package com.twitter.tweetypie

package repository

import com.fasterxml.jackson.databind.ObjectMapper

import com.fasterxml.jackson.module.scala.DefaultScalaModule

import com.twitter.finagle.tracing.Trace

import com.twitter.servo.cache.\_

import com.twitter.servo.repository.\_

import com.twitter.servo.util.Transformer

import com.twitter.snowflake.id.SnowflakeId

import com.twitter.stitch.Stitch

import com.twitter.tweetypie.client\_id.ClientIdHelper

import com.twitter.tweetypie.core.FilteredState.Unavailable.BounceDeleted

import com.twitter.tweetypie.core.FilteredState.Unavailable.TweetDeleted

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.repository.CachedBounceDeleted.isBounceDeleted

import com.twitter.tweetypie.repository.CachedBounceDeleted.toBounceDeletedTweetResult

import com.twitter.tweetypie.thriftscala.CachedTweet

import com.twitter.util.Base64Long

case class TweetKey(cacheVersion: Int, id: TweetId)

extends ScopedCacheKey("t", "t", cacheVersion, Base64Long.toBase64(id))

case class TweetKeyFactory(cacheVersion: Int) {

val fromId: TweetId => TweetKey = (id: TweetId) => TweetKey(cacheVersion, id)

val fromTweet: Tweet => TweetKey = (tweet: Tweet) => fromId(tweet.id)

val fromCachedTweet: CachedTweet => TweetKey = (ms: CachedTweet) => fromTweet(ms.tweet)

}

// Helper methods for working with cached bounce-deleted tweets,

// grouped together here to keep the definitions of "bounce

// deleted" in one place.

object CachedBounceDeleted {

// CachedTweet for use in CachingTweetStore

def toBounceDeletedCachedTweet(tweetId: TweetId): CachedTweet =

CachedTweet(

tweet = Tweet(id = tweetId),

isBounceDeleted = Some(true)

)

def isBounceDeleted(cached: Cached[CachedTweet]): Boolean =

cached.status == CachedValueStatus.Found &&

cached.value.flatMap(\_.isBounceDeleted).contains(true)

// TweetResult for use in CachingTweetRepository

def toBounceDeletedTweetResult(tweetId: TweetId): TweetResult =

TweetResult(

TweetData(

tweet = Tweet(id = tweetId),

isBounceDeleted = true

)

)

def isBounceDeleted(tweetResult: TweetResult): Boolean =

tweetResult.value.isBounceDeleted

}

object TweetResultCache {

def apply(

tweetDataCache: Cache[TweetId, Cached[TweetData]]

): Cache[TweetId, Cached[TweetResult]] = {

val transformer: Transformer[Cached[TweetResult], Cached[TweetData]] =

new Transformer[Cached[TweetResult], Cached[TweetData]] {

def to(cached: Cached[TweetResult]) =

Return(cached.map(\_.value))

def from(cached: Cached[TweetData]) =

Return(cached.map(TweetResult(\_)))

}

new KeyValueTransformingCache(

tweetDataCache,

transformer,

identity

)

}

}

object TweetDataCache {

def apply(

cachedTweetCache: Cache[TweetKey, Cached[CachedTweet]],

tweetKeyFactory: TweetId => TweetKey

): Cache[TweetId, Cached[TweetData]] = {

val transformer: Transformer[Cached[TweetData], Cached[CachedTweet]] =

new Transformer[Cached[TweetData], Cached[CachedTweet]] {

def to(cached: Cached[TweetData]) =

Return(cached.map(\_.toCachedTweet))

def from(cached: Cached[CachedTweet]) =

Return(cached.map(c => TweetData.fromCachedTweet(c, cached.cachedAt)))

}

new KeyValueTransformingCache(

cachedTweetCache,

transformer,

tweetKeyFactory

)

}

}

object TombstoneTtl {

import CachedResult.\_

def fixed(ttl: Duration): CachedNotFound[TweetId] => Duration =

\_ => ttl

/\*\*

\* A simple ttl calculator that is set to `min` if the age is less than `from`,

\* then linearly interpolated between `min` and `max` when the age is between `from` and `to`,

\* and then equal to `max` if the age is greater than `to`.

\*/

def linear(

min: Duration,

max: Duration,

from: Duration,

to: Duration

): CachedNotFound[TweetId] => Duration = {

val rate = (max - min).inMilliseconds / (to - from).inMilliseconds.toDouble

cached => {

if (SnowflakeId.isSnowflakeId(cached.key)) {

val age = cached.cachedAt - SnowflakeId(cached.key).time

if (age <= from) min

else if (age >= to) max

else min + (age - from) \* rate

} else {

// When it's not a snowflake id, cache it for the maximum time.

max

}

}

}

/\*\*

\* Checks if the given `cached` value is an expired tombstone

\*/

def isExpired(

tombstoneTtl: CachedNotFound[TweetId] => Duration,

cached: CachedNotFound[TweetId]

): Boolean =

Time.now - cached.cachedAt > tombstoneTtl(cached)

}

object CachingTweetRepository {

import CachedResult.\_

import CachedResultAction.\_

val failuresLog: Logger = Logger("com.twitter.tweetypie.repository.CachingTweetRepoFailures")

def apply(

cache: LockingCache[TweetId, Cached[TweetResult]],

tombstoneTtl: CachedNotFound[TweetId] => Duration,

stats: StatsReceiver,

clientIdHelper: ClientIdHelper,

logCacheExceptions: Gate[Unit] = Gate.False,

)(

underlying: TweetResultRepository.Type

): TweetResultRepository.Type = {

val cachingRepo: ((TweetId, TweetQuery.Options)) => Stitch[TweetResult] =

CacheStitch[(TweetId, TweetQuery.Options), TweetId, TweetResult](

repo = underlying.tupled,

cache = StitchLockingCache(

underlying = cache,

picker = new TweetRepoCachePicker[TweetResult](\_.value.cachedAt)

),

queryToKey = \_.\_1, // extract tweet id from (TweetId, TweetQuery.Options)

handler = mkHandler(tombstoneTtl, stats, logCacheExceptions, clientIdHelper),

cacheable = cacheable

)

(tweetId, options) =>

if (options.cacheControl.readFromCache) {

cachingRepo((tweetId, options))

} else {

underlying(tweetId, options)

}

}

val cacheable: CacheStitch.Cacheable[(TweetId, TweetQuery.Options), TweetResult] = {

case ((tweetId, options), tweetResult) =>

if (!options.cacheControl.writeToCache) {

None

} else {

tweetResult match {

// Write stitch.NotFound as a NotFound cache entry

case Throw(com.twitter.stitch.NotFound) =>

Some(StitchLockingCache.Val.NotFound)

// Write FilteredState.TweetDeleted as a Deleted cache entry

case Throw(TweetDeleted) =>

Some(StitchLockingCache.Val.Deleted)

// Write BounceDeleted as a Found cache entry, with the CachedTweet.isBounceDeleted flag.

// servo.cache.thriftscala.CachedValueStatus.Deleted tombstones do not allow for storing

// app-defined metadata.

case Throw(BounceDeleted) =>

Some(StitchLockingCache.Val.Found(toBounceDeletedTweetResult(tweetId)))

// Regular found tweets are not written to cache here - instead the cacheable result is

// written to cache via TweetHydration.cacheChanges

case Return(\_: TweetResult) => None

// Don't write other exceptions back to cache

case \_ => None

}

}

}

object LogLens {

private[this] val mapper = new ObjectMapper().registerModule(DefaultScalaModule)

def logMessage(logger: Logger, clientIdHelper: ClientIdHelper, data: (String, Any)\*): Unit = {

val allData = data ++ defaultData(clientIdHelper)

val msg = mapper.writeValueAsString(Map(allData: \_\*))

logger.info(msg)

}

private def defaultData(clientIdHelper: ClientIdHelper): Seq[(String, Any)] = {

val viewer = TwitterContext()

Seq(

"client\_id" -> clientIdHelper.effectiveClientId,

"trace\_id" -> Trace.id.traceId.toString,

"audit\_ip" -> viewer.flatMap(\_.auditIp),

"application\_id" -> viewer.flatMap(\_.clientApplicationId),

"user\_agent" -> viewer.flatMap(\_.userAgent),

"authenticated\_user\_id" -> viewer.flatMap(\_.authenticatedUserId)

)

}

}

def mkHandler(

tombstoneTtl: CachedNotFound[TweetId] => Duration,

stats: StatsReceiver,

logCacheExceptions: Gate[Unit],

clientIdHelper: ClientIdHelper,

): Handler[TweetId, TweetResult] = {

val baseHandler = defaultHandler[TweetId, TweetResult]

val cacheErrorState = HydrationState(modified = false, cacheErrorEncountered = true)

val cachedFoundCounter = stats.counter("cached\_found")

val notFoundCounter = stats.counter("not\_found")

val cachedNotFoundAsNotFoundCounter = stats.counter("cached\_not\_found\_as\_not\_found")

val cachedNotFoundAsMissCounter = stats.counter("cached\_not\_found\_as\_miss")

val cachedDeletedCounter = stats.counter("cached\_deleted")

val cachedBounceDeletedCounter = stats.counter("cached\_bounce\_deleted")

val failedCounter = stats.counter("failed")

val otherCounter = stats.counter("other")

{

case res @ CachedFound(\_, tweetResult, \_, \_) =>

if (isBounceDeleted(tweetResult)) {

cachedBounceDeletedCounter.incr()

HandleAsFailed(FilteredState.Unavailable.BounceDeleted)

} else {

cachedFoundCounter.incr()

baseHandler(res)

}

case res @ NotFound(\_) =>

notFoundCounter.incr()

baseHandler(res)

// expires NotFound tombstones if old enough

case cached @ CachedNotFound(\_, \_, \_) =>

if (TombstoneTtl.isExpired(tombstoneTtl, cached)) {

cachedNotFoundAsMissCounter.incr()

HandleAsMiss

} else {

cachedNotFoundAsNotFoundCounter.incr()

HandleAsNotFound

}

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

cachedDeletedCounter.incr()

HandleAsFailed(FilteredState.Unavailable.TweetDeleted)

// don't attempt to write back to cache on a cache read failure

case Failed(k, t) =>

// After result is found, mark it with cacheErrorEncountered

failedCounter.incr()

if (logCacheExceptions()) {

LogLens.logMessage(

failuresLog,

clientIdHelper,

"type" -> "cache\_failed",

"tweet\_id" -> k,

"throwable" -> t.getClass.getName

)

}

TransformSubAction[TweetResult](HandleAsDoNotCache, \_.mapState(\_ ++ cacheErrorState))

case res =>

otherCounter.incr()

baseHandler(res)

}

}

}

/\*\*

\* A LockingCache.Picker for use with CachingTweetRepository which prevents overwriting values in

\* cache that are newer than the value previously read from cache.

\*/

class TweetRepoCachePicker[T](cachedAt: T => Option[Time]) extends LockingCache.Picker[Cached[T]] {

private val newestPicker = new PreferNewestCached[T]

override def apply(newValue: Cached[T], oldValue: Cached[T]): Option[Cached[T]] = {

oldValue.status match {

// never overwrite a `Deleted` tombstone via read-through.

case CachedValueStatus.Deleted => None

// only overwrite a `Found` value with an update based off of that same cache entry.

case CachedValueStatus.Found =>

newValue.value.flatMap(cachedAt) match {

// if prevCacheAt is the same as oldValue.cachedAt, then the value in cache hasn't changed

case Some(prevCachedAt) if prevCachedAt == oldValue.cachedAt => Some(newValue)

// otherwise, the value in cache has changed since we read it, so don't overwrite

case \_ => None

}

// we may hit an expired/older tombstone, which should be safe to overwrite with a fresh

// tombstone of a new value returned from Manhattan.

case CachedValueStatus.NotFound => newestPicker(newValue, oldValue)

// we shouldn't see any other CachedValueStatus, but if we do, play it safe and don't

// overwrite (it will be as if the read that triggered this never happened)

case \_ => None

}

}

}