Clojure Collections

Looking Deep Inside Clojure Data Collections

by Steven Reynolds

Steven explains the benefits of immutability and explores how Clojure's data collections handle it.

Clojure embraces a functional programming style, controlling mutability tightly. Unless you take special steps to permit it, data collections in Clojure are not mutable; they cannot be changed.

Why bother with immutability? Clojure does so for two key reasons. Having a network of references to mutating objects is fundamentally very complex. Complexity is the enemy of software development. Secondly, such a network is exquisitely difficult to make correct while allowing concurrency.

In the familiar imperative or Object Oriented programming styles, when a structure is updated, it is mutated. The application holds a stable reference to a collection and the collection itself is changed. Clojure instead generally uses collections that cannot be changed; the update operations return a new version of the collection. Whenever a version of a collection is created, that version of the collection must remain accessible (because it cannot be changed). Hence these type of collections are sometimes called *persistent*. Of course, old versions of the collection can be garbage collected when there are no references to them.

The interesting challenge is to ensure that, when a new collection must be returned, Clojure doesn't need to copy the entire collection. Excessive copying causes performance degradation.

Lists

Clojure contain a fairly classical representation for lists. The next code sample creates some lists and exports a graph of each of them.

```
(defn list-ex []
  (let [w '(1 2 3)         ; Figure 1
        x (rest a)         ; Figure 2
        y (conj  a '(3 4)) ; Figure 3
        z (cons 1 a)       ; Figure 4
        lsaver (PersistentListSaver.)
        csaver (ConsSaver.)]
    (. lsaver save w "list_before.dot")
    (. lsaver save x "list_after_rest.dot")
    (. lsaver save y "list_after_conj.dot")
    (. csaver save z "list_after_cons.dot")
    ))
```

The original list, w, is created in the let form and contains the elements 1, 2, and 3. It is shown in Figure 1. The second list, x, is the rest of w. This is all of w except its head element (1 in this case).

PersisentListSaver is a Java class that uses reflection to dump the Clojure list to a graph. This graph is then drawn using GraphViz.

If you look at the list in Figure 1, you can see the representation has a first that contains the head of the list, and a rest that contains other elements.

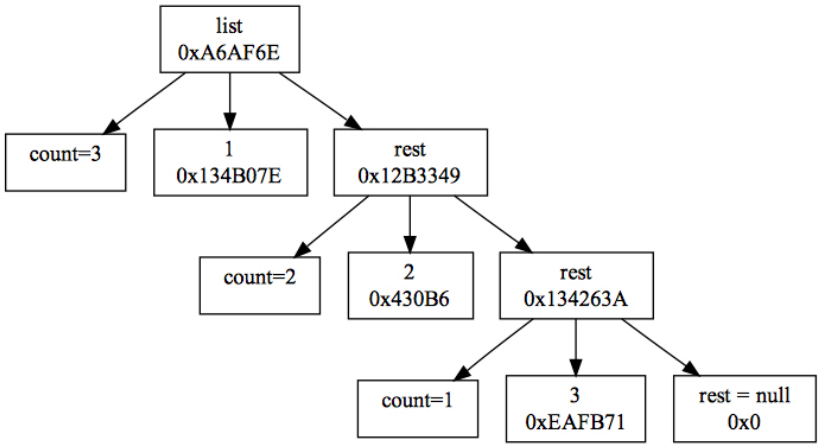

Figure 1: Clojure List

As you can see in Figure 2, when Clojure takes the rest of the original list, it doesn't need to copy any data. The hex numbers in the graph nodes are the result from calling System.identityHashCode on the node. The Java contract of this hash code is that it will always return the same number for the same Object. The usual JDK implementation is to return the address of the Object.

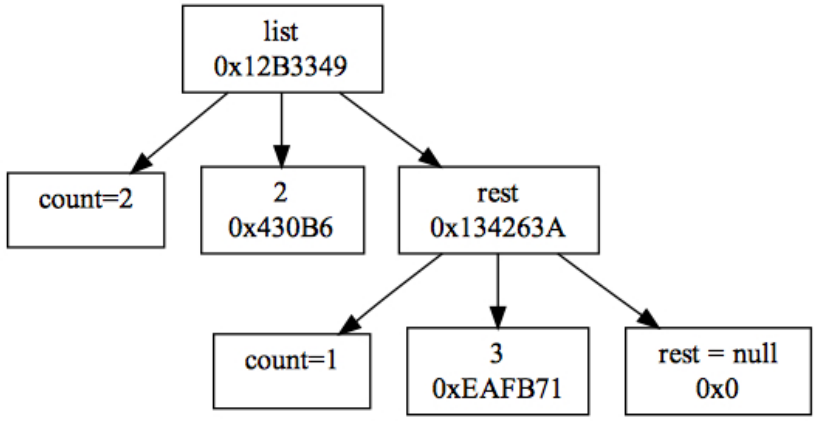

Figure 2: Clojure List from Figure 1 after rest applied

The graphs are simplified: some details left out, and element nodes are shown inline rather than with a separate node. In reality primitives are not stored in Clojure (or Java) collections. They're always storing a boxed Object. That boxing is left out of the graphs for greater clarity and to save paper.

If you add elements to a list, Clojure also arranges to share data. The code sample adds 3 and 4 to the list w using conj and saves this new list in y. The conj function adds elements to the collection at the most efficient location.

If you compare Figure 3 to Figure 1, you can see that conj added the elements at the front. Again, all the preexisting elements are shared.

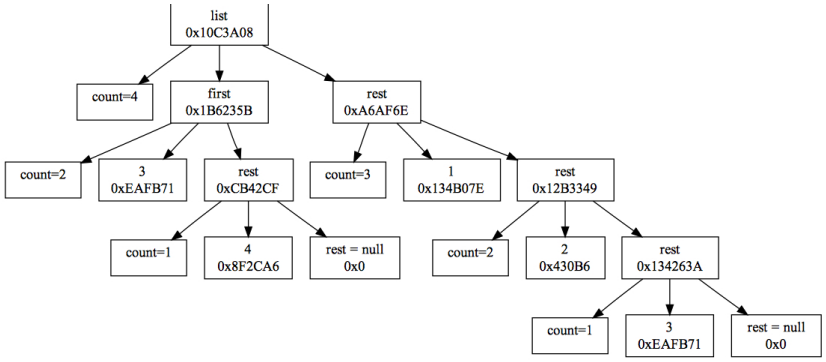

Figure 3: Clojure List from Figure 1 after conj applied

When you add elements to a list using the cons function (list z), Clojure creates a mixed structure with a Cons cell, a first element, and an ISeq, as shown in Figure 4. This shares elements, but future operations will not be quite as efficient. The cons function is specified to always add to the front of any data collection.

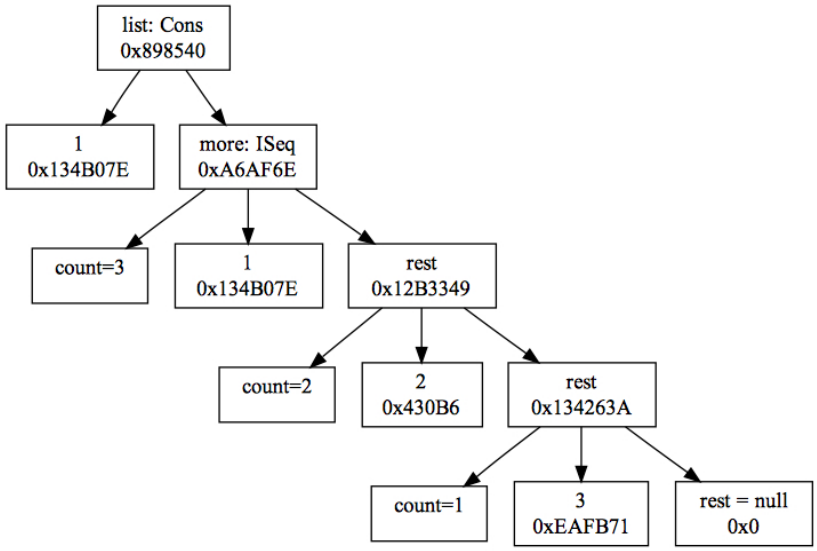

Figure 4: Clojure List from Figure 1 after cons applied

Maps

Another very important type of collection is maps. The next example shows a small map.

```
(defn amap-assoc []
  (let [x '{1 "one" 2 "two"}        ; Figure 5
        y (assoc  x 3 "three")      ; Figure 6
        pamsaver (PersistentArrayMapSaver.)]
    (. pamsaver save x "amap_before_assoc.dot")
    (. pamsaver save y "amap_after_assoc.dot")
    ))
```

In Clojure, small maps are stored in a simple structure backed by an array that contains the keys and values interleaved. You can see this in Figure 5. Lookups are done by a linear scan of the array. That's fast when the map is small. If you add an element to the map (use the function assoc), Clojure just makes a new map without sharing (Figure 6).

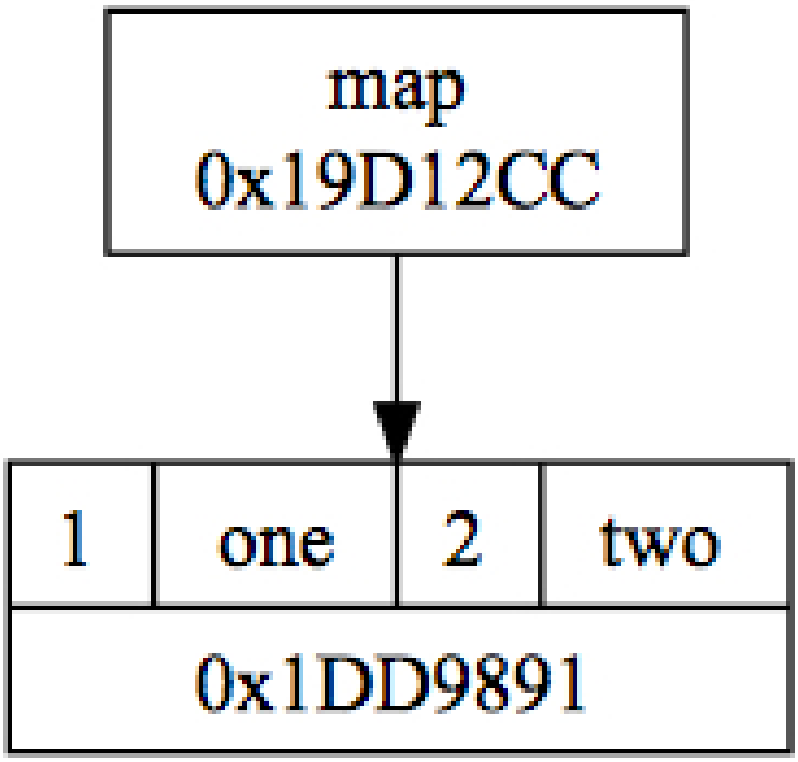

Figure 5: Clojure Array Map

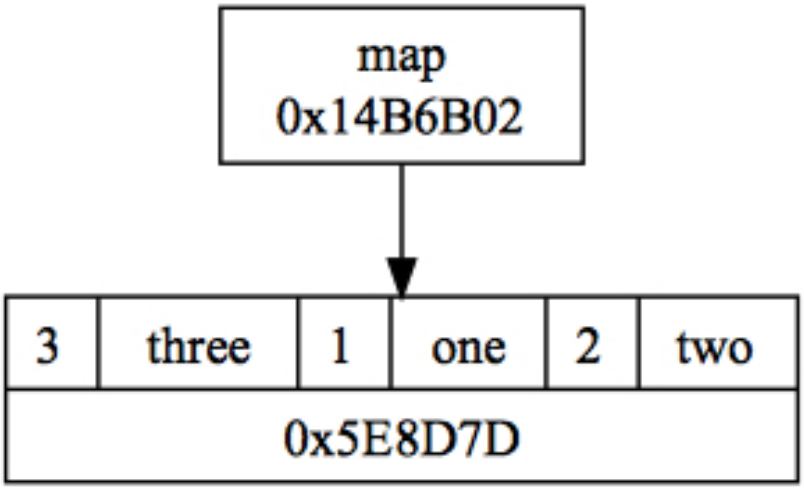

Figure 6: Clojure Array Map from Figure 5 after assoc applied

Longer maps are stored in a PersistentHashMap. They stored as a tree with a 32-way branching factor. PersistentHashMap uses a sparse array to store data at each tree level. The bitmap indicates which elements of the nominal full array are actually present in the realized sparse array.

You can read more about Clojure's PersistentHashMap at the Wikipedia article "Hash array mapped trie [U1]" and at the blog [U2] by arcanesentiment. The Clojure implementation is based on a paper [U3] by Phil Bagwell. This data structure is technically a trie rather than a tree because the keys can be variable length, for example a String.

The next example shows a small hash map.

```
(defn hmap-assoc []
  (let [x (hash-map 1 "one" 2 "two")      ; Figure 7
        y (assoc  x 3 "three")            ; Figure 8
        phmsaver (PersistentHashMapSaver.)]
    (. phmsaver save x "hmap_before_assoc.dot")
    (. phmsaver save y "hmap_after_assoc.dot")
    ))
```

In Figures 7 and 8, you can see the maps before and after an element is added. In this example, only one level of the trie is needed. If you look at Figure 8, you can see that the new map is not able to share any elements with the old map.

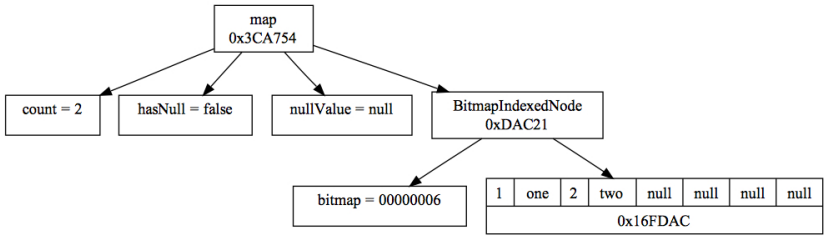

Figure 7: Clojure Hash Map

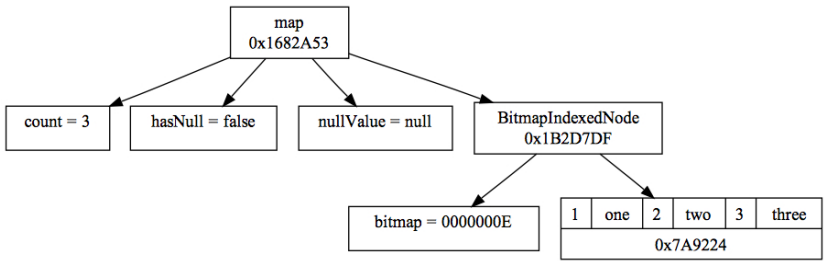

Figure 8: Clojure Hash Map from Figure 9 after assoc applied

Figure 9 shows a more realistic larger hash map that has 17 elements. I removed most of the nodes so that the figure will fit on the page. With a 32-way branching factor, this map instance only needs two levels to store its data; two levels create room for 32*32 elements.

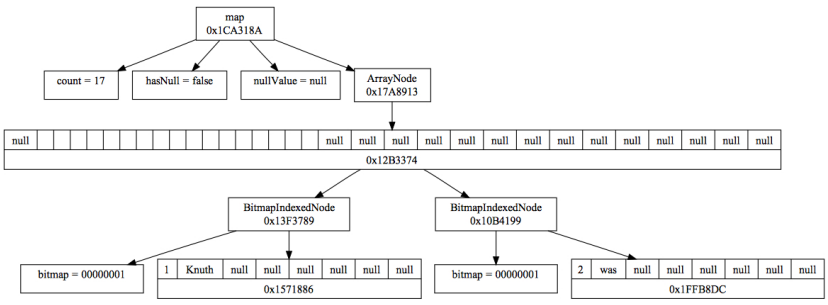

Figure 9: Larger Clojure Hash Map

If you add the key/value {18 "physics"} to this larger map, you get Figure 10. This figure shows that most of the hash map was shared. The new node is of course different, also the nodes above the new node are necessarily different. This technique is well known and is called path copying.

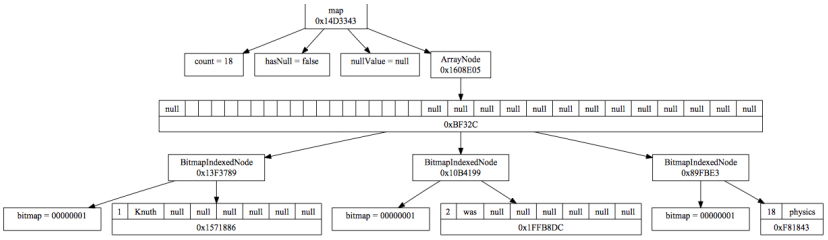

Figure 10: Clojure Hash Map from Figure 9 after assoc applied

Transient Collections

Clojure also has transient versions of the collections. These collections are allowed to change, and so can be faster in some situations. In particular, a transient is used when a collection is built up from another collection. That transient is converted to persistent when it is returned. That approach provides a nice speedup in a way that is invisible to the application code.

Conclusion

I have shown the internal representation of some Clojure collections; I believe that it is very illuminating to see the backing data for objects. It is like a physician using an MRI to see the internals of their patient. The graphs do show internal representations, and these representations will likely change with future versions of Clojure. You should not depend on the details. These graphs were made with Clojure 1.2.

The Clojure data collections are very sophisticated, and play a key role in how Clojure can both avoid mutability and also have excellent performance.

About the Author

xxxxxxxxxxxxxxx.

Send the author your feedback [U4] or discuss the article in the magazine forum [U5].

External resources referenced in this article:

[U1] http://en.wikipedia.org/wiki/Hash_array_mapped_trie

[U2] http://arcanesentiment.blogspot.com/2008/08/array-mapped-hash-tries-and-nature-of.html

[U3] http://lampwww.epfl.ch/papers/idealhashtrees.pdf

[U4] mailto:michael@pragprog.com?subject=collections

[U5] http://forums.pragprog.com/forums/134