```javascript
// 假设我们有一个求长方形面积的函数
function getArea(width, height) {
    return width * height
}
// 如果我们碰到的长方形的宽老是10
const area1 = getArea(10, 20)
const area2 = getArea(10, 30)
const area3 = getArea(10, 40)

// 我们可以使用闭包柯里化这个计算面积的函数
function getArea(width) {
    return height => {
        return width * height
    }
}

const getTenWidthArea = getArea(10)
// 之后碰到宽度为10的长方形就可以这样计算面积
const area1 = getTenWidthArea(20)

// 而且如果遇到宽度偶尔变化也可以轻松复用
const getTwentyWidthArea = getArea(20)
```

15.2.2. 使用闭包模拟私有方法

在 `JavaScript` 中，没有支持声明私有变量，但我们可以使用闭包来模拟私有方法

下面举个例子：

```javascript
var Counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
})();

var Counter1 = makeCounter();
var Counter2 = makeCounter();
console.log(Counter1.value()); /* logs 0 */
Counter1.increment();
Counter1.increment();
console.log(Counter1.value()); /* logs 2 */
Counter1.decrement();
console.log(Counter1.value()); /* logs 1 */
console.log(Counter2.value()); /* logs 0 */
```

上述通过使用闭包来定义公共函数，并令其可以访问私有函数和变量，这种方式也叫模块方式

两个计数器 `Counter1` 和 `Counter2` 是维护它们各自的独立性的，每次调用其中一个计数器时，通过改变这个变量的值，会改变这个闭包的词法环境，不会影响另一个闭包中的变量

15.2.3. 其他

例如计数器、延迟调用、回调等闭包的应用，其核心思想还是创建私有变量和延长变量的生命周期

15.3. 注意事项

如果不是某些特定任务需要使用闭包，在其它函数中创建函数是不明智的，因为闭包在处理速度和内存消耗方面对脚本性能具有负面影响

例如，在创建新的对象或者类时，方法通常应该关联于对象的原型，而不是定义到对象的构造器中。

原因在于每个对象的创建，方法都会被重新赋值

JavaScript | 复制代码

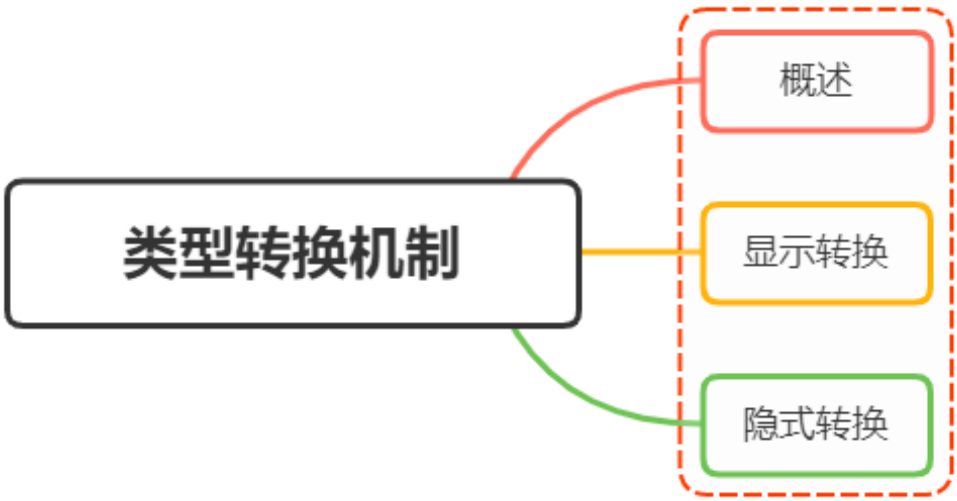
```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function() {
    return this.name;
  };

  this.getMessage = function() {
    return this.message;
  };
}
```

上面的代码中，我们并没有利用到闭包的好处，因此可以避免使用闭包。修改成如下：

JavaScript | 复制代码

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function() {
  return this.name;
};
MyObject.prototype.getMessage = function() {
  return this.message;
};
```

16. 谈谈 JavaScript 中的类型转换机制



16.1. 概述

前面我们讲到，`JS` 中有六种简单数据类型：`undefined`、`null`、`boolean`、`string`、`number`、`symbol`，以及引用类型：`object`

但是我们在声明的时候只有一种数据类型，只有到运行期间才会确定当前类型

JavaScript | 复制代码

```
let x = y ? 1 : a;
```

上面代码中，`x` 的值在编译阶段是无法获取的，只有等到程序运行时才能知道

虽然变量的数据类型是不确定的，但是各种运算符对数据类型是有要求的，如果运算子的类型与预期不符合，就会触发类型转换机制

常见的类型转换有：

- 强制转换（显示转换）
- 自动转换（隐式转换）

16.2. 显示转换

显示转换，即我们很清楚可以看到这里发生了类型的转变，常见的方法有：

- Number()
- parseInt()
- String()
- Boolean()

16.2.1. Number()

将任意类型的值转化为数值

先给出类型转换规则：

原始值	转换结果
Undefined	NaN
Null	0
true	1
false	0
String	根据语法和转换规则来转换
Symbol	Throw a TypeError exception
Object	先调用toPrimitive，再调用toNumber

实践一下：

```javascript
Number(324) // 324

// 字符串：如果可以被解析为数值，则转换为相应的数值
Number('324') // 324

// 字符串：如果不可以被解析为数值，返回 NaN
Number('324abc') // NaN

// 空字符串转为0
Number('') // 0

// 布尔值：true 转成 1，false 转成 0
Number(true) // 1
Number(false) // 0

// undefined：转成 NaN
Number(undefined) // NaN

// null：转成0
Number(null) // 0

// 对象：通常转换成NaN(除了只包含单个数值的数组)
Number({a: 1}) // NaN
Number([1, 2, 3]) // NaN
Number([5]) // 5
```

从上面可以看到，`Number` 转换的时候是很严格的，只要有一个字符无法转成数值，整个字符串就会被转为 `NaN`

16.2.2. parseInt()

`parseInt` 相比 `Number` ，就没那么严格了，`parseInt` 函数逐个解析字符，遇到不能转换的字符就停下来

```javascript
parseInt('32a3') //32
```

16.2.3. String()

可以将任意类型的值转化成字符串

给出转换规则图：

原始值	转换结果
Undefined	'Undefined'
Boolean	'true' or 'false'
Number	对应的字符串类型
String	String
Symbol	Throw a TypeError exception
Object	先调用toPrimitive，再调用toNumber

实践一下：

JavaScript | 复制代码

```
// 数值：转为相应的字符串
String(1) // "1"

//字符串：转换后还是原来的值
String("a") // "a"

//布尔值：true转为字符串"true"，false转为字符串"false"
String(true) // "true"

//undefined：转为字符串"undefined"
String(undefined) // "undefined"

//null：转为字符串"null"
String(null) // "null"

//对象
String({a: 1}) // "[object Object]"
String([1, 2, 3]) // "1,2,3"
```

16.2.4. Boolean()

可以将任意类型的值转为布尔值，转换规则如下：

数据类型	转换为 true 的值	转换为 false 的值
Boolean	true	false
String	非空字符串	""（空字符串）
Number	非零数值（包括无穷值）	0、NaN（参见后面的相关内容）
Object	任意对象	null
Undefined	N/A（不存在）	undefined

实践一下：

JavaScript 复制代码

```
Boolean(undefined) // false
Boolean(null) // false
Boolean(0) // false
Boolean(NaN) // false
Boolean('') // false
Boolean({}) // true
Boolean([]) // true
Boolean(new Boolean(false)) // true
```

16.3. 隐式转换

在隐式转换中，我们可能最大的疑惑是 ： 何时发生隐式转换?

我们这里可以归纳为两种情况发生隐式转换的场景：

- 比较运算（`==`、`!=`、`>`、`<`）、`if`、`while`需要布尔值地方
- 算术运算（`+`、`-`、`*`、`/`、`%`）

除了上面的场景，还要求运算符两边的操作数不是同一类型

16.3.1. 自动转换为布尔值

在需要布尔值的地方，就会将非布尔值的参数自动转为布尔值，系统内部会调用`Boolean`函数

可以得出个小结：

- undefined
- null
- false
- +0
- -0

- NaN
- ""

除了上面几种会被转化成 `false` ，其他都换被转化成 `true`

16.3.2. 自动转换成字符串

遇到预期为字符串的地方，就会将非字符串的值自动转为字符串

具体规则是：先将复合类型的值转为原始类型的值，再将原始类型的值转为字符串

常发生在 `+` 运算中，一旦存在字符串，则会进行字符串拼接操作

JavaScript 复制代码

```
'5' + 1 // '51'
'5' + true // "5true"
'5' + false // "5false"
'5' + {} // "5[object Object]"
'5' + [] // "5"
'5' + function (){} // "5function (){}"
'5' + undefined // "5undefined"
'5' + null // "5null"
```

16.3.3. 自动转换成数值

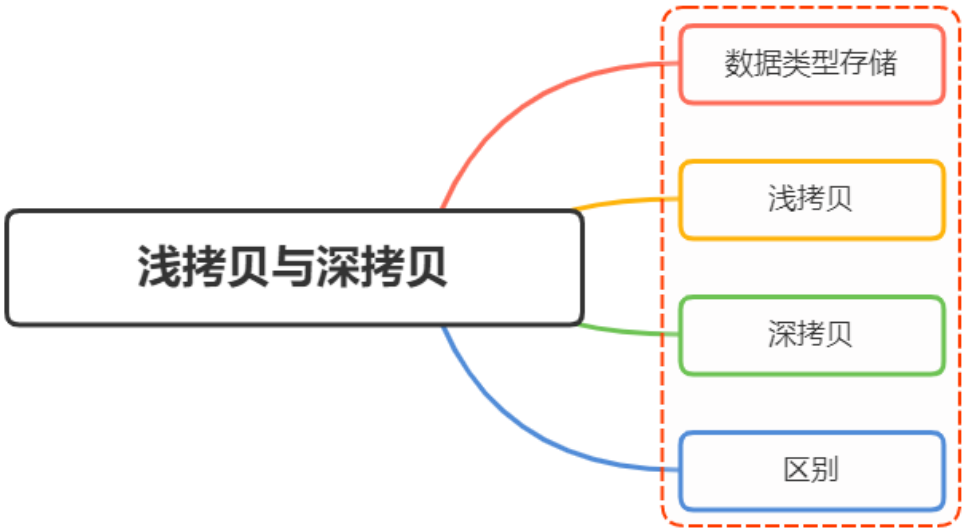
除了 `+` 有可能把运算子转为字符串，其他运算符都会把运算子自动转成数值

JavaScript 复制代码

```
'5' - '2' // 3
'5' * '2' // 10
true - 1  // 0
false - 1 // -1
'1' - 1   // 0
'5' * []    // 0
false / '5' // 0
'abc' - 1   // NaN
null + 1 // 1
undefined + 1 // NaN
```

`null` 转为数值时，值为 `0` 。 `undefined` 转为数值时，值为 `NaN`

17. 深拷贝浅拷贝的区别？如何实现一个深拷贝？



17.1. 数据类型存储

前面文章我们讲到，`JavaScript` 中存在两大数据类型：

- 基本类型
- 引用类型

基本类型数据保存在在栈内存中

引用类型数据保存在堆内存中，引用数据类型的变量是一个指向堆内存中实际对象的引用，存在栈中

17.2. 浅拷贝

浅拷贝，指的是创建新的数据，这个数据有着原始数据属性值的一份精确拷贝

如果属性是基本类型，拷贝的就是基本类型的值。如果属性是引用类型，拷贝的就是内存地址

即浅拷贝是拷贝一层，深层次的引用类型则共享内存地址

下面简单实现一个浅拷贝

```javascript
function shallowClone(obj) {
    const newObj = {};
    for(let prop in obj) {
        if(obj.hasOwnProperty(prop)){
            newObj[prop] = obj[prop];
        }
    }
    return newObj;
}
```

在 `JavaScript` 中，存在浅拷贝的现象有：

- `Object.assign`
- `Array.prototype.slice()`，`Array.prototype.concat()`
- 使用拓展运算符实现的复制

17.2.1. Object.assign

```javascript
var obj = {
    age: 18,
    nature: ['smart', 'good'],
    names: {
        name1: 'fx',
        name2: 'xka'
    },
    love: function () {
        console.log('fx is a great girl')
    }
}
var newObj = Object.assign({}, fxObj);
```

17.2.2. slice()

```javascript
const fxArr = ["One", "Two", "Three"]
const fxArrs = fxArr.slice(0)
fxArrs[1] = "love";
console.log(fxArr) // ["One", "Two", "Three"]
console.log(fxArrs) // ["One", "love", "Three"]
```

17.2.3. concat()

```javascript
const fxArr = ["One", "Two", "Three"]
const fxArrs = fxArr.concat()
fxArrs[1] = "love";
console.log(fxArr) // ["One", "Two", "Three"]
console.log(fxArrs) // ["One", "love", "Three"]
```

17.2.4. 拓展运算符

```javascript
const fxArr = ["One", "Two", "Three"]
const fxArrs = [...fxArr]
fxArrs[1] = "love";
console.log(fxArr) // ["One", "Two", "Three"]
console.log(fxArrs) // ["One", "love", "Three"]
```

17.3. 深拷贝

深拷贝开辟一个新的栈，两个对象属完成相同，但是对应两个不同的地址，修改一个对象的属性，不会改变另一个对象的属性

常见的深拷贝方式有：

- _.cloneDeep()
- jQuery.extend()
- JSON.stringify()
- 手写循环递归

17.3.1. _.cloneDeep()

JavaScript 复制代码

```
const _ = require('lodash');
const obj1 = {
    a: 1,
    b: { f: { g: 1 } },
    c: [1, 2, 3]
};
const obj2 = _.cloneDeep(obj1);
console.log(obj1.b.f === obj2.b.f);// false
```

17.3.2. jQuery.extend()

JavaScript 复制代码

```
const $ = require('jquery');
const obj1 = {
    a: 1,
    b: { f: { g: 1 } },
    c: [1, 2, 3]
};
const obj2 = $.extend(true, {}, obj1);
console.log(obj1.b.f === obj2.b.f); // false
```

17.3.3. JSON.stringify()

JavaScript 复制代码

```
const obj2=JSON.parse(JSON.stringify(obj1));
```

但是这种方式存在弊端，会忽略 `undefined` 、 `symbol` 和 `函数`

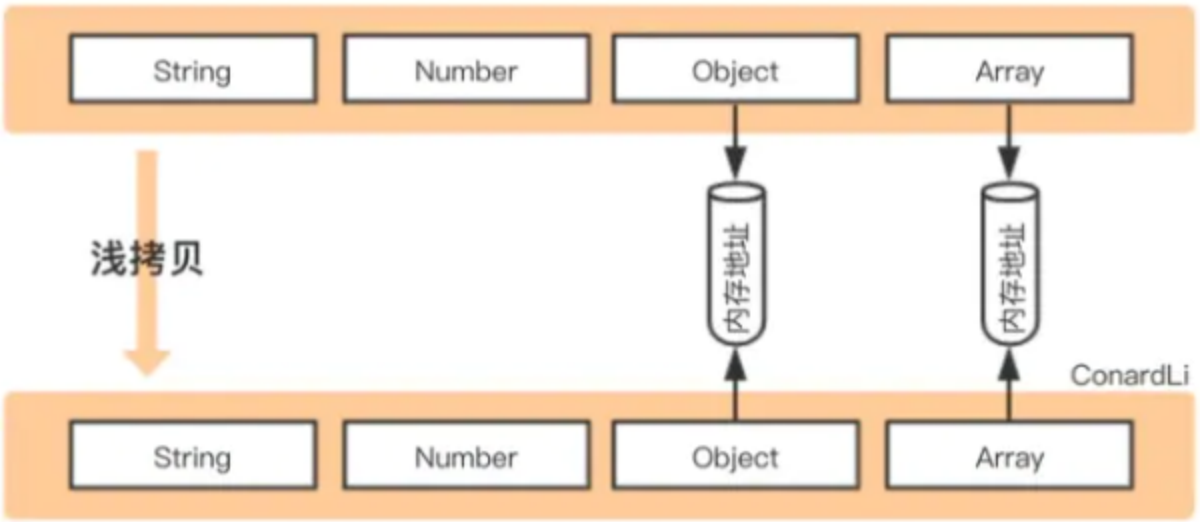
```javascript
const obj = {
    name: 'A',
    name1: undefined,
    name3: function() {},
    name4:  Symbol('A')
}
const obj2 = JSON.parse(JSON.stringify(obj));
console.log(obj2); // {name: "A"}
```

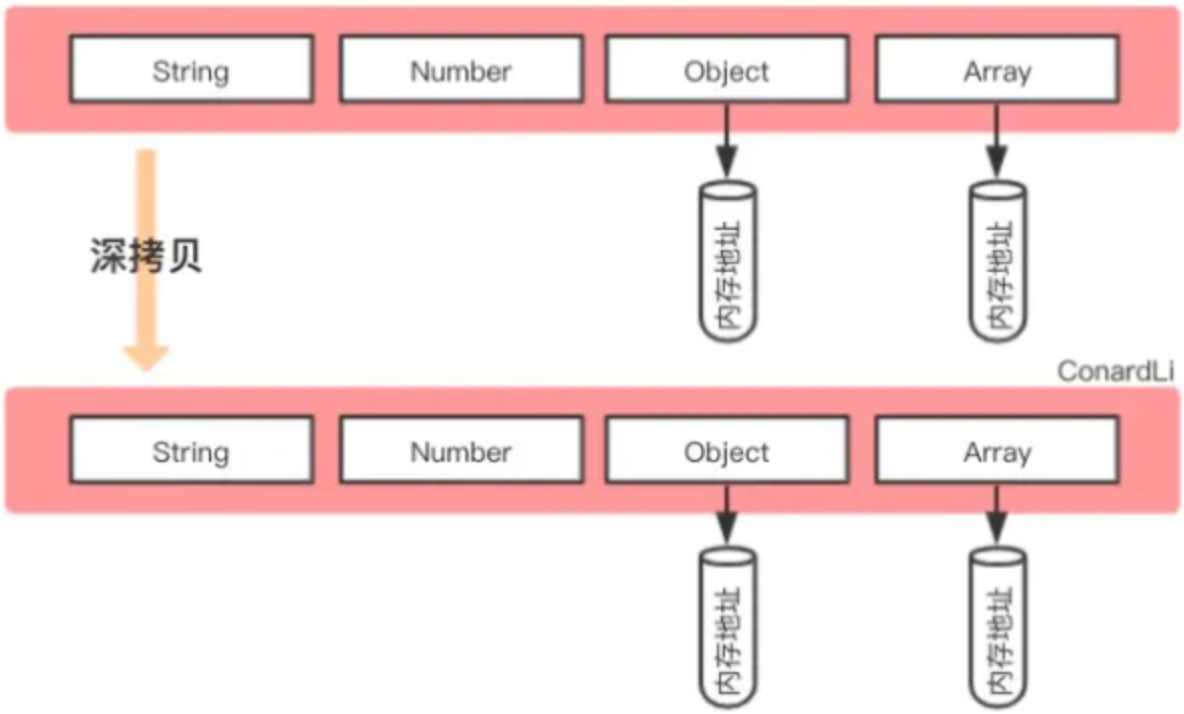
17.3.4. 循环递归

```javascript
function deepClone(obj, hash = new WeakMap()) {
  if (obj === null) return obj; // 如果是null或者undefined我就不进行拷贝操作
  if (obj instanceof Date) return new Date(obj);
  if (obj instanceof RegExp) return new RegExp(obj);
  // 可能是对象或者普通的值  如果是函数的话是不需要深拷贝
  if (typeof obj !== "object") return obj;
  // 是对象的话就要进行深拷贝
  if (hash.get(obj)) return hash.get(obj);
  let cloneObj = new obj.constructor();
  // 找到的是所属类原型上的constructor,而原型上的 constructor指向的是当前类本身
  hash.set(obj, cloneObj);
  for (let key in obj) {
    if (obj.hasOwnProperty(key)) {
      // 实现一个递归拷贝
      cloneObj[key] = deepClone(obj[key], hash);
    }
  }
  return cloneObj;
}
```

17.4. 区别

下面首先借助两张图，可以更加清晰看到浅拷贝与深拷贝的区别





从上图发现，浅拷贝和深拷贝都创建出一个新的对象，但在复制对象属性的时候，行为就不一样

浅拷贝只复制属性指向某个对象的指针，而不复制对象本身，新旧对象还是共享同一块内存，修改对象属性会影响原对象

```JavaScript
// 浅拷贝
const obj1 = {
    name : 'init',
    arr : [1,[2,3],4],
};
const obj3=shallowClone(obj1) // 一个浅拷贝方法
obj3.name = "update";
obj3.arr[1] = [5,6,7] ; // 新旧对象还是共享同一块内存

console.log('obj1',obj1) // obj1 { name: 'init',  arr: [ 1, [ 5, 6, 7 ], 4 ] }
console.log('obj3',obj3) // obj3 { name: 'update', arr: [ 1, [ 5, 6, 7 ], 4 ] }
```

但深拷贝会另外创造一个一模一样的对象，新对象跟原对象不共享内存，修改新对象不会改到原对象

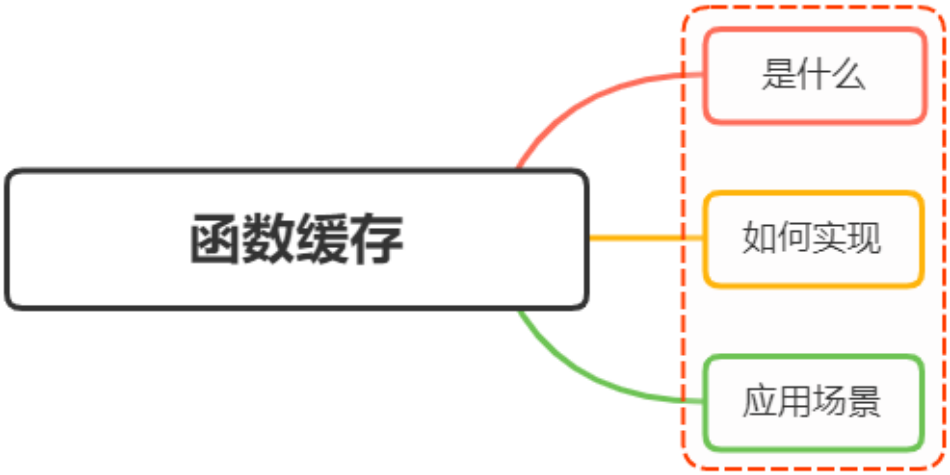
```JavaScript
// 深拷贝
const obj1 = {
    name : 'init',
    arr : [1,[2,3],4],
};
const obj4=deepClone(obj1) // 一个深拷贝方法
obj4.name = "update";
obj4.arr[1] = [5,6,7] ; // 新对象跟原对象不共享内存

console.log('obj1',obj1) // obj1 { name: 'init', arr: [ 1, [ 2, 3 ], 4 ] }
console.log('obj4',obj4) // obj4 { name: 'update', arr: [ 1, [ 5, 6, 7 ], 4 ] }
```

17.5. 小结

前提为拷贝类型为引用类型的情况下：

- 浅拷贝是拷贝一层，属性为对象时，浅拷贝是复制，两个对象指向同一个地址
- 深拷贝是递归拷贝深层次，属性为对象时，深拷贝是新开栈，两个对象指向不同的地址

18. Javascript中如何实现函数缓存？函数缓存有哪些应用场景?



18.1. 是什么

函数缓存，就是将函数运算过的结果进行缓存

本质上就是用空间（缓存存储）换时间（计算过程）

常用于缓存数据计算结果和缓存对象

JavaScript | 复制代码

```
const add = (a,b) => a+b;
const calc = memoize(add); // 函数缓存
calc(10,20);// 30
calc(10,20);// 30 缓存
```

缓存只是一个临时的数据存储，它保存数据，以便将来对该数据的请求能够更快地得到处理

18.2. 如何实现

实现函数缓存主要依靠闭包、柯里化、高阶函数，这里再简单复习下：

18.2.1. 闭包

闭包可以理解成，函数 + 函数体内可访问的变量总和

```javascript
(function() {
    var a = 1;
    function add() {
        const b = 2
        let sum = b + a
        console.log(sum); // 3
    }
    add()
})()
```

`add` 函数本身，以及其内部可访问的变量，即 `a = 1`，这两个组合在一起就形成了闭包

18.2.2. 柯里化

把接受多个参数的函数转换成接受一个单一参数的函数

```javascript
// 非函数柯里化
var add = function (x,y) {
    return x+y;
}
add(3,4) //7

// 函数柯里化
var add2 = function (x) {
    //**返回函数**
    return function (y) {
        return x+y;
    }
}
add2(3)(4) //7
```

将一个二元函数拆分成两个一元函数

18.2.3. 高阶函数

通过接收其他函数作为参数或返回其他函数的函数

```javascript
function foo(){
  var a = 2;

  function bar() {
    console.log(a);
  }
  return bar;
}
var baz = foo();
baz();//2
```

函数 `foo` 如何返回另一个函数 `bar`， `baz` 现在持有对 `foo` 中定义的`bar` 函数的引用。由于闭包特性， `a` 的值能够得到

下面再看看如何实现函数缓存，实现原理也很简单，把参数和对应的结果数据存在一个对象中，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，否则就返回计算结果

如下所示

```javascript
const memoize = function (func, content) {
  let cache = Object.create(null)
  content = content || this
  return (...key) => {
    if (!cache[key]) {
      cache[key] = func.apply(content, key)
    }
    return cache[key]
  }
}
```

调用方式也很简单

```javascript
const calc = memoize(add);
const num1 = calc(100,200)
const num2 = calc(100,200) // 缓存得到的结果
```

过程分析：

- 在当前函数作用域定义了一个空对象，用于缓存运行结果
- 运用柯里化返回一个函数，返回的函数由于闭包特性，可以访问到 `cache`