UNDERSCORE.JS

如果您有任何建議，或者拍磚，歡迎在微博上@愚人碼頭 聯繫我。

本文檔為Underscore.js (1.8.3) 中文文檔，
查看1.8.2版本的文檔請點擊:http://www.css88.com/doc/underscore1.8.2/
查看1.7.0版本的文檔請點擊:http://www.css88.com/doc/underscore1.7.0/
查看1.6.0版本的文檔請點擊:http://www.css88.com/doc/underscore1.6.0/
查看1.5.2版本的文檔請點擊:http://www.css88.com/doc/underscore1.5.2/

其他前端相關文檔：jQuery API中文文檔 、jQuery UI API中文文檔 、Zepto.js API 中文版

Underscore一個JavaScript實用庫，提供了一整套函數式編程的實用功能，但是沒有擴展任何JavaScript內置對象。它是這個問題的答案：「如果我在一個空白的HTML頁面前坐下， 並希望立即開始工作， 我需要什麼？「...它彌補了部分jQuery沒有實現的功能,同時又是Backbone.js必不可少的部分。（感謝@小鄧子daj 的翻譯建議）

Underscore提供了100多個函數,包括常用的: **map**, **filter**, **invoke** — 當然還有更多專業的輔助函數,如:函數綁定, JavaScript模板功能,創建快速索引, 強類型相等測試, 等等.

為了你能仔細研讀，這裡包含了一個完整的測試套件。

您也可以通過註釋閱讀源代碼。

享受Underscore，並希望獲得更多的使用功能（感謝@Jaward華仔 的翻譯建議），可以嘗試使用Underscore-contrib（愚人碼頭註：Underscore-contrib是一個Underscore的代碼貢獻庫）。

該項目代碼託管在GitHub上，你可以通過issues頁、Freenode的 `#documentcloud` 頻道、發送tweets給@documentcloud三個途徑報告bug以及參與特性討論。

Underscore是DocumentCloud的一個開源組件。

下載 *(右鍵另存為)*

開發版 (1.8.2)	*51kb, 未壓縮版, 含大量註釋*
生產版 (1.8.2)	*5.7kb, 最簡化並用Gzip壓縮* (Source Map)
不穩定版	*未發佈版本, 當前開發中的* `master` *分支, 如果使用此版本, 風險自負*

安裝（Installation）

- **Node.js** `npm install underscore`
- **Meteor.js** `meteor add underscore`
- **Require.js** `require(["underscore"], ...`
- **Bower** `bower install underscore`
- **Component** `component install jashkenas/underscore`

集合函數 (數組 或對象)

2017-08-19 更新，感謝 @愛吹牛逼的王小白 提供的翻譯建議；

each `_.each(list, iteratee, [context])` *Alias: **forEach***

遍歷**list**中的所有元素，按順序用每個元素當做參數調用 **iteratee** 函數。如果傳遞了**context**參數，則把**iteratee**綁定到**context**對象上。每次調用**iteratee**都會傳遞三個參數：`(element, index, list)`。如果**list**是個JavaScript對象，**iteratee**的參數是 `(value, key, list)`)。返回**list**以方便鏈式調用。

```
_.each([1, 2, 3], alert);
=> alerts each number in turn...
_.each({one: 1, two: 2, three: 3}, alert);
=> alerts each number value in turn...
```

注意：集合函數能在數組，對象，和類數組對象，比如 `arguments`, `NodeList` *和類似的數據類型上正常工作。 但是它通過鴨子類型工作，所以要避免傳遞帶有一個數值類型* `length` *屬性的對象。每個循環不能被破壞 - 打破， 使用**_.find**代替，這也是很好的注意。*

map `_.map(list, iteratee, [context])` *Alias: **collect***

通過對list裡的每個元素調用轉換函數(iteratee迭代器)生成一個與之相對應的數組。`iteratee` 傳遞三個參數：`value`，然後是迭代 `index`(或 `key` 愚人碼頭註：如果**list**是個JavaScript對象是，這個參數就是`key`)，最後一個是引用指向整個 `list`。

```
_.map([1, 2, 3], function(num){ return num * 3; });
=> [3, 6, 9]
_.map({one: 1, two: 2, three: 3}, function(num, key){ return num * 3; });
=> [3, 6, 9]
_.map([[1, 2], [3, 4]], _.first);
=> [1, 3]
```

reduce `_.reduce(list, iteratee, [memo], [context])` *Aliases: **inject, foldl***

別名為 **inject** 和 **foldl**, **reduce**方法把**list**中元素歸結為一個單獨的數值。**Memo**是reduce函數的初始值，會被每一次成功調用**iteratee**函數的返回值所取代 。這個迭代傳遞4個參數：`memo`, `value` 和 迭代的 `index`（或者 key）和最後一個引用的整個 `list`。

如果沒有**memo**傳遞給**reduce**的初始調用，**iteratee**不會被列表中的第一個元素調用。第一個元素將取代**memo**參數傳遞給列表中下一個元素調用的**iteratee**函數。

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

reduceRight `_.reduceRight(list, iteratee, memo, [context])` *Alias: **foldr***

reducRight是從右側開始組合元素的**reduce**函數， **Foldr**在 JavaScript 中不像其它有惰性求值的語言那麼有用（愚人碼頭註：lazy evaluation：一種求值策略，只有當表達式的值真正需要時才對表達式進行計算）。

```
var list = [[0, 1], [2, 3], [4, 5]];
var flat = _.reduceRight(list, function(a, b) { return a.concat(b); }, []);
=> [4, 5, 2, 3, 0, 1]
```

find `_.find(list, predicate, [context])` *Alias: **detect***

在**list**中逐項查找，返回第一個通過**predicate**迭代函數真值檢測的元素值，如果沒有元素通過檢測則返回 `undefined`。 如果找到匹配的元素，函數將立即返回，不會遍歷整個list。

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

filter `_.filter(list, predicate, [context])` *Alias:* ***select***

遍歷**list**中的每個值，返回所有通過**predicate**真值檢測的元素所組成的數組。（愚人碼頭註：如果存在原生**filter**方法，則用原生的**filter**方法。）

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

where `_.where(list, properties)`

遍歷**list**中的每一個值，返回一個數組，這個數組裡的元素包含 **properties** 所列出的鍵 - 值對。

```
_.where(listOfPlays, {author: "Shakespeare", year: 1611});
=> [{title: "Cymbeline", author: "Shakespeare", year: 1611},
    {title: "The Tempest", author: "Shakespeare", year: 1611}]
```

findWhere `_.findWhere(list, properties)`

遍歷整個**list**，返回匹配 **properties**參數所列出的所有 鍵 - 值 對的第一個值。

如果沒有找到匹配的屬性，或者**list**是空的，那麼將返回*undefined*。

```
_.findWhere(publicServicePulitzers, {newsroom: "The New York Times"});
=> {year: 1918, newsroom: "The New York Times",
  reason: "For its public service in publishing in full so many official reports,
  documents and speeches by European statesmen relating to the progress and
  conduct of the war."}
```

reject `_.reject(list, predicate, [context])`

返回**list**中沒有通過**predicate**真值檢測的元素集合，與**filter**相反。

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

every `_.every(list, [predicate], [context])` *Alias:* ***all***

如果**list**中的所有元素都通過**predicate**的真值檢測就返回*true*。（愚人碼頭註：如果存在原生的**every**方法，就使用原生的**every**。）

```
_.every([true, 1, null, 'yes'], _.identity);
=> false
```

some `_.some(list, [predicate], [context])` *Alias:* ***any***

如果**list**中有任何一個元素通過 **predicate** 的真值檢測就返回*true*。一旦找到了符合條件的元素, 就直接中斷對list的遍歷.（愚人碼頭註：如果存在原生的**some**方法，就使用原生的**some**。）

```
_.some([null, 0, 'yes', false]);
=> true
```

contains `_.contains(list, value, [fromIndex])` *Alias:* ***includes***

如果**list**包含指定的**value**則返回*true*（愚人碼頭註：使用===檢測）。如果**list** 是數組，內部使用**indexOf**判斷。使用**fromIndex**來給定開始檢索的索引位置。

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

invoke `_.invoke(list, methodName, *arguments)`

在**list**的每個元素上執行**methodName**方法。 任何傳遞給**invoke**的額外參數，**invoke**都會在調用**methodName**方法的時候傳遞給它。

```
_.invoke([[5, 1, 7], [3, 2, 1]], 'sort');
=> [[1, 5, 7], [1, 2, 3]]
```

pluck `_.pluck(list, propertyName)`

pluck也許是**map**最常使用的用例模型的簡化版本，即萃取數組對象中某屬性值，返回一個數組。

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

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

返回**list**中的最大值。如果傳遞**iteratee**參數，**iteratee**將作為**list**中每個值的排序依據。如果**list**為空，將返回*-Infinity*，所以你可能需要事先用isEmpty檢查 **list** 。

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

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

返回**list**中的最小值。如果傳遞**iteratee**參數，**iteratee**將作為**list**中每個值的排序依據。如果**list**為空，將返回*Infinity*，所以你可能需要事先用isEmpty檢查 **list** 。

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

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

返回一個排序後的**list**拷貝副本。如果傳遞**iteratee**參數，**iteratee**將作為**list**中每個值的排序依據。用來進行排序迭代器也可以是屬性名稱的字符串(比如 `length`)。

```
_.sortBy([1, 2, 3, 4, 5, 6], function(num){ return Math.sin(num); });
=> [5, 4, 6, 3, 1, 2]

var stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];
_.sortBy(stooges, 'name');
=> [{name: 'curly', age: 60}, {name: 'larry', age: 50}, {name: 'moe', age: 40}];
```

groupBy `_.groupBy(list, iteratee, [context])`

把一個集合分組為多個集合，通過 **iterator** 返回的結果進行分組. 如果 **iterator** 是一個字符串而不是函數, 那麼將使用 **iterator** 作為各元素的屬性名來對比進行分組.

```
_.groupBy([1.3, 2.1, 2.4], function(num){ return Math.floor(num); });
=> {1: [1.3], 2: [2.1, 2.4]}

_.groupBy(['one', 'two', 'three'], 'length');
=> {3: ["one", "two"], 5: ["three"]}
```

indexBy `_.indexBy(list, iteratee, [context])`

給定一個**list**，和 一個用來返回一個在列表中的每個元素鍵 的**iterator** 函數（或屬性名）， 返回一個每一項索引的對象。和groupBy非常像，但是當你知道你的鍵是唯一的時候可以使用**indexBy** 。

```
var stooges = [{name: 'moe', age: 40}, {name: 'larry', age: 50}, {name: 'curly', age: 60}];
_.indexBy(stooges, 'age');
=> {
  "40": {name: 'moe', age: 40},
  "50": {name: 'larry', age: 50},
  "60": {name: 'curly', age: 60}
}
```

countBy `_.countBy(list, iteratee, [context])`

排排序一個列表組成多個組，並且返回各組中的對象的數量的計數。類似 `groupBy` ，但是不是返回列表的值，而是返回在該組中值的數目。

```
_.countBy([1, 2, 3, 4, 5], function(num) {
  return num % 2 == 0 ? 'even': 'odd';
});
=> {odd: 3, even: 2}
```

shuffle `_.shuffle(list)`

返回一個隨機亂序的 **list** 副本, 使用 Fisher-Yates shuffle 來進行隨機亂序.

```
_.shuffle([1, 2, 3, 4, 5, 6]);
=> [4, 1, 6, 3, 5, 2]
```

sample `_.sample(list, [n])`

從 **list**中產生一個隨機樣本。傳遞一個數字表示從**list**中返回**n**個隨機元素。否則將返回一個單一的隨機項。

```
_.sample([1, 2, 3, 4, 5, 6]);
=> 4

_.sample([1, 2, 3, 4, 5, 6], 3);
=> [1, 6, 2]
```

toArray `_.toArray(list)`

把**list**(任何可以迭代的對象)轉換成一個數組，在轉換 **arguments** 對象時非常有用。

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

size `_.size(list)`

返回**list**的長度。

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

partition `_.partition(array, predicate)`

拆分一個數組（**array**）為兩個數組： 第一個數組其元素都滿足**predicate**迭代函數， 而第二個的所有元素均不能滿足**predicate**迭代函數。

```
_.partition([0, 1, 2, 3, 4, 5], isOdd);
=> [[1, 3, 5], [0, 2, 4]]
```

數組函數（Array Functions）

註：所有的數組函數也可以用於 ***arguments*** *(參數)對象。 但是，Underscore 函數不能用於稀疏（"sparse"）數組。*

first `_.first(array, [n])` *Alias: **head**, **take***

返回**array**（數組）的第一個元素。傳遞 **n**參數將返回數組中從第一個元素開始的**n**個元素（愚人碼頭註：返回數組中前 **n** 個元素.）。

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

initial `_.initial(array, [n])`

返回數組中除了最後一個元素外的其他全部元素。 在arguments對象上特別有用。傳遞 **n**參數將從結果中排除從最後一個開始的**n**個元素（愚人碼頭註：排除數組後面的 **n** 個元素）。

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

last `_.last(array, [n])`

返回**array**（數組）中最後一個元素。傳遞 **n**參數將返回數組中從最後一個元素開始的**n**個元素（愚人碼頭註：返回數組裡的後面的**n**個元素）。

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

rest `_.rest(array, [index])` *Alias: **tail, drop***

返回數組中除了第一個元素外的其他全部元素。傳遞 **index** 參數將返回從**index**開始的剩餘所有元素 。（感謝@德德德德撸 指出錯誤）

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

compact `_.compact(array)`

返回一個除去了所有*false*值的 **array**副本。 在javascript中, *false*, *null*, *0*, *""*, *undefined* 和 *NaN* 都是*false*值.

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

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

將一個嵌套多層的數組 **array**（數組）(嵌套可以是任何層數)轉換為只有一層的數組。 如果你傳遞 **shallow**參數，數組將只減少一維的嵌套。

```
_.flatten([1, [2], [3, [[4]]]]);
=> [1, 2, 3, 4];

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

without `_.without(array, *values)`

返回一個刪除所有**values**值後的 **array**副本。（愚人碼頭註：使用**===**表達式做相等測試。）

```
_.without([1, 2, 1, 0, 3, 1, 4], 0, 1);
=> [2, 3, 4]
```

union `_.union(*arrays)`

返回傳入的 **arrays**（數組）並集：按順序返回，返回數組的元素是唯一的，可以傳入一個或多個 **arrays** （數組）。

```
_.union([1, 2, 3], [101, 2, 1, 10], [2, 1]);
=> [1, 2, 3, 101, 10]
```

intersection `_.intersection(*arrays)`

返回傳入 **arrays**（數組）交集。結果中的每個值是存在於傳入的每個**arrays**（數組）裡。

```
_.intersection([1, 2, 3], [101, 2, 1, 10], [2, 1]);
=> [1, 2]
```

difference `_.difference(array, *others)`

類似於**without**，但返回的值來自**array**參數數組，並且不存在於**other** 數組。

```
_.difference([1, 2, 3, 4, 5], [5, 2, 10]);
=> [1, 3, 4]
```

uniq `_.uniq(array, [isSorted], [iteratee])` *Alias:* ***unique***

返回 **array**去重後的副本, 使用 **===** 做相等測試. 如果您確定 **array** 已經排序, 那麼給 **isSorted** 參數傳遞 *true*值, 此函數將運行的更快的算法. 如果要處理對象元素, 傳遞 **iteratee**函數來獲取要對比的屬性.

```
_.uniq([1, 2, 1, 3, 1, 4]);
=> [1, 2, 3, 4]
```

zip `_.zip(*arrays)`

將 每個**arrays**中相應位置的值合併在一起。在合併分開保存的數據時很有用. 如果你用來處理矩陣嵌套數組時, `_.zip.apply` 可以做類似的效果。

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

unzip `_.unzip(*arrays)`

與zip功能相反的函數，給定若干**arrays**，返回一串聯的新數組，其第一元素個包含所有的輸入數組的第一元素，其第二包含了所有的第二元素，依此類推。通過 `apply` 用於傳遞數組的數組。（感謝 @周文彬1986、 @未定的終點 指出示例錯誤）

```
_.unzip([['moe', 'larry', 'curly'], [30, 40, 50], [true, false, false]])
=> ["moe", 30, true], ["larry", 40, false], ["curly", 50, false]
```

object `_.object(list, [values])`

將數組轉換為對象。傳遞任何一個單獨 `[key, value]` 對的列表，或者一個鍵的列表和一個值得列表。 如果存在重複鍵，最後一個值將被返回。

```
_.object(['moe', 'larry', 'curly'], [30, 40, 50]);
=> {moe: 30, larry: 40, curly: 50}

_.object([['moe', 30], ['larry', 40], ['curly', 50]]);
=> {moe: 30, larry: 40, curly: 50}
```

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

返回**value**在該 **array** 中的索引值，如果**value**不存在 **array**中就返回*-1*。使用原生的**indexOf** 函數，除非它失效。如果您正在使用一個大數組，你知道數組已經排序，傳遞 `true` 給**isSorted**將更快的用二進制搜索..,或者，傳遞一個數字作為第三個參數，為了在給定的索引的數組中尋找第一個匹配值。

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

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

返回**value**在該 **array** 中的從最後開始的索引值，如果**value**不存在 **array**中就返回*-1*。如果支持原生的**lastIndexOf**，將使用原生的**lastIndexOf**函數。傳遞**fromIndex**將從你給定的索性值開始搜索。

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

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

使用二分查找確定**value**在**list**中的位置序號，**value**按此序號插入能保持**list**原有的排序。如果提供**iterator**函數，**iterator**將作為list排序的依據，包括你傳遞的**value** 。**iterator**也可以是字符串的屬性名用來排序(比如 `length`)。

```
_.sortedIndex([10, 20, 30, 40, 50], 35);
=> 3

var stooges = [{name: 'moe', age: 40}, {name: 'curly', age: 60}];
_.sortedIndex(stooges, {name: 'larry', age: 50}, 'age');
=> 1
```

findIndex `_.findIndex(array, predicate, [context])`

類似於 `_.indexOf` ，當**predicate**通過真檢查時，返回第一個索引值；否則返回*-1*。

```
_.findIndex([4, 6, 8, 12], isPrime);
=> -1 // not found
_.findIndex([4, 6, 7, 12], isPrime);
=> 2
```

findLastIndex `_.findLastIndex(array, predicate, [context])`

和 `_.findIndex` 類似，但反向迭代數組，當**predicate**通過真檢查時，最接近末端的索引值將被返回。

```
var users = [{'id': 1, 'name': 'Bob', 'last': 'Brown'},
             {'id': 2, 'name': 'Ted', 'last': 'White'},
             {'id': 3, 'name': 'Frank', 'last': 'James'},
             {'id': 4, 'name': 'Ted', 'last': 'Jones'}];
_.findLastIndex(users, {
  name: 'Ted'
});
=> 3
```

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

一個用來創建整數靈活編號的列表的函數，便於 `each` 和 `map` 循環。如果省略**start**則默認為 *0*；**step** 默認為 *1*.返回一個從**start** 到**stop**的整數的列表，用**step**來增加 （或減少）獨佔。值得注意的是，如果**stop**值在**start**前面（也就是**stop**值小於**start**值），那麼值域會被認為是零長度，而不是負增長。-如果你要一個負數的值域 ，請使用負數**step**.

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

與函數有關的函數（Function (uh, ahem) Functions）

bind `_.bind(function, object, *arguments)`

綁定函數 **function** 到對象 **object** 上, 也就是無論何時調用函數, 函數裡的 *this* 都指向這個 **object**.任意可選參數 **arguments** 可以傳遞給函數 **function** , 可以填充函數所需要的參數,這也被稱為 **partial application**。對於沒有結合上下文的**partial application**綁定，請使用partial。
(愚人碼頭註：partial application翻譯成「部分應用」或者「偏函數應用」。partial application可以被描述為一個函數，它接受一定數目的參數，綁定值到一個或多個這些參數，並返回一個新的函數，這個返回函數只接受剩餘未綁定值的參數。參見：http://en.wikipedia.org/wiki/Partial_application。感謝@一任風月憶秋年的建議)。

```
var func = function(greeting){ return greeting + ': ' + this.name };
func = _.bind(func, {name: 'moe'}, 'hi');
func();
=> 'hi: moe'
```

bindAll `_.bindAll(object, *methodNames)`

把**methodNames**參數指定的一些方法綁定到**object**上，這些方法就會在對象的上下文環境中執行。綁定函數用作事件處理函數時非常便利，否則函數被調用時*this*一點用也沒有。**methodNames**參數是必須的。

```
var buttonView = {
  label  : 'underscore',
  onClick: function(){ alert('clicked: ' + this.label); },
  onHover: function(){ console.log('hovering: ' + this.label); }
};
_.bindAll(buttonView, 'onClick', 'onHover');
// When the button is clicked, this.label will have the correct value.
jQuery('#underscore_button').bind('click', buttonView.onClick);
```

partial `_.partial(function, *arguments)`

局部應用一個函數填充在任意個數的 **arguments**，*不*改變其動態 `this` 值。和bind方法很相近。你可以傳遞 `_` 給**arguments**列表來指定一個不預先填充，但在調用時提供的參數。

```
var subtract = function(a, b) { return b - a; };
sub5 = _.partial(subtract, 5);
sub5(20);
=> 15

// Using a placeholder
subFrom20 = _.partial(subtract, _, 20);
```

```
subFrom20(5);
=> 15
```

memoize `_.memoize(function, [hashFunction])`

Memoizes方法可以緩存某函數的計算結果。對於耗時較長的計算是很有幫助的。如果傳遞了 **hashFunction** 參數，就用 **hashFunction** 的返回值作為key存儲函數的計算結果。**hashFunction** 默認使用function的第一個參數作為key。memoized值的緩存可作為返回函數的 `cache` 屬性。

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

delay `_.delay(function, wait, *arguments)`

類似**setTimeout**，等待**wait**毫秒後調用**function**。如果傳遞可選的參數**arguments**，當函數**function**執行時， **arguments** 會作為參數傳入。

```
var log = _.bind(console.log, console);
_.delay(log, 1000, 'logged later');
=> 'logged later' // Appears after one second.
```

defer `_.defer(function, *arguments)`

延遲調用**function**直到當前調用棧清空為止，類似使用延時為0的**setTimeout**方法。對於執行開銷大的計算和無阻塞UI線程的HTML渲染時候非常有用。 如果傳遞**arguments**參數，當函數**function**執行時， **arguments** 會作為參數傳入。

```
_.defer(function(){ alert('deferred'); });
// Returns from the function before the alert runs.
```

throttle `_.throttle(function, wait, [options])`

創建並返回一個像節流閥一樣的函數，當重複調用函數的時候，至少每隔 **wait**毫秒調用一次該函數。對於想控制一些觸發頻率較高的事件有幫助。（愚人碼頭註：詳見：javascript函數的throttle和debounce，感謝 @澳利澳先生 的翻譯建議）

默認情況下，**throttle**將在你調用的第一時間盡快執行這個**function**，並且，如果你在**wait**週期內調用任意次數的函數，都將盡快的被覆蓋。如果你想禁用第一次首先執行的話，傳遞 `{leading: false}` ，還有如果你想禁用最後一次執行的話，傳遞 `{trailing: false}` 。

```
var throttled = _.throttle(updatePosition, 100);
$(window).scroll(throttled);
```

debounce `_.debounce(function, wait, [immediate])`

返回 **function** 函數的防反跳版本, 將延遲函數的執行(真正的執行)在函數最後一次調用時刻的 **wait** 毫秒之後. 對於必須在一些輸入（多是一些用戶操作）停止到達*之後*執行的行為有幫助。 例如: 渲染一個Markdown格式的評論預覽, 當窗口停止改變大小之後重新計算佈局, 等等.

傳參 **immediate** 為 `true` ， **debounce**會在 **wait** 時間間隔的開始調用這個函數 。（愚人碼頭註：並且在 waite 的時間之內，不會再次調用。）在類似不小心點了提交按鈕兩下而提交了兩次的情況下很有用。（感謝 @ProgramKid 的翻譯建議）

```
var lazyLayout = _.debounce(calculateLayout, 300);
$(window).resize(lazyLayout);
```

once `_.once(function)`

創建一個只能調用一次的函數。重複調用改進的方法也沒有效果，只會返回第一次執行時的結果。 作為初始化函數使用時非常有用, 不用再設一個boolean值來檢查是否已經初始化完成.

```
var initialize = _.once(createApplication);
initialize();
initialize();
// Application is only created once.
```

after `_.after(count, function)`

創建一個函數, 只有在運行了 **count** 次之後才有效果. 在處理同組異步請求返回結果時, 如果你要確保同組裡所有異步請求完成之後才 執行這個函數, 這將非常有用。

```
var renderNotes = _.after(notes.length, render);
_.each(notes, function(note) {
  note.asyncSave({success: renderNotes});
});
// renderNotes is run once, after all notes have saved.
```

before `_.before(count, function)`

創建一個函數,調用不超過**count** 次。 當**count**已經達到時，最後一個函數調用的結果將被記住並返回。

```
var monthlyMeeting = _.before(3, askForRaise);
monthlyMeeting();
monthlyMeeting();
monthlyMeeting();
// the result of any subsequent calls is the same as the second call
```

wrap `_.wrap(function, wrapper)`

將第一個函數 **function** 封裝到函數 **wrapper** 裡面, 並把函數 **function** 作為第一個參數傳給 **wrapper**. 這樣可以讓 **wrapper** 在 **function** 運行之前和之後 執行代碼, 調整參數然後附有條件地執行.

```
var hello = function(name) { return "hello: " + name; };
hello = _.wrap(hello, function(func) {
  return "before, " + func("moe") + ", after";
});
hello();
=> 'before, hello: moe, after'
```

negate `_.negate(predicate)`

返回一個新的**predicate**函數的否定版本。

```
var isFalsy = _.negate(Boolean);
_.find([-2, -1, 0, 1, 2], isFalsy);
=> 0
```

compose `_.compose(*functions)`

返回函數集 **functions** 組合後的復合函數, 也就是一個函數執行完之後把返回的結果再作為參數賦給下一個函數來執行. 以此類推. 在數學裡, 把函數 *f()*, *g()*, 和 *h()* 組合起來可以得到復合函數 *f(g(h()))*。

```
var greet    = function(name){ return "hi: " + name; };
var exclaim  = function(statement){ return statement.toUpperCase() + "!"; };
var welcome = _.compose(greet, exclaim);
```

```
welcome('moe');
=> 'hi: MOE!'
```

對象函數（Object Functions）

keys `_.keys(object)`

檢索**object**擁有的所有可枚舉屬性的名稱。

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

allKeys `_.allKeys(object)`

檢索**object**擁有的和繼承的所有屬性的名稱。

```
function Stooge(name) {
  this.name = name;
}
Stooge.prototype.silly = true;
_.allKeys(new Stooge("Moe"));
=> ["name", "silly"]
```

values `_.values(object)`

返回**object**對象所有的屬性值。

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

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

它類似於map，但是這用於對象。轉換每個屬性的值。

```
_.mapObject({start: 5, end: 12}, function(val, key) {
  return val + 5;
});
=> {start: 10, end: 17}
```

pairs `_.pairs(object)`

把一個對象轉變為一個`[key, value]`形式的數組。

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

invert `_.invert(object)`

返回一個**object**副本，使其鍵（keys）和值（values）對換。對於這個操作，必須確保object裡所有的值都是唯一的且可以序列號成字符串.

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

create `_.create(prototype, props)`

創建具有給定原型的新對象， 可選附加**props** 作為 *own*的屬性。 基本上，和 `Object.create` 一樣， 但是沒有所有的屬性描述符。

```
var moe = _.create(Stooge.prototype, {name: "Moe"});
```

functions `_.functions(object)` *Alias:* ***methods***

返回一個對象裡所有的方法名, 而且是已經排序的 — 也就是說, 對象裡每個方法(屬性值是一個函數)的名稱.

```
_.functions(_);
=> ["all", "any", "bind", "bindAll", "clone", "compact", "compose" ...
```

findKey `_.findKey(object, predicate, [context])`

Similar to `_.findIndex` but for keys in objects. Returns the *key* where the **predicate** truth test passes or *undefined*.

extend `_.extend(destination, *sources)`

複製**source**對象中的所有屬性覆蓋到**destination**對象上，並且返回 **destination** 對象. 複製是按順序的, 所以後面的對象屬性會把前面的對象屬性覆蓋掉(如果有重複).

```
_.extend({name: 'moe'}, {age: 50});
=> {name: 'moe', age: 50}
```

extendOwn `_.extendOwn(destination, *sources)` *Alias:* ***assign***

類似於 **extend**, 但只複製*自己的*屬性覆蓋到目標對象。（愚人碼頭註：不包括繼承過來的屬性）

pick `_.pick(object, *keys)`

返回一個**object**副本，只過濾出**keys**(有效的鍵組成的數組)參數指定的屬性值。或者接受一個判斷函數，指定挑選哪個key。

```
_.pick({name: 'moe', age: 50, userid: 'moe1'}, 'name', 'age');
=> {name: 'moe', age: 50}
_.pick({name: 'moe', age: 50, userid: 'moe1'}, function(value, key, object) {
  return _.isNumber(value);
});
=> {age: 50}
```

omit `_.omit(object, *keys)`

返回一個**object**副本，只過濾出除去**keys**(有效的鍵組成的數組)參數指定的屬性值。 或者接受一個判斷函數，指定忽略哪個key。

```
_.omit({name: 'moe', age: 50, userid: 'moe1'}, 'userid');
=> {name: 'moe', age: 50}
_.omit({name: 'moe', age: 50, userid: 'moe1'}, function(value, key, object) {
  return _.isNumber(value);
});
=> {name: 'moe', userid: 'moe1'}
```

defaults `_.defaults(object, *defaults)`

用**defaults**對象填充**object** 中的 `undefined` 屬性。 並且返回這個**object**。一旦這個屬性被填充，再使用defaults方法將不會有任何效果。（感謝@一任風月憶秋年的拍磚）

```
var iceCream = {flavor: "chocolate"};
_.defaults(iceCream, {flavor: "vanilla", sprinkles: "lots"});
=> {flavor: "chocolate", sprinkles: "lots"}
```

clone `_.clone(object)`

創建 一個淺複製（淺拷貝）的克隆**object**。任何嵌套的對象或數組都通過引用拷貝，不會複製。

```
_.clone({name: 'moe'});
=> {name: 'moe'};
```

tap `_.tap(object, interceptor)`

用 **object**作為參數來調用函數**interceptor**，然後返回**object**。這種方法的主要意圖是作為函數鏈式調用 的一環, 為了對此對象執行操作並返回對象本身。

```
_.chain([1,2,3,200])
  .filter(function(num) { return num % 2 == 0; })
  .tap(alert)
  .map(function(num) { return num * num })
  .value();
=> // [2, 200] (alerted)
=> [4, 40000]
```

has `_.has(object, key)`

對象是否包含給定的鍵嗎？等同於 `object.hasOwnProperty(key)`，但是使用 `hasOwnProperty` 函數的一個安全引用，以防意外覆蓋。

```
_.has({a: 1, b: 2, c: 3}, "b");
=> true
```

property `_.property(key)`

返回一個函數，這個函數返回任何傳入的對象的 `key` 屬性。

```
var stooge = {name: 'moe'};
'moe' === _.property('name')(stooge);
=> true
```

propertyOf `_.propertyOf(object)`

和 `_.property` 相反。需要一個對象，並返回一個函數,這個函數將返回一個提供的屬性的值。

```
var stooge = {name: 'moe'};
_.propertyOf(stooge)('name');
=> 'moe'
```

matcher `_.matcher(attrs)`

返回一個斷言函數，這個函數會給你一個斷言可以用來辨別給定的對象是否匹配**attrs**指定鍵/值屬性。

```
var ready = _.matcher({selected: true, visible: true});
var readyToGoList = _.filter(list, ready);
```

isEqual `_.isEqual(object, other)`

執行兩個對象之間的優化深度比較，確定他們是否應被視為相等。

```
var stooge = {name: 'moe', luckyNumbers: [13, 27, 34]};
var clone  = {name: 'moe', luckyNumbers: [13, 27, 34]};
stooge == clone;
=> false
_.isEqual(stooge, clone);
=> true
```

isMatch `_.isMatch(object, properties)`

告訴你**properties**中的鍵和值是否包含在**object**中。

```
var stooge = {name: 'moe', age: 32};
_.isMatch(stooge, {age: 32});
=> true
```

isEmpty `_.isEmpty(object)`

如果**object** 不包含任何值(沒有可枚舉的屬性)，返回*true*。 對於字符串和類數組（array-like）對象，如果length屬性為0，那麼 `_.isEmpty` 檢查返回*true*。

```
_.isEmpty([1, 2, 3]);
=> false
_.isEmpty({});
=> true
```

isElement `_.isElement(object)`

如果**object**是一個DOM元素，返回*true*。

```
_.isElement(jQuery('body')[0]);
=> true
```

isArray `_.isArray(object)`

如果**object**是一個數組，返回*true*。

```
(function(){ return _.isArray(arguments); })();
=> false
_.isArray([1,2,3]);
=> true
```

isObject `_.isObject(value)`

如果**object**是一個對象，返回*true*。需要注意的是JavaScript數組和函數是對象，字符串和數字不是。

```
_.isObject({});
=> true
_.isObject(1);
=> false
```

isArguments `_.isArguments(object)`

如果**object**是一個參數對象，返回*true*。

```
(function(){ return _.isArguments(arguments); })(1, 2, 3);
=> true
_.isArguments([1,2,3]);
=> false
```

isFunction `_.isFunction(object)`

如果**object**是一個函數（Function），返回*true*。

```
_.isFunction(alert);
=> true
```

isString `_.isString(object)`

如果**object**是一個字符串，返回*true*。

```
_.isString("moe");
=> true
```

isNumber `_.isNumber(object)`

如果**object**是一個數值，返回*true* (包括 `NaN`)。

```
_.isNumber(8.4 * 5);
=> true
```

isFinite `_.isFinite(object)`

如果**object**是一個有限的數字，返回*true*。

```
_.isFinite(-101);
=> true

_.isFinite(-Infinity);
=> false
```

isBoolean `_.isBoolean(object)`

如果**object**是一個布爾值，返回*true*，否則返回*false*。

```
_.isBoolean(null);
=> false
```

isDate `_.isDate(object)`

Returns *true* if **object** is a Date.

```
_.isDate(new Date());
=> true
```

isRegExp `_.isRegExp(object)`

如果**object**是一個正則表達式，返回*true*。

```
_.isRegExp(/moe/);
=> true
```

isError `_.isError(object)`

如果**object**繼承至 Error 對象，那麼返回 *true*。

```
try {
  throw new TypeError("Example");
```

```
} catch (o_O) {
  _.isError(o_O)
}
=> true
```

isNaN `_.isNaN(object)`

如果**object**是 *NaN*，返回*true*。
注意： 這和原生的**isNaN** 函數不一樣，如果變量是*undefined*，原生的**isNaN** 函數也會返回 *true* 。

```
_.isNaN(NaN);
=> true
isNaN(undefined);
=> true
_.isNaN(undefined);
=> false
```

isNull `_.isNull(object)`

如果**object**的值是 *null*，返回*true*。

```
_.isNull(null);
=> true
_.isNull(undefined);
=> false
```

isUndefined `_.isUndefined(value)`

如果**value**是*undefined*，返回*true*。

```
_.isUndefined(window.missingVariable);
=> true
```

實用功能(Utility Functions)

noConflict `_.noConflict()`

放棄**Underscore** 的控制變量"_"。返回**Underscore** 對象的引用。

```
var underscore = _.noConflict();
```

identity `_.identity(value)`

返回與傳入參數相等的值. 相當於數學裡的: `f(x) = x`
這個函數看似無用, 但是在Underscore裡被用作默認的迭代器iterator.

```
var stooge = {name: 'moe'};
stooge === _.identity(stooge);
=> true
```

constant `_.constant(value)`

創建一個函數，這個函數 返回相同的值 用來作為 `_.constant` 的參數。

```
var stooge = {name: 'moe'};
stooge === _.constant(stooge)();
=> true
```

noop `_.noop()`

返回 `undefined` ，不論傳遞給它的是什麼參數。 可以用作默認可選的回調參數。

```
obj.initialize = _.noop;
```

times `_.times(n, iteratee, [context])`

調用給定的迭代函數**n**次,每一次調用**iteratee**傳遞 `index` 參數。生成一個返回值的數組。
注意: 本例使用 鏈式語法。

```
_(3).times(function(n){ genie.grantWishNumber(n); });
```

random `_.random(min, max)`

返回一個**min** 和 **max**之間的隨機整數。如果你只傳遞一個參數，那麼將返回 `0` 和這個參數之間的整數。

```
_.random(0, 100);
=> 42
```

mixin `_.mixin(object)`

允許用您自己的實用程序函數擴展Underscore。傳遞一個 `{name: function}` 定義的哈希添加到Underscore對象，以及面向對象封裝。

```
_.mixin({
  capitalize: function(string) {
    return string.charAt(0).toUpperCase() + string.substring(1).toLowerCase();
  }
});
_("fabio").capitalize();
=> "Fabio"
```

iteratee `_.iteratee(value, [context])`

一個重要的內部函數用來生成可應用到集合中每個元素的回調， 返回想要的結果 - 無論是等式，任意回調，屬性匹配，或屬性訪問。
通過 `_.iteratee` 轉換判斷的Underscore 方法的完整列表
是 `map` , `find` , `filter` , `reject` , `every` , `some` , `max` , `min` , `sortBy` , `groupBy` , `indexBy` , `countBy` , `sortedIndex` , `partition` , 和 `unique` .

```
var stooges = [{name: 'curly', age: 25}, {name: 'moe', age: 21}, {name: 'larry', age: 23}];
_.map(stooges, _.iteratee('age'));
=> [25, 21, 23];
```

uniqueId `_.uniqueId([prefix])`

為需要的客戶端模型或DOM元素生成一個全局唯一的id。如果**prefix**參數存在， id 將附加給它。

```
_.uniqueId('contact_');
=> 'contact_104'
```

escape `_.escape(string)`

轉義HTML字符串，替換 `&` , `<` , `>` , `"` , `'` , 和 `/` 字符。

```
_.escape('Curly, Larry & Moe');
=> "Curly, Larry &amp;amp; Moe"
```

unescape `_.unescape(string)`

和**escape**相反。轉義HTML字符串，替換 `&`, `<`, `>`, `&quot;`, ```, 和 `&#x2F;` 字符。

```
_.unescape('Curly, Larry &amp; Moe');
=> "Curly, Larry & Moe"
```

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

如果指定的**property** 的值是一個函數，那麼將在**object**上下文內調用它;否則，返回它。如果提供默認值，並且屬性不存在，那麼默認值將被返回。如果設置 `defaultValue` 是一個函數，它的結果將被返回。

```
var object = {cheese: 'crumpets', stuff: function(){ return 'nonsense'; }};
_.result(object, 'cheese');
=> "crumpets"
_.result(object, 'stuff');
=> "nonsense"
_.result(object, 'meat', 'ham');
=> "ham"
```

now `_.now()`

一個優化的方式來獲得一個當前時間的整數時間戳。可用於實現定時/動畫功能。

```
_.now();
=> 1392066795351
```

template `_.template(templateString, [settings])`

將 JavaScript 模板編譯為可以用於頁面呈現的函數, 對於通過JSON數據源生成複雜的HTML並呈現出來的操作非常有用。 模板函數可以使用 `<%= … %>` 插入變量, 也可以用 `<% … %>` 執行任意的 JavaScript 代碼。 如果您希望插入一個值, 並讓其進行HTML轉義,請使用 `<%- … %>`。 當你要給模板函數賦值的時候，可以傳遞一個含有與模板對應屬性的**data**對象 。 如果您要寫一個一次性的, 您可以傳對象 **data** 作為第二個參數給模板 **template** 來直接呈現, 這樣頁面會立即呈現而不是返回一個模板函數. 參數 **settings** 是一個哈希表包含任何可以覆蓋的設置 `_.templateSettings`.

```
var compiled = _.template("hello: <%= name %>");
compiled({name: 'moe'});
=> "hello: moe"

var template = _.template("<b><%- value %></b>");
template({value: '<script>'});
=> "<b>&lt;script&gt;</b>"
```

您也可以在JavaScript代碼中使用 `print`. 有時候這會比使用 `<%= ... %>` 更方便.

```
var compiled = _.template("<% print('Hello ' + epithet); %>");
compiled({epithet: "stooge"});
=> "Hello stooge"
```

如果ERB式的分隔符您不喜歡, 您可以改變Underscore的模板設置, 使用別的符號來嵌入代碼.定義一個 **interpolate** 正則表達式來逐字匹配嵌入代碼的語句, 如果想插入轉義後的HTML代碼則需要定義一個 **escape** 正則表達式來匹配,還有一個 **evaluate** 正則表達式來匹配您想要直接一次性執行程序而不需要任何返回值的語句.您可以定義或省略這三個的任意一個.例如, 要執行Mustache.js類型的模板:

```
_.templateSettings = {
  interpolate: /\{\{(.+?)\}\}/g
};

var template = _.template("Hello {{ name }}!");
```

```
template({name: "Mustache"});
=> "Hello Mustache!"
```

默認的, **template** 通過 `with` 語句來取得 data 所有的值. 當然, 您也可以在 **variable** 設置裡指定一個變量名. 這樣能顯著提升模板的渲染速度.

```
_.template("Using 'with': <%= data.answer %>", {variable: 'data'})({answer: 'no'});
=> "Using 'with': no"
```

預編譯模板對調試不可重現的錯誤很有幫助. 這是因為預編譯的模板可以提供錯誤的代碼行號和堆棧跟蹤, 有些模板在客戶端(瀏覽器)上是不能通過編譯的 在編譯好的模板函數上, 有 **source** 屬性可以提供簡單的預編譯功能.

```
<script>
  JST.project = <%= _.template(jstText).source %>;
</script>
```

鏈式語法(Chaining)

您可以在面向對象或者函數的風格下使用Underscore, 這取決於您的個人偏好. 以下兩行代碼都可以 把一個數組裡的所有數字乘以2.

```
_.map([1, 2, 3], function(n){ return n * 2; });
_([1, 2, 3]).map(function(n){ return n * 2; });
```

對一個對象使用 `chain` 方法, 會把這個對象封裝並 讓以後每次方法的調用結束後都返回這個封裝的對象, 當您完成了計算, 可以使用 `value` 函數來取得最終的值. 以下是一個同時使用了 **map/flatten/reduce** 的鏈式語法例子, 目的是計算一首歌的歌詞裡每一個單詞出現的次數.

```
var lyrics = [
  {line: 1, words: "I'm a lumberjack and I'm okay"},
  {line: 2, words: "I sleep all night and I work all day"},
  {line: 3, words: "He's a lumberjack and he's okay"},
  {line: 4, words: "He sleeps all night and he works all day"}
];

_.chain(lyrics)
  .map(function(line) { return line.words.split(' '); })
  .flatten()
  .reduce(function(counts, word) {
    counts[word] = (counts[word] || 0) + 1;
    return counts;
  }, {})
  .value();

=> {lumberjack: 2, all: 4, night: 2 ... }
```

In addition, the 此外, 數組原型方法 也通過代理加入到了鏈式封裝的Underscore對象, 所以您可以 在鏈式語法中直接使用 `reverse` 或 `push` 方法, 然後再接著其他的語句.

chain `_.chain(obj)`

返回一個封裝的對象. 在封裝的對象上調用方法會返回封裝的對象本身, 直道 `value` 方法調用為止.

```
var stooges = [{name: 'curly', age: 25}, {name: 'moe', age: 21}, {name: 'larry', age: 23}];
var youngest = _.chain(stooges)
  .sortBy(function(stooge){ return stooge.age; })
```

```
  .map(function(stooge){ return stooge.name + ' is ' + stooge.age; })
  .first()
  .value();
=> "moe is 21"
```

value `_(obj).value()`
獲取封裝對象的最終值.

```
_([1, 2, 3]).value();
=> [1, 2, 3]
```

更多鏈接 & 推薦閱讀（Links & Suggested Reading）

Underscore文檔也有 簡體中文 版

Underscore.lua, 一個Lua版本的Underscore, 函數都通用. 包含面向對象封裝和鏈式語法. (源碼)

Underscore.m, 一個 Objective-C 版本的 Underscore.js, 實現了大部分函數, 它的語法鼓勵使用鏈式語法. (源碼)

_.m, 另一個 Objective-C 版本, 這個版本與原始的 Underscore.js API 比較相近. (源碼)

Underscore.php, 一個PHP版本的Underscore, 函數都通用. 包含面向對象封裝和鏈式語法. (源碼)

Underscore-perl, 一個Perl版本的Underscore, 實現了大部分功能, 主要針對於Perl的哈希表和數組. (源碼)

Underscore.cfc, 一個 Coldfusion 版本的 Underscore.js, 實現了大部分函數. (源碼)

Underscore.cfc, a Coldfusion port of many of the Underscore.js functions. (source)

Underscore.string, 一個Underscore的擴展, 添加了多個字符串操作的函數,
如: `trim`, `startsWith`, `contains`, `capitalize`, `reverse`, `sprintf`, 還有更多.

Ruby的 枚舉 模塊.

Prototype.js, 提供類似於Ruby枚舉方式的JavaScript集合函數.

Oliver Steele的 Functional JavaScript, 包含全面的高階函數支持以及字符串的匿名函數.

Michael Aufreiter的 Data.js, 一個JavaScript的數據操作和持久化的類庫.

Python的 迭代工具.

PyToolz, 一個Python端口　　擴展了itertools和functools，包括很多的Underscore API。

Funcy, a practical collection of functional helpers for Python, partially inspired by Underscore.

Change Log

1.8.3 — *April 2, 2015* — Diff — Docs

- 新增一個 `_.create` 方法, 作為 `Object.create` 的一個簡化版本。
- 圍繞著一個iOS錯誤，可導致 `isArrayLike` 成為 JIT-ed，還修復一個傳遞 `0` 給 `isArrayLike` 時的bug。

1.8.2 — *Feb. 22, 2015* — Diff — Docs

- 恢復先前在1.8.1改變的老版本 Internet Explorer的邊界情況。
- `_.contains` 添加了一個 `fromIndex` 參數。

1.8.1 — *Feb. 19, 2015* — Diff — Docs

- 修復/改變一些老版本Internet Explorer和邊界情況的行為。 用老版本Internet Explorer和Underscore 1.8.1 測試一下您的應用，讓我們知道發生了什麼...

1.8.0 — *Feb. 19, 2015* — Diff — Docs

- 新增 `_.mapObject`，它類似於 `_.map`，但只是對於在對象中的值。
- 新增 `_.allKeys`，它返回一個對象上*所有*枚舉屬性名稱。
- 恢復一個1.7.0的變更，`_.extend` 僅複製「自己」的屬性（愚人碼頭註：不包括繼承過來的屬性）。 希望沒有傷害到你 — 如果它這次傷害到了你，我向你道歉。
- 添加 `_.extendOwn` — `_.extend` 的一個不太有用的形式，僅複製「自己」的屬性，（愚人碼頭註：不包括繼承過來的屬性）。
- 添加 `_.findIndex` 和 `_.findLastIndex` 功能，很好地補充了 `_.indexOf` 和 `_.lastIndexOf`。
- 增加了一個 `_.isMatch` 斷言（判斷）函數，它會告訴你，如果一個對象匹配的key-value屬性。和 `_.isEqual` 和 `_.matcher` 非常接近。
- 增加了一個 `_.isError` 功能。
- 恢復 `_.unzip` 功能，做為 `zip` 相反的功能。回滾。
- `_.result` 現在有一個可選的回退值（或函數提供回退值）。
- 添加 `_.propertyOf` 函數發生器作為 `_.property` 的鏡像版本。
- 棄用 `_.matches`。目前他已經有一個更加和諧的名稱 - `_.matcher`。
- 簡化各種多樣的代碼，以提高跨平台兼容性的變化，修復邊界情況的bug。

1.7.0 — *August 26, 2014* — Diff — Docs

- For consistency and speed across browsers, Underscore now ignores native array methods for `forEach`, `map`, `reduce`, `reduceRight`, `filter`, `every`, `some`, `indexOf`, and `lastIndexOf`. "Sparse" arrays are officially dead in Underscore.
- Added `_.iteratee` to customize the iterators used by collection functions. Many Underscore methods will take a string argument for easier `_.property`-style lookups, an object for `_.where`-style filtering, or a function as a custom callback.
- Added `_.before` as a counterpart to `_.after`.
- Added `_.negate` to invert the truth value of a passed-in predicate.
- Added `_.noop` as a handy empty placeholder function.
- `_.isEmpty` now works with `arguments` objects.
- `_.has` now guards against nullish objects.
- `_.omit` can now take an iteratee function.
- `_.partition` is now called with `index` and `object`.
- `_.matches` creates a shallow clone of your object and only iterates over own properties.
- Aligning better with the forthcoming ECMA6 `Object.assign`, `_.extend` only iterates over the object's own properties.
- Falsey guards are no longer needed in `_.extend` and `_.defaults` —if the passed in argument isn't a JavaScript object it's just returned.

- Fixed a few edge cases in `_.max` and `_.min` to handle arrays containing `NaN` (like strings or other objects) and `Infinity` and `-Infinity`.
- Override base methods like `each` and `some` and they'll be used internally by other Underscore functions too.
- The escape functions handle backticks (`` ` ``), to deal with an IE ≦ 8 bug.
- For consistency, `_.union` and `_.difference` now only work with arrays and not variadic args.
- `_.memoize` exposes the cache of memoized values as a property on the returned function.
- `_.pick` accepts `iteratee` and `context` arguments for a more advanced callback.
- Underscore templates no longer accept an initial `data` object. `_.template` always returns a function now.
- Optimizations and code cleanup aplenty.

1.6.0 — *February 10, 2014* — Diff — Docs

- Underscore 現在將自己註冊為AMD（Require.js），Bower和Component， 以及作為一個CommonJS的模塊和常規（Java）的腳本。 雖然比較醜陋，但也許是必要的。
- 添加了 `_.partition`, 一個拆分一個集合為兩個結果列表，第一個數組其元素都滿足**predicate**迭代函數， 而第二個的所有元素均不能滿足**predicate**迭代函數。
- 添加了 `_.property`, 創建一個迭代器，輕鬆從對象中獲取特定屬性。 與其他 Underscore 集合函數結合使用時很有用。
- 添加了 `_.matches`, 一個函數，它會給你一個斷言 可以用來辨別 給定的對象是否匹配指定鍵/值屬性的列表。
- 添加了 `_.constant`, 作為 `_.identity` 高階.
- 添加了 `_.now`, 一個優化的方式來獲得一個時間戳 — 在內部用來加快 `debounce` 和 `throttle`。
- `_.partial` 函數 現在可以用來部分適用的任何參數， 通過傳遞 `_`，無論你想要一個佔位符變量， 稍後填充。
- `_.each` 函數現在 返回一個列表的引用，方便鏈式調用。
- The `_.keys` 函數 現在 當空對象傳入的時候返回一個空數組。
- … 更多雜項重構.

1.5.2 — *Sept. 7, 2013* — Diff

- 增加了 `indexBy` 函數，他是 `countBy` and `groupBy` 功能相輔相成。
- 增加了 `sample` 函數，從數組中產生隨機元素。
- 一些有關函數的優化， `_.keys` 方面的實現（包含大幅提升的對象上 `each` 函數）。另外 `debounce` 中一個緊密的循環。

1.5.1 — *Jul. 8, 2013* — Diff

- 刪除 `unzip`，因為她簡單的應用了 `zip` 參數的一個數組。使用 `_.zip.apply(_, list)` 代替。

1.5.0 — *Jul. 6, 2013* — Diff

- 添加一個 `unzip` 新函數，作為 `_.zip` 功能相反的函數。
- `throttle` 函數現在增加一個 `options` 參數，如果你想禁用第一次首先執行的話，傳遞{leading: false}，還有如果你想禁用最後一次執行的話，傳遞{trailing: false}。
- Underscore現在提供了一個source map 方便壓縮文件的調試。
- `defaults` 函數現在只 重寫 `undefined` 值，不再重寫 `null` 值。
- 刪除不帶方法名參數調用 `_.bindAll` 的能力。
- 刪除計數為0，調用 `_.after` 的能力。調用的最小數量現在是 `1`（自然數）

1.4.4 — *Jan. 30, 2013* — Diff

- 添加 `_.findWhere`，在列表中找到的第一個元素，一組特定的鍵和值相匹配。
- 添加 `_.partial`，局部應用一個函數填充在任意數值的參數， *不*改變其動態 `this` 值。
- 通過去掉了一些的邊緣案件涉包括構造函數來簡化 `bind`。總之：不要 `_.bind` 你的構造器。
- 一個 `invoke` 的小優化。
- 修改壓縮版本中由於不當壓縮引起的 `isFunction` BUG。

1.4.3 — *Dec. 4, 2012* — Diff

- 改進Underscore和 與Adobe的JS引擎的兼容性，可用於script Illustrator，Photoshop和相關產品。
- 添加一個默認的 `_.identity` 迭代到 `countBy` 和 `groupBy` 中。
- `uniq` 函數現在接受 `array, iterator, context` 作為參數列表。
- `times` 函數現在放回迭代函數結果的映射數組。
- 簡化和修復 `throttle` BUG。

1.4.2 — *2012年10月1日* — 比較文件

- 為了保證向下兼容, 恢復了 1.4.0 候選版時的一些特性 當傳 `null` 到迭代函數時. 現在又變回非可選參數了.

1.4.1 — *Oct. 1, 2012* — 比較文件

- 修復 1.4.0 版本裡 `lastIndexOf` 函數的退化.

1.4.0 — *Sept. 27, 2012* — 比較文件

- 增加 `pairs` 函數, 把一個 JavaScript 對象轉換成 `[key, value]` 的組合 ... 同樣地, 也有 `object` 函數, 把 `[key, value]` 的數組組合轉換成對象.
- 增加 `countBy` 函數, 可以計算數組內符合條件的對象個數.
- 增加 `invert` 函數, 在對象裡實現一個簡單的鍵值對調.
- 增加 `where` 函數, 以便於篩選出一個數組裡包含指定鍵值的對象數組.
- 增加 `omit` 函數, 可以過濾掉對象裡的對應key的屬性.
- 增加 `random` 函數, 生成指定範圍內的隨機數.
- 用 `_.debounce` 創建的函數現在會返回上一次更新後的值, 就像 `_.throttle` 加工過的函數一樣.
- `sortBy` 函數現在使用了穩定的排序算法.
- 增加可選參數 `fromIndex` 到 `indexOf` 和 `lastIndexOf` 函數裡.
- Underscore 的迭代函數裡不再支持稀疏數組. 請使用 `for` 循環來代替 (或者會更好).
- `min` 和 `max` 函數現在可以用在 *非常*大的數組上.
- 模板引擎裡插入變量現在可以使用 `null` 和 `undefined` 作為空字符串.
- ~~Underscore 的迭代函數不再接受 `null` 作為非可選參數. 否則您將得到一個錯誤提示.~~
- 一些小幅修復和調整, 可以在此查看與之前版本的 比較. **1.4.0** 可能比較不向下兼容, 這取決於您怎麼使用Underscore — 請在升級後進行測試。

1.3.3 — *2012年4月10日*

- `_.template`的多處改進, 現在為潛在的更有效的服務器端預編譯 提供模板的`源(source)`作為屬性. 您現在也可以在創建模板的時候 設置 `variable` 選項, 之後可以通過這個變量名取到模板傳入的數據, 取代了 `with` 語句 — 顯著的改進了模板的渲染速度.
- 增加了 `pick` 函數, 它可以過濾不在所提供的白名單之內的其他屬性.
- 增加 `result` 函數, 在與API工作時很方便, 允許函數屬性或原始屬性(非函數屬性).
- 增加 `isFinite` 函數, 因為有時候僅僅知道某變量是一個 數的時候還不夠, 還要知道它是否是有限的數.
- `sortBy` 函數現在可以傳屬性名作為對象的排序標準.
- 修復 `uniq` 函數, 現在可以在稀疏數組上使用了.
- `difference` 函數現在在對比數組差異的時候只執行淺度的flatten, 取代之前的深度flatten.
- `debounce` 函數現在多了一個參數 `immediate`, 會影響到達時間間隔後執行的是最先的函數調用還是最後的函數調用.

1.3.1 — *2012年1月23日*

- 增加 `_.has` 函數, 作為 `hasOwnProperty` 更安全的版本.
- 增加 `_.collect` , 作為 `_.map` 的別名.
- 恢復一個舊的修改, `_.extend` 將再次可以正確複製 擁有undefined值的屬性.
- 修復在 `_.template` 的嵌入語句裡反轉義斜槓的bug.

1.3.0 — *2012年1月11日*

- 移除Underscore對AMD(RequireJS)的支持. 如果您想繼續在 RequireJS裡使用Underscore, 可以作為一個普通的script加載, 封裝或修改您的Underscore副本, 或者下載一個Underscore別的fork版本.

1.2.4 — *Jan. 4, 2012*

- 您現在可以寫 (您應該會這樣用, 因為這樣更簡單) `_.chain(list)` 來代替 `_(list).chain()`.
- 修復已反轉義的字符在Underscore模板裡的錯誤, 並增加了支持自定義支持, 使用`_.templateSettings`, 只需要定義一到兩個必備的正則表達式.
- 修復以數組作為第一參數傳給`_.wrap`函數的錯誤.
- 改進與ClojureScript的兼容性, 增加`call` 函數到`String.prototype`裡.

1.2.3 — *2011年12月7日*

- 動態範圍在已編譯的 `_.template` 函數中保留, 所以您可以使用 `this` 屬性, 如果您喜歡的話.
- `_.indexOf` 和 `_.lastIndexOf` 增加對稀疏數組的支持.
- `_.reduce` 和 `_.reduceRight` 現在都可以傳一個明確的 `undefined` 值. (您為什麼要這樣做並沒有任何原因)

1.2.2 — *2011年11月14日*

- 繼續改進 `_.isEqual` , 要讓它和語義上所說的一樣. 現在原生的JavaScript會一個對象與它的封裝起來的對象視為相等的, 還有, 數組只會對比他們數字元素 (#351).
- `_.escape` 不再嘗試在非雙重轉義的轉義HTML實體上進行轉換. 現在不管怎樣只會反轉義一次 (#350).
- 在 `_.template` 裡, 如果願意的話您可以省略嵌入表達式後面的分號: `<% }) %>` (#369).
- `_.after(callback, 0)` 現在會立即觸發callback函數, 把"after"做得更易於使用在異步交互的API上 (#366).

1.2.1 — *2011年10月24日*

- `_.isEqual` 函數的幾個重要bug修復, 現在能更好地用在複雜的數組上, 和擁有 `length` 屬性的非數組對象上了. (#329)
- **jrburke** 提供了導出Underscore以便AMD模塊的加載器可以加載, 還有 **tonylukasavage** 提供了導出Underscore給Appcelerator Titanium使用. (#335, #338)
- 您現在可以使用 `_.groupBy(list, 'property')` 作為 以指定的共同屬性來分組的快捷方法.
- `_.throttle` 函數現在調用的時候會立即自行一次, 此後才是再每隔指定時間再執行一次 (#170, #266).
- 大多數 `_.is[類型]` 函數不再使用ducktype寫法(詳見Ruby的duck type).
- `_.bind` 函數現在在構造函數(constructor)也能用了, 兼容ECMAScript 5標準. 不過您可能永遠也用不到 `_.bind` 來綁定一個構造函數.
- `_.clone` 函數不再封裝對象裡的非對象屬性.
- `_.find` 和 `_.filter` 現在作為 `_.detect` 和 `_.select` 的首選函數名.

1.2.0 — *2011年10月5日*

- `_.isEqual` 函數現在支持深度相等性對比, 檢測循環結構, 感謝Kit Cambridge.
- Underscore模版現在支持嵌入HTML轉義字符了, 使用 `<%- ... %>` 語句.
- Ryan Tenney 提供了 `_.shuffle` 函數, 它使用 Fisher-Yates算法的修改版, 返回一個亂序後的數組副本.
- `_.uniq` 現在可以傳一個可選的迭代器iterator, 用來確定一個數組以什麼樣的標準來確定它是否唯一的.
- `_.last` 現在增加了一個可選參數, 可以設置返回集合裡的最後N個元素.
- 增加了一個新函數 `_.initial`, 與 `_.rest` 函數相對, 它會返回一個列表除了最後N個元素以外的所有元素.

1.1.7 — *2011年7月13日*

增加 `_.groupBy`, 它可以將一個集合裡的元素進行分組. 增加 `_.union` 和 `_.difference`, 用來補充 (重命名過的) `_.intersection` 函數. 多方面的改進以支持稀疏數組. `_.toArray` 現在如果直接傳數組時, 將會返回此數組的副本. `_.functions` 現在會返回存在於原型鏈中的函數名.

1.1.6 — *2011年4月18日*

增加 `_.after` 函數, 被它改造過的函數只有在執行指定次數之後才會生效. `_.invoke` 現在將使用函數的直接引用. `_.every` 現在必須傳如迭代器函數, 為了符合ECMAScript 5標準. `_.extend` 當值為undefined的時候不再複製鍵值. `_.bind` 現在如果試圖綁定一個undefined值的時候將報錯.

1.1.5 — *2011年3月20日*

增加 `_.defaults` 函數, 用來合併JavaScript對象, 一般用來做生成默認值使用. 增加 `_.once` 函數, 用來把函數改造成只能運行一次的函數. `_.bind` 函數現在委託原生的ECMAScript 5版本(如可用). `_.keys` 現在傳非對象的值時, 將會拋出一個錯誤, 就和ECMAScript 5標準裡的一樣. 修復了 `_.keys` 函數在傳入稀疏數組時的bug.

1.1.4 — *2011年1月9日*

改進所有數組函數當傳值 `null` 時候的行為, 以符合ECMAScript 5標準. `_.wrap` 函數現在能正確地 給封裝的函數設置 `this` 關鍵字了. `_.indexOf` 函數增加了可選參數isSorted, 尋找索引的時候會將數組作為已排序處理, 將使用更快的二進制搜索. 避免使用 `.callee`, 保證 `_.isArray` 函數 在ECMAScript 5嚴格模式下能正常使用.

1.1.3 — *2010年12月1日*

在CommonJS裡, Underscore可以像這樣引入:
`var _ = require("underscore")`. 增加 `_.throttle` 和 `_.debounce` 函數. 移除 `_.breakLoop` 函數, 為了符合ECMAScript 5標準裡所說的每一種實現形式都是不能*break*的 — 這將去掉try/catch塊, 現在, 您遇到Underscore迭代器的拋出的異常時, 將會有更完善的堆棧跟蹤來檢查錯誤所在之處. 改進 **isType** 一類函數, 以便更好地兼容Internet Explorer瀏覽器. `_.template` 函數現在可以正確的反轉義模板中的反斜槓了. 改進 `_.reduce` 函數以兼容ECMAScript 5標準: 如果您不傳初始值, 將使用集合裡的第一項作為初始值. `_.each` 不再返回迭代後的集合, 為了與ECMAScript 5的 `forEach` 保持一致.

1.1.2

修復 `_.contains` 指向 `_.intersect` 函數的錯誤, 應該是指向 `_.include` 函數(_.cotains應該是_.include的別名), 增加 `_.unique`, 作為 `_.uniq` 函數的別名.

1.1.1

改進 `_.template` 函數的運行速度, 和處理多行插入值的性能. Ryan Tenney 提供了許多Underscore函數的優化方案. 增加了帶註釋版本的源代碼.

1.1.0

修改了 `_.reduce` 函數以符合ECMAScript 5規範, 取代了之前Ruby/Prototype.js版本的 `_.reduce`. 這是一個不向下兼容的修改. `_.template` 函數現在可以不傳參了, 並保留空格. `_.contains` 是一個 `_.include` 函數新的別名.

1.0.4

Andri Möll 提供了 `_.memoize` 函數, 以緩存計算結果, 來優化的耗時較長的函數, 使得運行速度變快.

1.0.3

修復了 `_.isEqual` 函數在對比包含 `NaN` 的對象時返回 `false` 的問題. 技術上改良後理論上是正確的, 但是語義上似乎有矛盾, 所以要注意避免對比含有NaN的對象.

1.0.2

修復 `_.isArguments` 在新版本Opera瀏覽器裡的bug, Opera裡會把arguments對象當作數組.

1.0.1

修復了 `_.isEqual` 函數的bug: 這個bug出現在當對比特定因素兩個對象時, 這兩個對象有著相同個數的值為undefined的key, 但不同名.

1.0.0

Underscore在這幾個月裡算是相對穩定了, 所以現在打算出測試版, 版本號為**1.0**. 從**0.6**版本開始進行改進, 包括 `_.isBoolean` 的改進, 和 `_.extend` 允許傳多個source對象.

0.6.0

主要版本, 整合了一系列的功能函數, 包括 Mile Frawley寫的在保留援用功能的基礎上, 對集合函數進行重構, 內部代碼更加簡潔. 新的 `_.mixin` 函數, 允許您自己的功能函數繼承Underscore對象. 增加 `_.times` 函數, 跟Ruby或Prototype.js裡的times的功能一樣. 對ECMAScript 5的 `Array.isArray` 函數提供原生支持, 還有 `Object.keys`.

0.5.8

修復了Underscore的集合函數, 以便可以用於DOM的 節點列表(NodeList) 和 HTML集合(HTMLCollection) 再一次地感謝 Justin Tulloss.

0.5.7

修改 `_.isArguments` 函數, 使用了更安全的實現方式, 還有 加快了 `_.isNumber` 的運行速度,
感謝 Jed Schmidt.

0.5.6

增加了 `_.template` 對自定義分隔符的支持, 由 Noah Sloan提供.

0.5.5

修復了一個在移動版Safari裡關於arguments對象的面向對象封裝的bug.

0.5.4

修復了 `_.template` 函數裡多個單引號在模板裡造成的錯誤. 瞭解更多請閱讀: Rick Strahl的博客文章.

0.5.2

幾個函數的重寫: `isArray`, `isDate`, `isFunction`, `isNumber`, `isRegExp`, 和 `isString`, 感謝Robert Kieffer提供的建議. 取代了 `Object#toString` 的對比方式, 現在以屬性來進行對比, 雖然說安全性有所降低, 但是速度比以前快了有一個數量級. 因此其他大多數的Underscore函數也有小幅度的速度提升. 增加了 `_.tap` 函數, 由Evgeniy Dolzhenko 提供, 與 Ruby 1.9的tap方法相似, 對鏈式語法裡嵌入其他功能(如登錄)很方便.

0.5.1

增加了 `_.isArguments` 函數. 許多小的安全檢查和優化由 Noah Sloan 和 Andri Möll提供.

0.5.0

[API變更] `_.bindAll` 現在會將context對象作為第一個參數. 如果不傳方法名, context對象的所有方法都會綁定到 context, 支持鏈式語法和簡易綁定. `_.functions` 現在只要一個參數, 然後返回所有的方法名(類型為Function的屬性). 調用 `_.functions(_)` 會列出所有的Underscore函數. 增加 `_.isRegExp` 函數, `isEqual` 現在也可以檢測兩個RegExp對象是否相等了. 所有以"is"開頭的函數已經縮減到同一個定義裡面, 由Karl Guertin 提供的解決方案.

0.4.7

增加 `isDate`, `isNaN`, 和 `isNull`. 優化 `isEqual` 函數對比兩個數組或兩個時間對象時的性能. 優化了 `_.keys` 函數, 現在的運行速度比以前加快了***25%–2倍*** (取決於您所使用的瀏覽器)會加速其所依賴的函數, 如 `_.each`.

0.4.6

增加 `range` 函數, Python裡同名函數range 的移植版, 用於生成靈活的整型數組. 原始版由Kirill Ishanov提供.

0.4.5

增加 `rest` 函數, 可以對數組和arguments對象使用, 增加了兩個函數的別名, `first` 的別名為 `head`, 還有 `rest` 的別名為 `tail`, 感謝 Luke Sutton的解決方案. 增加測試文件, 以確保所有Underscore的數組函數都可以在用在 ***arguments*** 對象上.

0.4.4

增加 `isString`, 和 `isNumber` 函數. 修復了 `_.isEqual(NaN, NaN)` 會返回 ***true*** 的問題.

0.4.3

開始使用原生的 `StopIteration` 瀏覽器對象(如果瀏覽器支持). 修復Underscore在CommonJS環境上的安裝.

0.4.2

把解除封裝的函數unwrapping改名為 `value`, 更清晰.

0.4.1

鏈式語法封裝的Underscore對象支持函數原型方法的調用, 您可以在封裝的數組上連續調用任意涵數. 增加 `breakLoop` 方法, 可以隨時在Underscore的迭代中 **中斷** 並跳出迭代. 增加 `isEmpty` 函數, 在數組和對象上都有用.

0.4.0

現在所有的Underscore函數都可以用面向對象的風格來調用了, 比如: `_([1, 2, 3]).map(...);`. Marc-André Cournoyer 提供了原始的解決方案. 封裝對象可以用鏈式語法連續調用函數. 添加了 `functions` 方法, 能以正序方式列出所有的 Underscore函數.

0.3.3

增加JavaScript 1.8的函數 `reduceRight`. 別名為 `foldr`, 另外 `reduce` 的別名為 `foldl`.

0.3.2

可以在 Rhino 上運行了. 只要在編譯器裡輸入: `load("underscore.js")`. 增加功能函數 `identity`.

0.3.1

所有迭代器在原始集合裡現在都作為第三個參數傳入, 和JavaScript 1.6的 **forEach** 一致. 迭代一個對象現在會以 `(value, key, collection)` 來調用, 更多詳情, 請查看 `_.each`.

0.3.0

增加 Dmitry Baranovskiy的 綜合優化, 合併 Kris Kowal的解決方案讓Underscore符合 CommonJS 標準,並和 Narwhal兼容.

0.2.0

添加 `compose` 和 `lastIndexOf`, 重命名 `inject` 為 `reduce`, 添加 `inject`, `filter`, `every`, `some`, 和 `forEach` 的別名.

0.1.1

添加 `noConflict`, 以便 "Underscore" 對象可以分配給其他變量.

0.1.0

Underscore.js 首次發佈.