package com.twitter.servo.cache

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.service.RetryPolicy

import com.twitter.finagle.partitioning.FailureAccrualException

import com.twitter.finagle.Backoff

import com.twitter.finagle.stats.{NullStatsReceiver, Stat, StatsReceiver}

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

import com.twitter.servo.util.{ExceptionCounter, RateLimitingLogger}

import com.twitter.util.\_

import scala.util.control.NoStackTrace

object LockingCache {

/\*\*

\* first argument is value to store, second argument is value in cache,

\* returns an Option of the value to be stored. None should be interpreted

\* as "don't store anything"

\*/

type Picker[V] = (V, V) => Option[V]

/\*\*

\* argument is value, if any, in cache.

\* return type is value, if any, to be stored in cache.

\* returning None means nothing will be done.

\*/

type Handler[V] = Option[V] => Option[V]

case class AlwaysSetHandler[V](value: Option[V]) extends Handler[V] {

override def apply(ignored: Option[V]) = value

}

case class PickingHandler[V](newValue: V, pick: Picker[V]) extends Handler[V] {

override def apply(inCache: Option[V]): Option[V] =

inCache match {

case None =>

// if nothing in cache, go ahead and store!

Some(newValue)

case Some(oldValue) =>

// if something in cache, store a picked value based on

// what's in cache and what's being stored

pick(newValue, oldValue)

}

// apparently case classes that extend functions don't get pretty toString methods

override lazy val toString = "PickingHandler(%s, %s)".format(newValue, pick)

}

case class UpdateOnlyPickingHandler[V](newValue: V, pick: Picker[V]) extends Handler[V] {

override def apply(inCache: Option[V]): Option[V] =

inCache match {

case None =>

// if nothing in cache, do not update

None

case Some(oldValue) =>

// if something in cache, store a picked value based on

// what's in cache and what's being stored

pick(newValue, oldValue)

}

// apparently case classes that extend functions don't get pretty toString methods

override lazy val toString = "UpdateOnlyPickingHandler(%s, %s)".format(newValue, pick)

}

}

trait LockingCacheFactory {

def apply[K, V](cache: Cache[K, V]): LockingCache[K, V]

def scope(scopes: String\*): LockingCacheFactory

}

/\*\*

\* A cache that enforces a consistent view of values between the time when a set

\* is initiated and when the value is actually updated in cache.

\*/

trait LockingCache[K, V] extends Cache[K, V] {

/\*\*

\* Look up a value and dispatch based on the result. The particular locking

\* approach is defined by the implementing class. May call handler multiple

\* times as part of more elaborate locking and retry looping.

\*

\* Overview of semantics:

\* `handler(None)` is called if no value is present in cache.

\* `handler(Some(value))` is called if a value is present.

\* `handler(x)` should return None if nothing should be done and `Some(value)`

\* if a value should be set.

\*

\* @return the value that was actually set

\*/

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

}

class OptimisticLockingCacheObserver(statsReceiver: StatsReceiver) {

import OptimisticLockingCache.\_

private[this] val scopedReceiver = statsReceiver.scope("locking\_cache")

private[this] val successCounter = scopedReceiver.counter("success")

private[this] val failureCounter = scopedReceiver.counter("failure")

private[this] val exceptionCounter = new ExceptionCounter(scopedReceiver)

private[this] val lockAndSetStat = scopedReceiver.stat("lockAndSet")

def time[V](f: => Future[Option[V]]): Future[Option[V]] = {

Stat.timeFuture(lockAndSetStat) {

f

}

}

def success(attempts: Seq[FailedAttempt]): Unit = {

successCounter.incr()

countAttempts(attempts)

}

def failure(attempts: Seq[FailedAttempt]): Unit = {

failureCounter.incr()

countAttempts(attempts)

}

def scope(s: String\*): OptimisticLockingCacheObserver =

s.toList match {

case Nil => this

case head :: tail =>

new OptimisticLockingCacheObserver(statsReceiver.scope(head)).scope(tail: \_\*)

}

private[this] def countAttempts(attempts: Seq[FailedAttempt]): Unit = {

attempts foreach { attempt =>

val name = attempt.getClass.getSimpleName

scopedReceiver.counter(name).incr()

attempt.maybeThrowable foreach { t =>

exceptionCounter(t)

scopedReceiver.scope(name).counter(t.getClass.getName).incr()

}

}

}

}

case class OptimisticLockingCacheFactory(

backoffs: Backoff,

observer: OptimisticLockingCacheObserver = new OptimisticLockingCacheObserver(NullStatsReceiver),

timer: Timer = new NullTimer,

// Enabling key logging may unintentionally cause inclusion of sensitive data

// in service logs and any accompanying log sinks such as Splunk. By default, this is disabled,

// however may be optionally enabled for the purpose of debugging. Caution is warranted.

enableKeyLogging: Boolean = false)

extends LockingCacheFactory {

def this(

backoffs: Backoff,

statsReceiver: StatsReceiver,

timer: Timer,

enableKeyLogging: Boolean

) = this(backoffs, new OptimisticLockingCacheObserver(statsReceiver), timer, enableKeyLogging)

override def apply[K, V](cache: Cache[K, V]): LockingCache[K, V] = {

new OptimisticLockingCache(cache, backoffs, observer, timer, enableKeyLogging)

}

override def scope(scopes: String\*): LockingCacheFactory = {

new OptimisticLockingCacheFactory(backoffs, observer.scope(scopes: \_\*), timer)

}

}

object OptimisticLockingCache {

private[this] val FutureNone = Future.value(None)

def emptyFutureNone[V] = FutureNone.asInstanceOf[Future[Option[V]]]

sealed abstract class FailedAttempt(val maybeThrowable: Option[Throwable])

extends Exception

with NoStackTrace

case class GetWithChecksumException(t: Throwable) extends FailedAttempt(Some(t))

case object GetWithChecksumEmpty extends FailedAttempt(None)

case object CheckAndSetFailed extends FailedAttempt(None)

case class CheckAndSetException(t: Throwable) extends FailedAttempt(Some(t))

case class AddException(t: Throwable) extends FailedAttempt(Some(t))

case class LockAndSetFailure(str: String, attempts: Seq[FailedAttempt])

extends Exception(

str,

// if the last exception was an RPC exception, try to recover the stack trace

attempts.lastOption.flatMap(\_.maybeThrowable).orNull

)

private def retryPolicy(backoffs: Backoff): RetryPolicy[Try[Nothing]] =

RetryPolicy.backoff(backoffs) {

case Throw(\_: FailureAccrualException) => false

case \_ => true

}

}

/\*\*

\* Implementation of a LockingCache using add/getWithChecksum/checkAndSet.

\*/

class OptimisticLockingCache[K, V](

override val underlyingCache: Cache[K, V],

retryPolicy: RetryPolicy[Try[Nothing]],

observer: OptimisticLockingCacheObserver,

timer: Timer,

enableKeyLogging: Boolean)

extends LockingCache[K, V]

with CacheWrapper[K, V] {

import LockingCache.\_

import OptimisticLockingCache.\_

def this(

underlyingCache: Cache[K, V],

retryPolicy: RetryPolicy[Try[Nothing]],

observer: OptimisticLockingCacheObserver,

timer: Timer,

) =

this(

underlyingCache: Cache[K, V],

retryPolicy: RetryPolicy[Try[Nothing]],

observer: OptimisticLockingCacheObserver,

timer: Timer,

false

)

def this(

underlyingCache: Cache[K, V],

backoffs: Backoff,

observer: OptimisticLockingCacheObserver,

timer: Timer

) =

this(

underlyingCache,

OptimisticLockingCache.retryPolicy(backoffs),

observer,

timer,

false

)

def this(

underlyingCache: Cache[K, V],

backoffs: Backoff,

observer: OptimisticLockingCacheObserver,

timer: Timer,

enableKeyLogging: Boolean

) =

this(

underlyingCache,

OptimisticLockingCache.retryPolicy(backoffs),

observer,

timer,

enableKeyLogging

)

private[this] val log = Logger.get("OptimisticLockingCache")

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

@deprecated("use RetryPolicy-based constructor", "0.1.2")

def this(underlyingCache: Cache[K, V], maxTries: Int = 10, enableKeyLogging: Boolean) = {

this(

underlyingCache,

Backoff.const(0.milliseconds).take(maxTries),

new OptimisticLockingCacheObserver(NullStatsReceiver),

new NullTimer,

enableKeyLogging

)

}

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

observer.time {

dispatch(key, handler, retryPolicy, Nil)

}

}

/\*\*

\* @param key

\* The key to look up in cache

\* @param handler

\* The handler that is applied to values from cache

\* @param retryPolicy

\* Used to determine if more attempts should be made.

\* @param attempts

\* Contains representations of the causes of previous dispatch failures

\*/

protected[this] def retry(

key: K,

failure: Try[Nothing],

handler: Handler[V],

retryPolicy: RetryPolicy[Try[Nothing]],

attempts: Seq[FailedAttempt]

): Future[Option[V]] =

retryPolicy(failure) match {

case None =>

observer.failure(attempts)

if (enableKeyLogging) {

rateLimitedLogger.log(

s"failed attempts for ${key}:\n ${attempts.mkString("\n ")}",

level = Level.INFO)

Future.exception(LockAndSetFailure("lockAndSet failed for " + key, attempts))

} else {

Future.exception(LockAndSetFailure("lockAndSet failed", attempts))

}

case Some((backoff, tailPolicy)) =>

timer

.doLater(backoff) {

dispatch(key, handler, tailPolicy, attempts)

}

.flatten

}

/\*\*

\* @param key

\* The key to look up in cache

\* @param handler

\* The handler that is applied to values from cache

\* @param retryPolicy

\* Used to determine if more attempts should be made.

\* @param attempts

\* Contains representations of the causes of previous dispatch failures

\*/

protected[this] def dispatch(

key: K,

handler: Handler[V],

retryPolicy: RetryPolicy[Try[Nothing]],

attempts: Seq[FailedAttempt]

): Future[Option[V]] = {

// get the value if nothing's there

handler(None) match {

case None =>

// if nothing should be done when missing, go straight to getAndConditionallySet,

// since there's nothing to attempt an add with

getAndConditionallySet(key, handler, retryPolicy, attempts)

case some @ Some(value) =>

// otherwise, try to do an atomic add, which will return false if something's there

underlyingCache.add(key, value) transform {

case Return(added) =>

if (added) {

// if added, return the value

observer.success(attempts)

Future.value(some)

} else {

// otherwise, do a checkAndSet based on the current value

getAndConditionallySet(key, handler, retryPolicy, attempts)

}

case Throw(t) =>

// count exception against retries

if (enableKeyLogging)

rateLimitedLogger.logThrowable(t, s"add($key) returned exception. will retry")

retry(key, Throw(t), handler, retryPolicy, attempts :+ AddException(t))

}

}

}

/\*\*

\* @param key

\* The key to look up in cache

\* @param handler

\* The handler that is applied to values from cache

\* @param retryPolicy

\* Used to determine if more attempts should be made.

\* @param attempts

\* Contains representations of the causes of previous dispatch failures

\*/

protected[this] def getAndConditionallySet(

key: K,

handler: Handler[V],

retryPolicy: RetryPolicy[Try[Nothing]],

attempts: Seq[FailedAttempt]

): Future[Option[V]] = {

// look in the cache to see what's there

underlyingCache.getWithChecksum(Seq(key)) handle {

case t =>

// treat global failure as key-based failure

KeyValueResult(failed = Map(key -> t))

} flatMap { lr =>

lr(key) match {

case Return.None =>

handler(None) match {

case Some(\_) =>

// if there's nothing in the cache now, but handler(None) return Some,

// that means something has changed since we attempted the add, so try again

val failure = GetWithChecksumEmpty

retry(key, Throw(failure), handler, retryPolicy, attempts :+ failure)

case None =>

// if there's nothing in the cache now, but handler(None) returns None,

// that means we don't want to store anything when there's nothing already

// in cache, so return None

observer.success(attempts)

emptyFutureNone

}

case Return(Some((Return(current), checksum))) =>

// the cache entry is present

dispatchCheckAndSet(Some(current), checksum, key, handler, retryPolicy, attempts)

case Return(Some((Throw(t), checksum))) =>

// the cache entry failed to deserialize; treat it as a None and overwrite.

if (enableKeyLogging)

rateLimitedLogger.logThrowable(

t,

s"getWithChecksum(${key}) returned a bad value. overwriting.")

dispatchCheckAndSet(None, checksum, key, handler, retryPolicy, attempts)

case Throw(t) =>

// lookup failure counts against numTries

if (enableKeyLogging)

rateLimitedLogger.logThrowable(

t,

s"getWithChecksum(${key}) returned exception. will retry.")

retry(key, Throw(t), handler, retryPolicy, attempts :+ GetWithChecksumException(t))

}

}

}

/\*\*

\* @param current

\* The value currently cached under key `key`, if any

\* @param checksum

\* The checksum of the currently-cached value

\* @param key

\* The key mapping to `current`

\* @param handler

\* The handler that is applied to values from cache

\* @param retryPolicy

\* Used to determine if more attempts should be made.

\* @param attempts

\* Contains representations of the causes of previous dispatch failures

\*/

protected[this] def dispatchCheckAndSet(

current: Option[V],

checksum: Checksum,

key: K,

handler: Handler[V],

retryPolicy: RetryPolicy[Try[Nothing]],

attempts: Seq[FailedAttempt]

): Future[Option[V]] = {

handler(current) match {

case None =>

// if nothing should be done based on the current value, don't do anything

observer.success(attempts)

emptyFutureNone

case some @ Some(value) =>

// otherwise, try a check and set with the checksum

underlyingCache.checkAndSet(key, value, checksum) transform {

case Return(added) =>

if (added) {

// if added, return the value

observer.success(attempts)

Future.value(some)

} else {

// otherwise, something has changed, try again

val failure = CheckAndSetFailed

retry(key, Throw(failure), handler, retryPolicy, attempts :+ failure)

}

case Throw(t) =>

// count exception against retries

if (enableKeyLogging)

rateLimitedLogger.logThrowable(

t,

s"checkAndSet(${key}) returned exception. will retry.")

retry(key, Throw(t), handler, retryPolicy, attempts :+ CheckAndSetException(t))

}

}

}

}

object NonLockingCacheFactory extends LockingCacheFactory {

override def apply[K, V](cache: Cache[K, V]): LockingCache[K, V] = new NonLockingCache(cache)

override def scope(scopes: String\*) = this

}

class NonLockingCache[K, V](override val underlyingCache: Cache[K, V])

extends LockingCache[K, V]

with CacheWrapper[K, V] {

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

handler(None) match {

case None =>

// if nothing should be done when nothing's there, don't do anything

Future.value(None)

case some @ Some(value) =>

set(key, value) map { \_ =>

some

}

}

}

}