package com.twitter.servo.cache

import com.twitter.servo.keyvalue.\_

import com.twitter.servo.util.{OptionOrdering, TryOrdering}

import com.twitter.util.{Future, Return, Throw, Time, Try}

object SimpleReplicatingCache {

/\*\*

\* Builds a SimpleReplicatingCache that writes a value multiple times to the same underlying

\* cache but under different keys. If the underlying cache is backed by enough shards, there

\* is a good chance that the different keys will end up on different shards, giving you similar

\* behavior to having multiple distinct caches.

\*/

def apply[K, K2, V](

underlying: LockingCache[K2, Cached[V]],

keyReplicator: (K, Int) => K2,

replicas: Int = 2

) = new SimpleReplicatingCache(

(0 until replicas).toSeq map { replica =>

new KeyTransformingLockingCache(

underlying,

(key: K) => keyReplicator(key, replica)

)

}

)

}

/\*\*

\* A very simple replicating cache implementation. It writes the same key/value pair to

\* multiple underlying caches. On read, each underlying cache is queried with the key; if the

\* results are not all the same for a given key, then the most recent value is chosen and

\* replicated to all caches.

\*

\* Some cache operations are not currently supported, because their semantics are a little fuzzy

\* in the replication case. Specifically: add and checkAndSet.

\*/

class SimpleReplicatingCache[K, V](underlyingCaches: Seq[LockingCache[K, Cached[V]]])

extends LockingCache[K, Cached[V]] {

private type CsValue = (Try[Cached[V]], Checksum)

private val cachedOrdering = new Ordering[Cached[V]] {

// sort by ascending timestamp

def compare(a: Cached[V], b: Cached[V]) = a.cachedAt.compare(b.cachedAt)

}

private val csValueOrdering = new Ordering[CsValue] {

// order by Try[V], ignore checksum

val subordering = TryOrdering(cachedOrdering)

def compare(a: CsValue, b: CsValue) = subordering.compare(a.\_1, b.\_1)

}

private val tryOptionCsValueOrdering = TryOrdering(OptionOrdering(csValueOrdering))

private val tryOptionCachedOrdering = TryOrdering(OptionOrdering(cachedOrdering))

/\*\*

\* release any underlying resources

\*/

def release(): Unit = {

underlyingCaches foreach { \_.release() }

}

/\*\*

\* Fetches from all underlying caches in parallel, and if results differ, will choose a

\* winner and push updated results back to the stale caches.

\*/

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

getWithChecksum(keys) map { csKvRes =>

val resBldr = new KeyValueResultBuilder[K, Cached[V]]

csKvRes.found foreach {

case (k, (Return(v), \_)) => resBldr.addFound(k, v)

case (k, (Throw(t), \_)) => resBldr.addFailed(k, t)

}

resBldr.addNotFound(csKvRes.notFound)

resBldr.addFailed(csKvRes.failed)

resBldr.result()

}

}

/\*\*

\* Fetches from all underlying caches in parallel, and if results differ, will choose a

\* winner and push updated results back to the stale caches.

\*/

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

Future.collect {

underlyingCaches map { underlying =>

underlying.getWithChecksum(keys)

}

} map { underlyingResults =>

val resBldr = new KeyValueResultBuilder[K, CsValue]

for (key <- keys) {

val keyResults = underlyingResults map { \_(key) }

resBldr(key) = getAndReplicate(key, keyResults) map {

// treat evictions as misses

case Some((Return(c), \_)) if c.status == CachedValueStatus.Evicted => None

case v => v

}

}

resBldr.result()

}

}

/\*\*

\* Looks at all the returned values for a given set of replication keys, returning the most recent

\* cached value if available, or indicate a miss if applicable, or return a failure if all

\* keys failed. If a cached value is returned, and some keys don't have that cached value,

\* the cached value will be replicated to those keys, possibly overwriting stale data.

\*/

private def getAndReplicate(

key: K,

keyResults: Seq[Try[Option[CsValue]]]

): Try[Option[CsValue]] = {

val max = keyResults.max(tryOptionCsValueOrdering)

max match {

// if one of the replication keys returned a cached value, then make sure all replication

// keys contain that cached value.

case Return(Some((Return(cached), cs))) =>

for ((underlying, keyResult) <- underlyingCaches zip keyResults) {

if (keyResult != max) {

replicate(key, cached, keyResult, underlying)

}

}

case \_ =>

}

max

}

private def replicate(

key: K,

cached: Cached[V],

current: Try[Option[CsValue]],

underlying: LockingCache[K, Cached[V]]

): Future[Unit] = {

current match {

case Throw(\_) =>

// if we failed to read a particular value, we don't want to write to that key

// because that key could potentially have the real newest value

Future.Unit

case Return(None) =>

// add rather than set, and fail if another value is written first

underlying.add(key, cached).unit

case Return(Some((\_, cs))) =>

underlying.checkAndSet(key, cached, cs).unit

}

}

/\*\*

\* Currently not supported. Use set or lockAndSet.

\*/

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

Future.exception(new UnsupportedOperationException("use set or lockAndSet"))

}

/\*\*

\* Currently not supported.

\*/

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

Future.exception(new UnsupportedOperationException("use set or lockAndSet"))

}

/\*\*

\* Calls set on all underlying caches. If at least one set succeeds, Future.Unit is

\* returned. If all fail, a Future.exception will be returned.

\*/

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

liftAndCollect {

underlyingCaches map { \_.set(key, value) }

} flatMap { seqTryUnits =>

// return Future.Unit if any underlying call succeeded, otherwise return

// the first failure.

if (seqTryUnits exists { \_.isReturn })

Future.Unit

else

Future.const(seqTryUnits.head)

}

}

/\*\*

\* Calls lockAndSet on the underlying cache for all replication keys. If at least one

\* underlying call succeeds, a successful result will be returned.

\*/

def lockAndSet(key: K, handler: LockingCache.Handler[Cached[V]]): Future[Option[Cached[V]]] = {

liftAndCollect {

underlyingCaches map { \_.lockAndSet(key, handler) }

} flatMap { seqTryOptionCached =>

Future.const(seqTryOptionCached.max(tryOptionCachedOrdering))

}

}

/\*\*

\* Returns Future(true) if any of the underlying caches return Future(true); otherwise,

\* returns Future(false) if any of the underlying caches return Future(false); otherwise,

\* returns the first failure.

\*/

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

liftAndCollect {

underlyingCaches map { \_.replace(key, value) }

} flatMap { seqTryBools =>

if (seqTryBools.contains(Return.True))

Future.value(true)

else if (seqTryBools.contains(Return.False))

Future.value(false)

else

Future.const(seqTryBools.head)

}

}

/\*\*

\* Performing an actual deletion on the underlying caches is not a good idea in the face

\* of potential failure, because failing to remove all values would allow a cached value to

\* be resurrected. Instead, delete actually does a replace on the underlying caches with a

\* CachedValueStatus of Evicted, which will be treated as a miss on read.

\*/

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

replace(key, Cached(None, CachedValueStatus.Evicted, Time.now))

}

/\*\*

\* Convets a Seq[Future[A]] into a Future[Seq[Try[A]]], isolating failures into Trys, instead

\* of allowing the entire Future to failure.

\*/

private def liftAndCollect[A](seq: Seq[Future[A]]): Future[Seq[Try[A]]] = {

Future.collect { seq map { \_ transform { Future(\_) } } }

}

}