迭代器详解

迭代器是什么，为什么要有迭代器？

在接触迭代器之前，回想一下我们生活中的例子。我们在乘火车或者客车的乘务员或者售票员做的事情，我们拿客车售票员来举例，对于坐满乘客或者要发车的客车，售票员要进行售票，**他会从后到前，依次左左右右的进行售票，当售票出现问题（上错车，没有钱等问题），售票员会暂时处理，务必保证每一位都要买票，售完之后进行发车。**刚才我们所说的售票员售票的过程就是一个生活中遍历的过程。售票员需要一个不差的监管的每一位乘客，务必买票，问题的核心在于如何保证按照一定规则，一定顺序能把乘客一个不落下售完票，其实这样的过程就是遍历。这种遍历的模式对应就是计算机中的设计模式中的迭代器模式。

迭代器模式：提供一种方法顺序访问一个集合的各个元素，而又不暴露该对象的内部表示。

顺便说一嘴，当要去遍历一个集合，不管里面内容是什么都需要遍历的时候，可以考虑迭代器模式。

在我们 JavaScript 中，可遍历的结构还是很多的：

数组的遍历（Array.prototype.forEach, for，for…in）：

```
for(let i=0; i<arr.length; i++) {
    console.log(i)          // 数组中的每一位
    console.log(arr[i]) //  数组中的索引
}

for(let i in arr) {
    console.log(i)         // 数组中的每一位
    console.log(arr[i])//  数组中的每一位
}

arr.forEach(function(item, index) {
    console.log(item)     // 数组中的每一位
    console.log(index)    // 数组中的索引
})
```

对象的遍历(for….in)：

```
for(let prop in obj) {
    console.log(prop)    // key 值
    console.log(obj[prop]) // value 值
}
```

Set 结构的遍历(forEach)：

```
set.forEach(function (value, othervalue, self) {
    console.log(value)          // set 中的值
    console.log(otherValue) // set 中的值
    console.log(self)           // set 对象自身
})
```

Map 结构的遍历(forEach)：

```
map.forEach(function (value, key, self) {
    console.log(value)          // map 中的 value 值
    console.log(key)            // map 中的 key 值
    console.log(self)           // map 对象自身
})
```

对于不同的结构，有不同的遍历方式。意味着，我们需要知道数据结构，我们才能采取特定的遍历方式。我们看一下这样的应用场景：

在服务器端之前采用的返回数组的方式，我们在客户端拿到数组，使用数组的遍历方式进行遍历。由于业务变动，使得数据的结构发生变化，之前返回的是数组，后面返回的是 set 结构。

以上两种数据结构客户端都必须事先知道集合的内部结构，或者提前商讨，访问代码和集合本身是紧耦合，无法将访问逻辑从集合类和客户端代码中分离出来，每一种集合对应一种遍历方法，客户端代码无法复用。

更恐怖的是，如果以后需要把 Array 更换为 Map，则原来的客户端代码必须全部重写。

为解决以上问题，有人提出 Iterator 方法，使用迭代器模式，总是用同一种逻辑来遍历集合：

```
for(Iterator it = c.iterater(); it.hasNext(); ) { ... }
```

奥秘在于客户端自身不维护遍历集合的"指针"，所有的内部状态（如当前元素位置，是否有下一个元素）都由 Iterator 来维护，而这个 Iterator 由集合类通过工厂方法生成，因此，它知道如何遍历整个集合。

客户端从不直接和集合类打交道，它总是控制 Iterator，就可以直接遍历整个集合。

介于这种新的思想，**对于不同的集合，使用相同的逻辑来遍历**。可以很好的解决遍历多种集合方式多样的问题，基于迭代器模式，参照前人经验。在 ECMAScript6 中提出的的迭代器的概念 Iterator。

迭代器是如何迭代数据的？

迭代器协议：
迭代器对象不是新的语法或新的内置对象，而一种协议（ 迭代器协议），所有遵守这个协议的对象，都可以称之为迭代器。也就是说我们上面 ES5 的写法得到的对象遵循迭代器协议，即包含 next，调用 next 返回一个 result{value，done}。
Iterator 的遍历过程是这样的：
（1）创建一个指针对象，指向当前数据结构的起始位置。也就是说，遍历器对象本质上，就是一个指针对象。
（2）第一次调用指针对象的 next 方法，可以将指针指向数据结构的第一个成员。
（3）第二次调用指针对象的 next 方法，指针就指向数据结构的第二个成员。
（4）不断调用指针对象的 next 方法，直到它指向数据结构的结束位置。
每一次调用 next 方法，都会返回数据结构的当前成员的信息。具体来说，就是返回一个包含 value 和 done 两个属性的对象。其中，value 属性是当前成员的值，done 属性是一个布尔值，表示遍历是否结束。
根据以上逻辑，我们利用已有的语法封装一个数组迭代器。

```
function createIterator(items) {
    var i=0
    return {
        next:function () {
            var done = (i > items.length)
            var value = done ? undefined: items[i++]
            return {
                done,
                value
            }
        }
    }
```

```
}
```

```
var iterator = createIterator([1,2,3,4,5])
console.log(iterator.next())    // {value: 1, done: false}
console.log(iterator.next())    // {value: 2, done: false}
console.log(iterator.next())    // {value: 3, done: false}
console.log(iterator.next())    // {value: 4, done: false}
console.log(iterator.next())    // {value: 5, done: false}
console.log(iterator.next())    // {value: undefined, done: true}
```

在上面这段代码中 createIterator 方法返回的对象有一个 next 方法，每次滴啊用是，items 数组的下一个值作为 value 返回，当 i=5 时，done 为 true，value 为 undefined。
以上的内容是我们按照上面的规定进行的封装。接下来看看我们 ES6 中给我们提供的迭代器是什么样子的。

JavaScript 默认迭代器接口有哪些？

ES6 中许多内置类型已经包含了默认的迭代器，只有当默认迭代器满足不了时，才会创建自定义的迭代器，怎么创建迭代器我们在讲生成器的时候会讲给大家。

集合迭代器
ES6 中有三种集合对象：数组、Map 和 Set，这三种类型都拥有默认的迭代器：

entries()：返回一个包含键值对的迭代器；
values()：返回一个包含集合中的值的迭代器；
keys()：返回一个包含集合中的键的迭代器；

调用 entries()迭代器会在每次调用 next()方法返回一个双项数组，此数组代表集合数据项中的键和值：

对于数组来说，第一项是数组索引，第二项是数组元素；
对于 Set 来说，第一项是值，第二项也是值，
对于 Map 来说，第一项是 key，第二项是 value；
values()迭代器能够返回集合中的每一个值；
keys()迭代器能够返回集合中的每一个键；
数组的默认迭代器使用实例：

```
let arr = [1,2,3,4]
let arrEntires = arr.entries()
    arrEntires.next()    //{value: [0, 1], done: false}
let arrKeys = arr.keys()
    arrKeys.next()       //{value: 0, done: false}
let arrValues = arr.values()
    arrValues.next()     //{value: 1, done: false}
```

Set 的默认迭代器使用实例：

```
let set = new Set([1,2,3,4])
let setEntires = set.entries()
    setEntires.next()    //{value: [1, 1], done: false}
let setKeys = set.keys()
    setKeys.next()       //{value: 1, done: false}
let setValues = set.values()
    setValues.next()     //{value: 1, done: false}
```

Map 的默认迭代器使用实例：

```
let map = new Map([[1,2],[3,4]])
let mapEntires = map.entries()
    mapEntires.next()    //{value: [1, 2], done: false}
let mapKeys = map.keys()
    mapKeys.next()       //{value: 1, done: false}
let mapValues = map.values()
    mapValues.next()     //{value: 2, done: false}
```

以上的 entries,keys,values 是默认的关于集合的迭代器方法，用于返回迭代器。但是没有看到对象的默认迭代器，是其他形式么，还是没有，原因是什么，下面接着来看。

ES6 中迭代器底层原理与 for-of？

可迭代对象（entries,keys,values 返回的迭代器对象）是包含 **Symbol.iterator 属性的对象，这个 Symbol.iterator 属性对应着能够返回该对象的迭代器的函数。**在 ES6 中，所有的集合对象（数组、Set 和 Map）以及字符串都是可迭代对象，因此它们都被指定了默认的迭代器。可迭代对象可以与 ES6 中新增的 for-of 循环配合使用。

迭代器解决了 for 循环中追踪索引的问题，而 for-of 循环，则是完全删除追踪集合索引的需要，更能专注于操作集合内容。for-of 循环在循环每次执行时会调用可迭代对象的 next()方法，并将结果对象的 value 值存储在一个变量上，循环过程直到结果对象 done 属性变成 true 为止：

```
let arr = [1,2,3];
for(let num of arr){
    console.log(num);
}
```

输出结果为：1，2，3

for-of 循环首先会调用 arr 数组中 Symbol.iterator 属性对象的函数，就会获取到该数组对应的迭代器，接下来 iterator.next()被调用，迭代器结果对象的 value 属性会被放入到变量 num 中。数组中的数据项会依次存入到变量 num 中，直到迭代器结果对象中的 done 属性变成 true 为止，循环就结束。

访问可迭代对象的默认迭代器

可以使用可迭代对象的 Symbol.iterator 来访问对象上可返回迭代器的函数：

```
let arr = [1,2,3];
//访问默认迭代器
let iterator = arr[Symbol.iterator]();
console.log(iterator.next().value); //1
console.log(iterator.next().value); //2
```

通过 Symbol.iterator 属性获取到该对象的可返回迭代器的函数，然后执行该函数得到对象的可迭代器同样的，可是使用 Symbol.iterator 属性来检查对象是否是可迭代对象。

对于集合对象 数组，Map，Set 的 for…of 遍历的默认迭代器接口。

Array 的默认迭代器接口[Symblo.iterator]是 values;

```
for(let item of [1,2,3]) {
    console.log(item)
}
// [1,2,3]
```

Set 的默认迭代器接口[Symblo.iterator]是 values;

```
for(let item of new Set([1,2,3,4])){
    console.log(item)
}
// [1,2,3,4]
```

Map 的默认迭代器接口[Symblo.iterator]是 entries;

```
for(let item of new Map([[1,2],[3,4]])){
    console.log(item)
}
// [1,2] [3,4]
```

上面的内容仅仅是为我们我们的结构提出了迭代器接口，还有些结构是没有迭代器接口的，那怎么办，我们需要写迭代器，迭代器该怎么写？

生成迭代器的生成器。

生成器（generator ） 是能返回一个迭代器的函数。生成器函数由放在 function 关键字之后的一个星号（ * ） 来表示，并能使用新的 yield 关键字。将星号紧跟在 function 关键字之后，或是在中间留出空格，都是没问题的。例如：

```
function*generator(){

    yield 1;
    yield 2;
    yield 3;
}

let iterator = generator();
console.log(iterator.next().value);//1
console.log(iterator.next().value);//2
```

生成器函数最有意思的地方是它们会在每一个 yield 语句后停止，例如在上面的代码中执行 yield 1 后，该函数不会在继续往下执行。等待下一次调用 next()后，才会继续往下执行 yield 2。
除了使用函数声明的方式创建一个生成器外，还可以使用函数表达式来创建一个生成器。由于生成器就是一个函数，同样可以使用对象字面量的方式，将对象的属性赋值为一个生成器函数。
接下来创建一个可迭代的对象
//创建可迭代对象

```
let obj = {
    items:[],

    *[Symbol.iterator](){
        for(let item of this.items){
            yield item;
        }
    }
}

obj.items.push(1);
obj.items.push(2);

for(let num of obj){
    console.log(num);
}
```

这里面我们创建一个对象，遍历里面的数组，并不是真正的遍历对象，说了这么多迭代器，一直都没有提及我们对象的迭代，下面来看看。
我们知道了 for…of 是为了迭代器服务的，我们还一直没有提及对象。那么定义对象再利用 for…of 遍历会怎么样。

```
let obj = {
   title: 'javascript',
   language: 'cn'
}

for(let item of obj){
   console.log(item)
}
```

我们看到，会报这样的错误：

Uncaught TypeError: obj is not iterable

说 obj 不能被迭代。

对象对我们来说，是键值存储的一种方式，尽管没有 map 那么好，key 只可以是字符串。Map 相对会好一些。有的时候对象也是需要被迭代的，但是**为什么不给对象设置可迭代的默认方法**？

在 ES6 中，对象默认下并不是可迭代对象，表现为其没有[Symbol.iterator]属性。而 for…of…循环，实际上是依次将迭代器(或任何可迭代的对象，如生成器函数)的值赋予指定变量并进行循环的语法，当对象没有默认迭代器的时候，当然不可以进行循环，而通过给对象增加一个默认迭代器，即[Symbol.iterator]属性，就可以实现，如下代码：

```
Object.prototype[Symbol.iterator] = function *keys(){
  for(let n of Object.keys(this)){ //  此处使用 Object.keys 获取可枚举的属性
    yield n
  }
}
```

以上代码确实获得了对象的所有键名，在生成器函数内，我们使用的是 Object.keys 获得所有可枚举的属性值，然而这并不是所有人都期望的，也许小明期望不可枚举的属性值也被遍历，而小新可能连[Symbol.iterator]也希望遍历出来，于是，这里产生了一些分歧，如何遍历有以下几种因素：
总结起来，对象的 property 至少有三个方面的因素：

属性是否可枚举，即其 enumerable 属性描述符 的值。
属性的类型，是字符串类型、还是 Symbol 类型。
属性所属，包含原型，还是仅仅包含实例本身。

鉴于各方意见不一，并且现有的遍历方式可以满足，于是标准组没有将[Symbol.iterator]加入。

迭代器高级功能

能够通过 next()方法向迭代器传递参数**，当一个参数传递给 next()方法时，该参数就会成为生成器内部 yield 语句中的变量值。**

```
//迭代器的高级功能
function * generator(){
    let first = yield 1;
```

```
    let second = yield first+2;
    let third = yield second+3;
}

let iterator = generator();
console.log(iterator.next()); //{value: 1, done: false}
console.log(iterator.next(4)); //{value: 6, done: false}
console.log(iterator.next(5)); //{value: 8, done: false}
console.log(iterator.next()); //{value: undefined, done: true}
```

示例代码中，当通过 next()方法传入参数时，会赋值给 yield 语句中的变量。

在迭代器中抛出错误

能传递给迭代器的不仅是数据，还可以是错误，迭代器可以选择一个 throw()方法，用于指示迭代器应在恢复执行时抛出一个错误：

```
//迭代器抛出错误

function * generator(){
    let first = yield 1;
    let second = yield first+2;
    let third = yield second+3;
}


let iterator = generator();
console.log(iterator.next()); //{value: 1, done: false}
console.log(iterator.next(4)); //{value: 6, done: false}
console.log(iterator.throw(new Error('Error!'))); //Uncaught Error: Error!
console.log(iterator.next()); //不会执行
```

在生成器中同样可以使用 try-catch 来捕捉错误：

```
function * generator(){
    let first = yield 1;
    let second;
    try{
        second = yield first+2;
```

```
    }catch(ex){
        second = 6
    }
    let third = yield second+3;
}


let iterator = generator();
console.log(iterator.next()); //{value: 1, done: false}
console.log(iterator.next(4)); //{value: 6, done: false}
console.log(iterator.throw(new Error('Error!'))); //{value: 9, done: false}
console.log(iterator.next()); //{value: undefined, done: true}
```

生成器的 return 语句

由于生成器是函数，你可以在它内部使用 return 语句，既可以让生成器早一点退出执行，也可以指定在 next() 方法最后一次调用时的返回值。大多数情况，迭代器上的
next() 的最后一次调用都返回了 undefined ，但你还可以像在其他函数中那样，使用
return 来指定另一个返回值。在生成器内， return 表明所有的处理已完成，因此 done
属性会被设为 true ，而如果提供了返回值，就会被用于 value 字段。比如，利用 return 让生成器更早的退出：

```
function * gene(){
    yield 1;
    return;
    yield 2;
    yield 3;
}

let iterator = gene();
console.log(iterator.next());//{value: 1, done: false}
console.log(iterator.next());//{value: undefined, done: true}
console.log(iterator.next());//{value: undefined, done: true}
```

由于使用 return 语句，能够让生成器更早结束，因此在第二次以及第三次调用 next()方法时，返回结果对象为：{value: undefined, done: true}。
还可以使用 return 语句指定最后返回值：

```
function * gene(){
    yield 1;
    return 'finish';
}

let iterator = gene();
console.log(iterator.next());//{value: 1, done: false}
console.log(iterator.next());//{value: "finish", done: true}
console.log(iterator.next());//{value: undefined, done: true}
```

当第二次调用 next()方法时，返回了设置的返回值：finish。第三次调用 next() 返回了一个对象，其 value 属性再次变回 undefined ，你在 return 语句中指定的任意值都只会在结果对象中出现一次，此后 value 字段就会被重置为 undefined 。

生成器委托

生成器委托是指：将生成器组合起来使用，构成一个生成器。组合生成器的语法需要 yield 和*，*落在 yield 关键字与生成器函数名之间即可：

```
function * gene1(){
    yield 'red';
    yield 'green';

}
function * gene2(){
    yield 1;
    yield 2;
}

function * combined(){
    yield * gene1();
    yield * gene2();
}
```

```
let iterator = combined();
console.log(iterator.next());//{value: "red", done: false}
console.log(iterator.next());//{value: "green", done: false}
console.log(iterator.next());//{value: 1, done: false}
console.log(iterator.next());//{value: 2, done: true}
console.log(iterator.next());//{value: undefined, done: true}
```

此例中将生成器 gene1 和 gene2 组合而成生成器 combined，每次调用 combined 的 next()方法时，实际上会委托到具体的生成器中，当 gene1 生成器中所有的 yield 执行完退出之后，才会继续执行 gene2，当 gene2 执行完退出之后，也就意味着 combined 生成器执行结束。
在使用生成器委托组合新的生成器时，前一个执行的生成器返回值可以作为下一个生成器的参数：
//利用生成器返回值

```
function * gene1(){
    yield 1;
    return 2;
}

function * gene2(count){

    for(let i=0;i<count;i++){
        yield 'repeat';
    }
}

function * combined(){
    let result = yield * gene1();
    yield result;
    yield*gene2(result);
}
let iterator = combined();
console.log(iterator.next());//{value: 1, done: false}
console.log(iterator.next());//{value: 2, done: false}
console.log(iterator.next());//{value: "repeat", done: false}
console.log(iterator.next());//{value: "repeat", done: false}
```

```
console.log(iterator.next());//{value: undefined, done: true}
```

此例中，生成器 gene1 的返回值，就作为了生成器 gene2 的参数。

异步任务

一个简单的任务运行器

生成器函数中 yield 能暂停运行，当再次调用 next()方法时才会重新往下运行。一个简单的任务执行器，就需要传入一个生成器函数，然后每一次调用 next()方法就会“一步步”往下执行函数：

```
//任务执行器
function run(taskDef) {
    // 创建迭代器，让它在别处可用
    let task = taskDef();
    // 启动任务
    let result = task.next();
    // 递归使用函数来保持对 next() 的调用
    function step() {
    // 如果还有更多要做的
    if (!result.done) {
        result = task.next();
        step();
    }
    }
    // 开始处理过程
    step();
}


run(function*() {
    console.log(1);
    yield;
    console.log(2);
    yield;
    console.log(3);
});
```

run() 函数接受一个任务定义（即一个生成器函数） 作为参数，它会调用生成器来创建一个
迭代器，并将迭代器存放在 task 变量上。第一次对 next() 的调用启动了迭代器，并将结果存储下来以便稍后使用。step() 函数查看 result.done 是否为 false，如果是就在递归调用自身之前调用 next() 方法。每次调用 next() 都会把返回的结果保
存在 result 变量上，它总是会被最新的信息所重写。对于 step() 的初始调用启动了处理
过程，该过程会查看 result.done 来判断是否还有更多要做的工作。

能够传递数据的任务运行器

如果需要传递数据的话，也很容易，也就是将上一次 yield 的值，传递给下一次 next()调用即可，仅仅只需要传送结果对象的 value 属性：

```
//任务执行器
function run(taskDef) {
    // 创建迭代器，让它在别处可用
    let task = taskDef();
    // 启动任务
    let result = task.next();
    // 递归使用函数来保持对 next() 的调用
    function step() {
        // 如果还有更多要做的
        if (!result.done) {
            result = task.next(result.value);
            console.log(result.value); //6 undefined
            step();
        }
    }
    // 开始处理过程
    step();
}


run(function*() {
    let value = yield 1;
    yield value+5;
```

```
});
```

异步任务

上面的例子是简单的任务处理器，甚至还是同步的。实现任务器也主要是迭代器在每一次调用 next()方法时彼此间传递静态参数。如果要将上面的任务处理器改装成异步任务处理器的话，就需要 yield 能够返回一个能够执行回调函数的函数，并且回调参数为该函数的参数即可。
什么是有回调函数的函数？
有这样的示例代码：

```
function fetchData(callback) {
    return function(callback) {
        callback(null, "Hi!");
    };
}
```

函数 fetchData 返回的是一个函数，并且所返回的函数能够接受一个函数 callback。当执行返回的函数时，实际上是调用回调函数 callback。但目前而言，回调函数 callback 还是同步的，可以改造成异步函数：

```
function fetchData(callback) {
    return function(callback) {
            setTimeout(function() {
                callback(null, "Hi!");
        }, 50);
    };
}
```

一个简单的异步任务处理器：

```
//异步任务处理器

function run(taskDef){

    //执行生成器，创建迭代器
    let task = taskDef();
    //启动任务
    let result = task.next();
```

```
    function step(){
        while(!result.done){
            if(typeof(result.value)==='function' ){
                result.value(()=>{
                    console.log('hello world');
                })
            }
            result = task.next();
            step();
        }
    }
    step();
}



run(function *(){
    //返回一个能够返回执行回调函数的函数，并且回调函数还是该
    //函数的参数
    yield function(callback){
        setTimeout(callback,3000);
    }
});
```

上面的示例代码就是一个简单的异步任务处理器，有这样几点要点：

使用生成器构造迭代器，所以在 run 方法中传入的是生成器函数；
生成器函数中 yield 关键字，返回的是一个能够执行回调函数的函数，并且回调函数是该函数的一个参数；

总结

使用迭代器可以用来遍历集合对象包含的数据，调用迭代器的 next()方法可以返回一个结果对象，其中 value 属性代表值，done 属性用来表示集合对象是否已经到了最后一项，如果集合对象的值全部遍历完后，done 属

性为 true；

Symbol.iterator 属性被用于定义对象的默认迭代器，使用该属性可以为对象自定义迭代器。当 Symbol.iterator 属性存在时，该对象可以被认为是可迭代对象；

可迭代对象可以使用 for-of 循环，for-of 循环不需要关注集合对象的索引，更能专注于对内容的处理；

数组、Set、Map 以及字符串都具有默认的迭代器；

扩展运算符可以作用于任何可迭代对象，让可迭代对象转换成数组，并且扩展运算符可以用于数组字面量中任何位置中，让可迭代对象的数据项一次填入到新数组中；

生成器是一个特殊的函数，语法上使用了*，yield 能够返回结果，并能暂停继续往下执行，直到调用 next()方法后，才能继续往下执行。使用生成器委托能够将两个生成器合并组合成一个生成器；

能够使用生成器构造异步任务处理器；