package com.twitter.tweetypie.serverutil

import com.github.benmanes.caffeine.cache.stats.CacheStats

import com.github.benmanes.caffeine.cache.stats.StatsCounter

import com.github.benmanes.caffeine.cache.AsyncCacheLoader

import com.github.benmanes.caffeine.cache.AsyncLoadingCache

import com.github.benmanes.caffeine.cache.Caffeine

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

import com.twitter.finagle.memcached.Client

import com.twitter.finagle.memcached.GetResult

import com.twitter.finagle.memcached.ProxyClient

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Return

import com.twitter.util.Throw

import com.twitter.util.{Promise => TwitterPromise}

import com.twitter.util.logging.Logger

import java.util.concurrent.TimeUnit.NANOSECONDS

import java.util.concurrent.CompletableFuture

import java.util.concurrent.Executor

import java.util.concurrent.TimeUnit

import java.util.function.BiConsumer

import java.util.function.Supplier

import java.lang

import java.util

import scala.collection.JavaConverters.\_

object CaffeineMemcacheClient {

val logger: Logger = Logger(getClass)

/\*\*

\* Helper method to convert between Java 8's CompletableFuture and Twitter's Future.

\*/

private def toTwitterFuture[T](cf: CompletableFuture[T]): Future[T] = {

if (cf.isDone && !cf.isCompletedExceptionally && !cf.isCancelled) {

Future.const(Return(cf.get()))

} else {

val p = new TwitterPromise[T] with TwitterPromise.InterruptHandler {

override protected def onInterrupt(t: Throwable): Unit = cf.cancel(true)

}

cf.whenComplete(new BiConsumer[T, Throwable] {

override def accept(result: T, exception: Throwable): Unit = {

if (exception != null) {

p.updateIfEmpty(Throw(exception))

} else {

p.updateIfEmpty(Return(result))

}

}

})

p

}

}

}

class CaffeineMemcacheClient(

override val proxyClient: Client,

val maximumSize: Int = 1000,

val ttl: Duration = Duration.fromSeconds(10),

stats: StatsReceiver = NullStatsReceiver)

extends ProxyClient {

import CaffeineMemcacheClient.\_

private[this] object Stats extends StatsCounter {

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

private val miss = stats.counter("misses")

private val totalLoadTime = stats.stat("loads")

private val loadSuccess = stats.counter("loads-success")

private val loadFailure = stats.counter("loads-failure")

private val eviction = stats.counter("evictions")

private val evictionWeight = stats.counter("evictions-weight")

override def recordHits(i: Int): Unit = hits.incr(i)

override def recordMisses(i: Int): Unit = miss.incr(i)

override def recordLoadSuccess(l: Long): Unit = {

loadSuccess.incr()

totalLoadTime.add(NANOSECONDS.toMillis(l))

}

override def recordLoadFailure(l: Long): Unit = {

loadFailure.incr()

totalLoadTime.add(NANOSECONDS.toMillis(l))

}

override def recordEviction(): Unit = recordEviction(1)

override def recordEviction(weight: Int): Unit = {

eviction.incr()

evictionWeight.incr(weight)

}

/\*\*

\* We are currently not using this method.

\*/

override def snapshot(): CacheStats = {

new CacheStats(0, 0, 0, 0, 0, 0, 0)

}

}

private[this] object MemcachedAsyncCacheLoader extends AsyncCacheLoader[String, GetResult] {

private[this] val EmptyMisses: Set[String] = Set.empty

private[this] val EmptyFailures: Map[String, Throwable] = Map.empty

private[this] val EmptyHits: Map[String, Value] = Map.empty

override def asyncLoad(key: String, executor: Executor): CompletableFuture[GetResult] = {

val f = new util.function.Function[util.Map[String, GetResult], GetResult] {

override def apply(r: util.Map[String, GetResult]): GetResult = r.get(key)

}

asyncLoadAll(Seq(key).asJava, executor).thenApply(f)

}

/\*\*

\* Converts response from multi-key to single key. Memcache returns the result

\* in one struct that contains all the hits, misses and exceptions. Caffeine

\* requires a map from a key to the result, so we do that conversion here.

\*/

override def asyncLoadAll(

keys: lang.Iterable[\_ <: String],

executor: Executor

): CompletableFuture[util.Map[String, GetResult]] = {

val result = new CompletableFuture[util.Map[String, GetResult]]()

proxyClient.getResult(keys.asScala).respond {

case Return(r) =>

val map = new util.HashMap[String, GetResult]()

r.hits.foreach {

case (key, value) =>

map.put(

key,

r.copy(hits = Map(key -> value), misses = EmptyMisses, failures = EmptyFailures)

)

}

r.misses.foreach { key =>

map.put(key, r.copy(hits = EmptyHits, misses = Set(key), failures = EmptyFailures))

}

// We are passing through failures so that we maintain the contract expected by clients.

// Without passing through the failures, several metrics get lost. Some of these failures

// might get cached. The cache is short-lived, so we are not worried when it does

// get cached.

r.failures.foreach {

case (key, value) =>

map.put(

key,

r.copy(hits = EmptyHits, misses = EmptyMisses, failures = Map(key -> value))

)

}

result.complete(map)

case Throw(ex) =>

logger.warn("Error loading keys from memcached", ex)

result.completeExceptionally(ex)

}

result

}

}

private[this] val cache: AsyncLoadingCache[String, GetResult] =

Caffeine

.newBuilder()

.maximumSize(maximumSize)

.refreshAfterWrite(ttl.inMilliseconds \* 3 / 4, TimeUnit.MILLISECONDS)

.expireAfterWrite(ttl.inMilliseconds, TimeUnit.MILLISECONDS)

.recordStats(new Supplier[StatsCounter] {

override def get(): StatsCounter = Stats

})

.buildAsync(MemcachedAsyncCacheLoader)

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

val twitterFuture = toTwitterFuture(cache.getAll(keys.asJava))

twitterFuture

.map { result =>

val values = result.values().asScala

values.reduce(\_ ++ \_)

}

}

}