package com.twitter.servo.cache

import com.twitter.util.Duration

import scala.collection.mutable

/\*\*

\* Used to produce differently-typed caches with the same configuration

\* and potentially with shared observation.

\*/

trait CacheFactory {

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

}

/\*\*

\* Builds an instance of NullCache.

\*/

object NullCacheFactory extends CacheFactory {

val cache = new NullCache[Nothing, Nothing]

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

cache.asInstanceOf[NullCache[K, V]]

}

/\*\*

\* Builds DeciderableCaches, which proxy to one of two caches built from the

\* argument CacheFactories depending on a decider value.

\*/

case class DeciderableCacheFactory(

primaryCacheFactory: CacheFactory,

secondaryCacheFactory: CacheFactory,

isAvailable: () => Boolean)

extends CacheFactory {

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

new DeciderableCache(

primaryCacheFactory(serializer, scopes: \_\*),

secondaryCacheFactory(serializer, scopes: \_\*),

isAvailable()

)

}

/\*\*

\* Builds MigratingCaches, which support gradual migrations from one cache

\* to another. See MigratingCache.scala for details.

\*/

case class MigratingCacheFactory(cacheFactory: CacheFactory, darkCacheFactory: CacheFactory)

extends CacheFactory {

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

new MigratingCache(

cacheFactory(serializer, scopes: \_\*),

darkCacheFactory(serializer, scopes: \_\*)

)

}

case class ObservableCacheFactory(cacheFactory: CacheFactory, cacheObserver: CacheObserver)

extends CacheFactory {

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

new ObservableCache(cacheFactory(serializer), cacheObserver.scope(scopes: \_\*))

}

/\*\*

\* Builds in-memory caches with elements that never expire.

\*/

case class MutableMapCacheFactory(

serialize: Boolean = false,

useSharedCache: Boolean = false,

keyTransformerFactory: KeyTransformerFactory = ToStringKeyTransformerFactory)

extends CacheFactory {

lazy val sharedCache = mkCache

def mkCache = {

new MutableMapCache[Object, Object](new mutable.HashMap)

}

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

val cache = if (useSharedCache) sharedCache else mkCache

if (serialize) {

new KeyValueTransformingCache(

cache.asInstanceOf[Cache[String, Array[Byte]]],

serializer,

keyTransformerFactory()

)

} else {

cache.asInstanceOf[Cache[K, V]]

}

}

}

/\*\*

\* Builds in-memory caches with TTL'd entries and LRU eviction policies.

\*/

case class InProcessLruCacheFactory(

ttl: Duration,

lruSize: Int,

serialize: Boolean = false,

useSharedCache: Boolean = false,

keyTransformerFactory: KeyTransformerFactory = ToStringKeyTransformerFactory)

extends CacheFactory {

def mkCache = new ExpiringLruCache[Object, Object](ttl, lruSize)

lazy val sharedCache = mkCache

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

val cache = if (useSharedCache) sharedCache else mkCache

if (serialize) {

new KeyValueTransformingCache(

cache.asInstanceOf[Cache[String, Array[Byte]]],

serializer,

keyTransformerFactory()

)

} else {

cache.asInstanceOf[Cache[K, V]]

}

}

}

/\*\*

\* Builds MemcacheCaches, which applies serialization, key-transformation,

\* and TTL mechanics to an underlying Memcache.

\*/

case class MemcacheCacheFactory(

memcache: Memcache,

ttl: Duration,

keyTransformerFactory: KeyTransformerFactory = ToStringKeyTransformerFactory)

extends CacheFactory {

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

new MemcacheCache(memcache, ttl, serializer, keyTransformerFactory[K]())

}

/\*\*

\* Builds KeyTransformers, which are required for constructing

\* KeyValueTransformingCaches.

\*/

trait KeyTransformerFactory {

def apply[K](): KeyTransformer[K]

}

/\*\*

\* Builds KeyTransformers by simply call the keys' toString methods.

\*/

object ToStringKeyTransformerFactory extends KeyTransformerFactory {

def apply[K]() = new ToStringKeyTransformer[K]()

}

/\*\*

\* Builds KeyTransformers that prefix all keys generated by an underlying

\* transformer with a string.

\*/

case class PrefixKeyTransformerFactory(

prefix: String,

delimiter: String = constants.Colon,

underlying: KeyTransformerFactory = ToStringKeyTransformerFactory)

extends KeyTransformerFactory {

def apply[K]() = new PrefixKeyTransformer[K](prefix, delimiter, underlying[K]())

}