ECMAScript 6 入门

作者：阮一峰

目录

其他

- 源码
- 修订历史
- 反馈意见

异步遍历器

上一章 下一章

---

1. 同步遍历器的问题

《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。

```
function idMaker() {
  let index = 0;

  return {
    next: function() {
      return { value: index++, done: false };
    }
  };
}

const it = idMaker();

it.next().value // 0
it.next().value // 1
it.next().value // 2
// ...
```

上面代码中，变量 `it` 是一个遍历器（iterator）。每次调用 `it.next()` 方法，就返回一个对象，表示当前遍历位置的信息。

这里隐含着一个规定，`it.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 `it.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。

```
function idMaker() {
  let index = 0;

  return {
    next: function() {
      return new Promise(function (resolve, reject) {
        setTimeout(() => {
          resolve({ value: index++, done: false });
        }, 1000);
      });
    }
  };
}
```

上面代码中，`next()` 方法返回的是一个 Promise 对象，这样就不行，不符合 Iterator 协议，只要代码里面包含异步操作都不行。也就是说，Iterator 协议里面 `next()` 方法只能包含同步操作。

目前的解决方法是，将异步操作包装成 Thunk 函数或者 Promise 对象，即 `next()` 方法返回值的 `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。

```
function idMaker() {
  let index = 0;

  return {
    next: function() {
```

```
      return {
        value: new Promise(resolve => setTimeout(() => resolve(index++), 1000)),
        done: false
      };
    }
  };
}

const it = idMaker();

it.next().value.then(o => console.log(o)) // 1
it.next().value.then(o => console.log(o)) // 2
it.next().value.then(o => console.log(o)) // 3
// ...
```

上面代码中， `value` 属性的返回值是一个 Promise 对象，用来放置异步操作。但是这样写很麻烦，不太符合直觉，语义也比较绕。

ES2018 引入了"异步遍历器"（Async Iterator），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。

---

2. 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关键的区别，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。因此，可以把它放在 await 命令后面。

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

```
async function runner() {
  const writer = openFile('someFile.txt');
  writer.next('hello');
  writer.next('world');
  await writer.return();
}

runner();
```

---

3. 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() 返回一个拥有异步遍历器接口的对象， for...of 循环自动调用这个对象的异步遍历器的 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
```

Node v10 支持异步遍历器，Stream 就部署了这个接口。下面是读取文件的传统写法与异步遍历器写法的差异。

```
// 传统写法
function main(inputFilePath) {
  const readStream = fs.createReadStream(
    inputFilePath,
    { encoding: 'utf8', highWaterMark: 1024 }
  );
  readStream.on('data', (chunk) => {
    console.log('>>> '+chunk);
  });
  readStream.on('end', () => {
    console.log('### DONE ###');
  });
}

// 异步遍历器写法
async function main(inputFilePath) {
  const readStream = fs.createReadStream(
    inputFilePath,
    { encoding: 'utf8', highWaterMark: 1024 }
  );

  for await (const chunk of readStream) {
    console.log('>>> '+chunk);
  }
  console.log('### DONE ###');
}
```

4. 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， `map` 是一个 Generator 函数，第一个参数是可遍历对象 `iterable` ，第二个参数是一个回调函数 `func` 。 `map` 的作用是将 `iterable` 每一步返回的值，使用 `func` 进行处理。上面有两个版本的 `map` ，前一个处理同步遍历器，后一个处理异步遍历器，可以看到两个版本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。如果像上面那个例子那样， `yield` 命令后面是一个字符串，会被自动包装成一个 Promise 对象。

```
function fetchRandom() {
  const url = 'https://www.random.org/decimal-fractions/'
    + '?num=1&dec=10&col=1&format=plain&rnd=new';
  return fetch(url);
}

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

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

上面代码中， `ag` 是 `asyncGenerator` 函数返回的异步遍历器对象。调用 `ag.next()` 以后，上面代码的执行顺序如下。

1. `ag.next()` 立刻返回一个 Promise 对象。
2. `asyncGenerator` 函数开始执行，打印出 `Start` 。
3. `await` 命令返回一个 Promise 对象， `asyncGenerator` 函数停在这里。
4. A 处变成 fulfilled 状态，产生的值放入 `result` 变量， `asyncGenerator` 函数继续往下执行。
5. 函数在 B 处的 `yield` 暂停执行，一旦 `yield` 命令取到值， `ag.next()` 返回的那个 Promise 对象变成 fulfilled 状态。
6. `ag.next()` 后面的 `then` 方法指定的回调函数开始执行。该回调函数的参数是一个对象 `{value, done}` ，其中 `value` 的值是 `yield` 命令后面的那个表达式的值， `done` 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) => {
  fetchRandom()
  .then(result => result.text())
```

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

如果异步 Generator 函数抛出错误，会导致 Promise 对象的状态变为 `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。

---

5. 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