IMMUTABLE DATA STRUCTURES FOR FUNCTIONAL JS

@AnjanaVakil

O'Reilly Fluent 2017
HI! 🙋‍♀️

I'M ANJANA
Functional programming rocks!
**Functional Programming**

- Pure functions:
  - input $\rightarrow$ output
  - side effects
  - data in, data out

**Mutable State**
Immutability rocks!
ROCKS ROCK!
Nobody sits like this rock sits.
You rock, rock.
The rock just sits, and is.
You show us how to just sit here, and that's what we need.

— I ❤ HUCKABEES (2004)
IN A LAND WHERE MUTATION REIGNS...
foo 0 1 2 3 4 5 6 7
zoo
zoo
Who put a 👽 in my zoo?!?
With mutability come overhead & bugs 😞
Mutation must be stopped!
(sorry, xavier)
Mutation must be stopped!
(sorry, xavier)

Let's journey to the land of...
IMMUTABLE DATA!

it just sits, and is (like rocks)
zoo
new zoo
zoo

new
zoo

new

Great! My zoo just sits, and is!
...but my code runs like 🐒 & 🐳.
COPYING WASTES TIME & SPACE
ISN’T THERE A BETTER WAY?
ISN’T THERE A BETTER WAY?

WE NEED A HERO...
PERSISTENT DATA STRUCTURES!
PERSISTENT DATA STRUCTURES!

MASTERS OF TIME & SPACE!
persistent data structures!

old versions stay put...

masters of time & space!

persistent data structures!
PERSISTENT DATA STRUCTURES!

OLD VERSIONS STAY PUT...

... NEW VERSIONS CREATED EFFICIENTLY!

MASTERS OF TIME & SPACE!
What's the source of their power?
TREES!
zoo
HOW DO WE UPDATE THINGS?
PATH
COPYING!

new

zoo

0 1
0 1
2 3
4 5
6 7
**PATH COPYING!**

new

zoo

**STRUCTURAL SHARING!**
AWESOME! I can reuse most of the data!
TREES + SHARING
TURN 🐢 & 🐬 INTO 🏌️‍♂️ & 🦐
BUT HOW DO WE ACCESS THINGS?
BUT HOW DO WE ACCESS THINGS?

OUR TREE REVEALS A SECRET IDENTITY...
IT'S A TRIE!
leaves hold values
paths represent keys
CONVERT INDEX TO BINARY
DESCEND TRIE BIT BY BIT
zoo[5]
zoo[5]
zoo[0b101]
zoo[5]
zoo[0b101]
zoo→1→0→1
zoo[5]
zoo[\texttt{0b101}]
zoo→1→0→1
\texttt{zoo[5]}
\texttt{zoo[0b101]}
\texttt{zoo→1→0→1}
zoo[5]
zoo[\text{\texttt{\textcolor{red}{0b101}}}]
zoo→1→0→1
IT'S A ！
IT'S A!

IT'S A!
IT'S A!
IT'S A!
IT'S...

!
BITMAPPED VECTOR TRIE!
LET'S TALK COMPLEXITY...
Immutable Array

- lookup: $O(1)$
- update: $O(n)$

Bitmapped Vector Trie

- lookup: $O(\log n)$
- update: $O(\log n)$
zoo[18977]

zoo[0b100101000100001000001]

zoo→1→0→0→1→0→1→0→...
this is taking forever….
WHO SAID THE 🌳 NEEDS
2-WAY BRANCHING?!? 
(⚠️ BIT PER LEVEL)
Who said the 🌳 needs 2-way branching?!?

([Index Bit Per Level])

We can choose...
FEWER BRANCHES

- deep trees
+ small nodes

MORE BRANCHES

+ shallow trees
- large nodes
32-way branching is a good balance! (👍 bits per level)
zoo[18977]

zoo[\texttt{0b100101000100001}]

zoo→10010→10001→00001
**Immutable Array**

- **lookup:** $O(1)$
- **update:** $O(n)$

**VS.**

**Bitmapped Vector Trie**

- **lookup:** $O(\log n)$
- **update:** $O(\log n)$
IMMUTABLE ARRAY
lookup: $O(1)$
update: $O(n)$

VS.

BITMAPPED VECTOR TRIE
lookup: $O(\log_{32} n)$
update: $O(\log_{32} n)$
**Immutable Array**

lookup: $O(1)$

update: $O(n)$

**Bitmapped Vector Trie**

lookup $\approx O(1)$

update $\approx O(1)$
WHAT ABOUT OBJECTS? 🤔 WE NEED NON-INTEGER KEYS TOO
Hash the key to get a number. Descend trie as before.
`zoo["g"]`

```plaintext
hash("g") === 5 === 0b101
```

```
zoo→1→0→1
```
zoo[hash("g")] 
zoo[\texttt{0b101}] 
zoo→1→0→1
Awesome! I can use whatever keys I want!
HASH
ARRAY MAPPED
TRIE!
LET’S RECAP!

**MUTABILITY:** 😞

**IMMUTABILITY:** 🤘

**COPYING:** 🐢 & 🐋

**SHARING:** 🐎 & 🦐
These are some data structures! But how do we use them in JavaScript? 🤔
LIBRARIES!
IMMUTABLE.JS!
MORI!
Immutable collections for JavaScript

Immutable data cannot be changed once created, leading to much simpler application development, no defensive copying, and enabling advanced memoization and change detection techniques with simple logic. Persistent data presents a mutative API which does not update the data in-place, but instead always yields new updated data.

Immutable.js provides many Persistent Immutable data structures including: List, Stack, Map, OrderedMap, Set, OrderedSet and Record.
var Imjs = require("immutable");

var a = Imjs.List.of(1,2);
// List [1, 2]

var a2 = a.push(3);
// List [1, 2, 3]

a.size;  // 2

a2.get(2);  // 3
```javascript
var o = Imjs.Map({"a": 1,"b": 2});
// Map {"a": 1, "b": 2}
var o2 = o.set("a", 3);
// Map {"a": 3, "b": 2}
o.get("a");    // 1
o2.get("a");   // 3
```
mori

A library for using ClojureScript's persistent data structures and supporting API from the comfort of vanilla JavaScript.

Rationale

JavaScript is a powerful and flexible dynamic programming language with a beautiful simple associative model at its core. However this design comes at the cost of ubiquitous mutability. Mori embraces the simple associative model but leaves mutability behind. Mori delivers the following benefits to JavaScript:

- Efficient immutable data structures - no cloning required
- Uniform iteration for all types
- Value based equality

Modern JavaScript engines like V8, JavaScriptCore, and SpiderMonkey deliver the performance needed to implement persistent data structures well.

Immutability

Mori delivers highly tuned persistent data structures based on the ones provided in Clojure. When using Mori data structures and operations you do not need to defensively clone as you often do in JavaScript. By providing immutable data structures, Mori encourages value oriented programming.
```javascript
var mori = require("mori");

var a = mori.vector(1, 2);  // [1 2]
var a2 = mori.conj(a, 3);  // [1 2 3]

mori.count(a);  // 2
mori.get(a2, 2);  // 3
```
var o = mori.hashMap("a", 1, "b", 2);
// {"a": 1, "b": 2}
var o2 = mori.assoc(o, "a", 3);
// {"a": 3, "b": 2}

mori.get(o, "a"); // 1
mori.get(o2, "a"); // 3
Immutable.js

facebook.github.io/immutable-js

- JavaScript all the way
- Object-oriented API
- Smaller

Mori

swannodette.github.io/mori

- ClojureScript under the hood
- Functional API
- Faster
LIVE LONG AND DON'T MUTATE!
THANKS!

@AnjanaVakil

Icons by EmojiOne
Template by SlidesCarnival

Inspiration/curiosity by Sal Becker & Recurse Center

Conference by O’Reilly
REFERENCES & MORE

J.N. L'orange, **UNDERSTANDING CLOJURE'S PERSISTENT VECTORS**, Blog series hypirion.com/musings/understanding-persistent-vector-pt-1

P. Bagwell, **IDEAL HASH TREES**, 2001 lampwww.epfl.ch/papers/idealhashtrees.pdf


D. Spiewak, **EXTREME CLEVERNESS: FUNCTIONAL DATA STRUCTURES IN SCALA**, Strange Loop 2011 infoq.com/presentations/Functional-Data-Structures-in-Scala

M. Thatte, **WHAT LIES BENEATH: A DEEP DIVE INTO CLOJURE’S DATA STRUCTURES**, EuroClojure 2015 youtu.be/7BFF50BHPPo

D. Nolen, **IMMUTABILITY, INTERACTIVITY & JAVASCRIPT**, FutureJS 2014 youtu.be/mS264h8KGwk

L. Byron, **IMMUTABLE DATA & REACT**, React.js Conf 2015 youtu.be/I7IdS-PbEgI