package com.twitter.servo.cache

import com.twitter.finagle.memcached.{CasResult, Client}

import com.twitter.finagle.service.RetryPolicy

import com.twitter.finagle.{Backoff, Memcached, TimeoutException, WriteException}

import com.twitter.hashing.KeyHasher

import com.twitter.io.Buf

import com.twitter.logging.Logger

import com.twitter.util.\_

case class MemcacheRetryPolicy(

writeExceptionBackoffs: Backoff,

timeoutBackoffs: Backoff)

extends RetryPolicy[Try[Nothing]] {

override def apply(r: Try[Nothing]) = r match {

case Throw(\_: WriteException) => onWriteException

case Throw(\_: TimeoutException) => onTimeoutException

case \_ => None

}

private[this] def onTimeoutException = consume(timeoutBackoffs.toStream) { tail =>

copy(timeoutBackoffs = Backoff.fromStream(tail))

}

private[this] def onWriteException = consume(writeExceptionBackoffs.toStream) { tail =>

copy(writeExceptionBackoffs = Backoff.fromStream(tail))

}

private[this] def consume(s: Stream[Duration])(f: Stream[Duration] => MemcacheRetryPolicy) = {

s.headOption map { duration =>

(duration, f(s.tail))

}

}

}

object FinagleMemcacheFactory {

val DefaultHashName = "fnv1-32"

def apply(client: Memcached.Client, dest: String, hashName: String = DefaultHashName) =

new FinagleMemcacheFactory(client, dest, hashName)

}

class FinagleMemcacheFactory private[cache] (

client: Memcached.Client,

dest: String,

hashName: String)

extends MemcacheFactory {

def apply(): Memcache = {

val keyHasher = KeyHasher.byName(hashName)

new FinagleMemcache(client.withKeyHasher(keyHasher).newTwemcacheClient(dest), hashName)

}

}

object FinagleMemcache {

val NoFlags = 0

val logger = Logger(getClass)

}

/\*\*

\* Adapter for a [[Memcache]] (type alias for [[TtlCache]]) from a Finagle Memcached

\* [[Client]].

\*/

class FinagleMemcache(client: Client, hashName: String = FinagleMemcacheFactory.DefaultHashName)

extends Memcache {

import FinagleMemcache.NoFlags

private[this] case class BufferChecksum(buffer: Buf) extends Checksum

def release(): Unit = {

client.close()

}

override def get(keys: Seq[String]): Future[KeyValueResult[String, Array[Byte]]] =

client.getResult(keys).transform {

case Return(gr) =>

val found = gr.hits.map {

case (key, v) =>

val bytes = Buf.ByteArray.Owned.extract(v.value)

key -> bytes

}

Future.value(KeyValueResult(found, gr.misses, gr.failures))

case Throw(t) =>

Future.value(KeyValueResult(failed = keys.map(\_ -> t).toMap))

}

override def getWithChecksum(keys: Seq[String]): Future[CsKeyValueResult[String, Array[Byte]]] =

client.getsResult(keys).transform {

case Return(gr) =>

try {

val hits = gr.hits map {

case (key, v) =>

val bytes = Buf.ByteArray.Owned.extract(v.value)

key -> (Return(bytes), BufferChecksum(

v.casUnique.get

)) // TODO. what to do if missing?

}

Future.value(KeyValueResult(hits, gr.misses, gr.failures))

} catch {

case t: Throwable =>

Future.value(KeyValueResult(failed = keys.map(\_ -> t).toMap))

}

case Throw(t) =>

Future.value(KeyValueResult(failed = keys.map(\_ -> t).toMap))

}

private val jb2sb: java.lang.Boolean => Boolean = \_.booleanValue

private val jl2sl: java.lang.Long => Long = \_.longValue

override def add(key: String, value: Array[Byte], ttl: Duration): Future[Boolean] =

client.add(key, NoFlags, ttl.fromNow, Buf.ByteArray.Owned(value)) map jb2sb

override def checkAndSet(

key: String,

value: Array[Byte],

checksum: Checksum,

ttl: Duration

): Future[Boolean] = {

checksum match {

case BufferChecksum(cs) =>

client.checkAndSet(key, NoFlags, ttl.fromNow, Buf.ByteArray.Owned(value), cs) map {

res: CasResult =>

res.replaced

}

case \_ =>

Future.exception(new IllegalArgumentException("unrecognized checksum: " + checksum))

}

}

override def set(key: String, value: Array[Byte], ttl: Duration): Future[Unit] =

client.set(key, NoFlags, ttl.fromNow, Buf.ByteArray.Owned(value))

override def replace(key: String, value: Array[Byte], ttl: Duration): Future[Boolean] =

client.replace(key, NoFlags, ttl.fromNow, Buf.ByteArray.Owned(value)) map jb2sb

override def delete(key: String): Future[Boolean] =

client.delete(key) map jb2sb

def incr(key: String, delta: Long = 1): Future[Option[Long]] =

client.incr(key, delta) map { \_ map jl2sl }

def decr(key: String, delta: Long = 1): Future[Option[Long]] =

client.decr(key, delta) map { \_ map jl2sl }

// NOTE: This is the only reason that hashName is passed as a param to FinagleMemcache.

override lazy val toString = "FinagleMemcache(%s)".format(hashName)

}