package com.twitter.servo.repository

import com.twitter.logging.{Level, Logger}

import com.twitter.servo.cache.\_

import com.twitter.servo.util.{Effect, Gate, RateLimitingLogger}

import com.twitter.util.\_

import scala.collection.mutable

import scala.util.Random

/\*\*

\* A set of classes that indicate how to handle cached results.

\*/

sealed abstract class CachedResultAction[+V]

object CachedResultAction {

/\*\* Indicates a key should be fetched from the underlying repo \*/

case object HandleAsMiss extends CachedResultAction[Nothing]

/\*\* Indicates a key should be returned as not-found, and not fetched from the underlying repo \*/

case object HandleAsNotFound extends CachedResultAction[Nothing]

/\*\* Indicates the value should be returned as found \*/

case class HandleAsFound[V](value: V) extends CachedResultAction[V]

/\*\* Indicates the value should not be cached \*/

case object HandleAsDoNotCache extends CachedResultAction[Nothing]

/\*\* Indicates that the given action should be applied, and the given function applied to the resulting value \*/

case class TransformSubAction[V](action: CachedResultAction[V], f: V => V)

extends CachedResultAction[V]

/\*\* Indicates the key should be returned as a failure \*/

case class HandleAsFailed(t: Throwable) extends CachedResultAction[Nothing]

/\*\* Indicates that the value should be refetched asynchronously, be immediately treated

\* as the given action. \*/

case class SoftExpiration[V](action: CachedResultAction[V]) extends CachedResultAction[V]

}

/\*\*

\* A set of classes representing the various states for a cached result.

\*/

sealed abstract class CachedResult[+K, +V] {

def key: K

}

object CachedResult {

import CachedResultAction.\_

/\*\* Indicates the key was not in cache \*/

case class NotFound[K](key: K) extends CachedResult[K, Nothing]

/\*\* Indicates there was an error fetching the key \*/

case class Failed[K](key: K, t: Throwable) extends CachedResult[K, Nothing]

/\*\* Indicates the cached value could not be deserialized \*/

case class DeserializationFailed[K](key: K) extends CachedResult[K, Nothing]

/\*\* Indicates the cached value could not be serialized \*/

case class SerializationFailed[K](key: K) extends CachedResult[K, Nothing]

/\*\* Indicates that a NotFound tombstone was found in cached \*/

case class CachedNotFound[K](

key: K,

cachedAt: Time,

softTtlStep: Option[Short] = None)

extends CachedResult[K, Nothing]

/\*\* Indicates that a Deleted tombstone was found in cached \*/

case class CachedDeleted[K](

key: K,

cachedAt: Time,

softTtlStep: Option[Short] = None)

extends CachedResult[K, Nothing]

/\*\* Indicates that value was found in cached \*/

case class CachedFound[K, V](

key: K,

value: V,

cachedAt: Time,

softTtlStep: Option[Short] = None)

extends CachedResult[K, V]

/\*\* Indicates that value should not be cached until \*/

case class DoNotCache[K](key: K, until: Option[Time]) extends CachedResult[K, Nothing]

type Handler[K, V] = CachedResult[K, V] => CachedResultAction[V]

type PartialHandler[K, V] = CachedResult[K, V] => Option[CachedResultAction[V]]

type HandlerFactory[Q, K, V] = Q => Handler[K, V]

/\*\*

\* companion object for Handler type

\*/

object Handler {

/\*\*

\* terminate a PartialHandler to produce a new Handler

\*/

def apply[K, V](

partial: PartialHandler[K, V],

handler: Handler[K, V] = defaultHandler[K, V]

): Handler[K, V] = { cachedResult =>

partial(cachedResult) match {

case Some(s) => s

case None => handler(cachedResult)

}

}

}

/\*\*

\* companion object for PartialHandler type

\*/

object PartialHandler {

/\*\*

\* Sugar to produce a PartialHandler from a PartialFunction. Successive calls to

\* isDefined MUST return the same result. Otherwise, take the syntax hit and wire

\* up your own PartialHandler.

\*/

def apply[K, V](

partial: PartialFunction[CachedResult[K, V], CachedResultAction[V]]

): PartialHandler[K, V] = partial.lift

/\*\*

\* chain one PartialHandler after another to produce a new PartialHandler

\*/

def orElse[K, V](

thisHandler: PartialHandler[K, V],

thatHandler: PartialHandler[K, V]

): PartialHandler[K, V] = { cachedResult =>

thisHandler(cachedResult) match {

case some @ Some(\_) => some

case None => thatHandler(cachedResult)

}

}

}

/\*\*

\* companion object for HandlerFactory type

\*/

object HandlerFactory {

def apply[Q, K, V](handler: Handler[K, V]): HandlerFactory[Q, K, V] = \_ => handler

}

def defaultHandlerFactory[Q, K, V]: HandlerFactory[Q, K, V] =

HandlerFactory[Q, K, V](defaultHandler)

/\*\*

\* This is the default Handler. Failures are treated as misses.

\*/

def defaultHandler[K, V]: Handler[K, V] = {

case NotFound(\_) | Failed(\_, \_) => HandleAsMiss

case DeserializationFailed(\_) | SerializationFailed(\_) => HandleAsMiss

case CachedNotFound(\_, \_, \_) | CachedDeleted(\_, \_, \_) => HandleAsNotFound

case CachedFound(\_, value, \_, \_) => HandleAsFound(value)

case DoNotCache(\_, Some(time)) if Time.now > time => HandleAsMiss

case DoNotCache(\_, \_) => HandleAsDoNotCache

}

/\*\*

\* A PartialHandler that bubbles memcache failures up instead of converting

\* those failures to misses.

\*/

def failuresAreFailures[K, V] = PartialHandler[K, V] {

case Failed(\_, t) => HandleAsFailed(t)

}

/\*\*

\* A PartialHandler that doesn't attempt to write back to cache if the initial

\* cache read failed, but still fetches from the underlying repo.

\*/

def failuresAreDoNotCache[K, V] = PartialHandler[K, V] {

case Failed(\_, \_) => HandleAsDoNotCache

}

/\*\*

\* A function that takes a cachedAt time and ttl, and returns an expiry time. This function

\* \_must\_ be deterministic with respect to the arguments provided, otherwise, you might get a

\* MatchError when using this with softTtlExpiration.

\*/

type Expiry = (Time, Duration) => Time

/\*\*

\* An Expiry function with an epsilon of zero.

\*/

val fixedExpiry: Expiry = (cachedAt: Time, ttl: Duration) => cachedAt + ttl

/\*\*

\* A repeatable "random" expiry function that perturbs the ttl with a random value

\* no greater than +/-(ttl \* maxFactor).

\*/

def randomExpiry(maxFactor: Float): Expiry = {

if (maxFactor == 0) {

fixedExpiry

} else { (cachedAt: Time, ttl: Duration) =>

{

val factor = (2 \* new Random(cachedAt.inMilliseconds).nextFloat - 1) \* maxFactor

cachedAt + ttl + Duration.fromNanoseconds((factor \* ttl.inNanoseconds).toLong)

}

}

}

/\*\*

\* soft-expires CachedFound and CachedNotFound based on a ttl.

\*

\* @param ttl

\* values older than this will be considered expired, but still

\* returned, and asynchronously refreshed in cache.

\* @param expiry

\* (optional) function to compute the expiry time

\*/

def softTtlExpiration[K, V](

ttl: Duration,

expiry: Expiry = fixedExpiry

): PartialHandler[K, V] =

softTtlExpiration(\_ => ttl, expiry)

/\*\*

\* soft-expires CachedFound and CachedNotFound based on a ttl derived from the value

\*

\* @param ttl

\* values older than this will be considered expired, but still

\* returned, and asynchronously refreshed in cache.

\* @param expiry

\* (optional) function to compute the expiry time

\*/

def softTtlExpiration[K, V](

ttl: Option[V] => Duration,

expiry: Expiry

): PartialHandler[K, V] = PartialHandler[K, V] {

case CachedFound(\_, value, cachedAt, \_) if expiry(cachedAt, ttl(Some(value))) < Time.now =>

SoftExpiration(HandleAsFound(value))

case CachedNotFound(\_, cachedAt, \_) if expiry(cachedAt, ttl(None)) < Time.now =>

SoftExpiration(HandleAsNotFound)

}

/\*\*

\* soft-expires CachedFound and CachedNotFound based on a ttl derived from both the value

\* and the softTtlStep

\*

\* @param ttl

\* values older than this will be considered expired, but still returned, and

\* asynchronously refreshed in cache.

\* @param expiry

\* (optional) function to compute the expiry time

\*/

def steppedSoftTtlExpiration[K, V](

ttl: (Option[V], Option[Short]) => Duration,

expiry: Expiry = fixedExpiry

): PartialHandler[K, V] = PartialHandler[K, V] {

case CachedFound(\_, value, cachedAt, softTtlStep)

if expiry(cachedAt, ttl(Some(value), softTtlStep)) < Time.now =>

SoftExpiration(HandleAsFound(value))

case CachedNotFound(\_, cachedAt, softTtlStep)

if expiry(cachedAt, ttl(None, softTtlStep)) < Time.now =>

SoftExpiration(HandleAsNotFound)

case CachedDeleted(\_, cachedAt, softTtlStep)

if expiry(cachedAt, ttl(None, softTtlStep)) < Time.now =>

SoftExpiration(HandleAsNotFound)

}

/\*\*

\* hard-expires CachedFound and CachedNotFound based on a ttl.

\*

\* @param ttl

\* values older than this will be considered a miss

\* @param expiry

\* (optional) function to compute the expiry time

\*/

def hardTtlExpiration[K, V](

ttl: Duration,

expiry: Expiry = fixedExpiry

): PartialHandler[K, V] =

hardTtlExpiration(\_ => ttl, expiry)

/\*\*

\* hard-expires CachedFound and CachedNotFound based on a ttl derived from the value

\*

\* @param ttl

\* values older than this will be considered a miss

\* @param expiry

\* (optional) function to compute the expiry time

\*/

def hardTtlExpiration[K, V](

ttl: Option[V] => Duration,

expiry: Expiry

): PartialHandler[K, V] = PartialHandler[K, V] {

case CachedFound(\_, value, cachedAt, \_) if expiry(cachedAt, ttl(Some(value))) < Time.now =>

HandleAsMiss

case CachedNotFound(\_, cachedAt, \_) if expiry(cachedAt, ttl(None)) < Time.now =>

HandleAsMiss

}

/\*\*

\* hard-expires a CachedNotFound tombstone based on a ttl

\*

\* @param ttl

\* values older than this will be considered expired

\* @param expiry

\* (optional) function to compute the expiry time

\*/

def notFoundHardTtlExpiration[K, V](

ttl: Duration,

expiry: Expiry = fixedExpiry

): PartialHandler[K, V] = PartialHandler[K, V] {

case CachedNotFound(\_, cachedAt, \_) =>

if (expiry(cachedAt, ttl) < Time.now)

HandleAsMiss

else

HandleAsNotFound

}

/\*\*

\* hard-expires a CachedDeleted tombstone based on a ttl

\*

\* @param ttl

\* values older than this will be considered expired

\* @param expiry

\* (optional) function to compute the expiry time

\*/

def deletedHardTtlExpiration[K, V](

ttl: Duration,

expiry: Expiry = fixedExpiry

): PartialHandler[K, V] = PartialHandler[K, V] {

case CachedDeleted(\_, cachedAt, \_) =>

if (expiry(cachedAt, ttl) < Time.now)

HandleAsMiss

else

HandleAsNotFound

}

/\*\*

\* read only from cache, never fall back to underlying KeyValueRepository

\*/

def cacheOnly[K, V]: Handler[K, V] = {

case CachedFound(\_, value, \_, \_) => HandleAsFound(value)

case \_ => HandleAsNotFound

}

/\*\*

\* use either primary or backup Handler, depending on usePrimary result

\*

\* @param primaryHandler

\* the handler to be used if usePrimary evaluates to true

\* @param backupHandler

\* the handle to be used if usePrimary evaluates to false

\* @param usePrimary

\* evaluates the query to determine which handler to use

\*/

def switchedHandlerFactory[Q, K, V](

primaryHandler: Handler[K, V],

backupHandler: Handler[K, V],

usePrimary: Q => Boolean

): HandlerFactory[Q, K, V] = { query =>

if (usePrimary(query))

primaryHandler

else

backupHandler

}

}

object CacheResultObserver {

case class CachingRepositoryResult[K, V](

resultFromCache: KeyValueResult[K, Cached[V]],

resultFromCacheMissReadthrough: KeyValueResult[K, V],

resultFromSoftTtlReadthrough: KeyValueResult[K, V])

def unit[K, V] = Effect.unit[CachingRepositoryResult[K, V]]

}

object CachingKeyValueRepository {

type CacheResultObserver[K, V] = Effect[CacheResultObserver.CachingRepositoryResult[K, V]]

}

/\*\*

\* Reads keyed values from a LockingCache, and reads through to an underlying

\* KeyValueRepository for misses. supports a "soft ttl", beyond which values

\* will be read through out-of-band to the originating request

\*

\* @param underlying

\* the underlying KeyValueRepository

\* @param cache

\* the locking cache to read from

\* @param newQuery

\* a function for converting a subset of the keys of the original query into a new

\* query. this is used to construct the query passed to the underlying repository

\* to fetch the cache misses.

\* @param handlerFactory

\* A factory to produce functions that specify policies about how to handle results

\* from cache. (i.e. to handle failures as misses vs failures, etc)

\* @param picker

\* used to choose between the value in cache and the value read from the DB when

\* storing values in the cache

\* @param observer

\* a CacheObserver for collecting cache statistics\*

\* @param writeSoftTtlStep

\* Write the soft\_ttl\_step value to indicate number of consistent reads from underlying store

\* @param cacheResultObserver

\* An [[Effect]] of type [[CacheResultObserver.CachingRepositoryResult]] which is useful for examining

\* the results from the cache, underlying storage, and any later read-throughs. The effect is

\* executed asynchronously from the request path and has no bearing on the Future[KeyValueResult]\*

\* returned from this Repository.

\*/

class CachingKeyValueRepository[Q <: Seq[K], K, V](

underlying: KeyValueRepository[Q, K, V],

val cache: LockingCache[K, Cached[V]],

newQuery: SubqueryBuilder[Q, K],

handlerFactory: CachedResult.HandlerFactory[Q, K, V] =

CachedResult.defaultHandlerFactory[Q, K, V],

picker: LockingCache.Picker[Cached[V]] = new PreferNewestCached[V]: PreferNewestCached[V],

observer: CacheObserver = NullCacheObserver,

writeSoftTtlStep: Gate[Unit] = Gate.False,

cacheResultObserver: CachingKeyValueRepository.CacheResultObserver[K, V] =

CacheResultObserver.unit[K, V]: Effect[CacheResultObserver.CachingRepositoryResult[K, V]])

extends KeyValueRepository[Q, K, V] {

import CachedResult.\_

import CachedResultAction.\_

protected[this] val log = Logger.get(getClass.getSimpleName)

private[this] val rateLimitedLogger = new RateLimitingLogger(logger = log)

protected[this] val effectiveCacheStats = observer.scope("effective")

/\*\*

\* Calculates the softTtlStep based on result from cache and underlying store.

\* The softTtlStep indicates how many times we have

\* performed & recorded a consistent read-through.

\* A value of None is equivalent to Some(0) - it indicates zero consistent read-throughs.

\*/

protected[this] def updateSoftTtlStep(

underlyingResult: Option[V],

cachedResult: Cached[V]

): Option[Short] = {

if (writeSoftTtlStep() && underlyingResult == cachedResult.value) {

cachedResult.softTtlStep match {

case Some(step) if step < Short.MaxValue => Some((step + 1).toShort)

case Some(step) if step == Short.MaxValue => cachedResult.softTtlStep

case \_ => Some(1)

}

} else {

None

}

}

protected case class ProcessedCacheResult(

hits: Map[K, V],

misses: Seq[K],

doNotCache: Set[K],

failures: Map[K, Throwable],

tombstones: Set[K],

softExpirations: Seq[K],

transforms: Map[K, (V => V)])

override def apply(keys: Q): Future[KeyValueResult[K, V]] = {

getFromCache(keys).flatMap { cacheResult =>

val ProcessedCacheResult(

hits,

misses,

doNotCache,

failures,

tombstones,

softExpirations,

transforms

) =

process(keys, cacheResult)

if (log.isLoggable(Level.TRACE)) {

log.trace(

"CachingKVR.apply keys %d hit %d miss %d noCache %d failure %d " +

"tombstone %d softexp %d",

keys.size,

hits.size,

misses.size,

doNotCache.size,

failures.size,

tombstones.size,

softExpirations.size

)

}

recordCacheStats(

keys,

notFound = misses.toSet,

doNotCache = doNotCache,

expired = softExpirations.toSet,

numFailures = failures.size,

numTombstones = tombstones.size

)

// now read through all notFound

val underlyingQuery = newQuery(misses ++ doNotCache, keys)

val writeToCacheQuery = if (doNotCache.nonEmpty) newQuery(misses, keys) else underlyingQuery

val futureFromUnderlying = readThrough(underlyingQuery, writeToCacheQuery)

// async read-through for the expired results, ignore results

val softExpirationQuery = newQuery(softExpirations, keys)

val futureFromSoftExpiry = readThrough(softExpirationQuery, softExpirationQuery, cacheResult)

// merge all results together

for {

fromUnderlying <- futureFromUnderlying

fromCache = KeyValueResult(hits, tombstones, failures)

fromUnderlyingTransformed = transformResults(fromUnderlying, transforms)

} yield {

futureFromSoftExpiry.onSuccess { readThroughResults =>

cacheResultObserver(

CacheResultObserver.CachingRepositoryResult(

cacheResult,

fromUnderlyingTransformed,

readThroughResults

)

)

}

KeyValueResult.sum(Seq(fromCache, fromUnderlyingTransformed))

}

}

}

/\*\*

\* Given results and a map of keys to transform functions, apply those transform functions

\* to the found results.

\*/

protected[this] def transformResults(

results: KeyValueResult[K, V],

transforms: Map[K, (V => V)]

): KeyValueResult[K, V] = {

if (transforms.isEmpty) {

results

} else {

results.copy(found = results.found.map {

case (key, value) =>

(key, transforms.get(key).map(\_(value)).getOrElse(value))

})

}

}

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

val uniqueKeys = keys.distinct

cache.get(uniqueKeys) handle {

case t: Throwable =>

rateLimitedLogger.logThrowable(t, "exception caught in cache get")

// treat total cache failure as a fetch that returned all failures

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

}

}

/\*\*

\* Buckets cache results according to the wishes of the CachedResultHandler

\*/

protected[this] def process(

keys: Q,

cacheResult: KeyValueResult[K, Cached[V]]

): ProcessedCacheResult = {

val cachedResultHandler = handlerFactory(keys)

val hits = Map.newBuilder[K, V]

val misses = new mutable.ArrayBuffer[K]

val failures = Map.newBuilder[K, Throwable]

val tombstones = Set.newBuilder[K]

val softExpiredKeys = new mutable.ListBuffer[K]

val doNotCache = Set.newBuilder[K]

val transforms = Map.newBuilder[K, (V => V)]

for (key <- keys) {

val cachedResult = cacheResult(key) match {

case Throw(t) => Failed(key, t)

case Return(None) => NotFound(key)

case Return(Some(cached)) =>

cached.status match {

case CachedValueStatus.Found =>

cached.value match {

case None => NotFound(key)

case Some(value) =>

CachedFound(

key,

value,

cached.cachedAt,

cached.softTtlStep

)

}

case CachedValueStatus.NotFound => CachedNotFound(key, cached.cachedAt)

case CachedValueStatus.Deleted => CachedDeleted(key, cached.cachedAt)

case CachedValueStatus.SerializationFailed => SerializationFailed(key)

case CachedValueStatus.DeserializationFailed => DeserializationFailed(key)

case CachedValueStatus.Evicted => NotFound(key)

case CachedValueStatus.DoNotCache => DoNotCache(key, cached.doNotCacheUntil)

}

}

def processAction(action: CachedResultAction[V]): Unit = {

action match {

case HandleAsMiss => misses += key

case HandleAsFound(value) => hits += key -> value

case HandleAsNotFound => tombstones += key

case HandleAsDoNotCache => doNotCache += key

case HandleAsFailed(t) => failures += key -> t

case TransformSubAction(subAction, f) =>

transforms += key -> f

processAction(subAction)

case SoftExpiration(subAction) =>

softExpiredKeys += key

processAction(subAction)

}

}

processAction(cachedResultHandler(cachedResult))

}

ProcessedCacheResult(

hits.result(),

misses,

doNotCache.result(),

failures.result(),

tombstones.result(),

softExpiredKeys,

transforms.result()

)

}

protected[this] def recordCacheStats(

keys: Seq[K],

notFound: Set[K],

doNotCache: Set[K],

expired: Set[K],

numFailures: Int,

numTombstones: Int

): Unit = {

keys.foreach { key =>

val wasntFound = notFound.contains(key)

val keyString = key.toString

if (wasntFound || expired.contains(key))

effectiveCacheStats.miss(keyString)

else

effectiveCacheStats.hit(keyString)

if (wasntFound)

observer.miss(keyString)

else

observer.hit(keyString)

}

observer.expired(expired.size)

observer.failure(numFailures)

observer.tombstone(numTombstones)

observer.noCache(doNotCache.size)

}

/\*\*

\* read through to the underlying repository

\*

\* @param cacheKeys

\* the keys to read and cache

\*/

def readThrough(cacheKeys: Q): Future[KeyValueResult[K, V]] = {

readThrough(cacheKeys, cacheKeys)

}

/\*\*

\* read through to the underlying repository

\*

\* @param writeToCacheQuery

\* the query to pass to the writeToCache method after getting a result back from the

\* underlying repository. this query can be exactly the same as underlyingQuery if

\* all readThrough keys should be cached, or it may contain a subset of the keys if

\* some keys should not be written back to cache.

\* @param cacheResult

\* the current cache results for underlyingQuery.

\*/

def readThrough(

underlyingQuery: Q,

writeToCacheQuery: Q,

cacheResult: KeyValueResult[K, Cached[V]] = KeyValueResult.empty

): Future[KeyValueResult[K, V]] = {

if (underlyingQuery.isEmpty) {

KeyValueResult.emptyFuture

} else {

underlying(underlyingQuery).onSuccess { result =>

if (writeToCacheQuery.nonEmpty) {

writeToCache(writeToCacheQuery, result, cacheResult)

}

}

}

}

/\*\*

\* Writes the contents of the given KeyValueResult to cache.

\*/

def writeToCache(

keys: Q,

underlyingResult: KeyValueResult[K, V],

cacheResult: KeyValueResult[K, Cached[V]] = KeyValueResult[K, Cached[V]]()

): Unit = {

lazy val cachedEmpty = {

val now = Time.now

Cached[V](None, CachedValueStatus.NotFound, now, Some(now), softTtlStep = None)

}

keys.foreach { key =>

// only cache Returns from the underlying repo, skip Throws.

// iff cached value matches value from underlying store

// (for both NotFound and Found results), increment softTtlStep

// otherwise, set softTtlStep to None

underlyingResult(key) match {

case Return(optUnderlyingVal) =>

val softTtlStep =

cacheResult(key) match {

case Return(Some(cacheVal)) => updateSoftTtlStep(optUnderlyingVal, cacheVal)

case \_ => None

}

val status =

optUnderlyingVal match {

case Some(\_) => CachedValueStatus.Found

case None => CachedValueStatus.NotFound

}

val cached =

cachedEmpty.copy(

value = optUnderlyingVal,

status = status,

softTtlStep = softTtlStep

)

cache

.lockAndSet(key, LockingCache.PickingHandler(cached, picker))

.onFailure {

case t: Throwable =>

rateLimitedLogger.logThrowable(t, "exception caught in lockAndSet")

}

case Throw(\_) => None

}

}

}

}