3. 循环与迭代

for

> 对于数组, 其实是通过数组的length属性,获取长度后进行遍历, for循环本身跟数组没有关系

for in

1. 如果对象的属性存在,for循环才会执行.
2. 用于遍历对象, 通过对象内对键值对数的数量,进行枚举, 但是如果对象属性不可枚举
    1. Symbol属性不会参与迭代
    2. 迭代不按照编写的属性迭代, [优先迭代数字小→大],再迭代非数字属性
    3. 不仅迭代自己的私有属性, 也会迭代一些扩展的公有属性
3. `for in`的性能明显要弱于其他三种循环, 因为它要迭代原型属性和实例, 除非明确要迭代一个属性未知的对象, 否则应该尽量避免使用`for in`

```
Object.prototype.sum = function sum() {}
let obj = {
  name: 'zhufeng',
  age: 12,
  0: 100,
  1: 200,
  teacher: 'zhouxiaotian',
  [Symbol('AA')]: 300
}
for (let key in obj) {
    if ( !obj.hasOwnProperty(key) ) break;
    console.log(key); //'0' '1' 'name' 'age' 'teacher' 'sum'
}
```

while

for of

continue/break

> continue: 结束本轮循环, 进入下轮 break: 结束整个循环

```
for (let i = 0; i <= 10; i++) {
  if (i >= 3) {
    console.log(i)
    i += 2
    continue // 则此以下的代码将不会执行
  } else {
    console.log(i)
    i += 3
    break
    // 直接打断整个循环
  }
}
console.log(i)
```

iterator

遍历器（Iterator）是一种机制(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「for of循环」，依次处理该数据结构的所有成员

- 拥有next方法用于依次遍历数据结构的成员
  - 每一次遍历返回的结果是一个对象 {done:false,value:xxx}
    - done:记录是否遍历完成
    - value:当前遍历的结果
  - 拥有Symbol.iterator属性的数据结构(值)，被称为可被遍历的，可以基于for of循环处理
    - 数组
    - 部分类数组：arguments/NodeList/HTMLCollection...
    - String
    - Set
    - Map
    - generator object
    - 对象默认不具备Symbol.iterator，属于不可被遍历的数据结构

```
class Iterator {
  constructor(assemble) {
    let self = this
    self.assemble = assemble
    self.index = 0
  }
  next() {
    let self = this,
      assemble = self.assemble
    if (self.index > assemble.length - 1) {
      return {
        done: true,
        value: undefined
      }
    }
    return {
      done: false,
      value: assemble[self.index++]
    }
  }
}
let arr = [10, 20, 30, 40]
let intor = new Iterator(arr)
console.log(intor.next())
console.log(intor.next())
```

```
console.log(intor.next())
console.log(intor.next())

arr[Symbol.iterator] = function () {
  return new Iterator(arr)
}
for (let item of arr) {
  console.log(item)
}
```

generator

```
// 模拟数据请求：执行方法，发送一个数据请求，传递的值是请求的时间，请求成功后的结果也是这个值
const query = interval => {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(interval)
    }, interval)
  })
}

// 需求：我们有三个请求，所用时间分别是1000/2000/3000，而且实现的需要时“串行”「第一个请求成功，再发第二个请求，第二个请求成功，再发第三个请求 ->都成功需要的总
/*
query(1000).then(result => {
  console.log(`第一个请求成功，结果是:${result}`);
  return query(2000);
}).then(result => {
  console.log(`第二个请求成功，结果是:${result}`);
  return query(3000);
}).then(result => {
  console.log(`第三个请求成功，结果是:${result}`);
});
*/

function* generator() {
  let result
  result = yield query(1000)
  console.log(`第一个请求成功，结果是:${result}`)

  result = yield query(2000)
  console.log(`第二个请求成功，结果是:${result}`)

  result = yield query(3000)
  console.log(`第三个请求成功，结果是:${result}`)
}
// 每一次next执行传递的值 都是作为上次yield执行的返回值处理
let itor = generator()
// console.log(itor.next()) //value:promise  done:false
itor.next().value.then(result => {
  itor.next(result).value.then(result => {
    itor.next(result).value.then(result => {
      itor.next(result)
    })
  })
})
```