package com.twitter.tweetypie.repository

import com.twitter.servo.cache.{CachedValueStatus => Status, LockingCache => KVLockingCache, \_}

import com.twitter.servo.repository.{CachedResult => Result}

import com.twitter.stitch.MapGroup

import com.twitter.stitch.Group

import com.twitter.stitch.Stitch

import com.twitter.util.Future

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.Time

import com.twitter.util.Try

/\*\*

\* Adapts a key-value locking cache to Arrow and

\* normalizes the results to `CachedResult`.

\*/

trait StitchLockingCache[K, V] {

val get: K => Stitch[Result[K, V]]

val lockAndSet: (K, StitchLockingCache.Val[V]) => Stitch[Unit]

val delete: K => Stitch[Boolean]

}

object StitchLockingCache {

/\*\*

\* Value intended to be written back to cache using lockAndSet.

\*

\* Note that only a subset of CachedValueStatus values are eligible for writing:

\* Found, NotFound, and Deleted

\*/

sealed trait Val[+V]

object Val {

case class Found[V](value: V) extends Val[V]

case object NotFound extends Val[Nothing]

case object Deleted extends Val[Nothing]

}

/\*\*

\* A Group for batching get requests to a [[KVLockingCache]].

\*/

private case class GetGroup[K, V](cache: KVLockingCache[K, Cached[V]], override val maxSize: Int)

extends MapGroup[K, Result[K, V]] {

private[this] def cachedToResult(key: K, cached: Cached[V]): Try[Result[K, V]] =

cached.status match {

case Status.NotFound => Return(Result.CachedNotFound(key, cached.cachedAt))

case Status.Deleted => Return(Result.CachedDeleted(key, cached.cachedAt))

case Status.SerializationFailed => Return(Result.SerializationFailed(key))

case Status.DeserializationFailed => Return(Result.DeserializationFailed(key))

case Status.Evicted => Return(Result.NotFound(key))

case Status.DoNotCache => Return(Result.DoNotCache(key, cached.doNotCacheUntil))

case Status.Found =>

cached.value match {

case None => Return(Result.NotFound(key))

case Some(value) => Return(Result.CachedFound(key, value, cached.cachedAt))

}

case \_ => Throw(new UnsupportedOperationException)

}

override protected def run(keys: Seq[K]): Future[K => Try[Result[K, V]]] =

cache.get(keys).map { (result: KeyValueResult[K, Cached[V]]) => key =>

result.found.get(key) match {

case Some(cached) => cachedToResult(key, cached)

case None =>

result.failed.get(key) match {

case Some(t) => Return(Result.Failed(key, t))

case None => Return(Result.NotFound(key))

}

}

}

}

/\*\*

\* Used in the implementation of LockAndSetGroup. This is just a

\* glorified tuple with special equality semantics where calls with

\* the same key will compare equal. MapGroup will use this as a key

\* in a Map, which will prevent duplicate lockAndSet calls with the

\* same key. We don't care which one we use

\*/

private class LockAndSetCall[K, V](val key: K, val value: V) {

override def equals(other: Any): Boolean =

other match {

case call: LockAndSetCall[\_, \_] => call.key == key

case \_ => false

}

override def hashCode(): Int = key.hashCode

}

/\*\*

\* A Group for `lockAndSet` calls to a [[KVLockingCache]]. This is

\* necessary to avoid writing back a key multiple times if it is

\* appears more than once in a batch. LockAndSetCall considers two

\* calls equal even if the values differ because multiple lockAndSet

\* calls for the same key will eventually result in only one being

\* chosen by the cache anyway, and this avoids conflicting

\* lockAndSet calls.

\*

\* For example, consider a tweet that mentions @jack twice

\* when @jack is not in cache. That will result in two queries to

\* load @jack, which will be deduped by the Group when the repo is

\* called. Despite the fact that it is loaded only once, each of the

\* two loads is oblivious to the other, so each of them attempts to

\* write the value back to cache, resulting in two `lockAndSet`

\* calls for @jack, so we have to dedupe them again.

\*/

private case class LockAndSetGroup[K, V](

cache: KVLockingCache[K, V],

picker: KVLockingCache.Picker[V])

extends MapGroup[LockAndSetCall[K, V], Option[V]] {

override def run(

calls: Seq[LockAndSetCall[K, V]]

): Future[LockAndSetCall[K, V] => Try[Option[V]]] =

Future

.collect {

calls.map { call =>

// This is masked to prevent interrupts to the overall

// request from interrupting writes back to cache.

cache

.lockAndSet(call.key, KVLockingCache.PickingHandler(call.value, picker))

.masked

.liftToTry

}

}

.map(responses => calls.zip(responses).toMap)

}

def apply[K, V](

underlying: KVLockingCache[K, Cached[V]],

picker: KVLockingCache.Picker[Cached[V]],

maxRequestSize: Int = Int.MaxValue

): StitchLockingCache[K, V] =

new StitchLockingCache[K, V] {

override val get: K => Stitch[Result[K, V]] = {

val group: Group[K, Result[K, V]] = GetGroup(underlying, maxRequestSize)

(key: K) => Stitch.call(key, group)

}

override val lockAndSet: (K, Val[V]) => Stitch[Unit] = {

val group = LockAndSetGroup(underlying, picker)

(key: K, value: Val[V]) => {

val now = Time.now

val cached: Cached[V] =

value match {

case Val.Found(v) => Cached[V](Some(v), Status.Found, now, Some(now))

case Val.NotFound => Cached[V](None, Status.NotFound, now, Some(now))

case Val.Deleted => Cached[V](None, Status.Deleted, now, Some(now))

}

Stitch.call(new LockAndSetCall(key, cached), group).unit

}

}

override val delete: K => Stitch[Boolean] =

(key: K) => Stitch.callFuture(underlying.delete(key))

}

}