package com.twitter.servo.cache

import com.google.common.cache.CacheBuilder

import com.twitter.finagle.memcached.util.NotFound

import com.twitter.servo.util.ThreadLocalStringBuilder

import com.twitter.util.{Duration, Future, Return}

import java.util.concurrent.TimeUnit

import scala.collection.mutable

import scala.collection.JavaConverters.\_

/\*\*

\* opaque trait used for getWithChecksum calls.

\* the implementation should be private to the cache,

\* to inhibit peeking

\*/

trait Checksum extends Any

object ScopedCacheKey {

private[ScopedCacheKey] val builder = new ThreadLocalStringBuilder(64)

}

/\*\*

\* base class for cache keys needing scoping

\*

\* @param globalNamespace

\* the project-level namespace

\* @param cacheNamespace

\* the cache-level namespace

\* @param version

\* the version of serialization for values

\* @param scopes

\* additional key scopes

\*/

abstract class ScopedCacheKey(

globalNamespace: String,

cacheNamespace: String,

version: Int,

scopes: String\*) {

import constants.\_

override lazy val toString = {

val builder = ScopedCacheKey

.builder()

.append(globalNamespace)

.append(Colon)

.append(cacheNamespace)

.append(Colon)

.append(version)

scopes foreach {

builder.append(Colon).append(\_)

}

builder.toString

}

}

/\*\*

\* Shared trait for reading from a cache

\*/

trait ReadCache[K, V] {

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

/\*\*

\* get the value with an opaque checksum that can be passed in

\* a checkAndSet operation. If there is a deserialization error,

\* the checksum is still returned

\*/

def getWithChecksum(keys: Seq[K]): Future[CsKeyValueResult[K, V]]

/\*\*

\* release any underlying resources

\*/

def release(): Unit

}

/\*\*

\* allows one ReadCache to wrap another

\*/

trait ReadCacheWrapper[K, V, This <: ReadCache[K, V]] extends ReadCache[K, V] {

def underlyingCache: This

override def get(keys: Seq[K]) = underlyingCache.get(keys)

override def getWithChecksum(keys: Seq[K]) = underlyingCache.getWithChecksum(keys)

override def release() = underlyingCache.release()

}

/\*\*

\* Simple trait for a cache supporting multi-get and single set

\*/

trait Cache[K, V] extends ReadCache[K, V] {

def add(key: K, value: V): Future[Boolean]

def checkAndSet(key: K, value: V, checksum: Checksum): Future[Boolean]

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

def set(pairs: Seq[(K, V)]): Future[Unit] = {

Future.join {

pairs map {

case (key, value) => set(key, value)

}

}

}

/\*\*

\* Replaces the value for an existing key. If the key doesn't exist, this has no effect.

\* @return true if replaced, false if not found

\*/

def replace(key: K, value: V): Future[Boolean]

/\*\*

\* Deletes a value from cache.

\* @return true if deleted, false if not found

\*/

def delete(key: K): Future[Boolean]

}

/\*\*

\* allows one cache to wrap another

\*/

trait CacheWrapper[K, V] extends Cache[K, V] with ReadCacheWrapper[K, V, Cache[K, V]] {

override def add(key: K, value: V) = underlyingCache.add(key, value)

override def checkAndSet(key: K, value: V, checksum: Checksum) =

underlyingCache.checkAndSet(key, value, checksum)

override def set(key: K, value: V) = underlyingCache.set(key, value)

override def replace(key: K, value: V) = underlyingCache.replace(key, value)

override def delete(key: K) = underlyingCache.delete(key)

}

/\*\*

\* Switch between two caches with a decider value

\*/

class DeciderableCache[K, V](primary: Cache[K, V], secondary: Cache[K, V], isAvailable: => Boolean)

extends CacheWrapper[K, V] {

override def underlyingCache = if (isAvailable) primary else secondary

}

private object MutableMapCache {

case class IntChecksum(i: Int) extends AnyVal with Checksum

}

/\*\*

\* implementation of a Cache with a mutable.Map

\*/

class MutableMapCache[K, V](underlying: mutable.Map[K, V]) extends Cache[K, V] {

import MutableMapCache.IntChecksum

protected[this] def checksum(value: V): Checksum = IntChecksum(value.hashCode)

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

val founds = Map.newBuilder[K, V]

val iter = keys.iterator

while (iter.hasNext) {

val key = iter.next()

synchronized {

underlying.get(key)

} match {

case Some(v) => founds += key -> v

case None =>

}

}

val found = founds.result()

val notFound = NotFound(keys, found.keySet)

KeyValueResult(found, notFound)

}

override def getWithChecksum(keys: Seq[K]): Future[CsKeyValueResult[K, V]] = Future {

val founds = Map.newBuilder[K, (Return[V], Checksum)]

val iter = keys.iterator

while (iter.hasNext) {

val key = iter.next()

synchronized {

underlying.get(key)

} match {

case Some(value) => founds += key -> (Return(value), checksum(value))

case None =>

}

}

val found = founds.result()

val notFound = NotFound(keys, found.keySet)

KeyValueResult(found, notFound)

}

override def add(key: K, value: V): Future[Boolean] =

synchronized {

underlying.get(key) match {

case Some(\_) =>

Future.False

case None =>

underlying += key -> value

Future.True

}

}

override def checkAndSet(key: K, value: V, cs: Checksum): Future[Boolean] =

synchronized {

underlying.get(key) match {

case Some(current) =>

if (checksum(current) == cs) {

// checksums match, set value

underlying += key -> value

Future.True

} else {

// checksums didn't match, so no set

Future.False

}

case None =>

// if nothing there, the checksums can't be compared

Future.False

}

}

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

synchronized {

underlying += key -> value

}

Future.Done

}

override def replace(key: K, value: V): Future[Boolean] = synchronized {

if (underlying.contains(key)) {

underlying(key) = value

Future.True

} else {

Future.False

}

}

override def delete(key: K): Future[Boolean] = synchronized {

if (underlying.remove(key).nonEmpty) Future.True else Future.False

}

override def release(): Unit = synchronized {

underlying.clear()

}

}

/\*\*

\* In-memory implementation of a cache with LRU semantics and a TTL.

\*/

class ExpiringLruCache[K, V](ttl: Duration, maximumSize: Int)

extends MutableMapCache[K, V](

// TODO: consider wiring the Cache interface directly to the

// Guava Cache, instead of introducing two layers of indirection

CacheBuilder.newBuilder

.asInstanceOf[CacheBuilder[K, V]]

.expireAfterWrite(ttl.inMilliseconds, TimeUnit.MILLISECONDS)

.initialCapacity(maximumSize)

.maximumSize(maximumSize)

.build[K, V]()

.asMap

.asScala

)

/\*\*

\* An empty cache that stays empty

\*/

class NullCache[K, V] extends Cache[K, V] {

lazy val futureTrue = Future.value(true)

override def get(keys: Seq[K]) = Future.value(KeyValueResult(notFound = keys.toSet))

override def getWithChecksum(keys: Seq[K]) = Future.value(KeyValueResult(notFound = keys.toSet))

override def add(key: K, value: V) = futureTrue

override def checkAndSet(key: K, value: V, checksum: Checksum) = Future.value(true)

override def set(key: K, value: V) = Future.Done

override def replace(key: K, value: V) = futureTrue

override def delete(key: K) = futureTrue

override def release() = ()

}