package com.twitter.servo.cache

import com.twitter.finagle.memcached.Client

import com.twitter.finagle.memcached.protocol.Value

import com.twitter.finagle.memcached.GetResult

import com.twitter.finagle.memcached.ProxyClient

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.tracing.Trace

import com.twitter.io.Buf

import com.twitter.logging.Logger

import com.twitter.util.Future

import scala.collection.breakOut

object HotKeyCachingCache {

private[cache] val logger = Logger.get(getClass)

}

/\*\*

\* Wrapper for a [[com.twitter.finagle.Memcached.Client]] that handles in-process caching for

\* values flagged for promotion ("hot keys") by a twemcache backend.

\*

\* This is similar conceptually to

\* [[com.twitter.servo.repository.HotKeyCachingKeyValueRepository]] but differs because

\* HotKeyCachingKeyValueRepository detects hot keys in the client, which requires tuning and

\* becomes less effective as the number of instances in the cluster grows. [[HotKeyMemcacheClient]]

\* uses detection in the memcache server, which is centralized and has a better view of frequently

\* accessed keys. This is a custom feature in twemcache, Twitter's memcache fork, that is not

\* enabled by default. Consult with the cache team if you want to use it.

\*

\* Usage:

\* {{{

\* new HotKeyMemcacheClient(

\* underlyingCache = Memcached.client. ... .newRichClient(destination),

\* inProcessCache = ExpiringLruInProcessCache(ttl = 10.seconds, maximumSize = 100),

\* statsReceiver = statsReceiver.scope("inprocess")

\* )

\* }}}

\*/

class HotKeyMemcacheClient(

override val proxyClient: Client,

inProcessCache: InProcessCache[String, Value],

statsReceiver: StatsReceiver,

label: Option[String] = None)

extends ProxyClient {

import HotKeyCachingCache.\_

private val promotions = statsReceiver.counter("promotions")

private val hits = statsReceiver.counter("hits")

private val misses = statsReceiver.counter("misses")

private def cacheIfPromoted(key: String, value: Value): Unit = {

if (value.flags.exists(MemcacheFlags.shouldPromote)) {

logger.debug(s"Promoting hot-key $key flagged by memcached backend to in-process cache.")

Trace.recordBinary("hot\_key\_cache.hot\_key\_promoted", s"${label.getOrElse("")},$key")

promotions.incr()

inProcessCache.set(key, value)

}

}

override def getResult(keys: Iterable[String]): Future[GetResult] = {

val resultsFromInProcessCache: Map[String, Value] =

keys.flatMap(k => inProcessCache.get(k).map(v => (k, v)))(breakOut)

val foundInProcess = resultsFromInProcessCache.keySet

val newKeys = keys.filterNot(foundInProcess.contains)

hits.incr(foundInProcess.size)

misses.incr(newKeys.size)

if (foundInProcess.nonEmpty) {

// If there are hot keys found in the cache, record a trace annotation with the format:

// hot key cache client label;the number of hits;number of misses;and the set of hot keys found in the cache.

Trace.recordBinary(

"hot\_key\_cache",

s"${label.getOrElse("")};${foundInProcess.size};${newKeys.size};${foundInProcess.mkString(",")}"

)

}

proxyClient.getResult(newKeys).map { result =>

result.hits.foreach { case (k, v) => cacheIfPromoted(k, v) }

result.copy(hits = result.hits ++ resultsFromInProcessCache)

}

}

/\*\*

\* Exposes whether or not a key was promoted to the in-process hot key cache. In most cases, users

\* of [[HotKeyMemcacheClient]] should not need to know this. However, they may if hot key caching

\* conflicts with other layers of caching they are using.

\*/

def isHotKey(key: String): Boolean = inProcessCache.get(key).isDefined

}

// TOOD: May want to turn flags into a value class in com.twitter.finagle.memcached

// with methods for these operations

object MemcacheFlags {

val FrequencyBasedPromotion: Int = 1

val BandwidthBasedPromotion: Int = 1 << 1

val Promotable: Int = FrequencyBasedPromotion | BandwidthBasedPromotion

/\*\*

\* Memcache flags are returned as an unsigned integer, represented as a decimal string.

\*

\* Check whether the bit in position 0 ([[FrequencyBasedPromotion]]) or the bit in position 1

\* ([[BandwidthBasedPromotion]]) is set to 1 (zero-index from least-significant bit).

\*/

def shouldPromote(flagsBuf: Buf): Boolean = {

val flags = flagsBuf match { case Buf.Utf8(s) => s.toInt }

(flags & Promotable) != 0

}

}