Array.prototype.keys()

`Array.prototype` 原型上的 `keys()` `values()` `entries()` 方法返回一个可遍历对象，以方便遍历数组对象

使用如下代码：

```
Array.prototype.keysCustom = function () {}
console.log([...([].keysCustom())]);
```

最终报错 `[].keysCustom is not a function or its return value is not iterable`

由以上错误信息我们可以知道， `keys` 必须是一个函数，并且其返回值必须是一个具备可遍历性对象

这里有个很关键的地方，就是要区分开什么是 `iterable` 和 `iterator` 这两个单词，也就是说，必须要明确什么叫 **遍历器方法**，**遍历器对象** 以及 **可遍历性对象** 这三个概念

回顾一下遍历器相关知识：

```
function iterator () {
    let index = 0;
    let _this = this;

    return {
        next() {
            return {
                value: _this[index++],
                done: (index < _this.length ? false : true)
            }
        }
    }
}

let o = {
    0: '111',
    1: '222',
    2: '333',
    length: 3,
    [Symbol.iterator]: iterator
}
```

在上面的代码中： `o` 对象拥有 `[Symbol.iterator]` 方法，所以 `o` 对象是一个 **可遍历性对象**，换句话说， `o` 对象具有遍历器接口，是可以被遍历的

而 `iterator` 方法，我们称之为 **遍历器方法**，也可以称之为 **遍历器对象生成函数**，上面说的遍历器接口也是这个函数，遍历器对象生成函数的目的是为了提供 **遍历器对象**，也就是具备 `next` 方法的一个对象，并且 `next` 方法必须具备 `value` 和 `done` 属性

再回顾一下上面的错误： `keys` 必须是一个函数，并且其返回值必须是一个具备可遍历性对象，由此可得到 `keys` 方法的内部基本结构，如下：

```
Array.prototype.keysCustom = function () {
    // 该方法返回一个对象，该对象必须具有可遍历性，因此该对象必须拥有正确的
[Symbol.iterator] 属性
    return {
        [Symbol.iterator] () {
            return {
                next() {
                    // return ...
                }
            }
        }
    }
}
```

知道了结构以后，我们再去从细节实现，通过 `next` 方法不断的遍历数组的 `key` ，也就是数组下标

```
Array.prototype.keysCustom = function () {
    const _this = this;

    // 该方法返回一个对象，该对象必须具有可遍历性，因此该对象必须拥有正确的
[Symbol.iterator] 属性
    return {
        [Symbol.iterator] () {
            let index = 0;
            return {
                next() {
                    return {
                        value: index,
                        done: (index++ < _this.length ? false : true)
                    }
                }
            }
        }
    }
}
```

到此， `keys` 方法的实现就基本完成了，不过上面的实现方式可以进行一次巧妙而又简单的变形：

```
Array.prototype.keysCustom = function () {
    const _this = this;
    let index = 0;

    // 该方法返回一个对象，该对象必须具有可遍历性，因此该对象必须拥有正确的
[Symbol.iterator] 属性
    // 这种变体的好处在于，这个返回的对象本身具备 [Symbol.iterator]，证明其是一个可遍历性
对象，同时你也会发现，这个对象本身又是遍历器对象
    return {
        next() {
            return {
                value: index,
                done: (index++ < _this.length ? false : true)
            }
        },

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

我们通常会在很多地方使用上面这种技巧，不是必须的，但却是很实用的

如果仔细观察 `console.log([].keys())` 这个语句的结果，你会发现该结果与我们实现的 `console.log([].keysCustom())` 并不相同，但是其本质是完全相同的，只不过原生的 `keys()` 方法在是现实时，经过了几次原型继承，换句话说就是 `next` 和 `[Symbol.iterator]` 被放置在了更深的原型链上面去了，但是实现原理别无二致