PROGRAMMING JAVA

A Look at Hashmaps in Java

01 Nov 2019 · 6 mins read

A Look at Hashmaps in Java

Note: these implementation details are correct for OpenJDK8. Most other versions of Java should be generally the same, but might have minor differences

This post is going to discuss the under-the-hood implementation of the Java Hashmap. If you have never used a hashmap before, this post is probably not for you.

How it works: the basics

The basics of how the hashmap works are not too complex. A hashmap is made up of some number of bins which will store the items. When a key and value are inserted into the hashmap, the key is hashed, and then this hash value is used to figure out which bin the key-value pair will go in. Each bin contains a linked list containing all of its items. So, to retreive an item, the key is hashed, which tells us which bin to look in. Then, we look through the linked list in the bin until we find a bin that matches the key.

Sizes and resizes

The hashmap is often discussed as an O(1) lookup data structure. This is true in the average case, but to keep the lookup fast, we need to make sure the bins don't get too large. After all, each bin contains a linked list which is O(n) lookup. To do this, the hashmap will resize at a certain point. When the number of bins grow, the number of hash collisions will shrink and the number of items per bin will shrink, so our lookups are faster. This is a time vs space tradeoff - the more bins you have, the faster the lookup, but the more space you are wasting on potentially empty bins.

When a hashmap is created, two important properties are decided: the capacity and the load factor. The capacity is the number of bins, which defaults to 16. The load factor is how full the hashmap is allowed to be before it's resized. This defaults to 0.75, so when there are 3/4 as many entries (key-value pairs) in the hashmap as the capacity, the capacity will be doubled.

This doubling is calling resizing and is triggered when the load factor is exceeded as described above. The size is always doubled, except in special cases like when the size would exceed `Integer.MAX_VALUE` (in which case your hashmap is way too big!). It's important to note that the hashmap won't resize back down once items are removed.

The hash function

The hash function is extremely important to ensure that hashmap operations are fast. If the hash function is not very random, we may end up with a lot of items in the same bin, which degrades the O(1) lookup performance of the hashmap.

The implementation

This is all in pretty abstract terms, so let's examine the data structures actually used to implement the hashmap. First, we have the `Node` inner class, which contains the key, value, key hash, and a reference to the next node in the bin (which is null if it's the last node).

Our hashmap is then implemented as an array (not ArrayList) of nodes (which represent bins). To actually determine which bin to use given the hash, a *bitwise and* is applied to the hash and the index of the last bin. This is clever because it's an extremely fast operation (faster than modulo, for example). This works well because `a & b` is always less or equal to both `a` and `b` when they're both positive, so we'll never go out of bounds. There is a chance that the hash will be negative, but as long as the last index is greater or equal to zero (which it always will be), the result will be between 0 and the last index.

During insertion:

- we use the hash to determine which bin to use
- if there are no nodes in the bin, create one with the current key, value, and hash
- if are nodes in the bin, check to see if any of them share our hash
    - if they do, check to see if the key is equal to our key either using `==` or `.equals` (either one can be true)
    - if we've determined an identical key is already in the bin, replace the value there
    - if not, add our own node to the end of the list (and potentially convert to tree, see below section)
- if we added a new node, resize the hashmap if we've now exceeded the load capacity

During lookup:

- we use the hash to determine which bin to use
- if there are no nodes in the bin, return `nul`
- if there are nodes in the bin, first check to see if any share our hash
    - if they do, check to see if the key is equal to our key either using `==` or `.equals` (either one can be true)
    - if we've determined that the key is a match, we can return that value, otherwise we keep looking and return null if we don't find a match

Trees

I previously mentioned that each node is actually a linked list. That's true in general, but in some special cases, a tree is used. While a tree (and balancing it) involves more initial overhead, it's better for a large number of items because lookup is only O(log n) rather than O(n) for a linked list.

There are variables called `TREEIFY_THRESHOLD`, `UNTREEIFY_THRESHOLD`, and `MIN_TREEIFY_CAPACITY` which control when a node is converted to or from a tree. `TREEIFY_THRESHOLD` is set to 8, which means that if there are eight or more items in a bin, it is worth the overhead to create and manage a tree, so that is what happens. Likewise, when there are fewer than `UNTREEIFY_THRESHOLD` items, which is set to 6, our tree will be converted back to a linked list.

`MIN_TREEIFY_CAPACITY` is the minimum capacity of the entire hashmap in order to start using trees. It is set to 64. That means that if the entire hashmap has less than 64 bins, the hashmap will be resized rather than using any trees.

Further reading

This post certainly didn't cover every part of this awesome data structure. If you want to learn more about the implementation, you can take a look at the source code. Again, the implementation details are implementation-specific and you shouldn't rely on them, but they are useful to know and can help paint a more complete picture of how hashmaps work.

Previous
Deploying a Personal Knowle...