001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.collections4.map;  
018  
019import java.util.AbstractCollection;  
020import java.util.AbstractSet;  
021import java.util.ArrayList;  
022import java.util.Collection;  
023import java.util.Iterator;  
024import java.util.Map;  
025import java.util.NoSuchElementException;  
026import java.util.Set;  
027  
028import org.apache.commons.collections4.KeyValue;  
029  
030/\*\*  
031 \* A StaticBucketMap is an efficient, thread-safe implementation of  
032 \* <code>java.util.Map</code> that performs well in in a highly  
033 \* thread-contentious environment. The map supports very efficient  
034 \* {@link #get(Object) get}, {@link #put(Object,Object) put},  
035 \* {@link #remove(Object) remove} and {@link #containsKey(Object) containsKey}  
036 \* operations, assuming (approximate) uniform hashing and  
037 \* that the number of entries does not exceed the number of buckets. If the  
038 \* number of entries exceeds the number of buckets or if the hash codes of the  
039 \* objects are not uniformly distributed, these operations have a worst case  
040 \* scenario that is proportional to the number of elements in the map  
041 \* (<i>O(n)</i>).<p>  
042 \*  
043 \* Each bucket in the hash table has its own monitor, so two threads can  
044 \* safely operate on the map at the same time, often without incurring any  
045 \* monitor contention. This means that you don't have to wrap instances  
046 \* of this class with {@link java.util.Collections#synchronizedMap(Map)};  
047 \* instances are already thread-safe. Unfortunately, however, this means  
048 \* that this map implementation behaves in ways you may find disconcerting.  
049 \* Bulk operations, such as {@link #putAll(Map) putAll} or the  
050 \* {@link Collection#retainAll(Collection) retainAll} operation in collection  
051 \* views, are <i>not</i> atomic. If two threads are simultaneously  
052 \* executing  
053 \*  
054 \* <pre>  
055 \* staticBucketMapInstance.putAll(map);  
056 \* </pre>  
057 \*  
058 \* and  
059 \*  
060 \* <pre>  
061 \* staticBucketMapInstance.entrySet().removeAll(map.entrySet());  
062 \* </pre>  
063 \*  
064 \* then the results are generally random. Those two statement could cancel  
065 \* each other out, leaving <code>staticBucketMapInstance</code> essentially  
066 \* unchanged, or they could leave some random subset of <code>map</code> in  
067 \* <code>staticBucketMapInstance</code>.<p>  
068 \*  
069 \* Also, much like an encyclopedia, the results of {@link #size()} and  
070 \* {@link #isEmpty()} are out-of-date as soon as they are produced.<p>  
071 \*  
072 \* The iterators returned by the collection views of this class are <i>not</i>  
073 \* fail-fast. They will <i>never</i> raise a  
074 \* {@link java.util.ConcurrentModificationException}. Keys and values  
075 \* added to the map after the iterator is created do not necessarily appear  
076 \* during iteration. Similarly, the iterator does not necessarily fail to  
077 \* return keys and values that were removed after the iterator was created.<p>  
078 \*  
079 \* Finally, unlike {@link java.util.HashMap}-style implementations, this  
080 \* class <i>never</i> rehashes the map. The number of buckets is fixed  
081 \* at construction time and never altered. Performance may degrade if  
082 \* you do not allocate enough buckets upfront.<p>  
083 \*  
084 \* The {@link #atomic(Runnable)} method is provided to allow atomic iterations  
085 \* and bulk operations; however, overuse of {@link #atomic(Runnable) atomic}  
086 \* will basically result in a map that's slower than an ordinary synchronized  
087 \* {@link java.util.HashMap}.  
088 \*  
089 \* Use this class if you do not require reliable bulk operations and  
090 \* iterations, or if you can make your own guarantees about how bulk  
091 \* operations will affect the map.<p>  
092 \*  
093 \* @param <K> the type of the keys in this map  
094 \* @param <V> the type of the values in this map  
095 \* @since 3.0 (previously in main package v2.1)  
096 \*/  
097public final class StaticBucketMap<K, V> extends AbstractIterableMap<K, V> {  
098  
099 /\*\* The default number of buckets to use \*/  
100 private static final int DEFAULT\_BUCKETS = 255;  
101 /\*\* The array of buckets, where the actual data is held \*/  
102 private final Node<K, V>[] buckets;  
103 /\*\* The matching array of locks \*/  
104 private final Lock[] locks;  
105  
106 /\*\*  
107 \* Initializes the map with the default number of buckets (255).  
108 \*/  
109 public StaticBucketMap() {  
110 this(DEFAULT\_BUCKETS);  
111 }  
112  
113 /\*\*  
114 \* Initializes the map with a specified number of buckets. The number  
115 \* of buckets is never below 17, and is always an odd number (StaticBucketMap  
116 \* ensures this). The number of buckets is inversely proportional to the  
117 \* chances for thread contention. The fewer buckets, the more chances for  
118 \* thread contention. The more buckets the fewer chances for thread  
119 \* contention.  
120 \*  
121 \* @param numBuckets the number of buckets for this map  
122 \*/  
123 @SuppressWarnings("unchecked")  
124 public StaticBucketMap(final int numBuckets) {  
125 int size = Math.max(17, numBuckets);  
126  
127 // Ensure that bucketSize is never a power of 2 (to ensure maximal distribution)  
128 if (size % 2 == 0) {  
129 size--;  
130 }  
131  
132 buckets = new Node[size];  
133 locks = new Lock[size];  
134  
135 for (int i = 0; i < size; i++) {  
136 locks[i] = new Lock();  
137 }  
138 }  
139  
140 //-----------------------------------------------------------------------  
141 /\*\*  
142 \* Determine the exact hash entry for the key. The hash algorithm  
143 \* is rather simplistic, but it does the job:  
144 \*  
145 \* <pre>  
146 \* He = |Hk mod n|  
147 \* </pre>  
148 \*  
149 \* <p>  
150 \* He is the entry's hashCode, Hk is the key's hashCode, and n is  
151 \* the number of buckets.  
152 \* </p>  
153 \*/  
154 private int getHash(final Object key) {  
155 if (key == null) {  
156 return 0;  
157 }  
158 int hash = key.hashCode();  
159 hash += ~(hash << 15);  
160 hash ^= (hash >>> 10);  
161 hash += (hash << 3);  
162 hash ^= (hash >>> 6);  
163 hash += ~(hash << 11);  
164 hash ^= (hash >>> 16);  
165 hash %= buckets.length;  
166 return (hash < 0) ? hash \* -1 : hash;  
167 }  
168  
169 /\*\*  
170 \* Gets the current size of the map.  
171 \* The value is computed fresh each time the method is called.  
172 \*  
173 \* @return the current size  
174 \*/  
175 @Override  
176 public int size() {  
177 int cnt = 0;  
178  
179 for (int i = 0; i < buckets.length; i++) {  
180 synchronized(locks[i]) {  
181 cnt += locks[i].size;  
182 }  
183 }  
184 return cnt;  
185 }  
186  
187 /\*\*  
188 \* Checks if the size is currently zero.  
189 \*  
190 \* @return true if empty  
191 \*/  
192 @Override  
193 public boolean isEmpty() {  
194 return (size() == 0);  
195 }  
196  
197 /\*\*  
198 \* Gets the value associated with the key.  
199 \*  
200 \* @param key the key to retrieve  
201 \* @return the associated value  
202 \*/  
203 @Override  
204 public V get(final Object key) {  
205 final int hash = getHash(key);  
206  
207 synchronized (locks[hash]) {  
208 Node<K, V> n = buckets[hash];  
209  
210 while (n != null) {  
211 if (n.key == key || (n.key != null && n.key.equals(key))) {  
212 return n.value;  
213 }  
214  
215 n = n.next;  
216 }  
217 }  
218 return null;  
219 }  
220  
221 /\*\*  
222 \* Checks if the map contains the specified key.  
223 \*  
224 \* @param key the key to check  
225 \* @return true if found  
226 \*/  
227 @Override  
228 public boolean containsKey(final Object key) {  
229 final int hash = getHash(key);  
230  
231 synchronized (locks[hash]) {  
232 Node<K, V> n = buckets[hash];  
233  
234 while (n != null) {  
235 if (n.key == key || (n.key != null && n.key.equals(key))) {  
236 return true;  
237 }  
238  
239 n = n.next;  
240 }  
241 }  
242 return false;  
243 }  
244  
245 /\*\*  
246 \* Checks if the map contains the specified value.  
247 \*  
248 \* @param value the value to check  
249 \* @return true if found  
250 \*/  
251 @Override  
252 public boolean containsValue(final Object value) {  
253 for (int i = 0; i < buckets.length; i++) {  
254 synchronized (locks[i]) {  
255 Node<K, V> n = buckets[i];  
256  
257 while (n != null) {  
258 if (n.value == value || (n.value != null && n.value.equals(value))) {  
259 return true;  
260 }  
261  
262 n = n.next;  
263 }  
264 }  
265 }  
266 return false;  
267 }  
268  
269 //-----------------------------------------------------------------------  
270 /\*\*  
271 \* Puts a new key value mapping into the map.  
272 \*  
273 \* @param key the key to use  
274 \* @param value the value to use  
275 \* @return the previous mapping for the key  
276 \*/  
277 @Override  
278 public V put(final K key, final V value) {  
279 final int hash = getHash(key);  
280  
281 synchronized (locks[hash]) {  
282 Node<K, V> n = buckets[hash];  
283  
284 if (n == null) {  
285 n = new Node<>();  
286 n.key = key;  
287 n.value = value;  
288 buckets[hash] = n;  
289 locks[hash].size++;  
290 return null;  
291 }  
292  
293 // Set n to the last node in the linked list. Check each key along the way  
294 // If the key is found, then change the value of that node and return  
295 // the old value.  
296 for (Node<K, V> next = n; next != null; next = next.next) {  
297 n = next;  
298  
299 if (n.key == key || (n.key != null && n.key.equals(key))) {  
300 final V returnVal = n.value;  
301 n.value = value;  
302 return returnVal;  
303 }  
304 }  
305  
306 // The key was not found in the current list of nodes, add it to the end  
307 // in a new node.  
308 final Node<K, V> newNode = new Node<>();  
309 newNode.key = key;  
310 newNode.value = value;  
311 n.next = newNode;  
312 locks[hash].size++;  
313 }  
314 return null;  
315 }  
316  
317 /\*\*  
318 \* Removes the specified key from the map.  
319 \*  
320 \* @param key the key to remove  
321 \* @return the previous value at this key  
322 \*/  
323 @Override  
324 public V remove(final Object key) {  
325 final int hash = getHash(key);  
326  
327 synchronized (locks[hash]) {  
328 Node<K, V> n = buckets[hash];  
329 Node<K, V> prev = null;  
330  
331 while (n != null) {  
332 if (n.key == key || (n.key != null && n.key.equals(key))) {  
333 // Remove this node from the linked list of nodes.  
334 if (null == prev) {  
335 // This node was the head, set the next node to be the new head.  
336 buckets[hash] = n.next;  
337 } else {  
338 // Set the next node of the previous node to be the node after this one.  
339 prev.next = n.next;  
340 }  
341 locks[hash].size--;  
342 return n.value;  
343 }  
344  
345 prev = n;  
346 n = n.next;  
347 }  
348 }  
349 return null;  
350 }  
351  
352 //-----------------------------------------------------------------------  
353 /\*\*  
354 \* Gets the key set.  
355 \*  
356 \* @return the key set  
357 \*/  
358 @Override  
359 public Set<K> keySet() {  
360 return new KeySet();  
361 }  
362  
363 /\*\*  
364 \* Gets the values.  
365 \*  
366 \* @return the values  
367 \*/  
368 @Override  
369 public Collection<V> values() {  
370 return new Values();  
371 }  
372  
373 /\*\*  
374 \* Gets the entry set.  
375 \*  
376 \* @return the entry set  
377 \*/  
378 @Override  
379 public Set<Map.Entry<K, V>> entrySet() {  
380 return new EntrySet();  
381 }  
382  
383 //-----------------------------------------------------------------------  
384 /\*\*  
385 \* Puts all the entries from the specified map into this map.  
386 \* This operation is <b>not atomic</b> and may have undesired effects.  
387 \*  
388 \* @param map the map of entries to add  
389 \*/  
390 @Override  
391 public void putAll(final Map<? extends K, ? extends V> map) {  
392 for (final Map.Entry<? extends K, ? extends V> entry : map.entrySet()) {  
393 put(entry.getKey(), entry.getValue());  
394 }  
395 }  
396  
397 /\*\*  
398 \* Clears the map of all entries.  
399 \*/  
400 @Override  
401 public void clear() {  
402 for (int i = 0; i < buckets.length; i++) {  
403 final Lock lock = locks[i];  
404 synchronized (lock) {  
405 buckets[i] = null;  
406 lock.size = 0;  
407 }  
408 }  
409 }  
410  
411 /\*\*  
412 \* Compares this map to another, as per the Map specification.  
413 \*  
414 \* @param obj the object to compare to  
415 \* @return true if equal  
416 \*/  
417 @Override  
418 public boolean equals(final Object obj) {  
419 if (obj == this) {  
420 return true;  
421 }  
422 if (obj instanceof Map<?, ?> == false) {  
423 return false;  
424 }  
425 final Map<?, ?> other = (Map<?, ?>) obj;  
426 return entrySet().equals(other.entrySet());  
427 }  
428  
429 /\*\*  
430 \* Gets the hash code, as per the Map specification.  
431 \*  
432 \* @return the hash code  
433 \*/  
434 @Override  
435 public int hashCode() {  
436 int hashCode = 0;  
437  
438 for (int i = 0; i < buckets.length; i++) {  
439 synchronized (locks[i]) {  
440 Node<K, V> n = buckets[i];  
441  
442 while (n != null) {  
443 hashCode += n.hashCode();  
444 n = n.next;  
445 }  
446 }  
447 }  
448 return hashCode;  
449 }  
450  
451 //-----------------------------------------------------------------------  
452 /\*\*  
453 \* The Map.Entry for the StaticBucketMap.  
454 \*/  
455 private static final class Node<K, V> implements Map.Entry<K, V>, KeyValue<K, V> {  
456 protected K key;  
457 protected V value;  
458 protected Node<K, V> next;  
459  
460 @Override  
461 public K getKey() {  
462 return key;  
463 }  
464  
465 @Override  
466 public V getValue() {  
467 return value;  
468 }  
469  
470 @Override  
471 public int hashCode() {  
472 return ((key == null ? 0 : key.hashCode()) ^  
473 (value == null ? 0 : value.hashCode()));  
474 }  
475  
476 @Override  
477 public boolean equals(final Object obj) {  
478 if (obj == this) {  
479 return true;  
480 }  
481 if (obj instanceof Map.Entry<?, ?> == false) {  
482 return false;  
483 }  
484  
485 final Map.Entry<?, ?> e2 = (Map.Entry<?, ?>) obj;  
486 return (  
487 (key == null ? e2.getKey() == null : key.equals(e2.getKey())) &&  
488 (value == null ? e2.getValue() == null : value.equals(e2.getValue())));  
489 }  
490  
491 @Override  
492 public V setValue(final V obj) {  
493 final V retVal = value;  
494 value = obj;  
495 return retVal;  
496 }  
497 }  
498  
499 /\*\*  
500 \* The lock object, which also includes a count of the nodes in this lock.  
501 \*/  
502 private final static class Lock {  
503 public int size;  
504 }  
505  
506 //-----------------------------------------------------------------------  
507 private class BaseIterator {  
508 private final ArrayList<Map.Entry<K, V>> current = new ArrayList<>();  
509 private int bucket;  
510 private Map.Entry<K, V> last;  
511  
512 public boolean hasNext() {  
513 if (current.size() > 0) {  
514 return true;  
515 }  
516 while (bucket < buckets.length) {  
517 synchronized (locks[bucket]) {  
518 Node<K, V> n = buckets[bucket];  
519 while (n != null) {  
520 current.add(n);  
521 n = n.next;  
522 }  
523 bucket++;  
524 if (current.size() > 0) {  
525 return true;  
526 }  
527 }  
528 }  
529 return false;  
530 }  
531  
532 protected Map.Entry<K, V> nextEntry() {  
533 if (!hasNext()) {  
534 throw new NoSuchElementException();  
535 }  
536 last = current.remove(current.size() - 1);  
537 return last;  
538 }  
539  
540 public void remove() {  
541 if (last == null) {  
542 throw new IllegalStateException();  
543 }  
544 StaticBucketMap.this.remove(last.getKey());  
545 last = null;  
546 }  
547 }  
548  
549 private class EntryIterator extends BaseIterator implements Iterator<Map.Entry<K, V>> {  
550  
551 @Override  
552 public Map.Entry<K, V> next() {  
553 return nextEntry();  
554 }  
555  
556 }  
557  
558 private class ValueIterator extends BaseIterator implements Iterator<V> {  
559  
560 @Override  
561 public V next() {  
562 return nextEntry().getValue();  
563 }  
564  
565 }  
566  
567 private class KeyIterator extends BaseIterator implements Iterator<K> {  
568  
569 @Override  
570 public K next() {  
571 return nextEntry().getKey();  
572 }  
573  
574 }  
575  
576 private class EntrySet extends AbstractSet<Map.Entry<K, V>> {  
577  
578 @Override  
579 public int size() {  
580 return StaticBucketMap.this.size();  
581 }  
582  
583 @Override  
584 public void clear() {  
585 StaticBucketMap.this.clear();  
586 }  
587  
588 @Override  
589 public Iterator<Map.Entry<K, V>> iterator() {  
590 return new EntryIterator();  
591 }  
592  
593 @Override  
594 public boolean contains(final Object obj) {  
595 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) obj;  
596 final int hash = getHash(entry.getKey());  
597 synchronized (locks[hash]) {  
598 for (Node<K, V> n = buckets[hash]; n != null; n = n.next) {  
599 if (n.equals(entry)) {  
600 return true;  
601 }  
602 }  
603 }  
604 return false;  
605 }  
606  
607 @Override  
608 public boolean remove(final Object obj) {  
609 if (obj instanceof Map.Entry<?, ?> == false) {  
610 return false;  
611 }  
612 final Map.Entry<?, ?> entry = (Map.Entry<?, ?>) obj;  
613 final int hash = getHash(entry.getKey());  
614 synchronized (locks[hash]) {  
615 for (Node<K, V> n = buckets[hash]; n != null; n = n.next) {  
616 if (n.equals(entry)) {  
617 StaticBucketMap.this.remove(n.getKey());  
618 return true;  
619 }  
620 }  
621 }  
622 return false;  
623 }  
624  
625 }  
626  
627 private class KeySet extends AbstractSet<K> {  
628  
629 @Override  
630 public int size() {  
631 return StaticBucketMap.this.size();  
632 }  
633  
634 @Override  
635 public void clear() {  
636 StaticBucketMap.this.clear();  
637 }  
638  
639 @Override  
640 public Iterator<K> iterator() {  
641 return new KeyIterator();  
642 }  
643  
644 @Override  
645 public boolean contains(final Object obj) {  
646 return StaticBucketMap.this.containsKey(obj);  
647 }  
648  
649 @Override  
650 public boolean remove(final Object obj) {  
651 final int hash = getHash(obj);  
652 synchronized (locks[hash]) {  
653 for (Node<K, V> n = buckets[hash]; n != null; n = n.next) {  
654 final Object k = n.getKey();  
655 if ((k == obj) || ((k != null) && k.equals(obj))) {  
656 StaticBucketMap.this.remove(k);  
657 return true;  
658 }  
659 }  
660 }  
661 return false;  
662 }  
663  
664 }  
665  
666  
667 private class Values extends AbstractCollection<V> {  
668  
669 @Override  
670 public int size() {  
671 return StaticBucketMap.this.size();  
672 }  
673  
674 @Override  
675 public void clear() {  
676 StaticBucketMap.this.clear();  
677 }  
678  
679 @Override  
680 public Iterator<V> iterator() {  
681 return new ValueIterator();  
682 }  
683  
684 }  
685  
686 /\*\*  
687 \* Prevents any operations from occurring on this map while the  
688 \* given {@link Runnable} executes. This method can be used, for  
689 \* instance, to execute a bulk operation atomically:  
690 \*  
691 \* <pre>  
692 \* staticBucketMapInstance.atomic(new Runnable() {  
693 \* public void run() {  
694 \* staticBucketMapInstance.putAll(map);  
695 \* }  
696 \* });  
697 \* </pre>  
698 \*  
699 \* It can also be used if you need a reliable iterator:  
700 \*  
701 \* <pre>  
702 \* staticBucketMapInstance.atomic(new Runnable() {  
703 \* public void run() {  
704 \* Iterator iterator = staticBucketMapInstance.iterator();  
705 \* while (iterator.hasNext()) {  
706 \* foo(iterator.next();  
707 \* }  
708 \* }  
709 \* });  
710 \* </pre>  
711 \*  
712 \* <b>Implementation note:</b> This method requires a lot of time  
713 \* and a ton of stack space. Essentially a recursive algorithm is used  
714 \* to enter each bucket's monitor. If you have twenty thousand buckets  
715 \* in your map, then the recursive method will be invoked twenty thousand  
716 \* times. You have been warned.  
717 \*  
718 \* @param r the code to execute atomically  
719 \*/  
720 public void atomic(final Runnable r) {  
721 if (r == null) {  
722 throw new NullPointerException();  
723 }  
724 atomic(r, 0);  
725 }  
726  
727 private void atomic(final Runnable r, final int bucket) {  
728 if (bucket >= buckets.length) {  
729 r.run();  
730 return;  
731 }  
732 synchronized (locks[bucket]) {  
733 atomic(r, bucket + 1);  
734 }  
735 }  
736  
737}