async 函数的含义和用法

作者： 阮一峰

分享

日期： 2015年5月11日

本文是《深入掌握 ECMAScript 6 异步编程》系列文章的最后一篇。

一、终极解决

异步操作是 JavaScript 编程的麻烦事，麻烦到一直有人提出各种各样的方案，试图解决这个问题。

从最早的回调函数，到 Promise 对象，再到 Generator 函数，每次都有所改进，但又让人觉得不彻底。它们都有额外的复杂性，都需要理解抽象的底层运行机制。

异步I/O不就是读取一个文件吗，干嘛要搞得这么复杂？**异步编程的最高境界，就是根本不用关心它是不是异步。**

async 函数就是隧道尽头的亮光，很多人认为它是异步操作的终极解决方案。

二、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){
  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。

三、async 函数的优点

async 函数对 Generator 函数的改进，体现在以下三点。

（1）内置执行器。 Generator 函数的执行必须靠执行器，所以才有了 co 函数库，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，只要一行。

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

（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 函数库约定，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，可以跟 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）。

四、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);
      }
      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。

五、async 函数的用法

同 Generator 函数一样，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);
  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"。

六、注意点

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);
  });
}
```

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);
  });
}
```

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 = [{}, {}, {}];
  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);
}
```

（完）

文档信息

相关文章

- **2018.02.23:** Node 定时器详解

  > JavaScript 是单线程运行，异步操作特别重要。

- **2017.09.19:** 《ES6 标准入门（第3版）》上市了！

  > 2017年6月，TC39 委员会正式发布了《ES2017 标准》。

- **2017.09.07:** asm.js 和 Emscripten 入门教程

  > Web 技术突飞猛进，但是有一个领域一直无法突破 ---- 游戏。

- **2017.08.09:** Koa 框架教程

  > Node 主要用在开发 Web 应用。这决定了使用 Node，往往离不开 Web 应用框架。