package com.twitter.servo.cache

import com.twitter.util.{Future, Return}

import scala.collection.mutable

/\*\*

\* uses a forwarding cache to lookup a value by a secondary index.

\* filters out values for which the requested secondary index does not

\* match the actual secondary index (these are treated as a miss)

\*/

class ForwardingCache[K, F, V](

forwardingCache: Cache[K, Cached[F]],

underlyingCache: SecondaryIndexingCache[F, \_, V],

primaryKey: V => F,

secondaryKey: SecondaryIndexingCache.IndexMapping[K, V],

lockingCacheFactory: LockingCacheFactory)

extends LockingCache[K, Cached[V]] {

protected[this] case class ForwardingChecksum(

forwardingChecksum: Checksum,

underlyingChecksum: Option[Checksum])

extends Checksum

protected[this] val lockingUnderlying = lockingCacheFactory(underlyingCache)

protected[this] val lockingForwarding = lockingCacheFactory(forwardingCache)

override def get(keys: Seq[K]): Future[KeyValueResult[K, Cached[V]]] = {

forwardingCache.get(keys) flatMap { flr =>

val (tombstones, notTombstones) = {

val tombstones = mutable.Map.empty[K, Cached[F]]

val notTombstones = mutable.Map.empty[F, K]

// split results into tombstoned keys and non-tombstoned key/pKeys

// while we're at it, produce a reverse-keymap of non-tombstones

flr.found foreach {

case (key, cachedPKey) =>

cachedPKey.value match {

case Some(pKey) => notTombstones += pKey -> key

case None => tombstones += key -> cachedPKey

}

}

(tombstones.toMap, notTombstones.toMap)

}

// only make call to underlyingCache if there are keys to lookup

val fromUnderlying = if (notTombstones.isEmpty) {

KeyValueResult.emptyFuture

} else {

// get non-tombstoned values from underlying cache

underlyingCache.get(notTombstones.keys.toSeq) map { lr =>

val (goodValues, badValues) = lr.found partition {

case (pKey, cachedValue) =>

// filter out values that somehow don't match the primary key and secondary key

cachedValue.value match {

case Some(value) =>

secondaryKey(value) match {

case Return(Some(sKey)) =>

pKey == primaryKey(value) && sKey == notTombstones(pKey)

case \_ => false

}

case None => true

}

}

val found = goodValues map { case (k, v) => notTombstones(k) -> v }

val notFound = (lr.notFound ++ badValues.keySet) map { notTombstones(\_) }

val failed = lr.failed map { case (k, t) => notTombstones(k) -> t }

KeyValueResult(found, notFound, failed)

} handle {

case t =>

KeyValueResult(failed = notTombstones.values map { \_ -> t } toMap)

}

}

fromUnderlying map { lr =>

// fill in tombstone values, copying the metadata from the Cached[F]

val withTombstones = tombstones map {

case (key, cachedPKey) =>

key -> cachedPKey.copy[V](value = None)

}

val found = lr.found ++ withTombstones

val notFound = flr.notFound ++ lr.notFound

val failed = flr.failed ++ lr.failed

KeyValueResult(found, notFound, failed)

}

}

}

// since we implement lockAndSet directly, we don't support getWithChecksum and checkAndSet.

// we should consider changing the class hierarchy of Cache/LockingCache so that this can

// be checked at compile time.

override def getWithChecksum(keys: Seq[K]): Future[CsKeyValueResult[K, Cached[V]]] =

Future.exception(new UnsupportedOperationException("Use lockAndSet directly"))

override def checkAndSet(key: K, cachedValue: Cached[V], checksum: Checksum): Future[Boolean] =

Future.exception(new UnsupportedOperationException("Use lockAndSet directly"))

protected[this] def maybeAddForwardingIndex(

key: K,

cachedPrimaryKey: Cached[F],

wasAdded: Boolean

): Future[Boolean] = {

if (wasAdded)

forwardingCache.set(key, cachedPrimaryKey) map { \_ =>

true

}

else

Future.value(false)

}

override def add(key: K, cachedValue: Cached[V]): Future[Boolean] = {

// copy the cache metadata to the primaryKey

val cachedPrimaryKey = cachedValue map { primaryKey(\_) }

cachedPrimaryKey.value match {

case Some(pKey) =>

// if a value can be derived from the key, use the underlying cache to add it

// the underlying cache will create the secondary index as a side-effect

underlyingCache.add(pKey, cachedValue)

case None =>

// otherwise, we're just writing a tombstone, so we need to check if it exists

forwardingCache.add(key, cachedPrimaryKey)

}

}

override def lockAndSet(

key: K,

handler: LockingCache.Handler[Cached[V]]

): Future[Option[Cached[V]]] = {

handler(None) match {

case Some(cachedValue) =>

cachedValue.value match {

case Some(value) =>

// set on the underlying cache, and let it take care of adding

// the secondary index

val pKey = primaryKey(value)

lockingUnderlying.lockAndSet(pKey, handler)

case None =>

// no underlying value to set, so just write the forwarding entry.

// secondaryIndexingCache doesn't lock for this set, so there's

// no point in our doing it. There's a slight risk of writing an

// errant tombstone in a race, but the only way to get around this

// would be to lock around \*all\* primary and secondary indexes,

// which could produce deadlocks, which is probably worse.

val cachedEmptyPKey = cachedValue.copy[F](value = None)

forwardingCache.set(key, cachedEmptyPKey) map { \_ =>

Some(cachedValue)

}

}

case None =>

// nothing to do here

Future.value(None)

}

}

override def set(key: K, cachedValue: Cached[V]): Future[Unit] = {

cachedValue.value match {

case Some(value) =>

// set on the underlying cache, and let it take care of adding

// the secondary index

val pKey = primaryKey(value)

underlyingCache.set(pKey, cachedValue)

case None =>

// no underlying value to set, so just write the forwarding entry

forwardingCache.set(key, cachedValue.copy[F](value = None))

}

}

override def replace(key: K, cachedValue: Cached[V]): Future[Boolean] = {

cachedValue.value match {

case Some(value) =>

// replace in the underlying cache, and let it take care of adding the secondary index

val pKey = primaryKey(value)

underlyingCache.replace(pKey, cachedValue)

case None =>

// no underlying value to set, so just write the forwarding entry

forwardingCache.replace(key, cachedValue.copy[F](value = None))

}

}

override def delete(key: K): Future[Boolean] = {

forwardingCache.delete(key)

}

override def release(): Unit = {

forwardingCache.release()

underlyingCache.release()

}

}