目录

[基础 2](#_Toc490039561)

[顶层api 2](#_Toc490039562)

[formJs 2](#_Toc490039563)

[List 3](#_Toc490039564)

[构造函数 3](#_Toc490039565)

[List.isList(value) 3](#_Toc490039566)

[List.of(...values) 3](#_Toc490039567)

[List对象属性 3](#_Toc490039568)

[size 3](#_Toc490039569)

[List持久化方法 4](#_Toc490039570)

[set 4](#_Toc490039571)

[delete 4](#_Toc490039572)

[insert 4](#_Toc490039573)

[clear 4](#_Toc490039574)

[push 4](#_Toc490039575)

[pop 5](#_Toc490039576)

[unshift 5](#_Toc490039577)

[shift 5](#_Toc490039578)

[update 5](#_Toc490039579)

[merge 6](#_Toc490039580)

[mergeWith 6](#_Toc490039581)

[mergeDeep 6](#_Toc490039582)

[mergeDeepWIth 7](#_Toc490039583)

[setSize 7](#_Toc490039584)

[深度持久化 7](#_Toc490039585)

[setIn 7](#_Toc490039586)

[deleteIn 8](#_Toc490039587)

[updateIn 8](#_Toc490039588)

[mergeIn 8](#_Toc490039589)

[mergeDeepIn 8](#_Toc490039590)

[list中间态方法 8](#_Toc490039591)

[withMutations 8](#_Toc490039592)

[asMutable 9](#_Toc490039593)

[asImmutable 9](#_Toc490039594)

[list序列化方法 9](#_Toc490039595)

[concat 9](#_Toc490039596)

[map 9](#_Toc490039597)

[flatMap 9](#_Toc490039598)

[filter 9](#_Toc490039599)

[zip 10](#_Toc490039600)

[zipWith 10](#_Toc490039601)

[[Symbol.iterator] 11](#_Toc490039602)

[filterNot 11](#_Toc490039603)

[reverse 11](#_Toc490039604)

[sort 11](#_Toc490039605)

[sortBy 12](#_Toc490039606)

[groupBy 12](#_Toc490039607)

[转换成js 12](#_Toc490039608)

[toJs 12](#_Toc490039609)

[toJSON 12](#_Toc490039610)

[toArray 12](#_Toc490039611)

[toObject 12](#_Toc490039612)

[读值 12](#_Toc490039613)

[get 12](#_Toc490039614)

[first 13](#_Toc490039615)

[last 13](#_Toc490039616)

# 基础

immutable中的map和list对应js中的object和array,不是es6里面那两个。

没有length ,而是 size

不能直接用obj.xxx

# 顶层api

## formJs

将 JavaScript Object 和 Array 彻底转换为 Immutable Map 和 List

fromJS() 的使用方式类似于 JSON.parse()，接收两个参数：json 数据和 reviver 函数。

Immutable.fromJS({a: {b: [10, 20, 30]}, c: 40}, function (key, value) {

var isIndexed = Immutable.Iterable.isIndexed(value);

return isIndexed ? value.toList() : value.toOrderedMap();

});

// true, "b", {b: [10, 20, 30]}

// false, "a", {a: {b: [10, 20, 30]}, c: 40}

// false, "", {"": {a: {b: [10, 20, 30]}, c: 40}}

# List

## 构造函数

参数类型

List<T>(): List<T>

List<T>(iter: Iterable.Indexed<T>): List<T>

List<T>(iter: Iterable.Set<T>): List<T>

List<K, V>(iter: Iterable.Keyed<K, V>): List<any>

List<T>(array: Array<T>): List<T>

List<T>(iterator: Iterator<T>): List<T>

List<T>(iterable: Object): List<T>

## List.isList(value)

判断 value 是否是 List 类型

## List.of(...values)

创建包含 ...values 的列表

## List对象属性

### size

const $arr1 = List([1, 2, 3]);

$arr1.size

## List持久化方法

### set

set(index: number, value: T): [List](http://facebook.github.io/immutable-js/docs/#/List)<T>

给指定位置设置值，返回一个新的对象，如果值已经存在，会覆盖。

index可以是负数，表示从最后开始。v.set(-1, "value")表示最后一个值。

index大于list长度时，会增加长度，中间为undefined。

### delete

别名：remove（）

效果等同于js中list.splice(index, 1)。

delete(index: number): [List](http://facebook.github.io/immutable-js/docs/#/List)<T>

ie8不能安全使用。

v.delete(-1)

### insert

insert(index: number, value: T): [List](http://facebook.github.io/immutable-js/docs/#/List)<T>

等同于list.splice(index, 0, value).

### clear

clear(): [List](http://facebook.github.io/immutable-js/docs/#/List)<T>

### push

在最后添加

push(...values: Array<T>): [List](http://facebook.github.io/immutable-js/docs/#/List)<T>

### pop

注意js中pop是返回被移除的值。在immutable中可以使用last()获取list最后一个值。

### unshift

unshift(...values: Array<T>): [List](http://facebook.github.io/immutable-js/docs/#/List)<T>

在头部添加。

### shift

在头部移除。

注意js中是返回被移除的值，在immutable中可以使用first()获取list最后一个值。

### update

update(index: number, notSetValue: T, updater: (value: T) => T): this update(index: number, updater: (value: T) => T): this update<R>(updater: (value: **this**) => R): R／／一个参数是传递list本身

function sum(collection) {

return collection.reduce((sum, x) => sum + x, 0)

}

List([ 1, 2, 3 ])

.map(x => x + 1)

.filter(x => x % 2 === 0)

.update(sum)

// 6

### merge

合并（例子看map）

merge(...collections: Array<[Collection.Indexed](https://facebook.github.io/immutable-js/docs/#/Collection.Indexed)<T> | Array<T>>): this

如果合并的任何值都不是集合(返回false)，然后在合并之前通过fromJS进行深度转换。但是，如果值是一个集合，但是包含非集合的JS对象或数组，那么这些嵌套的值将被保留。

合并后调用者数据在前，被调用者在后，重复的由被调用者覆盖。

const { Map } = require('immutable')

const one = Map({ a: 10, b: 20, c: 30 })

const two = Map({ b: 40, a: 50, d: 60 })

one.merge(two) // Map { "a": 50, "b": 40, "c": 30, "d": 60 }

### mergeWith

类似merge，但使用合并函数进行处理。

mergeWith(

merger: (oldVal: T, newVal: T, **key: number**) => T,  
...collections: Array<[Collection.Indexed](https://facebook.github.io/immutable-js/docs/#/Collection.Indexed)<T> | Array<T>>

): this

调用者是old，被调用者是new，返回map的顺序为old

const one = Map({ a: 10, b: 20, c: 30 })

const two = Map({ b: 40, a: 50, d: 60 })

one.mergeWith((oldVal, newVal) => oldVal / newVal, two)

// { "a": 0.2, "b": 0.5, "c": 30, "d": 60 }

### mergeDeep

mergeDeep(...collections: Array<[Collection.Indexed](https://facebook.github.io/immutable-js/docs/#/Collection.Indexed)<T> | Array<T>>): this

类似merge，但是当两个集合发生冲突时，它也会合并它们，并通过嵌套的数据反复地递归。

const one = Map({ a: Map({ x: 10, y: 10 }), b: Map({ x: 20, y: 50 }) })

const two = Map({ a: Map({ x: 2 }), b: Map({ y: 5 }), c: Map({ z: 3 }) })

one.mergeDeep(two)

// Map {

// "a": Map { "x": 2, "y": 10 },

// "b": Map { "x": 20, "y": 5 },

// "c": Map { "z": 3 }

// }

### mergeDeepWIth

mergeDeepWith(

merger: (oldVal: T, newVal: T, key: number) => T,  
...collections: Array<[Collection.Indexed](https://facebook.github.io/immutable-js/docs/#/Collection.Indexed)<T> | Array<T>>

): this

和mergeDeep()一样，但是当两个非集合发生冲突时，它使用合并函数来确定结果的值。就是说如果是里面的集合冲突会进行递归调用。

### setSize

setSize(size: number): [List](https://facebook.github.io/immutable-js/docs/#/List)<T>

返回一个新的list，如果size大于长度，增加部分的值为undefined，小于则删掉后面的值

结合withMutations效果可能更好。

## 深度持久化

**和普通函数差别: index(num) => keyPath(Iterable<any>)**

### setIn

setIn(keyPath: Iterable<any>, value: any): this

index不存在会创建

const list = List([ 0, 1, 2, List([ 3, 4 ])])

list.setIn([3, 0], 999);

// List [ 0, 1, 2, List [ 999, 4 ] ]

### deleteIn

### updateIn

### mergeIn

### mergeDeepIn

## list中间态方法

### withMutations

支持：

set clear push pop unshift shift update merge mergeWith mergeDeep mergeDeepWith setIn

批量修改

由于每一次修改都会生成一个新的immutable对象有一定的性能损耗，可以使用withMutations方法，使批量修复在临时集合拷贝上发生。

var list1 = List.of(1,2,3);

var list2 = list1.withMutations(function (list) {／／中间list

list.push(4).push(5).push(6);

});

console.log(list1.size === 3);

console.log(list2.size === 6);

注： immutable 还提供了 asMutable 和 asImmutable ， 但是只有在 withMutation 不能满足情况的时候使用。注意不要返回一个可变的拷贝，会导致非预知的行为。

重要！：只有特殊的几个方法能使用 withMutation 其中包括 set, push 和 pop。这些方法能直接作用于内部数据结构，而方法如 map, filter, sort, splice 一定不会修改可修改集合并返回新的不变集合。

### asMutable

### asImmutable

## list序列化方法

### concat

concat<C>(...valuesOrCollections: Array<Iterable<C> | C>): List<T | C>

### map

map<M>(mapper: (value: T, key: number, iter: this) => M, context?: any): List<M>

### flatMap

类似list.map(...).flatten(true)

### filter

总是返回新的list，即使没有匹配到任何数据。

filter<F>(

predicate: (value: T, index: number, iter: this) => boolean,

context?: any

): List<F>

filter(

predicate: (value: T, index: number, iter: this) => any,

context?: any

): this

### zip

zip(...collections: Array<Collection<any, any>>): List<any>

const a = List([ 1, 2, 3 ]);

const b = List([ 4, 5, 6 ]);

const c = a.zip(b); // List [ [ 1, 4 ], [ 2, 5 ], [ 3, 6 ] ]

### zipWith

zipWith<U, Z>(

zipper: (value: T, otherValue: U) => Z,

otherCollection: Collection<any, U>

): List<Z>

zipWith<U, V, Z>(

zipper: (value: T, otherValue: U, thirdValue: V) => Z,

otherCollection: Collection<any, U>,

thirdCollection: Collection<any, V>

): List<Z>

zipWith<Z>(

zipper: (...any: Array<any>) => Z,

...collections: Array<Collection<any, any>>

): List<Z>

类似zip，但是使用自定义的zipper方法。

const a = List([ 1, 2, 3 ]);

const b = List([ 4, 5, 6 ]);

const c = a.zipWith((a, b) => a + b, b);

// List [ 5, 7, 9 ]

### [Symbol.iterator]

### filterNot

filterNot(

predicate: (value: T, key: number, iter: this) => boolean,  
context?: any

): this

和filter相反

### reverse

### sort

sort(comparator?: (valueA: T, valueB: T) => number): this

返回: 0 不变；-1 <;1,>

默认使用< >判断。

### sortBy

sortBy<C>(

comparatorValueMapper: (value: T, key: number, iter: this) => C,  
comparator?: (valueA: C, valueB: C) => number

): this

### groupBy

## 转换成js

### toJs

深度转换成js数组

### toJSON

浅度转换成js数组

### toArray

浅转换为数组，放弃键

### toObject

浅转换成对象

## 读值

### get

get<NSV>(index: number, notSetValue: NSV): T | NSV

get(index: number): T | undefined

index可以是负数。

### has

has(key: number): boolean

includes

includes(value: T): boolean

使用immutablejs确定相等性。

别名： contains

### first

first(): T | undefined

### last