# es9新特性

# 1.rest扩展符，...

|  |
| --- |
|  |

## 可以扩展对象和打包参数，还可以将可迭代对象转换为数组

### 应用场景举例

|  |
| --- |
|  |

# 2.正则扩展

## 课堂笔记

|  |
| --- |
|  |

### 上面是以前的正则，现在可以使用新方法，可以给每一个分组起一个名字

|  |
| --- |
| <body>      <script>          let input = "今天是2024-02-14日,情人节快乐"  //以前的做法          // let reg = /([0-9]{4})-([0-9]{2})-([0-9]{2})/g          // let ret = reg.exec(input) //语法：正则对象.exec(输入字符串) 返回匹配结果          // console.log(ret);          //命名捕获          let reg = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g          let ret = reg.exec(input)          console.log(ret);      </script>  </body> |
|  |

### 然后可以根据名称获取需要的结果

|  |
| --- |
| <script>          let input = "今天是2024-02-14日,情人节快乐"          // let reg = /([0-9]{4})-([0-9]{2})-([0-9]{2})/g          // let ret = reg.exec(input) //语法：正则对象.exec(输入字符串) 返回匹配结果          // console.log(ret);          //命名捕获          let reg = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g          let ret = reg.exec(input)          console.log(ret.groups.year);      </script> |
|  |

### 也可以使用解构表达式

|  |
| --- |
| <script>          let input = "今天是2024-02-14日,情人节快乐"          // let reg = /([0-9]{4})-([0-9]{2})-([0-9]{2})/g          // let ret = reg.exec(input) //语法：正则对象.exec(输入字符串) 返回匹配结果          // console.log(ret);          //命名捕获          let reg = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g          let ret = reg.exec(input)          // console.log(ret.groups.year);          let {year,month,day} = ret.groups          console.log(year,month,day);      </script> |
|  |

# 3.Promise.finally

## 课堂笔记

|  |
| --- |
|  |

# 4.异步迭代

## 课堂笔记

### 异步生成器课堂案例1，

#### 面试题，同步执行异步对象数组里面的每一个异步成员的任务，要求时间间隔要于任务所需要的时间一致

|  |
| --- |
|  |

#### 上面的做法还是有点问题的，因为他是同步执行的，达不到这个效果，怎么办？

### 此时需要使用es9新的for await...of遍历方式，在这之前需要先做一个异步生成器，否则没有效果

|  |
| --- |
| <script>          //for await ... of需要配合异步生成器来用，否则没有我们需要的效果，这个需要注意          function timer(t){              return new Promise((resolve)=>{                  setTimeout(()=>{                       resolve("data="+t)                  },t)              })          }          //异步生成器          async function \*gen(){              yield await timer(1000)              yield await timer(2000)              yield await timer(3000)          }      async function doAsync() {            let g= gen()          let arr = [g.next(),g.next(),g.next()]          for await (let i of arr){            console.log("start",Date.now());            console.log(i);            console.log("end",Date.now());          }      }      doAsync()      </script> |
|  |

#### 在实际开发中可以把timer函数替换为实际的异步操作

### 异步生成器课堂案例2

|  |
| --- |
| <script>          //异步生成器          async function \*Gen() {              yield await Promise.resolve("data1")              yield await Promise.resolve("data2")              yield await Promise.resolve("data3")          }          let asyncIt = Gen()          async function traval() {              for await (let a of asyncIt){                  console.log(a);              }          }            traval()      </script> |
|  |

### 扩展：异步遍历器在nodejs中的应用

#### 以前的写法

|  |
| --- |
|  |

#### nodejs新写法

|  |
| --- |
|  |

# ES9中的异步迭代器（Async iterator）和异步生成器（Async generator）

ES9新增了异步迭代器（Async iterator），异步执行语句（for...await...of）和异步生成器（Async generator），本文带领大家了解这三个新特性，以及如何创建异步迭代器。

## 1. 迭代器（Iterator）

如果你还不了解ES6的迭代器，也就是iterator，先来看看这一部分。

iterator是一个特殊的对象，它包含一个next方法，next方法返回一个对象，这个对象包含两个属性，一个是value，表示成员的值，一个是done，done的值是一个布尔类型，表示迭代器是否结束。

复制代码

iterator.next() *// 返回 {value: '', done: false}*

迭代器内部会保存一个指针，指向迭代器的成员位置，每调用一次next方法，指针就会移动到下一个成员，直到指针指向迭代器最后一个成员后面的位置，这时，done的值为true，value的值一般为undefined，需要根据iterator的实际实现来决定。

### 1.1 创建iterator

实现一个函数，用来创建iterator。

几个关键点

* iterator是一个对象，并且含有一个next方法
* next方法返回一个对象，包含一个value属性和done属性，value表示返回的值，done是一个布尔类型，表示迭代器是否结束
* iterator内部包含一个内部指针，指向迭代器的成员的位置，每调用一次next方法，指针就会移动到下一个成员，直到指针指向迭代器最后一个成员后面的位置，这时done的值为true

*// 可以通过传入数组或者对象创建iterator*

**const** createIterator = items => {

**const** keys = Object.keys(items)

**const** len = keys.length

**let** pointer = 0

**return** {

next() {

**const** done = pointer >= len

**const** value = !done ? items[keys[pointer++]] : undefined

**return** {

value,

done

}

}

}

}

**const** iterator1 = createIterator([1, 2, 3])

iterator.next() *// { value: 1, done: false }*

iterator.next() *// { value: 2, done: false }*

iterator.next() *// { value: 3, done: false }*

iterator.next() *// { value: undefined, done: true }*

**const** iterator2 = createIterator({a: 'a', b: 'b', c: 'c'})

iterator.next() *// { value: 'a', done: false }*

iterator.next() *// { value: 'b', done: false }*

iterator.next() *// { value: 'c', done: false }*

iterator.next() *// { value: undefined, done: true }*

### 1.2 iterator和for...of

部署了iterator接口的数据结构，也就是具有Symbol.iterator方法的数据结构，就可以被for...of遍历。Symbol.iterator方法类似于上面实现的createIterator函数

* 数组原生具有iterator接口

**const** arr = [1, 2, 3]

**typeof** arr[Symbol.iterator] *// 'function'*

**for** (**const** val **of** arr) {

console.log(val)

}

*// 1*

*// 2*

*// 3*

* 对象默认没有iterator接口，但是我们可以自己部署

**const** obj = {a: 'a', b: 'b', c: 'c'}

**typeof** obj[Symbol.iterator] *// 'undefined'*

**for** (**const** val **of** obj) {

console.log(val)

}

*// TypeError: obj is not iterable*

给对象部署iterator接口

**const** obj = {a: 'a', b: 'b', c: 'c'}

obj[Symbol.iterator] = **function**() {

**const** self = **this**

**const** keys = Object.keys(self)

**const** len = keys.length

**let** pointer = 0

**return** {

next() {

**const** done = pointer >= len

**const** value = !done ? self[keys[pointer++]] : undefined

**return** {

value,

done

}

}

}

}

**for** (**const** val **of** obj) {

console.log(val)

}

*// a*

*// b*

*// c*

## 2. 生成器（Generator）

Generator是一个特殊的函数，函数体内部使用yield表达式，定义不同的内部状态，当执行Generator函数时，不会直接执行函数体，而是会返回一个遍历器对象（iterator）。

* Generator函数内部可以使用yield表达式，定义内部状态
* function关键字与函数名之间有一个\*

**function**\* **generator**() {

console.log('start');

**yield** 1

**yield** 2

**yield** 3

console.log('end')

}

**const** iterator = generator() *// 这时函数体并没有被执行，而是创建了一个iterator*

*// 当调用iterator的next方法时，函数体开始执行，*

iterator.next() *// 'start' {value: 1, done: false}*

iterator.next() *// {value: 2, done: false}*

iterator.next() *// {value: 3, done: false}*

iterator.next() *// 'end' {value: undefined, done: true}*

* 每调用一次next方法，函数体会从函数头部或上次停下来的地方开始执行，直到遇到下一个yield表达式或者return语句时停止
* yield表达式后面的值会作为next方法返回的对象的value属性值
* return会作为iterator结束的标记，并且return的值会作为next方法返回的对象的value属性值

改写一下上面的例子

**function**\* **generator**() {

**yield** 1

**yield** 2

**return** 3

}

**const** iterator = generator()

*// 当调用iterator的next方法时，函数体开始执行，*

iterator.next() *// {value: 1, done: false}*

iterator.next() *// {value: 2, done: false}*

iterator.next() *// {value: 3, done: true}*

Generator函数生成的iterator可以被for...of遍历

**function**\* **generator**() {

**yield** 1

**yield** 2

**yield** 3

}

**const** iterator = generator()

**typeof** iterator[Symbol.iterator] *// 'function'*

**for** (**const** val **of** iterator) {

console.log(val)

}

*// 1*

*// 2*

*// 3*

在这里我们只需要知道Generator函数会生成一个iterator就够了，但实际上Generator函数远不止这些，这里我们不做详细介绍了，感兴趣的同学可以看看阮一峰[Generator教程](https://link.juejin.cn/?target=http%3A%2F%2Fes6.ruanyifeng.com%2F%23docs%2Fgenerator%23%25E7%25AE%2580%25E4%25BB%258B)

## 3. 异步迭代器（Asynchronous Iterator）

ES9新增了异步迭代器

异步迭代器和同步迭代器相同，都是一个函数，并且含有一个next方法，区别在于同步迭代器的next方法返回一个含有value和done属性的对象，而异步迭代器的next方法返回一个Promise对象，并且Promise对象的值为含有value和done属性的对象。

*// 这是一个异步迭代器*

asyncIterator.next().then(res => {

console.log(res.value, res.done)

})

**const** createAsyncIterator = items => {

**const** keys = Object.keys(items)

**const** len = keys.length

**let** pointer = 0

**return** {

next() {

**const** done = pointer >= len

**const** value = !done ? items[keys[pointer++]] : undefined

**return** Promise.resolve({

value,

done

})

}

}

}

和同步迭代器相同，每调用一次next方法，异步迭代器内部的指针就移动到下一个成员

**const** aynscIterator = createAsyncIterator([1, 2, 3])

aynscIterator.next().then(({value, done}) => {

console.log(value, done) *// 1 false*

})

aynscIterator.next().then(({value, done}) => {

console.log(value, done) *// 2 false*

})

aynscIterator.next().then(({value, done}) => {

console.log(value, done) *// 3 false*

})

aynscIterator.next().then(({value, done}) => {

console.log(value, done) *// undefined true*

})

### 3.1 for await...of

for...of方法能够遍历具有Symbol.iterator接口的同步迭代器数据，但是不能遍历异步迭代器。 ES9新增的for...await...of可以用来遍历具有Symbol.asyncIterator方法的数据结构，也就是异步迭代器，且会等待前一个成员的状态改变后才会遍历到下一个成员，相当于async函数内部的await。

定义一个具有Symbol.asyncIterator方法的对象

**const** asyncItems = {

a: 1,

b: 2,

c: 3,

[Symbol.asyncIterator]() {

**const** items = **this**

**const** keys = Object.keys(items)

**const** len = keys.length

**let** pointer = 0

**return** {

next() {

**const** done = pointer >= len

**const** value = !done ? items[keys[pointer++]] : undefined;

**return** **new** Promise((resolve) => {

setTimeout(() => {

resolve({value, done})

}, 1000)

})

}

}

}

}

使用for await...of遍历该对象

*// await只能用在async函数中*

**async** **function** **run**() {

**for** **await** (**const** value **of** asyncItems) {

console.log(value);

}

}

run();

*// 1s后打印出 1*

*// 再过1s后打印出 2*

*// 再过1s后打印出 3*

上面的例子实现了每隔1s打印出对象的属性值的异步遍历器接口，可以看到， 当使用for...await..of，遍历时，会等待前一个Promise对象的状态改变后，再遍历到下一个成员。

### 3.2 异步生成器（Async Generator）

我们可以采取一种更方便的方式创建异步迭代器，就是利用异步生成器。

异步生成器和普通的生成器很像，但是其是async函数，内部可以使用await表达式，并且它返回一个具有Symbol.asyncIterator方法的对象。

定义一个异步生成器

**async** **function**\* **asyncGenerator**() {

**yield** **await** Promise.resolve(1);

**yield** **await** Promise.resolve(2);

**yield** **await** Promise.resolve(3);

}

使用for await...of遍历该对象

**const** asyncIterator = asyncGenerator()

**typeof** asyncIterator[Symbol.asyncIterator] *// 'function'*

**async** **function** **run**() {

**for** **await** (**const** value **of** asyncIterator) {

console.log(value);

}

}

run();

*// 1*

*// 2*

*// 3*

## 4. 总结

* 异步迭代器与同步迭代器相同的是，异步迭代器也是一个具有next方法的对象
* 异步迭代器对象的next方法返回一个Promise对象，Promise对象的值为一个对象，包含一个value属性和一个done属性
* for...await...of可以遍历具有Symbol.asyncIterator方法的数据结构，并且会等待上一个成员状态改变后再继续执行
* 异步生成器（Async Generator）可以用来创建异步迭代器，它是一个async类型的generator函数，内部可以使用await表达式等待异步方法的执行完成，并使用for...await...of遍历