lodash vs ramda

http://lodashjs.com http://ramda.cn

https://github.com/lodash/lodash https://github.com/ramda/ramda

“函数式编程以数据流动为导向，是函数装配的艺术。”

—-Silent Reverie

对比

	特点
lodash	是一个具有一致接口、模块化、高性能等特性的 JavaScript 工具库。
ramda	专门为函数式编程风格而设计，更容易创建函数式 pipeline(管道)、且从不改变用户已有数据。

对比

	函数库个数	函数用法
lodash	244	1.先数据参数,后函数参数 2.fp模块自动柯里化, 非fp模块无柯里化 3. 不支持占位符
ramda	245	1. 先函数参数,后数据参数 2. 自动柯里化 3. 支持占位符R._

对比

	使用

lodash
模块化
	npm install lodash
	var _ = require(‘lodash'); //full模块
	var _ = require(‘lodash/core'); //core模块(https://github.com/lodash/lodash/wiki/build-differences)
	var fp = require(‘lodash/fp'); //fp模块
	var array = require(‘lodash/array'); //array模块
	var object = require(‘lodash/fp/object'); //object模块
	var curryN = require(‘lodash/fp/curryN'); //fp模块下的柯里化函数

ramda	npm install ramda const R = require('ramda');

对比

	部分构建库
lodash	npm i -g lodash-cli lodash include=each,filter,map
ramda	npm run --silent partial-build compose reduce filter > dist/ramda.custom.js

示例

TODO List任务列表

```
var tasks = [
    {username: 'Michael', title: 'Curry stray functions', dueDate: '2014-05-06',
               complete: true, effort: 'low', priority: 'low'},
    {username: 'Scott', title: 'Add `fork` function', dueDate: '2014-05-14',
               complete: true, effort: 'low', priority: 'low'},
    {username: 'Michael', title: 'Write intro doc', dueDate: '2014-05-16',
               complete: true, effort: 'low', priority: 'low'},
    {username: 'Michael', title: 'Add modulo function', dueDate: '2014-05-17',
               complete: false, effort: 'low', priority: 'low'},
    {username: 'Michael', title: 'Separating generators', dueDate: '2014-05-24',
               complete: false, effort: 'medium', priority: 'medium'},
    {username: 'Scott', title: 'Fold algebra branch back in', dueDate: '2014-06-01',
               complete: false, effort: 'low', priority: 'low'},
    {username: 'Scott', title: 'Fix `and`/`or`/`not`', dueDate: '2014-06-05',
               complete: false, effort: 'low', priority: 'low'},
    {username: 'Michael', title: 'Types infrastucture', dueDate: '2014-06-06',
               complete: false, effort: 'medium', priority: 'high'},
    {username: 'Scott', title: 'Add `mapObj`', dueDate: '2014-06-09',
               complete: false, effort: 'low', priority: 'medium'},
    {username: 'Scott', title: 'Write using doc', dueDate: '2014-06-11',
               complete: false, effort: 'medium', priority: 'high'},
    {username: 'Michael', title: 'Finish algebraic types', dueDate: '2014-06-15',
               complete: false, effort: 'high', priority: 'high'},
    {username: 'Scott', title: 'Determine versioning scheme', dueDate: '2014-06-15',
                complete: false, effort: 'low', priority: 'medium'},
    {username: 'Michael', title: 'Integrate types with main code', dueDate: '2014-06-22',
               complete: false, effort: 'medium', priority: 'high'},
    {username: 'Richard', title: 'API documentation', dueDate: '2014-06-22',
               complete: false, effort: 'high', priority: 'medium'},
    {username: 'Scott', title: 'Complete build system', dueDate: '2014-06-22',
               complete: false, effort: 'medium', priority: 'high'},
    {username: 'Richard', title: 'Overview documentation', dueDate: '2014-06-25',
               complete: false, effort: 'medium', priority: 'high'}
];
```

示例

需求：

1. 输入username, 过滤出该用户未完成的任务列表（complete为false的数据），按date降序排列, 取前5条数据，数据包含“title”、“dueDate”
2. 过滤出complete为false的数据，根据username分组，按date降序排列，每组取前5条数据，数据包含“title”、“dueDate”

```
// Ramda V0.25.0
var incomplete = R.filter(R.whereEq({complete: false}));
var sortByDate = R.sortBy(R.prop('dueDate'));
var sortByDateDescend = R.compose(R.reverse, sortByDate);//从右往左执行
var importantFields = R.project(['title', 'dueDate']);//模拟 select 语句
var groupByUser = R.groupBy(R.prop('username'));//分组
var activeByUser = R.compose(groupByUser, incomplete);
var gloss = R.compose(importantFields, R.take(5), sortByDateDescend);
var topData = R.compose(gloss, incomplete);
var topDataAllUsers = R.compose(R.map(gloss), activeByUser);
var byUser = R.useWith(R.filter, [R.propEq("username"), R.identity]);//第1个参数传给R.propEq("username"), 第2个传给R.identity原值返回, 最后处理好后传给R.filter
console.log("Gloss for Scott:");
console.log(topData(byUser("Scott", tasks)));
console.log("====================");
console.log("Gloss for everyone:");
console.log(topDataAllUsers(tasks));
```

示例

使用lodash

```
// Lodash v4.17.4
var fp = require('./last/lodash/fp');
console.log("==========Lodash====================");
var incomplete = fp.filter({complete: false});
var sortByDate = fp.sortBy('dueDate');
var sortByDateDescend = fp.compose(fp.reverse, sortByDate);
var importantFields = fp.map(fp.pick(['title', 'dueDate']));
var groupByUser = fp.groupBy(fp.get('username'));
var activeByUser = fp.compose(groupByUser, incomplete);
var gloss = fp.compose(importantFields, fp.take(5), sortByDateDescend);
var topData = fp.compose(gloss, incomplete);
var topDataAllUsers = fp.compose(fp.mapValues(gloss), activeByUser);
var byUser = fp.curry(function(username, tasks){
   return fp.filter({"username": username})(tasks);
})
console.log("Gloss for Scott:");
console.log(topData(byUser("Scott", tasks)));
console.log("====================");
console.log("Gloss for everyone:");
console.log(topDataAllUsers(tasks));
```

从两者对同一需求实现上看：lodash有时候用法更加简洁如filter和sortBy， 但ramda也封装了更多方便的函数如project，useWith，我觉得两种用起来不分伯仲。

示例

```
 ethan@ethandeMacBook-Pro ~/Desktop/tmp/test node todolist.js
==========Ramda====================
Gloss for Scott:
[ { title: 'Complete build system', dueDate: '2014-06-22' },
  { title: 'Determine versioning scheme', dueDate: '2014-06-15' },
  { title: 'Write using doc', dueDate: '2014-06-11' },
  { title: 'Add `mapObj`', dueDate: '2014-06-09' },
  { title: 'Fix `and`/`or`/`not`', dueDate: '2014-06-05' } ]
====================
Gloss for everyone:
{ Michael:
   [ { title: 'Integrate types with main code',
       dueDate: '2014-06-22' },
     { title: 'Finish algebraic types', dueDate: '2014-06-15' },
     { title: 'Types infrastucture', dueDate: '2014-06-06' },
     { title: 'Separating generators', dueDate: '2014-05-24' },
     { title: 'Add modulo function', dueDate: '2014-05-17' } ],
  Scott:
   [ { title: 'Complete build system', dueDate: '2014-06-22' },
     { title: 'Determine versioning scheme', dueDate: '2014-06-15' },
     { title: 'Write using doc', dueDate: '2014-06-11' },
     { title: 'Add `mapObj`', dueDate: '2014-06-09' },
     { title: 'Fix `and`/`or`/`not`', dueDate: '2014-06-05' } ],
  Richard:
   [ { title: 'Overview documentation', dueDate: '2014-06-25' },
     { title: 'API documentation', dueDate: '2014-06-22' } ] }
=========Lodash====================
Gloss for Scott:
[ { title: 'Complete build system', dueDate: '2014-06-22' },
  { title: 'Determine versioning scheme', dueDate: '2014-06-15' },
  { title: 'Write using doc', dueDate: '2014-06-11' },
  { title: 'Add `mapObj`', dueDate: '2014-06-09' },
  { title: 'Fix `and`/`or`/`not`', dueDate: '2014-06-05' } ]
====================
Gloss for everyone:
{ Michael:
   [ { title: 'Integrate types with main code',
       dueDate: '2014-06-22' },
     { title: 'Finish algebraic types', dueDate: '2014-06-15' },
     { title: 'Types infrastucture', dueDate: '2014-06-06' },
     { title: 'Separating generators', dueDate: '2014-05-24' },
     { title: 'Add modulo function', dueDate: '2014-05-17' } ],
  Scott:
   [ { title: 'Complete build system', dueDate: '2014-06-22' },
     { title: 'Determine versioning scheme', dueDate: '2014-06-15' },
     { title: 'Write using doc', dueDate: '2014-06-11' },
     { title: 'Add `mapObj`', dueDate: '2014-06-09' },
     { title: 'Fix `and`/`or`/`not`', dueDate: '2014-06-05' } ],
  Richard:
   [ { title: 'Overview documentation', dueDate: '2014-06-25' },
     { title: 'API documentation', dueDate: '2014-06-22' } ] }
```

代码性能上呢?

代码性能测试工具

Benchmark.js

```
var suite = new Benchmark.Suite;

// add tests
suite.add('RegExp#test', function() {
  /o/.test('Hello World!');
})
.add('String#indexOf', function() {
  'Hello World!'.indexOf('o') > -1;
})
// add listeners
.on('cycle', function(event) {
  console.log(String(event.target));
})
.on('complete', function() {
  console.log('Fastest is ' + this.filter('fastest').map('name'));
})
// run async
.run({ 'async': true });

// logs:
// => RegExp#test x 4,161,532 +-0.99% (59 cycles)
// => String#indexOf x 6,139,623 +-1.00% (131 cycles)
// => Fastest is String#indexOf
```

结果最快的就是String对象的indexOf方法，其中，Ops/sec 测试结果以每秒钟执行测试代码的次数（Ops/sec）显示，这个数值越大越好。除了这个结果外，同时会显示测试过程中的统计误差，以及相对最好的慢了多少（%）

参考:https://segmentfault.com/a/1190000003486676　　https://github.com/bestiejs/benchmark.js

代码性能测试工具

https://jsperf.com/

jsPerf 提供了一个简便的方式来创建和共享测试用例，并可以比较不同JavaScript代码段的性能。jsPerf也是基于Benchmark来运行的。

https://jsperf.com/thor-indexof-vs-for

Test		Ops/sec

For loop without break	`for(var i = 0, len = array1.length; i < len; i++)`
	`{`
	`  var value = array1[i];`
	`  var result = false;`
	
	`  for(var j = 0, len = array2.length; j < len; j++)`
	`  {`
	`  if(value === array2[j])`
	`  {`
	`  result = true;`
	`  }`
	`  }`
	`}`
		51,920,434
		±1.06%
		5% slower

Array.indexOf()	`for(var i = 0, len = array1.length; i < len; i++)`
	`{`
	`  var value = array1[i];`
	`  var result = (array2.indexOf(value) !== -1);`
	`}`
		2,888,327
		±1.21%
		95%
		slower

For loop	`for(var i = 0, len = array1.length; i < len; i++)`
	`{`
	`  var value = array1[i];`
	`  var result = false;`
	
	`  for(var j = 0, len = array2.length; j < len; j++)`
	`  {`
	`  if(value === array2[j])`
	`  {`
	`  result = true;`
	`  break;`
	`  }`
	`  }`
	`}`
		54,837,867
		±1.18%
		fastest

参考:https://segmentfault.com/a/1190000003486676

示例及其性能对比

数组[{counter:0},{counter:1},{counter:2}…{counter:9999}]
从中取出counter，过滤出其中的奇数，然后对他们进行平方，
最后过滤出两位数.

```
//生成data
var data = _.range(10000).map(function(i) {
    return {
      counter: i
    }
});

var isOdd = function(num) { //是否是奇数
    return num % 2 === 1;
};
var square = function(num) { //平方
    return num * num;
};
var isLess3 = function(num) {  //2位数
    return num.toString().length < 3;
};
```

示例及其性能对比

```
var lodash = function(data) {
    return _.filter(_.map(_.filter(_.pluck(data, 'counter'), isOdd), square), isLess3);
};

var ramda1 = function(data) {
    return R.filter(isLess3,R.map(square,R.filter(isOdd, R.pluck('counter', data))));
};

// pipe方式实现.左往右执行函数组合
var ramda2 = R.pipe(R.pluck('counter'),R.filter(isOdd),R.map(square),R.filter(isLess3));

var native1 = function(data) {
    return data.map(function(value) {
        return value.counter;
      }).filter(function(value) {
        return value != null;
      }).filter(isOdd).map(square).filter(isLess3);
};

var native2 = function(data) {
    var result = [];
    var length = data.length;
    for (var i=0; i<data.length; i++) {
      var value = data[i].counter;
      var squared;
      if (isOdd(value)) {
        squared = square(value);
        if (isLess3(squared)) {
          result.push(squared);
        }
      }
    }
    return result;
};
```

示例及其性能对比

```
suite = new Benchmark.Suite;
// 添加测试
suite
.add('lodash', function() {
  lodash(data);
})
.add('ramda1', function() {
  ramda1(data);
})
.add('ramda2', function () {
  ramda2(data);
})
.add('native1', function () {
  native1(data);
})
.add('native2', function () {
  native2(data);
})
// 每个测试跑完后，输出信息
.on('cycle', function(event) {
  console.log(String(event.target));
})
.on('complete', function() {
  console.log('Fastest is ' + this.filter('fastest').map('name'));
})
// 这个选项与它的时间计算有关，默认勾上。
.run({ 'async': true });
```

示例对比结果

lodash:1,9,25,49,81
ramda1:1,9,25,49,81
ramda2:1,9,25,49,81
native1:1,9,25,49,81
native2:1,9,25,49,81
lodash x 1,211 ops/sec ±1.27% (86 runs sampled)
ramda1 x 312 ops/sec ±1.20% (84 runs sampled)
ramda2 x 308 ops/sec ±1.50% (83 runs sampled)
native1 x 843 ops/sec ±1.06% (83 runs sampled)
native2 x 2,390 ops/sec ±1.14% (88 runs sampled)
Fastest is native2

示例对比结果

https://jsperf.com/lodash-ramda/1

Testing in Chrome 63.0.3239 / Mac OS X 10.13.2

Test		Ops/sec
lodash	`lodash(data);`	919 ±3.92% 24% slower
ramda1	`ramda1(data);`	880 ±3.89% 28% slower
ramda2	`ramda2(data);`	867 ±4.07% 29% slower
native2	`native2(data);`	1,223 ±4.53% fastest
native1	`native1(data);`	730 ±3.94% 40% slower

在这个例子上可以执行效率: “natvie”>lodash>ramda

Why Lodash?

Lodash makes JavaScript easier by taking the hassle out of working with arrays, numbers, objects, strings, etc.
Lodash's modular methods are great for:

- Iterating arrays, objects, & strings
- Manipulating & testing values
- Creating composite functions

—https://lodash.com/

模块化

迭代

操作&测试对象值

创建组合函数

Why Lodash?

Lodash 为 JavaScript 语言增添了诸多特性，程序员可以用它轻松写出语义精确、执行高效的代码。此外，Lodash 已经完全模块化了。虽然它的某些特性最终会被淘汰，但仍有许多功能是值得我们继续使用的。同时，Lodash 这样的类库也在不断推动 JavaScript 语言本身的发展。
—-为什么不用 ES6 完全替换 Lodash(http://blog.csdn.net/zjerryj/article/details/76864598)

“Underscore和Lodash这两个类库为我们提供了一系列相当不错的跟函数式编程相关的方法。Underscore以API实现简洁著称。Lodash作为Underscore的后继者，除了对Underscore现有API功能使用上进行扩充外，更是添加了不少令人难忘的API，在性能上也更为出彩，而且还能根据需要构建自己的子集方法。”
—-由Underscore与Lodash的差异引发的思考(http://ju.outofmemory.cn/entry/106512)

“其实 lodash 或 ramda 之类的库，并不只是提供了一些数据处理的基础方法，同时还方便了我们以 Functional Programming 的思路来进行数据处理。对于习惯使用 ramda 的朋友来说，肯定已经很了解 Functional Programming 所带来的好处，但是很多使用 lodash 的朋友还是只把它当做一个 util 来使用。”
—-在 ES6 大行其道的今天，还有必要使用 lodash 之类的库吗？(https://www.zhihu.com/question/36942520/answer/113063307)

Why Ramda?

对于那些不熟悉函数式编程的人来说，Ramda 似乎没有什么帮助。Ramda 中的大部分功能在类似于 Underscore 和 Lodash 库中都已经有了。

这些人是对的。如果你希望一直使用之前一直在用的命令式和面向对象的方式进行编程，那么 Ramda 可能没有太多价值。

然而，它提供了一种不同的编码风格，这种编程风格天然适合于函数式编程语言：Ramda 可以让 "通过函数式组合构建复杂的逻辑" 变得更简单。注意，任何包含 compose 函数的库都可以进行函数式组合；这样做真正的意义是："make it simple(让编程变得简单)"。

参考:https://adispring.coding.me/2017/06/25/Why-Ramda/ http://ju.outofmemory.cn/entry/106512