ECMAScript 6 入门

作者：阮一峰

目录

其他

async 函数

---

1. 含义

上一章 下一章

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。

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

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。

---

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) {
  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。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');
}

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`。

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, reject) {
      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 {
    var val1 = await firstStep();
    var val2 = await secondStep(val1);
    var 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);
}
```

目前，`@std/esm` 模块加载器支持顶层 `await`，即 `await` 命令可以不放在 async 函数里面，直接使用。

```
// async 函数的写法
const start = async () => {
  const res = await fetch('google.com');
  return res.text();
};

start().then(console.log);

// 顶层 await 的写法
const res = await fetch('google.com');
console.log(await res.text());
```

上面代码中，第二种写法的脚本必须使用 `@std/esm` 加载器，才会生效。

---

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

---

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
  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) {

  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;
}
```

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. 异步遍历器

《遍历器》一章说过，Iterator 接口是一种数据遍历的协议，只要调用遍历器对象的 `next` 方法，就会得到一个对象，表示当前遍历指针所在的那个位置的信息。`next` 方法返回的对象的结构是 `{value, done}`，其中 `value` 表示当前的数据的值，`done` 是一个布尔值，表示遍历是否结束。

这里隐含着一个规定，`next` 方法必须是同步的，只要调用就必须立刻返回值。也就是说，一旦执行 `next` 方法，就必须同步地得到 `value` 和 `done` 这两个属性。如果遍历指针正好指向同步操作，当然没有问题，但对于异步操作，就不太合适了。目前的解决方法是，Generator 函数里面的异步操作，返回一个 Thunk 函数或者 Promise 对象，即 `value` 属性是一个 Thunk 函数或者 Promise 对象，等待以后返回真正的值，而 `done` 属性则还是同步产生的。

目前，有一个提案，为异步操作提供原生的遍历器接口，即 `value` 和 `done` 这两个属性都是异步产生，这称为"异步遍历器"（Async Iterator）。

异步遍历的接口

异步遍历器的最大的语法特点，就是调用遍历器的 `next` 方法，返回的是一个 Promise 对象。

```
asyncIterator
  .next()
  .then(
    ({ value, done }) => /* ... */
  );
```

上面代码中，`asyncIterator` 是一个异步遍历器，调用 `next` 方法以后，返回一个 Promise 对象。因此，可以使用 `then` 方法指定，这个 Promise 对象的状态变为 `resolve` 以后的回调函数。回调函数的参数，则是一个具有 `value` 和 `done` 两个属性的对象，这个跟同步遍历器是一样的。

我们知道，一个对象的同步遍历器的接口，部署在 `Symbol.iterator` 属性上面。同样地，对象的异步遍历器接口，部署在 `Symbol.asyncIterator` 属性上面。不管是什么样的对象，只要它的 `Symbol.asyncIterator` 属性有值，就表示应该对它进行异步遍历。

下面是一个异步遍历器的例子。

```
const asyncIterable = createAsyncIterable(['a', 'b']);
const asyncIterator = asyncIterable[Symbol.asyncIterator]();

asyncIterator
.next()
.then(iterResult1 => {
  console.log(iterResult1); // { value: 'a', done: false }
  return asyncIterator.next();
})
.then(iterResult2 => {
  console.log(iterResult2); // { value: 'b', done: false }
  return asyncIterator.next();
})
.then(iterResult3 => {
  console.log(iterResult3); // { value: undefined, done: true }
});
```

上面代码中，异步遍历器其实返回了两次值。第一次调用的时候，返回一个 Promise 对象；等到 Promise 对象 `resolve` 了，再返回一个表示当前数据成员信息的对象。这就是说，异步遍历器与同步遍历器最终行为是一致的，只是会先返回 Promise 对象，作为中介。

由于异步遍历器的 `next` 方法，返回的是一个 Promise 对象。因[illegible]令后面。

```
async function f() {
  const asyncIterable = createAsyncIterable(['a', 'b']);
  const asyncIterator = asyncIterable[Symbol.asyncIterator]();
  console.log(await asyncIterator.next());
  // { value: 'a', done: false }
  console.log(await asyncIterator.next());
  // { value: 'b', done: false }
  console.log(await asyncIterator.next());
  // { value: undefined, done: true }
}
```

上面代码中，`next`方法用`await`处理以后，就不必使用`then`方法了。整个流程已经很接近同步处理了。

注意，异步遍历器的`next`方法是可以连续调用的，不必等到上一步产生的Promise对象`resolve`以后再调用。这种情况下，`next`方法会累积起来，自动按照每一步的顺序运行下去。下面是一个例子，把所有的`next`方法放在`Promise.all`方法里面。

```
const asyncGenObj = createAsyncIterable(['a', 'b']);
const [{value: v1}, {value: v2}] = await Promise.all([
  asyncGenObj.next(), asyncGenObj.next()
]);

console.log(v1, v2); // a b
```

另一种用法是一次性调用所有的`next`方法，然后`await`最后一步操作。

```
const writer = openFile('someFile.txt');
writer.next('hello');
writer.next('world');
await writer.return();
```

---

for await...of

前面介绍过，`for...of`循环用于遍历同步的 Iterator 接口。新引入的`for await...of`循环，则是用于遍历异步的 Iterator 接口。

```
async function f() {
  for await (const x of createAsyncIterable(['a', 'b'])) {
    console.log(x);
  }
}
// a
// b
```

上面代码中，`createAsyncIterable()`返回一个异步遍历器，`for...of`循环自动调用这个遍历器的`next`方法，会得到一个Promise对象。`await`用来处理这个Promise对象，一旦`resolve`，就把得到的值（`x`）传入`for...of`的循环体。

`for await...of`循环的一个用途，是部署了 asyncIterable 操作的异步接口，可以直接放入这个循环。

```
let body = '';

async function f() {
  for await(const data of req) body += data;
  const parsed = JSON.parse(body);
  console.log('got', parsed);
}
```

上面代码中，`req`是一个 asyncIterable 对象，用来异步读取数据。可以看到，使用`for await...of`循环以后，代码会非常简洁。

如果`next`方法返回的 Promise 对象被`reject`，`for await...of`就会报错，要用`try...catch`捕捉。

```
async function () {
  try {
    for await (const x of createRejectingIterable()) {
      console.log(x);
```

```
    }
  } catch (e) {
    console.error(e);
  }
}
```

注意，`for await...of`循环也可以用于同步遍历器。

```
(async function () {
  for await (const x of ['a', 'b']) {
    console.log(x);
  }
})();
// a
// b
```

异步 Generator 函数

就像 Generator 函数返回一个同步遍历器对象一样，异步 Generator 函数的作用，是返回一个异步遍历器对象。

在语法上，异步 Generator 函数就是`async`函数与 Generator 函数的结合。

```
async function* gen() {
  yield 'hello';
}
const genObj = gen();
genObj.next().then(x => console.log(x));
// { value: 'hello', done: false }
```

上面代码中，`gen`是一个异步 Generator 函数，执行后返回一个异步 Iterator 对象。对该对象调用`next`方法，返回一个 Promise 对象。

异步遍历器的设计目的之一，就是 Generator 函数处理同步操作和异步操作时，能够使用同一套接口。

```
// 同步 Generator 函数
function* map(iterable, func) {
  const iter = iterable[Symbol.iterator]();
  while (true) {
    const {value, done} = iter.next();
    if (done) break;
    yield func(value);
  }
}

// 异步 Generator 函数
async function* map(iterable, func) {
  const iter = iterable[Symbol.asyncIterator]();
  while (true) {
    const {value, done} = await iter.next();
    if (done) break;
    yield func(value);
  }
}
```

上面代码中，可以看到有了异步遍历器以后，同步 Generator 函数和异步 Generator 函数的写法基本上是一致的。

下面是另一个异步 Generator 函数的例子。

```
async function* readLines(path) {
  let file = await fileOpen(path);

  try {
    while (!file.EOF) {
      yield await file.readLine();
    }
  } finally {
    await file.close();
```

```
  }
}
```

上面代码中，异步操作前面使用`await`关键字标明，即`await`后面的操作，应该返回 Promise 对象。凡是使用`yield`关键字的地方，就是`next`方法的停下来的地方，它后面的表达式的值（即`await file.readLine()`的值），会作为`next()`返回对象的`value`属性，这一点是与同步 Generator 函数一致的。

异步 Generator 函数内部，能够同时使用`await`和`yield`命令。可以这样理解，`await`命令用于将外部操作产生的值输入函数内部，`yield`命令用于将函数内部的值输出。

上面代码定义的异步 Generator 函数的用法如下。

```
(async function () {
  for await (const line of readLines(filePath)) {
    console.log(line);
  }
})()
```

异步 Generator 函数可以与`for await...of`循环结合起来使用。

```
async function* prefixLines(asyncIterable) {
  for await (const line of asyncIterable) {
    yield '> ' + line;
  }
}
```

异步 Generator 函数的返回值是一个异步 Iterator，即每次调用它的`next`方法，会返回一个 Promise 对象，也就是说，跟在`yield`命令后面的，应该是一个 Promise 对象。

```
async function* asyncGenerator() {
  console.log('Start');
  const result = await doSomethingAsync(); // (A)
  yield 'Result: '+ result; // (B)
  console.log('Done');
}

const ag = asyncGenerator();
ag.next().then({value, done} => {
  // ...
})
```

上面代码中，`ag`是`asyncGenerator`函数返回的异步 Iterator 对象。调用`ag.next()`以后，`asyncGenerator`函数内部的执行顺序如下。

1.打印出`Start`。
2.`await`命令返回一个 Promise 对象，但是程序不会停在这里，继续往下执行。
3.程序在`B`处暂停执行，`yield`命令立刻返回一个 Promise 对象，该对象就是`ag.next()`的返回值。
4.`A`处`await`命令后面的那个 Promise 对象 resolved，产生的值放入`result`变量。
5.`B`处的 Promise 对象 resolved，`then`方法指定的回调函数开始执行，该函数的参数是一个对象，`value`的值是表达式`'Result: ' + result`的值，`done`属性的值是`false`。

A 和 B 两行的作用类似于下面的代码。

```
return new Promise((resolve, reject) => {
  doSomethingAsync()
  .then(result => {
     resolve({
       value: 'Result: '+result,
       done: false,
     });
  });
});
```

如果异步 Generator 函数抛出错误，会被 Promise 对象`reject`，然后抛出的错误被`catch`方法捕获。

```
async function* asyncGenerator() {
  throw new Error('Problem!');
}

asyncGenerator()
.next()
.catch(err => console.log(err)); // Error: Problem!
```

注意，普通的 async 函数返回的是一个 Promise 对象，而异步 Generator 函数返回的是一个异步 Iterator 对象。可以这样理解，async 函数和异步 Generator 函数，是封装异步操作的两种方法，都用来达到同一种目的。区别在于，前者自带执行器，后者通过 `for await...of` 执行，或者自己编写执行器。下面就是一个异步 Generator 函数的执行器。

```
async function takeAsync(asyncIterable, count = Infinity) {
  const result = [];
  const iterator = asyncIterable[Symbol.asyncIterator]();
  while (result.length < count) {
    const {value, done} = await iterator.next();
    if (done) break;
    result.push(value);
  }
  return result;
}
```

上面代码中，异步 Generator 函数产生的异步遍历器，会通过 `while` 循环自动执行，每当 `await iterator.next()` 完成，就会进入下一轮循环。一旦 `done` 属性变为 `true`，就会跳出循环，异步遍历器执行结束。

下面是这个自动执行器的一个使用实例。

```
async function f() {
  async function* gen() {
    yield 'a';
    yield 'b';
    yield 'c';
  }

  return await takeAsync(gen());
}

f().then(function (result) {
  console.log(result); // ['a', 'b', 'c']
})
```

异步 Generator 函数出现以后，JavaScript 就有了四种函数形式：普通函数、async 函数、Generator 函数和异步 Generator 函数。请注意区分每种函数的不同之处。基本上，如果是一系列按照顺序执行的异步操作（比如读取文件，然后写入新内容，再存入硬盘），可以使用 async 函数；如果是一系列产生相同数据结构的异步操作（比如一行一行读取文件），可以使用异步 Generator 函数。

异步 Generator 函数也可以通过 `next` 方法的参数，接收外部传入的数据。

```
const writer = openFile('someFile.txt');
writer.next('hello'); // 立即执行
writer.next('world'); // 立即执行
await writer.return(); // 等待写入结束
```

上面代码中，`openFile` 是一个异步 Generator 函数。`next` 方法的参数，向该函数内部的操作传入数据。每次 `next` 方法都是同步执行的，最后的 `await` 命令用于等待整个写入操作结束。

最后，同步的数据结构，也可以使用异步 Generator 函数。

```
async function* createAsyncIterable(syncIterable) {
  for (const elem of syncIterable) {
    yield elem;
  }
}
```

上面代码中，由于没有异步操作，所以也就没有使用 `await` 关键字。

yield* 语句

`yield*` 语句也可以跟一个异步遍历器。

```
async function* gen1() {
  yield 'a';
  yield 'b';
  return 2;
}

async function* gen2() {
  // result 最终会等于 2
  const result = yield* gen1();
}
```

上面代码中，`gen2` 函数里面的 `result` 变量，最后的值是 `2`。

与同步 Generator 函数一样，`for await...of` 循环会展开 `yield*`。

```
(async function () {
  for await (const x of gen2()) {
    console.log(x);
  }
})();
// a
// b
```

留言