Immutable Collections

Michael J. Steindorfer

mail: michael@cwi.nl
twitter: @loopingoptimism
“A collection represents a group of objects [...]”

–java.util.Collection API
Data Type Semantics
   lists, sets, maps, multi-maps, bags, ...

Ordering
   unordered (hashed), ordered

Update Semantics
   mutable, transient, immutable

Processing Semantics
   sequential, concurrent, parallel

Encoding
   array/hashtable, linked list, tree, trie, ...

Content
   homogeneous, heterogeneous

...
Data Type Semantics
   sets, maps, multi-maps

Ordering
   unordered (hashed)

Update Semantics
   immutable

Processing Semantics
   sequential

Encoding
   trie

Content
   homogeneous, heterogeneous

...
Why Immutability?

• More Optimizations (Constant)
• Simple (Programming Model)
• Robust (Concurrency)
'Classes should be immutable unless there’s a very good reason to make them mutable.'

Item 15: Minimize mutability
“[Collections] should be immutable unless there’s a very good reason to make them mutable.”

Item 15: Minimize mutability
“Immutable [collections] provide many advantages, and their only disadvantage is the potential for performance problems under certain circumstances.”

  Item 15: Minimize mutability
Immutable collections are **challenging to optimize**, but overall they **constitute a more sensible default**.

–Personal Opinion
Outline

1. Core Principles & Encodings
2. Minimize Memory Footprint
3. Efficiency & Expressivity
4. Performance Challenges
Core Principles
Set<Integer> updatedSet =
    hugeSet.newInstanceWith(34);

Set<Integer> __tmp =
    new HashSet<>(hugeSet.size + 1);
__tmp.addAll(hugeSet);
__tmp.add(34);
Set<Integer> updatedSet =
    Collections.unmodifiableSet(__tmp);
Clojure
http://clojure.org

Rascal
http://rascal-mpl.org

Scala
http://scala-lang.org
Hash Array Mapped Trie
tree node as array

Prefix of Hash Code

Prefix of Hash Code
\{2, 32, 34, 898, 2942, 4098, 6014\}
Fig. 2: Examples for presentation.
class Node {
    int bitmap;
    Object[] content;
}
What can you expect?

- Shallow and Wide Search Trees ($\text{Depth} \leq 7$)
- Lookup/Insert/Delete in $O(\log_{32}(n)) \approx O(1)$
- Union/Subset (Structural Operations)
Minimize Memory Footprint
by Specializing Class Layouts
class Node {
    int bitmap;
    Object[] content;
}

Fig. 2: Examples for presentation.
```java
class ArrayList32 {
    /* 0 <= content.size <= 32 */
    Object[] content;
}
```
class ArrayList32 {
  Object[] content;
}

class List1 implements ImmutableList {
  Object slot0;
}

class List2 implements ImmutableList {
  Object slot0;
  Object slot1;
}

class List3 implements ImmutableList {
  Object slot0;
  Object slot1;
  Object slot2;
}

...
😊 Small Footprints
😊 Megamorphic Call Sites
😊 Code Bloat
abstract class List2 implements ImmutableList {
    Object slot0;
    Object slot1;

    @Override
    public ImmutableList insertAtIndex(int index, Object item) {
        switch (index) {
            case 0:
                return new List3(item, slot0, slot1);
            case 1:
                return new List3(slot0, item, slot1);
            case 2:
                return new List3(slot0, slot1, item);
            default:
                throw new IllegalArgumentException();
        }
    }
}
😊 Small Footprints
😊 Monomorphic Call Sites
😊 JIT-Friendly
Decoupling Data and Operations
abstract class List2 implements ImmutableList {
    Object slot0;
    Object slot1;
}

// viewing consecutive fields as arrays
createArrayView(List2.class,
        "slot0", "slot1", Object.class);
ImmutableList insertAtIndex(int index, Object item) {
    ArrayView<Object> src = getArrayView();
    ImmutableList newList = allocateList(src.length + 1);
    ArrayView<Object> dst = newList.getArrayView();

    // copy 'src' and insert element at position 'index'
    arrayviewcopy(src, 0, dst, 0, index);
    dst.set(index, item);
    arrayviewcopy(src, index, dst, index + 1, src.length - index);

    return newNode;
}
Efficiency & Expressivity
class Node {
    int bitmap;
    Object[] content;
}
More Performant ...
Up to 6x faster …

(speedup of iteration over Clojure / Scala)

... up to 28x faster ...

(speedup equality checking over Clojure / Scala)

... and up to 64% smaller.

(memory footprint reduction over Clojure / Scala)

Fig. 2: Examples for presentation.
Fig. 2: Examples for presentation.
class Node {
    int datamap;
    int nodemap;
    Object[] content;
}
Heterogeneous Data Structures for the Masses

- 0-Key: A
- 0-Val: 2
- 1-Key: B
- 1-Val: 34
- 4-Key: C
- 4-Val: 32
- 28-Key: D
- 28-Val: 1
- 24-Ref: E
- 2-Ref: F
- 0-Ref: G
- 5-Key: H
- 5-Val: 2

Example for True'15 Talk

Example for OOPS LA'15 Talk
More Expressive …
Numeric Data Sets

Storing and Processing with Collections
SPECIALIZED
PRIMITIVE
COLLECTION

data set of
int (32-bit)
data set of BigInteger (Object)

GENERIC COLLECTION
PRIMITIVE COLLECTION

data set of int (32-bit)
GENERIC COLLECTION

- data set of `Integer` (Boxed)
- data set of `BigInteger` (Object)
HETEROGENEOUS COLLECTIONS

data set of int (32-bit)

data set of BigInteger (Object)
Fig. 2: HAMT-based sets with values in internal nodes versus values at the leaves only.
```java
class Node {
    int datamap;
    int nodemap;
    Object[] content;
}
```
```java
class Node {
    BitVector bitmap;
    Object[] content;
}
```
class Node {
    BitVector bitmap;
}

Orthogonal To:
- Scala’s Union Types
- Valhalla’s’s Primitive Generics
Performance Challenges
Trees (as fast) as Arrays?

Improving Locality between Nodes.
Trees (as fast) as Arrays!
Memory Management & Garbage Collection
Summary
Heterogeneous Collections

class ArrayList32 {
    Object[] content;
}

class List1 implements ImmutableList {
    Object slot0;
}

class List2 implements ImmutableList {
    Object slot0;
    Object slot1;
}

class List3 implements ImmutableList {
    Object slot0;
    Object slot1;
    Object slot2;
}
...

Fig. 2: Examples for presentation.
Map<String, Long> wordCount = phrases.stream()  
    .flatMap(toWordStream)  
    .filter(word -> word.length() > 0)  
    .map(word -> new Tuple<>(word, 1L))  
    .collect(groupingBy(Tuple::getKey, counting()));
MISSING: IMMUTABLE COLLECTIONS

Map<String, Long> wordCount = phrases.stream()
    .flatMap(toWordStream)
    .filter(word -> word.length() > 0)
    .map(word -> new Tuple<>(word, 1L))
    .collect(groupingBy(Tuple::getKey, counting()));
usethesource/capsule
The Capsule Hash Trie Collections Library
Immutable Collections

Michael J. Steindorfer

mail: michael@cwi.nl
twitter: @loopingoptimism