package com.twitter.servo.cache

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Return

import com.twitter.util.Throw

/\*\*

\* MigratingReadCache supports a gradual migration from one cache to another. Reads from the

\* cache are compared to reads from the darkCache and new values are written to the darkCache

\* if necessary.

\*/

class MigratingReadCache[K, V](

cache: ReadCache[K, V],

darkCache: Cache[K, V],

statsReceiver: StatsReceiver = NullStatsReceiver)

extends ReadCache[K, V] {

private[this] val scopedStatsReceiver = statsReceiver.scope("migrating\_read\_cache")

private[this] val getScope = scopedStatsReceiver.scope("get")

private[this] val getMismatchedResultsCounter = getScope.counter("mismatched\_results")

private[this] val getMissingResultsCounter = getScope.counter("missing\_results")

private[this] val getUnexpectedResultsCounter = getScope.counter("unexpected\_results")

private[this] val getMatchingResultsCounter = getScope.counter("matching\_results")

private[this] val getWithChecksumScope = scopedStatsReceiver.scope("get\_with\_cheksum")

private[this] val getWithChecksumMismatchedResultsCounter =

getWithChecksumScope.counter("mismatched\_results")

private[this] val getWithChecksumMissingResultsCounter =

getWithChecksumScope.counter("missing\_results")

private[this] val getWithChecksumUnexpectedResultsCounter =

getWithChecksumScope.counter("unexpected\_results")

private[this] val getWithChecksumMatchingResultsCounter =

getWithChecksumScope.counter("matching\_results")

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

cache.get(keys) onSuccess { result =>

darkCache.get(keys) onSuccess { darkResult =>

keys foreach { k =>

(result(k), darkResult(k)) match {

// compare values, set if they differ

case (Return(Some(v)), Return(Some(dv))) if (v != dv) =>

getMismatchedResultsCounter.incr()

darkCache.set(k, v)

// set a value if missing

case (Return(Some(v)), Return.None | Throw(\_)) =>

getMissingResultsCounter.incr()

darkCache.set(k, v)

// remove if necessary

case (Return.None, Return(Some(\_)) | Throw(\_)) =>

getUnexpectedResultsCounter.incr()

darkCache.delete(k)

// do nothing otherwise

case \_ =>

getMatchingResultsCounter.incr()

()

}

}

}

}

}

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

cache.getWithChecksum(keys) onSuccess { result =>

// no point in the getWithChecksum from the darkCache

darkCache.get(keys) onSuccess { darkResult =>

keys foreach { k =>

(result(k), darkResult(k)) match {

// compare values, set if they differ

case (Return(Some((Return(v), \_))), Return(Some(dv))) if (v != dv) =>

getWithChecksumMismatchedResultsCounter.incr()

darkCache.set(k, v)

// set a value if missing

case (Return(Some((Return(v), \_))), Return.None | Throw(\_)) =>

getWithChecksumMissingResultsCounter.incr()

darkCache.set(k, v)

// remove if necessary

case (Return.None, Return(Some(\_)) | Throw(\_)) =>

getWithChecksumUnexpectedResultsCounter.incr()

darkCache.delete(k)

// do nothing otherwise

case \_ =>

getWithChecksumMatchingResultsCounter.incr()

()

}

}

}

}

}

override def release(): Unit = {

cache.release()

darkCache.release()

}

}

/\*\*

\* MigratingCache supports a gradual migration from one cache to another. Writes to the cache

\* are propogated to the darkCache. Reads from the cache are compared to reads from the darkCache

\* and new values are written to the darkCache if necessary.

\*

\* Writes to the darkCache are not locking writes, so there is some risk of inconsistencies from

\* race conditions. However, writes to the darkCache only occur if they succeed in the cache, so

\* if a checkAndSet fails, for example, no write is issued to the darkCache.

\*/

class MigratingCache[K, V](

cache: Cache[K, V],

darkCache: Cache[K, V],

statsReceiver: StatsReceiver = NullStatsReceiver)

extends MigratingReadCache(cache, darkCache, statsReceiver)

with Cache[K, V] {

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

cache.add(key, value) onSuccess { wasAdded =>

if (wasAdded) {

darkCache.set(key, value)

}

}

}

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

cache.checkAndSet(key, value, checksum) onSuccess { wasSet =>

if (wasSet) {

darkCache.set(key, value)

}

}

}

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

cache.set(key, value) onSuccess { \_ =>

darkCache.set(key, value)

}

}

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

cache.replace(key, value) onSuccess { wasReplaced =>

if (wasReplaced) {

darkCache.set(key, value)

}

}

}

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

cache.delete(key) onSuccess { wasDeleted =>

if (wasDeleted) {

darkCache.delete(key)

}

}

}

}

/\*\*

\* Like MigratingCache but for TtlCaches

\*/

class MigratingTtlCache[K, V](

cache: TtlCache[K, V],

darkCache: TtlCache[K, V],

ttl: (K, V) => Duration)

extends MigratingReadCache(cache, new TtlCacheToCache(darkCache, ttl))

with TtlCache[K, V] {

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

cache.add(key, value, ttl) onSuccess { wasAdded =>

if (wasAdded) {

darkCache.set(key, value, ttl)

}

}

}

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

cache.checkAndSet(key, value, checksum, ttl) onSuccess { wasSet =>

if (wasSet) {

darkCache.set(key, value, ttl)

}

}

}

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

cache.set(key, value, ttl) onSuccess { \_ =>

darkCache.set(key, value, ttl)

}

}

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

cache.replace(key, value, ttl) onSuccess { wasReplaced =>

if (wasReplaced) {

darkCache.set(key, value, ttl)

}

}

}

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

cache.delete(key) onSuccess { wasDeleted =>

if (wasDeleted) {

darkCache.delete(key)

}

}

}

override def release(): Unit = {

cache.release()

darkCache.release()

}

}

/\*\*

\* A MigratingTtlCache for Memcaches, implementing a migrating incr and decr. Race conditions

\* are possible and may prevent the counts from being perfectly synchronized.

\*/

class MigratingMemcache(

cache: Memcache,

darkCache: Memcache,

ttl: (String, Array[Byte]) => Duration)

extends MigratingTtlCache[String, Array[Byte]](cache, darkCache, ttl)

with Memcache {

def incr(key: String, delta: Long = 1): Future[Option[Long]] = {

cache.incr(key, delta) onSuccess {

case None =>

darkCache.delete(key)

case Some(value) =>

darkCache.incr(key, delta) onSuccess {

case Some(`value`) => // same value!

case \_ =>

val b = value.toString.getBytes

darkCache.set(key, b, ttl(key, b))

}

}

}

def decr(key: String, delta: Long = 1): Future[Option[Long]] = {

cache.decr(key, delta) onSuccess {

case None =>

darkCache.delete(key)

case Some(value) =>

darkCache.decr(key, delta) onSuccess {

case Some(`value`) => // same value!

case \_ =>

val b = value.toString.getBytes

darkCache.set(key, b, ttl(key, b))

}

}

}

}