package com.twitter.tweetypie.caching

import com.twitter.util.Duration

import com.twitter.util.Time

import scala.util.Random

import com.twitter.logging.Logger

/\*\*

\* Used to determine whether values successfully retrieved from cache

\* are [[CacheResult.Fresh]] or [[CacheResult.Stale]]. This is useful

\* in the implementation of a [[ValueSerializer]].

\*/

trait SoftTtl[-V] {

/\*\*

\* Determines whether a cached value was fresh.

\*

\* @param cachedAt the time at which the value was cached.

\*/

def isFresh(value: V, cachedAt: Time): Boolean

/\*\*

\* Wraps the value in Fresh or Stale depending on the value of `isFresh`.

\*

\* (The type variable U exists because it is not allowed to return

\* values of a contravariant type, so we must define a variable that

\* is a specific subclass of V. This is worth it because it allows

\* us to create polymorphic policies without having to specify the

\* type. Another solution would be to make the type invariant, but

\* then we would have to specify the type whenever we create an

\* instance.)

\*/

def toCacheResult[U <: V](value: U, cachedAt: Time): CacheResult[U] =

if (isFresh(value, cachedAt)) CacheResult.Fresh(value) else CacheResult.Stale(value)

}

object SoftTtl {

/\*\*

\* Regardless of the inputs, the value will always be considered

\* fresh.

\*/

object NeverRefresh extends SoftTtl[Any] {

override def isFresh(\_unusedValue: Any, \_unusedCachedAt: Time): Boolean = true

}

/\*\*

\* Trigger refresh based on the length of time that a value has been

\* stored in cache, ignoring the value.

\*

\* @param softTtl Items that were cached longer ago than this value

\* will be refreshed when they are accessed.

\*

\* @param jitter Add nondeterminism to the soft TTL to prevent a

\* thundering herd of requests refreshing the value at the same

\* time. The time at which the value is considered stale will be

\* uniformly spread out over a range of +/- (jitter/2). It is

\* valid to set the jitter to zero, which will turn off jittering.

\*

\* @param logger If non-null, use this logger rather than one based

\* on the class name. This logger is only used for trace-level

\* logging.

\*/

case class ByAge[V](

softTtl: Duration,

jitter: Duration,

specificLogger: Logger = null,

rng: Random = Random)

extends SoftTtl[Any] {

private[this] val logger: Logger =

if (specificLogger == null) Logger(getClass) else specificLogger

private[this] val maxJitterMs: Long = jitter.inMilliseconds

// this requirement is due to using Random.nextInt to choose the

// jitter, but it allows jitter of greater than 24 days

require(maxJitterMs <= (Int.MaxValue / 2))

// Negative jitter probably indicates misuse of the API

require(maxJitterMs >= 0)

// we want period +/- jitter, but the random generator

// generates non-negative numbers, so we generate [0, 2 \*

// maxJitter) and subtract maxJitter to obtain [-maxJitter,

// maxJitter)

private[this] val maxJitterRangeMs: Int = (maxJitterMs \* 2).toInt

// We perform all calculations in milliseconds, so convert the

// period to milliseconds out here.

private[this] val softTtlMs: Long = softTtl.inMilliseconds

// If the value is below this age, it will always be fresh,

// regardless of jitter.

private[this] val alwaysFreshAgeMs: Long = softTtlMs - maxJitterMs

// If the value is above this age, it will always be stale,

// regardless of jitter.

private[this] val alwaysStaleAgeMs: Long = softTtlMs + maxJitterMs

override def isFresh(value: Any, cachedAt: Time): Boolean = {

val ageMs: Long = (Time.now - cachedAt).inMilliseconds

val fresh =

if (ageMs <= alwaysFreshAgeMs) {

true

} else if (ageMs > alwaysStaleAgeMs) {

false

} else {

val jitterMs: Long = rng.nextInt(maxJitterRangeMs) - maxJitterMs

ageMs <= softTtlMs + jitterMs

}

logger.ifTrace(

s"Checked soft ttl: fresh = $fresh, " +

s"soft\_ttl\_ms = $softTtlMs, age\_ms = $ageMs, value = $value")

fresh

}

}

}