## 元素合并

reduce\_.reduce(list, iteratee, [memo], [context])

reduce方法把list中元素归结为一个单独的数值。

****Memo****是reduce函数的初始值，reduce的每一步都需要由iteratee返回。这个迭代传递4个参数：memo,value 和 迭代的index（或者 key）和最后一个引用的整个 list。

如果没有memo传递给reduce的初始调用，iteratee不会被列表中的第一个元素调用。第一个元素将取代 传递给列表中下一个元素调用iteratee的memo参数。

1. var sum = \_.reduce([1, 2, 3], function(memo, num){ return memo + num; }, 0);
2. => 6

## 查找和过滤

find\_.find(list, predicate, [context])

在list中逐项查找，返回第一个通过****predicate****迭代函数真值检测的元素值，如果没有值传递给测试迭代器将返回undefined。

如果找到匹配的元素，函数将立即返回，不会遍历整个list。

1. var even = \_.find([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; });
2. => 2

\_.filter(list, predicate, [context])

遍历list中的每个值，返回包含所有通过predicate真值检测的元素值。

1. var evens = \_.filter([1, 2, 3, 4, 5, 6], function(num){ return num % 2 == 0; });
2. => [2, 4, 6]

## 过滤

\_.where(list, properties)

遍历list中的每一个值，返回一个数组，这个数组包含****properties****所列出的属性的所有的 键 - 值对。

1. var list = [
2. {title:"AAA",year: 1982},
3. {title:"BBB",year: 1900},
4. {title:"CCC",year: 1982}
5. ];
6. console.log(
7. \_.where(list,{year: 1982})
8. );
9. => [ { title: 'AAA', year: 1982 }, { title: 'CCC', year: 1982 } ]

## 判断和提取

\_.contains(list, value, [fromIndex])

判断元素是否在list中，如果list包含指定的value则返回true（愚人码头注：使用===检测）。如果list 是数组，内部使用indexOf判断。使用****fromIndex****来给定开始检索的索引位置。

1. \_.contains([1, 2, 3], 3);
2. => true

\_.pluck(list, propertyName)

提取一个集合里指定的属性值，****pluck****也许是map最常使用的用例模型的简化版本，即萃取数组对象中某属性值，返回一个数组。

1. var stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];
2. \_.pluck(stooges, 'name');
3. => ["moe", "larry", "curly"]

## 判断最大最小值

\_.max(list, [iteratee], [context])

返回list中的****最大值****。如果传递iteratee参数，iteratee将作为list中每个值的排序依据。如果list为空，将返回-Infinity，所以你可能需要事先用isEmpty检查 list 。

1. var stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];
2. \_.max(stooges, function(stooge){ return stooge.age; });
3. => {name: 'curly', age: 60};

\_.min(list, [iteratee], [context])

返回list中的****最小值****。如果传递iteratee参数，iteratee将作为list中每个值的排序依据。如果list为空，将返回-Infinity，所以你可能需要事先用isEmpty检查 list 。

1. var numbers = [10, 5, 100, 2, 1000];
2. \_.min(numbers);
3. => 2

## sortBy 排序

\_.sortBy(list, iteratee, [context])

返回一个排序后的list拷贝副本。如果传递****iteratee****参数，****iteratee****将作为list中每个值的排序依据。迭代器也可以是字符串的属性的名称进行排序的(比如 length)。

1. \_.sortBy([1, 2, 3, 4, 5, 6], function(num){ return Math.sin(num); });
2. => [5, 4, 6, 3, 1, 2]
4. var stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];
5. \_.sortBy(stooges, 'name');
6. => [{name: 'curly', age: 60}, {name: 'larry', age: 50}, {name: 'moe', age: 40}];

## 转换和获取个数

\_.toArray(list)

把list(任何可以迭代的对象)转换成一个数组，在****转换**** arguments 对象时非常有用。

1. (function(){ return \_.toArray(arguments).slice(1); })(1, 2, 3, 4);
2. => [2, 3, 4]

\_.size(list)

返回list的长度。

1. \_.size({one: 1, two: 2, three: 3});
2. => 3

## 元素位置操作

\_.first(array, [n])

返回array（数组）的****第一个元素****。传递 n参数将返回数组中从第一个元素开始的n个元素。

1. \_.first([5, 4, 3, 2, 1]);
2. => 5

\_.initial(array, [n])

返回数组中除了最后一个元素外的其他全部元素。 在arguments对象上特别有用。传递 n参数将从结果中排除从最后一个开始的n个元素

1. \_.initial([5, 4, 3, 2, 1]);
2. => [5, 4, 3, 2]

\_.last(array, [n])

返回array（数组）的最后一个元素。传递 n参数将返回数组中从最后一个元素开始的n个元素。

1. \_.last([5, 4, 3, 2, 1]);
2. => 1

\_.rest(array, [index])

返回数组中除了第一个元素外的其他全部元素。****传递**** index 参数将返回从index开始的剩余所有元素 。

1. \_.rest([5, 4, 3, 2, 1]);
2. => [4, 3, 2, 1]

## 获取索引位置

\_.indexOf(array, value, [isSorted])

****返回value****在该 array 中的索引值，如果value不存在 array中就返回-1。使用原生的indexOf 函数，除非它失效。

如果您正在使用一个大数组，你知道数组已经排序，传递true给isSorted将更快的用二进制搜索..,或者，传递一个数字作为第三个参数，为了在给定的索引的数组中寻找第一个匹配值。

1. \_.indexOf([1, 2, 3], 2);
2. => 1

\_.lastIndexOf(array, value, [fromIndex])

返回value在该 array 中的从最后开始的索引值，如果value不存在 array中就返回-1。

如果支持原生的****lastIndexOf****，将使用原生的****lastIndexOf****函数。传递fromIndex将从你给定的索性值开始搜索。

1. \_.lastIndexOf([1, 2, 3, 1, 2, 3], 2);
2. => 4

\_.sortedIndex(list, value, [iteratee], [context])

使用二分查找确定value在list中的位置序号，value按此序号插入能保持list原有的排序。

如果提供****iterator****函数，****iterator****将作为list排序的依据，包括你传递的value 。iterator也可以是字符串的属性名用来排序(比如length)。

1. \_.sortedIndex([10, 20, 30, 40, 50], 35);
2. => 3
4. var stooges = [{name: 'moe', age: 40}, {name: 'curly', age: 60}];
5. \_.sortedIndex(stooges, {name: 'larry', age: 50}, 'age');
6. => 1

## 去除空值和嵌套

\_.compact(array)

返回一个除去所有****false****值的 array副本。 在javascript中, false, null, 0, "", undefined 和 NaN 都是false值.

1. \_.compact([0, 1, false, 2, '', 3]);
2. => [1, 2, 3]

flatten\_.flatten(array, [shallow])

将一个嵌套多层的数组 array（数组） (嵌套可以是任何层数)转换为****只有一层的数组****。 如果你传递 shallow参数，数组将只减少一维的嵌套。

1. \_.flatten([1, [2], [3, [[4]]]]);
2. => [1, 2, 3, 4];
4. \_.flatten([1, [2], [3, [[4]]]], true);
5. => [1, 2, 3, [[4]]];

## 元素位置操作

\_.range([start], stop, [step])

一个用来创建整数灵活编号的列表的函数，便于****each 和 map****循环。

如果省略start则默认为 0；step 默认为 1.返回一个从start 到stop的整数的列表，用step来增加 （或减少）独占。

值得注意的是，如果****stop值在start****前面（也就是stop值小于start值），那么值域会被认为是零长度，而不是负增长。-如果你要一个负数的值域 ，请使用负数step.

1. \_.range(10);
2. => [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
3. \_.range(1, 11);
4. => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
5. \_.range(0, 30, 5);
6. => [0, 5, 10, 15, 20, 25]
7. \_.range(0, -10, -1);
8. => [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
9. \_.range(0);
10. => []

## 元素位置操作

\_.zip(\*arrays)

将 每个arrays中相应位置的值****合并****在一起。在合并分开保存的数据时很有用. 如果你用来处理矩阵嵌套数组时, \_.zip.apply 可以做类似的效果。

1. \_.zip(['moe', 'larry', 'curly'], [30, 40, 50], [true, false, false]);
2. => [["moe", 30, true], ["larry", 40, false], ["curly", 50, false]]

\_.object(list, [values])

将数组转换为对象。传递任何一个单独[key, value]对的列表，或者一个键的列表和一个值得列表。 如果存在重复键，最后一个值将被返回。

1. \_.object(['moe', 'larry', 'curly'], [30, 40, 50]);
2. => {moe: 30, larry: 40, curly: 50}
4. \_.object([['moe', 30], ['larry', 40], ['curly', 50]]);
5. => {moe: 30, larry: 40, curly: 50}

## 绑定函数

bind:

****绑定****函数到对象上, 无论何时函数被调用, 函数里的this都指向对象.

1. var func = function(greeting){ return greeting + ': ' + this.name };
2. func = \_.bind(func, {name : 'moe'}, 'hi');
3. console.log(func());
4. => hi: moe

bindAll:

绑定方法名到对象上, 当这些方法被执行时将在对象的上下文执行. 绑定函数用作事件处理时非常方便, 否则函数调用时 this 关键字根本没什么用.

1. var buttonView = {
2. label : 'underscore',
3. onClick : function(){ console.log('clicked: ' + this.label); },
4. onHover : function(){ console.log('hovering: ' + this.label); }
5. };
6. var func = \_.bindAll(buttonView, 'onClick', 'onHover');
7. func.onClick();
8. => clicked: underscore

## 填充数据

partial:

局部应用一个函数填充在任意个数的****arguments****，不改变其动态this值。和bind方法很相近。你可以传递\_ 给****arguments****列表来指定一个不预先填充，但在调用时提供的参数。

1. var add = function(a, b) { return a + b; };
2. add5 = \_.partial(add, 5);
3. console.log(add5(10));
4. => 15

## 绑定函数

\_.memoize(function, [hashFunction])

Memoizes方法可以缓存某函数的计算结果。对于耗时较长的计算是很有帮助的。

如果传递了 hashFunction 参数，就用****hashFunction**** 的返回值作为key存储函数的计算结果。

****hashFunction**** 默认使用function的第一个参数作为key。memoized值的缓存可作为返回函数的cache属性。

1. var fibonacci = \_.memoize(function(n) {
2. return n < 2 ? n: fibonacci(n - 1) + fibonacci(n - 2);
3. });

## delay 和 defer

delay:

在等待xx毫秒之后调用函数，类似于setTimeout

1. var log = \_.bind(console.log, console);
2. \_.delay(log, 1000, 'sleep 1s');
3. => sleep 1s

defer:

****延迟****调用函数, 直到当前调用栈被清空为止, 跟使用 setTimeout 赋予0毫秒的延时很像。

对执行高消耗算法或大型HTML呈现而不阻碍UI更新线程很有用.

1. \_.defer(function(){ console.log('deferred'); });
2. => deferred

## after 和 before

****after:****

对****循环计数****，只有超过计数，才会调用指定的函数

1. var nums = [1,2,3,4];
2. var renderNums = \_.after(nums.length, function(){
3. console.log('render nums');
4. });
5. \_.each(nums, function(num) {
6. console.log('each:'+num);
7. renderNums();
8. });
9. => each:1
10. each:2
11. each:3
12. each:4
13. render nums

\_.before(count, function)

创建一个函数,调用****不超过****count 次。 当count已经达到时，最后一个函数调用的结果将被记住并返回。

1. var monthlyMeeting = \_.before(3, askForRaise);
2. monthlyMeeting();
3. monthlyMeeting();
4. monthlyMeeting();

## 函数传递

\_.wrap(function, wrapper)

将第一个函数 function 封装到****函数 wrapper**** 里面, 并把函数 function 作为第一个参数传给 wrapper. 这样可以让 wrapper 在 function 运行之前和之后 执行代码, 调整参数然后附有条件地执行.

1. var hello = function(name) { return "hello: " + name; };
2. hello = \_.wrap(hello, function(func) {
3. return "before, " + func("moe") + ", after";
4. });
5. hello();
6. => 'before, hello: moe, after'

## keys 和 values

\_.keys(object)

****检索****object拥有的所有可枚举属性的名称。

1. \_.keys({one: 1, two: 2, three: 3});
2. => ["one", "two", "three"]

\_.allKeys(object)

检索object拥有的和继承的所有属性的名称。

1. function Stooge(name) {
2. this.name = name;
3. }
4. Stooge.prototype.silly = true;
5. \_.allKeys(new Stooge("Moe"));
6. => ["name", "silly"]

\_.values(object)

返回object对象所有的属性值。

1. \_.values({one: 1, two: 2, three: 3});
2. => [1, 2, 3]

## 对象遍历

\_.mapObject(object, iteratee, [context])

它类似于map，但是这用于对象。****转换****每个属性的值。

1. \_.mapObject({start: 5, end: 12}, function(val, key) {
2. return val + 5;
3. });
4. => {start: 10, end: 17}

## 转换和对调

\_.pairs(object)

把一个对象转变为一个****[key, value]形式的数组****。

1. \_.pairs({one: 1, two: 2, three: 3});
2. => [["one", 1], ["two", 2], ["three", 3]]

\_.invert(object)

返回一个object副本，使其****键（keys）和值（values）对换****。对于这个操作，必须确保object里所有的值都是唯一的且可以序列号成字符串.

1. \_.invert({Moe: "Moses", Larry: "Louis", Curly: "Jerome"});
2. => {Moses: "Moe", Louis: "Larry", Jerome: "Curly"};

## 复制

extend:

复制对象的所有属性到目标对象上，****覆盖****已有属性

1. console.log(
2. \_.extend({name : 'moe'}, {age : 50})
3. );
4. => { name: 'moe', age: 50 }

defaults:

****复制****对象的所有属性到目标对象上，跳过已有属性

1. var iceCream = {flavor : "chocolate"};
2. console.log(
3. \_.defaults(iceCream, {flavor : "vanilla", sprinkles : "lots"})
4. );
5. => { flavor: 'chocolate', sprinkles: 'lots' }

## clone

clone:

引入方式****克隆****对象，不进行复制

1. console.log(
2. \_.clone({name : 'moe'});
3. );
4. => {name : 'moe'};

## tag

tag:

用****对象****作为参数来调用函数，作为函数链式调用的一环

1. console.log(
2. \_.chain([1,2,3,200])
3. .filter(function(num) { return num % 2 == 0; })
4. .tap(console.log)
5. .map(function(num) { return num \* num })
6. .value()
7. );
8. => [ 2, 200 ]
9. [ 4, 40000 ]

## 比较

\_.isEqual(object, other)

执行两个对象之间的优化****深度比较****，确定他们是否应被视为****相等****。

1. var stooge = {name: 'moe', luckyNumbers: [13, 27, 34]};
2. var clone = {name: 'moe', luckyNumbers: [13, 27, 34]};
3. stooge == clone;
4. => false
5. \_.isEqual(stooge, clone);
6. => true

## 改变命名空间

****Underscore****默认使用\_（下划线）来访问和创建对象，但这个名字可能不符合我们的命名规范，或容易引起命名冲突。

我们可以通过noConflict()方法来改变Underscore的命名，并恢复\_（下划线）变量之前的值，例如：

1. // Underscore对象
2. console.dir(\_);
3. // 将Underscore对象重命名为us, 后面都通过us来访问和创建Underscore对象
4. var us = \_.noConflict();
5. // 输出"自定义变量"
6. console.dir(\_);

## 随机数

\_.random(min, max)

返回一个min 和 max之间的随机整数。如果你只传递一个参数，那么将返回0和这个参数之间的整数。

1. \_.random(0, 100);
2. => 42

## 唯一ID

\_.uniqueId([prefix])

为需要的客户端模型或DOM元素生成一个全局唯一的id。如果prefix参数存在， id 将附加给它。

1. \_.uniqueId('contact\_');
2. => 'contact\_104'

## 扩展

允许用您自己的实用程序函数扩展****Underscore****。传递一个 {name: function}定义的哈希添加到****Underscore对象****，以及面向对象封装。

1. \_.mixin({
2. capitalize : function(string) {
3. return string.charAt(0).toUpperCase() + string.substring(1).toLowerCase();
4. }
5. });
6. console.log(\_("fabio").capitalize());
7. => Fabio

## 返回属性值

\_.result(object, property, [defaultValue])

如果指定的 property 的值是一个函数，那么将在object上下文内调用它;否则，返回它。

如果提供默认值，并且属性不存在，那么默认值将被返回。如果设置defaultValue是一个函数，它的结果将被返回。

1. var object = {cheese: 'crumpets', stuff: function(){ return 'nonsense'; }};
2. \_.result(object, 'cheese');
3. => "crumpets"
4. \_.result(object, 'stuff');
5. => "nonsense"
6. \_.result(object, 'meat', 'ham');
7. => "ham"