package com.twitter.servo.cache

import com.twitter.servo.util.Transformer

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

import scala.collection.mutable.ArrayBuffer

import scala.collection.{breakOut, mutable}

/\*\*

\* Adaptor from a ReadCache[K, V1] to an underlying ReadCache[K, V2]

\*

\* a Transformer is used to map between value types

\*/

class ValueTransformingReadCache[K, V1, V2](

underlyingCache: ReadCache[K, V2],

transformer: Transformer[V1, V2])

extends ReadCache[K, V1] {

// overridden to avoid mapping the unneeded keyMap

override def get(keys: Seq[K]): Future[KeyValueResult[K, V1]] = {

underlyingCache.get(keys) map { lr =>

// fold lr.found into found/deserialization failures

val found = mutable.Map.empty[K, V1]

val failed = mutable.Map.empty[K, Throwable]

lr.found foreach {

case (key, value) =>

transformer.from(value) match {

case Return(v) => found += key -> v

case Throw(t) => failed += key -> t

}

}

lr.copy(found = found.toMap, failed = lr.failed ++ failed.toMap)

} handle {

case t =>

KeyValueResult(failed = keys.map(\_ -> t).toMap)

}

}

// overridden to avoid mapping the unneeded keyMap

override def getWithChecksum(keys: Seq[K]): Future[CsKeyValueResult[K, V1]] = {

underlyingCache.getWithChecksum(keys) map { clr =>

clr.copy(found = clr.found map {

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

key -> (value flatMap { transformer.from(\_) }, checksum)

})

} handle {

case t =>

KeyValueResult(failed = keys.map(\_ -> t).toMap)

}

}

override def release() = underlyingCache.release()

}

/\*\*

\* Adaptor from a ReadCache[K, V1] to an underlying ReadCache[K2, V2]

\*

\* a Transformer is used to map between value types, and a

\* one-way mapping is used for keys, making it possible to

\* store data in the underlying cache using keys that can't

\* easily be reverse-mapped.

\*/

class KeyValueTransformingReadCache[K1, K2, V1, V2](

underlyingCache: ReadCache[K2, V2],

transformer: Transformer[V1, V2],

underlyingKey: K1 => K2)

extends ReadCache[K1, V1] {

// make keymapping for key recovery later

private[this] def mappedKeys(

keys: Seq[K1]

): (IndexedSeq[K2], Map[K2, K1]) = {

val k2s = new ArrayBuffer[K2](keys.size)

val k2k1s: Map[K2, K1] =

keys.map { key =>

val k2 = underlyingKey(key)

k2s += k2

k2 -> key

}(breakOut)

(k2s, k2k1s)

}

override def get(keys: Seq[K1]): Future[KeyValueResult[K1, V1]] = {

val (k2s, kMap) = mappedKeys(keys)

underlyingCache

.get(k2s)

.map { lr =>

// fold lr.found into found/deserialization failures

val found = Map.newBuilder[K1, V1]

val failed = Map.newBuilder[K1, Throwable]

lr.found.foreach {

case (key, value) =>

transformer.from(value) match {

case Return(v) => found += kMap(key) -> v

case Throw(t) => failed += kMap(key) -> t

}

}

lr.failed.foreach {

case (k, t) =>

failed += kMap(k) -> t

}

KeyValueResult(

found.result(),

lr.notFound.map { kMap(\_) },

failed.result()

)

}

.handle {

case t =>

KeyValueResult(failed = keys.map(\_ -> t).toMap)

}

}

override def getWithChecksum(keys: Seq[K1]): Future[CsKeyValueResult[K1, V1]] = {

val (k2s, kMap) = mappedKeys(keys)

underlyingCache

.getWithChecksum(k2s)

.map { clr =>

KeyValueResult(

clr.found.map {

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

kMap(key) -> (value.flatMap(transformer.from), checksum)

},

clr.notFound map { kMap(\_) },

clr.failed map {

case (key, t) =>

kMap(key) -> t

}

)

}

.handle {

case t =>

KeyValueResult(failed = keys.map(\_ -> t).toMap)

}

}

override def release(): Unit = underlyingCache.release()

}

class KeyTransformingCache[K1, K2, V](underlyingCache: Cache[K2, V], underlyingKey: K1 => K2)

extends KeyValueTransformingCache[K1, K2, V, V](

underlyingCache,

Transformer.identity,

underlyingKey

)

/\*\*

\* Adaptor from a Cache[K, V1] to an underlying Cache[K, V2]

\*

\* a Transformer is used to map between value types

\*/

class ValueTransformingCache[K, V1, V2](

underlyingCache: Cache[K, V2],

transformer: Transformer[V1, V2])

extends ValueTransformingReadCache[K, V1, V2](underlyingCache, transformer)

with Cache[K, V1] {

private[this] def to(v1: V1): Future[V2] = Future.const(transformer.to(v1))

override def add(key: K, value: V1): Future[Boolean] =

to(value) flatMap { underlyingCache.add(key, \_) }

override def checkAndSet(key: K, value: V1, checksum: Checksum): Future[Boolean] =

to(value) flatMap { underlyingCache.checkAndSet(key, \_, checksum) }

override def set(key: K, value: V1): Future[Unit] =

to(value) flatMap { underlyingCache.set(key, \_) }

override def replace(key: K, value: V1): Future[Boolean] =

to(value) flatMap { underlyingCache.replace(key, \_) }

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

underlyingCache.delete(key)

}

/\*\*

\* Adaptor from a Cache[K1, V1] to an underlying Cache[K2, V2]

\*

\* a Transformer is used to map between value types, and a

\* one-way mapping is used for keys, making it possible to

\* store data in the underlying cache using keys that can't

\* easily be reverse-mapped.

\*/

class KeyValueTransformingCache[K1, K2, V1, V2](

underlyingCache: Cache[K2, V2],

transformer: Transformer[V1, V2],

underlyingKey: K1 => K2)

extends KeyValueTransformingReadCache[K1, K2, V1, V2](

underlyingCache,

transformer,

underlyingKey

)

with Cache[K1, V1] {

private[this] def to(v1: V1): Future[V2] = Future.const(transformer.to(v1))

override def add(key: K1, value: V1): Future[Boolean] =

to(value) flatMap { underlyingCache.add(underlyingKey(key), \_) }

override def checkAndSet(key: K1, value: V1, checksum: Checksum): Future[Boolean] =

to(value) flatMap { underlyingCache.checkAndSet(underlyingKey(key), \_, checksum) }

override def set(key: K1, value: V1): Future[Unit] =

to(value) flatMap { underlyingCache.set(underlyingKey(key), \_) }

override def replace(key: K1, value: V1): Future[Boolean] =

to(value) flatMap { underlyingCache.replace(underlyingKey(key), \_) }

override def delete(key: K1): Future[Boolean] =

underlyingCache.delete(underlyingKey(key))

}

/\*\*

\* Adaptor from a TtlCache[K, V1] to an underlying TtlCache[K, V2]

\*

\* a Transformer is used to map between value types

\*/

class ValueTransformingTtlCache[K, V1, V2](

underlyingCache: TtlCache[K, V2],

transformer: Transformer[V1, V2])

extends ValueTransformingReadCache[K, V1, V2](underlyingCache, transformer)

with TtlCache[K, V1] {

private[this] def to(v1: V1): Future[V2] = Future.const(transformer.to(v1))

override def add(key: K, value: V1, ttl: Duration): Future[Boolean] =

to(value) flatMap { underlyingCache.add(key, \_, ttl) }

override def checkAndSet(

key: K,

value: V1,

checksum: Checksum,

ttl: Duration

): Future[Boolean] =

to(value) flatMap { underlyingCache.checkAndSet(key, \_, checksum, ttl) }

override def set(key: K, value: V1, ttl: Duration): Future[Unit] =

to(value) flatMap { underlyingCache.set(key, \_, ttl) }

override def replace(key: K, value: V1, ttl: Duration): Future[Boolean] =

to(value) flatMap { underlyingCache.replace(key, \_, ttl) }

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

underlyingCache.delete(key)

}

/\*\*

\* Adaptor from a TtlCache[K1, V1] to an underlying TtlCache[K2, V2]

\*

\* a Transformer is used to map between value types, and a

\* one-way mapping is used for keys, making it possible to

\* store data in the underlying cache using keys that can't

\* easily be reverse-mapped.

\*/

class KeyValueTransformingTtlCache[K1, K2, V1, V2](

underlyingCache: TtlCache[K2, V2],

transformer: Transformer[V1, V2],

underlyingKey: K1 => K2)

extends KeyValueTransformingReadCache[K1, K2, V1, V2](

underlyingCache,

transformer,

underlyingKey

)

with TtlCache[K1, V1] {

private[this] def to(v1: V1): Future[V2] = Future.const(transformer.to(v1))

override def add(key: K1, value: V1, ttl: Duration): Future[Boolean] =

to(value) flatMap { underlyingCache.add(underlyingKey(key), \_, ttl) }

override def checkAndSet(

key: K1,

value: V1,

checksum: Checksum,

ttl: Duration

): Future[Boolean] =

to(value) flatMap { underlyingCache.checkAndSet(underlyingKey(key), \_, checksum, ttl) }

override def set(key: K1, value: V1, ttl: Duration): Future[Unit] =

to(value) flatMap { underlyingCache.set(underlyingKey(key), \_, ttl) }

override def replace(key: K1, value: V1, ttl: Duration): Future[Boolean] =

to(value) flatMap { underlyingCache.replace(underlyingKey(key), \_, ttl) }

override def delete(key: K1): Future[Boolean] =

underlyingCache.delete(underlyingKey(key))

}

class KeyTransformingTtlCache[K1, K2, V](underlyingCache: TtlCache[K2, V], underlyingKey: K1 => K2)

extends KeyValueTransformingTtlCache[K1, K2, V, V](

underlyingCache,

Transformer.identity,

underlyingKey

)

class KeyTransformingLockingCache[K1, K2, V](

underlyingCache: LockingCache[K2, V],

underlyingKey: K1 => K2)

extends KeyValueTransformingCache[K1, K2, V, V](

underlyingCache,

Transformer.identity,

underlyingKey

)

with LockingCache[K1, V] {

import LockingCache.\_

override def lockAndSet(key: K1, handler: Handler[V]): Future[Option[V]] =

underlyingCache.lockAndSet(underlyingKey(key), handler)

}

class KeyTransformingCounterCache[K1, K2](

underlyingCache: CounterCache[K2],

underlyingKey: K1 => K2)

extends KeyTransformingCache[K1, K2, Long](underlyingCache, underlyingKey)

with CounterCache[K1] {

override def incr(key: K1, delta: Int = 1): Future[Option[Long]] = {

underlyingCache.incr(underlyingKey(key), delta)

}

override def decr(key: K1, delta: Int = 1): Future[Option[Long]] = {

underlyingCache.decr(underlyingKey(key), delta)

}

}