Generator函数的语法糖。

前文有一个Generator函数，依次读取两个文件。

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

var readFile = function (fileName) {
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

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

写成`async`函数，就是下面这样。

```
var asyncReadFile = async function (){
  var f1 = await readFile('/etc/fstab');
  var f2 = await readFile('/etc/shells');
  console.log(f1.toString());
  console.log(f2.toString());
};
```

一比较就会发现，`async`函数就是将Generator函数的星号（`*`）替换成`async`，将`yield`替换成`await`，仅此而已。

`async`函数对 Generator 函数的改进，体现在以下四点。

（1）内置执行器。Generator函数的执行必须靠执行器，所以才有了`co`模块，而`async`函数自带执行器。也就是说，`async`函数的执行，与普通函数一模一样，只要

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

一行。

```
var result = asyncReadFile();
```

上面的代码调用了`asyncReadFile`函数，然后它就会自动执行，输出最后结果。这完全不像Generator函数，需要调用`next`方法，或者用`co`模块，才能得到眞正执行，得到最后结果。

（2）更好的语义。`async`和`await`，比起星号和`yield`，语义更清楚了。`async`表示函数里有异步操作，`await`表示紧跟在后面的表达式需要等待结果。

（3）更广的适用性。`co`模块约定，`yield`命令后面只能是Thunk函数或Promise对象，而`async`函数的`await`命令后面，可以是Promise对象和原始类型的值（数值、字符串和布尔值，但这时等同于同步操作）。

（4）返回值是Promise。`async`函数的返回值是Promise对象，这比Generator函数的返回值是Iterator对象方便多了。你可以用`then`方法指定下一步的操作。

进一步说，`async`函数完全可以看作多个异步操作，包装成的一个Promise对象，而`await`命令就是内部`then`命令的语法糖。

语法

`async`函数的语法规则总体上比较简单，难点是错误处理机制。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

（1）`async`函数返回一个Promise对象。

`async`函数内部`return`语句返回的值，会成为`then`方法回调函数的参数。

```
async function f() {
  return 'hello world';
}

f().then(v => console.log(v))
// "hello world"
```

上面代码中，函数`f`内部`return`命令返回的值，会被`then`方法回调函数接收到。

`async`函数内部抛出错误，会导致返回的Promise对象变为`reject`状态。抛出的错误对象会被`catch`方法回调函数接收到。

```
async function f() {
  throw new Error('出错了');
}

f().then(
  v => console.log(v),
  e => console.log(e)
)
// Error: 出错了
```

（2）`async`函数返回的Promise对象，必须等到内部所有`await`命令的Promise对象执行完，才会发生状态改变。也就是说，只有`async`函数内部的异步操作执行完，才会执行`then`方法指定的回调函数。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

下面是一个例子。

```
async function getTitle(url) {
  let response = await fetch(url);
  let html = await response.text();
  return html.match(/<title>([\s\S]+)<\/title>/i)[1];
}
getTitle('https://tc39.github.io/ecma262/').then(console.log)
// "ECMAScript 2017 Language Specification"
```

（3）正常情况下，`await`命令后面是一个Promise对象。如果不是，会被转成一个立即`resolve`的Promise对象。

```
async function f() {
  return await 123;
}

f().then(v => console.log(v))
// 123
```

上面代码中，`await`命令的参数是数值`123`，它被转成Promise对象，并立即`resolve`。

`await`命令后面的Promise对象如果变为`reject`状态，则`reject`的参数会被`catch`方法的回调函数接收到。

```
async function f() {
  await Promise.reject('出错了');
}
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
f()
.then(v => console.log(v))
.catch(e => console.log(e))
// 出错了
```

注意，上面代码中，`await`语句前面没有`return`，但是`reject`方法的参数依然传入了`catch`方法的回调函数。这里如果在`await`前面加上`return`，效果是一样的。

只要一个`await`语句后面的Promise变为`reject`，那么整个`async`函数都会中断执行。

```
async function f() {
  await Promise.reject('出错了');
  await Promise.resolve('hello world'); // 不会执行
}
```

上面代码中，第二个`await`语句是不会执行的，因为第一个`await`语句状态变成了`reject`。

为了避免这个问题，可以将第一个`await`放在`try...catch`结构里面，这样第二个`await`就会执行。

```
async function f() {
  try {
    await Promise.reject('出错了');
  } catch(e) {
  }
  return await Promise.resolve('hello world');
}
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
f()
.then(v => console.log(v))
// hello world
```

另一种方法是`await`后面的Promise对象再跟一个`catch`方面，处理前面可能出现的错误。

```
async function f() {
  await Promise.reject('出错了')
    .catch(e => console.log(e));
  return await Promise.resolve('hello world');
}

f()
.then(v => console.log(v))
// 出错了
// hello world
```

如果有多个`await`命令，可以统一放在`try...catch`结构中。

```
async function main() {
  try {
    var val1 = await firstStep();
    var val2 = await secondStep(val1);
    var val3 = await thirdStep(val1, val2);

    console.log('Final: ', val3);
  }
  catch (err) {
    console.error(err);
  }
}
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

（4）如果`await`后面的异步操作出错，那么等同于`async`函数返回的Promise对象被`reject`。

```
async function f() {
  await new Promise(function (resolve, reject) {
    throw new Error('出错了');
  });
}

f()
.then(v => console.log(v))
.catch(e => console.log(e))
// Error：出错了
```

上面代码中，`async`函数`f`执行后，`await`后面的Promise对象会抛出一个错误对象，导致`catch`方法的回调函数被调用，它的参数就是抛出的错误对象。具体的执行机制，可以参考后文的“async函数的实现”。

防止出错的方法，也是将其放在`try...catch`代码块之中。

```
async function f() {
  try {
    await new Promise(function (resolve, reject) {
      throw new Error('出错了');
    });
  } catch(e) {
  }
  return await('hello world');
}
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

async函数的实现

async 函数的实现，就是将 Generator 函数和自动执行器，包装在一个函数里。

```
async function fn(args){
  // ...
}

// 等同于

function fn(args){
  return spawn(function*() {
    // ...
  });
}
```

所有的`async`函数都可以写成上面的第二种形式，其中的 spawn 函数就是自动执行器。

下面给出`spawn`函数的实现，基本就是前文自动执行器的翻版。

```
function spawn(genF) {
  return new Promise(function(resolve, reject) {
    var gen = genF();
    function step(nextF) {
      try {
        var next = nextF();
      } catch(e) {
        return reject(e);
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
      }
      if(next.done) {
        return resolve(next.value);
      }
      Promise.resolve(next.value).then(function(v) {
        step(function() { return gen.next(v); });
      }, function(e) {
        step(function() { return gen.throw(e); });
      });
    }
    step(function() { return gen.next(undefined); });
  });
}
```

`async`函数是非常新的语法功能，新到都不属于 ES6，而是属于 ES7。目前，它仍处于提案阶段，但是转码器`Babel`和`regenerator`都已经支持，转码后就能使用。

---

async 函数的用法

`async`函数返回一个Promise对象，可以使用`then`方法添加回调函数。当函数执行的时候，一旦遇到`await`就会先返回，等到触发的异步操作完成，再接着执行函数体内后面的语句。

下面是一个例子。

```
async function getStockPriceByName(name) {
  var symbol = await getStockSymbol(name);
  var stockPrice = await getStockPrice(symbol);
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
  return stockPrice;
}

getStockPriceByName('goog').then(function (result) {
  console.log(result);
});
```

上面代码是一个获取股票报价的函数，函数前面的`async`关键字，表明该函数内部有异步操作。调用该函数时，会立即返回一个`Promise`对象。

下面的例子，指定多少毫秒后输出一个值。

```
function timeout(ms) {
  return new Promise((resolve) => {
    setTimeout(resolve, ms);
  });
}

async function asyncPrint(value, ms) {
  await timeout(ms);
  console.log(value)
}

asyncPrint('hello world', 50);
```

上面代码指定50毫秒以后，输出"hello world"。

Async函数有多种使用形式。

```
// 函数声明
async function foo() {}
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
// 函数表达式
const foo = async function () {};

// 对象的方法
let obj = { async foo() {} };

// 箭头函数
const foo = async () => {};
```

注意点

第一点，`await`命令后面的Promise对象，运行结果可能是rejected，所以最好把`await`命令放在`try...catch`代码块中。

```
async function myFunction() {
  try {
    await somethingThatReturnsAPromise();
  } catch (err) {
    console.log(err);
  }
}

// 另一种写法

async function myFunction() {
  await somethingThatReturnsAPromise()
  .catch(function (err) {
    console.log(err);
  };
}
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

第二点，多个`await`命令后面的异步操作，如果不存在继发关系，最好让它们同时触发。

```
let foo = await getFoo();
let bar = await getBar();
```

上面代码中，`getFoo`和`getBar`是两个独立的异步操作（即互不依赖），被写成继发关系。这样比较耗时，因为只有`getFoo`完成以后，才会执行`getBar`，完全可以让它们同时触发。

```
// 写法一
let [foo, bar] = await Promise.all([getFoo(), getBar()]);

// 写法二
let fooPromise = getFoo();
let barPromise = getBar();
let foo = await fooPromise;
let bar = await barPromise;
```

上面两种写法，`getFoo`和`getBar`都是同时触发，这样就会缩短程序的执行时间。

第三点，`await`命令只能用在`async`函数之中，如果用在普通函数，就会报错。

```
async function dbFuc(db) {
  let docs = [{}, {}, {}];

  // 报错
  docs.forEach(function (doc) {
    await db.post(doc);
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
  });
}
```

上面代码会报错，因为await用在普通函数之中了。但是，如果将`forEach`方法的参数改成`async`函数，也有问题。

```
async function dbFuc(db) {
  let docs = [{}, {}, {}];

  // 可能得到错误结果
  docs.forEach(async function (doc) {
    await db.post(doc);
  });
}
```

上面代码可能不会正常工作，原因是这时三个`db.post`操作将是并发执行，也就是同时执行，而不是继发执行。正确的写法是采用`for`循环。

```
async function dbFuc(db) {
  let docs = [{}, {}, {}];

  for (let doc of docs) {
    await db.post(doc);
  }
}
```

如果确实希望多个请求并发执行，可以使用`Promise.all`方法。

```
async function dbFuc(db) {
  let docs = [{}, {}, {}];
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
  let promises = docs.map((doc) => db.post(doc));

  let results = await Promise.all(promises);
  console.log(results);
}

// 或者使用下面的写法

async function dbFuc(db) {
  let docs = [{}, {}, {}];
  let promises = docs.map((doc) => db.post(doc));

  let results = [];
  for (let promise of promises) {
    results.push(await promise);
  }
  console.log(results);
}
```

ES6将`await`增加为保留字。使用这个词作为标识符，在ES5是合法的，在ES6将抛出SyntaxError。

与Promise、Generator的比较

我们通过一个例子，来看Async函数与Promise、Generator函数的区别。

假定某个DOM元素上面，部署了一系列的动画，前一个动画结束，才能开始后一个。如果当中有一个动画出错，就不再往下执行，返回上一个成功执行的动画的返回值。

首先是Promise的写法。

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
function chainAnimationsPromise(elem, animations) {

  // 变量ret用来保存上一个动画的返回值
  var ret = null;

  // 新建一个空的Promise
  var p = Promise.resolve();

  // 使用then方法，添加所有动画
  for(var anim of animations) {
    p = p.then(function(val) {
      ret = val;
      return anim(elem);
    });
  }

  // 返回一个部署了错误捕捉机制的Promise
  return p.catch(function(e) {
    /* 忽略错误，继续执行 */
  }).then(function() {
    return ret;
  });

}
```

虽然Promise的写法比回调函数的写法大大改进，但是一眼看上去，代码完全都是Promise的API（then、catch等等），操作本身的语义反而不容易看出来。

接着是Generator函数的写法。

```
function chainAnimationsGenerator(elem, animations) {
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

```
  return spawn(function*() {
    var ret = null;
    try {
      for(var anim of animations) {
        ret = yield anim(elem);
      }
    } catch(e) {
      /* 忽略错误，继续执行 */
    }
    return ret;
  });

}
```

上面代码使用Generator函数遍历了每个动画，语义比Promise写法更清晰，用户定义的操作全部都出现在spawn函数的内部。这个写法的问题在于，必须有一个任务运行器，自动执行Generator函数，上面代码的spawn函数就是自动执行器，它返回一个Promise对象，而且必须保证yield语句后面的表达式，必须返回一个Promise。

最后是Async函数的写法。

```
async function chainAnimationsAsync(elem, animations) {
  var ret = null;
  try {
    for(var anim of animations) {
      ret = await anim(elem);
    }
  } catch(e) {
    /* 忽略错误，继续执行 */
  }
  return ret;
}
```

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com

可以看到Async函数的实现最简洁，最符合语义，几乎没有语义不相关的代码。它将Generator写法中的自动执行器，改在语言层面提供，不暴露给用户，因此代码量最少。如果使用Generator写法，自动执行器需要用户自己提供。

留言

上一章 下一章

open in browser PRO version Are you a developer? Try out the HTML to PDF API pdfcrowd.com