package com.twitter.servo.cache

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.servo.cache.thriftscala.CachedValueStatus.DoNotCache

import com.twitter.servo.util.{Gate, Transformer}

import com.twitter.util.{Duration, Return, Throw, Time}

import java.nio.ByteBuffer

object Cached {

private[this] val millisToTime: Long => Time =

ms => Time.fromMilliseconds(ms)

private val timeToMills: Time => Long =

time => time.inMilliseconds

/\*\*

\* Deserialize a CachedValue to a Cached[V]

\*

\* If the ByteBuffer contained in the `cachedValue` is backed by an `Array[Byte]` with its offset

\* at 0, we will apply the serializer directly to the backing array for performance reasons.

\*

\* As such, the `Serializer[V]` the caller provides MUST NOT mutate the buffer it is given.

\* This exhortation is also given in com.twitter.servo.util.Transformer, but repeated here.

\*/

def apply[V](cachedValue: CachedValue, serializer: Serializer[V]): Cached[V] = {

val value: Option[V] = cachedValue.value match {

case Some(buf) if buf.hasArray && buf.arrayOffset() == 0 =>

serializer.from(buf.array).toOption

case Some(buf) =>

val array = new Array[Byte](buf.remaining)

buf.duplicate.get(array)

serializer.from(array).toOption

case None => None

}

val status =

if (cachedValue.value.nonEmpty && value.isEmpty)

CachedValueStatus.DeserializationFailed

else

cachedValue.status

Cached(

value,

status,

Time.fromMilliseconds(cachedValue.cachedAtMsec),

cachedValue.readThroughAtMsec.map(millisToTime),

cachedValue.writtenThroughAtMsec.map(millisToTime),

cachedValue.doNotCacheUntilMsec.map(millisToTime),

cachedValue.softTtlStep

)

}

}

/\*\*

\* A simple metadata wrapper for cached values. This is stored in the cache

\* using the [[com.twitter.servo.cache.thriftscala.CachedValue]] struct, which is similar, but

\* untyped.

\*/

case class Cached[V](

value: Option[V],

status: CachedValueStatus,

cachedAt: Time,

readThroughAt: Option[Time] = None,

writtenThroughAt: Option[Time] = None,

doNotCacheUntil: Option[Time] = None,

softTtlStep: Option[Short] = None) {

/\*\*

\* produce a new cached value with the same metadata

\*/

def map[W](f: V => W): Cached[W] = copy(value = value.map(f))

/\*\*

\* serialize to a CachedValue

\*/

def toCachedValue(serializer: Serializer[V]): CachedValue = {

var serializedValue: Option[ByteBuffer] = None

val cachedValueStatus = value match {

case Some(v) =>

serializer.to(v) match {

case Return(sv) =>

serializedValue = Some(ByteBuffer.wrap(sv))

status

case Throw(\_) => CachedValueStatus.SerializationFailed

}

case None => status

}

CachedValue(

serializedValue,

cachedValueStatus,

cachedAt.inMilliseconds,

readThroughAt.map(Cached.timeToMills),

writtenThroughAt.map(Cached.timeToMills),

doNotCacheUntil.map(Cached.timeToMills),

softTtlStep

)

}

/\*\*

\* Resolves conflicts between a value being inserted into cache and a value already in cache by

\* using the time a cached value was last updated.

\* If the cached value has a writtenThroughAt, returns it. Otherwise returns readThroughAt, but

\* if that doesn't exist, returns cachedAt.

\* This makes it favor writes to reads in the event of a race condition.

\*/

def effectiveUpdateTime[V](writtenThroughBuffer: Duration = 0.second): Time = {

this.writtenThroughAt match {

case Some(wta) => wta + writtenThroughBuffer

case None =>

this.readThroughAt match {

case Some(rta) => rta

case None => this.cachedAt

}

}

}

}

/\*\*

\* Switch between two cache pickers by providing deciderable gate

\*/

class DeciderablePicker[V](

primaryPicker: LockingCache.Picker[Cached[V]],

secondaryPicker: LockingCache.Picker[Cached[V]],

usePrimary: Gate[Unit],

statsReceiver: StatsReceiver)

extends LockingCache.Picker[Cached[V]] {

private[this] val stats = statsReceiver.scope("deciderable\_picker")

private[this] val pickerScope = stats.scope("picker")

private[this] val primaryPickerCount = pickerScope.counter("primary")

private[this] val secondaryPickerCount = pickerScope.counter("secondary")

private[this] val pickedScope = stats.scope("picked\_values")

private[this] val pickedValuesMatched = pickedScope.counter("matched")

private[this] val pickedValuesMismatched = pickedScope.counter("mismatched")

override def apply(newValue: Cached[V], oldValue: Cached[V]): Option[Cached[V]] = {

val secondaryPickerValue = secondaryPicker(newValue, oldValue)

if (usePrimary()) {

val primaryPickerValue = primaryPicker(newValue, oldValue)

primaryPickerCount.incr()

if (primaryPickerValue == secondaryPickerValue) pickedValuesMatched.incr()

else pickedValuesMismatched.incr()

primaryPickerValue

} else {

secondaryPickerCount.incr()

secondaryPickerValue

}

}

override def toString(): String = "DeciderablePicker"

}

/\*\*

\* It's similar to the PreferNewestCached picker, but it prefers written-through value

\* over read-through as long as written-through value + writtenThroughExtra is

\* newer than read-through value. Same as in PreferNewestCached, if values cached

\* have the same cached method and time picker picks the new value.

\*

\* It intends to solve race condition when the read and write requests come at the

\* same time, but write requests is getting cached first and then getting override with

\* a stale value from the read request.

\*

\* If enabled gate is disabled, it falls back to PreferNewestCached logic.

\*

\*/

class PreferWrittenThroughCached[V](

writtenThroughBuffer: Duration = 1.second)

extends PreferNewestCached[V] {

override def apply(newValue: Cached[V], oldValue: Cached[V]): Option[Cached[V]] = {

// the tie goes to newValue

if (oldValue.effectiveUpdateTime(writtenThroughBuffer) > newValue.effectiveUpdateTime(

writtenThroughBuffer))

None

else

Some(newValue)

}

override def toString(): String = "PreferWrittenThroughCached"

}

/\*\*

\* prefer one value over another based on Cached metadata

\*/

class PreferNewestCached[V] extends LockingCache.Picker[Cached[V]] {

override def apply(newValue: Cached[V], oldValue: Cached[V]): Option[Cached[V]] = {

if (oldValue.effectiveUpdateTime() > newValue.effectiveUpdateTime())

None

else

Some(newValue)

}

override def toString(): String = "PreferNewestCached"

}

/\*\*

\* Prefer non-empty values. If a non-empty value is in cache, and the

\* value to store is empty, return the non-empty value with a fresh cachedAt

\* instead.

\*/

class PreferNewestNonEmptyCached[V] extends PreferNewestCached[V] {

override def apply(newValue: Cached[V], oldValue: Cached[V]) = {

(newValue.value, oldValue.value) match {

// Some/Some and None/None cases are handled by the super class

case (Some(\_), Some(\_)) => super.apply(newValue, oldValue)

case (None, None) => super.apply(newValue, oldValue)

case (Some(\_), None) => Some(newValue)

case (None, Some(\_)) => Some(oldValue.copy(cachedAt = Time.now))

}

}

}

/\*\*

\* Prefer do not cache entries if they're not expired. Otherwise uses fallbackPicker

\* @param fallBackPicker the picker to use when the oldvalue isn't do not cache or is expired.

\* Defaults to PreferNewestCache.

\*/

class PreferDoNotCache[V](

fallBackPicker: LockingCache.Picker[Cached[V]] = new PreferNewestCached[V]: PreferNewestCached[V],

statsReceiver: StatsReceiver)

extends LockingCache.Picker[Cached[V]] {

private[this] val pickDoNotCacheEntryCounter = statsReceiver.counter("pick\_do\_not\_cache\_entry")

private[this] val useFallbackCounter = statsReceiver.counter("use\_fallback")

override def apply(newValue: Cached[V], oldValue: Cached[V]): Option[Cached[V]] = {

if (oldValue.status == DoNotCache && oldValue.doNotCacheUntil.forall(

\_ > newValue.effectiveUpdateTime())) { // evaluates to true if dnc until is None

pickDoNotCacheEntryCounter.incr()

None

} else {

useFallbackCounter.incr()

fallBackPicker.apply(newValue, oldValue)

}

}

}

/\*\*

\* A Transformer of Cached values composed of a Transformer of the underlying values.

\*/

class CachedTransformer[A, B](underlying: Transformer[A, B])

extends Transformer[Cached[A], Cached[B]] {

def to(cachedA: Cached[A]) = cachedA.value match {

case None => Return(cachedA.copy(value = None))

case Some(a) =>

underlying.to(a) map { b =>

cachedA.copy(value = Some(b))

}

}

def from(cachedB: Cached[B]) = cachedB.value match {

case None => Return(cachedB.copy(value = None))

case Some(b) =>

underlying.from(b) map { a =>

cachedB.copy(value = Some(a))

}

}

}