JS 中的惰性求值

目录

最初的问题：

```
// 需求
// (1) 商品名转为大写
// (2) 取出三个价格低于 10 的商品

const gems = [
  { name: 'Sunstone', price: 4 },
  { name: 'Amethyst', price: 15 },
  { name: 'Prehnite', price: 20 },
  { name: 'Sugilite', price: 7 },
  { name: 'Diopside', price: 3 },
  { name: 'Feldspar', price: 13 },
  { name: 'Dioptase', price: 2 },
  { name: 'Sapphire', price: 20 }
];
```

立马想到：

```
const filter = item => item.price < 10;
const transform = item => ({
  name: item.name.toUpperCase(),
  price: item.price
});

const result = gems
  .filter(filter)
  .map(transform)
  .slice(0, 3);
```

求值策略

介绍：

在计算机科学中，求值策略(Evaluation strategy)是确定编程语言中表达式的求值的一组（通常确定性的）规则。重点典型的位于函数或算子上——求值策略定义何时和以何种次序求值给函数的实际参数，什么时候把它们代换入函数，和代换以何种形式发生。

-- 维基百科

类型：

- 严格求值（Strict evaluation）
  - 严格求值下，传给函数的实际参数总是在调用这个函数之前被求值。

- 非严格求值 / 惰性求值（Non-strict evaluation）
  - 非严格求值下，传给函数的实际参数并不会立即求值，是否需要求值依赖于这个实际参数在函数执行中有没有被使用。

多数现存编程语言对函数使用严格求值

JS 中的惰性求值

短路运算

```
 1 const func = function() {
 2   console.log('func is executed!');
 3 };
 4
 5 const a1 = true || func(); // func 未执行
 6
 7 const a2 = false || func(); // func 执行了
 8
 9 const b1 = true && func(); // func 执行了
10
11 const b2 = false && func(); // func 未执行
```

默认参数

```
let counter = 0;

const func = function() {
  console.log('func is executed!');
  counter++;
  return 10;
};

const a = function(val = func()) {
  console.log('before use "val"');
  console.log('val:', val);
};

a();

a(20);
```

缓存

示例一：

```
1 const func = function() {
2   console.log('func is executed!');
3   return 'func';
4 };
5
6 console.log(func());
7 console.log(func());
8 console.log(func());
```

memoize.js

```
1 const memoize = function(func) {
2   if (typeof func != 'function') {
3     throw new TypeError('Expected a function');
4   }
5
6   const cache = new Map();
7   const memorized = function(...args) {
8     const key = args[0];
9
10    if (!cache.has(key)) {
11      cache.set(key, func.apply(this, args));
12    }
13
14    return cache.get(key);
15  };
16
17  memorized.isMemorized = true;
18
19  return memorized;
20 };
```

示例一（优化版）：

```
const func = function() {
  console.log('func is executed!');
  return 'func';
};

const memorizedFunc = memoize(func);

console.log(memorizedFunc());
console.log(memorizedFunc());
console.log(memorizedFunc());
```

来个更震撼的：

```
let counter = 0;

let fib = function(n) {
  counter++;
  switch (n) {
    case 0:
      return 0;
    case 1:
      return 1;
    default:
      return fib(n - 1) + fib(n - 2);
  }
};

console.log('fib 20:', fib(20));
console.log('counter:', counter, '\n');

counter = 0;
fib = memoize(fib);

console.log('memorizedFib 20:', fib(20));
console.log('counter:', counter);

// console:

// fib 20: 6765
// counter: 21891

// memorizedFib 20: 6765
// counter: 21
```

惰性数组

最初的问题：

```
// 需求
// (1) 商品名转为大写
// (2) 取出三个价格低于 10 的商品

const gems = [
  { name: 'Sunstone', price: 4 },
  { name: 'Amethyst', price: 15 },
  { name: 'Prehnite', price: 20 },
  { name: 'Sugilite', price: 7 },
  { name: 'Diopside', price: 3 },
  { name: 'Feldspar', price: 13 },
  { name: 'Dioptase', price: 2 },
  { name: 'Sapphire', price: 20 }
];
```

立马想到：

```
const filter = item => item.price < 10;
const transform = item => ({
  name: item.name.toUpperCase(),
  price: item.price
});

const result = gems
  .filter(filter)
  .map(transform)
  .slice(0, 3);
```

loop：

```
const filter = item => item.price < 10;
const transform = item => ({
  name: item.name.toUpperCase(),
  price: item.price
});

const result = [];

for (let i = 0, len = gems.length; i < len; i++) {
  const item = gems[i];
  if (filter(item)) {
    result.push(transform(item));
  }
  if (result.length > 2) {
    break;
  }
}
```

lodash：

```
const filter = item => item.price < 10;
const transform = item => ({
  name: item.name.toUpperCase(),
  price: item.price
});

const result = _
  .chain(gems)
  .filter(filter)
  .map(transform)
  .take(3)
  .value();
```

自己实现一个 lazy.js：

```
const _chain = function*(arr) {
  for (let i of arr) {
    yield i;
    this.__index__++;
  }
};

const _filter = function*(flow, condition) {
  for (const data of flow) {
    if (condition(data, this.__index__, this.__value__)) {
      yield data;
    }
  }
};

const _map = function*(flow, transform) {
  for (const data of flow) {
    yield transform(data, this.__index__, this.__value__);
  }
};

const _stop = function*(flow, condition) {
  for (const data of flow) {
    yield data;
    if (condition()) {
      break;
    }
  }
};

const _take = function(flow, num) {
  let _count = 0;
  const _filter = function() {
    _count++;
    return _count >= num;
  };
  return _stop(flow, _filter);
};
```

```
class Lazy {
  constructor(value) {
    if (!(value instanceof Array)) {
      throw new TypeError('Only array is supported.');
    }
    this.__index__ = 0;
    this.__value__ = value;
    this.__iterator__ = _chain.call(this, value);
  }

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

  static chain(value) {
    return new Lazy(value);
  }

  map(callback) {
    this.__iterator__ = _map.call(this, this.__iterator__, callback);
    return this;
  }

  filter(callback) {
    this.__iterator__ = _filter.call(this, this.__iterator__, callback);
    return this;
  }

  take(num) {
    this.__iterator__ = _take.call(this, this.__iterator__, num);
    return this;
  }

  value() {
    const result = [];

    for (let n of this) {
      result.push(n);
    }

    return result;
  }
}
```

谢谢