package com.twitter.tweetypie.caching

import com.twitter.finagle.service.StatsFilter

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.stats.ExceptionStatsHandler

import com.twitter.finagle.stats.Counter

import com.twitter.util.Future

import com.twitter.util.logging.Logger

import com.twitter.finagle.memcached

import scala.util.control.NonFatal

/\*\*

\* Wrapper around a memcached client that performs serialization and

\* deserialization, tracks stats, provides tracing, and provides

\* per-key fresh/stale/failure/miss results.

\*

\* The operations that write values to cache will only write values

\* that the ValueSerializer says are cacheable. The idea here is that

\* the deserialize and serialize functions must be coherent, and no

\* matter how you choose to write these values back to cache, the

\* serializer will have the appropriate knowledge about whether the

\* values are cacheable.

\*

\* For most cases, you will want to use [[StitchCaching]] rather than

\* calling this wrapper directly.

\*

\* @param keySerializer How to convert a K value to a memcached key.

\*

\* @param valueSerializer How to serialize and deserialize V values,

\* as well as which values are cacheable, and how long to store the

\* values in cache.

\*/

class CacheOperations[K, V](

keySerializer: K => String,

valueSerializer: ValueSerializer[V],

memcachedClient: memcached.Client,

statsReceiver: StatsReceiver,

logger: Logger,

exceptionStatsHandler: ExceptionStatsHandler = StatsFilter.DefaultExceptions) {

// The memcached operations that are performed via this

// [[CacheOperations]] instance will be tracked under this stats

// receiver.

//

// We count all memcached failures together under this scope,

// because memcached operations should not fail unless there are

// communication problems, so differentiating the method that was

// being called will not give us any useful information.

private[this] val memcachedStats: StatsReceiver = statsReceiver.scope("memcached")

// Incremented for every attempt to `get` a key from cache.

private[this] val memcachedGetCounter: Counter = memcachedStats.counter("get")

// One of these two counters is incremented for every successful

// response returned from a `get` call to memcached.

private[this] val memcachedNotFoundCounter: Counter = memcachedStats.counter("not\_found")

private[this] val memcachedFoundCounter: Counter = memcachedStats.counter("found")

// Records the state of the cache load after serialization. The

// policy may transform a value that was successfully loaded from

// cache into any result type, which is why we explicitly track

// "found" and "not\_found" above. If `stale` + `fresh` is not equal

// to `found`, then it means that the policy has translated a found

// value into a miss or failure. The policy may do this in order to

// cause the caching filter to treat the value that was found in

// cache in the way it would have treated a miss or failure from

// cache.

private[this] val resultStats: StatsReceiver = statsReceiver.scope("result")

private[this] val resultFreshCounter: Counter = resultStats.counter("fresh")

private[this] val resultStaleCounter: Counter = resultStats.counter("stale")

private[this] val resultMissCounter: Counter = resultStats.counter("miss")

private[this] val resultFailureCounter: Counter = resultStats.counter("failure")

// Used for recording exceptions that occurred during

// deserialization. This will never be incremented if the

// deserializer returns a result, even if the result is a

// [[CacheResult.Failure]]. See the comment where this stat is

// incremented for more details.

private[this] val deserializeFailureStats: StatsReceiver = statsReceiver.scope("deserialize")

private[this] val notSerializedCounter: Counter = statsReceiver.counter("not\_serialized")

/\*\*

\* Load a batch of values from cache. Mostly this deals with

\* converting the [[memcached.GetResult]] to a

\* [[Seq[CachedResult[V]]]]. The result is in the same order as the

\* keys, and there will always be an entry for each key. This method

\* should never return a [[Future.exception]].

\*/

def get(keys: Seq[K]): Future[Seq[CacheResult[V]]] = {

memcachedGetCounter.incr(keys.size)

val cacheKeys: Seq[String] = keys.map(keySerializer)

if (logger.isTraceEnabled) {

logger.trace {

val lines: Seq[String] = keys.zip(cacheKeys).map { case (k, c) => s"\n $k ($c)" }

"Starting load for keys:" + lines.mkString

}

}

memcachedClient

.getResult(cacheKeys)

.map { getResult =>

memcachedNotFoundCounter.incr(getResult.misses.size)

val results: Seq[CacheResult[V]] =

cacheKeys.map { cacheKey =>

val result: CacheResult[V] =

getResult.hits.get(cacheKey) match {

case Some(memcachedValue) =>

memcachedFoundCounter.incr()

try {

valueSerializer.deserialize(memcachedValue.value)

} catch {

case NonFatal(e) =>

// If the serializer throws an exception, then

// the serialized value was malformed. In that

// case, we record the failure so that it can be

// detected and fixed, but treat it as a cache

// miss. The reason that we treat it as a miss

// rather than a failure is that a miss will

// cause a write back to cache, and we want to

// write a valid result back to cache to replace

// the bad entry that we just loaded.

//

// A serializer is free to return Miss itself to

// obtain this behavior if it is expected or

// desired, to avoid the logging and stats (and

// the minor overhead of catching an exception).

//

// The exceptions are tracked separately from

// other exceptions so that it is easy to see

// whether the deserializer itself ever throws an

// exception.

exceptionStatsHandler.record(deserializeFailureStats, e)

logger.warn(s"Failed deserializing value for cache key $cacheKey", e)

CacheResult.Miss

}

case None if getResult.misses.contains(cacheKey) =>

CacheResult.Miss

case None =>

val exception =

getResult.failures.get(cacheKey) match {

case None =>

// To get here, this was not a hit or a miss,

// so we expect the key to be present in

// failures. If it is not, then either the

// contract of getResult was violated, or this

// method is somehow attempting to access a

// result for a key that was not

// loaded. Either of these indicates a bug, so

// we log a high priority log message.

logger.error(

s"Key $cacheKey not found in hits, misses or failures. " +

"This indicates a bug in the memcached library or " +

"CacheOperations.load"

)

// We return this as a failure because that

// will cause the repo to be consulted and the

// value \*not\* to be written back to cache,

// which is probably the safest thing to do

// (if we don't know what's going on, default

// to an uncached repo).

new IllegalStateException

case Some(e) =>

e

}

exceptionStatsHandler.record(memcachedStats, exception)

CacheResult.Failure(exception)

}

// Count each kind of CacheResult, to make it possible to

// see how effective the caching is.

result match {

case CacheResult.Fresh(\_) => resultFreshCounter.incr()

case CacheResult.Stale(\_) => resultStaleCounter.incr()

case CacheResult.Miss => resultMissCounter.incr()

case CacheResult.Failure(\_) => resultFailureCounter.incr()

}

result

}

if (logger.isTraceEnabled) {

logger.trace {

val lines: Seq[String] =

(keys, cacheKeys, results).zipped.map {

case (key, cacheKey, result) => s"\n $key ($cacheKey) -> $result"

}

"Cache results:" + lines.mkString

}

}

results

}

.handle {

case e =>

// If there is a failure from the memcached client, fan it

// out to each cache key, so that the caller does not need

// to handle failure of the batch differently than failure

// of individual keys. This should be rare anyway, since the

// memcached client already does this for common Finagle

// exceptions

resultFailureCounter.incr(keys.size)

val theFailure: CacheResult[V] = CacheResult.Failure(e)

keys.map { \_ =>

// Record this as many times as we would if it were in the GetResult

exceptionStatsHandler.record(memcachedStats, e)

theFailure

}

}

}

// Incremented for every attempt to `set` a key in value.

private[this] val memcachedSetCounter: Counter = memcachedStats.counter("set")

/\*\*

\* Write an entry back to cache, using `set`. If the serializer does

\* not serialize the value, then this method will immediately return

\* with success.

\*/

def set(key: K, value: V): Future[Unit] =

valueSerializer.serialize(value) match {

case Some((expiry, serialized)) =>

if (logger.isTraceEnabled) {

logger.trace(s"Writing back to cache $key -> $value (expiry = $expiry)")

}

memcachedSetCounter.incr()

memcachedClient

.set(key = keySerializer(key), flags = 0, expiry = expiry, value = serialized)

.onFailure(exceptionStatsHandler.record(memcachedStats, \_))

case None =>

if (logger.isTraceEnabled) {

logger.trace(s"Not writing back $key -> $value")

}

notSerializedCounter.incr()

Future.Done

}

}