package com.twitter.tweetypie

package handler

import com.twitter.servo.cache.Cache

import com.twitter.servo.util.Scribe

import com.twitter.tweetypie.serverutil.ExceptionCounter

import com.twitter.tweetypie.thriftscala.PostTweetResult

import com.twitter.tweetypie.util.TweetCreationLock.Key

import com.twitter.tweetypie.util.TweetCreationLock.State

import com.twitter.util.Base64Long

import scala.util.Random

import scala.util.control.NoStackTrace

import scala.util.control.NonFatal

/\*\*

\* This exception is returned from TweetCreationLock if there is an

\* in-progress cache entry for this key. It is possible that the key

\* exists because the key was not properly cleaned up, but it's

\* impossible to differentiate between these cases. We resolve this by

\* returning TweetCreationInProgress and having a (relatively) short TTL

\* on the cache entry so that the client and/or user may retry.

\*/

case object TweetCreationInProgress extends Exception with NoStackTrace

/\*\*

\* Thrown when the TweetCreationLock discovers that there is already

\* a tweet with the specified uniqueness id.

\*/

case class DuplicateTweetCreation(tweetId: TweetId) extends Exception with NoStackTrace

trait TweetCreationLock {

def apply(

key: Key,

dark: Boolean,

nullcast: Boolean

)(

insert: => Future[PostTweetResult]

): Future[PostTweetResult]

def unlock(key: Key): Future[Unit]

}

object CacheBasedTweetCreationLock {

/\*\*

\* Indicates that setting the lock value failed because the state of

\* that key in the cache has been changed (by another process or

\* cache eviction).

\*/

case object UnexpectedCacheState extends Exception with NoStackTrace

/\*\*

\* Thrown when the process of updating the lock cache failed more

\* than the allowed number of times.

\*/

case class RetriesExhausted(failures: Seq[Exception]) extends Exception with NoStackTrace

def shouldRetry(e: Exception): Boolean =

e match {

case TweetCreationInProgress => false

case \_: DuplicateTweetCreation => false

case \_: RetriesExhausted => false

case \_ => true

}

def ttlChooser(shortTtl: Duration, longTtl: Duration): (Key, State) => Duration =

(\_, state) =>

state match {

case \_: State.AlreadyCreated => longTtl

case \_ => shortTtl

}

/\*\*

\* The log format is tab-separated (base 64 tweet\_id, base 64

\* uniqueness\_id). It's logged this way in order to minimize the

\* storage requirement and to make it easy to analyze. Each log line

\* should be 24 bytes, including newline.

\*/

val formatUniquenessLogEntry: ((String, TweetId)) => String = {

case (uniquenessId, tweetId) => Base64Long.toBase64(tweetId) + "\t" + uniquenessId

}

/\*\*

\* Scribe the uniqueness id paired with the tweet id so that we can

\* track the rate of failures of the uniqueness id check by

\* detecting multiple tweets created with the same uniqueness id.

\*

\* Scribe to a test category because we only need to keep this

\* information around for long enough to find any duplicates.

\*/

val ScribeUniquenessId: FutureEffect[(String, TweetId)] =

Scribe("test\_tweetypie\_uniqueness\_id") contramap formatUniquenessLogEntry

private[this] val UniquenessIdLog = Logger("com.twitter.tweetypie.handler.UniquenessId")

/\*\*

\* Log the uniqueness ids to a standard logger (for use when it's

\* not production traffic).

\*/

val LogUniquenessId: FutureEffect[(String, TweetId)] = FutureEffect[(String, TweetId)] { rec =>

UniquenessIdLog.info(formatUniquenessLogEntry(rec))

Future.Unit

}

private val log = Logger(getClass)

}

/\*\*

\* This class adds locking around Tweet creation, to prevent creating

\* duplicate tweets when two identical requests arrive simultaneously.

\* A lock is created in cache using the user id and a hash of the tweet text

\* in the case of tweets, or the source\_status\_id in the case of retweets.

\* If another process attempts to lock for the same user and hash, the request

\* fails as a duplicate. The lock lasts for 10 seconds if it is not deleted.

\* Given the hard timeout of 5 seconds on all requests, it should never take

\* us longer than 5 seconds to create a request, but we've observed times of up

\* to 10 seconds to create statuses for some of our more popular users.

\*

\* When a request with a uniqueness id is successful, the id of the

\* created tweet will be stored in the cache so that subsequent

\* requests can retrieve the originally-created tweet rather than

\* duplicating creation or getting an exception.

\*/

class CacheBasedTweetCreationLock(

cache: Cache[Key, State],

maxTries: Int,

stats: StatsReceiver,

logUniquenessId: FutureEffect[(String, TweetId)])

extends TweetCreationLock {

import CacheBasedTweetCreationLock.\_

private[this] val eventCounters = stats.scope("event")

private[this] def event(k: Key, name: String): Unit = {

log.debug(s"$name:$k")

eventCounters.counter(name).incr()

}

private[this] def retryLoop[A](action: => Future[A]): Future[A] = {

def go(failures: List[Exception]): Future[A] =

if (failures.length >= maxTries) {

Future.exception(RetriesExhausted(failures.reverse))

} else {

action.rescue {

case e: Exception if shouldRetry(e) => go(e :: failures)

}

}

go(Nil)

}

private[this] val lockerExceptions = ExceptionCounter(stats)

/\*\*

\* Obtain the lock for creating a tweet. If this method completes

\* without throwing an exception, then the lock value was

\* successfully set in cache, which indicates a high probability

\* that this is the only process that is attempting to create this

\* tweet. (The uncertainty comes from the possibility of lock

\* entries missing from the cache.)

\*

\* @throws TweetCreationInProgress if there is another process

\* trying to create this tweet.

\*

\* @throws DuplicateTweetCreation if a tweet has already been

\* created for a duplicate request. The exception has the id of

\* the created tweet.

\*

\* @throws RetriesExhausted if obtaining the lock failed more than

\* the requisite number of times.

\*/

private[this] def obtainLock(k: Key, token: Long): Future[Time] = retryLoop {

val lockTime = Time.now

// Get the current state for this key.

cache

.getWithChecksum(Seq(k))

.flatMap(initialStateKvr => Future.const(initialStateKvr(k)))

.flatMap {

case None =>

// Nothing in cache for this key

cache

.add(k, State.InProgress(token, lockTime))

.flatMap {

case true => Future.value(lockTime)

case false => Future.exception(UnexpectedCacheState)

}

case Some((Throw(e), \_)) =>

Future.exception(e)

case Some((Return(st), cs)) =>

st match {

case State.Unlocked =>

// There is an Unlocked entry for this key, which

// implies that a previous attempt was cleaned up.

cache

.checkAndSet(k, State.InProgress(token, lockTime), cs)

.flatMap {

case true => Future.value(lockTime)

case false => Future.exception(UnexpectedCacheState)

}

case State.InProgress(cachedToken, creationStartedTimestamp) =>

if (cachedToken == token) {

// There is an in-progress entry for \*this process\*. This

// can happen on a retry if the `add` actually succeeds

// but the future fails. The retry can return the result

// of the add that we previously tried.

Future.value(creationStartedTimestamp)

} else {

// There is an in-progress entry for \*a different

// process\*. This implies that there is another tweet

// creation in progress for \*this tweet\*.

val tweetCreationAge = Time.now - creationStartedTimestamp

k.uniquenessId.foreach { id =>

log.info(

"Found an in-progress tweet creation for uniqueness id %s %s ago"

.format(id, tweetCreationAge)

)

}

stats.stat("in\_progress\_age\_ms").add(tweetCreationAge.inMilliseconds)

Future.exception(TweetCreationInProgress)

}

case State.AlreadyCreated(tweetId, creationStartedTimestamp) =>

// Another process successfully created a tweet for this

// key.

val tweetCreationAge = Time.now - creationStartedTimestamp

stats.stat("already\_created\_age\_ms").add(tweetCreationAge.inMilliseconds)

Future.exception(DuplicateTweetCreation(tweetId))

}

}

}

/\*\*

\* Attempt to remove this process' lock entry from the cache. This

\* is done by writing a short-lived tombstone, so that we can ensure

\* that we only overwrite the entry if it is still an entry for this

\* process instead of another process' entry.

\*/

private[this] def cleanupLoop(k: Key, token: Long): Future[Unit] =

retryLoop {

// Instead of deleting the value, we attempt to write Unlocked,

// because we only want to delete it if it was the value that we

// wrote ourselves, and there is no delete call that is

// conditional on the extant value.

cache

.getWithChecksum(Seq(k))

.flatMap(kvr => Future.const(kvr(k)))

.flatMap {

case None =>

// Nothing in the cache for this tweet creation, so cleanup

// is successful.

Future.Unit

case Some((tryV, cs)) =>

// If we failed trying to deserialize the value, then we

// want to let the error bubble up, because there is no good

// recovery procedure, since we can't tell whether the entry

// is ours.

Future.const(tryV).flatMap {

case State.InProgress(presentToken, \_) =>

if (presentToken == token) {

// This is \*our\* in-progress marker, so we want to

// overwrite it with the tombstone. If checkAndSet

// returns false, that's OK, because that means

// someone else overwrote the value, and we don't have

// to clean it up anymore.

cache.checkAndSet(k, State.Unlocked, cs).unit

} else {

// Indicates that another request has overwritten our

// state before we cleaned it up. This should only

// happen when our token was cleared from cache and

// another process started a duplicate create. This

// should be very infrequent. We count it just to be

// sure.

event(k, "other\_attempt\_in\_progress")

Future.Unit

}

case \_ =>

// Cleanup has succeeded, because we are not responsible

// for the cache entry anymore.

Future.Unit

}

}

}.onSuccess { \_ => event(k, "cleanup\_attempt\_succeeded") }

.handle {

case \_ => event(k, "cleanup\_attempt\_failed")

}

/\*\*

\* Mark that a tweet has been successfully created. Subsequent calls

\* to `apply` with this key will receive a DuplicateTweetCreation

\* exception with the specified id.

\*/

private[this] def creationComplete(k: Key, tweetId: TweetId, lockTime: Time): Future[Unit] =

// Unconditionally set the state because regardless of the

// value present, we know that we want to transition to the

// AlreadyCreated state for this key.

retryLoop(cache.set(k, State.AlreadyCreated(tweetId, lockTime)))

.onSuccess(\_ => event(k, "mark\_created\_succeeded"))

.onFailure { case \_ => event(k, "mark\_created\_failed") }

// If this fails, it's OK for the request to complete

// successfully, because it's more harmful to create the tweet

// and return failure than it is to complete it successfully,

// but fail to honor the uniqueness id next time.

.handle { case NonFatal(\_) => }

private[this] def createWithLock(

k: Key,

create: => Future[PostTweetResult]

): Future[PostTweetResult] = {

val token = Random.nextLong

event(k, "lock\_attempted")

obtainLock(k, token)

.onSuccess { \_ => event(k, "lock\_obtained") }

.handle {

// If we run out of retries when trying to get the lock, then

// just go ahead with tweet creation. We should keep an eye on

// how frequently this happens, because this means that the

// only sign that this is happening will be duplicate tweet

// creations.

case RetriesExhausted(failures) =>

event(k, "lock\_failure\_ignored")

// Treat this as the time that we obtained the lock.

Time.now

}

.onFailure {

case e => lockerExceptions(e)

}

.flatMap { lockTime =>

create.transform {

case r @ Return(PostTweetResult(\_, Some(tweet), \_, \_, \_, \_, \_)) =>

event(k, "create\_succeeded")

k.uniquenessId.foreach { u => logUniquenessId((u, tweet.id)) }

// Update the lock entry to remember the id of the tweet we

// created and extend the TTL.

creationComplete(k, tweet.id, lockTime).before(Future.const(r))

case other =>

other match {

case Throw(e) =>

log.debug(s"Tweet creation failed for key $k", e)

case Return(r) =>

log.debug(s"Tweet creation failed for key $k, so unlocking: $r")

}

event(k, "create\_failed")

// Attempt to clean up the lock after the failed create.

cleanupLoop(k, token).before(Future.const(other))

}

}

}

/\*\*

\* Make a best-effort attempt at removing the duplicate cache entry

\* for this key. If this fails, it is not catastrophic. The worst-case

\* behavior should be that the user has to wait for the short TTL to

\* elapse before tweeting succeeds.

\*/

def unlock(k: Key): Future[Unit] =

retryLoop(cache.delete(k).unit).onSuccess(\_ => event(k, "deleted"))

/\*\*

\* Prevent duplicate tweet creation.

\*

\* Ensures that no more than one tweet creation for the same key is

\* happening at the same time. If `create` fails, then the key will

\* be removed from the cache. If it succeeds, then the key will be

\* retained.

\*

\* @throws DuplicateTweetCreation if a tweet has already been

\* created by a previous request. The exception has the id of the

\* created tweet.

\*

\* @throws TweetCreationInProgress. See the documentation above.

\*/

def apply(

k: Key,

isDark: Boolean,

nullcast: Boolean

)(

create: => Future[PostTweetResult]

): Future[PostTweetResult] =

if (isDark) {

event(k, "dark\_create")

create

} else if (nullcast) {

event(k, "nullcast\_create")

create

} else {

createWithLock(k, create).onFailure {

// Another process is creating this same tweet (or has already

// created it)

case TweetCreationInProgress =>

event(k, "tweet\_creation\_in\_progress")

case \_: DuplicateTweetCreation =>

event(k, "tweet\_already\_created")

case \_ =>

}

}

}