在上面代码第三行中，map1 使用 set 方法更新数据，结果返回一个新的 Map 类型数据 map2，map2 包含了更新后的数据，但是 map1 没有发生变化。这种特性让我们在引用数据的时候毫无后顾之忧，因为任何对数据的修改都不会影响最原始的数据。在 Immutable.js 诞生之前，我们可以使用深拷贝的方式模拟这一特性，但是会耗费过多的内存空间和计算力。Immutable.js 相比深拷贝的优势在于区分发生变化的数据和未变化的数据，对于上面的 map1 和 map2，b 是变化的数据，所以 map1 和 map2 各保存一份 b 数据，而 a 和 c 是未变化的数据，所以 map1 和 map2 仍然共享 a 和 c 的数据。

**概览**

Immutable Data 鼓励开发者使用纯函数式的开发方式，并从函数式开发中引入了惰性计算的特性。虽然加入了很多函数式的概念，Immutable.js 仍然提供了类似原生 JavaScript Array、Map 和 Set 中的方法，并且提供了在原生 JavasScript 数据和 Immutable 数据之间快速转换的机制。

Immutable.js 的 API 主要包含以下几部分：

* formJS()，将 JavaScript Object 和 Array 彻底转换为 Immutable Map 和 List
* is()，与 Object.is() 类似都是对值的比较，但它会将 Immutable Iterable 视为值类型数据而不是引用类型数据，如果两个 Immutable Iterable 的值相等，则返回 true。与 Object.is() 不同的是，is(0, -0) 的结果为 true
* List，有序索引集，类似于 JavaScript 中的 Array
* Map，无序 Iterable，读写 Key 的复杂度为 O(log32 N)
* OrderedMap，有序 Map，排序依据是数据的 set() 操作
* Set，元素为独一无二的集合，添加数据和判断数据是否存在的复杂度为 O(log32 N)
* OrderedSet，有序 Set，排序依据是数据的 add 操作。
* Stack，有序集合，且使用 unshift(v) 和 shift() 进行添加和删除操作的复杂度为 O(1)
* Range()，返回一个 Seq.Indexed 类型的数据集合，该方法接收三个参数 (start = 1, end = infinity, step = 1)，分别表示起始点、终止点和步长，如果 start 等于 end，则返回空的数据结合
* Repeat()，返回一个 Seq.indexed 类型的数据结合，该方法接收两个参数 (value，times)，value 表示重复生成的值，times 表示重复生成的次数，如果没有指定 times，则表示生成的 Seq 包含无限个 value
* Record，用于衍生新的 Record 类，进而生成 Record 实例。Record 实例类似于 JavaScript 中的 Object 实例，但只接收特定的字符串作为 key，且拥有默认值
* Seq，序列（may not be backed by a concrete data structure）
* Iterable，可以被迭代的 (Key, Value) 键值对集合，是 Immutable.js 中其他所有集合的基类，为其他所有集合提供了 基础的 Iterable 操作函数（比如 map() 和 filter）
* Collection，创建 Immutable 数据结构的最基础的抽象类，不能直接构造该类型

**1. fromJS()**

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}}

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

**2. List**

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() 是一个构造方法，可以用于创建新的 List 数据类型，上面代码演示了该构造方法接收的参数类型，此外 List 拥有两个静态方法：

* List.isList(value)，判断 value 是否是 List 类型
* List.of(...values)，创建包含 ...values 的列表

下面演示几个 List 常用的操作，更详细的 API 说明请参考官方文档：

// 1. 查看 List 长度

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

$arr1.size

// => 3

// 2. 添加或替换 List 实例中的元素

// set(index: number, value: T)

// 将 index 位置的元素替换为 value，即使索引越界也是安全的

const $arr2 = $arr1.set(-1, 0);

// => [1, 2, 0]

const $arr3 = $arr1.set(4, 0);

// => [ 1, 2, 3, undefined, 0 ]

// 3. 删除 List 实例中的元素

// delete(index: number)

// 删除 index 位置的元素

const $arr4 = $arr1.delete(1);

// => [ 1, 3 ]

// 4. 向 List 插入元素

// insert(index: number, value: T)

// 向 index 位置插入 value

const $arr5 = $arr1.insert(1, 1.5);

// => [ 1, 1.5, 2, 3 ]

// 5. 清空 List

// clear()

const $arr6 = $arr1.clear();

// => []

**3. Map**

Map 可以使用任何类型的数据作为 Key 值，并使用 Immutable.is() 方法来比较两个 Key 值是否相等：

Map().set(List.of(1), 'listofone').get(List.of(1));

// => 'listofone'

但是使用 JavaScript 中的引用类型数据（对象、数组）作为 Key 值时，虽然有时两个 Key 很像，但它们也是两个不同的 Key 值：

console.log(Map().set({}, 1).get({}))

// => undefined

Map() 是 Map 类型的构造方法，行为类似于 List()，用于创建新的 Map 实例，此外，还包含两个静态方法：Map.isMap() 和 Map.of()。下面介绍几个 Map 实例的常用操作，更详细的 API 使用说明请参考官方文档：

// 1. Map 实例的大小

const $map1 = Map({ a: 1 });

$map1.size

// => 1

// 2. 添加或替换 Map 实例中的元素

// set(key: K, value: V)

const $map2 = $map1.set('a', 2);

// => Map { "a": 2 }

// 3. 删除元素

// delete(key: K)

const $map3 = $map1.delete('a');

// => Map {}

// 4. 清空 Map 实例

const $map4 = $map1.clear();

// => Map {}

// 5. 更新 Map 元素

// update(updater: (value: Map<K, V>) => Map<K, V>)

// update(key: K, updater: (value: V) => V)

// update(key: K, notSetValue: V, updater: (value: V) => V)

const $map5 = $map1.update('a', () => (2))

// => Map { "a": 2 }

// 6. 合并 Map 实例

const $map6 = Map({ b: 2 });

$map1.merge($map6);

// => Map { "a": 1, "b": 2 }

**OrderedMap** 是 Map 的变体，它除了具有 Map 的特性外，还具有顺序性，当开发者遍历 **OrderedMap** 的实例时，遍历顺序为该实例中元素的声明、添加顺序。

**4. Set**

**Set** 和 ES6 中的 Set 类似，都是没有重复值的集合，**OrderedSet** 是 Set 的遍历，可以保证遍历的顺序性。

// 1. 创建 Set 实例

const $set1 = Set([1, 2, 3]);

// => Set { 1, 2, 3 }

// 2. 添加元素

const $set2 = $set1.add(1).add(4);

// => Set { 1, 2, 3, 4 }

// 3. 删除元素

const $set3 = $set1.delete(3);

// => Set { 1, 2 }

// 4. 并集

const $set4 = Set([2, 3, 4, 5, 6]);

$set1.union($set1);

// => Set { 1, 2, 3, 4, 5, 6 }

// 5. 交集

$set1.intersect($set4);

// => Set { 3, 2 }

// 6. 差集

$set1.subtract($set4);

// => Set { 1 }

**5. Stack**

Stack 是基于 Signle-Linked List 实现的可索引集合，使用 unshift(v) 和 shift() 执行添加和删除元素的复杂度为 O(1)。

// 1. 创建 Stack 实例

const $stack1 = Stack([1, 2, 3]);

// => Stack [ 1, 2, 3 ]

// 2. 取第一个元素

$stack1.peek()

// => 1

// 2. 取任意位置元素

$stack1.get(2)

// => 3

// 3. 判断是否存在

$stack1.has(10)

// => false

**6. Range() 和 Repeat()**

Range(start?, end?, step?) 接收三个可选参数，使用方法如下：

// 1. 不传参

Range();

// => Range [ 0...Infinity ]

// 2. 设置 start 起点

Range(10);

// => Range [ 10...Infinity ]

// 3. 设置 start 起点和 end 终点

Range(10, 20);

// => Range [ 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 ]

// 4. 设置 start 起点、end 终点和 step 步长

Range(10, 20, 3);

// => Range [ 10, 13, 16, 19 ]

Repeat(value, times?) 接收两个参数，其中 times 重复次数是可选参数：

Repeat('foo');

// => Repeat [ foo Infinity times ]

Repeat('foo', 3);

// => Repeat [ foo 3 times ]

类似 Range() 和 Repeat(value) 这样生成无限长度集合的操作，内部都存在惰性计算的机制，只有真实取值时才会生成相应的结果。使用 ES6 中的 Generator 函数，可以轻松实现一个惰性计算：

function\* bigArr() {

for (let i = 0; i < 100000; i++) {

console.log(`bigArr(${i}): ${i}`)

yield i;

}

}

const arr = bigArr();

for (let i = 0; i < 10; i++) {

console.log(arr.next());

}

// bigArr(0): 0

// => { value: 0, done: false }

// => bigArr(1): 1

// => { value: 1, done: false }

// => bigArr(2): 2

// => { value: 2, done: false }

// => bigArr(3): 3

// => { value: 3, done: false }

// => bigArr(4): 4

// => { value: 4, done: false }

// => bigArr(5): 5

// => { value: 5, done: false }

// => bigArr(6): 6

// => { value: 6, done: false }

// => bigArr(7): 7

// => { value: 7, done: false }

// => bigArr(8): 8

// => { value: 8, done: false }

// => bigArr(9): 9

// => { value: 9, done: false }

**7. Record**

Record 在表现上类似于 ES6 中的 Class，但在某些细节上还有所不同。通过 Record() 可以创建一个新的 Record 类，使用该类可以创建具体的 Record 实例，该实例包含在 Record() 构造函数中声明的所有属性和默认值。如果 Record 实例中的某个属性被删除了，则只会讲实例中的属性值恢复为默认值：

// 1. 创建 Record 实例

const A = Record({ a: 1, b: 2 });

const r = new A({ a: 3 });

// => Record { "a": 3, "b": 2 }

// 2. 删除实例属性

const rr = r.remove('a');

// => Record { "a": 1, "b": 2 }

此外，Record 实例还具有扩展性：

class ABRecord extends Record({a:1,b:2}) {

getAB() {

return this.a + this.b;

}

}

var myRecord = new ABRecord({b: 3})

myRecord.getAB()

// => 4

**8. Seq**

Seq 有两个特点：immutable，一旦创建就不能被修改；lazy，惰性求值。在下面的代码中，虽然组合了多种遍历操作，但实际上并不会有任何的求值操作，只是纯粹的声明一个 Seq：

var oddSquares = Immutable.Seq.of(1,2,3,4,5,6,7,8)

.filter(x => x % 2)

.map(x => x \* x);

如果要从 oddSquares 中取出索引为 1 的元素，则执行过程为：

console.log(oddSquares.get(1));

// filter(1)

// filter(2)

// filter(3)

// map(3)

// => 9

Seq() 是 Seq 的构造方法，它根据传入的参数类型，输出响应的 Seq 类型：

* 输入 Seq，输出 Seq
* 输入 Iterable，输出同类型的 Seq(Keyed, Indexed, Set)
* 输入 Array-like，输出 Seq.Indexed
* 输入附加 Iterator 的 Object，输出 Seq.Indexed
* 输入 Iterator，输出 Seq。indexed
* 输入 Object，输出 Seq.Keyed

默认情况下 Seq 的惰性计算结果不会被缓存，比如在下面的代码中，由于每个 join() 都会遍历执行 map，所以 map 总共执行了六次：

var squares = Seq.of(1,2,3).map(x => x \* x);

squares.join() + squares.join();

如果开发者知道 Seq 的结果会被反复用到，那么就可以使用 cacheResult() 将惰性计算的结果保存到内存中：

var squares = Seq.of(1,2,3).map(x => x \* x).cacheResult();

squares.join() + squares.join();

**9. Iterable 和 Collection**

Iterable 是键值对形式的集合，其实例可以执行遍历操作，是 immutable.js 中其他数据类型的基类，所有扩展自 Iterable 的数据类型都可以使用 Iterable 所声明的方法，比如 map 和 filter 等。

Collection 是 Concrete Data Structure 的基类，使用该类时需要至少继承其子类中的一个：Collection.Keyed / Collection.Indexed / Collection.Set。

**React**

在 React 官方文档的[《Advanced Performance》](https://facebook.github.io/react/docs/advanced-performance.html) 一节中，专门对 React 的性能瓶颈、优化方式做了详细的解析。当一个 React 组件的 props 和 state 发生变化时，React 会根据变化后的 props 和 state 创建一个新的 virtual DOM，然后比较新旧两个 vritual DOM 是否一致，只有当两者不同时，React 才会将 virtual DOM 渲染真实的 DOM 结点，而对 React 进行性能优化的核心就是减少渲染真实 DOM 结点的频率，间接地指出开发者应该准确判断 props 和 state 是否真正发生了变化。

在比对新旧 vritual DOM 和渲染真实 DOM 前，React 为开发者提供了 shouldComponentUpdate() 方法中断接下来的比对和渲染操作，默认情况下，该方法总会返回 true，如果它返回 false，则不执行比对和渲染操作：

// 最简单的实现：

shouldComponentUpdate (nextProps) {

return this.props.value !== nextProps.value;

}

看起来挺简单，实在不然。当我们需要比对的值是对象、数组等引用值时，就会出现问题：

// 假设 this.props.value 是 { foo: 'bar' }

// 假设 nextProps.value 是 { foo: 'bar' },

// 显然这两者引用的内存地址不同，但它们具有相同的值，这种时候不应该继续执行渲染

this.props.value !== nextProps.value; // true

如果数据是 Immutable Data 的话，那么数据发生变化就会生成新的对象，开发者只需要检查对象应用是否发生变化即可：

var SomeRecord = Immutable.Record({ foo: null });

var x = new SomeRecord({ foo: 'bar' });

var y = x.set('foo', 'baz');

x === y; // false

处理这一问题的另一种方式是通过 setter 设置 flag 对脏数据进行检查，但冗杂的代码是在让人头疼。

**总结**

Immutable.js 所提供的 Immutable Data 和 JavaScript 固有的 Mutable Data 各有优势，未来 ECAMScript 有可能制定一套原生的 Immutable Data 规范，在这之前，Immutable.js 是一个不错的选择。之前已经写文章熟悉过 Lodash 这一工具库，Immutable 内部也封装了诸多常用的数据操作函数，所以如果让我来选择的话，在 React 技术栈中我会优先选择 Immutable，而放弃 Lodash。

著作权归作者所有。  
商业转载请联系作者获得授权,非商业转载请注明出处。  
原文: <http://www.w3cplus.com/javascript/immutable-js.html> © [w3cplus.com](http://www.w3cplus.com)