生成器 | 迭代器

迭代器对象

迭代器是帮助我们对某个数据结构进行遍历的对象(迭代器对象用于遍历数据结构)

迭代器也是一个具体的对象，如果我们想要将一个对象变为迭代器，这个对象需要符合迭代器协议

```
//正确方式
function Iterator(arr) {
  let index = 0
  return {
    next: function () {
      if (index < arr.length) {
        return { done: false, value: arr[index++] }
      } else {
        return { done: true, value: undefined }
      }
    }
  }
}

const arr = [1, 2, 3, 4, 5]
const arrIterator = Iterator(arr)
console.log(arrIterator.next(arr));//{ done: false, value: 1 }
console.log(arrIterator.next(arr));//{ done: false, value: 2 }
console.log(arrIterator.next(arr));//{ done: false, value: 3 }
console.log(arrIterator.next(arr));//{ done: false, value: 4 }
console.log(arrIterator.next(arr));//{ done: false, value: 5 }
console.log(arrIterator.next(arr));//{ done: true, value: undefined
```

每次调用 next() 方法时，它返回一个包含两个属性的对象：
value：当前产生的值
done：布尔值，表示是否所有的值都已经被迭代完毕

可迭代对象

1 、迭代协议分为具体两个协议：可迭代协议、迭代器协议
2 、当一个对象实现了iterable protocol(可迭代)协议时，它就是一个可迭代对象
3 、这个对象的要求是必须实现 @@iterator 方法，在代码中的表达形式为使用 Symbol.iterator 访问该属性

```
//可迭代对象实现
const iterableObj = {
  //需要遍历的目标数组
  names: ["coderwhy", "XiaoYu", "JS"],
  [Symbol.iterator]: function() {
    //索引指针
    let index = 0
    //迭代器
    return {
      next: () => {
        if (index < this.names.length) {
          return { done: false, value: this.names[index++] }
        } else {
          return { done: true, value: undefined }
        }
      }
    }
  }
}
```

调用可迭代对象时返回一个迭代器，该迭代器已经与目标迭代数组完成深层绑定，因此可以直接调用该迭代器的next方法做到对应遍历功能

可迭代协议（Iterable Protocol）： 一个对象要被 for...of 循环使用，必须实现可迭代协议，即具有 [Symbol.iterator] 方法
迭代器协议（Iterator Protocol）： 迭代器对象必须实现 next() 方法，返回 { value, done }

原生迭代器对象

String、Array、TypeArray、Map、Set以及Intl.Segments都是内置的可迭代对象，它们的每个 prototype 对象都实现了 [Symbol.iterator]() 方法

```
class Classroom {
  constructor(address, name, students) {
    this.address = address
    this.name = name
    this.students = students
  }

  entry(newStudent) {
    this.students.push(newStudent)
  }
  //实现迭代器
  [Symbol.iterator]() {
    let index = 0
    return {
      next: () => {
        if (index < this.students.length) {
          return { done: false, value: this.students[index++] }
        } else {
          return { done: true, value: undefined }
        }
      },
      //可以做出其他针对性处理，不过很少见
      return: () => {
        console.log("迭代器提前终止了~")
        return { done: true, value: undefined }
      }
    }
  }
}
```

对象之所以不可遍历，在于对象不是可迭代对象，也没有迭代器。展开语法之所以能够遍历对象，在于展开语法使用的不是迭代器，而是复制属性

自定义对象可迭代性

生成器

ES6（ECMAScript 2015）中引入的一种特殊函数类型，它可以让我们在函数执行过程中暂停和恢复执行，为 JavaScript 提供协程能力

```
function* foo() {
  console.log("函数开始执行~")

  const value1 = 100
  console.log("第一段代码:", value1)
  yield//生成器暂停

  const value2 = 200
  console.log("第二段代码:", value2)
  yield

  const value3 = 300
  console.log("第三段代码:", value3)
  yield

  console.log("函数执行结束~")
}
```

生成器函数内部，可以使用 yield 关键字暂停函数执行，并返回一个值

```
function * foo(rv) {
  console.log('生成器1：', rv);
  const rv1 = yield 'v1';
  // 每一个rv都是上一阶段的内容
  console.log('生成器2：', rv1);
  const rv2 = yield `v2-${rv1}`;

  console.log('生成器3：', rv2);
  const rv3 = yield 'v3';
  console.log('生成器4：', rv3);
  const rv4 = yield 'v4';
  console.log('生成器5：', rv4);
  const rv5 = yield 'v5';
}
/**
 * 生成器1： 输入1
{ value: 'v1', done: false }
生成器2： 输入2
{ value: 'v2-输入2', done: false }
生成器3： 输入3
{ value: 'v3', done: false }
生成器4： 输入4
{ value: 'v4', done: false }
生成器5： 输入5
{ value: 'v5', done: false }
{ value: undefined, done: true }
 */
const generator = foo('输入1');
console.log(generator.next(''));
console.log(generator.next('输入2'));
console.log(generator.next('输入3'));
console.log(generator.next('输入4'));
console.log(generator.next('输入5'));
console.log(generator.next('输入6'));
```

generator.next(value) 的参数 value，将作为上一个 yield 表达式的返回值
首次调用 next() 时，如果不传参数，或者传入的参数会被忽略，因为此时还没有遇到第一个 yield
从第二次调用 next(value) 开始，传入的参数才会被传递给 yield 表达式，作为其返回值

reurn

如果生成器函数内部有 try...finally 结构，finally 块的代码会在 return() 调用后执行

throw

如果生成器内部有对应的 try...catch 块捕获异常，会按照正常的异常处理流程执行
如果未捕获异常，生成器函数会终止，异常向外传播

优化可迭代对象写法

```
//可迭代对象
const iterableObj = {
  //需要遍历的目标数组
  names: ["coderwhy", "XiaoYu", "JS"],
  //生成器替代迭代器
  *[Symbol.iterator]() {
    for(item of this.names) yield item
  }
}

for (const item of iterableObj) {
  console.log(item)//正常遍历输出
}
```

```
const iterableObj = {
  names: ["coderwhy", "XiaoYu", "JS"],
  //第三种写法
  *[Symbol.iterator]() { yield* this.names }
}

for (const item of iterableObj) {
  console.log(item)//正常遍历输出
}
```

yield暂停生成器函数并返回一个值，yield*迭代操作数，并产生它返回的每个值

async/await为了解决多层嵌套异步问题

1、情况一：普通值，异步函数的返回值相当于被包裹到Promise.resolve中；
2 、情况二：返回值是Promise，状态由会由Promise决定;
3 、情况三：异步函数的返回值是一个对象并且实现了thenable，那么会由对象的then方法来决定

返回值是一个Promise

普通函数一旦抛出异常，在终端输出是将会直接抛出堆栈跟踪的错误信息，并且不会继续往下执行
但async异步函数可以视为Promise进行使用，除了正常返回值，自然也包含了接收异常

await

和正常的Promise二阶段通过then调用接收不同，await直接将其作为返回值进行返回，形成同步代码的视觉效果，解决了异步回调等问题
await会等到Promise的状态变成fulfilled状态，之后继续执行异步函数

可以返回普通值，promise，thanable 对象，错误捕获reject

多层嵌套问题

```
// 1.第一种方案: 多次回调
// 回调地狱
requestData("why").then(res => {
  requestData(res + "aaa").then(res => {
    requestData(res + "bbb").then(res => {
      console.log(res)
    })
  })
})
```

细分主题 1

```
2 、链式调用
requestData("why").then(res => {
  return requestData(res + "aaa")
}).then(res => {
  return requestData(res + "bbb")
}).then(res => {
  console.log(res)
})
```

```
function* getData() {
  //左侧接收请求，右侧发送请求
  const res1 = yield requestData("why")//拿到res1的结果执行res2
结果
  const res2 = yield requestData(res1 + "aaa")//拿到res2结果执行
res3结果
  const res3 = yield requestData(res2 + "bbb")//...依次执行
  const res4 = yield requestData(res3 + "ccc")
  console.log(res4)
}
```

```
const generator = getData()
generator.next().value.then(res => {
  generator.next(res).value.then(res => {
    generator.next(res).value.then(res => {
      //无限嵌套下去
      generator.next(res)
    })
  })
})
```

调用的时候，一样需要面对嵌套问题，需要我们手动执行生成器函

3、Generator方案

```
//封装使用方案
function execGenerator(genFn) {
  //获取对应函数的generator
  const generator = genFn()
  function exec(res) {
    const result = generator.next(res)
    if (result.done) return result.value//result里的done是false跟true，为true证明已经到最后了，就返回
result的value
    result.value.then(res => {
      exec(res)//递归
    })
  }
  exec()
}

//使用
execGenerator(getData)//要自动执行就放进参数
```

工具函数
execGenerator自动执行生成器函数

原理同 npm 库co