package com.twitter.servo.cache

import com.twitter.finagle.stats.{Stat, StatsReceiver}

import com.twitter.logging.{Level, Logger}

import com.twitter.servo.util.{ExceptionCounter, WindowedAverage}

import com.twitter.util.\_

/\*\*

\* track hits and misses in caches, time reads and writes

\*/

trait CacheObserver {

/\*\*

\* register a hit

\*/

def hit(key: String): Unit

/\*\*

\* register a miss

\*/

def miss(key: String): Unit

/\*\*

\* time the read, and automatically handle hits and misses from the KeyValueResult

\*/

def read[K, T](

name: String,

keys: Seq[K]

)(

f: => Future[KeyValueResult[K, T]]

): Future[KeyValueResult[K, T]]

/\*\*

\* time the write

\*/

def write[K, T](name: String, key: K)(f: => Future[T]): Future[T]

/\*\*

\* time the incr, and record the success/failure

\*/

def incr[K](name: String, key: Seq[K])(f: => Future[Option[Long]]): Future[Option[Long]]

/\*\*

\* produce a new CacheObserver with a nested scope

\*/

def scope(s: String\*): CacheObserver

/\*\*

\* increment a counter tracking the number of expirations.

\*/

def expired(delta: Int = 1): Unit

/\*\*

\* Increment a counter tracking the number of failures.

\*/

def failure(delta: Int = 1): Unit

/\*\*

\* Increment a counter tracking the number of tombstones.

\*/

def tombstone(delta: Int = 1): Unit

/\*\*

\* Increment a counter tracking the number of not cached.

\*/

def noCache(delta: Int = 1): Unit

}

object NullCacheObserver extends CacheObserver {

override def hit(key: String) = ()

override def miss(key: String) = ()

override def read[K, T](name: String, keys: Seq[K])(f: => Future[KeyValueResult[K, T]]) = f

override def write[K, T](name: String, key: K)(f: => Future[T]) = f

override def incr[K](name: String, key: Seq[K])(f: => Future[Option[Long]]) = f

override def scope(s: String\*) = this

override def expired(delta: Int = 1) = ()

override def failure(delta: Int = 1): Unit = {}

override def tombstone(delta: Int = 1): Unit = {}

override def noCache(delta: Int = 1): Unit = {}

}

/\*\*

\* A CacheObserver that writes to a StatsReceiver

\*/

class StatsReceiverCacheObserver(

stats: StatsReceiver,

windowSize: Long,

log: Logger,

disableLogging: Boolean = false)

extends CacheObserver {

def this(

statsReceiver: StatsReceiver,

windowSize: Long,

scope: String

) =

this(

statsReceiver.scope(scope),

windowSize,

Logger.get(scope.replaceAll("([a-z]+)([A-Z])", "$1\_$2").toLowerCase)

)

def this(

statsReceiver: StatsReceiver,

windowSize: Long,

scope: String,

disableLogging: Boolean

) =

this(

statsReceiver.scope(scope),

windowSize,

Logger.get(scope.replaceAll("([a-z]+)([A-Z])", "$1\_$2").toLowerCase),

disableLogging

)

protected[this] val expirationCounter = stats.counter("expirations")

// needed to make sure we hand out the same observer for each scope,

// so that the hit rates are properly calculated

protected[this] val children = Memoize {

new StatsReceiverCacheObserver(stats, windowSize, \_: String, disableLogging)

}

protected[this] val exceptionCounter = new ExceptionCounter(stats)

private[this] val hitCounter = stats.counter("hits")

private[this] val missCounter = stats.counter("misses")

private[this] val failuresCounter = stats.counter("failures")

private[this] val tombstonesCounter = stats.counter("tombstones")

private[this] val noCacheCounter = stats.counter("noCache")

private[this] val windowedHitRate = new WindowedAverage(windowSize)

private[this] val windowedIncrHitRate = new WindowedAverage(windowSize)

private[this] val hitRateGauge = stats.addGauge("hit\_rate") {

windowedHitRate.value.getOrElse(1.0).toFloat

}

private[this] val incrHitRateGauge = stats.addGauge("incr\_hit\_rate") {

windowedIncrHitRate.value.getOrElse(1.0).toFloat

}

protected[this] def handleThrowable[K](name: String, t: Throwable, key: Option[K]): Unit = {

stats.counter(name + "\_failures").incr()

exceptionCounter(t)

if (!disableLogging) {

lazy val suffix = key

.map { k =>

"(" + k.toString + ")"

}

.getOrElse("")

log.warning("%s%s caught: %s", name, suffix, t.getClass.getName)

log.trace(t, "stack trace was: ")

}

}

override def hit(key: String): Unit = {

hits(1)

if (!disableLogging)

log.trace("cache hit: %s", key)

}

private[this] def hits(n: Int): Unit = {

windowedHitRate.record(n.toDouble, n.toDouble)

hitCounter.incr(n)

}

override def miss(key: String): Unit = {

misses(1)

if (!disableLogging)

log.trace("cache miss: %s", key)

}

private[this] def misses(n: Int): Unit = {

windowedHitRate.record(0.0F, n.toDouble)

missCounter.incr(n)

}

override def read[K, T](

name: String,

keys: Seq[K]

)(

f: => Future[KeyValueResult[K, T]]

): Future[KeyValueResult[K, T]] =

Stat

.timeFuture(stats.stat(name)) {

stats.counter(name).incr()

f

}

.respond {

case Return(lr) =>

if (log.isLoggable(Level.TRACE)) {

lr.found.keys.foreach { k =>

hit(k.toString)

}

lr.notFound.foreach { k =>

miss(k.toString)

}

} else {

hits(lr.found.keys.size)

misses(lr.notFound.size)

}

lr.failed foreach {

case (k, t) =>

handleThrowable(name, t, Some(k))

// count failures as misses

miss(k.toString)

failuresCounter.incr()

}

case Throw(t) =>

handleThrowable(name, t, None)

// count failures as misses

keys.foreach { k =>

miss(k.toString)

}

failuresCounter.incr()

}

override def write[K, T](name: String, key: K)(f: => Future[T]): Future[T] =

Stat.timeFuture(stats.stat(name)) {

stats.counter(name).incr()

f

} onFailure {

handleThrowable(name, \_, Some(key))

}

override def incr[K](name: String, key: Seq[K])(f: => Future[Option[Long]]) =

Stat.timeFuture(stats.stat(name)) {

stats.counter(name).incr()

f

} onSuccess { optVal =>

val hit = optVal.isDefined

windowedIncrHitRate.record(if (hit) 1F else 0F)

stats.counter(name + (if (hit) "\_hits" else "\_misses")).incr()

}

override def scope(s: String\*) =

s.toList match {

case Nil => this

case head :: tail => children(head).scope(tail: \_\*)

}

override def expired(delta: Int = 1): Unit = { expirationCounter.incr(delta) }

override def failure(delta: Int = 1): Unit = { failuresCounter.incr(delta) }

override def tombstone(delta: Int = 1): Unit = { tombstonesCounter.incr(delta) }

override def noCache(delta: Int = 1): Unit = { noCacheCounter.incr(delta) }

}

/\*\*

\* Wraps an underlying cache with calls to a CacheObserver

\*/

class ObservableReadCache[K, V](underlyingCache: ReadCache[K, V], observer: CacheObserver)

extends ReadCache[K, V] {

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

observer.read("get", keys) {

underlyingCache.get(keys)

}

}

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

observer.read[K, (Try[V], Checksum)]("get\_with\_checksum", keys) {

underlyingCache.getWithChecksum(keys)

}

}

override def release() = underlyingCache.release()

}

object ObservableCache {

def apply[K, V](

underlyingCache: Cache[K, V],

statsReceiver: StatsReceiver,

windowSize: Long,

name: String

): Cache[K, V] =

new ObservableCache(

underlyingCache,

new StatsReceiverCacheObserver(statsReceiver, windowSize, name)

)

def apply[K, V](

underlyingCache: Cache[K, V],

statsReceiver: StatsReceiver,

windowSize: Long,

name: String,

disableLogging: Boolean

): Cache[K, V] =

new ObservableCache(

underlyingCache,

new StatsReceiverCacheObserver(

statsReceiver = statsReceiver,

windowSize = windowSize,

scope = name,

disableLogging = disableLogging)

)

def apply[K, V](

underlyingCache: Cache[K, V],

statsReceiver: StatsReceiver,

windowSize: Long,

log: Logger

): Cache[K, V] =

new ObservableCache(

underlyingCache,

new StatsReceiverCacheObserver(statsReceiver, windowSize, log)

)

}

/\*\*

\* Wraps an underlying Cache with calls to a CacheObserver

\*/

class ObservableCache[K, V](underlyingCache: Cache[K, V], observer: CacheObserver)

extends ObservableReadCache(underlyingCache, observer)

with Cache[K, V] {

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

observer.write("add", key) {

underlyingCache.add(key, value)

}

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

observer.write("check\_and\_set", key) {

underlyingCache.checkAndSet(key, value, checksum)

}

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

observer.write("set", key) {

underlyingCache.set(key, value)

}

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

observer.write("replace", key) {

underlyingCache.replace(key, value)

}

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

observer.write("delete", key) {

underlyingCache.delete(key)

}

}

object ObservableTtlCache {

def apply[K, V](

underlyingCache: TtlCache[K, V],

statsReceiver: StatsReceiver,

windowSize: Long,

name: String

): TtlCache[K, V] =

new ObservableTtlCache(

underlyingCache,

new StatsReceiverCacheObserver(statsReceiver, windowSize, name)

)

}

/\*\*

\* Wraps an underlying TtlCache with calls to a CacheObserver

\*/

class ObservableTtlCache[K, V](underlyingCache: TtlCache[K, V], observer: CacheObserver)

extends ObservableReadCache(underlyingCache, observer)

with TtlCache[K, V] {

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

observer.write("add", key) {

underlyingCache.add(key, value, ttl)

}

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

observer.write("check\_and\_set", key) {

underlyingCache.checkAndSet(key, value, checksum, ttl)

}

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

observer.write("set", key) {

underlyingCache.set(key, value, ttl)

}

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

observer.write("replace", key) {

underlyingCache.replace(key, value, ttl)

}

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

observer.write("delete", key) {

underlyingCache.delete(key)

}

}

case class ObservableMemcacheFactory(memcacheFactory: MemcacheFactory, cacheObserver: CacheObserver)

extends MemcacheFactory {

override def apply() =

new ObservableMemcache(memcacheFactory(), cacheObserver)

}

@deprecated("use ObservableMemcacheFactory or ObservableMemcache directly", "0.1.2")

object ObservableMemcache {

def apply(

underlyingCache: Memcache,

statsReceiver: StatsReceiver,

windowSize: Long,

name: String

): Memcache =

new ObservableMemcache(

underlyingCache,

new StatsReceiverCacheObserver(statsReceiver, windowSize, name)

)

}

class ObservableMemcache(underlyingCache: Memcache, observer: CacheObserver)

extends ObservableTtlCache[String, Array[Byte]](underlyingCache, observer)

with Memcache {

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

observer.incr("incr", key) {

underlyingCache.incr(key, delta)

}

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

observer.incr("decr", key) {

underlyingCache.decr(key, delta)

}

}