图灵社区会员 aSINKz(1561821892@qq.com) 专享 尊重版权

附录A

ES2018 和 ES2019

从 ECMAScript 2015 开始，TC39 委员会改为每年发布一版新 ECMAScript 规范。这样各个提案可以独立发展，每年所有达到成熟阶段的提案会被打包发布到新一版标准中。不过，打包多少特性并不重要，主要取决于浏览器厂商实现的情况。一旦提案进入第 4 阶段（stage 4），其内容就不会更改，并通常会包含在下一版 ECMAScript 规范中，浏览器就会着手根据自己的计划实现提案的特性。

ECMAScript 2018 于 2018 年 1 月完成，主要增加了异步迭代、剩余和扩展操作符、正则表达式和期约等方面的特性。可以通过 TC39 委员会的 GitHub 仓库了解规范的最新动态。

> **注意** 本附录中介绍的特性相对较新，往往只有新近版本的浏览器才支持。使用这些特性之前，请参考 Can I Use 网站确定支持相应特性的浏览器及其版本。

A.1 异步迭代

在 ECMAScript 最近发布的几个版本中，异步执行和迭代器协议是两个极其热门的主题。异步执行用于释放对执行线程的控制以执行慢操作和收回控制，而迭代器协议则涉及为任意对象定义规范顺序。异步迭代只是这两个概念在逻辑上的统一。

同步迭代器在每次调用 `next()` 时都会返回 `{ value, done }` 对象。当然，这要求确定这个对象内容的计算和资源获取在 `next()` 调用退出时必须完成，否则这些值就无法确定。在使用**同步**迭代器迭代**异步**确定的值时，主执行线程会被阻塞，以等待异步操作完成。

有了异步迭代器，这个问题就迎刃而解了。异步迭代器在每次调用 `next()` 时会提供解决为 `{ value, done }` 对象的期约。这样，执行线程可以释放并在当前这步循环完成之前执行其他任务。

A.1.1 创建并使用异步迭代器

要理解异步迭代器，最简单的办法是用它跟同步迭代器进行比较。下面代码中创建了一个简单的 `Emitter` 类，该类包含一个同步生成器函数，该函数会产生一个同步迭代器，同步迭代器输出 0~4：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.syncIdx = 0;
  }

  *[Symbol.iterator]() {
    while(this.syncIdx < this.max) {
      yield this.syncIdx++;
```

```
    }
  }
}

const emitter = new Emitter(5);

function syncCount() {
  const syncCounter = emitter[Symbol.iterator]();

  for (const x of syncCounter) {
    console.log(x);
  }
}

syncCount();
// 0
// 1
// 2
// 3
// 4
```

这个例子之所以可以运行起来，主要是因为迭代器可以立即产生下一个值。假如你不想在确定下一个产生的值时阻塞主线程执行，也可以定义异步迭代器函数，让它产生期约包装的值。

为此，要使用迭代器和生成器的异步版本。ECMAScript 2018 为此定义了 `Symbol.asyncIterator`，以便定义和调用输出期约的生成器函数。同时，这一版规范还为异步迭代器增加了 `for-await-of` 循环，用于使用异步迭代器。

相应地，前面的例子可以扩展为同时支持同步和异步迭代：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.syncIdx = 0;
    this.asyncIdx = 0;
  }

  *[Symbol.iterator]() {
    while(this.syncIdx < this.max) {
      yield this.syncIdx++;
    }
  }

  async *[Symbol.asyncIterator]() {
  // *[Symbol.asyncIterator]() {
    while(this.asyncIdx < this.max) {
      // yield new Promise((resolve) => resolve(this.asyncIdx++));
    yield this.asyncIdx++
    }
  }
}

const emitter = new Emitter(5);

function syncCount() {
  const syncCounter = emitter[Symbol.iterator]();

  for (const x of syncCounter) {
    console.log(x);
```

图灵社区会员 aSINKz(1561821892@qq.com) 专享 尊重版权

```
  }
}

async function asyncCount() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for await(const x of asyncCounter) {
    console.log(x);
  }
}

syncCount();
// 0
// 1
// 2
// 3
// 4

asyncCount();
// 0
// 1
// 2
// 3
// 4
```

为了加深理解，可以把前面例子中的**同步**生成器传给 `for-await-of` 循环：

```
const emitter = new Emitter(5);

async function asyncIteratorSyncCount() {
  const syncCounter = emitter[Symbol.iterator]();

  for await(const x of syncCounter) {
    console.log(x);
  }
}

asyncIteratorSyncCount();
// 0
// 1
// 2
// 3
// 4
```

虽然这里迭代的是同步生成器产生的原始值，但 `for-await-of` 循环仍像它们被包装在期约中一样处理它们。这说明 `for-await-of` 循环可以流畅地处理同步和异步可迭代对象。但是常规 `for` 循环就不能处理异步迭代器了：

```
function syncIteratorAsyncCount() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for (const x of asyncCounter) {
    console.log(x);
  }
}

syncIteratorAsyncCount();
// TypeError: asyncCounter is not iterable
```

图灵社区会员 aSINKz(1561821892@qq.com) 专享 尊重版权

关于异步迭代器，要理解的非常重要的一个概念是 Symbol.asyncIterator 符号不会改变生成器函数的行为或者消费生成器的方式。注意在前面的例子中，生成器函数加上了 async 修饰符成为异步函数，又加上了星号成为生成器函数。Symbol.asyncIterator 在这里只起一个提示的作用，告诉将来消费这个迭代器的外部结构如 for-await-of 循环，这个迭代器会返回期约对象的序列。

A.1.2　理解异步迭代器队列

当然，前面的例子是假想的，因为迭代器返回的期约都会立即解决，所以跟同步迭代器的区别很难看出来。想象一下迭代器返回的期约会在不确定的时间解决，而且它们返回的顺序是乱的。异步迭代器应该尽可能模拟同步迭代器，包括每次迭代时代码的按顺序执行。为此，异步迭代器会维护一个回调队列，以保证早期值的迭代器处理程序总是会在处理晚期值之前完成，即使后面的值早于之前的值解决。

为验证这一点，下面的例子中的异步迭代器以随机时长返回期约。异步迭代队列可以保证期约解决的顺序不会干扰迭代顺序。结果应该按顺序打印一组整数（但间隔时间随机）：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.syncIdx = 0;
    this.asyncIdx = 0;
  }

  *[Symbol.iterator]() {
    while(this.syncIdx < this.max) {
      yield this.syncIdx++;
    }
  }
  async *[Symbol.asyncIterator]() {
    while(this.asyncIdx < this.max) {
      yield new Promise((resolve) => {
        setTimeout(() => {
          resolve(this.asyncIdx++);
        }, Math.floor(Math.random() * 1000));
      });
    }
  }
}

const emitter = new Emitter(5);

function syncCount() {
  const syncCounter = emitter[Symbol.iterator]();

  for (const x of syncCounter) {
    console.log(x);
  }
}

async function asyncCount() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for await(const x of asyncCounter) {
    console.log(x);
  }
```

```
}

syncCount();
// 0
// 1
// 2
// 3
// 4

asyncCount();
// 0
// 1
// 2
// 3
// 4
```

A.1.3　处理异步迭代器的 reject()

因为异步迭代器使用期约来包装返回值，所以必须考虑某个期约被拒绝的情况。由于异步迭代会按顺序完成，而在循环中跳过被拒绝的期间是不合理的。因此，被拒绝的期约会强制退出迭代器：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.asyncIdx = 0;
  }

  async *[Symbol.asyncIterator]() {
    while (this.asyncIdx < this.max) {
      if (this.asyncIdx < 3) {
        yield this.asyncIdx++;
      } else {
        throw 'Exited loop';
      }
    }
  }
}

const emitter = new Emitter(5);

async function asyncCount() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for await (const x of asyncCounter) {
    console.log(x);
  }
}

asyncCount();
// 0
// 1
// 2
// Uncaught (in promise) Exited loop
```

A.1.4　使用 next()手动异步迭代

for-await-of 循环提供了两个有用的特性：一是利用异步迭代器队列保证按顺序执行，二是隐藏异步迭代器的期约。不过，使用这个循环会隐藏很多底层行为。

因为异步迭代器仍遵守迭代器协议，所以可以使用 next()逐个遍历异步可迭代对象。如前所述，next()返回的值会包含一个期约，该期约可解决为{ value, done }这样的迭代结果。这意味着必须使用期约 API 获取方法，同时也意味着可以不使用异步迭代器队列。

```
const emitter = new Emitter(5);

const asyncCounter = emitter[Symbol.asyncIterator]();

console.log(asyncCounter.next());
// Promise<{value, done}>
```

A.1.5　顶级异步循环

一般来说，包括 for-await-of 循环在内的异步行为不能出现在异步函数外部。不过，有时候可能确实需要在这样的上下文使用异步行为。为此可以通过创建异步 IIFE 来达到目的：

```
class Emitter {
  constructor(max) {
    this.max = max;
    this.asyncIdx = 0;
  }

  async *[Symbol.asyncIterator]() {
    while(this.asyncIdx < this.max) {
      yield new Promise((resolve) => resolve(this.asyncIdx++));
    }
  }
}

const emitter = new Emitter(5);

(async function() {
  const asyncCounter = emitter[Symbol.asyncIterator]();

  for await(const x of asyncCounter) {
    console.log(x);
  }
})();
// 0
// 1
// 2
// 3
// 4
```

A.1.6　实现可观察对象

异步迭代器可以耐心等待下一次迭代而不会导致计算成本，那么这也为实现可观察对象（Observable）接口提供了可能。总体上看，这涉及捕获事件，将它们封装在期约中，然后把这些事件提供给迭代器，

而处理程序可以利用这些异步迭代器。在某个事件触发时，异步迭代器的下一个期约会解决为该事件。

> **注意**　可观察对象的话题超出了本书范围，因为它们很大程度上是作为第三方库实现的。有兴趣的读者可以了解一下非常流行的 RxJS 库。

下面这个简单的例子会捕获浏览器事件的可观察流。这需要一个期约的队列，每个期约对应一个事件。该队列也会保持事件生成的顺序，对这种问题来说保持顺序也是合理的。

```
class Observable {
  constructor() {
    this.promiseQueue = [];

    // 保存用于解决队列中下一个期约的程序
    this.resolve = null;

    // 把最初的期约推到队列
    // 该期约会解决为第一个观察到的事件
    this.enqueue();
  }

  // 创建新期约，保存其解决方法
  // 再把它保存到队列中
  enqueue() {
    this.promiseQueue.push(
      new Promise((resolve) => this.resolve = resolve));
  }

  // 从队列前端移除期约
  // 并返回它
  dequeue() {
    return this.promiseQueue.shift();
  }
}
```

要利用这个期约队列，可以在这个类上定义一个异步生成器方法。该生成器可用于任何类型的事件：

```
class Observable {
  constructor() {
    this.promiseQueue = [];

    // 保存用于解决队列中下一个期约的程序
    this.resolve = null;

    // 把最初的期约推到队列
    // 该期约会解决为第一个观察到的事件
    this.enqueue();
  }

  // 创建新期约，保存其解决方法
  // 再把它保存到队列中
  enqueue() {
    this.promiseQueue.push(
      new Promise((resolve) => this.resolve = resolve));
  }

  // 从队列前端移除期约
```

```
    // 并返回它
    dequeue() {
      return this.promiseQueue.shift();
    }

    async *fromEvent (element, eventType) {
      // 在有事件生成时，用事件对象来解决队列头部的期约
      // 同时把另一个期约加入队列
      element.addEventListener(eventType, (event) => {
        this.resolve(event);
        this.enqueue();
      });

      // 每次解决队列前面的期约
      // 都会向异步迭代器返回相应的事件对象
      while (1) {
        yield await this.dequeue();
      }
    }
  }
```

这样，这个类就定义完了。接下来在 DOM 元素上定义可观察对象就很简单了。假设页面上有一个<button>元素，可以像下面这样捕获该按钮上的一系列 click 事件，然后在控制台把它们打印出来：

```
  class Observable {
    constructor() {
      this.promiseQueue = [];

      // 保存用于解决队列中下一个期约的程序
      this.resolve = null;

      // 把最初的期约推到队列
      // 该期约会解决为第一个观察到的事件
      this.enqueue();
    }

    // 创建新期约，保存其解决方法
    // 再把它保存到队列中
    enqueue() {
      this.promiseQueue.push(
        new Promise((resolve) => this.resolve = resolve));
    }

    // 从队列前端移除期约
    // 并返回它
    dequeue() {
      return this.promiseQueue.shift();
    }

    async *fromEvent (element, eventType) {
      // 在有事件生成时，用事件对象来解决 队列头部的期约
      // 同时把另一个期约加入队列
      element.addEventListener(eventType, (event) => {
        this.resolve(event);
        this.enqueue();
      });

      // 每次解决队列前面的期约
      // 都会向异步迭代器返回相应的事件对象
```

```
    while (1) {
      yield await this.dequeue();
    }
  }
}

(async function() {
  const observable = new Observable();

  const button = document.querySelector('button');
  const mouseClickIterator = observable.fromEvent(button, 'click');

  for await (const clickEvent of mouseClickIterator) {
    console.log(clickEvent);
  }
})();
```

A.2　对象字面量的剩余操作符和扩展操作符

ECMAScript 2018 将数组中的剩余操作符和扩展操作符也移植到了对象字面量。这极大地方便了对象合并和通过其他对象创建新对象。

A.2.1　剩余操作符

剩余操作符可以在解构对象时将所有剩下未指定的可枚举属性收集到一个对象中。比如：

```
const person = { name: 'Matt', age: 27, job: 'Engineer' };
const { name, ...remainingData } = person;

console.log(name); // Matt
console.log(remainingData); // { age: 27, job: 'Engineer' }
```

每个对象字面量中最多可以使用一次剩余操作符，而且必须放在最后。因为每个对象字面量只能有一个剩余操作符，所以可以嵌套剩余操作符。嵌套时，因为子属性对应的剩余操作符没有歧义，所以返回对象的内容不会重叠：

```
const person = { name: 'Matt', age: 27, job: { title: 'Engineer', level: 10 } };

const { name, job: { title, ...remainingJobData }, ...remainingPersonData } = person;

console.log(name);                  // Matt
console.log(title);                 // Engineer
console.log(remainingPersonData);   // { age: 27 }
console.log(remainingJobData);      // { level: 10 }

const { ...a, job } = person;
// SyntaxError: Rest element must be last element
```

剩余操作符在对象间执行浅复制，因此会复制对象的引用而不会克隆整个对象：

```
const person = { name: 'Matt', age: 27, job: { title: 'Engineer', level: 10 } };

const { ...remainingData } = person;

console.log(person === remainingData);          // false
console.log(person.job === remainingData.job);  // true
```