

# 一、缓存函数memorization

- 缓存函数是指将上次的计算结果缓存起来，当下次调用时，如果遇到相同的参数，就直接返回缓存中的数据

```
1 let add = (a, b) => a + b;  
2 // 假设memoize函数可以实现缓存  
3 let calculate = memoize(add);  
4 calculate(10, 20); // 30  
5 calculate(10, 20); // 相同的参数，第二次调用时，从缓存中取出数据，而非重新计算一次
```

在本例中，我们定义了一个add函数，它的功能就是计算参数a和b的和，假设memoize存在并且能够实现缓存功能，当我们调用calculate的时候，第一次计算的结果是30，如果我们传入相同的参数，第二次调用的时候就直接从缓存中取出数据而非重新计算一次结果，这就是我们的缓存函数。

- 实现原理：把参数和对应的结果数据存到一个对象中，调用时，判断参数对应的数据是否存在，存在就返回对应的结果数据。

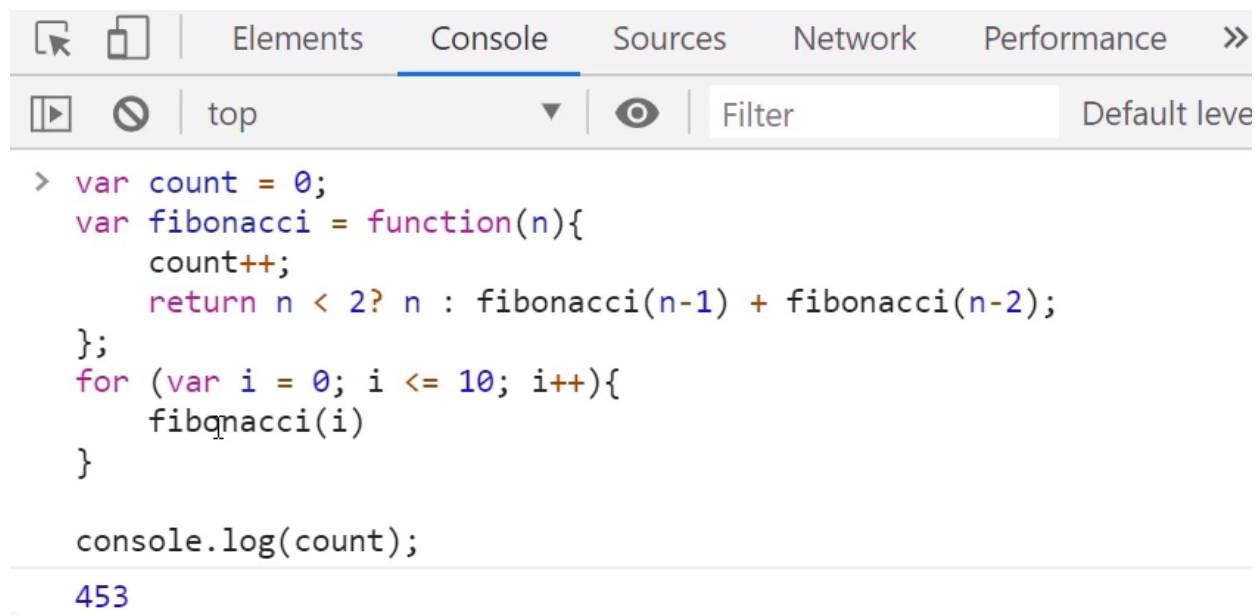
根据上面的原理，我们实现如下代码

```
1 let memoize = function(func) {  
2   let cache = {};  
3   return function(key) {  
4     if (!cache[key]){  
5       cache[key] = func.apply(this, arguments);  
6     }  
7     return cache[key];  
8   }  
9 }
```

我们把函数每一次的执行结果都放入一个对象中，在接下来的执行中，我们到对象中去查找是否已经有相应执行过的值，如果有直接返回该值，如果没有才真正的去执行函数体的求值部分。

缓存函数简单讲就是将函数的计算结果缓存起来，适用于需要大量重复的计算，或者大量计算又依赖于之前的结果的场景，比如递归。

以斐波那契数列为例：



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following JavaScript code and its output:

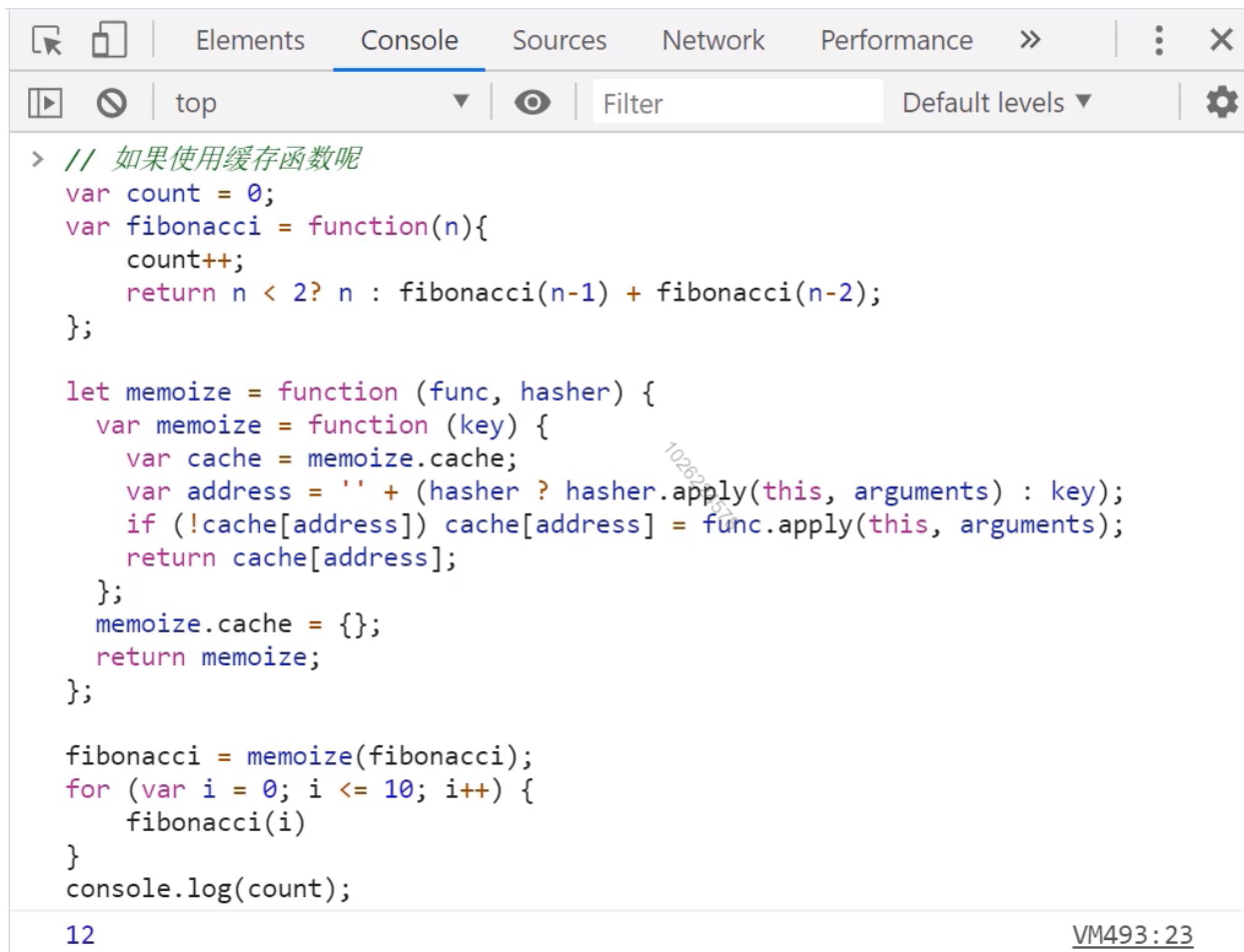
```
> var count = 0;
  var fibonacci = function(n){
    count++;
    return n < 2? n : fibonacci(n-1) + fibonacci(n-2);
  };
  for (var i = 0; i <= 10; i++){
    fibonacci(i)
  }

  console.log(count);
```

The output of the code is the number 453, indicating that the `fibonacci` function was called 453 times during the execution of the loop.

可以看到10次循环斐波那契函数被调用了453次。

如果使用缓存函数呢（示例使用了underscore的缓存函数）？



```
> // 如果使用缓存函数呢
var count = 0;
var fibonacci = function(n){
    count++;
    return n < 2? n : fibonacci(n-1) + fibonacci(n-2);
};

let memoize = function (func, hasher) {
    var memoize = function (key) {
        var cache = memoize.cache;
        var address = '' + (hasher ? hasher.apply(this, arguments) : key);
        if (!cache[address]) cache[address] = func.apply(this, arguments);
        return cache[address];
    };
    memoize.cache = {};
    return memoize;
};

fibonacci = memoize(fibonacci);
for (var i = 0; i <= 10; i++) {
    fibonacci(i)
}
console.log(count);
```

12 VM493:23

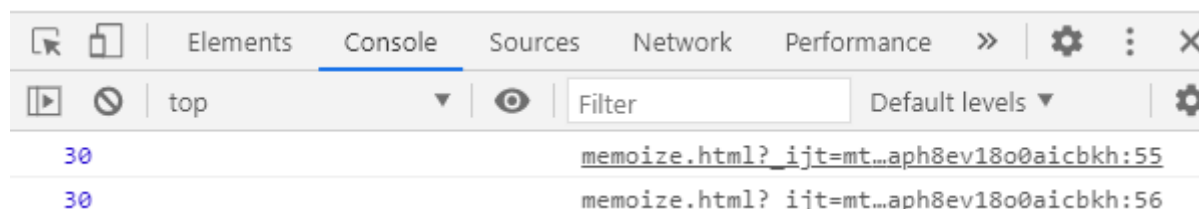
因为使用了缓存函数，10次循环斐波那契数列由原来的453次降低为了12次，这说明缓存函数完全可以应付大量的重复计算，或者是大量的计算又依赖于之前的结果的运算场景。

上面的缓存函数只能应付一元函数的场景，如下例可以看到如果第二个参数改变以后输出的值应该是40，但实际没有变化，还是30。

```

37 // 创建加法运算
38 let add = (a, b) => a + b;
39
40 let memoize = function(fn) {
41   let cache = {}; // 创建缓存对象
42   return function (key) {
43     // 第一次执行时key不存在，将加法运算的结果存入缓存函数
44     if (!cache[key]) {
45       cache[key] = fn.apply(this, arguments);
46     }
47     // 第二次执行时key已存在，直接返回
48     return cache[key];
49   }
50 }
51
52 // 将加法运算传入缓存函数
53 let calculate = memoize(add);
54
55 console.log(calculate(10, 20)); // 30
56 console.log(calculate(10, 30)); // 30

```



如果要实现多元函数的缓存函数怎么做呢？

首先我们想到缓存函数的实现原理是将参数和计算结果存入一个对象中，调用时判断参数是否存在，存在则直接返回结果。那么我们就从缓存对象的key上找解决方案，由于上面的缓存函数代码中都只取了函数的第一个参数作为key，这样就导致后面的参数无论怎么变化它始终返回了已经用第一个参数存入缓存对象的计算结果，所以我们通过argument对象获取所有的参数并转为字符串作为缓存对象的key。

```

36 // 创建加法运算
37 let add = (a, b) => a + b;
38
39 let memoize = function(fn) {
40   let cache = {}; // 创建缓存对象
41   return function () {
42     // 将传入的全部参数获取作为key
43     let key = [].slice.call(arguments).join(',');
44     // 执行时key不存在，将加法运算的结果存入缓存函数
45     if (!cache[key]) {
46       cache[key] = fn.apply(this, arguments);
47       // 第一次执行缓存了10, 20 输出{10,20: 30}
48       // 第二次执行缓存了10, 30 输出{10,20: 30, 10,30: 40}
49       console.log(cache);
50     }
51     // 执行时key已存在，直接返回
52     return cache[key];
53   }
54 }
55
56 // 将加法运算传入缓存函数
57 let calculate = memoize(add);
58
59 console.log(calculate(10, 20)); // 30
60 console.log(calculate(10, 20)); // 30
61 console.log(calculate(10, 30)); // 40
62 console.log(calculate(10, 30)); // 40

```

Elements	Console	Sources	Network	Performance	»	⚙	⋮	✕
▶	top	Filter	Default levels	⚙				
▶ {10, 20: 30}								
30								
30								
▶ {10, 20: 30, 10, 30: 40}								
40								
40								

当然，这种方式也有一定的局限性，如果传入的参数类型比较复杂就凉了（比如对象嵌套）。

underscore中关于memoize函数的实现如下：

```
1 let memoize = function (func, hasher) {
2   var memoize = function (key) {
3     var cache = memoize.cache;
4     var address = '' + (hasher ? hasher.apply(this, arguments) : key);
5     if (!cache[address]) cache[address] = func.apply(this, arguments);
6     return cache[address];
7   };
8   memoize.cache = {};
9   return memoize;
10 };
```

它多了一个参数hasher，这个参数也是一个function，它的作用就是用来计算key的，如果我们传入了hasher，则用hasher函数来计算key，否则就用memoize方法传入的第一个参数，接着就去判断，如果这个key没有被求值过就去执行，之后再将这个对象返回。

## 二、柯里化函数curry

- 在数学和计算机科学中，柯里化是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术

```
1 function girl(name, age, single) {
2   return `我叫${name},我今年${age}岁,我${single}单身`;
3 }
4 // 调用
5 let lincancan = girl('林灿灿', 18, '不是');
6 console.log(lincancan); // 我叫林灿灿,我今年18岁,我不是单身
```

在本例中，定义了一个普通的girl函数，这是一个没有被柯里化的函数，它接收三个参数，在一次调用中需要一次将全部参数传递进去。

柯里化是指一个函数在一次调用中不直接接收所需要的全部参数，而是首先接收第一个参数，然后返回一个新函数，然后继续在这个新函数中传入第二个参数并继续执行，之后又返回一个新函数继续传入第三个参数，以此类推，直到所有的参数都被依次传入这个柯里化的函数链中，这个函数的最后一个返回值就是我们所期望的那个值。

```
1 function girl(name) {
2   return function (age) {
3     return function (single) {
4       return `我叫${name},我今年${age}岁,我${single}单身`;
5     }
6   }
7 }
8 // 调用
9 let rst = girl('林灿灿')(18)('不是');
```

在本例中，girl函数接收第一个参数name，然后它返回一个函数，这个函数接收另一个参数age，然后这个函数又返回一个函数，这个函数接收参数single，通过调用也得到同样的结果。

需求：检测字符串中是否包含空格

```
1 // 封装函数
2 let matching = (reg, str) => reg.test(str);
3 matching(/\s+/g, 'hello world'); // true
4 matching(/\s+/g, 'abcdefg'); // false
```

上面的matching函数封装了一个正则校验的功能，正常来说，我们直接调用matching传入正则和待检验的字符串即可，本例中使用正则去检测字符串是否有空格，如果我们有很多的字符串需要检测的话，那么我们需要将第一个reg参数进行复用，这样就可以提高我们的工作效率，我们可以使用柯里化的思想来实现这个需求。



```

1 // 柯里化
2 let curry = (reg)=>{
3   return (str)=>{
4     return reg.test(str);
5   }
6 }
7 let hasSpace = curry(/\s+/g);
8 hasSpace('hello world'); // true
9 hasSpace('abcdefg'); // false
10 hasSpace('I Love China'); // true

```

我们定义一个curry函数，它接收一个正则表达式作为一个参数，同时它又返回一个函数，这个函数接收一个待检测字符串str作为参数，它的返回值就是正则匹配的结果，这样我们实际调用的时候只需要定义一个变量hasSpace去接收curry函数的返回值，此时返回的hasSpace也是一个函数，我们传入需要匹配的字符串即可达到代码复用。

需求：请获取数组对象的age属性

```

1 let persons = [
2   { 'name': 'Peter', age: 21 },
3   { 'name': 'Mark', age: 28 },
4   { 'name': 'Josn', age: 19 },
5   { 'name': 'Jane', age: 31 },
6   { 'name': 'Tony', age: 35 }
7 ]

```

可以直接使用高阶函数map方法直接获取

```

1 // 不用柯里化
2 let getAge = persons.map(item => {
3   return item.age;
4 })
5 console.log(getAge); // [21,28,19,31,35]

```



柯里化这个概念以及实现本身都非常的难，平时写代码几乎也用的不多，能使用的场景也不太多，关键是我们理解它的思想，所以大多数情况下都选择了其他简单的方式去实现，所以这里我们使用loadsh去实现我们柯里化的需求。

- loadsh是一个一致性、模块化、高性能的JavaScript实用工具库
- loadsh通过降低Array、Number、Object、String等等的使用难度从而让JavaScript变得更简单

## loadsh的安装和使用：

- 使用 `npm i loadsh -S` 进行安装

```
1 // loadsh的curry
2 const _ = require("loadsh");
3 let getProp = _.curry((key, obj) => {
4   return obj[key];
5 });
6 let age = persons.map(getProp("age"));
7 console.log(age); // [ 21, 28, 19, 31, 35 ]
```

如上例，先通过require将loadsh引入，我们为了获取age属性还要再写一个getProp函数，这个getProp函数通过使用loadsh的curry方法以后就实现了我们的柯里化，之后将符合我们需求的属性return出来，这样写看起来麻烦一些，但是getProp函数编写一次之后可以多次使用，在这个场景中我们将map的第一个参数柯里化之后，就能实现它的复用了，而且非常的灵活，这样就不需要每次map计算都新写一个匿名函数了，节省了不少代码。

## 三、偏函数

- 柯里化是将一个多参数函数转换成多个单参数函数，也就是将一个n元函数转换成n个一元函数

- 偏函数则是固定一个函数的一个或多个参数，也就是将一个n元函数转换成一个n - x元函数

元指的是函数参数的个数，也就是说柯里化最终会将函数的多个参数转换为单个的形式，而偏函数会固定一个或者多个参数。

偏函数实现案例：

```
1 // 使用bind实现
2 let add = (x, y) => x + y;
3 let rst = add.bind(null, 1);
4 rst(2); // 3
```

我们可以使用bind实现，bind的另一个简单的用法就是使一个函数拥有预设的初始参数，只要将这些参数作为bind的参数写在this后面，当绑定函数被调用时，这些参数会被插入到目标函数的参数列表的开始位置，传递给绑定函数的参数会跟在它们的后面。本例中就使用bind实现了一个简单的偏函数。