# 第七章 迭代器与生成器

# 迭代器

##### 在软件开发领域，“迭代” 的意思是按照顺序反复多次执行一段程序，通常会有明确的终止条件。

检查是否存在默认迭代器属性可以暴露迭代器工厂函数  
let num = 1;  
let obj = {};  
// 这两种类型没有实现迭代器工厂函数   
console.log(num[Symbol.iterator]); // undefined   
console.log(obj[Symbol.iterator]); // undefined  
let str = 'abc';  
let arr = ['a', 'b', 'c'];  
let map = new Map().set('a', 1).set('b', 2).set('c', 3);  
let set = new Set().add('a').add('b').add('c');  
let els = document.querySelectorAll('div');  
// 这些类型都实现了迭代器工厂函数   
console.log(str[Symbol.iterator]); // f values() { [native code] }  
console.log(arr[Symbol.iterator]); // f values() { [native code] }  
console.log(map[Symbol.iterator]); // f values() { [native code] }  
console.log(set[Symbol.iterator]); // f values() { [native code] }  
console.log(els[Symbol.iterator]); // f values() { [native code] }  
// 调用这个工厂函数会生成一个迭代器   
console.log(str[Symbol.iterator]()); // StringIterator {}   
console.log(arr[Symbol.iterator]()); // ArrayIterator {}   
console.log(map[Symbol.iterator]()); // MapIterator {}   
console.log(set[Symbol.iterator]()); // SetIterator {}   
console.log(els[Symbol.iterator]()); // ArrayIterator {}

#### 迭代器的基本操作

// 可迭代对象  
let arr = ['foo', 'bar'];  
// 迭代器工厂函数  
console.log(arr[Symbol.iterator]); // f values() { [native code] }  
// 迭代器  
let iter = arr[Symbol.iterator]();  
console.log(iter); // ArrayIterator {}  
// 执行迭代  
console.log(iter.next()); // { done: false, value: 'foo' }   
console.log(iter.next()); // { done: false, value: 'bar' }   
console.log(iter.next()); // { done: true, value: undefined }  
  
//只要迭代器到达 done: true 状态， 3 后续调用 next()就一直返回同样的值了  
console.log(iter.next()); // { done: true, value: undefined }  
console.log(iter.next()); // { done: true, value: undefined }

可迭代对象在迭代期间被修改,则会反映出来

let arr = ['bar','bar2','bar3']  
  
//创建一个引用arr的迭代器iter  
let iter = arr[Symbol.iterator]();  
  
console.log(iter.next()) //{value: "bar", done: false}  
//修改可迭代对象  
arr.splice(1,0,'bar0.1','bar0.2');  
console.log(iter.next()) //{value: "bar0.1", done: false}  
console.log(iter.next()) //{value: "bar0.2", done: false}  
console.log(iter.next()) //{value: "bar2", done: false}  
console.log(iter.next()) //{value: "bar3", done: false}  
console.log(iter.next()) //{value: undefined, done: true}



#### 迭代器维护着一个指向可迭代对象的引用，因此迭代器会阻止垃圾回收程序回收可迭代对象。

内置语言结构在发现还有更多值可以迭代，但不会消费这些值时，会自动调用 return()方法。

实现iterator接口的类Counter

class Counter {  
 constructor(limit) {  
 this.limit = limit;  
 }  
 [Symbol.iterator]() {  
 let count = 1,  
 limit = this.limit;  
 return {  
 next() {  
 if (count <= limit) {  
 return { done: false, value: count++ };  
 } else {  
 return { done: true };  
 }  
 },  
 return() {  
 console.log('Exiting early');  
 return { done: true };  
 }  
 };   
 }  
 }

for of发现迭代器提前结束，调用return()方法

return()方法必须返回一个有效的 IteratorResult 对象。简单情况下，可以只返回{ done: true }。

let counter1 = new Counter(5);  
 for (let i of counter1) {  
 if (i > 2) {  
 break; }  
 console.log(i);  
 }  
   
// 1  
// 2  
// Exiting early   
  
  
  
//for of发现迭代器提前结束，调用return()方法  
let counter2 = new Counter(5);  
try {  
 for (let i of counter2) {  
if (i > 2) {  
throw 'err';  
}  
 console.log(i);  
 }  
} catch(e) {}  
// 1  
// 2  
// Exiting early  
  
  
//解构时发现迭代器提前结束，调用return()方法  
let counter3 = new Counter(5);  
let [a, b] = counter3;  
// Exiting early

因为 return()方法是可选的，所以并非所有迭代器都是可关闭的。

如果迭代器没有关闭，则还可以继续从上次离开的地方继续迭代。比如，数组的迭代器就是不能关 闭的:

let a = [1, 2, 3, 4, 5];  
let iter = a[Symbol.iterator]();  
for (let i of iter) {  
 console.log(i);  
 if (i > 2) {  
 break }  
}  
 // 1 // 2 // 3  
 for (let i of iter) {  
 console.log(i);}  
 // 4 // 5

# 生成器

调用生成器函数会产生一个生成器对象。生成器对象一开始处于暂停执行(suspended)的状态。与 迭代器相似，生成器对象也实现了 Iterator 接口，因此具有 next()方法。调用这个方法会让生成器 开始或恢复执行。

next()方法的返回值类似于迭代器，有一个 done 属性和一个 value 属性。函数体为空的生成器 函数中间不会停留，调用一次 next()就会让生成器到达 done: true 状态。

function\* generatorFn() {}  
let generatorObject = generatorFn();   
console.log(generatorObject); // generatorFn {<suspended>}  
console.log(generatorObject.next()); // { done: true, value: undefined }

生成器函数只会在初次调用 next()方法后开始执行

function\* generatorFn() {  
 console.log('foobar');  
 }  
 // 初次调用生成器函数并不会打印日志  
 let generatorObject = generatorFn();  
// 初次调用 next()方法后开始执行  
generatorObject.next(); // foobar

生成器对象实现了 Iterable 接口

function\* generatorFn() {}  
 const g = generatorFn();  
 console.log(g === g[Symbol.iterator]());  
 // true 生成器默认的迭代器是自引用的

#### 通过**yield**中断执行

生成器函数在遇到 yield 关键字之前会正常执行。遇到这个关键字后，执行会停止，函数作用域的状态会被保留。停止执行的生 成器函数只能通过在生成器对象上调用 next()方法来恢复执行

function\* generatorFn() {  
 yield;  
 }  
   
let generatorObject = generatorFn();  
console.log(generatorObject.next());  
console.log(generatorObject.next());  
// { done: false, value: undefined }  
// { done: true, value: undefined }

此时的 yield 关键字有点像函数的中间返回语句，它生成的值会出现在 next()方法返回的对象里。 通过 yield 关键字退出的生成器函数会处在 done: false 状态;通过 return 关键字退出的生成器函 数会处于 done: true 状态。

function\* generatorFn() {  
yield 'foo';  
yield 'bar';  
return 'baz';  
}  
let generatorObject = generatorFn();  
console.log(generatorObject.next());   
console.log(generatorObject.next());  
console.log(generatorObject.next());  
  
// { done: false, value: 'foo' }   
// { done: false, value: 'bar' }  
// { done: true, value: 'baz' }

yield 关键字只能在生成器函数内部使用，用在其他地方会抛出错误。

#### 生成器对象作为可迭代对象

function\* generatorFn() {  
 yield 1;  
 yield 2;  
 yield 3;   
 }  
  
for (const x of generatorFn()) {   
console.log(x);  
}  
 // 1   
 // 2   
 // 3

定义一个生成器函数，它会根据配置的值迭代相应次数并产生迭代的索引。

function\* nTimes(n) {  
 for (let i = 0; i < n; ++i) {  
 yield i; }  
}  
for (let x of nTimes(3)) {  
 console.log(x);  
 }  
 // 0  
 // 1   
 // 2

生成器实现 range

function\* range(start, end) {  
 while(end > start) {  
 yield start++;  
 }  
}  
for (const x of range(4, 7)) {  
 console.log(x);  
}  
// 4 // 5 // 6

生成器实现填充数组zeros

function\* zeroes(n) {  
 while(n--) {  
 yield 0;   
 }  
 }  
console.log(Array.from(zeroes(8))); // [0, 0, 0, 0, 0, 0, 0, 0]

#### 提前关闭生成器

return()和 throw()方法都可以用于强制生成器进入关闭状态。