Hash Tables, Mutability, and Identity: How to Implement a Bi-Directional Hash Table in Java

Mutability can affect the behavior of hash tables. Misunderstanding mutability can cause objects stored in a hash table to become irretrievable and essentially disappear.

by Wayne Citrin MVB · **Oct. 07, 16 · Java Zone · Tutorial**

This article is featured in the new DZone Guide to Modern Java, Volume II. Get your free copy for more insightful articles, industry statistics, and more.

Data structures like hash tables and hash maps are essential for storing arbitrary objects and efficiently retrieving them. The object, or value, is stored in the table and associated with a key. Using the key, one can later retrieve the associated object. However, there are situations where, in addition to mapping keys to objects, one might also want to retrieve the key associated with a given object. This problem has been encountered time and again, and the solution is much trickier.

For example, we must efficiently keep track of the underlying remote objects corresponding to proxies when developing our JNBridge product. We do that by maintaining a table mapping unique keys to objects. This is straightforward to implement, using hash tables and maps available in many libraries. We also must be able to obtain an object's unique key given the object — provided it has one. (Yes, some libraries now offer bimaps, which purport to implement this functionality, but these didn't exist when we first needed this, so we had to implement it on our own. In addition, these bimaps don't provide everything we need. We'll discuss these issues later.)

Following are some helpful hints in implementing these forward/reverse map pairs.

Hashing Algorithm Requirements

Let's first review the requirements that must be fulfilled by whatever hashing algorithm is used, and how it relates to equality. While a hashing method can be implemented by the developer, it is expected to obey a contract. The contract, in general, cannot be verified by the underlying platform, but if it is not obeyed, data structures that rely on the hashing

method may not behave properly.

In Java, for example, the contract of the hashCode() method is:

- For any given object, the hashCode() method must return the same value throughout execution of the application (assuming that information used by the object's equality test also remains unchanged).

- If two objects are equal according to the objects' equality test, then they must both have the same hash code.

Connected with the hashCode() method is an equality test, which should implement an equivalence relation; that is, it must be reflexive, symmetric, and transitive. It should also be consistent: It should yield the same result throughout the execution of the application (again, assuming the information used by the equality test doesn't change).

In addition to these contracts, Java provides guidelines for the use and implementation of hash codes, although these are not binding and may just be advisory. While guidelines for some non-Java frameworks suggest that hash codes for mutable objects should only be based on aspects of the objects that are immutable — so that the hash codes never change — in Java culture, the guidelines for implementing hashCode() are less strict, and it is quite likely that an object's hash code can change. For example, the hash code of a hash table object can change as items are added and removed.

However, one informal guideline suggests that one should be careful using an object as a key in a hash-based collection when its hash code can change. As you'll see, the potential mutability of hash codes is a crucial consideration when implementing reverse maps.

If an object's hash code changes while it's stored inside a data structure that depends on the hash code (for example, if it's used as a key in a hash table), then the object may never be retrieved, as it will be placed in one hash bucket as it's added to the hash table, but looked for later in another hash bucket when the hash code has changed.

Implementation Assumptions

So, what does this mean when forward/reverse map pairs must be implemented? Let's start with some assumptions:

- The forward map maps from keys to values. The keys are a particular type that one chooses in advance, and values can be any object. The value objects can be implemented by anyone, and their hash codes and equality tests may not conform to implementation guidelines — they may not even obey the required contract.

- The reverse map maps from the user-provided values back to keys.

- For simplicity, the user-defined object cannot be null. (Although, if necessary, this can be accommodated, too, through some additional tests.)

- Keys and user-defined objects should be unique; that is, they should not be used in the tables more than once.

Since the key is under our control, a developer can use objects of any immutable class (for example, Integer or Long), and avoid the possibility that the key's hash code changes. For the forward map, a simple hash table or hash map can be used.

The reverse map is more of a problem. As discussed above, one cannot rely on the hash code method associated with the user-provided objects that are used as keys. While some classes, particularly Java base classes, may have well-behaved hash code functions, other Java classes might not. Therefore, it's unsafe to trust that user-defined classes will obey these rules, or that the programmers that defined them were even aware of these guidelines.

Find an Immutable Attribute

Therefore, one must come up with an attribute that every object has, that is guaranteed to never change, even when the contents of the object do change. The attribute's value should also be well-distributed, and therefore suitable for use in hashing. One such immutable attribute is the object's identity. When a hash table, for example, has items added to it, it's still the same hash table, even though the computed hash code might change.

Fortunately, Java provides a hashing method based on object identity, `java.lang.System.identityHashCode()`, which is guaranteed to return the same value on a given object, even if its state has changed, since it is guaranteed to use java.lang.Object's hashCode() method, which is identity-based, even if the target class overrides that method.

Java actually provides a class, `java.util.IdentityHashMap`, which uses identity hashing. The developer could have used IdentityHashMap for his reverse hash table, except that, unlike the Java Hashtable, IdentityHashMap is not synchronized, and hash tables must be thread-safe.

Creating Identity-Based Hashing

In order to write one's own identity-based hash tables, the developer must first ensure that, no matter what object is used as the key, identity-based hashing is always used. Unfortunately, the hash methods for these classes can't be overridden, since they're out of the developer's control. Instead, the key objects must be wrapped in classes that are in the developer's control, and where he can control the way the hash values are computed. In these wrappers, the hash method simply returns the identity-based hash code of the wrapped object.

In addition, since identity-based hashing is being used, one must also use reference-based equality, so that two objects are equal if — and only if — they're the same object, rather than simply equivalent objects. In Java, a developer must use the "==" operator, which is guaranteed to be reference equality, rather than equals(), which can be, and often is, redefined by the developer.

In Java, our identity-based wrappers look like this:

```
final class IdentityWrapper
```

```
final class IdentityWrapper
{
    private Object theObject;
    public IdentityWrapper(Object wrappedObject)
    {
        theObject = wrappedObject;
    }

    public boolean equals(Object obj2)
    {
        if (obj2 == null) return false;
        if (!(obj2 instanceof IdentityWrapper)) return false;
        return (theObject == ((IdentityWrapper)obj2).theObject);
    }

    public int hashCode()
    {
        return System.identityHashCode(theObject);
    }
}
```

Once these wrappers have been defined, the developer has everything he needs for a reverse hash table that works correctly:

```
Hashtable ht = new Hashtable();
…
ht.put(new IdentityWrapper(mutableUserDefinedObject), value);
…
mutableUserDefinedObject.modify();
…
Value v = (Value) ht.get(new
IdentityWrapper(mutableUserDefinedObject));
// retrieved v is the same as the value that was initially added.
```

If the IdentityWrapper classes are not used, the ht.get() operation is not guaranteed to retrieve the proper value. At this point, the developer has all that's needed to implement bidirectional hash tables.

The Trouble With Other Libraries

What about other existing libraries? In particular, what about Google's Guava library, which implements a HashBiMap class, as well as other classes implementing a BiMap interface? Why not use that, and avoid reinventing the wheel? Unfortunately, while HashBiMap implements a forward/reverse hashmap pair and makes sure that the two are always in sync, it does not use identity hashing, and will not work properly if one of the keys or values is a mutable object.

This can be seen by examining the HashBiMap source code. So, while HashBiMap solves part of the problem of implementing forward/reverse hashmap pairs, it does not address another, arguably more difficult part: the problem of storing mutable objects. The approach described here solves that issue.

In Conclusion

This piece discusses an important, but unfortunately somewhat obscure, issue in the implementation of hash tables: the way in which mutability can affect the behavior of hash tables, and the way in which misunderstanding the issue can cause objects stored in a hash table to become irretrievable and essentially disappear.

When these issues are understood, it becomes possible to implement hash tables where any object, no matter how complex or mutable, can be used as a key, and where bi-directional hash tables can be easily created.

More Java Goodness

For more insights on Jigsaw, reactive microservices, and more get your free copy of the new DZone Guide to Modern Java, Volume II!

And if you want to see other articles in the guide, check out:

- The Java 8 Design Principles
- Project Jigsaw Is Coming
- Reactive Microservices: Driving Application Modernization Efforts
- The Elements of Modern Java Style

- A Java Developer's Guide to Migration

Like This Article? Read More From DZone

Create DynamoDB Tables With Java

Diving Into Scala Maps

Collision Based Hashing Algorithm Disclosure

Free DZone Refcard

Getting Started With Kotlin

Topics: JAVA , HASH TABLE , MUTABILITY , ALGORITHM

Java Partner Resources

Red Hat Developer Program

Predictive Analytics + Big Data Quality: A Love Story

Melissa Data