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.io.IOException;  
020import java.io.ObjectInputStream;  
021import java.io.ObjectOutputStream;  
022import java.io.Serializable;  
023import java.util.Collection;  
024import java.util.HashMap;  
025import java.util.Iterator;  
026import java.util.Map;  
027import java.util.Set;  
028import java.util.concurrent.TimeUnit;  
029  
030/\*\*  
031 \* Decorates a <code>Map</code> to evict expired entries once their expiration  
032 \* time has been reached.  
033 \* <p>  
034 \* When putting a key-value pair in the map this decorator uses a  
035 \* {@link ExpirationPolicy} to determine how long the entry should remain alive  
036 \* as defined by an expiration time value.  
037 \* </p>  
038 \* <p>  
039 \* When accessing the mapped value for a key, its expiration time is checked,  
040 \* and if it is a negative value or if it is greater than the current time, the  
041 \* mapped value is returned. Otherwise, the key is removed from the decorated  
042 \* map, and <code>null</code> is returned.  
043 \* </p>  
044 \* <p>  
045 \* When invoking methods that involve accessing the entire map contents (i.e  
046 \* {@link #containsKey(Object)}, {@link #entrySet()}, etc.) this decorator  
047 \* removes all expired entries prior to actually completing the invocation.  
048 \* </p>  
049 \* <p>  
050 \* <strong>Note that {@link PassiveExpiringMap} is not synchronized and is not  
051 \* thread-safe.</strong> If you wish to use this map from multiple threads  
052 \* concurrently, you must use appropriate synchronization. The simplest approach  
053 \* is to wrap this map using {@link java.util.Collections#synchronizedMap(Map)}.  
054 \* This class may throw exceptions when accessed by concurrent threads without  
055 \* synchronization.  
056 \* </p>  
057 \*  
058 \* @param <K> the type of the keys in this map  
059 \* @param <V> the type of the values in this map  
060 \* @since 4.0  
061 \*/  
062public class PassiveExpiringMap<K, V>  
063 extends AbstractMapDecorator<K, V>  
064 implements Serializable {  
065  
066 /\*\*  
067 \* A {@link org.apache.commons.collections4.map.PassiveExpiringMap.ExpirationPolicy ExpirationPolicy}  
068 \* that returns a expiration time that is a  
069 \* constant about of time in the future from the current time.  
070 \*  
071 \* @param <K> the type of the keys in the map  
072 \* @param <V> the type of the values in the map  
073 \* @since 4.0  
074 \*/  
075 public static class ConstantTimeToLiveExpirationPolicy<K, V>  
076 implements ExpirationPolicy<K, V> {  
077  
078 /\*\* Serialization version \*/  
079 private static final long serialVersionUID = 1L;  
080  
081 /\*\* the constant time-to-live value measured in milliseconds. \*/  
082 private final long timeToLiveMillis;  
083  
084 /\*\*  
085 \* Default constructor. Constructs a policy using a negative  
086 \* time-to-live value that results in entries never expiring.  
087 \*/  
088 public ConstantTimeToLiveExpirationPolicy() {  
089 this(-1L);  
090 }  
091  
092 /\*\*  
093 \* Construct a policy with the given time-to-live constant measured in  
094 \* milliseconds. A negative time-to-live value indicates entries never  
095 \* expire. A zero time-to-live value indicates entries expire (nearly)  
096 \* immediately.  
097 \*  
098 \* @param timeToLiveMillis the constant amount of time (in milliseconds)  
099 \* an entry is available before it expires. A negative value  
100 \* results in entries that NEVER expire. A zero value results in  
101 \* entries that ALWAYS expire.  
102 \*/  
103 public ConstantTimeToLiveExpirationPolicy(final long timeToLiveMillis) {  
104 super();  
105 this.timeToLiveMillis = timeToLiveMillis;  
106 }  
107  
108 /\*\*  
109 \* Construct a policy with the given time-to-live constant measured in  
110 \* the given time unit of measure.  
111 \*  
112 \* @param timeToLive the constant amount of time an entry is available  
113 \* before it expires. A negative value results in entries that  
114 \* NEVER expire. A zero value results in entries that ALWAYS  
115 \* expire.  
116 \* @param timeUnit the unit of time for the <code>timeToLive</code>  
117 \* parameter, must not be null.  
118 \* @throws NullPointerException if the time unit is null.  
119 \*/  
120 public ConstantTimeToLiveExpirationPolicy(final long timeToLive,  
121 final TimeUnit timeUnit) {  
122 this(validateAndConvertToMillis(timeToLive, timeUnit));  
123 }  
124  
125 /\*\*  
126 \* Determine the expiration time for the given key-value entry.  
127 \*  
128 \* @param key the key for the entry (ignored).  
129 \* @param value the value for the entry (ignored).  
130 \* @return if {@link #timeToLiveMillis} ≥ 0, an expiration time of  
131 \* {@link #timeToLiveMillis} +  
132 \* {@link System#currentTimeMillis()} is returned. Otherwise, -1  
133 \* is returned indicating the entry never expires.  
134 \*/  
135 @Override  
136 public long expirationTime(final K key, final V value) {  
137 if (timeToLiveMillis >= 0L) {  
138 // avoid numerical overflow  
139 final long now = System.currentTimeMillis();  
140 if (now > Long.MAX\_VALUE - timeToLiveMillis) {  
141 // expiration would be greater than Long.MAX\_VALUE  
142 // never expire  
143 return -1;  
144 }  
145  
146 // timeToLiveMillis in the future  
147 return now + timeToLiveMillis;  
148 }  
149  
150 // never expire  
151 return -1L;  
152 }  
153 }  
154  
155 /\*\*  
156 \* A policy to determine the expiration time for key-value entries.  
157 \*  
158 \* @param <K> the key object type.  
159 \* @param <V> the value object type  
160 \* @since 4.0  
161 \*/  
162 @FunctionalInterface  
163 public interface ExpirationPolicy<K, V>  
164 extends Serializable {  
165  
166 /\*\*  
167 \* Determine the expiration time for the given key-value entry.  
168 \*  
169 \* @param key the key for the entry.  
170 \* @param value the value for the entry.  
171 \* @return the expiration time value measured in milliseconds. A  
172 \* negative return value indicates the entry never expires.  
173 \*/  
174 long expirationTime(K key, V value);  
175 }  
176  
177 /\*\* Serialization version \*/  
178 private static final long serialVersionUID = 1L;  
179  
180 /\*\*  
181 \* First validate the input parameters. If the parameters are valid, convert  
182 \* the given time measured in the given units to the same time measured in  
183 \* milliseconds.  
184 \*  
185 \* @param timeToLive the constant amount of time an entry is available  
186 \* before it expires. A negative value results in entries that NEVER  
187 \* expire. A zero value results in entries that ALWAYS expire.  
188 \* @param timeUnit the unit of time for the <code>timeToLive</code>  
189 \* parameter, must not be null.  
190 \* @throws NullPointerException if the time unit is null.  
191 \*/  
192 private static long validateAndConvertToMillis(final long timeToLive,  
193 final TimeUnit timeUnit) {  
194 if (timeUnit == null) {  
195 throw new NullPointerException("Time unit must not be null");  
196 }  
197 return TimeUnit.MILLISECONDS.convert(timeToLive, timeUnit);  
198 }  
199  
200 /\*\* map used to manage expiration times for the actual map entries. \*/  
201 private final Map<Object, Long> expirationMap = new HashMap<>();  
202  
203 /\*\* the policy used to determine time-to-live values for map entries. \*/  
204 private final ExpirationPolicy<K, V> expiringPolicy;  
205  
206 /\*\*  
207 \* Default constructor. Constructs a map decorator that results in entries  
208 \* NEVER expiring.  
209 \*/  
210 public PassiveExpiringMap() {  
211 this(-1L);  
212 }  
213  
214 /\*\*  
215 \* Construct a map decorator using the given expiration policy to determine  
216 \* expiration times.  
217 \*  
218 \* @param expiringPolicy the policy used to determine expiration times of  
219 \* entries as they are added.  
220 \* @throws NullPointerException if expiringPolicy is null  
221 \*/  
222 public PassiveExpiringMap(final ExpirationPolicy<K, V> expiringPolicy) {  
223 this(expiringPolicy, new HashMap<K, V>());  
224 }  
225  
226 /\*\*  
227 \* Construct a map decorator that decorates the given map and uses the given  
228 \* expiration policy to determine expiration times. If there are any  
229 \* elements already in the map being decorated, they will NEVER expire  
230 \* unless they are replaced.  
231 \*  
232 \* @param expiringPolicy the policy used to determine expiration times of  
233 \* entries as they are added.  
234 \* @param map the map to decorate, must not be null.  
235 \* @throws NullPointerException if the map or expiringPolicy is null.  
236 \*/  
237 public PassiveExpiringMap(final ExpirationPolicy<K, V> expiringPolicy,  
238 final Map<K, V> map) {  
239 super(map);  
240 if (expiringPolicy == null) {  
241 throw new NullPointerException("Policy must not be null.");  
242 }  
243 this.expiringPolicy = expiringPolicy;  
244 }  
245  
246 /\*\*  
247 \* Construct a map decorator that decorates the given map using the given  
248 \* time-to-live value measured in milliseconds to create and use a  
249 \* {@link ConstantTimeToLiveExpirationPolicy} expiration policy.  
250 \*  
251 \* @param timeToLiveMillis the constant amount of time (in milliseconds) an  
252 \* entry is available before it expires. A negative value results in  
253 \* entries that NEVER expire. A zero value results in entries that  
254 \* ALWAYS expire.  
255 \*/  
256 public PassiveExpiringMap(final long timeToLiveMillis) {  
257 this(new ConstantTimeToLiveExpirationPolicy<K, V>(timeToLiveMillis),  
258 new HashMap<K, V>());  
259 }  
260  
261 /\*\*  
262 \* Construct a map decorator using the given time-to-live value measured in  
263 \* milliseconds to create and use a  
264 \* {@link ConstantTimeToLiveExpirationPolicy} expiration policy. If there  
265 \* are any elements already in the map being decorated, they will NEVER  
266 \* expire unless they are replaced.  
267 \*  
268 \* @param timeToLiveMillis the constant amount of time (in milliseconds) an  
269 \* entry is available before it expires. A negative value results in  
270 \* entries that NEVER expire. A zero value results in entries that  
271 \* ALWAYS expire.  
272 \* @param map the map to decorate, must not be null.  
273 \* @throws NullPointerException if the map is null.  
274 \*/  
275 public PassiveExpiringMap(final long timeToLiveMillis, final Map<K, V> map) {  
276 this(new ConstantTimeToLiveExpirationPolicy<K, V>(timeToLiveMillis),  
277 map);  
278 }  
279  
280 /\*\*  
281 \* Construct a map decorator using the given time-to-live value measured in  
282 \* the given time units of measure to create and use a  
283 \* {@link ConstantTimeToLiveExpirationPolicy} expiration policy.  
284 \*  
285 \* @param timeToLive the constant amount of time an entry is available  
286 \* before it expires. A negative value results in entries that NEVER  
287 \* expire. A zero value results in entries that ALWAYS expire.  
288 \* @param timeUnit the unit of time for the <code>timeToLive</code>  
289 \* parameter, must not be null.  
290 \* @throws NullPointerException if the time unit is null.  
291 \*/  
292 public PassiveExpiringMap(final long timeToLive, final TimeUnit timeUnit) {  
293 this(validateAndConvertToMillis(timeToLive, timeUnit));  
294 }  
295  
296 /\*\*  
297 \* Construct a map decorator that decorates the given map using the given  
298 \* time-to-live value measured in the given time units of measure to create  
299 \* {@link ConstantTimeToLiveExpirationPolicy} expiration policy. This policy  
300 \* is used to determine expiration times. If there are any elements already  
301 \* in the map being decorated, they will NEVER expire unless they are  
302 \* replaced.  
303 \*  
304 \* @param timeToLive the constant amount of time an entry is available  
305 \* before it expires. A negative value results in entries that NEVER  
306 \* expire. A zero value results in entries that ALWAYS expire.  
307 \* @param timeUnit the unit of time for the <code>timeToLive</code>  
308 \* parameter, must not be null.  
309 \* @param map the map to decorate, must not be null.  
310 \* @throws NullPointerException if the map or time unit is null.  
311 \*/  
312 public PassiveExpiringMap(final long timeToLive, final TimeUnit timeUnit, final Map<K, V> map) {  
313 this(validateAndConvertToMillis(timeToLive, timeUnit), map);  
314 }  
315  
316 /\*\*  
317 \* Constructs a map decorator that decorates the given map and results in  
318 \* entries NEVER expiring. If there are any elements already in the map  
319 \* being decorated, they also will NEVER expire.  
320 \*  
321 \* @param map the map to decorate, must not be null.  
322 \* @throws NullPointerException if the map is null.  
323 \*/  
324 public PassiveExpiringMap(final Map<K, V> map) {  
325 this(-1L, map);  
326 }  
327  
328 /\*\*  
329 \* Normal {@link Map#clear()} behavior with the addition of clearing all  
330 \* expiration entries as well.  
331 \*/  
332 @Override  
333 public void clear() {  
334 super.clear();  
335 expirationMap.clear();  
336 }  
337  
338 /\*\*  
339 \* All expired entries are removed from the map prior to determining the  
340 \* contains result.  
341 \* {@inheritDoc}  
342 \*/  
343 @Override  
344 public boolean containsKey(final Object key) {  
345 removeIfExpired(key, now());  
346 return super.containsKey(key);  
347 }  
348  
349 /\*\*  
350 \* All expired entries are removed from the map prior to determining the  
351 \* contains result.  
352 \* {@inheritDoc}  
353 \*/  
354 @Override  
355 public boolean containsValue(final Object value) {  
356 removeAllExpired(now());  
357 return super.containsValue(value);  
358 }  
359  
360 /\*\*  
361 \* All expired entries are removed from the map prior to returning the entry set.  
362 \* {@inheritDoc}  
363 \*/  
364 @Override  
365 public Set<Entry<K, V>> entrySet() {  
366 removeAllExpired(now());  
367 return super.entrySet();  
368 }  
369  
370 /\*\*  
371 \* All expired entries are removed from the map prior to returning the entry value.  
372 \* {@inheritDoc}  
373 \*/  
374 @Override  
375 public V get(final Object key) {  
376 removeIfExpired(key, now());  
377 return super.get(key);  
378 }  
379  
380 /\*\*  
381 \* All expired entries are removed from the map prior to determining if it is empty.  
382 \* {@inheritDoc}  
383 \*/  
384 @Override  
385 public boolean isEmpty() {  
386 removeAllExpired(now());  
387 return super.isEmpty();  
388 }  
389  
390 /\*\*  
391 \* Determines if the given expiration time is less than <code>now</code>.  
392 \*  
393 \* @param now the time in milliseconds used to compare against the  
394 \* expiration time.  
395 \* @param expirationTimeObject the expiration time value retrieved from  
396 \* {@link #expirationMap}, can be null.  
397 \* @return <code>true</code> if <code>expirationTimeObject</code> is ≥ 0  
398 \* and <code>expirationTimeObject</code> < <code>now</code>.  
399 \* <code>false</code> otherwise.  
400 \*/  
401 private boolean isExpired(final long now, final Long expirationTimeObject) {  
402 if (expirationTimeObject != null) {  
403 final long expirationTime = expirationTimeObject.longValue();  
404 return expirationTime >= 0 && now >= expirationTime;  
405 }  
406 return false;  
407 }  
408  
409 /\*\*  
410 \* All expired entries are removed from the map prior to returning the key set.  
411 \* {@inheritDoc}  
412 \*/  
413 @Override  
414 public Set<K> keySet() {  
415 removeAllExpired(now());  
416 return super.keySet();  
417 }  
418  
419 /\*\*  
420 \* The current time in milliseconds.  
421 \*/  
422 private long now() {  
423 return System.currentTimeMillis();  
424 }  
425  
426 /\*\*  
427 \* Add the given key-value pair to this map as well as recording the entry's expiration time based on  
428 \* the current time in milliseconds and this map's {@link #expiringPolicy}.  
429 \* <p>  
430 \* {@inheritDoc}  
431 \*/  
432 @Override  
433 public V put(final K key, final V value) {  
434 // remove the previous record  
435 removeIfExpired(key, now());  
436  
437 // record expiration time of new entry  
438 final long expirationTime = expiringPolicy.expirationTime(key, value);  
439 expirationMap.put(key, Long.valueOf(expirationTime));  
440  
441 return super.put(key, value);  
442 }  
443  
444 @Override  
445 public void putAll(final Map<? extends K, ? extends V> mapToCopy) {  
446 for (final Map.Entry<? extends K, ? extends V> entry : mapToCopy.entrySet()) {  
447 put(entry.getKey(), entry.getValue());  
448 }  
449 }  
450  
451 /\*\*  
452 \* Normal {@link Map#remove(Object)} behavior with the addition of removing  
453 \* any expiration entry as well.  
454 \* {@inheritDoc}  
455 \*/  
456 @Override  
457 public V remove(final Object key) {  
458 expirationMap.remove(key);  
459 return super.remove(key);  
460 }  
461  
462 /\*\*  
463 \* Removes all entries in the map whose expiration time is less than  
464 \* <code>now</code>. The exceptions are entries with negative expiration  
465 \* times; those entries are never removed.  
466 \*  
467 \* @see #isExpired(long, Long)  
468 \*/  
469 private void removeAllExpired(final long now) {  
470 final Iterator<Map.Entry<Object, Long>> iter = expirationMap.entrySet().iterator();  
471 while (iter.hasNext()) {  
472 final Map.Entry<Object, Long> expirationEntry = iter.next();  
473 if (isExpired(now, expirationEntry.getValue())) {  
474 // remove entry from collection  
475 super.remove(expirationEntry.getKey());  
476 // remove entry from expiration map  
477 iter.remove();  
478 }  
479 }  
480 }  
481  
482 /\*\*  
483 \* Removes the entry with the given key if the entry's expiration time is  
484 \* less than <code>now</code>. If the entry has a negative expiration time,  
485 \* the entry is never removed.  
486 \*/  
487 private void removeIfExpired(final Object key, final long now) {  
488 final Long expirationTimeObject = expirationMap.get(key);  
489 if (isExpired(now, expirationTimeObject)) {  
490 remove(key);  
491 }  
492 }  
493  
494 /\*\*  
495 \* All expired entries are removed from the map prior to returning the size.  
496 \* {@inheritDoc}  
497 \*/  
498 @Override  
499 public int size() {  
500 removeAllExpired(now());  
501 return super.size();  
502 }  
503  
504 /\*\*  
505 \* Read the map in using a custom routine.  
506 \*  
507 \* @param in the input stream  
508 \* @throws IOException if an error occurs while reading from the stream  
509 \* @throws ClassNotFoundException if an object read from the stream can not be loaded  
510 \*/  
511 @SuppressWarnings("unchecked")  
512 // (1) should only fail if input stream is incorrect  
513 private void readObject(final ObjectInputStream in)  
514 throws IOException, ClassNotFoundException {  
515 in.defaultReadObject();  
516 map = (Map<K, V>) in.readObject(); // (1)  
517 }  
518  
519 /\*\*  
520 \* Write the map out using a custom routine.  
521 \*  
522 \* @param out the output stream  
523 \* @throws IOException if an error occurs while writing to the stream  
524 \*/  
525 private void writeObject(final ObjectOutputStream out)  
526 throws IOException {  
527 out.defaultWriteObject();  
528 out.writeObject(map);  
529 }  
530  
531 /\*\*  
532 \* All expired entries are removed from the map prior to returning the value collection.  
533 \* {@inheritDoc}  
534 \*/  
535 @Override  
536 public Collection<V> values() {  
537 removeAllExpired(now());  
538 return super.values();  
539 }  
540}