第一章 前端数据处理

珠峰算法I课程

场景介绍

- 数学处理
  - 数据可视化
  - 游戏
  - 动画
- 迭代处理
  - SQL的等效方法（服务端返回数据）
  - 视图数据处理

课程目录

- 数学函数Math
  - 分页操作
  - 数组最值
  - 随机数生成
  - 素数判断
- 数组和链式操作
  - JS原生的数组操作
  - 扩展库
  - 算法举例
    - 括号匹配
    - 子数组整除
  - SQL对应的数据处理
- 迭代器和Generator
- Ramda介绍
  - zip
  - converge
  - innerJoin
  - flatten
  - intersperse
- 总结
  - 遍历方法的总结
  - 链式操作的优缺点
  - 什么时候应该拷贝数据?
  - 为什么需要Ramda这些库?

重要的Math函数

Math.abs	求绝对值
Math.ceil	向上取整
Math.floor	向下取整
Math.max	求最大值
Math.min	求最小值
Math.random	0-1之间的随机数
Math.sqrt	平方根
Math.sign	求数值的符号
Math.pow	求幂

例001-分页计算

- 在一个分页表格中，给定每页显示条数(pageSize)和元素的序号(index)，求页码

```
const pageNo = Math.ceil( (index + 1) /
pageSize )
    pageSize = 10

    0 -> 1
    9 -> 1
    10 -> 2
    11 -> 2
```

例002-数组最值

```
const A = [1,2,3,5,6]
const max = Math.max(...A)
// 等价于 ： const max = Math.max.apply(null, A) , 早期写法
// 等价于 ： const max = Math.max(1,2,3,5,6)

//同理
const min = Math.min(...A)
```

例003-生成20-30之间的随机整数

```
Math.round( 20 + Math.random() * 10 )
```

例004-判断一个数是否是素数

```
1   function is_prime(n) {
2     if (n <= 1) {return false}
3     const N = Math.floor(Math.sqrt(n))
4     let is_prime = true
5     for(let i = 2; i <= N; i++) {
6       if( n % i === 0) {
7         is_prime = false
8         break
9       }
1     }
0     return is_prime
1   }
1
1
1
```

$$N^2 \leq n \leq (N+1)^2$$

数组相关操作

Array.length	长度	[1,2,3].length -- 3
indexOf	获取元素的序号	[1,2,3].indexOf(2) – 1
Array.isArray()	判断是不是数组	Array.isArray([]) -- true
forEach	遍历	和for循环类似，不能break
push/pop/shift/unshift	入栈、出栈、入队、出队	见下文
map	映射-1对1	[1,2,3].map(x=>x*2)---- [4,5,6]
reduce	聚合-多对1	[1,2,3].reduce((x,y)=>x+y) --- 6
filter	筛选	[1,2,3].filter(x=>x>2) --- [3]
Array.from	创建数组	见下文
concat	合并数组（或元素）	[1,2].concat([3,4]) -- [1,2,3,4]
slice	剪切	见下文
splice	删除/替换/插入	见下文

reduceRight	从右向左reduce	
sort	排序	见下文
every	所有元素符合某个条件	[1,2,3].every(x=>x>0) -- true

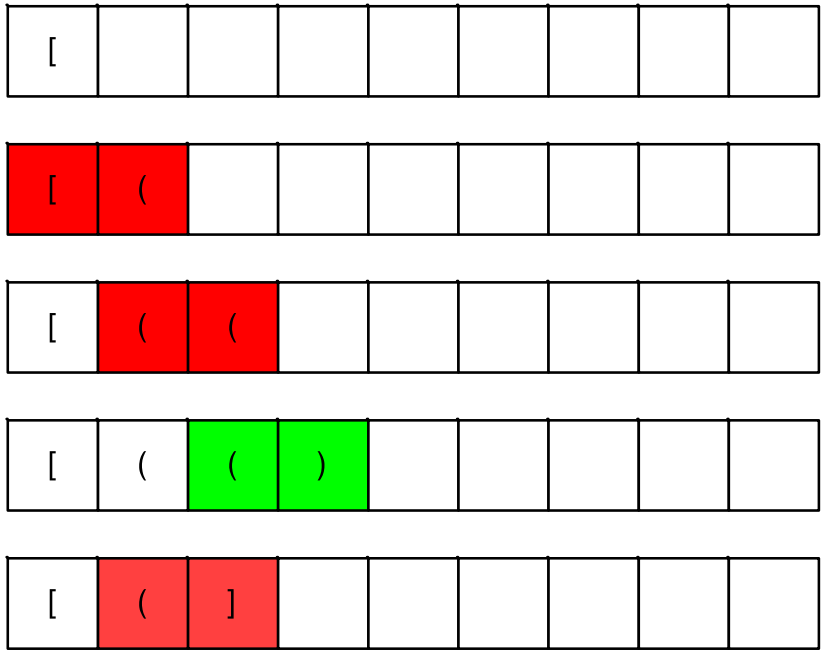
例005-括号匹配问题

- 给定一个括号表达式，中间只有[]和()，判断这个表达式是两边括号是不是平衡的？

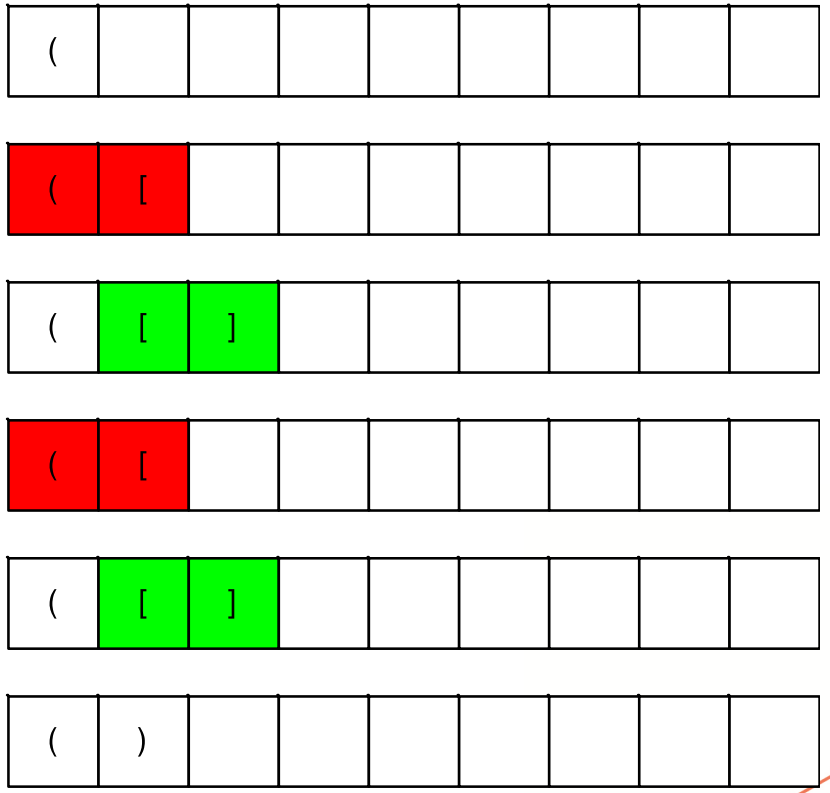
比如[(())]是平衡的，比如[()(())]就是不平衡的。

例005-基于栈的解法（先入先出)

[(()]



([][])



```
function is_balance( str ) {
  const [first, ...others] = str
  const stack = [first]
  while (others.length > 0) {
    const c = stack[stack.length - 1]
    const n = others.shift()
    if (!match(n, c)) {
      stack.push(n)
    } else {
      stack.pop()
    }
  }

  return stack.length === 0
}
```

```
function match(n,c){
  return (c=='[' && n==']')
    || (c=='(' && n==')')
}
```

集合Set的一些操作

new Set()

- add(element) // 添加、去重
- has(element) // 判断是否存在
- delete(element) // 删除
- values() // 返回Iterator

例006-数组去重

```
[...new Set(['a', 'b', 'a', 'c', 'f'])]
```

例007-子数组和整除

写一个函数，给定一个数组，判断数组中某一项，或者任意多项的和，是否被另一个整数整除。 比如：

solve([3,5,8],13) = true

solve([3, 9], 15) = false

solve([7, 8, 2], 7) = true

solve([1,2,3], 6) = true

相当于判断子数组的余数和 solve([7,8,2], 7) 等价于 solve([0, 1, 2], 7)

子问题结构

- 数组$a_1, a_2, \ldots, a_n$ 对数字N的子数组和余数集合定义为 $S_n = \{s_1, s_2, s_3 \ldots, s_m\}$。
- 比如[1,2,3]的$S_3 = \{1,2,3,4,5,6\}$, $S_2 = \{1,2,3\}, S_1 = \{1\}$。$S_k$和$S_{k-1}$存在子问题关系。

- S_{k-1}有p项, $S_k = S_{k-1} \cup a_k \,\%\, N \cup \{1 \le i \le p \mid (s_i + a_k)\%N\}$
- N=6 S2 = [1,2,3] S3 = [1,2,3] U [4,5,6]

子问题结构的解

```
1  function solve(arr, N) {
2    const s = new Set([arr.shift() % N ])
3    while(arr.length > 0) {
4      const ak = arr.shift()
5      const items = [...s]
6      items.forEach(x => {
7        s.add( (x + ak) % N )
8      })
9      s.add(ak)
10   }
11   return s.has(0)
12 }
```

关于子问题结构会在后续章节（递归、回溯、动态规划等继续介绍）

例008-数组的替换(splice用法)

```
const arr = [1,2,3,4,5,6,7]
// 替换 [3,4] => 'x'
console.log( arr.splice(2,2,'x') )
// [3, 4]
console.log( arr )
// [1, 2, "x", 5, 6, 7]

arr.splice(2,1) // 删除'x'
arr.splice(2, 0, 'y') // 在5后面添加2
console.log(arr)
// [1, 2, "y", 5, 6, 7]
```

例009-012 类似SQL的数据处理

ID	名字	小组ID	分数
1	Ruler	1	92
2	Super	1	81
3	Dog	1	30
4	Beaty	2	75
5	Jason	2	88
6	Water	2	59
7	Codez	3	21
8	Wanderful	3	98
9	Caous	3	67

ID	组名
1	Red
2	Yellow
3	Green

```
const students = [
  {id : 1, name : 'Ruler', group_id : 1, score : 92},
  {id : 2, name : 'Super', group_id : 1, score : 81},
  {id : 3, name : 'Dog', group_id : 1, score : 30},
  {id : 4, name : 'Beaty', group_id : 2, score : 75},
  {id : 5, name : 'Jason', group_id : 2, score : 88},
  {id : 6, name : 'Water', group_id : 2, score : 59},
  {id : 7, name : 'Codez', group_id : 3, score : 21},
  {id : 8, name : 'Wanderful', group_id : 3, score : 98},
  {id : 9, name : 'Caous', group_id : 3, score : 67}
]

const groups = [
  {id : 1, name : 'Red'},
  {id : 2, name : 'Yellow'},
  {id : 3, name : 'Green'}
]
```

例009-投射(projection)

>60 合格

```
const studentsWithGrade = students.map( student => {
  return {
    ...student,
    grade : student.score >= 60 ? '通过' : '不合格'
  }
})
```

创建新对象， 这样后续的操作就不会影响原来的students

- 可以出现BUG的地方减少一个是一个
- 性能消耗

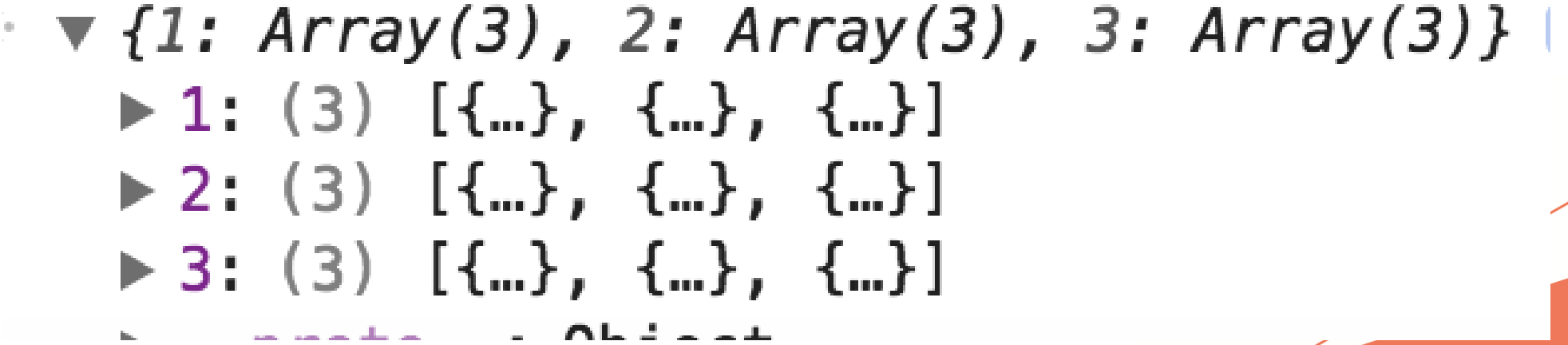
例010-过滤

```
// 通过的学员
const passedStudents = students.filter(x => x.score > 60)

// 组1的学员
const group1Students = students.filter(x => x.group_id === 1)
```

例011-分组

```
const studentsInGroups = students.reduce(
  (groups, student) => {
    groups[student.group_id] =
      [...( groups[student.group_id] || []), student]
    return groups
  },
  {}
)
```



例012-联合

```
const studentsWithGroupInfo = students.map(student => {
  const group = groups.find(g => g.id === student.group_id)
  return {
    ...student,
    groupName : group.name
  }
})
```

```
▼ (9) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▶ 0: {id: 1, name: "Ruler", group_id: 1, score: 92, groupName: "Red"}
  ▶ 1: {id: 2, name: "Super", group_id: 1, score: 81, groupName: "Red"}
  ▶ 2: {id: 3, name: "Dog", group_id: 1, score: 30, groupName: "Red"}
  ▶ 3: {id: 4, name: "Beaty", group_id: 2, score: 75, groupName: "Yellow"}
  ▶ 4: {id: 5, name: "Jason", group_id: 2, score: 88, groupName: "Yellow"}
  ▶ 5: {id: 6, name: "Water", group_id: 2, score: 59, groupName: "Yellow"}
  ▶ 6: {id: 7, name: "Codez", group_id: 3, score: 21, groupName: "Green"}
  ▶ 7: {id: 8, name: "Wanderful", group_id: 3, score: 98, groupName: "Green"}
  ▶ 8: {id: 9, name: "Caous", group_id: 3, score: 67, groupName: "Green"}
```

例013-排序

```
const sortedByScoreAsc = students.sort( (a, b) => {
  return a.score - b.score
})

const sortedByScoreDesc = students.sort( (a, b) => {
  return b.score - a.score
})
```

补充数组操作Ramda

zip	两两对齐
flatten	展平
converge	汇聚
innerJoin	联合
intersperse	间隔插入
groupBy	分组成对象
groupWith	分组成数组

Ramda Library – 一个实用的js函数库

npm install ramda

```
const R = require('ramda')
```

```
import R from 'ramda'
```

http://ramdajs.com/docs/

例013-zip

两两元素配对

```
R.zip([1, 2, 3], ['a', 'b', 'c']);
//=> [[1, 'a'], [2, 'b'], [3, 'c']]
```

例014-fatten

数组展平

```
R.flatten([1, 2, [3, 4], 5, [6, [7, 8, [9, [10, 11], 12]]]]);
//=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

例015-converge

多次聚合再进行聚合

```
var average = R.converge(R.divide, [R.sum, R.length])
average([1, 2, 3, 4, 5, 6, 7]) //=> 4

var strangeConcat = R.converge(R.concat, [R.toUpper, R.toLower])
strangeConcat("Yodel") //=> "YODELyodel"
```

例016-innerJoin

```
R.innerJoin(
  (record, id) => record.id === id,
  [{id: 824, name: 'Richie Furay'},
    {id: 956, name: 'Dewey Martin'},
    {id: 313, name: 'Bruce Palmer'},
    {id: 456, name: 'Stephen Stills'},
    {id: 177, name: 'Neil Young'}],
  [177, 456, 999]
);
//=> [
// {id: 456, name: 'Stephen Stills'},
// {id: 177, name: 'Neil Young'}
// ]
```

例017- intersperse

插入分隔符

```
R.intersperse('n', ['ba', 'a', 'a'])
//=> ['ba', 'n', 'a', 'n', 'a']
```

例018-groupBy

之前分组的代码可以简化为：

```
R.groupBy(student => student.group_id, students)
```

例019-groupWith

```
R.groupWith(R.equals, [0, 1, 1, 2, 3, 5, 8, 13, 21])
//=> [[0], [1, 1], [2], [3], [5], [8], [13], [21]]

R.groupWith((a, b) => a + 1 === b, [0, 1, 1, 2, 3, 5, 8, 13, 21])
//=> [[0, 1], [1, 2, 3], [5], [8], [13], [21]]

R.groupWith((a, b) => a % 2 === b % 2, [0, 1, 1, 2, 3, 5, 8, 13, 21])
//=> [[0], [1, 1], [2], [3, 5], [8], [13, 21]]

R.groupWith(R.eqBy(isVowel), 'aestiou')
//=> ['ae', 'st', 'iou']
```

迭代器和生成器

- 迭代器Iterator（也被称作游标Cursor)，是一种设计模式
- 迭代器提供了一种**遍历内容**的方法（比如javascript迭代器中的next)，而不需要关心内部构造。
- 生成器（Generator)本身也是一种设计模式，用于构造复杂对象。Javascript中的生成器，用于构造迭代器。

例020-迭代器的遍历

```
const s = new Set([1,2,3,4,5])
const it = s.values()
console.log(it)
// SetIterator {1, 2, 3, 4, 5}

let val = null
while( !(val = it.next()).done ){
  console.log(val)
}
// {value: 1, done: false}
// {value: 2, done: false}
// {value: 3, done: false}
// {value: 4, done: false}
// {value: 5, done: false}
```

例20-迭代器的遍历

```
const it1 = s.values()
console.log([...it1])
// (5) [1, 2, 3, 4, 5]

const it2 = s.values()
for(const val of it2){
  console.log(val)
}
// 1
// 2
// 3
// 4
// 5
```

例20-迭代器的遍历

Array.from(arrayLike, mapFn, thisArg)

- arrLike :想要转换成数组的伪数组对象或可迭代对象
- mapFn :如果指定了该参数，新数组中的每个元素会执行该回调函数
- thisArg:可选参数，执行回调函数 mapFn 时 this 对象

```
const it3 = s.values()
const arr = Array.from(
            Array(5), it3.next, it3).map(x => x.value)
console.log(arr)
// (5) [1, 2, 3, 4, 5]
```

例21-生成器构造无穷斐波那契数列

```
function* fibonacci(){
  let a = 1, b = 1
  yield a; yield b
  while(true){
    const t = b
    b = a + b; a = t
    yield b
  }
}
const it = fibonacci()
const feb10 = Array.from(
            Array(10), it.next, it).map(x=>x.value)
console.log(feb10)
// [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

例22-数组展平的生成器实现

```
function* flatten(arr){
  for(let i = 0; i < arr.length; i++){
    if(Array.isArray(arr[i])){
      yield * flatten(arr[i])
    }else{
      yield arr[i]
    }
  }
}
console.log([...flatten([1,2,[3,4,[[5]]]])])
// (5)[1,2,3,4,5]
```

Generator处理异步逻辑

形如：

const x = yield request(url)

将异步逻辑变成了同步形式，那么如何做呢？

Iterator.next方法的第一个参数，提供了将值向yield出传递的方法。

例23-Generator异步语法

```
function request(url){
  return cb => {
    setTimeout(() => {
      cb(Math.random())
    }, 1000)
  }
}

create_runner( function*(){
  const val1 = yield request('some url')
  const val2 = yield request('some url')
  console.log(val1, val2)
})()
```

例23-Generator异步语法

```
function create_runner( genFunc ){
  const it = genFunc()
  function run(data){
    const itVal = it.next(data)
    if(!itVal.done){
      itVal.value(run)
    }
  }
  return run
}
```

生成器的优势

- 简化语法、写起来少考虑一些问题 (例22少考虑了concat)
- 节省空间(例22中函数体不需要定义数组承载值)
- 分散执行片段(节省单位时间的处理量 ） ——对于单线程的前端非常重要
- 构造异步语法

问题补充

- 笛卡尔积
- 中文排序

例24-笛卡尔积

集合X 和 Y 的笛卡尔积 可以表示为： $X \times Y = \{ (x, y) | x \in X \wedge y \in Y \}$

写一个函数，求数组的笛卡尔积

例如

[1,2] X ['a', 'b'] = [[1, 'a'], [1, 'b'], [2, 'a'], [2, 'b']]

```
function cartesian_product(...Matrix) {
  if(Matrix.length === 0) return []
  if(Matrix.length === 1) return Matrix[0]
  return Matrix.reduce((A, B) => {
    const product = []
    for(let i = 0; i < A.length; i++)
      for(let j = 0; j < B.length; j++){
        product.push (
          Array.isArray(A[i]) ?
                [...A[i], B[j]] : [A[i], B[j]] )
      }
    return product
  })
}
```

例25-中文排序

将含有中文字符的数组按照拼音排序：

```
["王成成", "王峰", "蒋雪", "李明"]
  .sort((a,b) => a.localeCompare(b, 'zh'))

//["蒋雪", "李明", "王成成", "王峰"]
```

归纳和总结

- 遍历方法的总结
- 链式操作的优缺点
- 什么时候应该拷贝数据?
- 为什么需要Ramda这些库?

遍历方法总结

	数组	Break	迭代器
for	支持	支持	不支持
for...of..	支持	支持	支持
map	支持	不支持	不支持
forEach	支持	不支持	不支持
for...in..	支持	支持	不支持
while	支持	支持	支持

链式操作

数组的很多操作可以构成链式操作，类似这样的格式：

...map(...).filter(...).sort(...).map(....)

链式操作就是对象方法返回类型是**自身**的。比如map是属于数组的方法，它返回数组，所以构成了链式操作。

优势： 语义清晰、思考方便
问题： 性能、空间
数据量小的时候很有用（<1W)

什么时候需要拷贝数据？

javascript中push/pop/shift/unshift/splice等都在原始数据上进行修改，concat/slice/map/reduce都会对原始数据进行浅拷贝。

参考**例009-投射**，使用**map**对**students**数组进行处理的时候，每一次map都将原来数据进行了深拷贝。那么为什么要进行这种复制操作呢？

例26-改变原始数据很危险

```
function sort(a){
  for(let i = 1; i < a.length; i++){
    let card = a[i] // 抓到的牌
    let j = i // j代表最终牌插入的位置
    while(j > 0 && card < a[j-1]){
      a[j] = a[j-1]
      j--
    }
    a[j] = card
  }
}
            const A = [2,3,5,3]
            sort(A)
            console.log(A)
```

比如说这个插入排序的实现，直接改变了原始数组

例26-改变原始数据很危险

```
const A = [2,3,5,3]
sort(A)
console.log(A)
// [2,3,3,5]
```

- 将来无法再使用原来的数据A. (A已经变了)
- 程序员忘记了A已经发生了变化(更可怕)

```
function sort(_a){
  const a = [...a]
  for(let i = 1; i < a.length; i++){
    let card = a[i] // 抓到的牌
    let j = i // j代表最终牌插入的位置
    while(j > 0 && card < a[j-1]){
      a[j] = a[j-1]
      j--
    }
    a[j] = card
  }
  return a
}

const A = [2,3,5,3]
const B = sort(A)
console.log(B)
```

Ramda相关

为什么要用ramda?

JS标准函数库数据处理函数较少

如何记忆这么多函数？

http://ramdajs.com/docs

http://ramda.cn/docs/ 中文

读一遍文档知道每个函数做什么的

在第4章还会详细介绍ramda的一切