1 Class ConcurrentHashMap<K,V>

1.1 javaDoc

java.lang.Object
java.util.AbstractMap<K,V>
java.util.concurrent.ConcurrentHashMap<K,V>

All Implemented Interfaces:
Serializable, ConcurrentMap<K,V>, Map<K,V>

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as Hashtable, and includes versions of methods corresponding to each method of **Hashtable**. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details.

Retrieval operations (including get) generally do not block, so may overlap with update operations (including put and remove). Retrievals reflect the results of the most recently completed update operations holding upon their onset. (More formally, an update operation for a given key bears a happens-before relation with any (non-null) retrieval for that key reporting the updated value.) For aggregate operations such as putAll and clear, concurrent retrievals may reflect insertion or removal of only some entries. Similarly, Iterators, Spliterators and Enumerations return elements reflecting the state of the hash table at some point at or since the creation of the iterator/enumeration. They do not throw ConcurrentModificationException. However, iterators are designed to be used by only one thread at a time. Bear in mind that the results of aggregate status methods including size, isEmpty, and containsValue are typically useful only when a map is not undergoing concurrent updates in other threads. Otherwise the results of these methods reflect transient states that may be adequate for monitoring or estimation purposes, but not for program control.

The table is dynamically expanded when there are too many collisions (i.e., keys that have distinct hash codes but fall into the same slot modulo the table size), with the expected average effect of maintaining roughly two bins per mapping (corresponding to a 0.75 load factor threshold for resizing). There may be much variance around this average as mappings are added and removed, but overall, this maintains a commonly accepted time/space tradeoff for hash tables. However, resizing this or any other kind of hash table may be a relatively slow operation. When possible, it is a good idea to provide a size estimate as an optional initialCapacity constructor argument. An additional optional loadFactor constructor argument provides a further means of customizing initial table capacity by specifying the table density to be used in calculating the amount of space to allocate for the given number of elements. Also, for compatibility with previous versions of this class, constructors may optionally specify an expected concurrencyLevel as an additional hint for internal sizing. Note that using many keys with exactly the same hashCode() is a sure way to slow down performance of any hash table. To ameliorate impact, when keys are Comparable, this class may use comparison order among keys to help break ties.

A **Set** projection of a **ConcurrentHashMap** may be created (using **newKeySet()** or **newKeySet(int)**), or viewed (using **keySet(Object)** when only keys are of interest, and the mapped values are (perhaps transiently) not used or all take the same mapping value.

A ConcurrentHashMap can be used as scalable frequency map (a form of histogram or multiset) by using LongAdder values and initializing via computeIfAbsent. For example, to add a count to a ConcurrentHashMap<String,LongAdder> freqs, you can use freqs.computeIfAbsent(k -> new LongAdder()).increment();

This class and its views and iterators implement all of the optional methods of the Map and Iterator interfaces.

Like Hashtable but unlike HashMap, this class does not allow null to be used as a key or value.

ConcurrentHashMaps support a set of sequential and parallel bulk operations that, unlike most Stream methods, are designed to be safely, and often sensibly, applied even with maps that are being concurrently updated by other threads; for example, when computing a snapshot summary of the values in a shared registry. There are three kinds of operation, each with four forms, accepting functions with Keys, Values, Entries, and (Key, Value) arguments and/or return values. Because the elements of a ConcurrentHashMap are not ordered in any particular way, and may be processed in different orders in different parallel executions, the correctness of supplied functions should not depend on any ordering, or on any other objects or values that may transiently change while computation is in progress; and except for forEach actions, should ideally be side-effect-free. Bulk operations on Map.Entry objects do not support method setValue.

- forEach: Perform a given action on each element. A variant form applies a given transformation on each element before performing the action.
- search: Return the first available non-null result of applying a given function on each element; skipping further search when a result is found.
- reduce: Accumulate each element. The supplied reduction function cannot rely on ordering (more formally, it should be both associative and commutative). There are five variants:
  - Plain reductions. (There is not a form of this method for (key, value) function arguments since there is no corresponding return type.)
  - Mapped reductions that accumulate the results of a given function applied to each element.
  - Reductions to scalar doubles, longs, and ints, using a given basis value.

These bulk operations accept a parallelismThreshold argument. Methods proceed sequentially if the current map size is estimated to be less than the given threshold. Using a value of $Long.MAX\_VALUE$ suppresses all parallelism. Using a value of 1 results in maximal parallelism by partitioning into enough subtasks to fully utilize the ForkJoinPool.commonPool() that is used for all parallel computations. Normally, you would initially choose one of these extreme values, and then measure performance of using in-between values that trade off overhead versus throughput.

The concurrency properties of bulk operations follow from those of ConcurrentHashMap: Any non-null result returned from get(key) and related access methods bears a happens-before relation with the associated insertion or update. The result of any bulk operation reflects the composition of these per-element relations (but is not necessarily atomic with respect to the map as a whole unless it is somehow known to be quiescent). Conversely, because keys and values in the map are never null, null serves as a reliable atomic indicator of the current lack of any result. To maintain this property, null serves as an implicit basis for all non-scalar reduction operations. For the double, long, and int versions, the basis should be one that, when combined with any other value, returns that other value (more formally, it should be the identity element for the reduction). Most common reductions have these properties; for example, computing a sum with basis 0 or a minimum with basis MAX_VALUE.

Search and transformation functions provided as arguments should similarly return null to indicate the lack of any result (in which case it is not used). In the case of mapped reductions, this also enables transformations to serve as filters, returning null (or, in the case of primitive specializations, the identity basis) if the element should not be combined. You can create compound transformations and filterings by composing them yourself under this ”null means there is nothing there now” rule before using them in search or reduce operations.

Methods accepting and/or returning Entry arguments maintain key-value associations. They may be useful for example when finding the key for the greatest value. Note that ”plain” Entry arguments can be supplied using new AbstractMap.SimpleEntry(k,v).

Bulk operations may complete abruptly, throwing an exception encountered in the application of a supplied function. Bear in mind when handling such exceptions that other concurrently executing functions could also have thrown exceptions, or would have done so if the first exception had not occurred.

Speedups for parallel compared to sequential forms are common but not guaranteed. Parallel operations involving brief functions on small maps may execute more slowly than sequential forms if the underlying work to parallelize the computation is more expensive than the computation itself. Similarly, parallelization may not lead to much actual parallelism if all processors are busy performing unrelated tasks.

All arguments to all task methods must be non-null.

notes:
不允许null key, null value.
null有特殊意义，表示不存在。

1.2 Implementation notes.

Overview:

The primary design goal of this hash table is to maintain concurrent readability (typically method get(), but also iterators and related methods) while minimizing update contention. Secondary goals are to keep space consumption about the same or better than java.util.HashMap, and to support high initial insertion rates on an empty table by many threads.

This map usually acts as a binned (bucketed) hash table. Each key-value mapping is held in a **Node**. Most nodes are instances of the basic **Node** class with hash, key, value, and next fields. However, various subclasses exist: **TreeNodes** are arranged in balanced trees, not lists. **TreeBins** hold the roots of sets of **TreeNodes**. **ForwardingNodes** are placed at the heads of bins during resizing. **ReservationNodes** are used as placeholders while establishing values in computeIfAbsent and related methods. The types **TreeBin**, **ForwardingNode**, and **ReservationNode** do not hold normal user keys, values, or hashes, and are readily distinguishable during search etc because they have negative hash fields and null key and value fields. (These special nodes are either uncommon or transient, so the impact of carrying around some unused fields is insignificant.)

The table is lazily initialized to a power-of-two size upon the first insertion. Each bin in the table normally contains a list of Nodes (most often, the list has only zero or one Node). Table accesses require volatile/atomic reads, writes, and CASes. Because there is no other way to arrange this without adding further indirections, we use intrinsics (sun.misc.Unsafe) operations.

We use the top (sign) bit of Node hash fields for control purposes – it is available anyway because of addressing constraints. Nodes with negative hash fields are specially handled or ignored in map methods.

Insertion (via put or its variants) of the first node in an empty bin is performed by just CASing it to the bin. This is by far the most common case for put operations under most key/hash distributions. Other update operations (insert, delete, and replace) require locks. We do not want to waste the space required to associate a distinct lock object with each bin, so instead use the first node of a bin list itself as a lock. Locking support for these locks relies on builtin ”synchronized” monitors.

Using the first node of a list as a lock does not by itself suffice though: When a node is locked, any update must first validate that it is still the first node after locking it, and retry if not. Because new nodes are always appended to lists, once a node is first in a bin, it remains first until deleted or the bin becomes invalidated (upon resizing).

The main disadvantage of per-bin locks is that other update operations on other nodes in a bin list protected by the same lock can stall, for example when user equals() or mapping functions take a long time. However, statistically, under random hash codes, this is not a common problem. Ideally, the frequency of nodes in bins follows a Poisson distribution ($http://en.wikipedia.org/wiki/Poisson_distribution$) with a parameter of about 0.5 on average, given the resizing threshold of 0.75, although with a large variance because of resizing granularity. Ignoring variance, the expected occurrences of list size k are (exp(-0.5) * pow(0.5, k) / factorial(k)). The first values are:

0: 0.60653066
1: 0.30326533
2: 0.07581633
3: 0.01263606
4: 0.00157952
5: 0.00015795
6: 0.00001316
7: 0.00000094
8: 0.00000006
more: less than 1 in ten million

Lock contention probability for two threads accessing distinct elements is roughly $1/(8 * \#elements)$ under random hashes.

Actual hash code distributions encountered in practice sometimes deviate significantly from uniform randomness. This includes the case when $N > (1 << 30)$, so some keys MUST collide. Similarly for dumb or hostile usages in which multiple keys are designed to have identical hash codes or ones that differs only in masked-out high bits. So we use a secondary strategy that applies when the number of nodes in a bin exceeds a threshold. These TreeBins use a balanced tree to hold nodes (a specialized form of red-black trees), bounding search time to O(log N). Each search step in a TreeBin is at least twice as slow as in a regular list, but given that N cannot exceed (1 << 64) (before running out of addresses) this bounds search steps, lock hold times, etc, to reasonable constants (roughly 100 nodes inspected per operation worst case) so long as keys are Comparable (which is very common – String, Long, etc). TreeBin nodes (TreeNodes) also maintain the same ”next” traversal pointers as regular nodes, so can be traversed in iterators in the same way.

The table is resized when occupancy exceeds a percentage threshold (nominally, 0.75, but see below). Any thread noticing an overfull bin may assist in resizing after the initiating thread allocates and sets up the replacement array. However, rather than stalling, these other threads may proceed with insertions etc. The use of TreeBins shields us from the worst case effects of overfilling while resizes are in progress. Resizing proceeds by transferring bins, one by one, from the table to the next table. However, threads claim small blocks of indices to transfer (via field transferIndex) before doing so, reducing contention. A generation stamp in field sizeCtl ensures that resizings do not overlap. Because we are using power-of-two expansion, the elements from each bin must either stay at same index, or move with a power of two offset. We eliminate unnecessary node creation by catching cases where old nodes can be reused because their next fields won't change. On average, only about one-sixth of them need cloning when a table doubles. The nodes they replace will be garbage collectable as soon as they are no longer referenced by any reader thread that may be in the midst of concurrently traversing table. Upon transfer, the old table bin contains only a special forwarding node (with hash field "MOVED") that contains the next table as its key. On encountering a forwarding node, access and update operations restart, using the new table.

Each bin transfer requires its bin lock, which can stall waiting for locks while resizing. However, because other threads can join in and help resize rather than contend for locks, average aggregate waits become shorter as resizing progresses. The transfer operation must also ensure that all accessible bins in both the old and new table are usable by any traversal. This is arranged in part by proceeding from the last bin (table.length - 1) up towards the first. Upon seeing a forwarding node, traversals (see class Traverser) arrange to move to the new table without revisiting nodes. To ensure that no intervening nodes are skipped even when moved out of order, a stack (see class TableStack) is created on first encounter of a forwarding node during a traversal, to maintain its place if later processing the current table. The need for these save/restore mechanics is relatively rare, but when one forwarding node is encountered, typically many more will be. So Traversers use a simple caching scheme to avoid creating so many new TableStack nodes. (Thanks to Peter Levart for suggesting use of a stack here.)

The traversal scheme also applies to partial traversals of ranges of bins (via an alternate Traverser constructor) to support partitioned aggregate operations. Also, read-only operations give up if ever forwarded to a null table, which provides support for shutdown-style clearing, which is also not currently implemented.

Lazy table initialization minimizes footprint until first use, and also avoids resizings when the first operation is from a putAll, constructor with map argument, or deserialization. These cases attempt to override the initial capacity settings, but harmlessly fail to take effect in cases of races.

The element count is maintained using a specialization of LongAdder. We need to incorporate a specialization rather than just use a LongAdder in order to access implicit contention-sensing that leads to creation of multiple CounterCells. The counter mechanics avoid contention on updates but can encounter cache thrashing if read too frequently during concurrent access. To avoid reading so often, resizing under contention is attempted only upon adding to a bin already holding two or more nodes. Under uniform hash distributions, the probability of this occurring at threshold is around 13%, meaning that only about 1 in 8 puts check threshold (and after resizing, many fewer do so).

TreeBins use a special form of comparison for search and related operations (which is the main reason we cannot use existing collections such as TreeMaps). TreeBins contain Comparable elements, but may contain others, as well as elements that are Comparable but not necessarily Comparable for the same T, so we cannot invoke compareTo among them. To handle this, the tree is ordered primarily by hash value, then by Comparable.compareTo order if applicable. On lookup at a node, if elements are not comparable or compare as 0 then both left and right children may need to be searched in the case of tied hash values. (This corresponds to the full list search that would be necessary if all elements were non-Comparable and had tied hashes.) On insertion, to keep a total ordering (or as close as is required here) across rebalancings, we compare classes and identityHashCodes as tie-breakers. The red-black balancing code is updated from pre-jdk-collections ($http://gee.cs.oswego.edu/dl/classes/collections/RBCell.java$) based in turn on Cormen, Leiserson, and Rivest "Introduction to Algorithms" (CLR).

TreeBins also require an additional locking mechanism. While list traversal is always possible by readers even during updates, tree traversal is not, mainly because of tree-rotations that may change the root node and/or its linkages. TreeBins include a simple read-write lock mechanism parasitic on the main bin-synchronization strategy: Structural adjustments associated with an insertion or removal are already bin-locked (and so cannot conflict with other writers) but must wait for ongoing readers to finish. Since there can be only one such waiter, we use a simple scheme using a single "waiter" field to block writers. However, readers need never block. If the root lock is held, they proceed along the slow traversal path (via next-pointers) until the lock becomes available or the list is exhausted, whichever comes first. These cases are not fast, but maximize aggregate expected throughput.

Maintaining API and serialization compatibility with previous versions of this class introduces several oddities. Mainly: We leave untouched but unused constructor arguments refering to concurrencyLevel. We accept a loadFactor constructor argument, but apply it only to initial table capacity (which is the only time that we can guarantee to honor it.) We also declare an unused "Segment" class that is instantiated in minimal form only when serializing.

Also, solely for compatibility with previous versions of this class, it extends AbstractMap, even though all of its methods are overridden, so it is just useless baggage.

This file is organized to make things a little easier to follow while reading than they might otherwise: First the main static declarations and utilities, then fields, then main public methods (with a few factorings of multiple public methods into internal ones), then sizing methods, trees, traversers, and bulk operations.