第16章 异步编程

随着计算机的不断发展，用户对计算机应用的要求越来越高，需要提供更多、更智能、响应速度更快的功能。这就离不开异步编程的话题。同时，随着互联网时代的崛起，网络应用要求能够支持更多的并发量，这显然也要用到大量的异步编程。那么从这节课开始，我们会学习到底什么是异步编程，以及在JS中如何实现异步编程。
本章我们将学习如下内容：

- 什么是异步编程。
- 回调和Promise。
- 生成器Generator。
- ES7中的异步实现Async和Await。

16-1 异步编程概述

16-1-1 什么是异步编程?

我们先来看看到底什么是异步。提到异步就不得不提另外一个概念：同步。那什么又叫同步呢。很多初学者在刚接触这个概念时会想当然的认为同步就是同时进行。显然，这样的理解是错误的，咱不能按字面意思去理解它。同步，英文全称叫做Synchronization。它是指同一时间只能做一件事，也就是说一件事情做完了才能做另外一件事。

比如咱们去火车站买票，假设窗口只有1个，那么同一时间只能处理1个人的购票业务，其余的需要进行排队。这种one by one的动作就是同步。这种同步的情况其实有很多，任何需要排队的情况都可以理解成同步。那如果在程序中呢，我们都知道代码的执行是一行接着一行的，比如下面这段代码：

```
let ary = [];
for(let i = 0;i < 100;i++){
    ary[i] = i;
}
console.log(ary);
```

这段代码的执行就是从上往下依次执行，循环没执行完，输出的代码就不会执行，这就是典型的同步。在程序中，绝大多数代码都是同步的。

同步操作的优点在于做任何事情都是依次执行，井然有序，不会存在大家同时抢一个资源的问

题。你想想，如果火车站取消排队机制，那么大家势必会争先恐后去抢着买票，造成的结果就是秩序大乱，甚至可能引发一系列安全问题。如果代码不是同步执行的又会发生什么呢？有些代码需要依赖前面代码执行后的结果，但现在大家都是同时执行，那结果就不一定能获取到。而且这些代码可能在对同一数据就进行操作，也会让这个数据的值出现不确定的情况。

当然同步也有它的缺点。由于是依次进行，假如其中某一个步骤花的时间比较长，那么后续动作就会等待它的完成，从而影响效率。

不过，在有些时候我们还是希望能够在效率上有所提升，也就是说可以让很多操作同时进行。这就是另外一个概念：异步。假设火车站有10个人需要买票，现在只有1个窗口提供服务，如果平均每个人耗费5分钟，那么总共需要50分钟才能办完所有人的业务。火车站为了提高效率，加开了9个窗口，现在一共有10个窗口提供服务，那么这10个人就可以同时办理了，总共只需要5分钟，他们所有人的业务都可以办完。这就是异步带来的优势。

16-1-2 异步的实现

1. 多线程

像刚才例子中开多个窗口的方式称为多线程。线程可以理解成一个应用程序中的执行任务，每个应用程序至少会有一个线程，它被称为主线程。如果你想实现异步处理，就可以通过开启多个线程，这些线程可以同时执行。这是异步实现的一种方式。不过这种方式还是属于阻塞式的。

什么叫做阻塞式呢。你想想，开10个窗口可以满足10个人同时买票。但是现在有100个人呢？不可能再开90个窗口吧，所以每个窗口实际上还是需要排队。也就是说虽然我可以通过开启多个线程来同时执行很多任务，但是每个任务中的代码仍然是同步的。当某个任务的代码执行时间过长，也只会影响到当前线程的代码，而其他线程的代码不会受到影响。

2. 单线程非阻塞式

假设现在火车站不想开那么多窗口，还是只有1个窗口提供服务，那如何能够提高购票效率呢？我们可以这样做，把购票的流程分为两步，第一步：预定及付款。第二步：取票。其中，第一步可以让购票者在网上操作。第二步到火车站的窗口取票。这样，最耗时的工作已经提前完成，不需要排队。到火车站时，虽然只有1个窗口，1次也只能接待1个人，但是取票的动作很快，平均每个人耗时不到1分钟，10个人也就不到10分钟就可以处理完成。这样既提高了效率，又少开了窗口。这也是一种异步的实现。我们可以看到，开1个窗口，就相当于只有1个线程。然后把耗时的一些操作分成两部分，先把快速能做完的事情做了，这样保证它不会阻塞其他代码的运行。剩下耗时的部分再单独执行。这就是单线程阻塞式的异步实现机制。

16-1-3 JS中的异步实现

我们知道JS引擎就是以单线程的机制来运行代码。那么在JS代码中想要实现异步就只有采用单

线程非阻塞式的方式。比如下面这段代码：

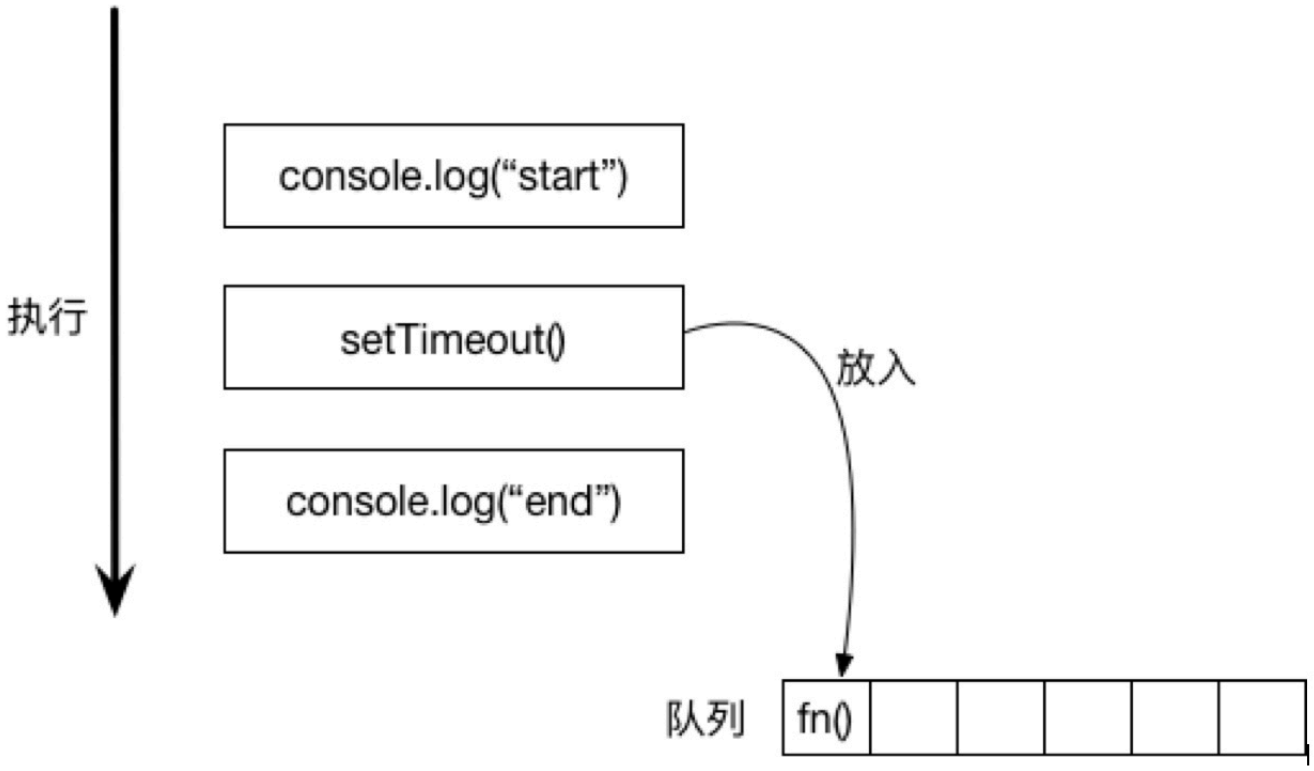
```
console.log("start");
setTimeout(function(){
    console.log("timeout");
},5000);
console.log("end");
```

这段代码先输出一个字符串"start"，然后用时间延迟函数，等到5000秒钟后输出"timeout"，在代码的最后输出"end"。最后的执行结果是：

```
start
end
//等待5秒后
timeout
```

从结果可以看到end的输出并没有等待时间函数执行完，实际上setTimeout就是异步的实现。代码的执行流程如下：

首先执行输出字符串"start"，然后开始执行setTimeout函数。由于它是一个异步操作，所以它会被分为两部分来执行，先调用setTimeout方法，然后把要执行的函数放到一个队列中。代码继续往下执行，当把所有的代码都执行完后，放到队列中的函数才会被执行。这样，所有异步执行的函数都不会阻塞其他代码的执行。虽然，这些代码都不是同时执行，但是由于任何代码都不会被阻塞，所以执行效率会很快。



大家认真看这个图片，然后思考一个问题：当setTimeout执行后，什么时候开始计时的呢？由于单线程的原因，不可能在setTimeout后就开始执行，因为一个线程同一时间只能做一件事情。执行后续代码的同时就不可能又去计时。那么只可能是在所有代码执行完后才开始计时，然后5秒后执行队列中的回调函数，是这样吗？我们用一段代码来验证下：

```
console.log("start");
setTimeout(function(){
    console.log("timeout");
},5000);
for(let i = 0;i <= 500000;i++){
    console.log("i:",i);
}
console.log("end");
```

这段代码在之前的基础上加了一个循环，循环次数为50万次，然后每次输出i的值。这段循环是比较耗时的，从实际运行来看，大概需要14秒左右（具体时间可自行测算）。这个时间已经远远大于setTimeout的等待时间。按照之前的说法，应该先把所有同步的代码执行完，然后再执行异步的回调方法，结果应该是：

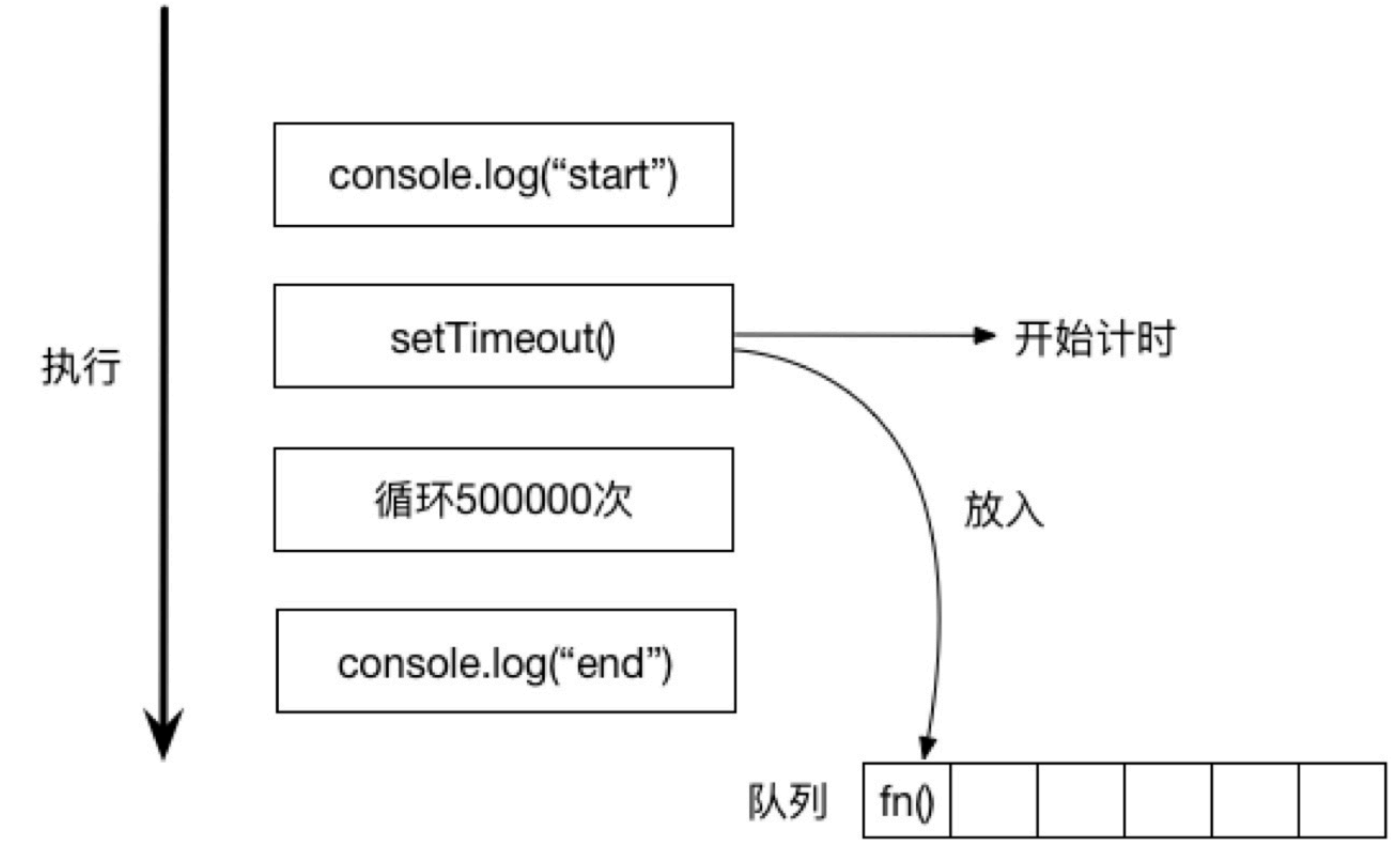
```
start
i:1
(...) //一直输出到500000
//耗时14秒左右
end
//等待5秒后
timeout
```

但实际的运行结果是：

```
start
i:1
(...) //一直输出到500000
//耗时14秒左右
end
//没有等待
timeout
```

从结果可以看到setTimeout的计时应该是早就开始了，但是JS是单线程运行，那谁在计时呢？要解释这个问题，大家一定要先搞明白一件事。JS的单线程并不是指整个JS引擎只有1个线程。它是指运行代码只有1个线程，但是它还有其他线程来执行其他任务。比如时间函数的计时、AJAX技术中的和后台交互等操作。所以，实际情况应该是：JS引擎中执行代码的线程开始运行代码，

当执行到异步方法时，把异步的回调方法放入到队列中，然后由专门计时的线程开始计时。代码线程继续运行。如果计时的时间已到，那么它会通知代码线程来执行队列中对应的回调函数。当然，前提是代码线程已经把同步代码执行完后。否则需要继续等待，就像这个例子中一样。



最后，大家一定要注意一件事情，由于执行代码只有1个线程，所以在任何同步代码中出现死循环，那么它后续的同步代码以及异步的回调函数都无法执行，比如：

```
console.log("start");
setTimeout(function(){
    console.log("timeout");
},5000);
console.log("end");
for(;;){}
```

timeout用于也不会输出，因为执行代码的线程已经陷入死循环中。

16-2 Promise实现异步

前面一讲中我们了解了什么是异步，以及JS中实现异步的原理。这一节咱们将学习JS中实现异步的具体方法。前面我们已经看到了一个用setTimeout实现的异步操作：

```
console.log("start");
setTimeout(function(){
    console.log("timeout");
},5000);
console.log("end");
```

16-2-1 回调函数

在调用setTimeout函数时我们传递了一个函数进去，这个函数并没有立即被调用，而是在5秒后被调用。这种函数也被称为回调函数（关于回调函数请参看前面的内容）。由于JS中的函数是一等公民，它和其他数据类型一样，可以作为参数传递也可以作为返回值返回，所以经常能够看到回调函数使用。

回调地狱

在异步实现中，回调函数的使用是不可避免的。之前我不是讲过吗，JS的异步是单线程非阻塞式的。它将一个异步动作分为两步，第一步执行异步方法，然后代码接着往下执行。然后在后面的某个时刻调用第二步的回调函数，完成后续动作。有的时候，我们希望在异步操作中加入同步的行为。比如，我想打印4句话，但是每句话都在前一句话的基础上延迟2秒输出。代码如下：

```
setTimeout(function(){
    console.log("first");
    setTimeout(function(){
        console.log("second");
        setTimeout(function(){
            console.log("third");
            setTimeout(function(){
                console.log("fourth");
            },2000);
        },2000);
    },2000);
},2000);
```

这段代码能够实现想要的功能，但是总觉得哪里不对。如果输出的内容越来越多，嵌套的代码也会增多。那无论是编写还是阅读起来都会很恐怖。造成这种情况的罪魁祸首就是回调函数。因为

你想在前面的异步操作完成后再进行接下来的动作，那只能在它的回调函数中进行，这样就会越套越多，代码越来越来复杂，俗称"回调地狱"。

16-2-2 Promise

为了解决这个问题，在ES6中加入了一个新的对象Promise。Promise提供了一种更合理、更强大的异步解决方案。接下来我们来看看它的用法。

```
new Promise(function(resolve,reject){
    //dosomething
});
```

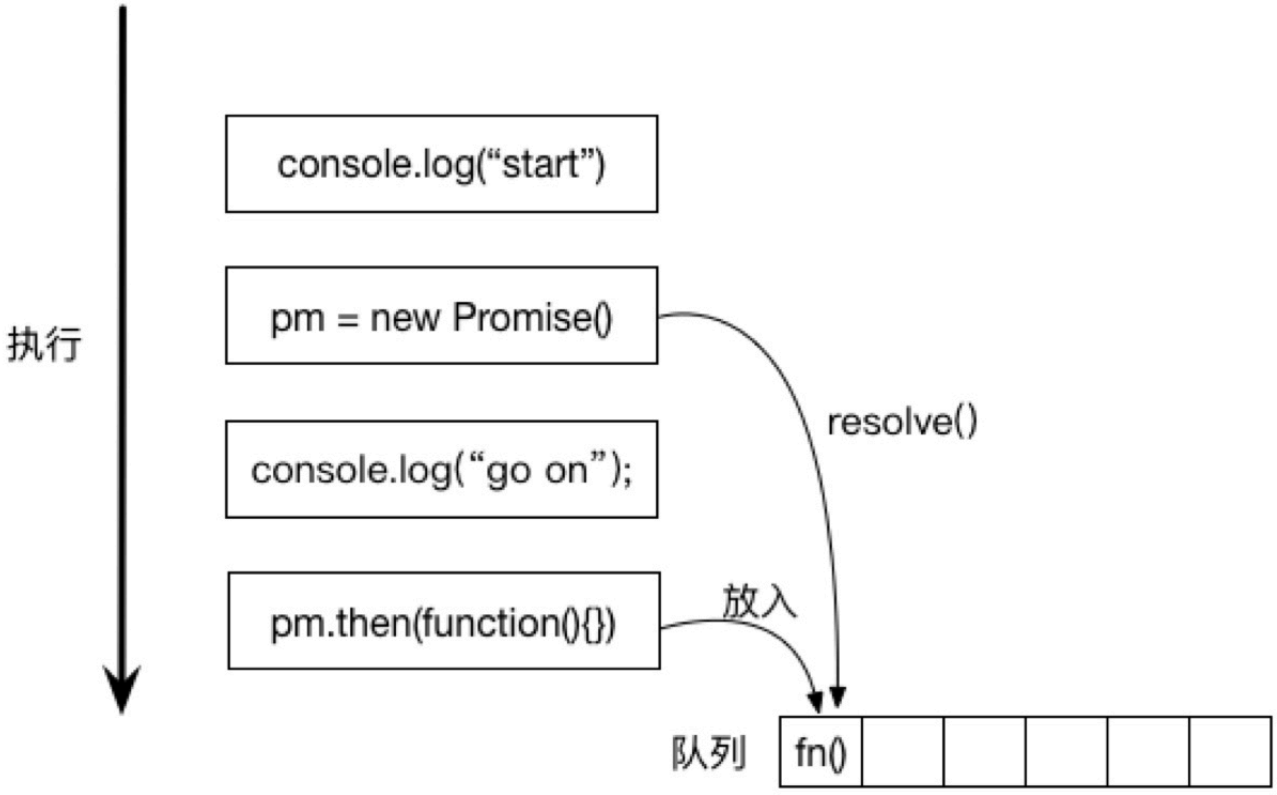
首先需要创建一个Promise对象，该对象的构造函数中接收一个回调函数，回调函数中可以接收两个参数，resolve和reject。注意，这个回调函数是在Promise创建后就会调用。它实际上就是异步操作的第一步。那第二步操作再在哪里做呢？Promise把两个步骤分开了，第二步通过Promise对象的then方法实现。

```
let pm = new Promise(function(resolve,reject){
    //dosomething
});
console.log("go on");
pm.then(function(){
    console.log("异步完成");
});
```

不过要注意的是，then方法的回调函数不是说只要then方法一调用它就会调用，而是在Promise的回调函数中通过调用resolve触发的。

```
let pm = new Promise(function(resolve,reject){
    resolve();
});
console.log("go on");
pm.then(function(){
    console.log("异步完成");
});
```

实际上Promise实现异步的原理和之前纯用回调函数的原理是一样的。只是Promise的做法是显示的将两个步骤分开来写。**then方法的回调函数同样会先放入队列中，等待所有的同步方法执行完后，同时Promise中的resolve也被调用后，该回调函数才会执行。**



调用resolve时还可以把数据传递给then的回调函数。

```
let pm = new Promise(function(resolve,reject){
    resolve("this is data");

});
console.log("go on");
pm.then(function(data){
    console.log("异步完成",data);
});
```

效果：

```
Jie-Xie:desktop Jie$ node 1
go on
异步完成 this is data
```

reject是出现错误时调用的方法。它触发的不是then中的回调函数，而是catch中的回调函数。比如：

```
let err = false;
let pm = new Promise(function(resolve,reject){
    if(!err){
        resolve("this is data");
```

```
    }else{
        reject("fail");
    }

});
console.log("go on");
pm.then(function(data){
    console.log("异步完成",data);
});
pm.catch(function(err){
    console.log("出现错误",err);
});
```

下面，我把刚才时间函数的异步操作用Promise实现一次。当然，其中setTimeout还是需要使用，只是在它外面包裹一个Promise对象。

```
let pm = new Promise(function(resolve,reject){
    setTimeout(function(){
        resolve();
    },2000);

});
console.log("go on");
pm.then(function(){
    console.log("异步完成");
});
```

效果和之前一样，但是代码复杂了不少，感觉有点多此一举。接下来做做同步效果。

```
let timeout = function(time){
    return new Promise(function(resolve,reject){
        setTimeout(function(){
            resolve();
        },time);
    });
}
console.log("go on");
timeout(2000).then(function(){
    console.log("first");
    return timeout(2000);
}).then(function(){
    console.log("second");
    return timeout(2000);
}).then(function(){
    console.log("third");
```

```
    return timeout(2000);
}).then(function(){
    console.log("fourth");
    return timeout(2000);
});
```

由于需要多次创建Promise对象，所以用了timeout函数将它封装起来，每次调用它都会返回一个新的Promise对象。当then方法调用后，其内部的回调函数默认会将当前的Promise对象返回。当然也可以手动返回一个新的Promise对象。我们这里就手动返回了一个新的计时对象，因为需要重新开始计时。后面继续用then方法来触发异步完成的回调函数。这样就可以做到同步的效果，从而避免了过多的回调嵌套带来的"回调地狱"问题。

实际上Promise的应用还是比较多，比如前面讲到的fetch，它就利用了Promise来实现AJAX的异步操作：

```
let pm = fetch("/users"); // 获取Promise对象
pm.then((response) => response.text()).then(text => {
    test.innerText = text; // 将获取到的文本写入到页面上
})
.catch(error => console.log("出错了"));
```

> 注意：response.text()返回的不是文本，而是Promise对象。所以后面又跟了一个then，然后从新的Promise对象中获取文本内容。

Promise作为ES6提供的一种新的异步编程解决方案，但是它也有问题。比如，代码并没有因为新方法的出现而减少，反而变得更加复杂，同时理解难度也加大。因此它并不是异步实现的最终形态，后续我们还会继续介绍其他的异步实现方法。

16-3 迭代器与生成器

上一节中我们学习了如何使用Promise来实现异步操作。但是它也会存在一些问题，比如代码量增多，不易理解。那么这一节咱们将一起来探索其他解决异步的方法。生成器作为ES6新增加的语法，它也能够处理异步的操作。不过再讲生成器之前，咱们还得理解另外一个东西：迭代器。

16-3-1 迭代器(Iterator)

迭代器是一种接口，也可以说是一种规范。它提供了一种统一的遍历数据的方法。我们都知道数组、集合、对象都有自己的循环遍历方法。比如数组的循环：

```
let ary = [1,2,3,4,5,6,7,8,9,10];

//for循环
for(let i = 0;i < ary.length;i++){
    console.log(ary[i]);
}

//forEach循环
ary.forEach(function(ele){
    console.log(ele);
});

//for-in循环
for(let i in ary){
    console.log(ary[i]);
}

//for-of循环
for(let ele of ary){
    console.log(ele);
}
```

集合的循环：

```
let list = new Set([1,2,3,4,5,6,7,8,9,10]);
for(let ele of list){
    console.log(ele);
}
```

对象的循环：

```
let obj = {
    name : 'tom',
    age : 25,
    gender : '男',
    intro : function(){
        console.log('my name is '+this.name);
    }
}

for(let attr in obj){
    console.log(attr);
}
```

从以上的代码可以看到，数组可以用for、forEach、for-in以及for-of来遍历。集合能用for-of。对象能用for-in。也就是说，以上数据类型的遍历方式都各有不同，那么有没有统一的方式遍历这些数据呢？这就是迭代器存在的意义。它可以提供统一的遍历数据的方式，只要在想要遍历的数据结构中添加一个支持迭代器的属性即可。这个属性写法是这样的：

```
const obj = {
    [Symbol.iterator]:function(){}
}
```

`[Symbol.iterator]`属性名是固定的写法，只要是拥有该属性的对象，就能够用迭代器的方式进行遍历。

迭代器的遍历方法是首先获得一个迭代器的指针，初始时该指针指向第一条数据之前。接着通过调用next方法，改变指针的指向，让其指向下一条数据。每一次的next都会返回一个对象，该对象有两个属性。其中value代表想要获取的数据，done是个布尔值，false表示当前指针指向的数据有值。true表示遍历已经结束。

```
let ary = [1,2,3];
let it = ary[Symbol.iterator](); // 获取数组中的迭代器
console.log(it.next()); // { value: 1, done: false }
console.log(it.next()); // { value: 2, done: false }
console.log(it.next()); // { value: 3, done: false }
console.log(it.next()); // { value: undefined, done: true }
```

数组是支持迭代器遍历的，所以可以直接获取其中的迭代器。集合也是一样。

```
let list = new Set([1,2,3]);
let it = list.entries(); // 获取set集合中的迭代器
```

```
console.log(it.next()); // { value: [ 1, 1 ], done: false }
console.log(it.next()); // { value: [ 2, 2 ], done: false }
console.log(it.next()); // { value: [ 3, 3 ], done: false }
console.log(it.next()); // { value: undefined, done: true }
```

> set集合中每次遍历出来的值是一个数组，里面的第一和第二个元素都是一样的。

由于数组和集合都支持迭代器，所以它们都可以用同一种方式来遍历。es6中提供了一种新的循环方法叫做for-of。它实际上就是使用迭代器来进行遍历，换句话说只有支持了迭代器的数据结构才能使用for-of循环。在JS中，默认支持迭代器的结构有：

- Array
- Map
- Set
- String
- TypedArray
- 函数的 arguments 对象
- NodeList 对象

这里面并没有包含自定义的对象，所以当我们创建一个自定义对象后，是无法通过for-of来循环遍历它。除非将iterator接口加入到该对象中：

```
let obj = {
    name: 'xiejie',
    age: 18,
    gender: '男',
    intro: function () {
        console.log('my name is ' + this.name);
    },
    [Symbol.iterator]: function () {
        let i = 0;
        let keys = Object.keys(this); // 获取当前对象的所有属性并形成一个数组
        return {
            next: function () {
                return {
                    value: keys[i++], // 外部每次执行next都能得到数组中的第i个元素
                    done: i > keys.length // 如果数组的数据已经遍历完则返回true
                }
            }
        }
    }
}
for (let attr of obj) {
    console.log(attr);
```

```
    // name
    // age
    // gender
    // intro
}
let it = obj[Symbol.iterator]();
console.log(it.next()); // { value: 'name', done: false }
console.log(it.next()); // { value: 'age', done: false }
console.log(it.next()); // { value: 'gender', done: false }
console.log(it.next()); // { value: 'intro', done: false }
console.log(it.next()); // { value: undefined, done: true }
```

通过自定义迭代器就能让自定义对象使用`for-of`循环。迭代器的概念及使用方法我们清楚了，接下来就是生成器。

16-3-2 生成器(Generator)

生成器也是ES6新增加的一种特性。它的写法和函数非常相似，只是在声明时多了一个`*`号。

```
function* say(){}
const say = function*(){}
```

> 注意：这个`*`只能写在function关键字的后面。

生成器函数和普通函数并不只是一个`*`号的区别。普通函数在调用后，必然开始执行该函数，直到函数执行完或遇到return为止。中途是不可能暂停的。但是生成器函数则不一样，它可以通过yield关键字将函数的执行挂起，或者理解成暂停。它的外部在通过调用next方法，让函数继续执行，直到遇到下一个yield，或函数执行完毕。

```
function* say(){
    yield "开始";
    yield "执行中";
    yield "结束";
}
let it = say(); // 调用say方法，得到一个迭代器
console.log(it.next()); // { value: '开始', done: false }
console.log(it.next()); // { value: '执行中', done: false }
console.log(it.next()); // { value: '结束', done: false }
console.log(it.next()); // { value: undefined, done: true }
```

调用say函数，这句和普通函数的调用没什么区别。但是此时say函数并没有执行，而是返回了一个该生成器的迭代器对象。接下来就和之前一样，执行next方法，say函数执行，当遇到yield

时，函数被挂起，并返回一个对象。对象中包含value属性，它的值是yield后面跟着的数据。并且done的值为false。再次执行next，函数又被激活，并继续往下执行，直到遇到下一个yield。当所有的yield都执行完了，再次调用next时得到的value就是undefined，done的值为true。

如果你能理解刚才讲的迭代器，那么此时的生成器也就很好理解了。它的yield，其实就是next方法执行后挂起的地方，并得到你返回的数据。那么这个生成器有什么用呢？它的yield关键字可以将执行的代码挂起，外部通过next方法让它继续运行。

这和异步操作的原理非常类似，把一个操作分为两部分，先执行一部分，然后再执行另外一部分。所以生成器可以处理和异步相关的操作。我们知道，异步操作主要是依靠回调函数实现。但是纯回调函数的方式去处理同步效果会带来“回调地域“的问题。Promise可以解决这个问题。但是Promise写起来代码比较复杂，不易理解。而生成器又提供了一种解决方案。看下面这个例子：

```
function* delay() {
    yield new Promise((resolve, reject) => { setTimeout(() => { resolve() },
 2000) })
    console.log("go on");
}
let it = delay(); // 得到一个迭代器
// it.next()会执行到第一个 yield 得到的值为{ value: Promise { <pending> }, done:
 false }
// it.next().value 将会得到一个 Promise
// Promise会在2秒以后调用then方法
// 2秒后调用then方法执行迭代器的下一步
it.next().value.then(() => {
    it.next();
});
```

这个例子实现了等待2秒钟后，打印字符串"go on"。下面我们来分析下这段代码。在delay这个生成器中，yield后面跟了一个Promise对象。这样，当外部调用next时就能得到这个Promise对象。然后调用它的then函数，等待2秒钟后Promise中会调用resolve方法，接着then中的回调函数被调用。也就是说，此时指定的等待时间已到。然后在then的回调函数中继续调用生成器的next方法，那么生成器中的代码就会继续往下执行，最后输出字符串"go on"。

例子中时间函数外面为什么要包裹一个Promise对象呢？这是因为时间函数本身就是一个异步方法，给它包裹一个Promise对象后，外部就可以通过then方法来处理异步操作完成后的动作。这样，在生成器中，就可以像写同步代码一样来实现异步操作。比如，利用fetch来获取远程服务器的数据（为了测试方便，我将用MockJS来拦截请求）。

```
<body>
    <script src="./jquery-1.12.4.min.js"></script>
```

```
    <script src="./mock-min.js"></script>
    <script>
        // 拦截Ajax请求
        Mock.mock(/\.json/, {
            'stuents|5-10': [{
                'id|+1': 1,
                'name': '@cname',
                'gender': /[男女]/, //在正则表达式匹配的范围内随机
                'age|15-30': 1, //年龄在15-30之间生成，值1只是用来确定数据类型
                'phone': /1\d{10}/,
                'addr': '@county(true)', //随机生成中国的一个省、市、县数据
                'date': "@date('yyyy-MM-dd')"
            }]
        });
        function* getUsers() {
            let data = yield new Promise((resolve, reject) => {
                $.ajax({
                    type: "get",
                    url: "/users.json",
                    success: function (data) {
                        resolve(data)
                    }
                });
            });
            console.log("得到的data为:", data);
        }
        let it = getUsers(); // 返回一个迭代器
        // it.next().value 会得到一个Promise，Promise里面向服务器发送请求获取数据
        // 数据获取成功以后调用then方法，并将获取到的数据传递给then方法
        // then方法里面再次开启迭代器，执行第二句代码，并将数据传递过去
        // 在getUsers函数里面data变量接收了传递过来的数据，并打印出来
        it.next().value.then((data) => {
            it.next(data);
        });
    </script>
</body>
```

在Promise中调用JQuery的AJAX方法，当数据返回后调用resolve，触发外部then方法的回调函数，将数据返回给外部。外部的then方法接收到data数据后，再次调用next，移动生成器的指针，并将data数据传递给生成器。所以，在生成器中你可以看到，我声明了一个data变量来接收异步操作返回的数据，这里的代码就像同步操作一样，但实际上它是个异步操作。当异步的数据返回后，才会执行后面的打印操作。这里的关键代码就是yield后面一定是一个Promise对象，因为只有这样外部才能调用then方法来等待异步处理的结果，然后再继续做接下来的操作。

之前我们还讲过一个替代AJAX的方法fetch，它本身就是用Promise的方法来实现异步，所以代码写起来会更简单：

```
function* getUsers(){
    let response = yield fetch("/users");
    let data = yield response.json();
    console.log("data",data);
}
let it = getUsers();
it.next().value.then((response) => {
    it.next(response).value.then((data) => {
        it.next(data);
    });
});
```

> 由于mock无法拦截fetch请求，所以我用nodejs+express搭建了一个mock-server服务器。

这里的生成器我用了两次yield，这是因为fetch是一个异步操作，获得了响应信息后再次调用json方法来得到其中返回的JSON数据。这个方法也是个异步操作。

从以上几个例子可以看出，如果单看生成器的代码，异步操作可以完全做的像同步代码一样，比起之前的回调和Promise都要简单许多。但是，生成器的外部还是需要做很多事情，比如需要频繁调用next，如果要做同步效果依然需要嵌套回调函数，代码依然很复杂。市面也有很多的插件可以辅助我们来执行生成器，比如比较常见的co模块。它的使用很简单：

```
co(getUsers);
```

引入co模块后，将生成器传入它的方法中，这样它就能自动执行生成器了。关于co模块这里我就不再多讲，有兴趣的话可以参考这篇文章：http://es6.ruanyifeng.com/#docs/generator-async

16-4 async和await

上一节咱们学习了如何利用生成器实现异步操作。在生成器中，利用yield将异步操作挂起，外部通过执行器让生成器的代码继续执行。这样，在生成器中，可以将异步的操作做成同步的效果，实现了异步代码的简化。不过，这种方式需要编写外部的执行器，而执行器的代码写起来一点也不简单。当然也可以使用一些插件，比如co模块来简化执行器的编写。

在ES7中，加入了async函数来处理异步。它实际上只是生成器的一种语法糖而已，简化了外部执行器的代码，同时利用await替代yield，async替代生成器的`*`号。下面还是来看个例子:

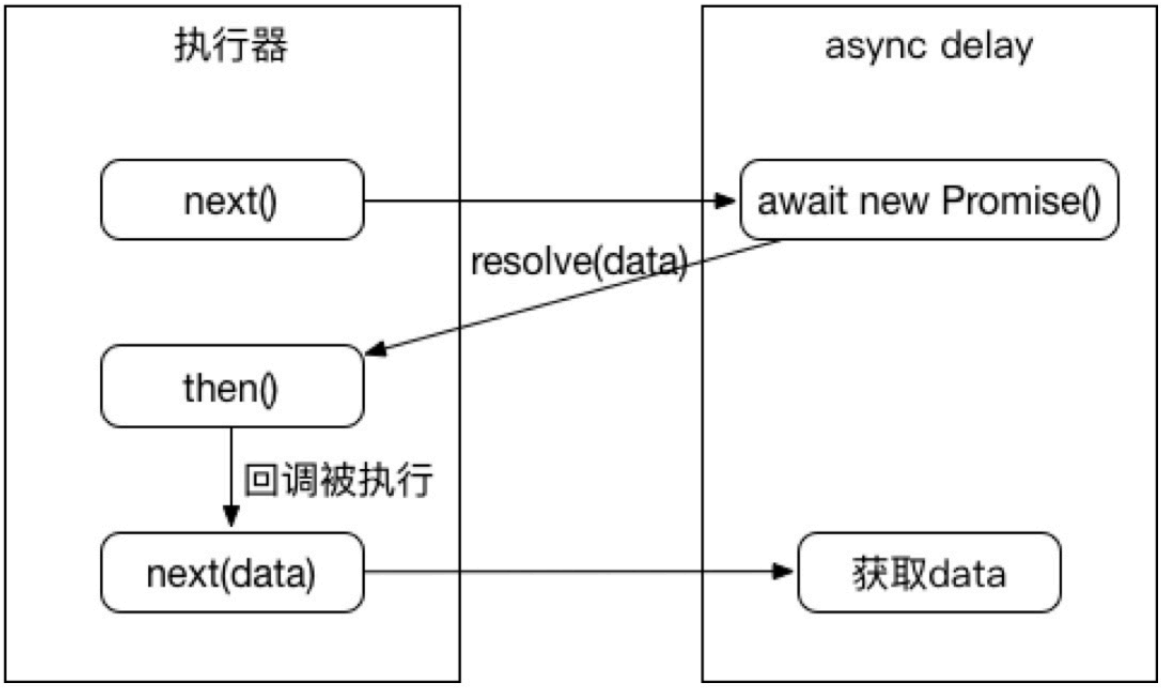
```
async function delay(){
    await new Promise((resolve) => {setTimeout(()=>{resolve()},2000)});
    console.log("go on);
}
delay();
```

这个例子我们之前用生成器也写过，其中把生成器的(*)号被换成了async。async关键字必须写在function的前面。如果是箭头函数，则写在参数的前面：

```
const delay = async () => {}
```

在函数中，第一句用了await。它替代了之前的yield。后面同样需要跟上一个Promise对象。接下来的打印语句会在上面的异步操作完成后执行。外部调用时就和正常的函数调用一样，但它的实现原理和生成器是类似的。因为有了async关键字，所以它的外部一定会有相应的执行器来执行它，并在异步操作完成后执行回调函数。只不过这一切都被隐藏起来了，由JS引擎帮助我们完成。我们需要做的就是加上关键字，在函数中使用await来执行异步操作。这样，可以大大的简化异步操作。同时，能够像同步方法一样去处理它们。

接下来我们再来看看更细节的一些问题。await后面必须是一个Promise对象，这个很好理解。因为该Promise对象会返回给外部的执行器，并在异步动作完成后执行resolve，这样外部就可以通过回调函数处理它，并将结果传递给生成器。



那如果await后面跟的不是Promise对象又会发生什么呢?

```
const delay = async () => {
    let data = await "hello";
    console.log(data);
}
```

这样的代码是允许的，不过await会自动将hello字符串包装一个Promise对象。就像这样：

```
let data = await new Promise((resolve,reject) => resolve("hello"));
```

创建了Promise对象后，立即执行resolve，并将字符串hello传递给外部的执行器。外部执行器的回调函数再将这个hello传递回来，并赋值给data变量。所以，执行该代码后，马上就会输出字符串hello。虽然代码能够这样写，但是await在这里的意义并不大，所以await还是应该用来处理异步方法，同时该异步方法应该使用Promise对象。

async函数里面除了有await关键字外，感觉和其他函数没什么区别，那它能有返回值吗？答案是肯定的，

```
const delay = async () => {
    await new Promise((resolve) => {setTimeout(()=>{resolve()},2000)});
    return "finish";
}
let result = delay();
console.log(result);
```

在delay函数中先执行等待2秒的异步操作，然后返回字符串finish。外部调用时我用一个变量接收它的返回值。最后输出的结果是：

```
// 没有任何等待立即输出
Promise { <pending> }
// 2秒后程序结束
```

我们可以看到，没有任何等待立即输出了一个Promise对象。而整个程序是在2秒钟后才结束的。由此看出，获取async函数的返回结果实际上是return出来的一个Promise对象。假如return后面跟着的本来就是一个Promise对象，那么它会直接返回。但如果不是，则会像await一样包裹一个Promise对象返回。所以，想要得到返回的具体内容应该这样：

```
const delay = async () => {
    await new Promise((resolve) => {setTimeout(()=>{resolve()},2000)});
    return "finish";
}
let result = delay();
console.log(result);
result.then(function(data){
    console.log("data:",data);
});
```

执行的结果：

```
// 没有任何等待立即输出
Promise { <pending> }
//等待2秒后输出
data: finish
```

那如果函数没有任何返回值，得到的又是什么呢？我将上面代码中取掉return，再次运行：

```
// 没有任何等待立即输出
Promise { <pending> }
//等待2秒后输出
data: undefined
```

可以看到，仍然可以得到Promise对象，但由于函数没有返回值，所以就不会有任何数据传递出来，那么打印的结果就是undefined。

最后我们还是来梳理一下async的执行顺序。大致的顺序为：先执行同步代码，然后通过执行器来执行async里面的每一句代码，如果有返回值，在外部要通过`then()`方法的回调函数来接

收，最后才被执行，示例如下：

```
const delay = async () => {
    console.log('first');
    let data = await new Promise((resolve,reject) => resolve("hello"));
    console.log('aa');
    console.log(data);
    let data2 = await new Promise((resolve) => {setTimeout(()=>{resolve('Yes
')},2000)});
    console.log(data2);
    return 'World';
}
let result = delay();
console.log(result);
result.then(function(data){
    console.log("data:",data);
});
console.log(11);
console.log(22);

// first
// Promise { <pending> }
// 11
// 22
// aa
// hello
// Yes
// data: World
```

效果：首先执行async函数，打印出first，然后是暂停里面的代码，返回一个promise，来到外部。在外部执行完所有同步的代码，输出 `Promise { <pending> }` ， `11` 和 `22` 。接下来回到async函数，输出 `aa` 和 `hello` ，然后等两秒钟后，输出 `Yes` ，最后回到外部，执行 `then()` 方法，打印出 `data: World` 。
async的基本原理我们清楚了，下面我们把之前的AJAX例子用async重写下：

```
Mock.mock(/\.json/,{
    'stuents|5-10' : [{
        'id|+1' : 1,
        'name' : '@cname',
        'gender' : /[男女]/, //在正则表达式匹配的范围内随机
        'age|15-30' : 1, //年龄在15-30之间生成，值1只是用来确定数据类型
        'phone' : /1\d{10}/,
        'addr' : '@county(true)', //随机生成中国的一个省、市、县数据
        'date' : "@date('yyyy-MM-dd')"
    }]
```

```
});
async function getUsers(){
    let data = await new Promise((resolve,reject) => {
        $.ajax({
            type:"get",
            url:"/users.json",
            success:function(data){
                resolve(data)
            }
        });
    });
    console.log("data",data);
}
getUsers();
```

这是用JQuery的AJAX方法实现。

```
async function getUsers(){
    let response = await fetch("/users");
    let data = await response.json();
    console.log("data",data);
}
getUsers();
```

这是fetch方法的实现。

从这两个例子可以看出，async和生成器两种方式都很类似，但async可以不借助任何的第三方模块，也更易于理解，async表示该函数要做异步处理。await表示后面的代码是一个异步操作，等待该异步操作完成后再执行后面的动作。如果异步操作有返回的数据，则在左边用一个变量来接收它。

我们知道，await可以让异步操作变为同步的效果。但是，有的时候为了提高效率，我们需要让多个异步操作同时进行怎么办呢？方法就是执行异步方法时不加await，这样它们就可以同时进行，然后在获取结果时用await。比如：

```
function time(ms){
    return new Promise((resolve,reject) => {
        setTimeout(()=>{resolve()},ms);
    });
}
const delay = async () => {
    let t1 = time(2000);
    let t2 = time(2000);
    await t1;
```

```
    console.log("t1 finish");
    await t2;
    console.log("t2 finish");
}
delay();
```

我先把时间函数的异步操作封装成了函数，并返回Promise对象。在delay函数中调用了两次time方法，但没有用await。也就是说这两个时间函数的执行是"同时"（其实还是有先后顺序）进行的。然后将它们的Promise对象分别用t1和t2表示。先用await t1。表示等待t1的异步处理完成，然后输出t1 finish。接着再用await t2，等待t2的异步处理完成，最后输出t2 finish。由于这两个时间函数是同时执行，而且它们的等待时间也是一样的。所以，当2秒过后，它们都会执行相应的回调函数。运行的结果就是：等待2秒后，先输出t1 finish，紧接着立即输出 t2 finish。

```
const delay = async () => {
    await time(2000);
    console.log("t1 finish");
    await time(2000);;
    console.log("t2 finish");
}
```

如果是这样写，那么执行的结果会是等待2秒后输出t1 finish。再等待2秒后输出t2 finish。async确实是一个既好用、又简单的异步处理方法。但是它的问题就是不兼容老的浏览器，只有支持了ES7的浏览器才能使用它。
最后，还需要注意一个问题：await关键字必须写在async定义的函数中。