# for of 循环的原理

for...of 是 ES6 引入用来遍历所有数据结构的统一方法。

**这里的所有数据结构只指具有 iterator 接口的数据。一个数据只要部署了 Symbol.iterator，就具有了 iterator 接口，就可以使用 for...of 循环遍历它的成员。也就是说，for...of 循环内部调用的数据结构为 Symbol.iterator 方法。**

for...of 循环可以使用的范围包括数组、Set 和 Map 结构、某些类似数组的对象（比如 arguments 对象、DOM NodeList 对象）、 Generator 对象，以及字符串。也就是说上面提到的这些数据类型原生就具备了 iterator 接口。

所以千万不要错误地认为 for...of 只是用来遍历数组的。

## Iterator

### 为什么引入 Iterator

为什么会有 会引入 Iterator 呢，是因为 ES6 添加了 Map, Set，再加上原有的数组，对象，一共就是 4 种表示 “集合”的数据结构。没有 Map 和 Set 之前，我们都知道 for...in 一般是常用来遍历对象，for 循环

常用来遍历数据，现在引入的 Map, Set，难道还要单独为他们引入适合用来遍历各自的方法么。聪明的你肯定能想到，我们能不能提供一个方法来遍历所有的数据结构呢，这个方法能遍历所有的数据结构，一定是这些数据结构要有一些通用的一些特征，然后这个公共的方法会根据这些通用的特征去进行遍历。

Iterator 就可以理解为是上面我们所说的通用的特征。

我们来看看官方对 Iterator 是怎么解释的：遍历器（Iterator）就是这样一种机制。它是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署 Iterator 接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。通俗点理解就是为了解决不同数据结构遍历的问题，引入了 Iterator.

## Iterator 是什么样子的呢

我们来模拟实现以下：

```
function makeIterator(array) {
  var nextIndex = 0;
  return {
    next: function() {
      return nextIndex < array.length ?
        {
            value: array[nextIndex++],
            done: false
        }
        :
        {
            value: undefined,
            done: true
        };
    }
  };
}
const it = makeIterator(['a', 'b']);

it.next()
// { value: "a", done: false }
it.next()
// { value: "b", done: false }
it.next()
// { value: undefined, done: true }
```

简单解释一下上面 array[nextIndex++]是什么意思，

假如 nextIndex 当前为 0，则 nextIndex++的意思为 1.返回 0 2. 值自增（nextIndex 现在为 1）。之前遇到一道面试题就是考察 i++和 ++i

好了，接着来看 Iterator 的整个的遍历过程：

1. 创建一个指针对象（上面代码中的 it），指向当前数据的起始位置
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（上面代码中的 a）。
3. 第二次调用指针对象的 next 方法，可以将指针指向数据结构的第二个成员（上面代码中的 b）。
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,即是否要有必要再一次调用。

Iterator 的特点

- 各种数据结构，提供一个统一的、简便的访问接口
- 使得数据结构的成员能够按某种次序排列
- ES6 创造了一种新的遍历命令 for...of 循环，Iterator 接口主要供 for...of 消费

## 默认 Iterator 接口

部署在 Symbol.iterator 属性，或者说，一个数据结构只要具

有 Symbol.iterator 属性，就认为是"可遍历的"。

原生具备 Iterator 接口的数据结构如下。

- Array
- Map
- Set
- String：字符串是一个类似数组的对象，也原生具有 Iterator 接口。

- 函数的 arguments 对象
- NodeList 对象

除了原生具备 Iterator 接口的数据之外，其他数据结构（主要是对象）的 Iterator 接口，都需要自己在 Symbol.iterator 属性上面部署，这样才会被 for...of 循环遍历。

**对象（Object）之所以没有默认部署 Iterator 接口，是因为对象的哪个属性先遍历，哪个属性后遍历是不确定的，需要开发者手动指定。本质上，遍历器是一种线性处理，对于任何非线性的数据结构，部署遍历器接口，就等于部署一种线性转换。不过，严格地说，对象部署遍历器接口并不是很必要，因为这时对象实际上被当作 Map 结构使用，ES5 没有 Map 结构，而 ES6 原生提供了。**

一个对象如果要具备可被 for...of 循环调用的 Iterator 接口，就必须在 Symbol.iterator 的属性上部署遍历器生成方法（原型链上的对象具有该方法也可）。

```
class RangeIterator {
  constructor(start, stop) {
    this.value = start;
    this.stop = stop;
  }

  [Symbol.iterator]() { return this; }

  next() {
    let value = this.value;
    if (value < this.stop) {
      this.value++;
      return {
        done: false,
        value: value
      };
    }
    return {
      done: true,
```

```
      value: undefined
    };
  }
}

function range(start, stop) {
  return new RangeIterator(start, stop);
}

for (let value of range(0, 3)) {
  console.log(value); // 0, 1, 2
}
```

如果 Symbol.iterator 方法对应的不是遍历器生成函数（即会返回一个遍历器对象），解释引擎将会报错。

```
const obj = {};

obj[Symbol.iterator] = () => 1;

// TypeError: Result of the Symbol.iterator method is not an object
console.log([...obj] )
```

字符串是一个类似数组的对象，也原生具有 Iterator 接口。

```
const someString = "hi";
typeof someString[Symbol.iterator]
// "function"
```

## 调用 Iterator 的场景

除了 for...of，还有下面几个场景

- 解构赋值：对数组和 Set 结构进行解构赋值时，会默认调用 Symbol.iterator 方法。
- 扩展运算符：扩展运算符内部就调用 Iterator 接口。
- yield*：yield*后面跟的是一个可遍历的结构，它会调用该结构的遍历器接口。
- 接受数组作为参数的场合
    - Array.from()

- `Map()`, `Set()`, `WeakMap()`, `WeakSet()`（比如 `new Map([['a',1],['b',2]])`）
- `Promise.all()`
- `Promise.race()`

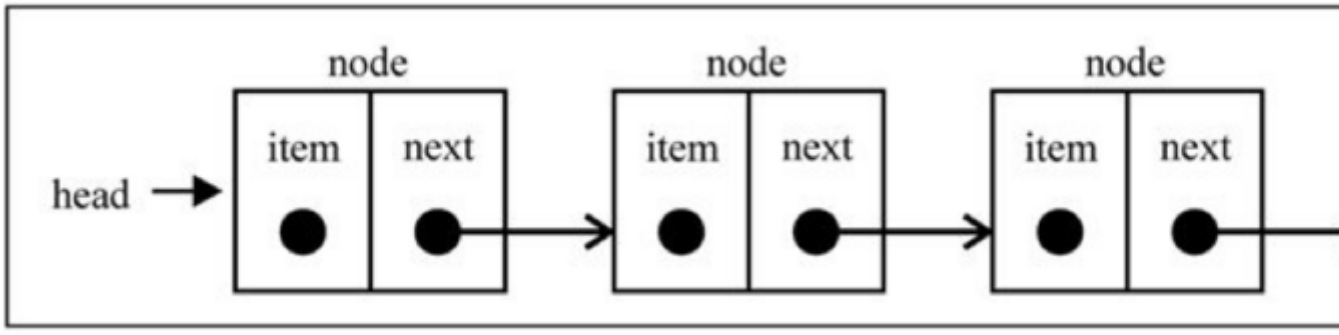
## Iterator 的实现思想

看到 `next` 这个你有没有感到很熟悉，链表中 每个元素由一个存储元素本身的节点和一个指向下一个元素的引用（即 next 属性）组成。是不是很类似，不错，`Iterator` 的实现思想就是来源于单向链表。

下面来简单介绍一下单向链表。

## 单向链表

链表存储有序的元素集合，但不同于数组，链表中每个元素在内存中并不是连续放置的。每个元素由一个存储元素本身的节点和一个指向下一个元素的节点（也称为指针或链接）组成，下图展示了一个链表的结构。




和数组相比较，链表的一个好处已在于，添加或移除元素的时候不需要移动其他元素。然而，链表需要指针，因此实现链表时需要额外注意。数组的另一个细节是可以直接访问任何位置的任何元素，而**想要访问链表中间的一个元素，需要从起点（表头）开始迭代列表知道找到所有元素**。

现实生活中也有一些链表的例子，比如说寻宝游戏。你有一条线索，这条线索是指向寻找下一条线索的地点的指针。你顺着这条链接去到下一个地点，得到另一条指向再下一处的线索，得到列表中间的线索的唯一办法，就是从起点（第一条线索）顺着列表寻找。

具体怎么实现一个单向链表，这里就不展开讲了。

## for...of 循环

关于 for...of 的原理，相信你看完上面的内容已经掌握的差不多了，现在我们以数组为例，说一下，for...of 和之前我们经常使用的其他循环方式有什么不同。

最原始的写法就是 for 循环。

```
for (let i = 0; i < myArray.length; index++) {
  console.log(myArray[i]);
}
```

这种写法比较麻烦，因此数组提供内置的 forEach 方法。

```
myArray.forEach((value) => {
  console.log(value);
});
```

这种写法的问题在于，无法中途跳出 forEach 循环，break 命令或 return 命令都不能奏效。

for...in 循环可以遍历数组的键名。

```
const arr = ['red', 'green', 'blue'];
for(let v in arr) {
    console.log(v); // '0', '1', '2
}
```

for...in 循环有几个缺点:

- 数组的键名是数字，但是 for...in 循环是以字符串作为键名“0”、“1”、“2”等等。
- for...in 循环不仅遍历数字键名，还会遍历手动添加的其他键，甚至包括原型链上的键
- 某些情况下，for...in 循环会以任意顺序遍历键名。

for...in 循环主要是为遍历对象而设计的，不适用于遍历数组。

for...of 和上面几种做法（for 循环，forEach, for...in）相比，有一些显著的优点

- 有着同 for...in 一样的简洁语法，但是没有 for...in 那些缺点。

- 不同于 forEach 方法，它可以与 break、continue 和 return 配合使用。
- 提供了遍历所有数据结构的统一操作接口。

# 总结

- for...of 可以用来遍历所有具有 iterator 接口的数据结构。（一个数据结构只要部署了 Symbol.iterator 属性，就被视为具有 iterator 接口）。也就是说 for...of 循环内部调用是数据结构的 Symbol.iterator
- forEach 循环中无法用 break 命令或 return 命令终止。而 for...of 可以。
- for...in 遍历数组遍历的是键名，所有适合遍历对象，for...of 遍历数组遍历的是键值。