package com.twitter.servo.repository

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.servo.cache.{CacheObserver, Cached, LockingCache}

import com.twitter.servo.repository

import com.twitter.servo.repository.CachedResult.{Handler, HandlerFactory}

import com.twitter.servo.util.\_

import com.twitter.util.\_

import scala.util.control.NoStackTrace

object DarkmodingKeyValueRepositoryFactory {

val DefaultEwmaHalfLife = 5.minutes

val DefaultRecentWindow = 10.seconds

val DefaultWindowSize = 5000

val DefaultAvailabilityFromSuccessRate =

Availability.linearlyScaled(highWaterMark = 0.98, lowWaterMark = 0.75, minAvailability = 0.02)

def DefaultEwmaTracker = new EwmaSuccessRateTracker(DefaultEwmaHalfLife)

def DefaultRecentWindowTracker = SuccessRateTracker.recentWindowed(DefaultRecentWindow)

def DefaultRollingWindowTracker = SuccessRateTracker.rollingWindow(DefaultWindowSize)

/\*\*

\* Wraps an underlying repository, which can be manually or automatically darkmoded.

\*

\* Auto-darkmoding is based on success rate (SR) as reported by a [[SuccessRateTracker]].

\*

\* @param readFromUnderlying Open: operate normally. Closed: read from backupRepo regardless of SR.

\* @param autoDarkmode Open: auto-darkmoding kicks in based on SR. Closed: auto-darkmoding will not kick in regardless of SR.

\* @param stats Used to record success rate and availability; often should be scoped to this repo for stats naming

\* @param underlyingRepo The underlying repo; read from when not darkmoded

\* @param backupRepo The repo to read from when darkmoded; defaults to an always-failing repo.

\* @param successRateTracker Strategy for reporting SR, usually over a moving window

\* @param availabilityFromSuccessRate Function to calculate availability based on success rate

\* @param shouldIgnore don't count certain exceptions as failures, e.g. cancellations

\*/

def darkmoding[Q <: Seq[K], K, V](

readFromUnderlying: Gate[Unit],

autoDarkmode: Gate[Unit],

stats: StatsReceiver,

underlyingRepo: KeyValueRepository[Q, K, V],

backupRepo: KeyValueRepository[Q, K, V] =

KeyValueRepository.alwaysFailing[Q, K, V](DarkmodedException),

successRateTracker: SuccessRateTracker = DefaultRecentWindowTracker,

availabilityFromSuccessRate: Double => Double = DefaultAvailabilityFromSuccessRate,

shouldIgnore: Throwable => Boolean = SuccessRateTrackingRepository.isCancellation

): KeyValueRepository[Q, K, V] = {

val (successRateTrackingRepoFactory, successRateGate) =

SuccessRateTrackingRepository.withGate[Q, K, V](

stats,

availabilityFromSuccessRate,

successRateTracker.observed(stats),

shouldIgnore

)

val gate = mkGate(successRateGate, readFromUnderlying, autoDarkmode)

Repository.selected(

q => gate(()),

successRateTrackingRepoFactory(underlyingRepo),

backupRepo

)

}

/\*\*

\* Produces a caching repository around an underlying repository, which

\* can be manually or automatically darkmoded.

\*

\* @param underlyingRepo The underlying repo from which to read

\* @param cache The typed locking cache to fall back to when darkmoded

\* @param picker Used to break ties when a value being written is already present in cache

\* @param readFromUnderlying Open: operate normally. Closed: read from cache regardless of SR.

\* @param autoDarkmode Open: auto-darkmoding kicks in based on SR. Closed: auto-darkmoding will not kick in regardless of SR.

\* @param cacheObserver Observes interactions with the cache; often should be scoped to this repo for stats naming

\* @param stats Used to record various stats; often should be scoped to this repo for stats naming

\* @param handler a [[Handler]] to use when not darkmoded

\* @param successRateTracker Strategy for reporting SR, usually over a moving window

\* @param availabilityFromSuccessRate Function to calculate availability based on success rate

\* @param shouldIgnore don't count certain exceptions as failures, e.g. cancellations

\*/

def darkmodingCaching[K, V, CacheKey](

underlyingRepo: KeyValueRepository[Seq[K], K, V],

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

picker: LockingCache.Picker[Cached[V]],

readFromUnderlying: Gate[Unit],

autoDarkmode: Gate[Unit],

cacheObserver: CacheObserver,

stats: StatsReceiver,

handler: Handler[K, V],

successRateTracker: SuccessRateTracker = DefaultRecentWindowTracker,

availabilityFromSuccessRate: Double => Double = DefaultAvailabilityFromSuccessRate,

shouldIgnore: Throwable => Boolean = SuccessRateTrackingRepository.isCancellation,

writeSoftTtlStep: Gate[Unit] = Gate.False,

cacheResultObserver: CachingKeyValueRepository.CacheResultObserver[K, V] =

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

): CachingKeyValueRepository[Seq[K], K, V] = {

val (successRateTrackingRepoFactory, successRateGate) =

SuccessRateTrackingRepository.withGate[Seq[K], K, V](

stats,

availabilityFromSuccessRate,

successRateTracker.observed(stats),

shouldIgnore

)

val gate = mkGate(successRateGate, readFromUnderlying, autoDarkmode)

new CachingKeyValueRepository[Seq[K], K, V](

successRateTrackingRepoFactory(underlyingRepo),

cache,

repository.keysAsQuery,

mkHandlerFactory(handler, gate),

picker,

cacheObserver,

writeSoftTtlStep = writeSoftTtlStep,

cacheResultObserver = cacheResultObserver

)

}

/\*\*

\* Create a composite gate suitable for controlling darkmoding, usually via decider

\*

\* @param successRate gate that should close and open according to success rate (SR) changes

\* @param readFromUnderlying if open: returned gate operates normally. if closed: returned gate will be closed regardless of SR

\* @param autoDarkMode if open: close gate according to SR. if closed: gate ignores SR changes

\* @return

\*/

def mkGate(

successRate: Gate[Unit],

readFromUnderlying: Gate[Unit],

autoDarkMode: Gate[Unit]

): Gate[Unit] =

readFromUnderlying & (successRate | !autoDarkMode)

/\*\*

\* Construct a [[CachedResult.HandlerFactory]] with sane defaults for use with a caching darkmoded repository

\* @param softTtl TTL for soft-expiration of values in the cache

\* @param expiry Used to apply the softTTL (e.g. fixed vs randomly perturbed)

\*/

def mkDefaultHandler[K, V](

softTtl: Option[V] => Duration,

expiry: CachedResult.Expiry

): Handler[K, V] =

CachedResult.Handler(

CachedResult.failuresAreDoNotCache,

CachedResult.Handler(CachedResult.softTtlExpiration(softTtl, expiry))

)

private[repository] def mkHandlerFactory[CacheKey, V, K](

handler: Handler[K, V],

successRateGate: Gate[Unit]

): HandlerFactory[Seq[K], K, V] =

query =>

if (successRateGate(())) handler

else CachedResult.cacheOnly

}

/\*\*

\* This exception is returned from a repository when it is auto-darkmoded due to low backend

\* success rate, or darkmoded manually via gate (usually a decider).

\*/

class DarkmodedException extends Exception with NoStackTrace

object DarkmodedException extends DarkmodedException