

---

# 目錄

简介	1.1
underscore 基础篇	1.2
结构	1.2.1
松弛绑定	1.2.2
局部变量的妙用	1.2.3
undefined 的处理	1.2.4
迭代！迭代！迭代！	1.2.5
rest 参数	1.2.6
创建对象的正确姿势	1.2.7
underscore 集合篇	1.3
什么能够被迭代	1.3.1
map-reduce	1.3.2
真值检测函数	1.3.3
查询	1.3.4
随机取样与洗牌算法	1.3.5
更好用的排序	1.3.6
模拟一段 SQL	1.3.7
拾遗	1.3.8
underscore 数组篇	1.4
定位	1.4.1
展开一个数组	1.4.2
数组运算	1.4.3
拾遗	1.4.4
underscore 函数篇	1.5
上下文绑定	1.5.1
偏函数	1.5.2
缓存	1.5.3

---

延时执行	1.5.4
throttle 与 debounce	1.5.5
函数组合	1.5.6
指定调用次数	1.5.7
拾遗	1.5.8
underscore 对象篇	1.6
Object(obj)	1.6.1
属性操作	1.6.2
克隆与扩展	1.6.3
白名单	1.6.4
对象相等性判断	1.6.5
类型判断	1.6.6
underscore 实用工具篇	1.7
工具汇总	1.7.1
模板引擎	1.7.2
underscore 内容拾遗	1.8
面向对象风格的支持	1.8.1
mixin	1.8.2
链式调用	1.8.3

---

## 写作意图

起初，我分析 `underscore` 的源码只是想更深入的了解 函数式编程（**Functional Programming**），但分析结束后，我就觉得单纯的源码注释不足以记录我的收获、理解和感悟，所以我想把这些写下来，我粗略地将写作意图概括如下：

- 函数式编程近些年非常火爆，诸如 `haskell` 这样的纯函数式编程语言获得了非常高的社区活跃度。`JavaScript` 支持多范式编程，抛开 `underscore` 和 `lodash` 这样的生来为了函数编程的库不谈，诸如 `redux` 这样的库也大量运用了函数式编程，即便作为一个 `react+redux` 的业务开发者，想要深入理解的 `redux` 的实现机制，也不得不学习函数式编程。因此，学习函数式编程，将会成为 `JavaScript` 开发者的必须。
- 在阅读 `underscore` 的源码期间，被作者 `jashkenas`（他同时也是 `backbone` 和 `coffee` 的作者）的功力深深折服，一些功能可能我也能写出，但绝对写不了如此健壮。所以，深入学习 `underscore` 源码，不仅有助于我们认识函数式编程，也能深化我们对于 `JavaScript` 中一些基础知识的理解和掌握。
- 随着 `backbone` 的衰落和 `lodash` 的崛起，`underscore` 的热度已经不及当年，但是截止这篇文章的开始前的一个月，`underscore` 仍然有最新的 bug 修复，可见作者 `jashkenas` 仍然没有放弃 `underscore` 的维护。所以现在分析 `underscore` 的源码仍然不显得过时。相较于 `lodash`，`underscore` 的源码更加短小，也不太涉及 `JavaScript` 中的一些奇淫巧技，所以，分析 `underscore` 更加适合 `JavaScript` 开发者的进阶。在完成了 `underscore` 的源码分析后，希望我自己有时间，也希望读者有意愿再去分析 `lodash` 的源码，后者在性能和功能上都已经超越了 `underscore`，并且长时间霸占了 `npm` 了最热 `package` 的位置。

## 章节安排

### `underscore` 基础篇

在基础部分，将会阐述 `underscore` 的大致结构及一些广泛用到的内部函数（`internal function`），这些函数被大量用到了 `underscore` 的 API 实现中，是我们之后理解 `underscore` 源码的必须途径。

之后，我们按照官方 API 文档的顺序来阐述 underscore 的源码实现，由于很多 API 的实现可以举一反三，所以，本书并不会啰嗦的阐述每个 API 的实现，如果真的由此需求，可以配合我写的 [underscore 中文注释](#) 辅助阅读。

## underscore 集合篇

不同于数学当中的集合，在 underscore 中，简单地定义集合为一个可迭代的序列，相较于原生的 ES5 提供的迭代方法，underscore 不仅能够对数组进行迭代，还能够对对象进行迭代。

## underscore 数组篇

这一章节我们将介绍 underscore 中提供的针对数组的操作，部分 API 已经在集合篇中有过阐述，不再赘述。

## underscore 函数篇

在 JavaScript 中，函数是第一型的对象，函数在 JavaScript 中的地位因此可见一斑。这一章节也是我认为最为重要的一章，在本章中，能够见到许多实用的针对函数的操作，以及函数式编程中的重要概念。

## underscore 对象篇

本章中，将介绍 underscore 中操作对象的 API。

## underscore 实用工具篇

underscore 还提供了不少工具函数，来提供一些周边功能，如字符逃逸等。但其中最重要的是其提供的模板引擎工具，我将会花费很大笔墨对其进行描述。

## underscore 内容拾遗

最后，在收尾阶段，我们还会介绍 underscore 提供的面向对象风格（OOP Style），链式调用（Chain）等内容。

## 感谢

本文基于 underscore 的 [1.8.3 版本](#) 进行分析, 在阅读官方文档时遇到的困难时, 特别感谢 [underscore 中文教程](#) 提供的帮助。

欢迎转载或者引用, 但请注明出处, 这算是对我工作成果的认可和尊重。也欢迎拍砖, 相应问题可以发到 [discussion](#), 我会最快时间进行更正或者解答。

## underscore 基础篇

这一部分，我们会介绍一些 underscore 中的基础内容，包括有 underscore 的结构，模块封装，以及常用的内部函数等，这一部分是我们之后理解 underscore 的各个 API 的必经之路。

让我们先从 underscore 的结构入手。

## 结构

### 作用域包裹

与其他第三库一样，**underscore** 也通过 立即执行函数 来包裹自己的业务逻辑。一般而言，这些库的立即执行函数主要有以下目的：

- 避免全局污染：所有库的逻辑，库所定义和使用的变量全部被封装到了该函数的作用域中。
- 隐私保护：但凡在立即执行函数中声明的函数、变量等，除非是自己想暴露，否则绝无可能在外部获得。

```
(function(){  
    // .....执行逻辑  
})();
```

所以，当我们撰写自己的库的时候，也可以考虑在最外层包裹上一个立即执行函数。既不受外部影响，也不给外部添麻烦。

### — 对象

**underscore** 有下划线的意思，所以 **underscore** 通过一个下划线变量 `_` 来标识自身，值得注意的是，`_` 是一个函数对象，之后，所有的 **api** 都会被挂载到这个对象上，如 `_.each`，`_.map` 等：

```
var _ = function (obj) {  
    if (obj instanceof _) return obj;  
    if (!(this instanceof _)) return new _(obj);  
    this._wrapped = obj;  
};
```

那么问题来了，为什么 `_` 会被设计成一个函数对象，而不是普通对象 `{}` 呢。显然，这样的设计意味着之后可能存在这样的代码片：

```
var xxx = _(obj);
```

这样做的目的是什么呢？我会在之后的 [underscore 内容拾遗 / 面向对象风格的支持](#) 再进行解释。

## 执行环境判断

underscore 既能够服务于浏览器，又能够服务于诸如 nodejs 所搭建的服务端，underscore 对象 `_` 将会依托于当前的所处环境，挂载到不同的全局空间当中（浏览器的全局对象是 `window`，node 的全局对象是 `global`）。下面代码展示了 underscore 是如何判断自己所处环境的，这个判断逻辑也为我们自己想要撰写前后端通用的库的时候提供了帮助：

```
var root = typeof self == 'object' && self.self === self && self
  ||
    typeof global == 'object' && global.global === global &&
    global ||
    this;
```

### window or self ?

在 underscore 的判断所处环境的代码中，似乎我们没有看到 `window` 对象的引用，其实，在浏览器环境下，`self` 保存的就是当前 `window` 对象的引用。那么相比较于使用 `window`，使用 `self` 有什么优势呢？我们看到 [MDN](#) 上有这么一句话：

The Window.self read-only property returns the window itself, as a WindowProxy. It can be used with dot notation on a window object (that is, window.self) or standalone (self). The advantage of the standalone notation is that a similar notation exists for non-window contexts, such as in Web Workers.

概括来说，就是 `self` 还能用于一些不具有窗口的上下文环境中，比如 [Web Workers](#)。所以，为了服务于更多场景，underscore 选择了更加通用的 `self` 对象。



其次，如果处于 node 环境，那么 underscore 的对象 `_` 还将被作为模块导出：

```
if (typeof exports !== 'undefined' && !exports.nodeType) {  
  if (typeof module !== 'undefined' && !module.nodeType && module  
    .exports) {  
    exports = module.exports = _;  
  }  
  exports._ = _;  
} else {  
  root._ = _;  
}
```

## 松弛绑定

默认情况下，underscore 对象 `_` 会覆盖全局对象上同名的 `_` 属性。但是，underscore 也不过于蛮横，他会保存之前已经存在的 `_` 属性，因为像是 lodash 这样的一些库也喜欢将自己的对象命名为 `_`：

```
var previousUnderscore = root._;
```

当用户已经在全局对象上绑定了 `_` 对象时，可以通过 underscore 提供的 `noConflict` 函数来重命名 underscore 对象，或者说是手动获得 underscore 对象，避免与之前的 `_` 冲突：

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

看到 `noConflict` 的源码实现，我们发现，在其内部，将会恢复原来全局对象上的 `_`：

```
/**
 * 返回一个 underscore 对象，把_所有权交还给原来的拥有者（比如 lodash）
 */
_.noConflict = function () {
  // 回复原来的_指代的对象
  root._ = previousUnderscore;
  // 返回 underscore 对象
  return this;
};
```

## 局部变量的妙用

underscore 本身也依赖了不少 js 的原生方法，如下代码所示，underscore 会通过局部变量来保存一些他经常用到的方法或者属性，这样做的好处有如下两点：

- 在后续使用到这些方法或者属性时，避免了冗长的代码书写。
- 减少了对对象成员的访问深度，( `Array.prototype.push` --> `push` ), 这样做能带来一定的性能提升，具体可以参看 [《高性能 javascript》](#)

```
var ArrayProto = Array.prototype, ObjProto = Object.prototype;
var SymbolProto = typeof Symbol !== 'undefined' ? Symbol.prototype : null;

var push = ArrayProto.push,
    slice = ArrayProto.slice,
    toString = ObjProto.toString,
    hasOwnProperty = ObjProto.hasOwnProperty;
```

## undefined 的处理

### 不可靠的 undefined

在 JavaScript 中，假设我们想判断一个是否是 `undefined`，那么我们通常会这样写：

```
if(a === undefined){}
```

但是，JavaScript 中的 `undefined` 并不可靠，我们试着写这样一个函数：

```
function test(a) {  
  var undefined = 1;  
  console.log(undefined); // => 1  
  if(a===undefined) {  
    // ...  
  }  
}
```

可以看到，`undefined` 被轻易地修改为了 `1`，使得我们之后的对于 `undefined` 理解引起歧义。所以，在 JavaScript 中，把 `undefined` 直接解释为“未定义”是有风险的，因为这个标识符可能被篡改。

在 ES5 之前，全局的 `undefined` 也是可以被修改的，而在 ES5 中，该标识符被设计为了只读标识符，假如你现在的浏览器不是太老，你可以在控制台中输入以下语句测试一下：

```
undefined = 1;  
console.log(undefined); // => undefined
```

### 曲线救国

现在我们能够明确的，标识符 `undefined` 并不能真正反映“未定义”，所以我们得通过其他手段获得这一语义。幸好 `JJavaScript` 还提供了 `void` 运算符，该运算符会对指定的表达式求值，并返回受信的 `undefined`：

```
void expression
```

最常见的用法是通过以下运算来获得 `undefined`，表达式为 `0` 时的运算开销最小：

```
void 0;  
// or  
void(0);
```

在 `underscore` 中，所有需要获得 `undefined` 地方，都通过 `void 0` 进行了替代。

当然，曲线救国的方式不只一种，我们看到包裹 `jquery` 的立即执行函数：

```
(function(window, undefined) {  
    // ...  
})(window)
```

在这个函数中，我们没有向其传递第二参数（形参名叫 `undefined`），那么第二个参数的值就会被传递上“未定义”，因此，通过这种方式，在该函数的作用域中所有的 `undefined` 都为受信的 `undefined`。

## 参考资料

[MDN: undefined](#)

[MDN: void](#)

## 迭代！迭代！迭代

### 使用迭代，而不是循环

在函数式编程，更推荐使用 迭代：

```
var results = _.map([1,2,3],function(elem){
  return elem*2;
}); // => [2,4,6]
```

而不是 循环：

```
var results = [];
var elems = [1,2,3];
for(var i=0,length=elems.length;i<length;i++) {
  results.push(elems[i]*2);
} // => [2,4,6]
```

## iteratee

对于一个迭代来说，他至少由如下两个部分构成：

- 被迭代集合
- 当前迭代过程

在 `underscore` 中，当前迭代过程是一个函数，他被称为 `iteratee`（直译为被迭代者），他将对当前的迭代元素进行处理。我们看到 `_.map` 的实现：

```
_.map = _.collect = function (obj, iteratee, context) {
  iteratee = cb(iteratee, context);
  var keys = !isArrayLike(obj) && _.keys(obj),
      length = (keys || obj).length,
      results = Array(length); // 定长初始化数组
  for (var index = 0; index < length; index++) {
    var currentKey = keys ? keys[index] : index;
    results[index] = iteratee(obj[currentKey], currentKey, o
bj);
  }
  return results;
};
```

我们传递给 `_.map` 的第二个参数就是一个 `iteratee`，他可能是函数，对象，甚至是字符串，`underscore` 会将其统一处理为一个函数。这个处理由 `underscore` 的内置函数 `cb` 来完成。下面我们看一下 `cb` 的实现：

```
var cb = function (value, context, argCount) {
  // 是否用自定义的iteratee
  if (_.iteratee !== builtinIteratee) return _.iteratee(value,
context);
  // 针对不同的情况
  if (value == null) return _.identity;
  if (_.isFunction(value)) return optimizeCb(value, context, a
rgCount);
  if (_.isObject(value)) return _.matcher(value);
  return _.property(value);
};
```

`cb` 将根据不同情况来为我们的迭代创建一个迭代过程 `iteratee`，服务于每轮迭代：

- `value` 为 `null`

如果传入的 `value` 为 `null`，亦即没有传入 `iteratee`，则 `iteratee` 的行为只是返回当前迭代元素自身，比如：

```
var results = _.map([1,2,3]); // => results: [1,2,3]
```

- `value` 为一个函数

如果传入 `value` 是一个函数，那么通过内置函数 `optimizeCb` 对其进行优化，`optimizeCb` 的作用放到之后讲，先来看个传入函数的例子：

```
var results = _.map([1,2,3], function(value,index,obj) {  
  return '['+obj+']' + '\s '+index+ ' position is '+value;  
});  
// => results: [  
//  "[1,2,3]'s 0 position is 1",  
//  "[1,2,3]'s 1 position is 2",  
//  "[1,2,3]'s 2 position is 3"  
// ]
```

- `value` 为一个对象

如果 `value` 传入的是一个对象，那么返回的 `iteratee` ( `_.matcher` ) 的目的是想要知道当前被迭代元素是否匹配给定的这个对象：

```
var results = _.map([{name:'yoyoyohamapi'}, {name: 'wxj', age:13}],  
  , {name: 'wxj'});  
// => results: [false,true]
```

- `value` 是字面量，如数字，字符串等

如果以上情况都不是，那么传入的 `value` 会是一个字面量（直接量），他指示了一个对象的属性 `key`，返回的 `iteratee` ( `_.property` ) 将用来获得该属性对应的值：

```
var results = _.map([{name:'yoyoyohamapi'}, {name:'wxj'}], 'name')  
;  
// => results: ['yoyoyohamapi', 'wxj'];
```

## 自定义 `iteratee`



在 `cb` 函数的代码中，我们也发现了 `underscore` 支持通过覆盖其提供的 `_.iteratee` 函数来自定义 `iteratee`，更确切的说，来自己决定如何产生一个 `iteratee`：

```
var cb = function (value, context, argCount) {  
  // ...  
  if (_.iteratee !== builtinIteratee) return _.iteratee(value, context);  
  // ...  
}
```

我们看一下 `iteratee` 函数的实现：

```
_.iteratee = builtinIteratee = function (value, context) {  
  return cb(value, context, Infinity);  
};
```

默认的 `_.iteratee` 函数仍然是把生产 `iteratee` 的工作交给 `cb` 完成，并且通过变量 `buildIteratee` 保存了默认产生器的引用，方便之后我们覆盖了 `_.iteratee` 后，`underscore` 能够通过比较 `_.iteratee` 与 `buildIteratee` 来知悉这次覆盖（也就知悉了用户想要自定义 `iteratee` 的生产过程）。

比如当传入的 `value` 是对象时，我们不想返回一个 `_.matcher` 来判断当前对象是否满足条件，而是返回当前元素自身（虽然这么做很无聊），就可以这么做：

```
_.iteratee = function(value, context) {  
  // 现在，value为对象时，也是返回自身  
  if (value == null || _.isObject(value)) return _.identity;  
  if (_.isFunction(value)) return optimizeCb(value, context, argCount);  
  return _.property(value);  
}
```

现在运行之前的例子，看一下有什么不同：

```
var results = _.map([{'name': 'yoyoyohamapi'}, {'name': 'wxj', age: 13}],  
  , {'name': 'wxj'});  
// => results: [{'name': 'yoyoyohamapi'}, {'name': 'wxj', age: 13}];
```

重置默认的 `_.iteratee` 改变迭代过程中的行为只在 `underscore` 最新的 `master` 分支支持，发布版的 1.8.3 并不支持，我们可以看到发布版的 1.8.3 中的 `cb` 代码如下，并没有判断 `_.iteratee` 是否被覆盖：

```
var cb = function (value, context, argCount) {  
  if (value == null) return _.identity;  
  if (_.isFunction(value)) return optimizeCb(value, context, argCount);  
  if (_.isObject(value)) return _.matcher(value);  
  return _.property(value);  
};
```

## optimizeCb

在上面的分析中，我们知道，我们知道，当传入的 `value` 是一个函数时，`value` 还要经过一个叫 `optimizeCb` 的内置函数才能获得最终的 `iteratee`：

```
var cb = function (value, context, argCount) {  
  // ...  
  if (_.isFunction(value)) return optimizeCb(value, context, argCount);  
  // ...  
};
```

顾名思义，`optimizeCb` 有优化回调的意思，所以他是一个对最终返回的 `iteratee` 进行优化的过程，我们看到他的源码：

```
/** 优化回调(特指函数中传入的回调)
 *
 * @param func 待优化回调函数
 * @param context 执行上下文
 * @param argCount 参数个数
 * @returns {function}
 */
var optimizeCb = function(func, context, argCount) {
  // 一定要保证回调的执行上下文存在
  if (context === void 0) return func;
  switch (argCount == null ? 3 : argCount) {
    case 1: return function(value) {
      return func.call(context, value);
    };
    case 2: return function(value, other) {
      return func.call(context, value, other);
    };
    case 3: return function(value, index, collection) {
      return func.call(context, value, index, collection);
    };
    case 4: return function(accumulator, value, index, collection) {
      return func.call(context, accumulator, value, index, collection);
    };
  }
  return function() {
    return func.apply(context, arguments);
  };
};
```

`optimizeCb` 的总体思路就是：传入待优化的回调函数 `func`，以及迭代回调需要的参数个数 `argCount`，根据参数个数分情况进行优化。

- `argCount == 1`，即 `iteratee` 只需要 1 个参数

在 `underscore` 的 `_.times` 函数的实现中，`_.times` 的作用是执行一个传入的 `iteratee` 函数 `n` 次，并返回由每次执行结果组成的数组。它的迭代过程 `iteratee` 只需要 1 个参数 -- 当前迭代的索引：

```
// 执行 iteratee 函数 n 次，返回每次执行结果构成的数组
_.times = function (n, iteratee, context) {
  var accum = Array(Math.max(0, n));
  iteratee = optimizeCb(iteratee, context, 1);
  for (var i = 0; i < n; i++) accum[i] = iteratee(i);
  return accum;
};
```

看一个 `_.times` 的使用例子：

```
function getIndex(index) {
  return index;
}
var results = _.times(3, getIndex); // => [0,1,2]
```

- `argCount == 2`，即 `iteratee` 需要 2 个参数

该情况在 `underscore` 没用使用，所以最新的 `master` 分支已经不再考虑这个参数个数为 2 的情况。

- `argCount == 3`（默认），即 `iteratee` 需要 3 个参数

这 3 个参数是：

- `value`：当前迭代元素的值
- `index`：迭代索引
- `collection`：被迭代集合

在 `_.map`，`_.each`，`_.filter` 等函数中，都是给 `argCount` 赋值了 3：

```
_.each([1,2,3], function(value,index,collection) {  
    console.log("被迭代的集合："+collection+"；迭代索引："+index+"；当前迭代的元素值："+value);  
});  
// =>  
// 被迭代的集合：1,2,3；迭代索引：0；当前迭代的元素值：1  
// 被迭代的集合：1,2,3；迭代索引：1；当前迭代的元素值：2  
// 被迭代的集合：1,2,3；迭代索引：2；当前迭代的元素值：3
```

- `argCount == 4`，即 `iteratee` 需要 4 个参数

这 4 个参数分别是：

- `accumulator`：累加器
- `value`：迭代元素
- `index`：迭代索引
- `collection`：当前迭代集合

那么这个累加器是什么意思呢？在 `underscore` 中的内部函数 `createReducer` 中，就涉及到了 4 个参数的情况。该函数用来生成 `reduce` 函数的工厂，`underscore` 中的 `_.reduce` 及 `_.reduceRight` 都是由它创建的：

```
/**
 * reduce 函数的工厂函数，用于生成一个 reducer，通过参数决定 reduce 的
方向
 * @param dir 方向 left or right
 * @returns {function}
 */
var createReduce = function (dir) {
  var reducer = function (obj, iteratee, memo, initial) {
    var keys = !isArrayLike(obj) && _.keys(obj),
        length = (keys || obj).length,
        index = dir > 0 ? 0 : length - 1;
    // memo 用来记录最新的 reduce 结果
    // 如果 reduce 没有初始化 memo，则默认为首个元素（从左开始则为
第一个元素，从右则为最后一个元素）
    if (!initial) {
      memo = obj[keys ? keys[index] : index];
      index += dir;
    }
    for (; index >= 0 && index < length; index += dir) {
      var currentKey = keys ? keys[index] : index;
      // 执行 reduce 回调，刷新当前值
      memo = iteratee(memo, obj[currentKey], currentKey, o
bj);
    }
    return memo;
  };

  return function (obj, iteratee, memo, context) {
    // 如果参数正常，则代表已经初始化了 memo
    var initial = arguments.length >= 3;
    // reducer 因为引入了累加器，所以优化函数的第三个参数传入了 4，
    // 这样，新的迭代回调第一个参数就是当前的累加结果
    return reducer(obj, optimizeCb(iteratee, context, 4), me
mo, initial);
  };
};
```

我们可以看到，`createReduce` 最终创建的 `reducer` 就是需要一个累加器，该累加器需要被初始化，看一个利用 `_.reduce` 函数求和的例子：

```
var sum = _.reduce([1,2,3,4,5], function(accumulator, value, index, collection){  
    return accumulator+value;  
},0); // => 15;
```

## rest 参数

什么是 rest 参数，就是自由参数，松散参数，这里的自由和松散都是值得参数个数是随意的，与之对应的是 -- 固定参数。

### 从一个加法器开始说起

现在，我们完成一个函数，该函数支持对两个数进行求和，并将结果返回。

```
function add(a,b){  
    return a+b;  
}
```

但我们想对更多的数求和呢？那我们首先想到用数组来传递。

```
function add(numbers){  
    return _.reduce(numbers, function(accum,current){  
        return accum+current;  
    },0);  
}
```

或者直接利用 JavaScript 中的 `arguments`：

```
function add(){  
    var numbers = Array.prototype.slice.call(arguments);  
    return _.reduce(numbers, function(accum,current){  
        return accum+current;  
    },0);  
}  
add(4,3,4,1,1); // => 13
```

现在，我们获得了一个更加自由的加法函数。但是，如果现在的需求变为必须传递至少一个数到加法器呢？



```
function add(a){
  var rest = Array.prototype.slice.call(arguments, 1);
  return _.reduce(rest,function(accum, current){
    return accum+current;
  },a);
}
add(2,3,4,5); // => 14
```

在这个 `add` 实现中，我们已经开始有了 `rest` 参数的雏形，除了自由和松散，`rest` 还有一层意思，就是他的字面意思 -- 剩余，所以在许多语言环境中，`rest` 参数从最后一个形参开始，表示剩余的参数。

## 更理想的方式

然而最后一个 `add` 函数还是把对 `rest` 参数的获取耦合到了 `add` 的执行逻辑中，同时，这样做还会引起歧义，因为在 `add` 函数的使用者看来，`add` 函数似乎只需要一个参数 `a`。而在 `python`，`java` 等语言中，`rest` 参数是需要显示声明的，这种生命能让函数调用者知道哪些参数是 `rest` 参数，比如 `python` 中通过 `*` 标识 `rest` 参数：

```
def add(a, *numbers):
    sum = a
    for n in numbers:
        sum = sum + n * n
    return sum
```

所以，更理想的方式是，提供一个更直观的方式让开发者知道哪个参数是 `rest` 参数，比如，现在有一个函数，其支持 `rest` 参数，那么我们总是假定这类函数的最后一个参数是 `rest` 参数，为此，我们需要创建一个工厂函数，他接受一个现有的函数，包装该函数，使之支持 `rest` 参数：

```
function add(a, rest){
  return _.reduce(rest,function(accum, current){
    return accum+current;
  },a);
}

function genRestFunc(func) {
  // 新返回的函数支持rest参数
  return function(){
    // 获得形参个数
    var argLength = func.length;
    // rest参数的起始位置为最后一个形参位置
    var startIndex = argLength-1;
    // 最终需要的参数数组
    var args = Array(argLength);
    // 设置rest参数
    var rest = Array.prototype.slice.call(arguments, startIndex);
    // 设置最终调用时需要的参数
    for(var i=0;i<startIndex;i++) {
      args[i] = arguments[i]
    }
    args[startIndex] = rest;
    // => args:[a,b,c,d,[rest[0],rest[1],rest[2]] ]
    return func.apply(this, args);
  }
}

addWithRest = genRestFunc(add);

addWithRest(1,2,3,4); // => 10
```

记住，在 JavaScript 中，函数也是对象，并且我们能够通过函数对象的 `length` 属性获得其形参个数

最后，我们来看一下 `underscore` 的官方实现，他暴露了一个 `_.restArgs` 函数，通过给该函数传递一个 `func` 参数，能够使得 `func` 支持 rest 参数：

```
/**
 * 一个包装器，包装函数func，使之支持rest参数
 * @param func 需要rest参数的函数
```

```
* @param startIndex 从哪里开始标识rest参数, 如果不传递, 默认最后一个参数为rest参数
* @returns {Function} 返回一个具有rest参数的函数
*/
var restArgs = function (func, startIndex) {
    // rest参数从哪里开始, 如果没有, 则默认视函数最后一个参数为rest参数
    // 注意, 函数对象的length属性, 揭示了函数的形参个数
    /*
    ex: function add(a,b) {return a+b;}
    console.log(add.length;) // 2
    */
    startIndex = startIndex == null ? func.length - 1 : +startIndex;
    // 返回一个支持rest参数的函数
    return function () {
        // 校正参数, 避免出现负值情况
        var length = Math.max(arguments.length - startIndex, 0);
        // 为rest参数开辟数组存放
        var rest = Array(length);
        // 假设参数从2个开始: func(a,b,*rest)
        // 调用: func(1,2,3,4,5); 实际的调用是:func.call(this, 1,2,[3,4,5]);
        for (var index = 0; index < length; index++) {
            rest[index] = arguments[index + startIndex];
        }
        // 根据rest参数不同, 分情况调用函数, 需要注意的是, rest参数总是最后一个参数, 否则会有歧义
        switch (startIndex) {
            case 0:
                // call的参数一个个传
                return func.call(this, rest);
            case 1:
                return func.call(this, arguments[0], rest);
            case 2:
                return func.call(this, arguments[0], arguments[1], rest);
        }
        // 如果不是上面三种情况, 而是更通用的(应该是作者写着写着发现这个switch case可能越写越长, 就用了apply)
        var args = Array(startIndex + 1);
```

```
// 先拿到前面参数
for (index = 0; index < startIndex; index++) {
    args[index] = arguments[index];
}
// 拼接上剩余参数
args[startIndex] = rest;
return func.apply(this, args);
};

// 别名
_.restArgs = restArgs;
```

测试一下

```
function add(a, rest){
    return _.reduce(rest, function(accum, current){
        return accum+current;
    }, a);
}

var addWithRest = _.restArgs(add);
addWithRest(1, 2, 3, 4); // => 10
```

注意，`restArgs` 函数也是 underscore 最新的 master 分支上才支持的，1.8.3 版本不具备这个功能。

## ES6 中的 rest

现在，最新的 [ES6 标准](#) 已经能够支持 rest 参数，他的用法如下：

```
function f(x, ...y) {
    // y is an Array
    return x * y.length;
}
f(3, "hello", true) == 6
```

所以如果你的项目能够用到 ES6 了，就用 ES6 的写法吧，毕竟他是标准。

# 创建对象的正确姿势

## 无类

对于熟悉面向对象的同学，比如 java 开发者，一开始接触到 JavaScript 会非常懊恼，因为在 JavaScript 中，是没有类的概念的，即便 ES6 引入了

`class`，`extends` 等关键字，那也只是语法糖（syntax sugar），而不能让我们真正创建一个类。我们知道，类的作用就在于继承和派生。作为面向对象的三大特征之一的继承，其优劣在此不再赘述，下面我们看一下如何在缺乏类支持的 JavaScript 实现继承。

## is-a

我们说 A 继承自 B，实际上可以转义为 is-a（什么是什么）关系：A 是 B，比如 Student 继承自 Person，Student 是一个 Person，只不过比 Person 更加具体。

换言之，继承描述了一种层次关系，或者说是一种递进关系，一种更加具体化的递进过程。所以，继承也不真正需要“类”来支撑，他只需要这个递进关系。

JavaScript 中虽然没有类，但是是有对象的概念，我们仍然可以借用对象来描述这个递进关系，只不过 JavaScript 换了一种描述方式，叫做 原型（prototype）。顾名思义，原型描述了一个对象的来由：

原型 ----> 对象

显然，二者就构成了上面我们提到的层次递进关系，在 js 中，原型和对象间的联系链条是通过对象的 `__proto__` 属性来完成的。举个更具体的例子，学生对象（student）的原型是人（person），因为学生源于人，在 JavaScript 中我们可以这样实现二者的递进关系：

```
var person = {
  name: '',
  eat: function() {
    console.log("吃饭");
  }
};

var student = {
  name: 'wxj',
  learn: function() {
    console.log("学习");
  }
};
student.__proto__ = person;
// 由于student is a person，所以他也能够eat
student.eat();
```

但是上面的代码片是存在问题的，他描述的是“某个学生是某个人”。你只需要通过上面的代码片了解如何在 JavaScript 中通过 `__proto__` 实现一种层次递进关系，完成功能的扩展和复用。

## 原型继承

上面例子的继承虽然达到了目的，但是还不是我们熟悉的传统的面向对象的继承的写法，面向对象的继承的应当是“Class extends Class”，而不是上面代码片体现的“object extends object”。在 JavaScript 中，借助于构造函数

（**constructor**），`new` 运算符和构造函数的 `prototype` 属性，我们能够模拟一个类似“Class extends Class”的继承（比如在上例中，我们想要实现“Student extends Person”），这种方式称之为原型继承：

```
// 声明一个叫Person的构造函数，为了让他更像是一个类，我们将其大写
function Person(name) {
    this.name = name;
}

// Student '类'
function Student(name) {
    this.name = name;
}

// 通过函数的prototype属性，我们声明了Person的原型，并且可以在该原型上挂
// 载我们想要的属性或者方法
Person.prototype.eat = function() {
    console.log(this.name+"在吃饭");
}

// 现在让Student来继承Person
Student.prototype = new Person();
// 扩展Student
Student.prototype.learn = function() {
    console.log(this.name+"在学习");
}

// 实例化一个Student
var student = new Student("wxj");

student.eat(); // "wxj在吃饭"
student.learn(); // "wxj在学习"
```

`new Person()` 实际上是自动为我们解决了如下几件事：

- 创建一个对象，并设置其指向的原型：

```
var obj = {'__proto__': Person.prototype};
```

- 调用 `Person()` 构造方法，并且将上下文（`this`）绑定到 `obj` 上，即通过 `Person` 构造 `obj`：



```
Person.apply(obj, arguments);
```

- 返回创建的对象:

```
return obj;
```

所以，`Student.prototype = new Person(); //...; var student = new Student("wxj");` 的等效过程如下：

```
// 继承
Student.prototype = {'__proto__': Person.prototype};
Person.apply(Student.prototype, arguments);
Student.prototype.constructor = Parent;

//...

// 实例化Student
var student = {'__proto__': Student.prototype};
Student.apply(student, "wxj");
student.constructor = Student;
```

那么，我们在调用 `student.eat()` 时，沿着 `__proto__` 提供的线索，最终在 `Person.prototype` 这个原型上找到该方法。

有了这些知识，我们也不难模拟出一个 `new` 来实现对象的创建:

```
function newObj(constructor) {
  var obj = {
    '__proto__': constructor.prototype
  };

  return function() {
    constructor.apply(obj, arguments);
    return obj;
  }
}

// 测试
function Person(name) {
  this.name = name;
}

// Student '类'
function Student(name) {
  this.name = name;
}

Person.prototype.eat = function() {
  console.log(this.name+"在吃饭");
}

// 继承
Student.prototype = newObj(Person)();
// 扩展Student
Student.prototype.learn = function() {
  console.log(this.name+"在学习");
}

// 实例化
var student = newObj(Student)("wxj");
student.eat(); // => "wxj在吃饭"
student.learn(); // => "wxj在学习"
```

## Object.create

另外，ES5 更为我们提供了新的对象创建方式：

```
Object.create(proto, [ propertiesObject ])
```

现在，我们可以这样创建一个继承自 `proto` 的对象：

```
function Person(name) {  
    this.name = name;  
}  
  
Person.prototype.eat = function() {  
    console.log(this.name+"在吃饭");  
}  
  
var student = Object.create(Person.prototype);  
student.name = 'wxj';  
student.eat(); // "wxj在吃饭"
```

在构造对象上，`Object.create(proto)` 的过程如下：

- 创建一个临时的构造函数，并将其原型指向 `proto`：

```
var Temp = function() {}; // 一般会通过闭包将Temp常驻内存，避免每次create时都创建空的构造函数  
Temp.prototype = proto;
```

- 通过 `new` 新建对象，该对象由这个临时的构造函数构造，注意，不会像构造函数传递任何参数：

```
var obj = new Temp();
```

- 清空临时构造函数的原型，并返回创建的对象

```
Temp.prototype = null; // 防止内存泄漏  
return obj;
```

完整的 `Object.create` 参看 [MDN](#)。

## 为什么要用 `Object.create()`

如此看来，`Object.create` 似乎也只是 `new` 的一次包裹，并无任何优势可言。但是，正式这次包裹，使我们新建对象更加灵活。使用 `new` 运算符最大的限制条件是：被 `new` 运算的只能是一个构造函数，如果你想由一个普通对象构造新的对象，使用 `new` 就将会报错：

```
var person = {
  name: '',
  eat: function() {
    console.log(this.name+"在吃饭");
  }
};

var student = new person;
// =>"Uncaught TypeError: person is not a constructor(...)"
```

但是 `Object.create` 就不依赖构造函数，因为在上面对其工作流程的介绍中，我们知道，`Object.create` 内部已经维护了一个构造函数，并将该构造函数的 `prototype` 属性指向传入的对象，因此，他比 `new` 更加灵活：

```
var student = Object.create(person);
student.name = "wxj";
student.eat(); // 'wxj在吃饭'
```

另外，`Object.create` 还能传递第二参数，该参数是一个属性列表，能够初始化或者添加新对象的属性，则更加丰富了创建的对象时的灵活性和扩展性，也正是由此功能，`Object.create` 的内部实现不需要向临时构造函数传递参数：

```
var student = Object.create(person,{
  name: {value:'wxj',writable: false}
});
student.name = "yoyoyo";
student.eat(); // "wxj在吃饭"
```

更多用例参看 [MDN](#)。

## underscore 是如何创建对象的

下面我们来看看 underscore 是怎样创建对象的：

```
var nativeCreate = Object.create;

// ...

// Ctor: 亦即constructor的缩写，这个空的构造函数将在之后广泛用于对象创建，
// 这个做法是出于性能上的考虑，避免每次调用`baseCreate`都要创建空的构造函数

var Ctor = function () {};

// ....

/**
 * 创建一个对象，该对象继承自prototype
 * 并且保证该对象在其原型上挂载属性不会影响所继承的prototype
 * @param {object} prototype
 */
var baseCreate = function (prototype) {
  if (!_.isObject(prototype)) return {};
  // 如果存在原生的创建方法（Object.create），则用原生的进行创建
  if (nativeCreate) return nativeCreate(prototype);
  // 利用Ctor这个空函数，临时设置对象原型
  Ctor.prototype = prototype;
  // 创建对象，result.__proto__ === prototype
  var result = new Ctor;
  // 还原Ctor原型
  Ctor.prototype = null;
  return result;
};
```

我们可以看到，underscore 利用 `baseCreate` 创建对象的时候会先检查当前环境是否已经支持了 `Object.create`，如果不支持，会创建一个简易的 polyfill：

```
// 利用Ctor这个空函数，临时设置对象原型
Ctor.prototype = prototype;
// 创建对象，result.__proto__ === prototype
var result = new Ctor;
// 防止内存泄漏，因为闭包的原因，Ctor常驻内存
Ctor.prototype = null;
```

而之所以叫 `baseCreate`，也是因为其只做了原型继承，而不像 `Object.create` 那样还支持传递属性列表。

## ES6 中的 `class` 及 `extends` 语法糖

在 ES6 中，支持了 `class` 和 `extends` 关键字，让我们在撰写类和继承的时候更加靠近 `java` 等语言的写法：

```
class Person {
  constructor(name){
    this.name=name;
  }

  eat() {
    console.log(this.name+'在吃饭');
  }
}

class Student extends Person{
  constructor(name){
    super(name);
  }

  learn(){
    console.log(this.name+"在学习");
  }
}
// 测试
var student = new Student("wxj");
student.eat(); // "wxj在吃饭"
student.learn(); // "wxj在学习"
```

但要注意，这只是语法糖，ES6 并没有真正实现类的概念。我们看下 Babel（一款流行的 ES6 编译器）对上面程序的编译结果，当中我们能看到如下语句：

```
Object.defineProperty(target, descriptor.key, descriptor);
Object.create();
```

可见，`class` 的实现还是依赖于 ES5 提供的 `Object.defineProperty` 和 `Object.create` 方法。

## 参考资料

- [MDN 继承与原型链](#)
- [Javascript – How Prototypal Inheritance really works](#)

- [JavaScript 中的类继承](#)
- [MDN Object.create](#)



## underscore 集合篇

不同于数学当中的集合，在 `underscore` 中，简单地定义集合为一个可迭代的序列，相较于原生的 ES5 提供的迭代方法，`underscore` 不仅能够对数组进行迭代，还能够对对象进行迭代。

从现在开始，我们将会进入对于 `underscore` 中集合函数的分析和阐述。

## 什么能够被迭代

在 ES5 中，只能对数组对象进行迭代，而 `underscore` 提供的迭代方法，除了支持 `array`，还支持 `object` 的迭代，对 `object` 迭代的依据是对象的键序列 `keys`，我们可以查看 `underscore` 中的 `_.each` 方法：

```
_.each = _.forEach = function (obj, iteratee, context) {  
  // 首先要优化回调过程  
  iteratee = optimizeCb(iteratee, context);  
  var i, length;  
  // 区分数组和对象的迭代过程  
  if (isArrayLike(obj)) {  
    for (i = 0, length = obj.length; i < length; i++) {  
      // 数组的迭代回调传入三个参数(迭代值, 迭代索引, 迭代对象)  
      iteratee(obj[i], i, obj);  
    }  
  } else {  
    var keys = _.keys(obj);  
    for (i = 0, length = keys.length; i < length; i++) {  
      // 对象的迭代回调传入三个参数(迭代值, 迭代的key, 迭代对象)  
      iteratee(obj[keys[i]], keys[i], obj);  
    }  
  }  
  // 返回对象自身, 以便进行链式构造  
  return obj;  
};
```

可以看到，`array` 是直接迭代数组的每个元素，而 `object` 迭代的元素是对象的每个 `k-v` 对。看一下用例：

```
var obj = {  
  name: 'wxj',  
  age: 13,  
  sex: 'male'  
};  
  
_.each(obj, function(value, key, obj){  
  console.log('['+key+'] is:', value);  
});  
  
// name's value is: wxj  
// ...  
// ...
```

# map-reduce

## map，reduce 是两个独立的过程

如果你有一定的 JavaScript 编程经历，那么你一定已经接触过

`Array.prototype` 提供的 `map` 和 `reduce` 函数。现在你需要明白的是，二者不仅仅是存在于 JavaScript 的两个 API，更是函数式编程语言的重要组成部分，是一种对列表的操作思路。

map-reduce 由如下两个独立的部分组成：

- **map**（映射）：一个映射过程就是将各个元素，按照一定的规则，逐个映射为新的元素。这是一个 一一对应 的过程。用数学公式描述就是（其中，函数  $f$  就是这个规则）：

$$\begin{bmatrix} newElem1 \\ newElem2 \\ newElem3 \\ \dots \\ newElemN \end{bmatrix} = f\left(\begin{bmatrix} elem1 \\ elem2 \\ elem3 \\ \dots \\ elemN \end{bmatrix}\right)$$

- **reduce**（规约）：一个规约过程仍然需要迭代指定列表的每个元素，然后仍然按照一定规则，合并这些元素到一个目标对象上。这是一个 由多至一 的过程，或者说是一个 逐步累积 的过程：

$$newElem = f\left(\begin{bmatrix} elem1 \\ elem2 \\ elem3 \\ \dots \\ elemN \end{bmatrix}\right)$$

## map 在 underscore 中的实现

map 的实现思路如下：

- 创建一个新列表或者元素

- 遍历原列表或者原对象的值，用指定的函数 `func` 作用于每个遍历到的元素，输出一个新的元素放入新列表或者对象中

```
_.map = _.collect = function (obj, iteratee, context) {  
  iteratee = cb(iteratee, context);  
  // 同样, 根据obj是对象还是数组分别考虑  
  var keys = !isArrayLike(obj) && _.keys(obj),  
      length = (keys || obj).length,  
      results = Array(length); // 定长初始化数组  
  for (var index = 0; index < length; index++) {  
    var currentKey = keys ? keys[index] : index;  
    results[index] = iteratee(obj[currentKey], currentKey, o  
bj);  
  }  
  return results;  
};
```

使用用例：

- 对数组使用 `_.map` 函数：

```
var array = [1,2,3,4,5];  
var doubledArray = _.map(array, function(elem, index, array){  
  return 2*elem;  
}); // => doubledArray: [2,4,6,8,10]
```

- 对一般对象使用 `_.map` 函数：

```
var obj = {  
  name: 'wxj',  
  age: 13,  
  sex: 'male'  
};  
var wxjInfos = _.map(obj, function(value ,key, obj){  
  return [key, value].join(':');  
}); // => wxjInfos: ['name:wxj', 'age:13', 'sex:male']
```

## reduce 在 underscore 中的实现

相较于 map，reduce 函数在 underscore 中的实现更为复杂一些，因此，underscore 而外撰写一个内部函数 `createReducer` 用来创建 reduce 函数，其完成了如下几件事：

- 区分 reduce 的方向 `dir`，是从序列 开端 开始做规约过程，还是从序列 末端 开始做规约过程。
- 判断用户在使用 `_.reduce` 或者 `_.reduceRight` 时，是否传入了第三个参数，即是否传入了规约起点，判断结果又 `initial` 变量进行标识。

而对于一个 reduce 函数来说，其执行过程大致如下：

- 设置一个变量 `memo` 用以缓存当前当前的规约过程的结果，如果用户未初始化 `memo`，则 `memo` 为序列的一个参数 - 遍历当前集合，对最近迭代到的元素按传入的 `func` 进行规约操作，刷新 `memo`。
- 规约过程完成，返回 `memo`。

```

var createReduce = function (dir) {
  // Wrap code that reassigns argument variables in a separate function than
  // the one that accesses `arguments.length` to avoid a perf hit. (#1991)
  var reducer = function (obj, iteratee, memo, initial) {
    var keys = !isArrayLike(obj) && _.keys(obj),
        length = (keys || obj).length,
        index = dir > 0 ? 0 : length - 1;
    // 如果reduce没有初始化memo, 则默认为首个元素(从左开始则为第一个元素,从右则为最后一个元素)
    if (!initial) {
      memo = obj[keys ? keys[index] : index];
      index += dir;
    }
    for (; index >= 0 && index < length; index += dir) {
      var currentKey = keys ? keys[index] : index;
      // 执行reduce回调,刷新当前值
      memo = iteratee(memo, obj[currentKey], currentKey, obj);
    }
    return memo;
  };

  return function (obj, iteratee, memo, context) {
    // 如果参数正常,则代表已经初始化了memo
    var initial = arguments.length >= 3;
    // 所有的传入回调都要通过optimizeCb进行优化,
    // reducer因为引入了累加器,所以优化函数的第三个参数传入了4,
    // 这样,新的迭代回调第一个参数就是当前的累加结果:
    // _.reduce([1,2,3],function(prev,current){})
    return reducer(obj, optimizeCb(iteratee, context, 4), memo, initial);
  };
};

```

最终，underscore 暴露给了 2 个方向的 reduce API 给用户：

```
// 由左至右进行规约
_.reduce = _.foldl = _.inject = createReduce(1);
// 由右至左进行规约
_.reduceRight = _.foldr = createReduce(-1);
```

使用用例：

- 对数组使用 `_.reduce`

```
var array = [1, 2, 3, 4, 5];
var sum = _.reduce(array, function(prev, current){
  return prev+current;
}, 0);
// => sum: 15
```

- 一般对象也可以进行 `_.reduce`

```
var scores = {
  english: 93,
  math: 88,
  chinese: 100
};
var total = _.reduce(scores, function(prev, value, key){
  return prev+value;
}, 0);
// => total: 281
```



# 真值检测函数

## 概述

在 `underscore` 中，除了 `_.each`、`_.map`、`_.reduce` 等函数操作集合，还提供了 `_.filter`、`_.reject`、`_.every`、`_.some` 这几个基于逻辑判断的集合操作函数。无一例外的是，这些函数都依赖于用户提供的真值检测函数用来判断当前迭代元素是否满足条件。

以 `_.filter` 的实现为例，`underscore` 将真值检测函数参数命名为 `predicate`，十分形象。`predicate` 仍然会被函数 `cb` 优化，如果忘记 `cb` 函数的意图和流程，请回过头来看 [underscore 基础篇 -- 迭代！迭代！迭代！](#)

```
_.filter = _.select = function (obj, predicate, context) {  
  var results = [];  
  predicate = cb(predicate, context);  
  _.each(obj, function (value, index, list) {  
    if (predicate(value, index, list)) results.push(value);  
  });  
  return results;  
};
```

根据传入的元素信息，真值检测函数返回逻辑值 `true` 或者 `false`，决定当前迭代元素是否需要被淘汰。更加方便的是，`underscore` 还提供了 `_.negate` 用来反义 `predicate` 的执行结果：

```
_.negate = function (predicate) {  
  return function () {  
    return !predicate.apply(this, arguments);  
  };  
};
```

这也是一个函数层面的抽象，如果不提供 `_.negate`，我们要实现与 `_.filter` 反义的 `_.reject` 函数，就需要这样书写：

```
_.reject = function (obj, predicate, context) {  
  var results = [];  
  predicate = cb(predicate, context);  
  _.each(obj, function (value, index, list) {  
    if (!predicate(value, index, list)) results.push(value);  
  });  
  return results;  
};
```

显然，与 `_.filter` 有太多重复代码了。

而有了 `_.negate` 这个抽象，实现 `_.reject` 我们只需要一行代码：

```
_.reject = function (obj, predicate, context) {  
  return _.filter(obj, _.negate(cb(predicate)), context);  
};
```

## `_.filter`

`_.filter(coll, predicate)`：保留 `coll` 中满足 `predicate` 的元素

```
var array = [12, 50, 2, 4, 6, 13, 12];  
var lessThan10 = _.filter(array, function(elem){  
  return elem < 10;  
});  
// => lessThan10: [2, 4, 6]
```

## `_.reject`

`_.reject(coll, predicate)`：根据 `predicate`，排除 `coll` 中的元素。

```
var array = [12, 50, 2, 4, 6, 13, 12];  
var lessThan10 = _.reject(array, function(elem) {  
  return elem >= 10;  
}); // => lessThan10: [2, 4, 6]
```

## **\_\_.every = \_\_.all**

`__.every(coll, predicate)` : 判断 `coll` 的 每个元素 是否都满足 `predicate`

```
var array = [12, 50, 2, 4, 6, 13, 12];
var everyLessThan10 = __.every(array, function(elem){
  return elem < 10;
});
// everyLessThan10: false
```

## **\_\_.some = \_\_.all**

`__.some(coll, predicate)` : 判断 `coll` 的 某个元素 是否满足 `predicate`

```
var array = [12, 50, 2, 4, 6, 13, 12];
var someLessThan10 = __.every(array, function(elem){
  return elem < 10;
});
// someLessThan10: true
```

## 查询

这里提到的查询主要是指完成如下意图的查询: 如果集合是数组:

1. 根据给定元素或者查询条件, 查询元素在数组中的位置
2. 根据查询条件, 获得元素或者元素集合
3. 仅判断元素是否存在

对于元素位置查询, underscore 提供了以下 API:

- `_.indexOf`
- `_.lastIndexOf`
- `_.findIndex`
- `_.findLastIndex`
- `_.sortedIndex`

`_.indexOf` 及 `_.lastIndexOf` 只支持对于数组元素的搜索。

对于元素查询, underscore 提供了以下 API:

- `_.find = _.detect`
- `_.findWhere`
- `_.where`

如果集合是对象, 即集合是键值对构成的, 则提供了以下 API:

- `_.findKey`
- `_.pluck`

对于判断元素是否存在, underscore 提供了以下 API:

- `_.contains`

## createIndexFinder

underscore 中通过内置的工厂函数 `createIndexFinder` 来创建一个索引查询器, `_.indexOf` 及 `_.lastIndexOf` 正是由该函数所创建的。

`createIndexFinder(dir, predicateFind, sortedIndex)` 接受 3 个参数:

- `dir` : 查询方向, `_.indexOf` 即是正向查询, `_.lastIndexOf` 即是反向查询。
- `predicateFind` : 真值检测函数, 该函数只有在查询元素不是数字 (`NaN`) 才会使用。
- `sortedIndex` : 有序数组的索引获得函数。如果设置了该参数, 将假定数组已经有序, 从而更加高效的通过针对有序数组的查询函数 (比如二分查找等) 来优化查询性能。

```
var createIndexFinder = function (dir, predicateFind, sortedIndex) {
  return function (array, item, idx) {
    var i = 0, length = getLength(array);
    // 如果设定了查询起点, 且查询起点格式正确(数字)
    if (typeof idx == 'number') {
      // 校正查询起点
      if (dir > 0) {
        i = idx >= 0 ? idx : Math.max(idx + length, i);
      } else {
        length = idx >= 0 ? Math.min(idx + 1, length) :
idx + length + 1;
      }
    } else if (sortedIndex && idx && length) {
      // 如果传递sortedIndex函数, 则先假设array为排序好的, 获得i
tem在array中的位置
      idx = sortedIndex(array, item);
      // 验证这个假设是否正确
      return array[idx] === item ? idx : -1;
    }
    // 如果待查找item不是数字, 是NaN(JS中, NaN===NaN 为false), 需
要通过predicateFind来查找
    if (item !== item) {
      idx = predicateFind(slice.call(array, i, length), _
isNaN);
      return idx >= 0 ? idx + i : -1;
    }
    // 否则直接通过 === 进行查找
    for (idx = dir > 0 ? i : length - 1; idx >= 0 && idx < l
ength; idx += dir) {
      if (array[idx] === item) return idx;
    }
    return -1;
  };
};
```

`createIndexFinder` 将会返回一个索引查询器, 该索引查询器支持三个参数:

- `array` : 待搜索数组

- `item` : 待搜索对象
- `idx` : 查询起点，从数组的哪个位置开始查找。如果以数字的方式设置了查询起点，或者未设置查询起点，则无法使用 `sortedIndex` 方法进行查询优化。通常，我们可以设置该值为语义更加明显的 `true`（代表启用查询优化）来对有序数组进行查询优化。

## `_.indexOf`

`_.indexOf(array, item, sorted)` : 查询 `array` 在 `coll` 中第一次出现的位置。

源码：

```
_.indexOf = createIndexFinder(1, _.findIndex, _.sortedIndex);
```

值得一提的是，`_.indexOf` 方法的创建过程中被传递了 `_.findIndex` 作为元素的真值预测函数，以及 `_.sortedIndex` 作为当数组有序时获得索引的方式。这两个函数将在之后介绍。

用例：

```
// 创建一个有序的大容量数组
var array = [];
for(var i=0;i < 1000000;i++) {
    array[i] = i;
}

console.time("以数字方式设置了查询起点，搜索耗时");
_.indexOf(array, 500000);
console.timeEnd("以数字方式设置了查询起，搜索耗时");
// 以数字方式设置了查询起，搜索耗时：1.561ms

console.time("以非数字方式设置了查询起点，搜索耗时");
_.indexOf(array, 500000, true);
console.timeEnd("以非数字方式设置了查询起点，搜索耗时");
// 以非数字方式设置了查询起点，搜索耗时：0.308ms
```

可以看到，经 `_.sortedIndex` 优化后的 `_.indexOf` 搜索性能更高。

## `_.lastIndexOf`

`_.lastIndexOf(array, item, sorted)`：查询 `item` 在 `array` 中最后一次出现的位置。

源码：

```
_.lastIndexOf = createIndexFinder(-1, _.findLastIndex);
```

用例：

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

注意，这些查找的索引的函数的都是非贪婪的，一旦查找到，立即返回索引并停止查找。

## `_.sortedIndex`

`_.sortedIndex(array, obj, iteratee)`：根据比较条件 `iteratee`，查询 `obj` 在 `array` 中的位置，如果查询失败，则返回 `obj` 应当出现的位置。

源码：



```
_.sortedIndex = function (array, obj, iteratee, context) {
  iteratee = cb(iteratee, context, 1);
  var value = iteratee(obj);
  var low = 0, high = getLength(array);
  while (low < high) {
    var mid = Math.floor((low + high) / 2);
    if (iteratee(array[mid]) < value) low = mid + 1; else high = mid;
  }
  return low;
};
```

`_.sortedIndex` 使用了二分查找作为查找算法。

用例：

```
_.sortedIndex([10, 20, 30, 40, 50], 20); // => 1

// `_.sortedIndex` 如果查找的元素不存在，将返回元素应当存在的位置
_.sortedIndex([10, 20, 30, 40, 50], 35); // => 3

// `_.sortedIndex` 也支持对对象集合的搜索。
_.sortedIndex([{name: 'wxj'}, {name: 'lx'}, {name: 'lcx'}, {name: 'wxj'}]);
// => 0
```

## createPredicateIndexFinder

除了 `createIndexFinder` 以外，`underscore` 还内置了一个

`createPredicateIndexFinder` 的工厂函数，该函数甚至可以看做是 `createIndexFinder` 的增强版，因为其不仅能查询直接量在集合中的位置，也支持通过一个真值检测函数查找位置。`createPredicateIndexFinder` 接受 1 个参数：

- `dir`：搜索方向

```
var createPredicateIndexFinder = function (dir) {  
  /**  
   * 返回的位置查询函数  
   * @param array 待搜索数组  
   * @param predicate 真值检测函数  
   * @param context 执行上下文  
   */  
  return function (array, predicate, context) {  
    predicate = cb(predicate, context);  
    var length = getLength(array);  
    var index = dir > 0 ? 0 : length - 1;  
    for (; index >= 0 && index < length; index += dir) {  
      // 只找到第一次满足条件的位置  
      if (predicate(array[index], index, array)) return index;  
    }  
    return -1;  
  };  
};
```

它将返回一个索引查询函数，该查询函数接受 3 个参数：

- `array` ：待搜索数组
- `predicate` ：真值检测函数
- `context` ：执行上下文

从源码中可以看到，现在，索引的查询是依赖于传入的真值检测函数

`predicate`。并且，在[迭代！迭代！迭代！](#)中，我们知道，如果传入的 `predicate` 是一个立即数，会被 `cb` 优化为一个 `_.property(predicate)` 函数，用来获得对象的某个属性。

## `_.findIndex`

`_.findIndex(array, predicate)` ：根据条件 `predicate`，查询元素在 `array` 中出现的位置。

源码：

```
_.findIndex = createPredicateIndexFinder(1);
```

用例：

```
// 下面的调用将不会返回3，因为`12`会被修正为`_.property(12)`：
_.findIndex([4, 6, 8, 12], 12);
// => -1

_.findIndex([4, 6, 8, 12], function(value){
  return value===0;
}); // => 3

_.findIndex([{name: 'wxj'}, {name: 'zxy'}], {
  name: 'zxy'
}); // => 1
```

## \_.findLastIndex

`_.findLastIndex(array, predicate)`：根据条件 `predicate`，查询元素在 `array` 中出现的位置。

源码：

```
_.findLastIndex = createPredicateIndexFinder(-1);
```

用例：

```
_.findLastIndex([4, 6, 8, 12, 5, 12], function(value){
  return value===12;
}); // => 5

_.findLastIndex([{name: 'wxj'}, {name: 'zxy'}, {name: 'zxy'}], {
  name: 'zxy'
}); // => 2
```

## \_.findKey

`_.findKey(obj, predicate)` : 返回 `obj` 中第一个满足条件的 `predicate` 的 `key`。

源码：

```
_.findKey = function (obj, predicate, context) {  
  predicate = cb(predicate, context);  
  var keys = _.keys(obj), key;  
  for (var i = 0, length = keys.length; i < length; i++) {  
    key = keys[i];  
    if (predicate(obj[key], key, obj)) return key;  
  }  
};
```

用例：

```
var student = {  
  name: 'wxj',  
  age: 18  
};  
  
_.findKey(student, function(value, key, obj) {  
  return value === 18;  
});  
// => "age"
```

## \_.pluck

`_.pluck(obj, key)` : 取出 `obj` 中 `key` 对应的值。

源码：

```
_.pluck = function (obj, key) {  
    // 迭代集合，每个迭代元素返回其对应属性的对应值  
    return _.map(obj, _.property(key));  
};
```

用例：

```
var students = [  
    {name: 'wxj', age: 18},  
    {name: 'john', age: 14},  
    {name: 'bob', age: 23}  
];  
  
_.pluck(students, 'name');  
// ["wxj", "john", "bob"]
```

---

## \_.find = \_.detect

\_.find(obj, predicate) : obj 中满足条件 predicate 的元素。

源码：

```
_.find = _.detect = function (obj, predicate, context) {  
    var keyFinder = isArrayLike(obj) ? _.findIndex : _.findKey;  
    var key = keyFinder(obj, predicate, context);  
    if (key !== void 0 && key !== -1) return obj[key];  
};
```

可以看到，`_.find` 既能检索数组（利用 `_.findIndex` 先确定元素下标），又能检索对象（利用 `_.findKey` 先确定 key）。

用例

```
var obj = {
  name: 'wxj',
  age: 18,
  height: 163
};

var arr = [
  { name: 'wxj', age: 18},
  { name: 'zxy', age: 44}
];

_.find(obj, function(value, key, obj){
  return value%2 === 0;
});
// => 18

_.find(arr, function(elem) {
  return elem.name === 'wxj';
});
// => { name: 'wxj', age: 18}
```

## where

在 SQL 中，我们通常会利用到 `where` 关键字：

```
select * from users where password="123456" and username="wxj"
```

那么在 JavaScript 中，我们想要模拟一个 `where` 函数，就需要为其传递两个参数：

- `table` : 集合 `users`
- `attrs` : 属性匹配列表 `{ password: '123456', username='wxj'}`

并且注意到，`where` 的核心过程仍然是集合过滤，所以 `Array.prototype.filter` 将是 `where` 的核心：

```
function where(table, attrs) {
  // 遍历table
  return table.filter(function(elem){
    // 观察当前遍历到的对象是否满足where条件
    return Object.keys(attrs).every(function(attr){
      return elem[attr] === attrs[attr];
    });
  });
}

// 测试
var users = [
  {name: 'wxj', age: 18, sex: 'male'},
  {name: 'zxy', age: 18, sex: 'male'},
  {name: 'zhangsan', age: 14, sex: 'famale'}
];

var ret = where(users, {age: 18, sex: 'male'});
// => [
//   {name: 'wxj', age: 18, sex: 'male'},
//   {name: 'zxy', age: 18, sex: 'male'},
// ]
```

## \_.where

`_.where(obj, attrs)`：类似于 SQL 中的 `where` 限定符，获得满足 `attrs` 的元素。

源码：

```
_.where = function (obj, attrs) {
  return _.filter(obj, _.matcher(attrs));
};
```

用例：

```
var users = [
  {name: 'wxj', age: 18, sex: 'male'},
  {name: 'zxy', age: 18, sex: 'male'},
  {name: 'zhangsan', age: 14, sex: 'famale'}
];

var ret = _.where(users, {age: 18, sex: 'male'});
// => [
//   {name: 'wxj', age: 18, sex: 'male'},
//   {name: 'zxy', age: 18, sex: 'male'},
// ]
```

`_.where` 的实现十分简洁，利用到了 `_.filter` 进行迭代，真值检测函数用到了 `_.matcher`。因此，有必要看一下 `_.matcher` 函数是干什么的。

## `_.matcher`

`_.matcher(obj, attrs)`：返回一个校验过程，用以校验对象的属性是否匹配给定的属性列表。

源码：

```
_.matcher = _.matches = function (attrs) {
  attrs = _.extendOwn({}, attrs);
  return function (obj) {
    return _.isMatch(obj, attrs);
  };
};
```

用例：



```
var users = [
  {name: 'wxj', age: 18, sex: 'male'},
  {name: 'zxy', age: 18, sex: 'male'},
  {name: 'zhangsan', age: 14, sex: 'famale'}
];

var matcher = _.matcher({age: 18, sex: 'male'});

var ret = _.filter(users, matcher);
// => [
//   {name: 'wxj', age: 18, sex: 'male'},
//   {name: 'zxy', age: 18, sex: 'male'},
// ]
```

可以看到，`_.matcher` 接受传入的属性列表 `attrs`，最终返回一个校验过程，通过 `_.isMatch` 来校验 `obj` 中属性的是否与 `attrs` 的属性相匹配。

## \_.isMatch

`_.isMatch(obj, attrs)`：判断 `obj` 是否满足 `attrs`。

源码：

```
_.isMatch = function (object, attrs) {
  var keys = _.keys(attrs), length = keys.length;
  if (object == null) return !length;
  var obj = Object(object);
  for (var i = 0; i < length; i++) {
    var key = keys[i];
    // 一旦遇到value不等，或者attrs中的key不在obj中，立即返回false

    if (attrs[key] !== obj[key] || !(key in obj)) return false;
  }
  return true;
};
```

用例：

```
var users = [
  {name: 'wxj', age: 18, sex: 'male'},
  {name: 'zxy', age: 18, sex: 'male'},
  {name: 'zhangsan', age: 14, sex: 'famale'}
];
_.isMatch(users[1], {age: 18, sex: 'male'}); // => true
_.isMatch(users[2], {age: 18, sex: 'male'}); // => false
```

## \_.findWhere

`_.findWhere(obj, attrs)`：与 `where` 类似，但只返回第一条查询到的记录。

源码：

```
_.findWhere = function (obj, attrs) {
  return _.find(obj, _.matcher(attrs));
};
```

用例：

```
var users = [
  {name: 'wxj', age: 18, sex: 'male'},
  {name: 'zxy', age: 18, sex: 'male'},
  {name: 'zhangsan', age: 14, sex: 'famale'}
];

var ret = _.findWhere(users, {age: 18, sex: 'male'});
// => { name: 'wxj', age: 18, sex: 'male' }
```

## \_.contains = \_.includes = \_.include

`_.contains(obj, item, fromIndex)`：判断 `obj` 是否包含 `item`，可以设置查询起点 `fromIndex`。

源码：

```
_.contains = _.includes = _.include = function (obj, item, fromIndex, guard) {  
  // 如果不是数组，则根据值查找  
  if (!isArrayLike(obj)) obj = _.values(obj);  
  if (typeof fromIndex !== 'number' || guard) fromIndex = 0;  
  return _.indexOf(obj, item, fromIndex) >= 0;  
};
```

用例：

```
_.contains([1, 2, 3, 4, 5], 2);  
// => true  
  
_.contains([1, 2, 3, 4, 5], 2, 4);  
// => false  
  
_.contains({name: 'wxj', age: 13, sex: 'male'}, 'male');
```

# 随机取样与洗牌算法

## 洗牌算法

在一些场景下，我们需要抽样一个序列的部分，一般来说，我们会需要抽样过程是随机的，假定样本容量为  $N$ ，序列下标为  $0 \dots M$ ，且  $N \leq M$ ，那么我们这样完成抽样过程 1. 随机生成  $N$  个序号 2. 根据序号，逐个从序列中取出元素

可以看到，过程很简单，唯一需要被设计的地方就是 -- 如何均匀地生成随机序号，这样才有助于我们获得一个高质量的样本。换一种视角，我们获得的样本集合就是原集合的部分乱序结果。例如：

原集合： `[1, 2, 3, 4, 5]`

乱序结合（假设  $N = 3$ ）： `[5, 2, 1, 3, 4]`

样本直接就可以来自于乱序集合的前三个： `[5, 2, 1]`

这就是最常见的乱序数组的算法 -- [洗牌算法](#)。

算法的思路在宏观上可以概括为：将集合视为牌堆，不停地从牌堆中抽牌构成新的牌堆，直至新牌堆的牌数到达预设数量。我们尝试用 JavaScript 模拟一下这个算法：

```
// 洗牌算法
function shuffle(set, n) {
  var length = set.length;
  var i;
  // 初始化sample
  var sample = set.map(function(elem){
    return elem;
  });
  // 最后一个元素的位置
  var last = length - 1;
  // 洗到第n个就可以停止
  for(i=0; i<n; i++) {
    // 从剩余未乱序的集合中选出一个随机位置
    var randIndex = Math.floor(Math.random()*(last-i+1)+i);
    // 交换随机位置上与当前位置的值，完成乱序
    var tmp = sample[i];
    sample[i] = sample[randIndex];
    sample[randIndex] = tmp;
  }
  // 返回前n个样本
  return sample.slice(0, n);
}

// 测试
var set = [1,2,3,4,5,7,8,9,10];

var newSet = shuffle(set4);
// => [2, 8, 3, 5]
```

## **\_.sample**

`_.sample(coll, n)` : 从 `coll` 随机取出 `n` 个样本。

`underscore` 中的抽样函数正基于洗牌算法的，我们看到源码：

```
_.sample = function (obj, n, guard) {
  if (n == null || guard) {
    if (!isArrayLike(obj)) obj = _.values(obj);
    return obj[_.random(obj.length - 1)];
  }
  // 如果是对象, 乱序key的排列
  var sample = isArrayLike(obj) ? _.clone(obj) : _.values(obj);
  ;
  var length = getLength(sample);
  // 校正参数n, 使得0<=n<length
  n = Math.max(Math.min(n, length), 0);
  var last = length - 1;
  // 开始洗牌算法, 洗出来n个就停止了
  for (var index = 0; index < n; index++) {
    // 从[index, last]即剩余未乱序部分获得一个随机位置
    var rand = _.random(index, last);
    // 交换当前值与随机位置上的值
    var temp = sample[index];
    sample[index] = sample[rand];
    sample[rand] = temp;
    // 此时, 排序后的第一个数据sample[0]已经确定
  }
  return sample.slice(0, n);
};
```

有两点值得注意：

- 如果是对一个对象进行抽样，那么是对这个对象的值集合进行抽样。
- 如果没有设置 `n`，则随机返回一个元素，不进行集合乱序。

用例：

```
_.sample([1, 2, 3, 4, 5, 6], 3);  
// => [1, 6, 2]  
  
_.sample([1, 2, 3, 4, 5, 6]);  
// => 4  
  
_.sample({name: 'wxj', age: 13, sex: 'male'}, 2);  
// [13, "wxj"]
```

## \_.shuffle

\_.shuffle(coll) : 获得 coll 乱序副本。

基于 `_.sample`，underscore 还提供了一个函数用来直接返回一个乱序后的集合副本，这也体现了 `_.sample` 不直接返回元素，而是乱序数组的带来的复用好处：

源码：

```
_.shuffle = function (obj) {  
  return _.sample(obj, Infinity);  
};
```

用例：

```
_.shuffle([1, 2, 3, 4, 5, 6]);  
// => [2, 4, 3, 5, 6, 1]  
  
_.shuffle({name: 'wxj', age: 13, sex: 'male'});  
// => ["wxj", "male", 13]
```

## 更好用的排序

### `Array.prototype.sort`

在 ES5 中，已经对集合提供一个排序方法 `Array.prototype.sort`：

```
var students = [
  {name: 'wxj', age: 18},
  {name: 'john', age: 14},
  {name: 'bob', age: 23}
];

var sortedStudents = students.sort(function(left, right) {
  return right.age < left.age;
});

// => sortStudents: [
//   {name: 'john', age: 14},
//   {name: 'wxj', age: 18},
//   {name: 'bob', age: 23},
// ]
```

### `_.sortBy`

`_.sortBy(obj, iteratee)`：根据比较条件 `iteratee`，对 `obj` 进行排序。

源码：



```

_.sortBy = function (obj, iteratee, context) {
  var index = 0;
  iteratee = cb(iteratee, context);
  // 先通过map生成新的对象集合, 该对象提供了通过iteratee计算后的值, 方便
  排序
  // [{value:1,index:0,criteria: sin(1)}, ...]
  // 再排序.sort
  // 最后再通过pluck把值摘出来
  return _.pluck(_.map(obj, function (value, key, list) {
    return {
      value: value,
      index: index++,
      criteria: iteratee(value, key, list)
    };
  })).sort(function (left, right) {
    var a = left.criteria;
    var b = right.criteria;
    if (a !== b) {
      if (a > b || a === void 0) return 1;
      if (a < b || b === void 0) return -1;
    }
    return left.index - right.index;
  })), 'value');
};

```

下面解释一下 `_.sortBy` 的工作流程：

- 通过 `_.map` 生成一个新的集合，该集合的每个元素是一个对象，它具有三个属性：
    - `value` ： 元素值
    - `index` ： 索引位置
    - `criteria` ： 排序准则，该准则将通过被优化的 `iteratee` 计算得到。
- `underscore` 看到了元素间的比较仍将落脚到“值比较”的本质。

假设我们现在有集合：

```
var students = [  
  {name: 'wxj', age: 18},  
  {name: 'john', age: 14},  
  {name: 'bob', age: 23}  
];
```

假设我们需要按照年龄进行排序，那么传入的 `iteratee` 为：

```
var iteratee = function(value, key, index, elem) {  
  return elem.age;  
}
```

经过该过程，该集合将变为：

```
var newStudents = [  
  {  
    value: {name: 'wxj', age: 18},  
    index: 0,  
    criteria: 18  
  },  
  {  
    value: {name: 'john', age: 14},  
    index: 0,  
    criteria: 14  
  },  
  {  
    value: {name: 'bob', age: 23},  
    index: 0,  
    criteria: 23  
  }  
];
```

- 利用 `Array.prototype.sort` 以及我们确定的排序准则 `criteria` 对新生成的集合进行排序：

```
var sortedStudents = newStudents.sort(function (left, right) {
  var a = left.criteria;
  var b = right.criteria;
  if (a !== b) {
    if (a > b || a === void 0) return 1;
    if (a < b || b === void 0) return -1;
  }
  return left.index - right.index;
});

// => sortedStudents: [
//   {
//     value: {name: 'john', age: 14},
//     index: 0,
//     criteria: 14
//   },
//   {
//     value: {name: 'wxj', age: 18},
//     index: 0,
//     criteria: 18
//   },
//   {
//     value: {name: 'bob', age: 23},
//     index: 0,
//     criteria: 23
//   }
// ];
```

- 再通过 `_.pluck` 取出 `value` 属性，过滤掉不需要的 `index` 及 `criteria` 属性：

```
var ret = _.pluck(sortedStudents, 'value');
// => ret: [
//   {name: 'john', age: 14},
//   {name: 'wxj', age: 18},
//   {name: 'bob', age: 23},
// ]
```

最后，我们看一看 `_.sortBy` 带来的便捷性：

```
var students = [  
  {name: 'wxj', age: 18},  
  {name: 'john', age: 14},  
  {name: 'bob', age: 23}  
];  
  
var sortedStudents = _.sortBy(students, 'age');  
  
// => sortStudents: [  
//   {name: 'john', age: 14},  
//   {name: 'wxj', age: 18},  
//   {name: 'bob', age: 23},  
// ]
```

## 模拟一段 SQL

有过关系型数据库开发经验的同学一定不会对 SQL 语句陌生，看一条最熟悉的 SQL 语句

```
SELECT username, id FROM users where age<30 group by sex;
```

下面会分别通过 面向对象 的思维和 函数式编程 的思维来模拟上面语句。

## 面向对象（OO）

在面向对象的世界中，我们首先需要有一个类，我们称之为 `SQL`，实例化 `SQL` 对象时，我们需要传递给他 `table` 表明需要操作的表。并且，我们向该“类”的原型中添加 `where`，`select` 等方法，为了方便，我们还支持到链式调用：

```
function SQL(table){
  this.table = table;
  this._result = null;
  this._getRows = function() {
    return this._result?this._result: this.table;
  }

  this._doSelect = function(rows, keys) {
    return rows.map(function(row) {
      return _.keys(row).reduce(function(elem, key) {
        if (_.indexOf(keys, key) > -1) {
          elem[key] = row[key];
        }
        return elem;
      }, {});
    });
  }
}
```

// 投影

```
SQL.prototype.select = function(keys) {
  var rows = this._getRows();
  if (_.isArray(rows)){
    this._result = this._doSelect(rows, keys);
  } else if (_.isObject(rows)) {
    this._result = _.keys(rows).map(function(key) {
      return this._doSelect(rows[key], keys);
    });
  }
  return this;
}

// 限定，这里偷点懒，我们还是用到了高阶函数
// 直接传递字符串进行parse太复杂
SQL.prototype.where = function(predicate) {
  var rows = this._getRows();
  if (_.isArray(rows)) {
    this._result = rows.filter(predicate);
  } else if (_.isObject(rows)) {
    this._result = _.keys(rows).reduce(function(groups, group) {
      groups[group] = rows[group].filter(predicate);
      return groups;
    }, {});
  }
  return this;
}

// 分组
SQL.prototype.groupBy = function(key) {
  var rows = this._getRows();
  this._result = rows.reduce(function(groups, row){
    // 获得当前key对应的值，如果分组中不存在，则新建
    var group = row[key];
    if(!groups[group]) {
      groups[group] = [];
    }
    groups[group].push(row);
    return groups;
  }, {});
}
```

```
        return this;
    }

    // 获得结果
    SQL.prototype.getResult = function() {
        return this._result;
    }
}
```

用例：

```
var users = [
  {id: 0, name: 'wxj', age: 18, sex: 'male'},
  {id: 1, name: 'john', age: 28, sex: 'male'},
  {id: 2, name: 'bob', age: 33, sex: 'male'},
  {id: 3, name: 'tom', age: 22, sex: 'male'},
  {id: 4, name: 'alice', age: 18, sex: 'female'},
  {id: 5, name: 'rihana', age: 35, sex: 'female'},
  {id: 6, name: 'sara', age: 20, sex: 'female'}
];

var sql = new SQL(users);

var predicate = function(row) {
  return row.age<30;
}

var result = sql.groupBy('sex').where(predicate).select(['username', 'id']).getResult();
// => result:
//{
//  male: [
//    {id: 0, name: 'wxj', age: 18, sex: 'male'},
//    {id: 1, name: 'john', age: 28, sex: 'male'},
//    {id: 3, name: 'tom', age: 22, sex: 'male'}
//  ],
//  female: [
//    {id: 4, name: 'alice', age: 18, sex: 'female'},
//    {id: 6, name: 'sara', age: 20, sex: 'female'}
//  ]
//}
```

## 函数式（FP）

可以看到，在面向对象的抽象中，我们为了做一个查询，却造了太多不需要的东西，比如“类”，比如实例化对象的过程。同时，因为我们将 `table` 保存为对象的成员（可以视作需要维护的内部状态），且成员方法都存在着改变 `table` 的可能，所以就导致了我们的成员方法不是 [纯函数](#)（输入一定，则输出一定的函数），也就导致这些成员方法难于测试。



因而，我们将该用函数的眼光来抽象 `where` 、 `groupBy` 、 `select` 等过程，使得各自无关，并且保持纯度：

```
// 限定
function where(rows, predicate) {
  if (_.isArray(rows)) {
    return rows.filter(predicate);
  } else if (_.isObject(rows)) {
    return _.keys(rows).reduce(function (groups, group) {
      groups[group] = rows[group].filter(predicate);
      return groups;
    }, {});
  } else {
    return rows;
  }
}

// 分组
function groupBy(rows, key) {
  return rows.reduce(function (groups, row) {
    // 获得当前key对应的值，如果分组中不存在，则新建
    var group = row[key];
    if (!groups[group]) {
      groups[group] = [];
    }
    groups[group].push(row);
    return groups;
  }, {});
}

function doSelect(rows, keys) {
  return rows.map(function (row) {
    return _.keys(row).reduce(function (elem, key) {
      if (_.indexOf(keys, key) > -1) {
        elem[key] = row[key];
      }
      return elem;
    }, {});
  });
}
```

```
// 投影
function select(rows, keys) {
  if (_.isArray(rows)) {
    return doSelect(rows, keys);
  } else if (_.isObject(rows)) {
    return _.keys(rows).map(function (key) {
      return doSelect(rows[key], keys);
    });
  } else {
    return rows;
  }
}
```

我们还可以整合这三个方法为一个查询方法：

```
function query(table, by, predicate, keys) {
  return select(where(groupBy(table, by), predicate), keys);
}
```

现在，完成类似查询时，我们不再需要实例化对象，并且 `query` 也保持了纯度：

```
var users = [
  {id: 0, name: 'wxj', age: 18, sex: 'male'},
  {id: 1, name: 'john', age: 28, sex: 'male'},
  {id: 2, name: 'bob', age: 33, sex: 'male'},
  {id: 3, name: 'tom', age: 22, sex: 'male'},
  {id: 4, name: 'alice', age: 18, sex: 'female'},
  {id: 5, name: 'rihana', age: 35, sex: 'female'},
  {id: 6, name: 'sara', age: 20, sex: 'female'}
];

var predicate = function(elem) {
  return elem.age < 30;
};

var result = query(users, 'sex', predicate, ['name', 'id']);
// => result:
//{
//  male: [
//    {id: 0, name: 'wxj', age: 18, sex: 'male'},
//    {id: 1, name: 'john', age: 28, sex: 'male'},
//    {id: 3, name: 'tom', age: 22, sex: 'male'}
//  ],
//  female: [
//    {id: 4, name: 'alice', age: 18, sex: 'female'},
//    {id: 6, name: 'sara', age: 20, sex: 'female'}
//  ]
//}
```

## underscore 中的实现

`_.where`，`_.select` 函数在前面的章节已有过相应介绍，故不再重复赘述。接下来我们将认识到以下API：

- `_.groupBy`
- `_.indexBy`
- `_.countBy`
- `_.partition`

这几个函数都由内部函数 `group` 所创建，所以我们首先看到该函数。

## group

内置函数 `group(behavior, partition)` 接受 2 个参数：

- `behavior`：获得组别后的行为，即当确定一个分组后，在该分组上施加的行为
- `partition`：是否是进行划分，即是否是将一个集合一分为二

```
var group = function (behavior, partition) {  
  // 返回一个分组函数  
  return function (obj, iteratee, context) {  
    // 分组结果初始化  
    // 如果是进行划分(二分)的话，则结果分为两个组  
    var result = partition ? [[], []] : {};  
    iteratee = cb(iteratee, context);  
    _.each(obj, function (value, index) {  
      // 根据`iteratee`计算得到分组的组别key  
      var key = iteratee(value, index, obj);  
      // 获得组别后，执行定义的行为  
      behavior(result, value, key);  
    });  
    return result;  
  };  
};
```

`group` 将返回一个分组函数，其接受 3 个参数：

- `obj`：待分组集合对象
- `iteratee`：集合迭代器，同样会被内置函数 `cb` 优化
- `context`：执行上下文

下面我们看一看各个由 `group` 所创建的分组函数。

## \_.groupBy

`_.groupBy(obj, iteratee)`：对 `obj` 按照 `iteratee` 进行分组

当 `iteratee` 确定了一个分组后，`_.groupBy` 的行为是：

- 如果分组结果中存在该分组，将元素追加进该分组
- 否则新建一个分组，并将元素放入

源码：

```
_.groupBy = group(function (result, value, key) {  
    if (_.has(result, key)) result[key].push(value); else result  
    [key] = [value];  
});
```

用例：

```
_.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`：对集合按照指定的关键字进行索引

`_.indexBy(obj, iteratee)`：对 `obj` 按照 `iteratee` 进行索引。

当 `iteratee` 确定了一个分组后，`_.indexBy` 的行为：

- 设置该分组（索引）的对象为当前元素

源码：

```
_.indexBy = group(function (result, value, key) {  
    result[key] = value;  
});
```

用例：

```
var students = [
  {name: 'wxj', age: 18},
  {name: 'john', age: 18},
  {name: 'alice': age: 23},
  {name: 'bob': age: 30}
];

_.indexBy(students, 'age');
// => {
//   '18': {name: 'john', age: 18},
//   '23': {name: 'alice': age: 23},
//   '30': {name: 'bob': age: 30}
//}
```

## \_.countBy

`_.countBy(obj, iteratee)` : 对 `obj` 按照 `iteratee` 进行计数。

当 `iteratee` 确定了一个分组后，`_.countBy` 的行为：

- 如果该分组已存在，则计数加一
- 否则开始计数

源码：

```
_.countBy = group(function (result, value, key) {
  if (_.has(result, key)) result[key]++; else result[key] = 1;
});
```

用例：

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

## `_.partition`

`_.partition(obj, iteratee)` : 将 `obj` 按照 `iteratee` 进行分组。

当 `iteratee` 确定了一个分组后, `_.partition` 的行为:

- 将元素放入对应分组

源码:

```
_.partition = group(function (result, value, pass) {  
  // 分组后的行为,  
  result[pass ? 0 : 1].push(value);  
}, true);
```

用例:

```
_.partition([0, 1, 2, 3, 4, 5], function(num){  
  return num % 2 !== 0;  
});  
// => [[1, 3, 5], [0, 2, 4]]
```

## 拾遗

最后，我们介绍一些集合部分遗漏的 API。

### `_.max`

`_.max(coll, iteratee)`：根据 `iteratee` 声明的大小关系，获得 `coll` 最大元素。

源码：



```
_.max = function (obj, iteratee, context) {  
  // 默认返回-Infinity  
  var result = -Infinity, lastComputed = -Infinity,  
      value, computed;  
  if (iteratee == null || (typeof iteratee == 'number' && type  
of obj[0] != 'object') && obj != null) {  
    // 如果没有传递iteratee, 则按值进行比较  
    obj = isArrayLike(obj) ? obj : _.values(obj);  
    for (var i = 0, length = obj.length; i < length; i++) {  
      value = obj[i];  
      if (value != null && value > result) {  
        result = value;  
      }  
    }  
  } else {  
    // 否则, 以iteratee为最大值依据, 每次传入当前迭代值给iteratee,  
    算出最大值  
    iteratee = cb(iteratee, context);  
    _.each(obj, function (v, index, list) {  
      computed = iteratee(v, index, list);  
      if (computed > lastComputed || computed === -Infinity  
&& result === -Infinity) {  
        result = v;  
        lastComputed = computed;  
      }  
    });  
  }  
  return result;  
};
```

用例：

```
var students = [  
  {name: 'wxj', age: 18},  
  {name: 'john', age: 14},  
  {name: 'bob', age: 23}  
];  
  
_.max(students, 'age');  
// {name: 'bob', age:23}
```

## \_.min

`_.min(coll, iteratee)`：根据 `iteratee` 声明的大小关系，获得 `coll` 最小元素。

源码：

```
_.min = function (obj, iteratee, context) {
  var result = Infinity, lastComputed = Infinity,
      value, computed;
  if (iteratee == null || (typeof iteratee == 'number' && type
of obj[0] != 'object') && obj != null) {
    obj = isArrayLike(obj) ? obj : _.values(obj);
    for (var i = 0, length = obj.length; i < length; i++) {
      value = obj[i];
      if (value != null && value < result) {
        result = value;
      }
    }
  } else {
    iteratee = cb(iteratee, context);
    _.each(obj, function (v, index, list) {
      computed = iteratee(v, index, list);
      if (computed < lastComputed || computed === Infinity
&& result === Infinity) {
        result = v;
        lastComputed = computed;
      }
    });
  }
  return result;
};
```

用例：

```
var students = [
  {name: 'wxj', age: 18},
  {name: 'john', age: 14},
  {name: 'bob', age: 23}
];

_.min(students, 'age');
// {name: 'john', age:14}
```

## `_.invoke`

`_.invoke(coll, method)` : 迭代 `coll`，对每个元素调用其成员方法 `method`

源码：

```
_.invoke = restArgs(function (obj, method, args) {  
  // 通过闭包避免每次重复调用_.isFunction(method)  
  var isFunc = _.isFunction(method);  
  return _.map(obj, function (value) {  
    var func = isFunc ? method : value[method];  
    // 如果对象上不存在方法，则返回null  
    return func == null ? func : func.apply(value, args);  
  });  
});
```

用例：

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

## underscore 数组篇

这一章节我们将介绍 underscore 中提供的针对数组的操作，部分 API 已经在[集合篇](#)中有过阐述，不再赘述。

## 定位

### `_.initial`

`_.initial(array, n)` : 获得 `array` 的除了最后 `n` 个元素以外的元素。

源码：

```
_.initial = function (array, n, guard) {  
    return slice.call(array, 0, Math.max(0, array.length - (n ==  
    null || guard ? 1 : n)));  
};
```

`initial` 的实现很简单，就是基于 `Array.prototype.slice`，并且对 `n` 进行了校正，如果没有传递 `n`，则返回除了最末元素以外的所有元素。

用例：

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

### `_.rest = _.tail = _.drop`

`_.rest(array, n)` : 返回 `array` 中除了前 `n` 个元素外的所有元素。

源码：

```
_.rest = _.tail = _.drop = function (array, n, guard) {  
    return slice.call(array, n == null || guard ? 1 : n);  
};
```

用例：

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

## \_.first = \_.head = \_.take

\_.first(array, n)：获得 array 的前 n 个元素。

假设 array 长度为 length，则 \_.first(array, n) 就相当于  
\_.initial(array, length-n)。

源码：

```
_.first = _.head = _.take = function (array, n, guard) {  
  if (array == null || array.length < 1) return void 0;  
  if (n == null || guard) return array[0];  
  return _.initial(array, array.length - n);  
};
```

若不传递 n，则返回数组首个元素

用例：

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

## 展开一个数组

有时候，我们会遇到嵌套的数组：

```
[1, [2], [3, [[[4]]]]]
```

我们希望将他展开为这样，称这种展开方式为 浅展开：

```
[1, 2, 3, [[4]]]
```

也会希望展开到扁平，称这种展开方式为 深度展开：

```
[1, 2, 3, 4]
```

### **\_.flatten**

`_.flatten(array, shallow)`：展开 `array`，通过 `shallow` 指明是 深度展开 还是 浅展开。

源码：

```
_.flatten = function (array, shallow) {  
  return flatten(array, shallow, false);  
};
```

用例：



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

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

## flatten

我们看到，`_.flatten` 的逻辑由内部函数 `flatten` 所完成，该内部函数被多个 API 所使用：

```
var flatten = function (input, shallow, strict, output) {
  output = output || [];
  var idx = output.length; // 输出数组的下标
  for (var i = 0, length = getLength(input); i < length; i++)
  {
    // 获得元素值
    var value = input[i];
    if (isArrayLike(value) && (!_isArray(value) || !_isArguments(value))) {
      if (shallow) {
        // 如果不是深度展开
        // 只是从value(数组)中不断抽出元素赋值output中
        // 例如, value=[1,[3],[4,5]]
        // output = [...1,3,[4,5]...]
        var j = 0, len = value.length;
        while (j < len) output[idx++] = value[j++];
      } else {
        // 否则需要递归展开
        flatten(value, shallow, strict, output);
        // 刷新下标
        idx = output.length;
      }
    } else if (!strict) {
      // 如果不是严格模式, 则value可以不是数组
      output[idx++] = value;
    }
  }
  return output;
};
```

`flatten` 接受四个参数：

- `input` : 待展开数组
- `shallow` : 是否是浅展开, 反之为深度展开
- `strict` : 是否为严格模式
- `output` : 可以指定输出数组, 如果指定了输出数组, 则将展开后的数组添加至输出数组尾部

对传入数组 `input` 进行遍历, 对于遍历到的元素 `value` :

- 如果 `value` 为数组，则需要展开，浅展开很简单，只是展开该元素最外一层数组，深度展开则需要递归调用 `flatten`。
- 如果 `value` 不为数组，则只有在非严格模式下该 `value` 才会被赋值到新的数组中。

```
// 指定了严格模式
flatten([1, [2], [3, [[[4]]]], {name: 'wxj'}], true, true)
// => [2, 3, [[4]]]

// 未指定严格模式
flatten([1, [2], [3, [[[4]]]], {name: 'wxj'}], true, false);
// => [1, 2, 3, [[4]], {name: 'wxj'}]

// 指定了输出数组
var output = [5, 6, 7];
flatten([1, [2], [3, [[[4]]]], false, false, output);
// => output: [5, 6, 7, 1, 2, 3, 4]
```

## 数组运算

### `_.uniq = _.unique`

`_.uniq(array, isSorted, iteratee)`：根据 `iteratee` 设置的重复标准，对 `array` 进行去重。通过 `isSorted`，提高对有序数组的去重效率。

源码：

```
_.uniq = _.unique = function (array, isSorted, iteratee, context) {
  // 如果第二个参数不是 bool，则应当理解为是比较函数，且默认是没有排序的
  // 数组
  if (!_isBoolean(isSorted)) {
    context = iteratee;
    iteratee = isSorted;
    isSorted = false;
  }
  if (iteratee != null) iteratee = cb(iteratee, context);
  var result = [];
  var seen = []; // 标识数组
  for (var i = 0, length = getLength(array); i < length; i++) {
    var value = array[i],
        computed = iteratee ? iteratee(value, i, array) : value;

    // 如果排好序了，直接通过比较操作!==
    if (isSorted) {
      // 如果已经排序，seen 只需要反映最近一次见到的元素
      // !i: 第一个元素放入结果数组
      // seen !== computed 没有见过的元素放入结果数组
      if (!i || seen !== computed) result.push(value);
      // 刷新最近一次所见
      seen = computed;
    } else if (iteratee) {
      // 如果尚未排序，且存在比较函数，亦即不能直接通过 === 判断
      // 那么我们无法直接通过_.contains(result, value) 判断 value
    }
  }
  return result;
};
```

```
lue 是否已经存在
    // 例如_.unique([{age:13, name:"tom"},{age:15, name:"
jack"},{age:13, name:"bob"}], 'age']
    // 这种情况下就需要借助于 seen 这个辅助数组存储计算后的数组元素

    if (!_.contains(seen, computed)) {
        seen.push(computed);
        result.push(value);
    }
} else if (!_.contains(result, value)) {
    // 否则直接通过 contains 进行判断
    result.push(value);
}
}
return result;
};
```

underscore 提供了一个的数组去重功能十分健壮，他完成了如下优化：

- 通过指定 `iteratee`，让我们灵活定义“重复标准”，而不简单只是值比较（`===`）。
- 通过指定 `isSorted`，提高对排序后的数组去重性能。

用例：

```
// 一般去重
_.uniq([1, 1, 1, 1, 2, 2, 2, 3, 3]);
// => [1,2,3]

// 指定 `iteratee`
_.unique([ { age: 13, name: 'tom' }, { age: 15, name: 'jack' }, {
  age: 13, name: 'bob' } ], 'age');
// => [{age:13, name:"tom"}, {age:15, name: "jack"}]

// 性能对比
var arr = [];
for (i = 0; i < 100000; i++) {
  arr.push(i);
  arr.push(i);
}

console.time('未声明已排序');
_.uniq(arr);
console.timeEnd('未声明已排序');
// => "未声明已排序: 72054.154ms"

console.time('声明已排序');
_.uniq(arr, true);
console.timeEnd('声明已排序');
// => "声明已排序: 5.667ms"
```

## \_.union

`_.union(...arrays)` : 求数组并集。

源码：

```
// master
_.union = restArgs(function(arrays) {
  return _.uniq(flatten(arrays, true, true));
});

// 1.8.3
_.union = function() {
  return _.uniq(flatten(arguments, true, true));
};
```

可以看到，在 `master` 分支上，`_.union` 函数支持 `rest` 参数，由于 `rest` 参数最终会被处理成数组，所以，假定我们在调用 `_.union` 的时候传入两个数组 `[1, 2, 3, [4, 5]]` 和 `[6, [7]]`，那么 `arrays` 实际上是：`[[1, 2, 3, [4, 5], [6, [7]]]`。所以，我们先要浅展开 `arrays`，然后去重，得到数组的并集。

用例：

```
_.union([1, 2, 3, [4, 5]], [6, [7]]);
// => [1, 2, 3, [4, 5], 6, [7]]

// 由于设置了 `flatten` 的 `strict` 为 `true`，所以，只能传递数组求并集
_.union({name: 'wxj'}, [6, [7]]);
// => [6, [7]]
```

## \_.intersection

`_.intersection(...arrays)`：求取数组交集

源码：

```
_.intersection = function (array) {  
    var result = [];  
    var argsLength = arguments.length;  
    for (var i = 0, length = getLength(array); i < length; i++)  
    {  
        var item = array[i];  
        if (_.contains(result, item)) continue;  
        var j;  
        //  
        for (j = 1; j < argsLength; j++) {  
            if (!_.contains(arguments[j], item)) break;  
        }  
        if (j === argsLength) result.push(item);  
    }  
    return result;  
};
```

数组交集的求取思路为：遍历第一个数组的每个元素，在之后的所有数组中找寻是否有该元素，有则放入结果数组。

用例：

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

## \_.difference

`_.difference(...array)`：求取数组差集

源码：



```
// master
_.difference = restArgs(function(array, rest) {
  rest = flatten(rest, true, true);
  return _.filter(array, function(value){
    return !_.contains(rest, value);
  });
});

// 1.8.3
_.difference = function(array) {
  var rest = flatten(arguments, true, true, 1);
  return _.filter(array, function(value){
    return !_.contains(rest, value);
  });
};
```

数组差集的求取思路为：令剩余的数组为 `rest`，遍历第一个数组 `array`，保留 `array` 中存在的，而 `rest` 中不存在的值。

用例：

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

## `_.unzip`

`_.unzip(array)`：解压 `array`

假设我们有三个数组：

- `names`：['moe', 'larry', 'curly']
- `ages`：[18, 23, 30]
- `sexes`：['male', 'female', 'male']

他们构成了一个新的数组 `infos`：

```
var infos = [names, ages, sex]
```

数组解压就是将多个数组的对应位置的元素抽离出来，组成新的数组：

```
[[ 'moe', 18, 'male'], [ 'larry', 23, 'female'], [ 'curly', 30, 'male' ]]
```

源码：

```
_.unzip = function (array) {  
  // 原数组的长度反映了最后分的组数目  
  var length = array && _.max(array, getLength).length || 0;  
  // 结果数组与原数组等长  
  var result = Array(length);  
  // 分别抽离元素放到新数组  
  for (var index = 0; index < length; index++) {  
    result[index] = _.pluck(array, index);  
  }  
  return result;  
};
```

用例：

```
var names = [ 'moe', 'larry', 'curly' ];  
var ages = [ 18, 23, 30 ];  
var sexes = [ 'male', 'female', 'male' ];  
  
var students = _.unzip([names, ages, sexes]);  
// => students: [ [ 'moe', 18, 'male' ], [ 'larry', 23, 'female' ], [ 'curly', 30, 'male' ] ]
```

## \_.zip

\_.zip(array) : 压缩 array

给定若干个数组，对他们进行的压缩过程为：

1. 各个数组对应位置抽离元素，组成新的数组
2. 将这些新数组再合并为一个数组

比如我们给定若干个数组：

```
var student1 = ['moe', 18, 'male'];
var student2 = ['larry', 23, 'female'];
var student3 = ['curly', 30, 'male'];
```

那么 `_.zip` 操作的结果为：

```
var infos = [['moe', 'larry', 'curly'], [18, 23, 30], ['male', 'female', 'male']]
```

所以，`_.zip` 操作的过程也十分类似 `_.unzip`，只是传入参数由单个数组变为多个数组，那么我们只需要将这多个数组先合并一下就可以了，`underscore` 也正是这么做的。

源码：

```
// master
_.zip = restArgs(_.unzip);

// 1.8.3
_.zip = function() {
  return _.unzip(arguments);
};
```

用例：

```
var student1 = ['moe', 18, 'male'];
var student2 = ['larry', 23, 'female'];
var student3 = ['curly', 30, 'male'];

var infos = _.unzip(student1, student2, student3);
// => [['moe', 'larry', 'curly'], [18, 23, 30], ['male', 'female', 'male']]
```



## 拾遗

### `_.compact`

`_.compact(array)` : 去除 `array` 中所有“否定”项目。

在 JavaScript 中，这些值被认为具有“否定”意向：

- `false`
- `null`
- `0`
- `""`
- `undefined`
- `NaN`

怎么验证呢？

```
Boolean(false); // => false
Boolean(null); // => false
Boolean(0); // => false
Boolean(""); // => false
Boolean(undefined); // => false
Boolean(NaN); // => false
```

源码：

```
_.compact = function (array) {
  return _.filter(array, Boolean);
};
```

用例：

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

## \_.object

\_.object(list, values) : 将 list 转换为对象

源码：

```
_.object = function (list, values) {  
  var result = {};  
  for (var i = 0, length = getLength(list); i < length; i++) {  
    if (values) {  
      result[list[i]] = values[i];  
    } else {  
      result[list[i][0]] = list[i][1];  
    }  
  }  
  return result;  
};
```

看到源码，根据传入参数的形式不同，有两种策略将数组转化为对象：

- 若没有传入值序列 values，则视 list 中每个元素为一个 k-v 对：

```
var list = [['moe', 30], ['larry', 40], ['curly', 50]];
```

- 如果传入了 values，则视 list 为 key 集合

```
var list = ['moe', 'larry', 'curly'];  
var values = [30, 40, 50];
```

用例：

```
_.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}
```

## \_.range

`_.range(start, stop, step)` : 设置步长 `step` , 产生一个 `[start, n)` 的序列。

如果有过 python 开发经验, 那对区间产生函数一定不会陌生, 该函数对于我们快速产生一个落在区间范围内的数组大有裨益, python 中, 区间函数为 `range` :

```
## 产生 [n,m) 内的数组
range(1, 11);
## => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

## 指定步长
range(1, 11, 2)
## => [1, 3, 5, 7, 9]

## 从 [0, n)
range(5)
## => [1, 2, 3, 4]
```

`underscore` 也提供了一个用法类似的区间函数 `_.range` :

```
_.range = function (start, stop, step) {  
  // 校正终点  
  if (stop == null) {  
    stop = start || 0;  
    start = 0;  
  }  
  // 校正步长  
  if (!step) {  
    step = stop < start ? -1 : 1;  
  }  
  
  // 计算最终数组的长度  
  var length = Math.max(Math.ceil((stop - start) / step), 0);  
  var range = Array(length);  
  
  for (var idx = 0; idx < length; idx++, start += step) {  
    range[idx] = start;  
  }  
  
  return range;  
};
```

其接受 3 个参数：

- `start`：起点，如果只提供了该参数，那么起点被认为是 `0`，终点为 `start`
- `stop`：终点
- `step`：步长，如果未提供该参数，则需要根据 `start < stop` 来判定行进方向，正向 `1`，负向 `-1`

用例：



```
_.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);  
// => []
```

## \_.chunk

`_.chunk(array, count)`：将 `array` 划分为若干份，每份 `count` 个元素，再合并到一个数组

源码：

```
_.chunk = function (array, count) {  
  if (count == null || count < 1) return [];  
  
  var result = [];  
  var i = 0, length = array.length;  
  while (i < length) {  
    result.push(slice.call(array, i, i += count));  
  }  
  return result;  
};
```

用例：

```
_.chunk([1, 2, 3, 4, 5, 6, 7], 2);  
// => [[1, 2], [3, 4], [5, 6], [7]]
```



## underscore 函数篇

在 JavaScript 中，函数是第一型的对象，函数在 JavaScript 中的地位因此可见一斑。这一章节也是我认为最为重要的一章，在本章中，能够见到许多实用的针对函数的操作，以及函数式编程中的重要概念。

## 上下文绑定

### 恼人的 `this`

对于 JavaScript 初学者来说，`this` 关键字一直是一个恼人的东西，关于 `this` 的解释和描述，最好的文章是 [You-Dont-Know-JS](#) 下的 [this & Object Prototypes](#) 章节。文中，作者认为，人们通常会因为 `this` 的字面意思而产生如下两种误解

- itself

把 `this` 理解为 `itself`，也就是认为 `this` 指向了其所在的函数：

```
function foo(num) {
    console.log( "foo: " + num );

    // keep track of how many times `foo` is called
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}

// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 0 -- WTF?
```

当我们认为 `this` 指代的是所在函数 `foo` 时，就想要最后打印 `foo.count` 的结果是 10，然而事与愿违。不过这种误解也是情有可原的，因为在 js 中，函数也是对象，而在对象中，似乎 `this` 确是指向了自身 `itself`：

```
var obj = {  
  count : 10;  
  foo: function(){  
    console.log(this.count);  
  }  
}  
  
obj.foo(); // => 10
```

- its scope

还有一种误解显得“高明”许多，他们认为 `this` 由其函数所在的词法作用域决定，比如下例：

```
function foo() {  
  var a = 2;  
  this.bar();  
}  
  
function bar() {  
  console.log('a is:' + this.a );  
}  
  
foo(); //a is:undefined
```

这个例子非常有意思，他不只犯了一个错误：

- 首先，代码书写者认为 `this` 由其函数所在的词法作用域决定，函数 `foo` 所在的词法作用域是在全局作用域，所以 `this` 就指向了全局作用域，那么，由于函数 `bar` 所在的词法作用域也是全局作用域，`this.bar()` 理应能够工作。事实也确实如此，但是工作的原因却并非如此，之后会做解释。
- 代码书写者认为 `this` 也能像闭包那样被使用：当内部作用域中 `this.a` 寻找失败时，会去外部作用域寻找 `this.a`。千万要记住，`this` 并不具备跨作用域的能力。

## this 究竟是谁？

`this` 关键字并不神秘，我们需要知道，`this` 指代的对象并不能在函数定义就确定，他由函数执行时所在的上下文确定。换言之，一个函数如果不跑起来，他内部的 `this` 就永远无法确定。看到如下代码：

```
function foo() {  
  console.log(this.a);  
}  
  
var a = 2;  
foo(); // => 2
```

当函数 `foo` 调用时，亦即 `foo()` 语句执行时，其所在的上下文是全局对象，所以，`this` 绑定到了全局对象（在浏览器环境的话就是 `window` 对象，`foo()` 也相当于是 `window.foo()`）。再看下面的一个代码：

```
function foo() {  
  console.log(this.a);  
}  
  
var obj = {  
  a: 2,  
  foo: foo  
}  
  
obj.foo(); // => 2
```

当成员方法 `obj.foo` 被调用时，以及 `obj.foo()` 语句执行时，函数 `foo` 所在的执行上下文变成了 `obj`，即 `this` 指向了 `obj`。

综上，我们知道，想要知道 `this` 最终“情归何处”，就一定是在运行时（**runtime**）环境，而不是函数定义时。但是，js 同样为我们提供了手段，来手动绑定函数中的 `this` 关键字，我将他分为两类：1. 调用型绑定：进行一次函数调用，并绑定当中的 `this` 2. 定义型绑定：不进行函数调用，新创建一个函数，指定其中的 `this` 归属

注意，这两型的绑定都不会改变“`this` 只有在运行时才能够绑定”这一事实。

## 调用型绑定

在 `Function.prototype` 上，提供了 `apply` 和 `call` 两个方法来绑定 `this` 指代的上下文对象，二者的第一个参数都是待绑定对象，而第二个参数都是调用参数，只是表现形式不同，`apply` 传递的参数是数组型的，`call` 则是逐个传递参数：

```
function add(a, b) {
  var result = a+b;
  console.log(this.name+" wanna get result:"+result);
}

var caller1 = {
  name: 'wxj'
};

var caller2 = {
  name: 'zxy'
};

// call需要逐个传递参数
add.call(caller1, 3, 4);
// => "wxj wanna get result:7"

// apply传递参数数组
add.apply(caller2, [3,4]);
// => "zxy wanna get result:7"
```

## 定义型绑定

而在 ES5 中，`Function.prototype` 还提供了 `bind` 来绑定 `this` 指代的上下文，与调用型绑定不同的是，定义型绑定不会立即调用函数，而是返回一个被固定了执行上下文的新函数：

```
function showName() {  
  console.log('my name is:'+this.name);  
}  
  
var student = {  
  name: 'wxj',  
  age: 13  
};  
  
var showWxjName = showName.bind(student);  
showWxjName();  
// => "my name is:wxj"
```

定义型绑定也不能改变“`this` 只有在运行时才能够绑定”这一事实，他只是创建了一个新函数，然后让新函数在执行过程中通过 `call` 或者 `apply` 来调用绑定了执行上下文的老函数。看到 MDN 给出的 polyfill：



```

if (!Function.prototype.bind) {
  Function.prototype.bind = function (oThis) {
    if (typeof this !== "function") {
      // closest thing possible to the ECMAScript 5
      // internal IsCallable function
      throw new TypeError("Function.prototype.bind - what is try
ing to be bound is not callable");
    }

    var aArgs = Array.prototype.slice.call(arguments, 1),
        // 待绑定对象
        fToBind = this,
        fNOP = function () {},
        fBound = function () {
          return fToBind.apply(this instanceof fNOP
                               ? this
                               : oThis || this,
                               aArgs.concat(Array.prototype.slic
e.call(arguments)));
        };

    fNOP.prototype = this.prototype;
    fBound.prototype = new fNOP();

    return fBound;
  };
}

```

这个 polyfill 非常值得玩味，当中出现的各个 `this` 令人眼花缭乱，下面着重解释一下该 polyfill：

- 设置待绑定函数

```

// ...
fToBind = this;
// ...

```

当我们讨论 `this` 的时候，一定要到执行环境下：

```
// ...
function foo() {
  console.log("a is:" + a);
}

var obj = {
  a: 2
};

// bind被调用，处于执行环境
foo.bind(obj);
// ...
```

此处的 `this` 绑定到了 `foo`，即 `fToBind === foo`。

- 创建绑定函数

```
// ...
fNOP = function () {},
fBound = function () {
  // `apply`时进行上下文校正
  return fToBind.apply(this instanceof fNOP
    ? this
    : oThis || this,
    aArgs.concat(Array.prototype.slice.call(
arguments))));
};
fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();
// ...
```

注意，以上语句的意图是要照顾到绑定后的函数作为构造函数时（是否被 `new` 操作）的情况：

```
function func() {  
    this.name = "anonymous";  
}  
  
var Person = func.bind(obj);  
var person = new Person();
```

我们知道，`new` 操作符会将构造函数中的 `this` 的绑定到实例化对象上，如果我们不进行上下文校验，那么 `this` 会被绑定到 `obj` 上，这显然不是我们期望看到的。

所以，当 `fBound` 执行时，如果是通过 `new` 操作符调用 `fBound`，即 `fBound` 被当做构造函数使用，执行上下文应当是实例化对象本身，由于实例化对象已经绑到了 `this`，所以不需要再改变 `this` 的指向。

## executeBound

`underscore` 中也提供了绑定函数上下文的方法 `_.bind` 和 `_.bindAll`，当所处环境不支持 `Function.prototype.bind` 时，二者的绑定上下文及执行过程都由内部函数 `executeBound` 负责：

```
// 执行绑定后的函数  
var executeBound = function(sourceFunc, boundFunc, context, callingContext, args) {  
    if (!(callingContext instanceof boundFunc))  
        return sourceFunc.apply(context, args);  
    var self = baseCreate(sourceFunc.prototype);  
    var result = sourceFunc.apply(self, args);  
    if (_.isObject(result))  
        return result;  
    return self;  
};
```

`executeBound` 接受 5 个参数：

- `sourceFunc`：待绑定函数
- `boundFunc`：绑定后函数

- `context` : 待绑定上下文
- `callingContext` : 执行上下文，通常就是 `this`
- `args` : 函数执行所需参数

并且，类似于 MDN 中提供的 `polyfill`，`executeBound` 也考虑到了当绑定后的函数 `boundFunc` 作为构造函数被 `new` 运算的情形，进行了执行上下文的修正。另外，为了支持链式调用，所以有如下语句：

```
var result = sourceFunc.apply(self, args);
if (._isObject(result))
    return result;
return self;
```

## `._bind`

`._bind(func, context)` : 将 `func` 的执行上下文绑定到 `context`

源码：

```
._bind = function (func, context) {
    if (nativeBind && func.bind === nativeBind) return nativeBind.apply(func, slice.call(arguments, 1));
    if (!._isFunction(func)) throw new TypeError('Bind must be called on a function');
    var args = slice.call(arguments, 2);
    var bound = function () {
        return executeBound(func, bound, context, this, args.concat(slice.call(arguments)));
    };
    return bound;
};
```

`._bind` 将为我们返回一个绑定了上下文的函数，该函数的执行过程会限定执行上下文。

用例：

```
// 测试一般函数绑定
function add(a,b) {
    var result = a+b;
    console.log(this.name + ' wanna get add result:' + result);
}

var obj = {
    name: 'wxj'
}

var bound = _.bind(add, obj, 3, 4);
bound();
// => "wxj wanna get add result:7"

// 测试绑定函数作为构造函数使用
function constr() {
    console.log('my name is:' + this.name);
}

var Person = _.bind(constr, obj);
// 一般函数使用
Person();
// => "my name is wxj"

// 构造函数使用
var person = new Person();
// => "my name is:undefined"
```

## \_.bindAll

`_.bindAll(obj)` : 绑定对象 `obj` 的所有指定成员方法中的执行上下文 到 `obj`

先看到下面这样一个例子：

```
// 我们模拟一个DOM元素
var button = {
  title: 'button#1',
  onClick: function() {
    console.log(this.title + ' has been clicked!');
  }
};

button.onClick();
// => "button#1 has been clicked!"

setTimeout(button.onClick, 0);
// => "undefined has been clicked!"
```

悲剧，当我们把成员方法作为 `setTimeout` 的回调传入后，`this` 在运行时的绑定事与愿违。记住，`button.onClick` 不是 `button.onClick()`，`setTimeout` 的回调函数执行时的上下文不再是 `button` 对象，为此，我们就需要绑定 `button` 所有成员方法的 `this` 到该对象上，`underscore` 通过 `_.bindAll` 来实现这个目标。

源码：

```
_.bindAll = function (obj) {
  var i, length = arguments.length,
      key;
  if (length <= 1) throw new Error('bindAll must be passed function names');
  for (i = 1; i < length; i++) {
    key = arguments[i];
    obj[key] = _.bind(obj[key], obj);
  }
  return obj;
};
```

用例：

```
var button = {
  title: 'button#1',
  onClick: function() {
    console.log(this.title + ' has been clicked!');
  },
  onHover: function() {
    console.log(this.title + ' hovering!');
  }
}

_.bindAll(button, 'onClick', 'onHover');

setTimeout(button.onClick, 0);
setTimeout(button.onHover, 0);
// => "button#1 has been clicked!"
// => "button#1 hovering!"
```

# 偏函数

## 引子

在 JavaScript 中，有这么一个 API -- `Number.parseInt(string[, radix])` : 根据指定的进制数将字符串转换成对应整数：

```
parseInt("16", 10); // => 16
parseInt("16", 16); // => 22
parseInt("100", 2); // => 4
```

那么我们先分别封装的对应的进制转换操作呢？

```
function parseInt10(string) {
  return parseInt(string, 10);
}

function parseInt2(string) {
  return parseInt(string, 2);
}

function parseInt16(string) {
  return parseInt(string, 16);
}
```

这种封装还不具有函数式的抽象，我们更希望是下面这种封装：

```
var parseInt10 = factory(parseInt, 10);
var parseInt2 = factory(parseInt, 2);
var parseInt16 = factory(parseInt, 16);
```

`factory` 是一个工厂函数，且是一个高阶函数，他将完成：

接受一个函数，传递给该函数部分参数，返回一个新的函数。让新的函数继续接受参数：



(1) 原函数:  $sourceFunction(x, y, z)$

(2) 新函数:  $newFunction(y, z) = factory(sourceFunction, x)$

即，新函数是原函数的一部分。

```
function factory(sourceFunc) {  
  // 获得参数数组  
  var args = Array.prototype.slice.call(arguments, 1);  
  // 返回新的函数  
  return function() {  
    // 新的函数继续接受参数  
    var newArgs = Array.prototype.slice.call(arguments, 0);  
    return sourceFunc.apply(this, newArgs.concat(args));  
  }  
}
```

测试一下：

```
var parseInt2 = factory(parseInt, 2);  
parseInt2('100'); // => 4
```

上面的工厂函数我们就称之为偏函数（**partial**），**partial** 反映了新函数是原函数的一部分。underscore 的 `_.partial` 就能返回一个偏函数。

## `_.partial`

`_.partial(func, ...args)`：偏应用一个函数 `func`。

源码：

```
_.partial = restArgs(function (func, boundArgs) {
  var placeholder = _.partial.placeholder;
  // 返回一个partial后的新函数
  var bound = function () {
    // position用来标识当前赋值的arguments最新位置
    var position = 0, length = boundArgs.length;
    // 初始化新函数执行的参数
    var args = Array(length);
    for (var i = 0; i < length; i++) {
      // 对于最终调用时的位置`i`的参数
      // 如果绑定参数的对应位置是占位符，代表略过，以新的参数赋值之
      // 并刷新最新的赋值位置`position`
      // 否则以绑定参数赋值之
      args[i] = boundArgs[i] === placeholder ? arguments[position++] : boundArgs[i];
    }
    // 如果arguments还没有被消费完，则剩余arguments全部灌入args
    while (position < arguments.length) args.push(arguments[position++]);
    // 执行绑定函数的时候，不改变上下文
    return executeBound(func, bound, this, this, args);
  };
  return bound;
});
```

相比较我们缩写的偏函数，undersocre 提供的偏函数还支持传递占位符，占位符用来标识不想被初始化的参数。

用例：

```
var subtract = function(a, b) {  
  return b - a;  
};  
  
sub5 = _.partial(subtract, 5);sub5(20);  
// => 15  
  
// 通过使用了一个placeholder(默认被理解的占位符为_), 我们这次先赋值了b,  
// 暂缓了对a的赋值  
subFrom20 = _.partial(subtract, _, 20);  
subFrom20(5);  
// => 15
```

# 缓存

## 引子

fibonacci 数列相信大家不会陌生：

$$\begin{aligned} F * n &= F * n - 1 + F_{n-2} \quad (n \geq 2) \\ F_0 &= 0 \\ F_1 &= 1 \end{aligned}$$

在 JavaScript 中，我们对其递归求解：

```
function fibonacci(n){  
    return n<2 ? n : fibonacci(n-1) + fibonacci(n-2);  
}  
  
fibonacci(5); // => 5
```

我们仔细看一下 `fibonacci(5)` 的递归过程：

```
fibonacci(5)  
fibonacci(5) = fibonacci(4)+fibonacci(3)  
fibonacci(4) = fibonacci(3)+fibonacci(2)  
fibonacci(3) = fibonacci(2)+fibonacci(1)  
// ....
```

可以看到，在该递归过程中，引起了很多的重复计算，所以我们考虑将计算结果进行缓存。为此，我们需要利用到高阶函数和闭包：

```
var fibonacciWithMemo = (function() {
  var memo = {};
  return function fibonacci(n) {
    // 如果尚未建立缓存
    if(!memo[n]) {
      memo[n] = n < 2 ? n : fibonacci(n-1) + fibonacci(n-2);
    }
    return memo[n];
  }
})();
```

测试：

```
console.time("没有缓存优化的fibonacci");
fibonacci(40);
console.timeEnd("没有缓存优化的fibonacci");
// => "没有缓存优化的fibonacci: 1505.980ms"

console.time("使用缓存优化的fibonacci");
fibonacciWithMemo(40);
console.timeEnd("使用缓存优化的fibonacci");
// => "使用缓存优化的fibonacci: 0.281ms"
```

上面我们的缓存工厂函数还只能创建一个带缓存记忆的 fibonacci 计算函数，underscore 则为我们封装了一个通用的缓存函数创建器 -- `_.memoize`。

## `_.memoize`

`_.memoize(func, hasher)`：为函数 `func` 提供记忆功能，`hasher` 定义如何获得缓存。

```
_.memoize = function (func, hasher) {  
  var memoize = function (key) {  
    // 执行记忆函数的时候，先获得缓存  
    var cache = memoize.cache;  
    // 获得缓存地址  
    var address = '' + (hasher ? hasher.apply(this, arguments  
    ) : key);  
    // 如果缓存没有命中，则需要调用函数执行  
    if (!_.has(cache, address)) cache[address] = func.apply(  
    this, arguments);  
    // 否则直接返回缓存  
    return cache[address];  
  };  
  // 初始化记忆函数的缓存  
  memoize.cache = {};  
  return memoize;  
};
```

`_.memoize` 将缓存绑定到了缓存函数的 `cache` 属性上，在创建一个缓存函数时，除了提供原函数 `func` 用来计算值外，还可以提供 `hasher` 函数来自定义如何获得缓存的位置。如果 `hasher` 不设置，则以缓存函数的参数 `key` 标识缓存位置。

用例：

- 不传递 `hasher`：

```
var fibonacci = _.memoize(function(n) {  
    return n < 2 ? n: fibonacci(n-1) + fibonacci(n-2);  
});  
  
fibonacci(5); // => 5  
console.log(fibonacci.cache);  
// {  
//   0: 0,  
//   1: 1,  
//   2: 2,  
//   3: 2,  
//   4: 3,  
//   5: 5  
// }
```

- 传递 hasher :

```
var hasher = function() {  
    var n = arguments[0];  
    return n;  
}  
  
var fibonacci = _.memoize(function(n) {  
    return n < 2 ? n: fibonacci(n-1) + fibonacci(n-2);  
}, hasher);  
  
fibonacci(5); // => 5  
console.log(fibonacci.cache);  
// {  
//   0: 0,  
//   1: 1,  
//   2: 2,  
//   3: 2,  
//   4: 3,  
//   5: 5  
// }
```





## 延时执行

在 `underscore` 中，提供了两个函数，用于延时执行函数。

### `_.delay`

`_.delay(wait, func)`：等待 `wait` 毫秒，执行函数 `func`。

源码：

```
// master
_.delay = restArgs(function (func, wait, args) {
  return setTimeout(function () {
    return func.apply(null, args);
  }, wait);
});

// 1.8.3
_.delay = function(func, wait) {
  var args = slice.call(arguments, 2);
  return setTimeout(function(){
    return func.apply(null, args);
  }, wait);
};
```

可以看到，`_.delay` 只是 `js` 内置函数 `setTimeout` 的一次封装，但要注意的，`_.delay` 会将待延迟执行的函数上下文“绑定”到 `null`，所以，传入回调的时候，一定要确保上下文已经正确绑定（也就是先替换掉函数中 `this`，避免之后 `this` 被错误绑定）。

这里的绑定之所以打引号，是因为调用 `apply` 或者 `call` 的时候，如果传递的第一个参数是 `null` 或者 `undefined`，那么 `this` 就会被绑定到全局对象上。

用例：

```
var log = _.bind(console.log, console);
_.delay(log, 1000, "show after 1 sec");
// => "show after 1 sec" //至少1s之后打印该字符串
```

## \_.defer

\_.defer(func) : 将 func 异步化。

源码：

```
_.defer = _.partial(_.delay, _, 1);
```

这就是偏函数的能力，现在创建新函数变得如此简单，基础函数是 `_.delay`，通过指定 `wait` 参数，我们创建了一个新函数，该函数会被推迟至少 1ms 才执行，亦即，我们将传入的任务异步化。

用例：

```
_.defer(function() { console.log('2');});
console.log('1');
// => "1"
// => "2"
```

## throttle 与 debounce

我们先来看一个前端开发中遇到的场景：

在页面中，我们有一个“查询”按钮，单击该按钮，会通过 **ajax** 异步查询一些数据

假设这个查询是耗时的，并且会给后端造成一定压力，那么如果短时间内我们频繁点击该按钮，请求就会不断发出，这样给后端造成的压力是难以想象的。因此，我们会想到控制我们的查询速率。

### 独占型提交

通过设置一个 **flag** 来标识当前状态为正在请求中，如果已在请求中，则不允许再次请求，请求完成，刷新该 **flag** 来允许新的请求提交：

```
var isQuerying = false;

var sendQuery = function(complete) {
    // 如果已经在查询了，那么需要等待
    if(isQuerying){
        console.log("waiting");
        return;
    }
    console.log("send query");
    // 标识当前正在查询中
    isQuerying = true;
    // 我们模拟一个耗时操作
    setTimeout(function(){
        complete && complete();
    }, 2000);
}

var complete = function() {
    // 在回调中， 我们刷新标记量
    isQuerying = false;
    console.log("completed");
}

$("#queryBtn").click(function(){sendQuery(complete);});

// 测试一下， 可以看到新的请求不再立即被送出
for(var i=0;i<100;i++) {
    $("#queryBtn").click();
}
```

## 节制型提交

在独占型提交中，如果一个请求已经在进行中，那么再多的点击都会被废弃。如果我们只是想限制请求速率，而不想废弃掉之后的点击，那我们得考虑新的提交方式。回归到限制提交的问题本质：

控制回调发生的速率，不需要回调发生那么快

但是，如果我们新建一个 `wrapper` 来包裹一下 `sendQuery` 业务：在 `click` 事件后，不会直接调用 `sendQuery`，而是调用 `wrapper`，在 `wrapper` 的执行过程中，我们有选择地考虑是否执行 `sendQuery`，借此控制住 `sendQuery` 的调用频率。

假设我们想至少等待 1s 才能发出一次新的查询请求，即请求的调用频次不能超过 1 次/秒，可以这样设计：

1. 开始：`click` 事件到来，`wrapper` 被调用
2. 获得当前时间，比较当前时间距上次 `sendQuery` 执行的时间是否已经足够 1s。
3. 如果已经足够，那么这次查询请求可以立即被执行，否则计算应该等待的时间，延后执行该请求。

看代码：

```
var previous = 0; // 记录上次执行的时间点
var waiting = 1000; // 需要等待的时间

var sendQuery = function() {
  // 执行的时候，刷新previous
  previous = (new Date()).getTime();
  console.log("sending query");
}

var func = function() {
  // 获得当前时间
  var now = (new Date()).getTime();
  // 获得需要等待的时间
  var remain = waiting - (now - previous);
  // 判断是否立即执行
  if(remain <= 0) {
    sendQuery();
  } else {
    setTimeout(sendQuery, remain);
  }
}
```

为了防止变量的全局污染，我们再用一个立即执行函数包裹下作用域：

```
var delayedQuery = (function() {
  var previous = 0; // 记录上次执行的时间点
  var waiting = 1000; // 需要等待的时间

  var sendQuery = function() {
    // 执行的时候，刷新previous
    previous = (new Date()).getTime();
    console.log("sending query");
  }

  return function() {
    // 获得当前时间
    var now = (new Date()).getTime();
    // 获得需要等待的时间
    var remain = waiting - (now - previous);
    console.log("need waiting " + remain + " ms");
    // 判断是否立即执行
    if (remain <= 0) {
      console.log("immediately");
      sendQuery();
    } else {
      console.log("delayed");
      setTimeout(sendQuery, remain);
    }
  }
})();

$("#queryBtn").click(delayedQuery);
```

但是，我们的业务代码 `sendQuery` 还是耦合了刷新 `previous` 的逻辑。其次，如果每个延迟执行的诉求都要去做这样一个包裹，样板代码就显得太多了。现在我们撰写一个通用函数，我们将（1）需要控制调用频度的函数和（2）对调用频度的限制平率告诉通用函数，它返回一个限制了执行频率的函数。

```
/**
 * throttle
 * @param {Function} func 待控制频率的函数
 * @param {Number} waiting 每次调用的最小等待周期
```

```
*/  
function throttle(func, waiting) {  
    var previous = 0;  
  
    // 创建一个func的wrapper，如要是解耦func与previous等变量  
    var later = function() {  
        // 刷新previous  
        previous = (new Date()).getTime();  
        // 执行调用  
        func();  
    }  
    // 返回一个被控制了调用频率的  
    return function() {  
        // 获得当前时间  
        var now = (new Date()).getTime();  
        // 获得需要等待的时间  
        var remain = waiting - (now - previous);  
        console.log("need waiting " + remain + " ms");  
        // 判断是否立即执行  
        if (remain <= 0) {  
            console.log("immediately");  
            later();  
        } else {  
            console.log("delayed");  
            setTimeout(later, remain);  
        }  
    }  
}  
  
// 现在，刷新previous不再需要耦合到sendQuery中  
var sendQuery = function() {  
    console.log("sending query");  
}  
  
delayedQuery = throttle(sendQuery, 1000);
```

[查看演示](#)

## Underscore 中的 throttle

可以看出来，这里我已经用了 `throttle` 来命名我们的函数了，`throttle`，也就是节流阀的意思，很形象是吧，通过这样一个阀门，我们限制函数的执行频次。

但是在上面的代码中，还有一点小问题，一些查询请求虽然被延迟执行了，但是在某个时间点，他们好像一起执行了，这是因为 `setTimeout(func, wait)` 并不能保证 `func` 的执行开始时间，只能保证 `func` 在不早于从现在起到 `wait` 毫秒后发生。所以被延后执行的那些查询在某个相近的时间点同时发生了。

出现这个错误的原因就是：

我们只保障了第一次回调和接下来所有回调的间隔执行，而没有保障到各个回调间相互的间隔执行。

下面可以看一下 Underscore 中 `throttle` 的实现，比刚才我们写的 `throttle` 函数健壮许多。

```
_.throttle = function (func, wait, options) {  
  // timeout标识最近一次被追踪的调用  
  // context和args缓存func执行时需要的上下文，result缓存func执行结果  
  var timeout, context, args, result;  
  // 最近一次func被调用的时间点  
  var previous = 0;  
  if (!options) options = {};  
  
  // 创建一个延后执行的函数包裹住func的执行过程  
  var later = function () {  
    // 执行时，刷新最近一次调用时间  
    previous = options.leading === false ? 0 : _.now();  
    // 清空为此次执行设置的定时器  
    timeout = null;  
    result = func.apply(context, args);  
    if (!timeout) context = args = null;  
  };  
  
  // 返回一个throttle化的函数  
  var throttled = function () {  
    // 我们尝试调用func时，会首先记录当前时间戳  
    var now = _.now();  
    // 是否是第一次调用
```



```
        if (!previous && options.leading === false) previous = now;

        // func还要等待多久才能被调用 = 预设的最小等待期 - (当前时间 - 上一次调用的时间)
        var remaining = wait - (now - previous);
        // 记录执行时需要的上下文和参数
        context = this;
        args = arguments;
        // 如果计算后能被立即执行
        if (remaining <= 0 || remaining > wait) {
            // 清除之前的设置的延时执行，就不存在某些回调一同发生的情况了
            if (timeout) {
                clearTimeout(timeout);
                timeout = null;
            }
            // 刷新最近一次func调用的时间点
            previous = now;
            // 执行func调用
            result = func.apply(context, args);
            // 再次检查timeout，因为func执行期间可能有新的timeout被设置，如果timeout被清空了，代表不再有等待执行的func，也清空context和args
            if (!timeout) context = args = null;
        } else if (!timeout && options.trailing !== false) {
            // 如果设置了trailing edge，那么暂缓此次调用尝试的执行
            timeout = setTimeout(later, remaining);
        }
        return result;
    };

    // 不再控制函数执行调用频率
    throttled.cancel = function () {
        clearTimeout(timeout);
        previous = 0;
        timeout = context = args = null;
    };

    return throttled;
};
```

与我们所写的 `throttle` 不同的是，Underscore 中的 `throttle` 并不需要为每次回调都设置一个定时器来延后执行。他的定时器只记录最新一次的调用尝试。比如 `waiting` 为 `1s`，在 `1.5s` 内我们单击了查询按钮 `20` 次，真正会被送出的查询只有至多两次，分别是第一次和最后一次。这就不会出现上面延时中出现的“某时刻一些延时函数同时发生”的情况了。

在 Underscore 的 `throttle` 实现中，有个令人疑惑的判断条件：

```
if (remaining <= 0 || remaining > wait) {  
  // ...  
}
```

当该条件成立时，可以立即执行 `func`，`remaining<=0` 的条件很容易理解，就是不再需要等待时 可以执行，那么如何理解 `remaining > wait` 呢？

显然，`remaining>wait` 等同于 `now<previous`，亦即：`previous` 的被刷新晚于 `now` 的被设置。

这种情况就发生在我们当前尝试调用时，并且设置了当前时间点 `now` 之后，上次延时的函数 `later` 开始了执行，并刷新了 `previous`，此时出现了 `now` 早于 `previous` 的情况。举个栗子：

- 开始时，我们 `click` 了一次查询按钮，我们将之命名为 `click1`，此时 `previous==0`
- 在 `0.4s` 时我们 `click` 了一次查询按钮 `click2`，`now==0.4`，`previous==0`，则这次点击的查询会至少等待 `0.6s` 才送出，也就是最快要在 `1s` 的时候 `click2` 的查询请求才送出。（由于 `setTimeout(func, wait)` 并不能保证 `func` 的执行开始时间，只能保证 `func` 不早于从现在起到 `wait` 毫秒后发生，所以 `click2` 的查询请求并不一定在 `1s` 时就能够被送出）
- 在 `1.2s` 时，产生 `click3`，`now==1.2`

那么就会存在如下两种情况：

- `click2` 的查询先于 `click3` 发生，比如在 `1.1s` 时 `click2` 的回调被执行，那么 `click3` 的回调要等 `1-(1.2-1.1)==0.9s` 才发生
- `1.3s` 时 `click2` 的查询请求开始执行，`previous==1.3`，`remaining=1-(1.2-1.3)==1.1>1`，此时，Underscore 会让 `click3` 的查询请求也开始执行（既不会停止 `click` 的

查询请求，也不会停止 `click3` 的查询请求），`click3` 和 `click2` 的返回结果取最近一次。

## leading edge 与 trailing edge

underscore中的 `throttle` 函数提供了第三个参数 `options` 来进行选项配置，并且支持如下两个参数：

- `leading`：是否设置节流前缘 -- `leading edge`。前缘的作用是保证第一次尝试调用的 `func` 会被立即执行，否则第一次调用也必须等待 `wait` 时间，默认为 `true`。
- `trailing`：是否设置节流后缘 -- `trailing edge`。后缘的作用是：当最近一次尝试调用 `func` 时，如果 `func` 不能立即执行，会延后 `func` 的执行，默认为 `true`。

这两个配置会带来总共四种组合，通过[这个演示](#)，观察不同组合的效果。

## debounce

在实际项目中，我们还有一种需求，就是如果过于频繁的尝试调用某个函数时，只允许一次调用成功执行。仍然以点击查询按钮异步查询为例，假设我们每次点击的时间间隔都在 `1s` 内，那么所有的点击只有一次能送出请求，要么是第一次，要么是最后一次。显然，`throttle` 是做不到这点的，`throttle` 会至少送出两次请求。针对于此，Underscore 又撰写了 `debounce` 函数。

顾名思义，`debounce` -- 防反跳，就是不再跳起，不再响应的意思。

`throttle` 和 `debounce` 并非 Underscore 独有，他们不仅仅是函数，也是解决问题的方式，诸如 `jquery`，`lodash` 等知名库都提供了这两个方法。

从下面的 `debounce` 实现我们可以看到，不同于 `throttle`，`debounce` 不再计算 `remain` 时间，其提供的 `immediate` 参数类似于 `throttle` 中的对于 `leading-edge` 和 `trailing-edge` 的控制：

- `immediate === true`，开启 `leading-edge`，可以执行时立即执行
- `immediate === false`（默认）开启 `trailing-edge`，可以执行时也必须延后至少 `wait` 个时间才能执行。

因此，`debounce` 后的 `func` 要么立即获得响应，要么延迟一段时间才响应，[查看演示](#)。

```
_.debounce = function (func, wait, immediate) {  
    var timeout, result;  
  
    var later = function (context, args) {  
        timeout = null;  
        if (args) result = func.apply(context, args);  
    };  
  
    var debounced = restArgs(function (args) {  
        // 每次新的尝试调用func，会使抛弃之前等待的func  
        if (timeout) clearTimeout(timeout);  
        // 如果允许新的调用尝试立即执行，  
        if (immediate) {  
            // 如果之前尚没有调用尝试，那么此次调用可以立马执行，否则  
            // 就需要等待  
            var callNow = !timeout;  
            // 刷新timeout  
            timeout = setTimeout(later, wait);  
            // 如果能被立即执行，立即执行  
            if (callNow) result = func.apply(this, args);  
        } else {  
            // 否则，这次尝试调用会延时wait个时间  
            timeout = _.delay(later, wait, this, args);  
        }  
  
        return result;  
    });  
  
    debounced.cancel = function () {  
        clearTimeout(timeout);  
        timeout = null;  
    };  
  
    return debounced;  
};
```

## 应用场景

### debounce

一定要记住，`debounce` 满足的是：

高频下只响应一次

- 遇上疯狂打字员，在输入框快速输入文字（高频），但是我们只想在其完全停止输入时再对输入文字做出处理（一次）。
- AJAX，多数场景下，每个异步请求在短时间只能响应一次。比如下拉刷新，不停的到底（高频），但只发送一次 `ajax` 请求（一次）。

### throttle

相比 `debounce`，`throttle` 要更加宽松一些，其目的在于：

按频率执行调用。

1. 游戏中的按键响应，比如格斗，比如射击，需要控制出拳和射击的速率。
2. 自动完成，按照一定频率分析输入，提示自动完成。
3. 鼠标移动和窗口滚动，鼠标稍微移动一下，窗口稍微滚动一下会带来大量的事件，因而需要控制回调的发生频率。

## 参考资料

- [Debounce and Throttle: a visual explanation](#)
- [知乎@长天之云的回答](#)
- [MDN setTimeout](#)
- [浅谈 Underscore.js 中 .throttle 和 .debounce 的差异](#)
- [Underscore之throttle函数源码分析以及使用注意事项](#)

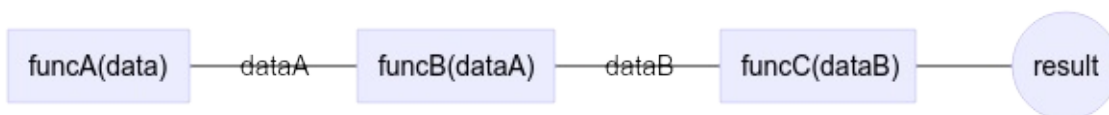
## 函数组合

### 管道（**pipeline**）

在 linux 系统中，大家一定使用过管道（pipe）：

```
cat test.sh | grep -n 'echo'
```

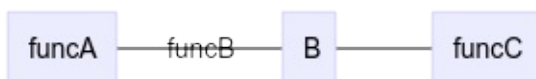
| 左边命令的输出将会作为右边命令的输入，这就构成了一条流水线：



在该视角下，我们通过不同处理器不断处理数据，并产出新的数据。管道并不是一种技术实现，而是一种技术机制，下面我们在 JavaScript 中实现上面的流水线机制：

```
function pipeline() {  
  // 传入pipeline的会是一系列函数，亦即流水线上各个工段的加工期  
  var funcs = Array.prototype.slice.call(arguments, 0, arguments  
    .length-1);  
  var datas = Array.prototype.slice.call(arguments, arguments.le  
    ngth-1);  
  return funcs.reduce(function(result, func) {  
    return func(result);  
  }, datas);  
}  
  
// 测试  
function A(data) {  
  return 2*data;  
}  
  
function B(data) {  
  return -data;  
}  
  
function C(data) {  
  return data+3;  
}  
  
pipeline(A,B,C,2); // => -1
```

再考虑另外一种流水线视角：



相比于第一种流水线视角，在第二种视角中，我们不再不断加工数据，而是不断组合我们的处理器，产生新得处理器：

```
function pipe() {
  var funcs = Array.prototype.slice.call(arguments);
  return function() {
    var result = Array.prototype.slice.call(arguments);
    return funcs.reduce(function(result, func, index) {
      if(index === 0) {
        return func.apply(this, result);
      }
      return func.call(this, result);
    }, result);
  }
}

//测试
var newFunc = pipe(A, B, C);
newFunc(2); // => -1
```

## \_.compose

`_.compose(...funcs)` : 组合各个函数为一新函数。

underscore 也为我们提供了组合函数的 API -- `_.compose` :

```
_.compose = function () {
  var args = arguments; // 函数序列
  var start = args.length - 1; // 以传入的最后一个函数作为首个处理器
  return function () {
    // 逐个执行函数，获得结果
    var i = start;
    var result = args[start].apply(this, arguments);
    while (i--) result = args[i].call(this, result);
    return result;
  };
};
```

与我们撰写的 `pipe` 不同的是，underscore 提供的 `_.compose` 接受函数的顺序与执行组合顺序是相反的。在函数式编程中，从右向左的组合顺序是标准：



```
var newFunc = _.compose(C, B ,A);  
newFunc(2); // => -1
```

## 参考资料

- [Coding by Composing](#)

## 指定调用次数

underscore 还为我们提供了一系列与函数调用次数相关的 API，下面我们逐一分析。

### \_.after

`_.after(times, func)`：创建一个函数，该函数运行 `times` 次后，才执行传入的回调 `func`。

源码：

```
_.after = function (times, func) {  
  return function () {  
    if (--times < 1) {  
      return func.apply(this, arguments);  
    }  
  };  
};
```

假设我们异步请求一个博客文章列表，当所有文章都请求成功后，才执行最终的回调：

```
var TOTAL_COUNT = 20;  
var showComplete = _.after(TOTAL_COUNT, function() {  
  console.log("请求成功");  
});  
  
for(var i=0;i<TOTAL_COUNT;i++) {  
  $.ajax({  
    url: 'getBlog',  
    success: showComplete  
  });  
}
```

## \_\_.before

`__.before(times, func)` : 创建一个函数，该函数每次调用都会执行回调函数 `func`，直至指定 `times` 次后，执行结果不再改变。

源码：

```
__.before = function (times, func) {  
    // memo暂存最近一次的调用结果，当调用次数达到times次后，memo不再改变  
    var memo;  
    return function () {  
        if (--times > 0) {  
            memo = func.apply(this, arguments);  
        }  
        // 清除函数引用，节省内存  
        if (times <= 1) func = null;  
        return memo;  
    };  
};
```

用例：

```
var n = 3;  
var before3 = __.before(3, function(){return --n;});  
before3(); //=> 3  
before3(); //=> 2;  
before3(); //=> 2; // 到第三次时，结果不再发生变化  
before3(); //=> 2;
```

## \_\_.once

`__.once(func)` : 保证 `func` 只执行一次。

源码：

```
__.once = __.partial(__.before, 2);
```

用例：

`_.once` 还可用作初始化函数：

```
var init = function() {  
  console.log("initializing!");  
}  
var initOnce = _.once(init);  
initOnce();  
initOnce();  
//=> "initializing";
```

## 拾遗

### `_.wrap`

`_.wrap(func)`：使用 `wrapper` 对 `func` 进行包裹，使 `func` 的执行前后能融入更多业务逻辑

源码：

```
_.wrap = function (func, wrapper) {  
  // 借助于偏函数将`func`传递给wrapper  
  return _.partial(wrapper, func);  
};
```

用例：

```
// 一个纯粹的函数，  
var request = function() {  
  console.log('requesting');  
}  
  
// 一个日志wrapper，每次函数执行前后，进行日志标记  
var logger = function(func) {  
  console.log('before func executing');  
  func();  
  console.log('afert func executing');  
}  
  
// 现在为函数  
var loggedRequest = _.wrap(request, logger);  
  
loggedRequest();  
// "before func executing"  
// "requesting"  
// "afert func executing"
```



## underscore 对象篇

本章中，将介绍 underscore 中操作对象的 API。

## Object(obj)

在 underscore 对象 API 中，很多函数内部都可以见到下面的一段代码：

```
var obj = Object(obj);
```

这段代码的意义是：

- 如果 `obj` 是一个对象，那么 `Object(obj)` 返回 `obj`
- 如果 `obj` 是 `undefined` 或 `null`，那么 `Object(obj)` 返回一个 `{}`
- 如果 `obj` 是一个原始值 (Primitive value)，那么 `Object(obj)` 返回一个被包裹的原始值：

```
var obj = 2;  
obj = Object(obj); // 相当于 new Number(obj);  
// => obj: Number {[[PrimitiveValue]]: 2}  
var value = obj.valueOf();  
// => value: 2
```

一言以蔽之，`Object(obj)` 就是将传入的 `obj` 进行对象化。



## 属性操作

### `_.has`

`_.has(obj, key)` : 判断 `obj` 是否含有 `key` 属性。

该函数依赖于 JavaScript 原生的 `Object.prototype.hasOwnProperty`，因而他判断的是自身的属性，而不会去寻找原型链上的属性：

```
_.has = function(obj, key) {  
  return obj != null && hasOwnProperty.call(obj, key);  
}
```

用例：

```
var foo = new Object();  
foo.bar = 'bar';  
_.has(foo, 'bar'); // => true  
_.has(foo, 'toString'); // => false
```

### `_.keys`

`_.keys(obj)` : 获得对象的所有属性名。

源码：

```
_.keys = function (obj) {  
    if (!_.isObject(obj)) return [];  
    // 如果原生的Object.keys方法存在的话，  
    // 直接调用Object.keys来获得自有属性  
    if (nativeKeys) return nativeKeys(obj);  
    var keys = [];  
    // 通过_.has, 剔除掉非自有属性  
    for (var key in obj) if (_.has(obj, key)) keys.push(key);  
    // IE9之前存在枚举bug，需要校正最后的属性集合  
    if (hasEnumBug) collectNonEnumProps(obj, keys);  
    return keys;  
};
```

如果支持 ES5 的 `Object.keys`，则用该方法获得对象的属性，否则通过 `for in` 来取得对象的 `key`。获得对象 `key` 集合之后，需要判断 `key` 集合是否应当修复，因为在 IE9 之前的版本，存在一个 bug，一些 `key` 不能被 `for key in ...` 所迭代，我们判断该 bug 的方式是：

```
var hasEnumBug = !{toString: null}.propertyIsEnumerable('toString');
```

该 bug 下不能被 `for..in` 迭代的 `key` 有：

```
var nonEnumerableProps = ['valueOf', 'isPrototypeOf', 'toString',  
    ,  
    'propertyIsEnumerable', 'hasOwnProperty', 'toLocaleString'];
```

比如，我们重载（遮蔽了）原型链上的 `toString` 方法，显然，`toString` 应当属于 `obj` 的 `key` 集合，但在 IE9 以前，可能就不是这样：

```
var obj = {
  toString: function() {
    return 'i am an object';
  },
  bar: 'bar'
};

var keys = []
for(var key in obj) {
  keys.push(key);
}
console.log(keys);
// => 小于IE9: ['bar']
// => other: ['bar', 'toString']
```

在 MDN 提供的 `Object.keys` 的 polyfill 中，也对此 bug 进行了修复，修复的思路很简单，就是如果发现了不可枚举属性被重载，则把该属性放入 `key` 集合中：

```
if (!Object.keys) {
  Object.keys = (function () {
    'use strict';
    var hasOwnProperty = Object.prototype.hasOwnProperty,
        hasDontEnumBug = !({ toString: null }).propertyIsEnumerable('toString'),
        dontEnums = [
          'toString',
          'toLocaleString',
          'valueOf',
          'hasOwnProperty',
          'isPrototypeOf',
          'propertyIsEnumerable',
          'constructor'
        ],
        dontEnumsLength = dontEnums.length;

    return function (obj) {
      if (typeof obj !== 'object' && (typeof obj !== 'function' || obj === null)) {
        throw new TypeError('Object.keys called on non-o
```

```
bject');  
    }  
  
    var result = [],  
        prop, i;  
  
    for (prop in obj) {  
        if (hasOwnProperty.call(obj, prop)) {  
            result.push(prop);  
        }  
    }  
  
    if (hasDontEnumBug) {  
        for (i = 0; i < dontEnumsLength; i++) {  
            // 如果不可枚举被重载了，则进行修复  
            if (hasOwnProperty.call(obj, dontEnums[i]))  
            {  
                result.push(dontEnums[i]);  
            }  
        }  
    }  
    return result;  
};  
}());  
}
```

而 `underscore` 则提供了一个内置函数 `collectNonEnumProps` 来修复属性：

```
var collectNonEnumProps = function (obj, keys) {
  var nonEnumIdx = nonEnumerableProps.length;
  // 通过对象构造函数获得对象的原型
  var constructor = obj.constructor;
  // 如果构造函数合法，且具有`prototype`属性，那么`prototype`就是该obj的原型
  // 否则默认obj的原型为Object.prototype
  var proto = _.isFunction(constructor) && constructor.prototype || ObjProto;

  // 如果对象有constructors属性，且当前的属性集合不存在构造函数这一属性
  var prop = 'constructor';
  // 需要将constructor属性添加到属性集合中
  if (_.has(obj, prop) && !_.contains(keys, prop)) keys.push(prop);

  // 将不可枚举的属性也添加到属性集合中
  while (nonEnumIdx--) {
    prop = nonEnumerableProps[nonEnumIdx];
    // 注意，添加的属性只能是自有属性 (obj[prop] !== proto[prop])

    if (prop in obj && obj[prop] !== proto[prop] && !_.contains(keys, prop)) {
      keys.push(prop);
    }
  }
};
```

与 MDN 中的 polyfill 不同的是，`collectNonEnumProps` 没有通过 `Object.prototype.hasOwnProperty` 来判断属性的自有性，而是通过 `obj[prop] !== proto[prop]` 来判断。

用例：

```
var obj = {
  toString: function() {
    return 'i am an object';
  },
  bar: 'bar'
};

_.keys(obj);
// => ["toString", "bar"]
```

## **\_.allKeys**

`_.allKeys(obj)` : 获得 `obj` 所有属性，包括原型链上的属性。

源码：

```
_.allKeys = function (obj) {
  if (!_isObject(obj)) return [];
  var keys = [];
  for (var key in obj) keys.push(key);
  // Ahem, IE < 9.
  if (hasEnumBug) collectNonEnumProps(obj, keys);
  return keys;
};
```

用例：

```
function Person(name) {  
    this.name = name;  
}  
  
Person.prototype.say = function() {  
    console.log('hello');  
}  
  
var wxj = new Person('wxj');  
_.keys(wxj);  
// => ['name']  
  
_.allKeys(wxj);  
// => ['name', 'say']
```

## \_.values

`_.values(obj)` : 获得 `obj` 的所有值。

源码：

```
_.values = function (obj) {  
    // 很简单，遍历key集合，取到对应的value  
    var keys = _.keys(obj);  
    var length = keys.length;  
    // 定长初始化, 提前分配内存空间  
    var values = Array(length);  
    for (var i = 0; i < length; i++) {  
        values[i] = obj[keys[i]];  
    }  
    return values;  
};
```

用例：

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

## **.invert**

`.invert(obj)` : 调换 `obj` 的属性和值。

源码：

```
_.invert = function (obj) {  
  var result = {};  
  var keys = _.keys(obj);  
  for (var i = 0, length = keys.length; i < length; i++) {  
    result[obj[keys[i]]] = keys[i];  
  }  
  return result;  
};
```

用例：

```
var obj = {name: 'wxj', age:13};  
var newObj = _.invert(obj);  
// => {wxj:name, 13:'age'}
```

## **\_.functions**

`_.functions(obj)` : 获得 `obj` 含有的方法。

源码：

```
_.functions = _.methods = function (obj) {  
  var names = [];  
  for (var key in obj) {  
    if (_.isFunction(obj[key])) names.push(key);  
  }  
  return names.sort();  
};
```

注意到，最后返回的函数列表会按字典序进行排序。



用例：

```
var obj = {  
  say: function() {  
    console.log("hello");  
  },  
  run: function() {  
    console.log("running");  
  }  
};  
var funcs = _.functions(obj);  
// funcs: ["run", "say"];
```

# 克隆与扩展

## 扩展

underscore 中提供了两个方法用于对象属性的扩展：

- `_.extend`
- `_.extendOwn`

这两个函数都是由内部函数 `createAssigner` 进行创建的。

### `createAssigner`

`createAssigner` 接受 2 个参数：

- `keysFunc` ：获得对象属性的方式
- `defaults` ：来声明是否要覆盖属性

返回一个属性分配器。

```
var createAssigner = function (keysFunc, defaults) {
  return function (obj) {
    // 参数的长度反映了传入对象的个数
    var length = arguments.length;
    if (defaults) obj = Object(obj);
    if (length < 2 || obj == null) return obj;
    // 遍历每个对象，从中不断获得属性，赋给obj
    for (var index = 1; index < length; index++) {
      var source = arguments[index],
          keys = keysFunc(source),
          l = keys.length;
      for (var i = 0; i < l; i++) {
        var key = keys[i];
        // 如果不是defaults模式，则会覆盖原来key的value
        // 如果是defaults模式，则只会设置原来为undefined的属性
        if (!defaults || obj[key] === void 0) obj[key] =
source[key];
      }
    }
    return obj;
  };
};
```

## .\_extends

.\_extends(obj, ...objs) : 用 ...objs 扩展 obj 。

源码：

```
// _.allKeys会获得对象自身及其原型链上的所有属性
_.extends = createAssigner(_.allKeys)
```

用例：

```
var student = {
  name: 'yoyoyohamapi',
  age: 13,
  sex: 'girl'
};
function Hobby(){
  this.hobby = "soccer";
}
Hobby.prototype.hate = "volleyball";
_.extend(student, new Hobby(), {sex: 'boy'});
// => student: {
//   name: "yoyoyohamapi",
//   age: 13,
//   hobby: "soccer",
//   hate: "volleyball", // 原型链上的属性也被继承了
//   sex: "boy" // 会覆盖属性
//}
```

## \_.extendOwn = \_.assign

`_.extendOwn(obj, ...objs)` : 用 `...objs` 扩展 `obj`，不会继承原型链上的属性。

源码：

```
// _.keys只会获得自身属性
_.extendOwn = _.assign = createAssigner(_.keys);
```

用例：

```
var student = {
  name: "yoyoyohamapi",
  age: 13,
  sex: 'girl'
};
function Hobby(){
  this.hobby = "soccer";
}
Hobby.prototype.hate = "volleyball";
_.extend(student, new Hobby(), {sex: 'boy'});
// => student: {
//   name: "yoyoyohamapi",
//   age: 13,
//   hobby: "soccer",
//   sex: "boy" // 会存在属性覆盖
//}
```

## \_.defaults

`_.defaults(obj, attrs)` : 设置 `obj` 的默认属性。

在创建一个库或者组件时，我们会为这个库或者组件定义一个基本配置，考虑到灵活性，也允许用户自定义一些配置项，最终，这个插件的配置将会有 用户配置 和 默认配置 共同决定。`underscore` 中的 `_.defaults` 函数正是用来完成这个任务的。

源码：

```
_.defaults = createAssigner(_.allKeys, true);
```

通过向 `createAssigner` 传递第二个参数 `true`，后续的对象都会填充 `obj` 中为 `undefined` 的属性。

用例：

```
var defaultOptions = {
  method: 'GET',
  'content-type': 'application/json',
  accept: 'application/json'
};

var options = {
  method: 'POST'
};

var opts = _.defaults(options, defaultOptions);
// opts => {
//   method: 'POST',
//   'content-type': 'application/json',
//   accept: 'application/json'
//}
```

## Object.assign

通过 MDN 给出的 polyfill 看出，`Object.assign` 在扩展对象属性的时候，不会拷贝原型链上的属性：

```
if (typeof Object.assign !== 'function') {
  Object.assign = function(target) {
    'use strict';
    if (target == null) {
      throw new TypeError('Cannot convert undefined or null to object');
    }

    target = Object(target);
    for (var index = 1; index < arguments.length; index++) {
      var source = arguments[index];
      if (source !== null) {
        for (var key in source) {
          if (Object.prototype.hasOwnProperty.call(source, key)) {
            target[key] = source[key];
          }
        }
      }
    }
    return target;
  };
}
```

用例:

```
var student = {
  name: 'yoyoyohamapi',
  age: 13,
  sex: 'girl'
};
function Hobby(){
  this.hobby = "soccer";
}
Hobby.prototype.hate = "volleyball";
Object.assign(student, new Hobby(), {sex: 'boy'});
// => student: {
//   name: "yoyoyohamapi",
//   age: 13,
//   hobby: "soccer",
//   sex: "boy" // 会覆盖属性
//}
```

注意，无论是 `underscore` 提供的 `extend` 相关方法还是 `Object.assign`，对象属性的复制都是浅复制：

```
var son = {
  family: {
    dad: 'wxj'
  }
}

var daughter = _.extend({}, son);
daughter.family === son.family; // true

daughter = Object.assign({}, son);
daughter.family === son.family; // true
```

## 克隆

`underscore` 中提供了一个方法用于克隆一个对象：

- `_.clone`



## \_.clone

`_.clone(obj)` : 克隆一个 `obj` 的对象副本。

`underscore` 提供的克隆是浅克隆，亦即，对象中的数组、对象属性引用不会发生变化：

```
_.clone = function (obj) {  
  if (!_isObject(obj)) return obj;  
  // 可以看出，如果obj是一个对象，那么该函数基于_.extend实现  
  return _.isArray(obj) ? obj.slice() : _.extend({}, obj);  
};
```

用例：

```
var family = {  
  name: '304',  
  dad: {  
    name: 'wxj',  
    age: 32  
  }  
}  
  
var cloned = _.clone(family);  
  
// 克隆是浅克隆  
cloned.dad === family.dad; // => true  
family.dad.name = 'john';  
cloned.dad.name; // => 'john'
```

## 深度克隆

`underscore` 并没提供深度克隆的方法，不过借助于递归，我们构造一个深度克隆的方法：

```
var deepClone = function(obj) {  
  // 如果待克隆对象是数组  
  if(_.isArray(obj)) {  
    return _.map(obj, function(elem){  
      return _.isArray(elem) || _.isObject(elem) ? deepClone(e  
lem) : elem;  
    });  
  } else if(_.isObject(obj)){  
    return _.reduce(obj, function(newObj, value, key){  
      newObj[key] = _.isArray(value) || _.isObject(value) ? de  
epClone(value) : value;  
      return newObj;  
    }, {});  
  } else {  
    return obj;  
  }  
}
```

测试：

```
var family = {  
  name: '304',  
  dad: {  
    name: 'wxj',  
    age: 32  
  }  
}  
  
var cloned = _.clone(family);  
  
// 克隆是浅克隆  
cloned.dad === family.dad; // => false  
family.dad.name = 'john';  
cloned.dad.name; // => 'wxj'
```

深度克隆并非那么容易实现的，他包含有太多语义界定和边界用例，详情可以参看知乎上的这篇讨论：

[JavaScript 如何完整实现深度克隆对象？](#)



## 白名单

有时候，我们只需要一个对象的部分属性，即丢弃到该对象部分属性，例如，在一个 `server` 程序中，我们最终发到后端的请求包会过滤掉一些他不需要的参数。在 `underscore` 中，就通过 `_.pick` 方法来达到这一目的。

### `_.pick(obj, whitelist)`

`_.pick(obj, whitelist)` ：跟据白名单 `whitelist` ，选出 `obj` 中的属性

`_.pick` 方法不是原地的，他将会返回一个新对象，而不是直接在 `obj` 上直接过滤属性。白名单 `whitelist` 既支持传递参数序列（`keys`），也支持传递一个函数来做过滤规则

源码：

```
_.pick = restArgs(function (obj, keys) {
  var result = {}, iteratee = keys[0];
  if (obj == null) return result;
  // 白名单机制最终都会落脚到一个真值检验函数上，无论第二个参数是一个函数
  // 还是一个个的keys
  if (_.isFunction(iteratee)) {
    if (keys.length > 1) iteratee = optimizeCb(iteratee, keys[1]);
    keys = _.allKeys(obj);
  } else {
    // 如果第二个参数是一系列的key，则暗示pick的条件根据key是否在obj
    // 中进行pick
    iteratee = keyInObj;
    // 将keys展平
    keys = flatten(keys, false, false);
    // 对象化`obj`，保证后续操作顺利进行
    obj = Object(obj);
  }
  // 开始pick
  for (var i = 0, length = keys.length; i < length; i++) {
    var key = keys[i];
    var value = obj[key];
    if (iteratee(value, key, obj)) result[key] = value;
  }
  return result;
});
```

用例：

```
var student = {
  name: 'wxj',
  age: 13,
  family: {
    son: 'lcx'
  },
  hobby: 'soccer'
};
var student2 = _.pick(student, 'name', 'age');
var student3 = _.pick(student, function(value, key, obj){return
key === 'age';})
// =>
// student2: {
//   name: 'wxj',
//   age: 13
// },
// student3: {
//   age: 13
// }
```

## **`_.omit(obj, blacklist)`**

`_.omit(obj, blacklist)` : 与 `_.pick` 相反，过滤掉黑名单 `blacklist` 中的属性

`_.omit` 的实现很容易，就是通过 `_.negate` 对传入的白名单函数取反（构成了黑名单函数），再执行 `_.pick`，再次体现了 `underscore` 中高度的函数复用：

```
// 1.8.3
_.omit = function(obj, iteratee, context) {
  if (_.isFunction(iteratee)) {
    iteratee = _.negate(iteratee);
  } else {
    var keys = _.map(flatten(arguments, false, false, 1), String);
    iteratee = function(value, key) {
      return !_.contains(keys, key);
    };
  }
  return _.pick(obj, iteratee, context);
};

// master
_.omit = restArgs(function (obj, keys) {
  // 可以看到，_.omit的实现基于_.pick
  var iteratee = keys[0], context;
  if (_.isFunction(iteratee)) {
    iteratee = _.negate(iteratee);
    // 如果传入了条件函数，那么取下一个参数为执行上下文
    if (keys.length > 1) context = keys[1];
  } else {
    keys = _.map(flatten(keys, false, false), String);
    iteratee = function (value, key) {
      return !_.contains(keys, key);
    };
  }
  return _.pick(obj, iteratee, context);
});
```

用例：

```
var student = {  
  name: 'wxj',  
  age: 13,  
  family: {  
    son: 'lcx'  
  },  
  hobby: 'soccer'  
};  
var student2 = _.pick(student, 'family', 'hobby');  
// =>  
// student2: {  
//   name: 'wxj',  
//   age: 13  
// },
```



## 对象相等性判断

### 原生 JavaScript 的比较问题

在原生 JavaScript 中，在判断 `a` 与 `b` 的相等性时，面临如下的比较问题：

- `0 === -0`
- `null == undefined`
- `NaN != NaN`
- `NaN !== NaN`

`0 === -0` :

对于该问题，我们可以借助如下等式解决：

```
1/0 === 1/0  
1/-0 !== 1/0
```

`null == undefined` :

对于该问题，我们可以通过如下等式解决：

```
null === null;  
null !== undefined;
```

`NaN != NaN` 及 `NaN !== NaN` :

如果我们要认为 `NaN` 等于 `NaN`（这更加符合认知和语义），我们只需要：

```
if(a !== a)  
    return b !== b;
```

### `_.isEqual(a, b)`

`_.isEqual(a, b)` : 判断 `a` , `b` 是否相等

在 `underscore` 中提供了该函数用于判断两个变量是否相等，其源码如下：

```
_.isEqual = function (a, b) {  
    return eq(a, b);  
};
```

其内部是通过 `eq` 函数实现的：

```
eq = function (a, b, aStack, bStack)  
    if (a === b) return a !== 0 || 1 / a === 1 / b;  
    if (a == null || b == null) return a === b;  
    if (a !== a) return b !== b;  
    // Exhaust primitive checks  
    var type = typeof a;  
    if (type !== 'function' && type !== 'object' && typeof b !==  
        'object') return false;  
    // 如果a,b是function或者object（数组也是object）则意味着二者需要深度比较  
    return deepEq(a, b, aStack, bStack);  
};
```

在 `eq` 的源码中，我们也发现，当 `a`，`b` 是数组或者对象时，会调用 `deepEq` 进行深度比较

在 1.8.3 版本中，没有区分 `eq` 及 `deepEq` 方法

## deepEq：深度比较

对于如下几个类型（通过 `Object.prototype.toString()` 取到），需要进行深度比较：

- `[object RegExp]`：正则表达式
- `[object String]`：字符串
- `[object Number]`：数字
- `[object Date]`：日期
- `[object Boolean]`：Bool值
- `[object Symbol]`：Symbol

这些类型的比较方式在 `deepEq` 中源码中可以轻松的理解，希望大家能仔细阅读，掌握这些比较的 trick：

```
deepEq = function(a, b, aStack, bStack) {

  // ...

  var className = toString.call(a);
  if (className !== toString.call(b)) return false;
  switch (className) {
    // Strings, numbers, regular expressions, dates, and booleans are compared by value.
    case '[object RegExp]':
      // RegExps are coerced to strings for comparison (Note: '' + /a/i === '/a/i')
    case '[object String]':
      // Primitives and their corresponding object wrappers are equivalent; thus, `"5"` is
      // equivalent to `new String("5")`.
      return '' + a === '' + b;
    case '[object Number]':
      // `NaN`s are equivalent, but non-reflexive.
      // Object(NaN) is equivalent to NaN.
      if (+a !== +a) return +b !== +b;
      // An `egal` comparison is performed for other numeric values.
      return +a === 0 ? 1 / +a === 1 / b : +a === +b;
    case '[object Date]':
    case '[object Boolean]':
      // Coerce dates and booleans to numeric primitive values. Dates are compared by their
      // millisecond representations. Note that invalid dates with millisecond representations
      // of `NaN` are not equivalent.
      // var a = new Date();
      // +a; // => 1464867835038
      return +a === +b;
    case '[object Symbol]':
      return SymbolProto.valueOf.call(a) === SymbolProto.valueOf.call(b);
```

```
    }  
  
    // ...  
  
}
```

## 对象及数组的深度比较

而对于对象（键值对）和数组的深度比较就较为繁琐了，需要逐个比较当中的元素，并且这些元素也可能是数组或者对象，所以将会用到递归。

由于递归是耗时的，当两个变量不是数组，而是一般的实例化对象时，可以先考虑二者的不等性：当二者的构造函数就不等时，则认为二者不等：

```
deepEq = function(a, b, aStack, bStack) {  
  
    // ...  
    var areArrays = className === '[object Array]';  
    if (!areArrays) {  
        if (typeof a !== 'object' || typeof b !== 'object') return  
false;  
  
        // Objects with different constructors are not equivalent,  
        // but `Object`s or `Array`s  
        // from different frames are.  
        // 构造函数不同的对象必然不等，  
        var aCtor = a.constructor, bCtor = b.constructor;  
        if (aCtor !== bCtor && !(_.isFunction(aCtor) && aCtor in  
instanceof aCtor &&  
            _.isFunction(bCtor) && bCtor instanceof bCtor)  
            && ('constructor' in a && 'constructor' in b)) {  
            return false;  
        }  
    }  
    // ...  
  
}
```

而对于数组或者对象这样包含元素或者属性的结构，需要用如下的递归流程来进行比较：

```
deepEq = function(a, b, aStack, bStack) {

    // ...
    aStack = aStack || [];
    bStack = bStack || [];
    var length = aStack.length;
    while (length--) {
        // Linear search. Performance is inversely proportional to
        the number of
        // unique nested structures.
        if (aStack[length] === a) return bStack[length] === b;
    }

    // Add the first object to the stack of traversed objects.
    aStack.push(a);
    bStack.push(b);

    // Recursively compare objects and arrays.
    if (areArrays) {
        // Compare array lengths to determine if a deep comparison
        is necessary.
        length = a.length;
        if (length !== b.length) return false;
        // Deep compare the contents, ignoring non-numeric propert
        ies.
        while (length--) {
            if (!eq(a[length], b[length], aStack, bStack)) return fa
            lse;
        }
    } else {
        // Deep compare objects.
        var keys = _.keys(a), key;
        length = keys.length;
        // Ensure that both objects contain the same number of pro
        perties before comparing deep equality.
        if (_.keys(b).length !== length) return false;
        while (length--) {
```

```
        // Deep compare each member
        key = keys[length];
        if (!(_.has(b, key) && eq(a[key], b[key], aStack, bStack
))) return false;
    }
}
// Remove the first object from the stack of traversed objects.
aStack.pop();
bStack.pop();
return true;
// ...

}
```

很多 `underscore` 分析文章中对 `aStack` 及 `bStack` 的作用一笔带过，说是暂存 `a, b` 的值，这么说，和不分析也区别不大。

实际上，作者在这里的 `aStack` 和 `bStack` 是用于检测循环引用的，算法参考自 [ES 5.1 15.12.3 章节中的 JO](#)。假定我有如下的数组或者对象，当中包含循环引用：

```
var aArr = [1, 2, 3];
aArr[3] = aArr;
var bArr = [1, 2, 3];
bArr[3] = bArr;

var aObj = {
  title: 'underscore'
};
aObj.toSelf = aObj;

var bObj = {
  title: 'underscore'
};
bObj.toSelf = bObj;
```

如果不借助于上面提到的算法，不借助于辅助的空间 `aStack` 以及 `bStack`，仅仅是使用普通的递归，那么遇到上述的循环引用结构（`cyclic structure`），将陷入无穷递归，由于js会对递归调用次数进行限制，所以会报错：

```
// 递归...

// 比较: aArr[3], bArr[3]
// 比较: aArr[3], bArr[3]
// 比较: aArr[3], bArr[3]
// 比较: aArr[3], bArr[3]
// Uncaught RangeError: Maximum call stack size exceeded

// ....
```

而借助于 `aStack` 以及 `bStack`，我们将能避开无限递归，过程如下图所示：

```
// 递归...
aStack PUSH: [aArr]
bStack PUSH: [bArr]

// 比较: aArr[3], bArr[3]
// 因为：
aStack[0] === aArr[3]
bStack[0] === bArr[3]
// 所以：
aArr[3] === bArr[3]
return true

// ...
```

从中可以看到，`aStack` 及 `bStack` 这两个栈的目的是：

记录了上级（父级）的引用，当子元素指向父元素时（循环引用），可以通过引用比较跳出无限递归。

## 测试

```
var aArr = [1, 2, 3];
aArr[3] = aArr;
var bArr = [1, 2, 3];
bArr[3] = bArr;

var aObj = {
  title: 'a'
};
aObj.toSelf = aObj;

var bObj = {
  title: 'b'
};
bObj.toSelf = bObj;
_.isEqual(aObj, bObj); // => false

var aObj = {
  title: 'underscore'
};
aObj.toSelf = aObj;

var bObj = {
  title: 'underscore'
};
bObj.toSelf = bObj;

_.isEqual(aArr, bArr); // => true
_.isEqual(aObj, bObj); // => true
_.isEqual(0, -0); // => false
_.isEqual(NaN, NaN); // => true
```



## 类型判断

underscore 中提供了多个 `isXXX` 方法用以判断对象属于哪种“类型”。

大部分元素都可以通过 `Object.prototype.toString` 的结果进行判断，比如：

```
var string = new String('wxj');
Object.prototype.toString.call(string); // => '[object String]'
```

对于这些类型，underscore 皆是通过 `Object.prototype.toString` 来进行类型判断：`Arguments`，`Function`，`String`，`Number`，`Date`，`RegExp`，`Error`，`Symbol`，`Map`，`WeakMap`，`Set`，`WeakSet`：

```
_.each(['Arguments', 'Function', 'String', 'Number', 'Date', 'RegExp', 'Error', 'Symbol', 'Map', 'WeakMap', 'Set', 'WeakSet'], function (name) {
  _['is' + name] = function (obj) {
    return toString.call(obj) === '[object ' + name + ']';
  };
});
```

### `_.isArguments(obj)`

`_.isArguments(obj)`：判断 `obj` 是否是 `arguments`。

对于 `Arguments` 判断，IE9 以前的版本，`Object.prototype.toString` 返回的会是 `'[object Object]'` 而不是 `'[object Arguments]'`，需要通过判断对象是否具有 `callee` 来确定其是否为 `Arguments` 类型。underscore 对此进行了修正：

```
if (!_.isArguments(arguments)) {  
  // 判断obj是否是arguments  
  // 只有arguments才会有callee属性  
  _.isArguments = function (obj) {  
    return _.has(obj, 'callee');  
  };  
}
```

## \_.isFunction(obj)

`_.isFunction(obj)` : 判断 `obj` 是否是函数。

此外，`underscore` 还修正了在早期 V8 引擎下的对于对象是否为函数的判断：

```
var nodelist = root.document && root.document.childNodes;  
if (typeof /./ != 'function' && typeof Int8Array != 'object' &&  
typeof nodelist != 'function') {  
  _.isFunction = function (obj) {  
    return typeof obj == 'function' || false;  
  };  
}
```

## 用例

```
_.isNumber(1/0);           // => true  
_.isNumber('2');           // => false  
_.isDate(new Date());      // => true  
_.isString('');            // => true  
_.isRegExp(/[0-9]/);       // => true  
_.isError(new Error('error')); // => true  
_.isSymbol(Symbol());      // => true
```

## \_.isElement(obj)

`_.isElement(obj)` : 判断 `obj` 是否是 DOM 节点

源码：

```
_.isElement = function (obj) {  
  return !(obj && obj.nodeType === 1);  
};
```

该方法是通过 `Node.nodeType` 来判断元素是否为 DOM 节点。`nodeType` 的取值参看 [MDN](#)。

用例：

```
var body = document.getElementsByTagName('body')[0];  
_.isElement(body); // => true
```

## \_.isArray

`_.isArray(obj)`：判断 `obj` 是否是数组

源码：

```
_.isArray = nativeisArray || function (obj) {  
  return toString.call(obj) === '[object Array]';  
};
```

对于数组的判断，优先使用ES5中的 `Array.isArray`，否则通过 `Object.prototype.toString` 来进行判断。

用例：

```
var arr = [1, 2, 3, 4];  
_.isArray(arr); // => true
```

## \_.isObject

`_.isObject(obj)`：判断 `obj` 是否是对象

源码：

```
_.isObject = function (obj) {  
    var type = typeof obj;  
    return type === 'function' || type === 'object' && !!obj;  
};
```

看到源码可以知道，函数也被视为对象，`undefined`，`null`，`NaN` 等则不被认为是对象

用例：

```
_.isObject(Object.keys); // => true  
_.isObject(null); // => false  
_.isObject(undefined); // => false  
_.isObject(NaN); // => false  
_.isObject({}); // => true
```

## `_.isFinite`

`_.isFinite`：判断 `obj` 是否有限

源码：

```
// master  
_.isFinite = function (obj) {  
    return !_.isSymbol(obj) && isFinite(obj) && !isNaN(parseFloat(obj));  
};  
  
// 1.8.3  
_.isFinite = function(obj) {  
    return isFinite(obj) && !isNaN(parseFloat(obj));  
};
```

该函数依赖于 JavaScript 全局提供的 `isFinite`，`isFinite(obj)` 之后为何还要接上 `isNaN(parseFloat(obj))` 是进行一些修正，作者认为，`bool` 值不应当考虑其有限性：

```
isFinite(true); // => true
!isNaN(parseFloat(true)); // => false

isFinite(null); // => true
!isNaN(parseFloat(null)); // => false;
```

用例：

```
_.isFinite('2000'); // true
_.isFinite(1/0);    // false
_.isFinite(Infinity); // false
_.isFinite(true);   // false
_.isFinite(null);   // false
```

## \_\_\_.isNaN

`___.isNaN(obj)`：判断 `obj` 是否是一个非数字

源码：

```
___.isNaN = function (obj) {
  return _.isNumber(obj) && isNaN(obj);
};
```

相比较原生的 `isNaN`（原生的 `isNaN` 会先传入参数转为数字），`underscore` 强制限定 `NaN` 必须为数字。

用例：

```
// 我们对比原生的`NaN`和`_.NaN`
isNaN('NaN'); // => true
_.isNaN('NaN'); // => false
_.isNaN(NaN); // => true

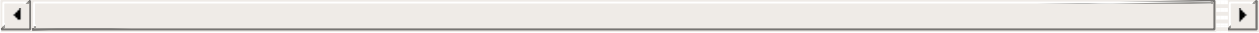
isNaN(undefined); // => true
_.isNaN(undefined); // => false
```

## \_.isBoolean

`_.isBoolean` : 判断 `obj` 是否是布尔类型

源码：

```
_.isBoolean = function(obj) {
  return obj === true || obj === false || toString.call(obj) ===
    '[object Boolean]';
};
```



用例：

```
_.isBoolean(undefined); // => false
_.isBoolean(true); // => true
```

## \_.isNull

`_.isNull` : 判断 `obj` 是否是 `null`

源码：

```
_.isNull = function(obj) {
  return obj === null;
};
```

用例：

```
_.isNull(null); // => true
```

## \_.isUndefined

\_.isUndefined : 判断 obj 是否是 undefined

源码：

```
_.isUndefined = function(obj) {  
  return obj === void 0;  
};
```

用例：

```
_.isUndefined(undefined); // => true
```

## 工具函数

underscore 还提供了不少工具函数，来提供一些周边功能，如字符逃逸等。但其中最重要的是其提供的模板引擎工具，我将会花费很大笔墨对其进行描述。



## 工具汇总

### `_.noConflict`

`_.noConflict()`：将 `_` 对象的所有权交还。

源码：

```
_.noConflict = function () {  
    // 回复原来的_指代的对象  
    root._ = previousUnderscore;  
    // 返回underscore对象  
    return this;  
};
```

假定在我们的工程中，已经占据了 `_` 对象，那么，之后我们引入 `underscore`，`_` 就会被 `underscore` 夺走：

```
_ = {  
    owner: 'wxj'  
};  
  
// 引入underscore后  
  
console.log(_.owner); // => undefined
```

`_.noConflict` 能够返回被 `underscore` 占据的 `_`，并重新制定 `underscore` 对象：

```
var underscore = _.noConflict();  
  
console.log(_.owner); // => 'wxj'
```

### `_.identity`

`_.identity(value)` : 返回 `value` 本身。

源码：

```
_.identity = function (value) {  
    return value;  
};
```

很多没接触过 FP 编程的开发者看到这个方法会很诧异，要获得值本身的话，直接用 `value` 不就好，干嘛还有包一层函数？假定有这样一个情景，我们想要复制一个数组：

```
var arr = [1,2,3,4,5];  
var dstArr = arr.map((item, index) => {  
    return item;  
});
```

而借助于 `_.identity` 方法，我们可以这样做：

```
var arr = [1,2,3,4,5];  
var dstArr = arr.map(_.identity);
```

## `_.constant`

`_.constant(value)` : 返回一个函数，该函数将返回 `value` 本身。

源码：

```
_.constant = function (value) {  
    return function () {  
        return value;  
    };  
};
```

似乎比 `_.identity` 更加过分，更加匪夷所思，因为 `_.constant` 又包了一层。但我们注意到，`_.constant` 内部形成了什么？对的，一个闭包，他将为我们缓存住当时的 `value`，最终返回的所谓 常量 不过就是当时缓存的值：

```
var a = 2;
var getConstA = _.constant(a);

a = 3;
getConstA(); // => 2
```

## `_.noop`

`_.noop`：保存了一个空函数的引用。

源码：

```
_.noop = function () {
};
```

`_.noop` 指向了一个空函数引用，避免了创建空函数时不必要的开销，也节省了可能存在的判断开销：

```
var a = {
  doSomething: null;
};
a.doSomething = function() {console.log("doing");}
// 某个业务需要的地方，需要额外判断
_.isFunction(a.doSomething) && a.doSomething();
// 现在
var a = {
  doSomething: _.noop
};
a.doSomething = function() {console.log("doing");}
// 不再需要判断
a.doSomething();
```

## \_.propertyOf

`_.propertyOf(obj)` : 返回一个属性获得函数，该函数能够获得 `obj` 的属性。

源码：

```
_.propertyOf = function (obj) {  
  return obj == null ? function () {} : function (key) {  
    return obj[key];  
  };  
};
```

可能又有人觉得大材小用了，要获得对象属性直接用 `.` 操作符不就好了吗，比如 `obj.name`。考虑一个场景，我们想要获得一个对象所有的值，传统的做法会是：

```
var student = {  
  name: 'wxj',  
  age: 18  
};  
var values = _.keys(student).map((key) => student[key]); // ['wxj', 18]
```

借助于 `_.propertyOf`，将可以这样做：

```
getStudentProp = _.propertyOf(student);  
var values = _.keys(student).map(getStudentProp);
```

## \_.times

`_.times(n, iteratee)` : 执行函数 `iteratee` `n` 次，返回保存了每次执行结果的数组。

源码：

```
_.times = function (n, iteratee, context) {  
  var accum = Array(Math.max(0, n));  
  iteratee = optimizeCb(iteratee, context, 1);  
  for (var i = 0; i < n; i++) accum[i] = iteratee(i);  
  return accum;  
};
```

从源码中可以看到，`iteratee` 将会是一个迭代函数，并且其第一个参数将会被传入当前迭代的索引值。

用例：

```
function getIndex(index) {  
  return index;  
}  
  
var indexes = _.times(3, getIndex);  
// => [0,1,2]
```

## \_.random

`_.random(min, max)`：返回 `[min,max]` 之间的随机数，如果没有传递 `max`，则返回 `[0,min]` 之间的随机数。

源码：

```
_.random = function (min, max) {  
  // 如果没有设定右边界，则返回[0, min]之间的随机数  
  if (max == null) {  
    max = min;  
    min = 0;  
  }  
  return min + Math.floor(Math.random() * (max - min + 1));  
};
```

用例：

```
console.log(_.random(0,9)); // => 6
console.log(_.random(5)); // => 3
```

## \_\_.now

\_\_.now() : 方法返回自 1970 年 1 月 1 日 00:00:00 UTC 到当前时间的毫秒数。

源码：

```
__.now = Date.now || function () {
    return new Date().getTime();
};
```

该方法会优先尝试使用 ES5.1 提供的 `Date.now()` 方法。

用例：

```
var now = __.now(); // => 1482133330807
var nowDate = new Date(now); // => Mon Dec 19 2016 15:42:10 GMT+0800 (CST)
```

## \_\_.result

\_\_.result(obj, prop, fallback) : 如果 `prop` 是 `obj` 的成员方法，则调用之，否则获得 `prop` 属性。若 `obj.prop` 为 `undefined`，可以传递一个 `fallback` 做为后置处理。

源码：

```
__.result = function (object, prop, fallback) {
    var value = object == null ? void 0 : object[prop];
    if (value === void 0) {
        value = fallback;
    }
    return __.isFunction(value) ? value.call(object) : value;
}
```

用例：

```
var student = {
  name: 'wxj',
  sayHello: function () {
    console.log('hello');
  }
}
_.result(student, 'sayHello'); // => 'hello'
_.result(student, 'name'); // => 'wxj'
_.result(student, age, 13); // => 13
```

## \_\_.uniqueId

\_\_.uniqueId(prefix)：根据传入的 prefix，生成唯一 ID。

源码：

```
var idCounter = 0;

/**
 * 根据传入的prefix，产生全局唯一的id
 * @param {string} prefix id前缀
 */
__.uniqueId = function (prefix) {
  var id = ++idCounter + '';
  return prefix ? prefix + id : id;
};
```

可以看到，该id产生器的起点为 1；

用例：

```
var prefix = "wxj";
__.uniqueId(prefix); // => wxj1
__.uniqueId(prefix); // => wxj2
```

## createEscaper

underscore 中提供了 `_.escape(string)` 及 `_.unescape(string)` 来对 HTML 字符串进行逃逸 / 反逃逸，他们都由内部函数 `createEscaper(map)` 进行创建：

```
// 定义一系列需要逃逸的html字符
var escapeMap = {
  '&': '&amp;',
  '<': '&lt;',
  '>': '&gt;',
  '"': '&quot;',
  "'": '&#x27;',
  '`': '&#x60;';
};

var createEscaper = function (map) {
  var escaper = function (match) {
    // 每次捕获通过map中创建的映射进行替换
    return map[match];
  };
  // Regexes for identifying a key that needs to be escaped.
  // 动态创建正则表达式，不捕获
  var source = '(?:' + _.keys(map).join('|') + ')';
  // 测试正则与替换正则
  var testRegexp = RegExp(source);
  var replaceRegexp = RegExp(source, 'g');

  return function (string) {
    string = string == null ? '' : '' + string;
    return testRegexp.test(string) ? string.replace(replaceRegexp, escaper) : string;
  };
};
```

`createEscaper` 接受一个映射表 `map` 参数，该映射表反映的是待转义字符与转义后字符的映射关系，之后，`createEscaper` 会动态创建匹配测试正则 `testRegexp` 及替换正则 `replaceRegexp`，最终返回的转义函数将利用这两个



正则完成 HTML 字符的转义或者反转义。

从 `createEscaper` 源码中我们也看到了，`String.prototype.replace(regex|substr, newSubStr|function)` 的第二个参数不仅能够接受字符串对匹配的到字符串进行直接替换，还能接受函数进行间接替换，该函数接受的一个参数，其代表的是匹配到的字符子串。

## `_.escape`

`_.escape(string)`：对 `string` 进行 HTML 字符逃逸。

源码：

```
/**
 * 转义html特殊字符
 * @param {string} string 待转义字符
 */
_.escape = createEscaper(escapeMap);
```

用例：

```
var html = '<a href="http://yoyoyohamapi.me">吴小姐的博客</a>';
var escaped = _.escape(html);
// => '&lt;a href=&quot;http://yoyoyohamapi.me&quot;&gt;吴小姐的博客&lt;/a&gt;'
```

## `_.unescape`

`_.unescape(string)`：对 `string` 进行 HTML 字符反逃逸。

源码：

```
/**
 * 取消转义
 * @param {string} string 待取消转义字符
 */
_.unescape = createEscaper(unescapeMap);
```

用例：

```
var html = '<a href="http://yoyoyohamapi.me">吴小姐的博客</a>';
var unescaped = _.unescape(escaped);
// => '<a href="http://yoyoyohamapi.me">吴小姐的博客</a>'
```

# 模板引擎

## 什么是模板

我喜欢这样描述模板引擎（系统）：

- 给定一个模板字符串，不同于一般的字符串，该字符串存在一些规则，并且需要填充数据或者逻辑：

```
const tpl = 'hello {{name}}'
```

`{{name}}` 就描述了一个规则：插值（interpolation），双花括号内的变量 `name` 将在之后被替换。

- 通过一个渲染器对该模板进行渲染并返回最终需要呈现的内容，渲染器通常要知道两件事儿：（1）待渲染模板 （2）待填充数据：

```
const data = {name: 'wxj'};
const content = render(tpl, data);
// => 'hello wxj'
```

由此，我们不难得出模板引擎核心在于规则及渲染器。

## 实现一个基本的模板引擎

### 规则

有了上面的认识，我们现在可以着手开始实现一个基本的 JavaScript 模板引擎，首先我们定义规则：

- 插值：比如 `{{name}}` 直接用渲染数据进行替换，我们通过双花括号包裹。

显然，解析规则我们需要利用到正则表达式，因此我们最终的规则定义如下：

```
const rules = {
  interpolate: /{{([\s\S]+?)}}/
};

// 最终的匹配正则
const matcher = new RegExp([
  rules.interpolate.source
].join('|'), 'g');
```

## 渲染器

渲染器的核心逻辑很简单，就是遍历传入的模板字符串，当子串匹配到规则时，根据规则进行处理，最终返回一个新（渲染好的）字符串。由于我们现在只需要做插值替换，所以利用 `String.prototype.replace` 进行一下全局替换即可：

```
function render (tpl, data) {
  return tpl.replace(matcher, (match, interpolate)=>{
    return data[interpolate];
  });
}
```

如此，我们就实现了一个最简单的模板引擎，认识了其最基本的要素，但是，该系统还十分单薄，我们需要对其进行优化。

## 优化：支持模板插入代码逻辑

上面我们实现的模板只能够对模板进行数据填充，假设我们有下面内容的模板：

```
Students:
{ for(i=0;i<students.length;i++) }
{{ students[i].name }}
```

期望该模板渲染的结果为：

```
Students: wxj lcx
```

在该模板中，显然，我们支持了传入代码逻辑（一个 `for` 循环），并且设置其规则为通过 `{}` 进行包裹。为了能支持上述的模板，我们需要新建该规则服务于执行逻辑：

```
const rules = {
  // 插值
  interpolate: /{{([\s\S]+?)}}/,
  // 逻辑
  evaluate: /{([\s\S+?])}/
}

// 最终的匹配正则
const matcher = new RegExp([
  rules.interpolate.source,
  rules.evaluate.source
].join('|'), 'g');
```

渲染器现在就不单是进行字符串替换操作了，还应当支持执行传入的逻辑：

```
function render(tpl, data) {
  // 拼接字符串
  let concating = `content +=`;
  let index = 0;
  tpl.replace(matcher, (match, interpolate, evaluate, offset)
=> {
    concating += tpl.slice(index, offset);
    // 刷新拼接起点
    index = offset + match.length;
    if(evaluate) {
      // 如果是执行逻辑
      concating += `';\n${evaluate}\n content +=`;
    } else if(interpolate) {
      // 如果是插值
      concating += `'+${interpolate}+'`;
    }
    return match;
  });
  // 剩余字符拼接
  concating += tpl.slice(index);
  concating += `';\n`;
  concating = `with(obj) {\n${concating}}`;
  // 通过函数来支持逻辑执行
  const body = `let content = `'; \n${concating}; \nreturn con
tent;`;
  const renderFunc = new Function('obj', body);
  return renderFunc(data);
}
```

可以看到，通过 `new Function(arguments, body)` 来动态构建渲染函数，我们支持了向模板传入执行逻辑。值得注意的是，该构造函数将接收两个参数：

- `obj`：待渲染的数据对象，借助于 `with(expression)`，我们限定了函数体中的数据来源不会受到外部作用域的干扰。
- `body`：渲染函数的函数体，由我们动态拼接而成。

现在，测试一下：

```
const tpl = 'Students: ' +
  '{ for(i=0;i<students.length;i++) }' +
  '{{ students[i].name }} ';

data = {
  students: [{
    id: 1,
    name: 'wxj'
  }, {
    id: 2,
    name: 'lcx'
  }]
};

const content = render(tpl, data);
// content: 'Students: wxj lcx'
```

## 优化：特殊字符逃逸

在上述的模板系统中，我稍微修改了一下模板，仅只是简单的加上了换行符

`'\n'`：

```
Students: \n
{ for(i=0;i<students.length;i++) }
{{ students[i].name }}
```

期望该模板渲染的结果为：

```
Students:
wxj lcx
```

再次尝试调用我们上一步我们写好的渲染器：

```
const tpl = 'Students: \n' +
  '{ for(i=0;i<students.length;i++) }' +
  '{{ students[i].name }} ';

const content = render(tpl, data);
```

很遗憾，报错了，原因就是出在了换行符 `'\n'` 上，我们拼接字符串的时候没有用反斜杠 `\` 对转义字符进行逃逸（escape）：

```
// Error:
let body = `console.log('I love u!\n')`;
console.log(body);
// =>:
// "console.log('I love u!
// '))"

// Correct:
body = `console.log('I love u!\n')`;
console.log(body);
// => "console.log('I love u!\n')"
```

现在，我们修改渲染器对转义字符进行逃逸：

```
// 需要逃逸的字符
const escapes = {
  '"': '"',
  '\\': '\\',
  '\r': '\r',
  '\n': '\n',
  '\u2028': '\u2028', // 行分隔符
  '\u2029': '\u2029' // 行结束符
};

// 逃逸正则
const escapeRegExp = /\\"|'|\r|\n|\u2028|\u2029/g;

// 逃逸替换函数
function escape(match) {
```



```

        return '\\' + escapes[match];
    }

    function render(tpl, data) {
        // 拼接字符串
        let concating = `content +=`;
        let index = 0;
        tpl.replace(matcher, (match, interpolate, evaluate, offset)
=> {
            // 逃逸
            concating += tpl.slice(index, offset).replace(escapeRegExp, escape);
            // 刷新拼接起点
            index = offset + match.length;
            if(evaluate) {
                // 如果是执行逻辑
                concating += `';\n${evaluate}\n content +=`;
            } else if(interpolate) {
                // 如果是插值
                concating += `'+${interpolate}+'`;
            }
            return match;
        });
        // 剩余字符拼接
        concating += tpl.slice(index).replace(escapeRegExp, escape);
        ;
        concating += `';\n`;
        concating = `with(obj) {\n${concating}}`;
        // 通过函数来支持逻辑执行
        const body = `let content = ''; \n${concating}; \nreturn content;`;
        const renderFunc = new Function('obj', body);
        return renderFunc(data);
    }

```

优化：预编译

刚才我们实现的模板系统，仅只是一个简单的“替换形”模板，假设我们的模板结构不变，仅仅内容（装填数据）发生变化，就不得不重新进行渲染整个模板，不仅将造成性能上损失，也将造成复用性上的损失。接下来，我们会将 构造模板结构 和 装填数据 拆开，首先编译模板结构：

```
function template(tpl) {
  // 拼接字符串
  let concating = `content +=`;
  let index = 0;
  tpl.replace(matcher, (match, interpolate, evaluate, offset)
=> {
    // 逃逸
    concating += tpl.slice(index, offset).replace(escapeRegExp, escape);
    // 刷新拼接起点
    index = offset + match.length;
    if(evaluate) {
      // 如果是执行逻辑
      concating += `';\n${evaluate}\n content +=`;
    } else if(interpolate) {
      // 如果是插值
      concating += `'+${interpolate}+'`;
    }
  });
  return match;
});
// 剩余字符拼接
concating += tpl.slice(index).replace(escapeRegExp, escape);
;
concating += `';\n`;
concating = `with(obj) {\n${concating}}`;
// 通过函数来支持逻辑执行
const body = `let content = `'; \n${concating}; \nreturn content;`;
const renderFunc = new Function('obj', body);
return function(data) {
  return renderFunc(data);
}
}
```

现在，我们通过函数 `template(tpl)` 来编译传入的模板 `tpl`，该函数会返回一个新的函数，由这个函数完成接收数据以后的最终渲染：

```
const tpl = 'hello {{name}}';  
// 编译模板  
const compiled = template(tpl);  
const content1 = compiled({name: 'wxj'}); // => 'hello wxj'  
const content2 = compiled({name: 'zxy'}); // => 'hello zxy';
```

可以看到，模板只需一次编译，就能再多地多个时刻进行复用。

然而这个模板引擎也并不完美，假如我们的模板中包含了 `this`：

```
tpl = '{console.log(`name`, this.name);}';  
  
// 模板管理器  
const tplManager = {  
  name: 'manager',  
  log: template(tpl)  
};  
// => 'name '  
tplManager.log({});
```

并没有输出我们期望的字符串 `'name: manager'`，所以我们还需要略微修改 `template` 函数，考虑模板运行时的上下文问题：

```
function template(tpl) {  
  // ...  
  return function(data) {  
    return renderFunc.call(this, data);  
  }  
}
```

间接调用函数时，通过 `Function.prototype.call()` 来保证上下文的正确绑定也是一种好习惯。

## \_\_template

`_.template = function (text, settings, oldSettings)`：根据传入的文本 `text` 及配置 `settings`，生成模板。

事实上，在前文的描述当中，我们就大致实现了 `underscore` 中的 `_.template()` 方法，之所以不直接上该函数的源码，是因为模板引擎并非一个简单的函数，读者若能够体会到从认识模板，到撰写一个基本模板引擎，再到优化模板引擎这个过程，将受益更多，也不会烦恼并且纠结于源码中一些细节的来源：

```
// 默认情况下，undersocre使用[ERB风格的模板](http://www.stuartellis.eu/articles/erb/)
// 但是也可以手动配置
_.templateSettings = {
  // 执行体通过<% %>包裹
  evaluate: /<%([\s\S]+?)%>/g,
  // 插入立即数通过<%= %>包裹
  interpolate: /<%=([\s\S]+?)%>/g,
  // 逃逸通过<%- %>包裹
  escape: /<%-([\s\S]+?)%>/g
};

// 如果不想使用interpolation、evaluation、escaping正则，
// 必须使用一个noMatch正则来保证不匹配的情况
var noMatch = /(.)^/;

// 定义需要逃逸的字符，以便他们之后能够被运用到模板中的字符串字面量中
var escapes = {
  "'": "'",
  '\\': '\\',
  '\r': 'r',
  '\n': 'n',
  '\u2028': '\u2028', // 行分隔符
  '\u2029': '\u2029' // 行结束符
};

// 逃逸正则
var escapeRegExp = /\\"|'|\r|\n|\u2028|\u2029/g;

/**
 * 转义字符
 * @param {string} match
```

```
*/
var escapeChar = function (match) {
    return '\\' + escapes[match];
};

// JavaScript micro-templating, similar to John Resig's implemen
tation.
// Underscore templating handles arbitrary delimiters, preserves
whitespace,
// and correctly escapes quotes within interpolated code.
// NB: `oldSettings` only exists for backwards compatibility.
/**
 * underscore实现的一个js微模板引擎
 * @param {string} text
 * @param {object} settings 模板配置
 * @param {object} oldSettings 该参数用以向后兼容
 */
_.template = function (text, settings, oldSettings) {
    // 校正模板配置
    if (!settings && oldSettings) settings = oldSettings;
    // 获得最终的模板配置
    settings = _.defaults({}, settings, _.templateSettings);

    // Combine delimiters into one regular expression via altern
ation.
    // 获得最终的匹配正则
    // /<%-([\s\S]+?)%>|<%=([\s\S]+?)%>|<%([\s\S]+?)%>|$/g

    var matcher = RegExp([
        (settings.escape || noMatch).source, // /<%([\s\S]+?)%>/
        g.source === '<%([\s\S]+?)%>'
        (settings.interpolate || noMatch).source,
        (settings.evaluate || noMatch).source
    ].join('|') + '|$', 'g');

    // Compile the template source, escaping string literals app
ropriately.
    //
    var index = 0;
    // source用来保存最终的函数执行体
```

```
var source = "__p+='";
// 正则替换模板内容，逐个匹配，逐个替换
text.replace(matcher, function (match, escape, interpolate,
evaluate, offset) {
    // offset 匹配到的子字符串在原字符串中的偏移量。
    // (比如，如果原字符串是"abcd"，匹配到的子字符串是"bc"，那么这个
    参数将是1)

    // 开始拼接字符串。进行字符逃逸
    source += text.slice(index, offset).replace(escapeRegExp
, escapeChar);
    // 从下一个匹配位置开始
    index = offset + match.length;

    if (escape) {
        source += "'+\n((__t=(" + escape + "))==null?'':_.es
cape(__t))+\n'";
    } else if (interpolate) {
        source += "'+\n((__t=(" + interpolate + "))==null?'
':__t))+\n'";
    } else if (evaluate) {
        source += "';\n" + evaluate + "\n__p+='";
    }

    // Adobe VMs need the match returned to produce the corr
    ect offset.
    return match;
});

//
source += "';\n";
// 如果没有在settings中声明变量，则用with限定作用域
if (!settings.variable) source = 'with(obj||{}){\n' + source
+ '}\n';

source = "var __t,__p='',__j=Array.prototype.join," +
"print=function(){__p+=__j.call(arguments,'');};\n" +
source + 'return __p;\n';

var render;
```

```
try {
  // 动态创建渲染函数
  render = new Function(settings.variable || 'obj', '_', source);
} catch (e) {
  e.source = source;
  throw e;
}

// 最终返回一个模板函数，通过给模板传递数据
// 最后通过render来渲染结果
var template = function (data) {
  return render.call(this, data, _);
};

// 保留编译后的源码
var argument = settings.variable || 'obj';
template.source = 'function(' + argument + '){\n' + source +
  '}';

return template;
};
```

相比较上文写的模板引擎，underscore 提供的模板引擎还有如下特色：

- 采用了 **ERB 风格** 的模板，同时，也支持自定义模板风格

```
_.templateSettings = {
  interpolate: /\{\{(.+?)\}\}/g
};
var template = _.template("Hello {{ name }}!");
template({name: "Mustache"});
// => "Hello Mustache!"
```

- 支持对 HTML 内容进行逃逸：

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

- 支持传递一个变量标记，这样，模板编译的时候，将不会用到 `with` 来限定作用域，从而显著提升模板性能：

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

- 在返回的模板函数中，提供了一个 `source` 属性，来获得编译后的模板函数源码，从而支持服务端使用 JST（JavaScript Template）。比如我们在服务端的模板文件使用了 JST 如下：

```
window.JST = {};
JST.contact = <% _.template("<div class='contact'><%= name %> ..
..").source %>
```

后端模板引擎通过 `<% %>` 将源码打印到模板内，获得缓存的模板文件，假设叫 `contact.js`：

```
window.JST = {};
JST.contact = function(obj) {
  // ....
}
```

当我们前端请求 `contact.js`（字符串）后，就能使用该模板，而不用将繁重的模板编译工作放在前端进行：

```
<script src='http://xxx/contact.js'></script>
<script>
  const html = JST.contact({name: 'wxj'})
</script>
```



## 多说一句：with

上文中我们提到，省略 `with`，将显著提高模板性能，我们可以测试一下 `with` 的性能：

```
const student = {
  name: 'wxj'
};

// 设置一个大循环，访问`student`的`name`属性
const n = 10000000;
let stuName;

console.time('Not Using with');
for(let i=0;i<n;i++) {
  stuName = student.name;
}
console.timeEnd('Not Using with');

console.time('Using with');
for(let i=0;i<n;i++) {
  with(student) {
    stuName = name;
  }
}
console.timeEnd('Using with');

// => Not Using with: 2262.372ms
// => Using with: 47.398ms
```

可以看到，是否使用 `with`，性能差距确实非常大。造成 `with` 性能低下的原因就在于，当面临如下的代码段时：

```
with(obj) {
  console.log(name);
}
```

为了确定变量 `name` 的值，JavaScript 引擎需要先查找 `with` 语句包裹的变量，而后是局部变量，最后是全局变量。这样，如下的代码段多了一层查找开销：

```
console.log(obj.name);
```

## 参考资料

- [JavaScript Micro-Templating](#)
- [JST templates](#)
- [Javascript Performance Boosting Tips from Nicholas Zakas](#)

## 内容拾遗

在收尾阶段，我们还会介绍 underscore 提供的面向对象风格（**OOP Style**），链式调用（**Chain**）等内容。

## 面向对象风格的支持

我们知道，在函数式编程范式中，函数是拿来就用的，即便对象，也只是函数的一个参数，被函数支配：

```
const arr = [1, 2, 3];
_.map(arr, n => n*2);
```

而在面向对象的世界中，函数往往是隶属于某个对象的 成员方法：

```
const arr = [1, 2, 3];
arr.map(n => n*2);
```

### `_()`

虽然 `underscore` 是推崇函数式编程（FP）的，但是也提供了以面向对象风格的来进行函数调用，仅需要通过 `_()` 来包裹下对象即可：

```
_([1, 2, 3]).map(n => n*2);
```

在 [underscore 基础篇 -- 结构](#) 一节中，我们介绍过 `_` 是一个函数对象，当我们进行如下调用时：

```
_([1, 2, 3]);
```

会创建一个新的 `underscore` 对象（从而能够调用 `underscore` 提供的方法），并且记录被包裹的对象为 `obj`：

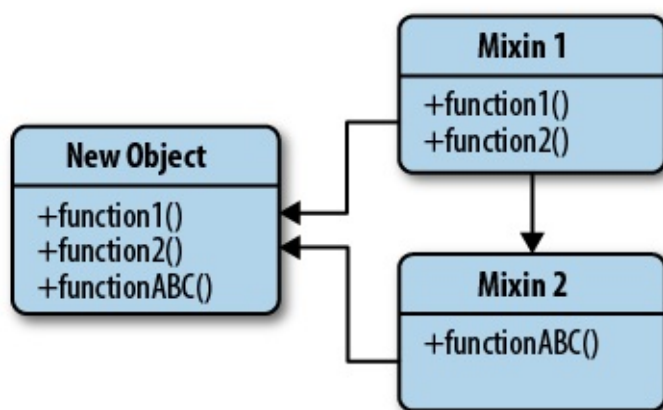
```
var _ = function (obj) {
  if (obj instanceof _) return obj;
  if (!(this instanceof _)) return new _(obj);
  this._wrapped = obj;
};
```



## mixin

mixin（混入）模式是增加代码复用度的一个广泛使用的设计模式。什么是 mixin 模式，简言之，就是向一个对象混入一系列方法，使之具备更强大的能力，这一些列方法我们又会包裹在一个称之为 mixin 的对象中，这样，其他对象也能够给通过该 mixin 进行扩展：

Mixins



相较于继承，我们发现，mixin 的对象组织更加松散，因而也就不用面对如何定义类，类的抽象程度到什么地方诸如此类的在继承中出现的问题。反正想要功能，ok，直接混入一下就好。

图片摘自 [Learning JavaScript Design Patterns by Addy Osmani](#)

## mixin 模式的一般实现

mixin 的一般实现是为 mixin 先创建一个对象。下例就是一个模态框的 mixin：

```
const modalMixin = {
  open() {
    console.log(`opening`);
  },
  close() {
    console.log('closing')
  }
};
```

然后创建一个 `extend` 方法，该方法用来拷贝 `mixin` 对象中的方法到一般对象中，借此增强原对象的功能：

```
/**
 * extend
 * @param {Object} obj
 * @param {Object} mixin
 */
function extend(obj, mixin) {
    Object.keys(mixin).map((func) => {
        obj[func] = mixin[func]
    });
}

// 一个普通的对话框
function Dialog(title) {
    this.title = title;
}

Dialog.prototype.confirm = function () {
    console.log('confirm?');
}

// 通过mixin扩展该对话框的功能，使之能够关闭和打开
extend(Dialog.prototype, modalMixin);
let dialog = new Dialog('News: xxxx');
dialog.open(); // => 'opening'
```

## mixin 模式的函数实现

在 JavaScript 这类支持多范式编程的语言中，还会直接将 `mixin` 设计为一个函数，从而跳过中间过程 `extend`：

```
function modalMixin() {  
  this.open = function () {  
    console.log('opening')  
  };  
  this.close = function () {  
    console.log('closing');  
  };  
  // 返回对象，便于进行链式的混入  
  return this;  
}
```

通过委托（delegate）模式的 `Function.prototype.call` 或者 `Function.prototype.apply` 进行对象和 mixin 的绑定

```
modalMixin.call(Dialog.prototype);  
dialog.open(); // => 'opening'
```

再通过闭包做下缓存，避免每次混入时都要重新创建函数：

```
const modalMixin = (function () {  
  const open = function () {  
    console.log('opening')  
  };  
  const close = function () {  
    console.log('closing');  
  };  
  return function() {  
    this.open = open;  
    this.close = close;  
    return this;  
  };  
})();
```

## `_.mixin()`

`_.mixin(obj)`：为 underscore 对象混入 `obj` 具有的功能。



源码：

```
_.mixin = function (obj) {
  _.each(_.functions(obj), function (name) {
    var func = _[name] = obj[name];
    _.prototype[name] = function () {
      var args = [this._wrapped];
      push.apply(args, arguments);
      return chainResult(this, func.apply(_, args));
    };
  });
  return _;
};
```

underscore 中的 `mixin(obj)` 将会用传入的 `obj` 来扩充 `_` 原型上的功能，并且也是用的委托模式的 `apply`，另外通过 `chainResult` 考虑了扩充的函数能够进行链式调用（`chainResult` 将在后文进行介绍）。源码中的这个代码片还值得看看：

```
_.mixin = function(obj) {
  // ...
  var args = [this._wrapped];
  push.apply(args, arguments);
  return chainResult(this, func.apply(_, args));
  // ...
}
```

该代码片考虑到了当我们想要以面向对象的风格调用 underscore 函数时的情况：

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

这时候，`capitalize` 的 `this` 会被绑定到 `_` 对象上，同时，函数的首个参数将会被设置为 `'fabio'`，从而相当于这样调用：

```
_.capitalize('fabio');  
// => "Fabio"
```

最后要提的是，`underscore` 默认通过 `_.mixin(_)` 这一语句混入了所有方法到包裹后的对象上，以便包裹后的对象能够调用 `underscore` 上的方法：

```
_  
  .mixin({  
    map: function(array) {  
      console.log('map is broken!');  
    }  
  });  
  
_  
  .mixin({  
    map: function(array) {  
      console.log('map is broken!');  
    }  
  });  
  
_([1, 2, 3]).map(n => n*2);  
// => map is broken!
```

## 不要滥用 `mixin`

`mixin` 不总是那么美好的，从其实现原理我们就可以看到，`mixin` 的侵入性很强，他很有可能污染原来的原型，从而引起误会：

```
_.mixin({  
  map: function(array) {  
    console.log('map is broken!');  
  }  
});  
  
_([1, 2, 3]).map(n => n*2);  
// => map is broken!
```

## 参考资料

- [A fresh look at JavaScript Mixins](#)
- [Learning JavaScript Design Patterns - Mixin](#)

## 链式调用

### jquery 的链式调用

相信大家对于链式调用不会感到陌生，我们在 jquery 经常使用：

```
$('.div').css('color', 'red').show();
```

想要实现链式调用，通常我们会在支持链式调用的函数中返回对象本身：

```
let car = {
  run(seconds) {
    console.log('begin run')
    for(let i=0;i<seconds;i++) {}
    return this;
  },
  stop() {
    console.log('stopped!');
  }
};

car.run(5).stop();
// => begin run
// => stopped
```

但是，这样做并不优雅，这需要我们手动地在函数中添加 `return this` 语句。更好的做法是我们创建一个通用函数，他能为指定的对象方法增加链式调用机制：

```
/**
 * 为指定的对象方法添加链式调用功能
 * @param {Object} obj
 * @param {Array} functions
 */
function chained(obj, functions) {
  functions.forEach((funcName) => {
    const func = obj[funcName];
    // 用支持链式调用的方法覆盖原有方法
    obj[funcName] = function () {
      func.apply(this, arguments);
      return this;
    };
  });
}

let car = {
  run(seconds) {
    console.log('begin run')
    for(let i=0;i<seconds;i++) {}
  },
  stop() {
    console.log('stopped!');
  }
};

chained(car, ['run']);
car.run(5).stop();
```

## **`_.chain()`**

`_.chain(obj)` : 为 `underscore` 对象的方法增加链式调用能力。

下面这段代码将找出 `stooges` 中的最年轻的并且输出其信息：

```
const stooges = [{name: 'curly', age: 25}, {name: 'moe', age: 21}, {name: 'larry', age: 23}];
const youngest = _.chain(stooges)
  .sortBy(stooge => stooge.age)
  .map(stooge => stooge.name + ' is ' + stooge.age)
  .first()
  .value();
// => "moe is 21"
```

`_.chain` 源码如下：

```
_.chain = function (obj) {
  // 获得一个经underscore包裹后的实例
  var instance = _(obj);
  // 标识当前实例支持链式调用
  instance._chain = true;
  return instance;
};
```

当我们期望对 `obj` 接下来的行为链化的话，会创建一个包裹了 `obj` 的 `underscore` 实例对象，并标识该实例需要进行链式调用，最后返回该实例以继续链式调用。

并且，`underscore` 还提供了一个帮助函数 `chainResult`，该函数将会判断方法调用结果，如果该方法的调用者被标识了需要链化，则链化当前的方法执行结果：

```
var chainResult = function (instance, obj) {
  return instance._chain ? _(obj).chain() : obj;
};

_.each(['concat', 'join', 'slice'], function (name) {
  var method = ArrayProto[name];
  _.prototype[name] = function () {
    return chainResult(this, method.apply(this._wrapped, arguments));
  };
});
```

当我们想要获得链式调用的结果时，可以看到，需要通过 `_.prototype.value` 方法获取，该方法返回了被包裹的对象：

```
_.prototype.value = function () {  
    return this._wrapped;  
};
```