package com.twitter.tweetypie

package config

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.Backoff

import com.twitter.finagle.memcached

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.servo.cache.{Serializer => CacheSerializer, \_}

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

import com.twitter.tweetypie.core.\_

import com.twitter.tweetypie.handler.CacheBasedTweetCreationLock

import com.twitter.tweetypie.repository.\_

import com.twitter.tweetypie.serverutil.\_

import com.twitter.tweetypie.thriftscala.\_

import com.twitter.tweetypie.util.\_

import com.twitter.util.Timer

/\*\*

\* Provides configured caches (most backed by memcached) wrapped with appropriate metrics and locks.

\*

\* All memcached-backed caches share:

\* - one Finagle memcached client from backends.memcacheClient

\* - one in memory caffeine cache

\* - one Twemcache pool

\*

\* Each memcached-backed cache specialization provides its own:

\* - key prefix or "namespace"

\* - value serializer/deserializer

\* - stats scope

\* - log name

\*/

trait Caches {

val memcachedClientWithInProcessCaching: memcached.Client

val tweetCache: LockingCache[TweetKey, Cached[CachedTweet]]

val tweetResultCache: LockingCache[TweetId, Cached[TweetResult]]

val tweetDataCache: LockingCache[TweetId, Cached[TweetData]]

val tweetCreateLockerCache: Cache[TweetCreationLock.Key, TweetCreationLock.State]

val tweetCountsCache: LockingCache[TweetCountKey, Cached[Count]]

val deviceSourceInProcessCache: LockingCache[String, Cached[DeviceSource]]

val geoScrubCache: LockingCache[UserId, Cached[Time]]

}

object Caches {

object NoCache extends Caches {

override val memcachedClientWithInProcessCaching: memcached.Client = new NullMemcacheClient()

private val toLockingCache: LockingCacheFactory = NonLockingCacheFactory

val tweetCache: LockingCache[TweetKey, Cached[CachedTweet]] =

toLockingCache(new NullCache)

val tweetResultCache: LockingCache[TweetId, Cached[TweetResult]] =

toLockingCache(new NullCache)

val tweetDataCache: LockingCache[TweetId, Cached[TweetData]] =

toLockingCache(new NullCache)

val tweetCreateLockerCache: Cache[TweetCreationLock.Key, TweetCreationLock.State] =

new NullCache

val tweetCountsCache: LockingCache[TweetCountKey, Cached[Count]] =

toLockingCache(new NullCache)

val deviceSourceInProcessCache: LockingCache[String, Cached[DeviceSource]] =

toLockingCache(new NullCache)

val geoScrubCache: LockingCache[UserId, Cached[Time]] =

toLockingCache(new NullCache)

}

def apply(

settings: TweetServiceSettings,

stats: StatsReceiver,

timer: Timer,

clients: BackendClients,

tweetKeyFactory: TweetKeyFactory,

deciderGates: TweetypieDeciderGates,

clientIdHelper: ClientIdHelper,

): Caches = {

val cachesStats = stats.scope("caches")

val cachesInprocessStats = cachesStats.scope("inprocess")

val cachesMemcacheStats = cachesStats.scope("memcache")

val cachesMemcacheObserver = new StatsReceiverCacheObserver(cachesStats, 10000, "memcache")

val cachesMemcacheTweetStats = cachesMemcacheStats.scope("tweet")

val cachesInprocessDeviceSourceStats = cachesInprocessStats.scope("device\_source")

val cachesMemcacheCountStats = cachesMemcacheStats.scope("count")

val cachesMemcacheTweetCreateStats = cachesMemcacheStats.scope("tweet\_create")

val cachesMemcacheGeoScrubStats = cachesMemcacheStats.scope("geo\_scrub")

val memcacheClient = clients.memcacheClient

val caffieneMemcachedClient = settings.inProcessCacheConfigOpt match {

case Some(inProcessCacheConfig) =>

new CaffeineMemcacheClient(

proxyClient = memcacheClient,

inProcessCacheConfig.maximumSize,

inProcessCacheConfig.ttl,

cachesMemcacheStats.scope("caffeine")

)

case None =>

memcacheClient

}

val observedMemcacheWithCaffeineClient =

new ObservableMemcache(

new FinagleMemcache(

caffieneMemcachedClient

),

cachesMemcacheObserver

)

def observeCache[K, V](

cache: Cache[K, V],

stats: StatsReceiver,

logName: String,

windowSize: Int = 10000

) =

ObservableCache(

cache,

stats,

windowSize,

// Need to use an old-school c.t.logging.Logger because that's what servo needs

com.twitter.logging.Logger(s"com.twitter.tweetypie.cache.$logName")

)

def mkCache[K, V](

ttl: Duration,

serializer: CacheSerializer[V],

perCacheStats: StatsReceiver,

logName: String,

windowSize: Int = 10000

): Cache[K, V] = {

observeCache(

new MemcacheCache[K, V](

observedMemcacheWithCaffeineClient,

ttl,

serializer

),

perCacheStats,

logName,

windowSize

)

}

def toLockingCache[K, V](

cache: Cache[K, V],

stats: StatsReceiver,

backoffs: Stream[Duration] = settings.lockingCacheBackoffs

): LockingCache[K, V] =

new OptimisticLockingCache(

underlyingCache = cache,

backoffs = Backoff.fromStream(backoffs),

observer = new OptimisticLockingCacheObserver(stats),

timer = timer

)

def mkLockingCache[K, V](

ttl: Duration,

serializer: CacheSerializer[V],

stats: StatsReceiver,

logName: String,

windowSize: Int = 10000,

backoffs: Stream[Duration] = settings.lockingCacheBackoffs

): LockingCache[K, V] =

toLockingCache(

mkCache(ttl, serializer, stats, logName, windowSize),

stats,

backoffs

)

def trackTimeInCache[K, V](

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

stats: StatsReceiver

): Cache[K, Cached[V]] =

new CacheWrapper[K, Cached[V]] {

val ageStat: Stat = stats.stat("time\_in\_cache\_ms")

val underlyingCache: Cache[K, Cached[V]] = cache

override def get(keys: Seq[K]): Future[KeyValueResult[K, Cached[V]]] =

underlyingCache.get(keys).onSuccess(record)

private def record(res: KeyValueResult[K, Cached[V]]): Unit = {

val now = Time.now

for (c <- res.found.values) {

ageStat.add(c.cachedAt.until(now).inMilliseconds)

}

}

}

new Caches {

override val memcachedClientWithInProcessCaching: memcached.Client = caffieneMemcachedClient

private val observingTweetCache: Cache[TweetKey, Cached[CachedTweet]] =

trackTimeInCache(

mkCache(

ttl = settings.tweetMemcacheTtl,

serializer = Serializer.CachedTweet.CachedCompact,

perCacheStats = cachesMemcacheTweetStats,

logName = "MemcacheTweetCache"

),

cachesMemcacheTweetStats

)

// Wrap the tweet cache with a wrapper that will scribe the cache writes

// that happen to a fraction of tweets. This was added as part of the

// investigation into missing place ids and cache inconsistencies that

// were discovered by the additional fields hydrator.

private[this] val writeLoggingTweetCache =

new ScribeTweetCacheWrites(

underlyingCache = observingTweetCache,

logYoungTweetCacheWrites = deciderGates.logYoungTweetCacheWrites,

logTweetCacheWrites = deciderGates.logTweetCacheWrites

)

val tweetCache: LockingCache[TweetKey, Cached[CachedTweet]] =

toLockingCache(

cache = writeLoggingTweetCache,

stats = cachesMemcacheTweetStats

)

val tweetDataCache: LockingCache[TweetId, Cached[TweetData]] =

toLockingCache(

cache = TweetDataCache(tweetCache, tweetKeyFactory.fromId),

stats = cachesMemcacheTweetStats

)

val tweetResultCache: LockingCache[TweetId, Cached[TweetResult]] =

toLockingCache(

cache = TweetResultCache(tweetDataCache),

stats = cachesMemcacheTweetStats

)

val tweetCountsCache: LockingCache[TweetCountKey, Cached[Count]] =

mkLockingCache(

ttl = settings.tweetCountsMemcacheTtl,

serializer = Serializers.CachedLong.Compact,

stats = cachesMemcacheCountStats,

logName = "MemcacheTweetCountCache",

windowSize = 1000,

backoffs = Backoff.linear(0.millis, 2.millis).take(2).toStream

)

val tweetCreateLockerCache: Cache[TweetCreationLock.Key, TweetCreationLock.State] =

observeCache(

new TtlCacheToCache(

underlyingCache = new KeyValueTransformingTtlCache(

underlyingCache = observedMemcacheWithCaffeineClient,

transformer = TweetCreationLock.State.Serializer,

underlyingKey = (\_: TweetCreationLock.Key).toString

),

ttl = CacheBasedTweetCreationLock.ttlChooser(

shortTtl = settings.tweetCreateLockingMemcacheTtl,

longTtl = settings.tweetCreateLockingMemcacheLongTtl

)

),

stats = cachesMemcacheTweetCreateStats,

logName = "MemcacheTweetCreateLockingCache",

windowSize = 1000

)

val deviceSourceInProcessCache: LockingCache[String, Cached[DeviceSource]] =

toLockingCache(

observeCache(

new ExpiringLruCache(

ttl = settings.deviceSourceInProcessTtl,

maximumSize = settings.deviceSourceInProcessCacheMaxSize

),

stats = cachesInprocessDeviceSourceStats,

logName = "InprocessDeviceSourceCache"

),

stats = cachesInprocessDeviceSourceStats

)

val geoScrubCache: LockingCache[UserId, Cached[Time]] =

toLockingCache[UserId, Cached[Time]](

new KeyTransformingCache(

mkCache[GeoScrubTimestampKey, Cached[Time]](

ttl = settings.geoScrubMemcacheTtl,

serializer = Serializer.toCached(CacheSerializer.Time),

perCacheStats = cachesMemcacheGeoScrubStats,

logName = "MemcacheGeoScrubCache"

),

(userId: UserId) => GeoScrubTimestampKey(userId)

),

cachesMemcacheGeoScrubStats

)

}

}

}