ECMAScript 6 入门

作者：阮一峰

目录

其他

- 源码
- 修订历史
- 反馈意见

async 函数

上一章 下一章

---

1. 含义

ES2017 标准引入了 async 函数，使得异步操作变得更加方便。

async 函数是什么？一句话，它就是 Generator 函数的语法糖。

前文有一个 Generator 函数，依次读取两个文件。

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

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

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

上面代码的函数 `gen` 可以写成 `async` 函数，就是下面这样。

```
const asyncReadFile = async function () {
  const f1 = await readFile('/etc/fstab');
  const 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，只要一行。

```
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，才能真正执行，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，但这时会自动转成立即 resolved 的 Promise 对象）。

（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。

---

2. 基本用法

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) {
  const symbol = await getStockSymbol(name);
  const 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。

下面是另一个例子，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。

由于 async 函数返回的是 Promise 对象，可以作为 await 命令的参数。所以，上面的例子也可以写成下面的形式。

```
async function timeout(ms) {
  await new Promise((resolve) => {
    setTimeout(resolve, ms);
```

```
  });
}

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

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

async 函数有多种使用形式。

```
// 函数声明
async function foo() {}

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

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

// Class 的方法
class Storage {
  constructor() {
    this.cachePromise = caches.open('avatars');
  }

  async getAvatar(name) {
    const cache = await this.cachePromise;
    return cache.match(`/avatars/${name}.jpg`);
  }
}

const storage = new Storage();
storage.getAvatar('jake').then(…);

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

---

3. 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。

返回 Promise 对象

`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
```

Promise 对象的状态变化

`async` 函数返回的 Promise 对象，必须等到内部所有 `await` 命令后面的 Promise 对象执行完，才会发生状态改变，除非遇到 `return` 语句或者抛出错误。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。

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"
```

上面代码中，函数 `getTitle` 内部有三个操作：抓取网页、取出文本、匹配页面标题。只有这三个操作全部完成，才会执行 `then` 方法里面的 `console.log` 。

---

await 命令

正常情况下， `await` 命令后面是一个 Promise 对象，返回该对象的结果。如果不是 Promise 对象，就直接返回对应的值。

```
async function f() {
  // 等同于
  // return 123;
  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` ，这时等同于 `return 123` 。

另一种情况是， `await` 命令后面是一个 `thenable` 对象（即定义 `then` 方法的对象），那么 `await` 会将其等同于 Promise 对象。

```
class Sleep {
  constructor(timeout) {
    this.timeout = timeout;
  }
  then(resolve, reject) {
    const startTime = Date.now();
    setTimeout(
      () => resolve(Date.now() - startTime),
      this.timeout
    );
  }
}

(async () => {
  const sleepTime = await new Sleep(1000);
  console.log(sleepTime);
})();
// 1000
```

上面代码中， `await` 命令后面是一个 `Sleep` 对象的实例。这个实例不是 Promise 对象，但是因为定义了 `then` 方法， `await` 会将其视为 `Promise` 处理。

这个例子还演示了如何实现休眠效果。JavaScript 一直没有休眠的语法，但是借助 `await` 命令就可以让程序停顿指定的时间。下面给出了一个简化的 `sleep` 实现。

```
function sleep(interval) {
  return new Promise(resolve => {
    setTimeout(resolve, interval);
  })
}

// 用法
async function one2FiveInAsync() {
  for(let i = 1; i <= 5; i++) {
    console.log(i);
    await sleep(1000);
  }
}

one2FiveInAsync();
```

`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');
}

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。

任何一个 `await` 语句后面的 Promise 对象变为 `reject` 状态，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` 。

有时，我们希望即使前一个异步操作失败，也不要中断后面的异步操作。这时可以将第一个 `await` 放在 `try...catch` 结构里面，这样不管这个异步操作是否成功，第二个 `await` 都会执行。

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

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` 方法，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
```

错误处理

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 函数的实现原理”。

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, reje
```

```
      throw new Error('出错了');
    });
  } catch(e) {
  }
  return await('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 {
    const val1 = await firstStep();
    const val2 = await secondStep(val1);
    const val3 = await thirdStep(val1, val2);

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

下面的例子使用 `try...catch` 结构，实现多次重复尝试。

```
const superagent = require('superagent');
const NUM_RETRIES = 3;

async function test() {
  let i;
  for (i = 0; i < NUM_RETRIES; ++i) {
    try {
      await superagent.get('http://google.com/this-throws-an-error');
      break;
    } catch(err) {}
  }
  console.log(i); // 3
}

test();
```

上面代码中，如果 `await` 操作成功，就会使用 `break` 语句退出循环；如果失败，会被 `catch` 语句捕捉，然后进入下一轮循环。

使用注意点

第一点，前面已经说过， `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);
  });
}
```

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);
  });
}
```

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。

```
function dbFuc(db) { //这里不需要 async
  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。当三个请求都会 `resolved` 时，下面两种写法效果相同。

```
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);
}
```

第四点，async 函数可以保留运行堆栈。

```
const a = () => {
  b().then(() => c());
};
```

上面代码中，函数 `a` 内部运行了一个异步任务 `b()` 。当 `b()` 运行的时候，函数 `a()` 不会中断，而是继续执行。等到 `b()` 运行结束，可能 `a()` 早就运行结束了， `b()` 所在的上下文环境已经消失了。如果 `b()` 或 `c()` 报错，错误堆栈将不包括 `a()` 。

现在将这个例子改成 `async` 函数。

```
const a = async () => {
  await b();
  c();
};
```

上面代码中， `b()` 运行的时候， `a()` 是暂停执行，上下文环境都保存着。一旦 `b()` 或 `c()` 报错，错误堆栈将包括 `a()` 。

---

4. async 函数的实现原理

async 函数的实现原理，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) {
    const gen = genF();
    function step(nextF) {
      let next;
      try {
        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); });
  });
}
```

---

5. 与其他异步处理方法的比较

我们通过一个例子，来看 async 函数与 Promise、Generator 函数的比较。

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。

```
function chainAnimationsPromise(elem, animations) {

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

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

  // 使用then方法，添加所有动画
  for(let 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) {

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

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。

---

6. 实例：按顺序完成异步操作

实际开发中，经常遇到一组异步操作，需要按照顺序完成。比如，依次远程读取一组 URL，然后按照读取的顺序输出结果。

Promise 的写法如下。

```
function logInOrder(urls) {
  // 远程读取所有URL
  const textPromises = urls.map(url => {
    return fetch(url).then(response => response.text());
  });
```

```
  // 按次序输出
  textPromises.reduce((chain, textPromise) => {
    return chain.then(() => textPromise)
      .then(text => console.log(text));
  }, Promise.resolve());
}
```

上面代码使用 `fetch` 方法，同时远程读取一组 URL。每个 `fetch` 操作都返回一个 Promise 对象，放入 `textPromises` 数组。然后，`reduce` 方法依次处理每个 Promise 对象，然后使用 `then`，将所有 Promise 对象连起来，因此就可以依次输出结果。

这种写法不太直观，可读性比较差。下面是 async 函数实现。

```
async function logInOrder(urls) {
  for (const url of urls) {
    const response = await fetch(url);
    console.log(await response.text());
  }
}
```

上面代码确实大大简化，问题是所有远程操作都是继发。只有前一个 URL 返回结果，才会去读取下一个 URL，这样做效率很差，非常浪费时间。我们需要的是并发发出远程请求。

```
async function logInOrder(urls) {
  // 并发读取远程URL
  const textPromises = urls.map(async url => {
    const response = await fetch(url);
    return response.text();
  });

  // 按次序输出
  for (const textPromise of textPromises) {
    console.log(await textPromise);
  }
}
```

上面代码中，虽然 `map` 方法的参数是 `async` 函数，但它是并发执行的，因为只有 `async` 函数内部是继发执行，外部不受影响。后面的 `for..of` 循环内部使用了 `await`，因此实现了按顺序输出。

---

7. 顶层 await

根据语法规格，`await` 命令只能出现在 async 函数内部，否则都会报错。

```
// 报错
const data = await fetch('https://api.example.com');
```

上面代码中，`await` 命令独立使用，没有放在 async 函数里面，就会报错。

目前，有一个语法提案，允许在模块的顶层独立使用 `await` 命令。这个提案的目的，是借用 `await` 解决模块异步加载的问题。

```
// awaiting.js
let output;
async function main() {
  const dynamic = await import(someMission);
  const data = await fetch(url);
  output = someProcess(dynamic.default, data)
}
```

```
main();
export { output };
```

上面代码中，模块 `awaiting.js` 的输出值 `output`，取决于异步操作。我们把异步操作包装在一个 async 函数里面，然后调用这个函数，只有等里面的异步操作都执行，变量 `output` 才会有值，否则就返回 `undefined`。

上面的代码也可以写成立即执行函数的形式。

```
// awaiting.js
let output;
(async function main() {
  const dynamic = await import(someMission);
  const data = await fetch(url);
  output = someProcess(dynamic.default, data);
})();
export { output };
```

下面是加载这个模块的写法。

```
// usage.js
import { output } from "./awaiting.js";

function outputPlusValue(value) { return output + value }

console.log(outputPlusValue(100));
setTimeout(() => console.log(outputPlusValue(100), 1000);
```

上面代码中，`outputPlusValue()` 的执行结果，完全取决于执行的时间。如果 `awaiting.js` 里面的异步操作没执行完，加载进来的 `output` 的值就是 `undefined`。

目前的解决方法，就是让原始模块输出一个 Promise 对象，从这个 Promise 对象判断异步操作有没有结束。

```
// awaiting.js
let output;
export default (async function main() {
  const dynamic = await import(someMission);
  const data = await fetch(url);
  output = someProcess(dynamic.default, data);
})();
export { output };
```

上面代码中，`awaiting.js` 除了输出 `output`，还默认输出一个 Promise 对象（async 函数立即执行后，返回一个 Promise 对象），从这个对象判断异步操作是否结束。

下面是加载这个模块的新的写法。

```
// usage.js
import promise, { output } from "./awaiting.js";

function outputPlusValue(value) { return output + value }

promise.then(() => {
  console.log(outputPlusValue(100));
  setTimeout(() => console.log(outputPlusValue(100), 1000);
});
```

上面代码中，将 `awaiting.js` 对象的输出，放在 `promise.then()` 里面，这样就能保证异步操作完成以后，才去读取 `output`。

这种写法比较麻烦，等于要求模块的使用者遵守一个额外的使用协议，按照特殊的方法使用这个模块。一旦你忘了要用 Promise 加载，只使用正常的加载方法，依赖这个模块的代码就可能出错。而且，如果上面的 `usage.js` 又有对外的输出，等于这个依赖链的所有模块都要使用 Promise 加载。

顶层的 `await` 命令，就是为了解决这个问题。它保证只有异步操作完成，模块才会输出值。

```
// awaiting.js
const dynamic = import(someMission);
const data = fetch(url);
export const output = someProcess((await dynamic).default, await data);
```

上面代码中，两个异步操作在输出的时候，都加上了 `await` 命令。只有等到异步操作完成，这个模块才会输出值。

加载这个模块的写法如下。

```
// usage.js
import { output } from "./awaiting.js";
function outputPlusValue(value) { return output + value }

console.log(outputPlusValue(100));
setTimeout(() => console.log(outputPlusValue(100), 1000);
```

上面代码的写法，与普通的模块加载完全一样。也就是说，模块的使用者完全不用关心，依赖模块的内部有没有异步操作，正常加载即可。

这时，模块的加载会等待依赖模块（上例是 `awaiting.js`）的异步操作完成，才执行后面的代码，有点像暂停在那里。所以，它总是会得到正确的 `output`，不会因为加载时机的不同，而得到不一样的值。

下面是顶层 `await` 的一些使用场景。

```
// import() 方法加载
const strings = await import(`/i18n/${navigator.language}`);

// 数据库操作
const connection = await dbConnector();

// 依赖回滚
let jQuery;
try {
  jQuery = await import('https://cdn-a.com/jQuery');
} catch {
  jQuery = await import('https://cdn-b.com/jQuery');
}
```

注意，如果加载多个包含顶层 `await` 命令的模块，加载命令是同步执行的。

```
// x.js
console.log("X1");
await new Promise(r => setTimeout(r, 1000));
console.log("X2");

// y.js
console.log("Y");

// z.js
import "./x.js";
import "./y.js";
console.log("Z");
```

上面代码有三个模块，最后的 `z.js` 加载 `x.js` 和 `y.js` ，打印结果是 `X1` 、 `Y` 、 `X2` 、 `Z` 。这说明， `z.js` 并没有等待 `x.js` 加载完成，再去加载 `y.js` 。

顶层的 `await` 命令有点像，交出代码的执行权给其他的模块加载，等异步操作完成后，再拿回执行权，继续向下执行。

留言