package com.twitter.servo.cache

import com.twitter.util.Future

/\*\*

\* Represents multiple underlying ReadCaches selected by key at invocation time.

\*/

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

/\*\* Retrieves the underlying cache for the given key. \*/

def underlyingCache(key: K): This

/\*\* Retrieves tuples of the underlying caches and the keys they apply to. \*/

def underlyingCacheForKeys(keys: Seq[K]): Seq[(This, Seq[K])]

/\*\* Retrieves all underlying caches. \*/

def underlyingCaches: Seq[This]

private[this] def collectUnderlying[V2](

keys: Seq[K]

)(

f: (This, Seq[K]) => Future[KeyValueResult[K, V2]]

): Future[KeyValueResult[K, V2]] = {

Future.collect(

underlyingCacheForKeys(keys) collect {

case (cacheForKey, keys) if !keys.isEmpty =>

f(cacheForKey, keys)

}

) map {

KeyValueResult.sum(\_)

}

}

override def get(keys: Seq[K]) = collectUnderlying(keys) { \_.get(\_) }

override def getWithChecksum(keys: Seq[K]) = collectUnderlying(keys) { \_.getWithChecksum(\_) }

override def release(): Unit = {

underlyingCaches foreach { \_.release() }

}

}

/\*\*

\* Represents multiple underlying Caches selected by key at invocation time.

\*/

trait SelectedCacheWrapper[K, V]

extends Cache[K, V]

with SelectedReadCacheWrapper[K, V, Cache[K, V]] {

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

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

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

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

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

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

}

/\*\*

\* GateSelectedCache implements SelectedCache to choose between two underlying

\* caches based on a function.

\*/

class SelectedCache[K, V](primary: Cache[K, V], secondary: Cache[K, V], usePrimary: K => Boolean)

extends SelectedCacheWrapper[K, V] {

override def underlyingCache(key: K) = if (usePrimary(key)) primary else secondary

override def underlyingCacheForKeys(keys: Seq[K]) = {

keys partition (usePrimary) match {

case (primaryKeys, secondaryKeys) => Seq((primary, primaryKeys), (secondary, secondaryKeys))

}

}

override def underlyingCaches = Seq(primary, secondary)

}

/\*\*

\* Factory for SelectedCache instances that use a simple function to migrate

\* users from a secondary cache (function returns false) to a primary cache

\* (function returns true). Serves a purpose similar to CacheFactory, but

\* cannot extend it due to type constraints.

\*

\* The function is expected to produce stable results by key over time to

\* prevent accessing stale cache entries due to keys flapping between the

\* two caches.

\*/

class SelectedCacheFactory[K](

primaryFactory: CacheFactory,

secondaryFactory: CacheFactory,

usePrimary: K => Boolean) {

def apply[V](serializer: Serializer[V], scopes: String\*): Cache[K, V] =

new SelectedCache(

primaryFactory[K, V](serializer, scopes: \_\*),

secondaryFactory[K, V](serializer, scopes: \_\*),

usePrimary

)

}