> 迭代器模式提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露该对象的内部表示。
> ——《设计模式：可复用面向对象软件的基础》

迭代器模式是设计模式中少有的**目的性极强的模式**。所谓“目的性极强”就是说它不操心别的，它就解决这一个问题——遍历。

“公元前”的迭代器模式

遍历作为一种合理、高频的使用需求，几乎没有语言会要求它的开发者手动去实现。在JS中，本身也内置了一个比较简陋的数组迭代器的实现——Array.prototype.forEach。

通过调用forEach方法，我们可以轻松地遍历一个数组：

```
const arr = [1, 2, 3]
arr.forEach((item, index)=>{
    console.log(`索引为${index}的元素是${item}`)
})
```

但forEach方法并不是万能的，比如下面这种场景：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>事件代理</title>
</head>
<body>
    <a href="#">链接1号</a>
    <a href="#">链接2号</a>
    <a href="#">链接3号</a>
    <a href="#">链接4号</a>
    <a href="#">链接5号</a>
    <a href="#">链接6号</a>
</body>
</html>
```

我想拿到所有的a标签，我可以这样做：

```
const aNodes = document.getElementsByTagName('a')
console.log('aNodes are', aNodes)
```

我想取其中一个a标签，可以这样做：

```
const aNode = aNodes[i]
```

在这个操作的映衬下，aNodes看上去多么像一个数组啊！但当你尝试用数组的原型方法去遍历它时：

```
aNodes.forEach((aNode, index){
    console.log(aNode, index)
})
```

你发现报错了：

```
> aNodes.forEach((aNode, index) => {
      console.log(aNode, index)
  })
▶Uncaught TypeError: aNodes.forEach is not a function
      at <anonymous>:1:8
```

震惊，原来这个aNodes是个假数组！准确地说，它是一个**类数组**

现在问题就出现了：普通数组是不是集合？是！aNodes是不是集合？是！同样是集合，同样有遍历需求，我们却要针对不同的数据结构执行不同的遍历手段，好累！再回头看看迭代器的定义是什么——遍历集合的同时，我们**不需要关心集合的内部结构**。而forEach只能做到允许我们不关心数组这一种集合的内部结构，看来想要一套统一的遍历方案，我们非得请出一个**更强的通用迭代器**不可了。

这个小节的标题定语里有三个字“公元前”，这个“公元前”怎么定义呢？其实它说的就是ES标准内置迭代器之前的那些日子——差不多四五年之前，彼时还没有这么多轮子，jQuery风头正盛。当时面试可不问什么Vue原理、React原理、Webpack这些，当时问的最多的是**你读过jQuery源码吗**？答读过，好，那咱们就有的聊了。答没有？fine，看来你只是个调包侠，回见吧——因为前端的技术点在那时还很有限，所以可考察的东西也就这么点，读jQuery源码的程序员和不读jQuery源码的程序员在面试官眼里有着质的区别。但这也从一个侧面反映出来，jQuery这个库其实是非常优秀的，至少jQuery里有太多优秀的设计模式可以拿来考考你。就包括咱们当年想用一个真·迭代器又不想自己搞的时候，也是请jQuery实现的迭代器来帮忙：

首先我们要在页面里引入jQuery：

```
  <script src="https://cdn.bootcss.com/jquery/3.3.0/jquery.min.js"
type="text/javascript"></script>
```

借助jQuery的each方法，我们可以用同一套遍历规则遍历不同的集合对象：

```
const arr = [1, 2, 3]
    const aNodes = document.getElementsByTagName('a')

    $.each(arr, function (index, item) {
      console.log(`数组的第${index}个元素是${item}`)
    })

    $.each(aNodes, function (index, aNode) {
      console.log(`DOM类数组的第${index}个元素是${aNode.innerText}`)
    })
```

输出结果完全没问题：

当然啦，遍历jQuery自己的集合对象也不在话下：

```
const jQNodes = $('a')
$.each(jQNodes, function (index, aNode) {
      console.log(`jQuery集合的第${index}个元素是${aNode.innerText}`)
    })
```

输出结果仍然没问题：

可以看出，jQuery的迭代器为我们统一了不同类型集合的遍历方式，使我们在访问集合内每一个成员时不用去关心集合本身的内部结构以及集合与集合间的差异，这就是迭代器存在的价值~

ES6对迭代器的实现

在“公元前”，JS原生的集合类型数据结构，只有Array（数组）和Object（对象）；而ES6中，又新增了Map和Set。四种数据结构各自有着自己特别的内部实现，但我们仍期待以同样的一套规则去遍历它们，所以ES6在推出新数据结构的同时也推出了一套**统一的接口机制**——迭代器（Iterator）。

ES6约定，任何数据结构只要具备Symbol.iterator属性（这个属性就是Iterator的具体实现，它本质上是当前数据结构默认的迭代器生成函数），就可以被遍历——准确地说，是被for...of...循环和迭代器的next方法遍历。 事实上，for...of...的背后正是对next方法的反复调用。

在ES6中，针对Array、Map、Set、String、TypedArray、函数的 arguments 对象、NodeList 对象这些原生的数据结构都可以通过for...of...进行遍历。原理都是一样的，此处我们拿最简单的数组进行举例，当我们用for...of...遍历数组时：

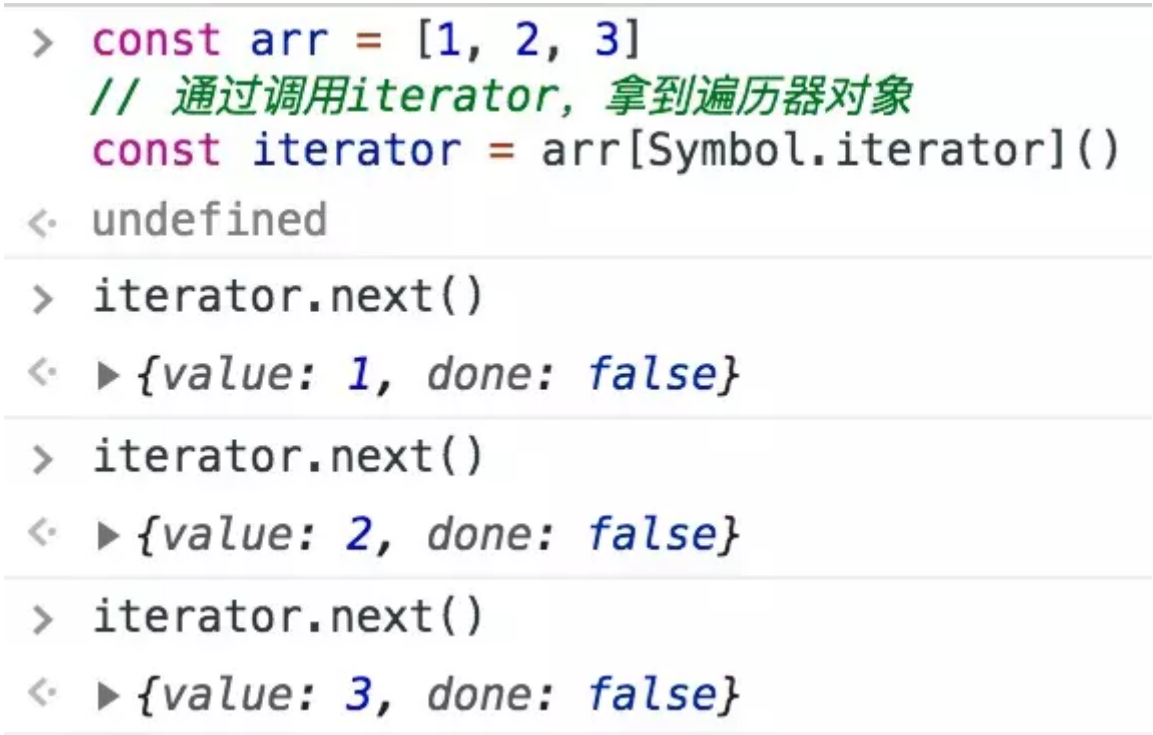
```
const arr = [1, 2, 3]
const len = arr.length
for(item of arr) {
    console.log(`当前元素是${item}`)
}
```

之所以能够按顺序一次一次地拿到数组里的每一个成员，是因为我们借助数组的Symbol.iterator生成了它对应的迭代器对象，通过反复调用迭代器对象的next方法访问了数组成员，像这样：

```
const arr = [1, 2, 3]
// 通过调用iterator，拿到迭代器对象
const iterator = arr[Symbol.iterator]()

// 对迭代器对象执行next，就能逐个访问集合的成员
iterator.next()
iterator.next()
iterator.next()
```

丢进控制台，我们可以看到next每次会按顺序帮我们访问一个集合成员：



而for...of...做的事情，基本等价于下面这通操作：

```
// 通过调用iterator，拿到迭代器对象
const iterator = arr[Symbol.iterator]()

// 初始化一个迭代结果
let now = { done: false }

// 循环往外迭代成员
while(!now.done) {
    now = iterator.next()
    if(!now.done) {
        console.log(`现在遍历到了${now.value}`)
    }
}
```

可以看出，for...of...其实就是iterator循环调用换了种写法。在ES6中我们之所以能够开心地用for...of...遍历各种各种的集合，全靠迭代器模式在背后给力。

ps：此处推荐阅读**迭代协议**，相信大家读过后会对迭代器在ES6中的实现有更深的理解。

一起实现一个迭代器生成函数吧!

ok，看过了迭代器从古至今的操作，我们一起来实现一个自定义的迭代器。

楼上我们说**迭代器对象**全凭**迭代器生成函数**帮我们生成。在ES6中，实现一个迭代器生成函数并不是什么难事儿，因为ES6早帮我们考虑好了全套的解决方案，内置了贴心的**生成器**（Generator）供我们使用：

> 注：本小册不要求ES6基础，但生成器语法比较简单，推荐不了解的同学阅读阮老师的生成器教学光速入门

```
// 编写一个迭代器生成函数
function *iteratorGenerator() {
    yield '1号选手'
    yield '2号选手'
    yield '3号选手'
}

const iterator = iteratorGenerator()

iterator.next()
iterator.next()
iterator.next()
```

丢进控制台，不负众望：

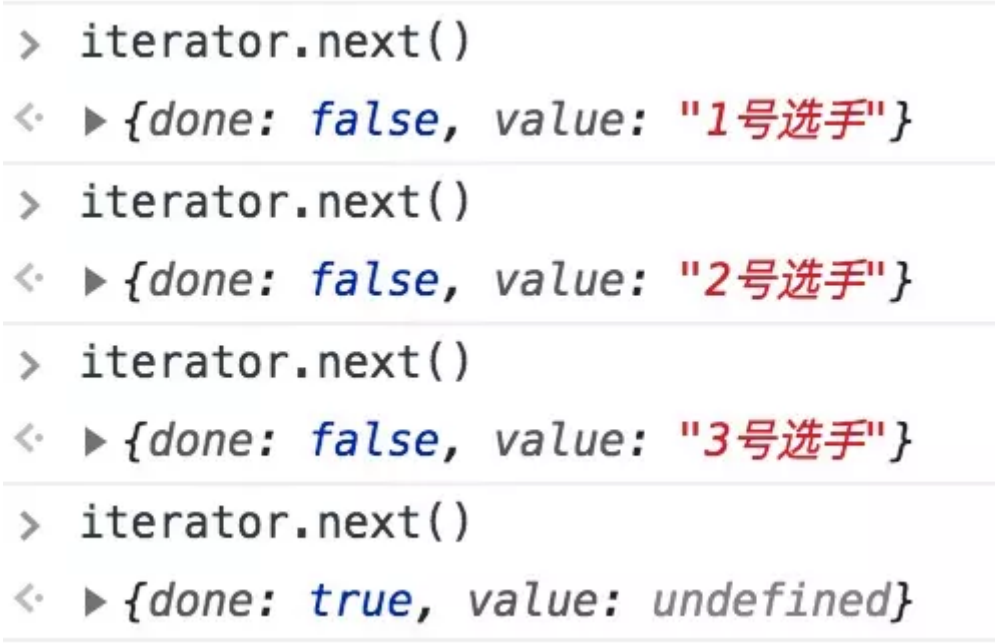
写一个生成器函数并没有什么难度，但在面试的过程中，面试官往往对生成器这种语法糖背后的实现逻辑更感兴趣。下面我们要做的，不仅仅是写一个迭代器对象，而是用ES5去写一个能够生成迭代器对象的迭代器生成函数（解析在注释里）：

```
// 定义生成器函数，入参是任意集合
function iteratorGenerator(list) {
    // idx记录当前访问的索引
    var idx = 0
    // len记录传入集合的长度
    var len = list.length
    return {
        // 自定义next方法
        next: function() {
            // 如果索引还没有超出集合长度，done为false
            var done = idx >= len
            // 如果done为false，则可以继续取值
            var value = !done ? list[idx++] : undefined

            // 将当前值与遍历是否完毕（done）返回
            return {
                done: done,
                value: value
            }
        }
    }
}

var iterator = iteratorGenerator(['1号选手', '2号选手', '3号选手'])
iterator.next()
iterator.next()
iterator.next()
```

此处为了记录每次遍历的位置，我们实现了一个闭包，借助自由变量来做我们的迭代过程中的“游标”。运行一下我们自定义的迭代器，结果符合预期：



小结

迭代器模式比较特别，它非常重要，重要到语言和框架都争着抢着帮我们实现。但也正因为如此，大家业务开发中需要手动写迭代器的场景几乎没有，所以很少有同学会去刻意留意迭代器模式、思考它背后的实现机制。通过阅读本节，希望大家可以领略迭代器模式的妙处（为什么会有，为什么要用）和迭代器模式的实现思路（方便面试）。至此，我们的设计模式之旅就告一段落了~

（阅读过程中有任何想法或疑问，或者单纯希望和笔者交个朋友啥的，欢迎大家添加我的微信xyalinode与我交流哈~）