迭代器与生成器

理解迭代

循环是迭代的基础
正常for循环不能解决两个问题:

- 迭代之前需要事先知道和使用数据结构
- 遍历顺序并不是数据结构固有的。（索引访问是数组特有的，并不适合其他隐式顺序的数据结构）

forEach不能表示迭代终止

无需实现知道如何迭代就实现迭代操作----->迭代器模式

迭代器模式

把有些结构称为"可迭代对象"，实现了正式的Iterable接口。而且可以通过迭代器Iterator消费

可迭代协议

实现Iterable接口（可迭代协议）要求具备两种能力：

- 支持迭代的自我识别能力
- 创建实现Iterator接口的对象的能力

意味着必须暴露一个属性"默认迭代器"。必须用Symbol.iterator为key。默认迭代器属性必须引用一个迭代器工厂函数，必须返回一个新迭代器。

很多内置类型实现了Iterable接口

- 字符串
- 数组
- 映射
- 集合
- arguments对象
- NodeList等DOM集合类型

我们可以检查是否存在默认迭代器属性可以暴露这个工厂函数

```
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 {}
```

实际代码不需要显式调用这个工厂函数。实现可迭代协议的所有类型都会自动兼容接收可迭代对象的任何语言特性。
可迭代对象的原生语言特性：

- for-of循环
- 数组结构
- 拓展操作符
- Array.from
- 创建集合
- 创建映射
- Promise.all() 接收由期约组成的可迭代对象
- Promise.race() 接收由期约组成的可迭代对象
- yield* 操作符，生成器中使用

这些原生结构都会自动调用这个工厂函数，从而创建迭代器

迭代器协议

迭代器是一种一次性使用的对象，用于迭代与关联的可迭代对象
迭代器API，next()方法，成功则返回IteratorResult 对象，包含迭代器返回的下一个值；不调用next()则无法知道当前位置

IteratorResult对象属性:

- done 布尔值，表示是否还可以在此调用next()，获取下一个值
- value 包含可迭代对象的下一个值

done：true状态称为"耗尽"

```
// 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 }
```

到达done：true状态后，调用next()就一直返回相同的值了

```
let arr = ['foo'];
let iter = arr[Symbol.iterator]();
console.log(iter.next()); // { done: false, value: 'foo' }
console.log(iter.next()); // { done: true, value: undefined }
console.log(iter.next()); // { done: true, value: undefined }
console.log(iter.next()); // { done: true, value: undefined }
```

每个迭代器都表示对可迭代对象的一次性有序遍历。不同迭代器的实例相互之间没有联系，只会独立地遍历可迭代对象

迭代器不与可迭代对象的 某一时刻快照 绑定，而是用游标来记录可迭代对象的历程。如果可迭代对象在迭代期间修改了，迭代器也会反映相应变化

比较显式的迭代器实现和原生的迭代器

```
// 这个类实现了可迭代接口（Iterable）
// 调用默认的迭代器工厂函数会返回
// 一个实现迭代器接口（Iterator）的迭代器对象
class Foo {
 [Symbol.iterator]() {
 return {
 next() {
 return { done: false, value: 'foo' };
 }
 }
 }
}
let f = new Foo();
// 打印出实现了迭代器接口的对象
console.log(f[Symbol.iterator]()); // { next: f() {} }
// Array 类型实现了可迭代接口（Iterable）
// 调用 Array 类型的默认迭代器工厂函数
// 会创建一个 ArrayIterator 的实例
let a = new Array();
// 打印出 ArrayIterator 的实例
console.log(a[Symbol.iterator]()); // Array Iterator {}
```

自定义迭代器

与Iterable接口类似，任何实现Iterator接口的对象都可以作为迭代器使用

```
class Counter {
 // Counter 的实例应该迭代 limit 次
 constructor(limit) {
 this.count = 1;
```

```
 this.limit = limit;
 }
 next() {
 if (this.count <= this.limit) {
 return { done: false, value: this.count++ };
 } else {
 return { done: true, value: undefined };
 }
 }
 [Symbol.iterator]() {
 return this;
 }
}
let counter = new Counter(3);
for (let i of counter) {
 console.log(i);
}
// 1
// 2
// 3
```

上述代码每个实例只能迭代一次。可以使用闭包

```
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, value: undefined };
 }
 }
 };
 }
}
let counter = new Counter(3);
for (let i of counter) { console.log(i); }
// 1
// 2
// 3
for (let i of counter) { console.log(i); }
// 1
// 2
// 3
```

for-of自动调用Symbol.iterator工厂函数!

每个迭代器也实现了Iterable接口

```
let arr = ['foo', 'bar', 'baz'];
let iter1 = arr[Symbol.iterator]();
```

```
console.log(iter1[Symbol.iterator]); // f values() { [native code] }
let iter2 = iter1[Symbol.iterator]();
console.log(iter1 === iter2); // true
```

提前终止迭代器

可选的return()方法，指定迭代器提前关闭时执行的逻辑

- for-of通过break，continue，return，throw提前退出
- 结构操作并未消费所有值

```
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 };
 }
 };
 }
}
let counter1 = new Counter(5);
for (let i of counter1) {
 if (i > 2) {
 break;
 }
 console.log(i);
}
// 1
// 2
// Exiting early
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
let counter3 = new Counter(5);
let [a, b] = counter3;
```

```
// Exiting early
```

如果迭代器没有关闭，还可以继续从上次迭代离开的地方继续迭代。数组的迭代器无法关闭

```
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
```

因为 return()方法是可选的，所以并非所有迭代器都是可关闭的。
可以通过测试迭代器实例的return属性是不是函数，来测试能否关闭迭代器（并不准确）

```
let a = [1, 2, 3, 4, 5];
let iter = a[Symbol.iterator]();
iter.return = function() {
 console.log('Exiting early');
 return { done: true };
 };
for (let i of iter) {
 console.log(i);
 if (i > 2) {
 break
 }
}
// 1
// 2
// 3
// 提前退出
for (let i of iter) {
 console.log(i);
}
// 4
// 5
```