package com.twitter.servo.keyvalue

import com.twitter.finagle.memcached.util.NotFound

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

import scala.collection.immutable

object KeyValueResult {

private[this] val Empty = KeyValueResult()

private[this] val EmptyFuture = Future.value(Empty)

def empty[K, V]: KeyValueResult[K, V] =

Empty.asInstanceOf[KeyValueResult[K, V]]

def emptyFuture[K, V]: Future[KeyValueResult[K, V]] =

EmptyFuture.asInstanceOf[Future[KeyValueResult[K, V]]]

/\*\*

\* Builds a KeyValueResult using pairs of keys to Try[Option[V]]. These values are split

\* out to build the separate found/notFound/failed collections.

\*/

def build[K, V](data: (K, Try[Option[V]])\*): KeyValueResult[K, V] = {

val bldr = new KeyValueResultBuilder[K, V]

data.foreach { case (k, v) => bldr.update(k, v) }

bldr.result()

}

/\*\*

\* Builds a future KeyValueResult using a future sequence of key-value tuples. That

\* sequence does not necessarily match up with the sequence of keys provided. The

\* sequence of pairs represent the found results. notFound will be filled in from the

\* missing keys.

\*/

def fromPairs[K, V](

keys: Iterable[K] = Nil: immutable.Nil.type

)(

futurePairs: Future[TraversableOnce[(K, V)]]

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

fromMap(keys) {

futurePairs map { \_.toMap }

}

}

/\*\*

\* Builds a future KeyValueResult using a future map of found results. notFound will be filled

\* in from the missing keys.

\*/

def fromMap[K, V](

keys: Iterable[K] = Nil: immutable.Nil.type

)(

futureMap: Future[Map[K, V]]

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

futureMap map { found =>

KeyValueResult[K, V](found = found, notFound = NotFound(keys.toSet, found.keySet))

} handle {

case t =>

KeyValueResult[K, V](failed = keys.map { \_ -> t }.toMap)

}

}

/\*\*

\* Builds a future KeyValueResult using a future sequence of optional results. That

\* sequence must match up pair-wise with the given sequence of keys. A value of Some[V] is

\* counted as a found result, a value of None is counted as a notFound result.

\*/

def fromSeqOption[K, V](

keys: Iterable[K]

)(

futureSeq: Future[Seq[Option[V]]]

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

futureSeq map { seq =>

keys.zip(seq).foldLeft(new KeyValueResultBuilder[K, V]) {

case (bldr, (key, tryRes)) =>

tryRes match {

case Some(value) => bldr.addFound(key, value)

case None => bldr.addNotFound(key)

}

} result ()

} handle {

case t =>

KeyValueResult[K, V](failed = keys.map { \_ -> t }.toMap)

}

}

/\*\*

\* Builds a future KeyValueResult using a future sequence of Try results. That

\* sequence must match up pair-wise with the given sequence of keys. A value of Return[V] is

\* counted as a found result, a value of Throw is counted as a failed result.

\*/

def fromSeqTry[K, V](

keys: Iterable[K]

)(

futureSeq: Future[Seq[Try[V]]]

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

futureSeq map { seq =>

keys.zip(seq).foldLeft(new KeyValueResultBuilder[K, V]) {

case (bldr, (key, tryRes)) =>

tryRes match {

case Return(value) => bldr.addFound(key, value)

case Throw(t) => bldr.addFailed(key, t)

}

} result ()

} handle {

case t =>

KeyValueResult[K, V](failed = keys.map { \_ -> t }.toMap)

}

}

/\*\*

\* Builds a future KeyValueResult using a sequence of future options. That sequence must

\* match up pair-wise with the given sequence of keys. A value of Some[V] is

\* counted as a found result, a value of None is counted as a notFound result.

\*/

def fromSeqFuture[K, V](

keys: Iterable[K]

)(

futureSeq: Seq[Future[Option[V]]]

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

fromSeqTryOptions(keys) {

Future.collect {

futureSeq map { \_.transform(Future(\_)) }

}

}

}

/\*\*

\* Builds a future KeyValueResult using a future sequence of Try[Option[V]]. That sequence must

\* match up pair-wise with the given sequence of keys. A value of Return[Some[V]] is

\* counted as a found result, a value of Return[None] is counted as a notFound result, and a value

\* of Throw[V] is counted as a failed result.

\*/

def fromSeqTryOptions[K, V](

keys: Iterable[K]

)(

futureSeq: Future[Seq[Try[Option[V]]]]

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

futureSeq map { seq =>

keys.zip(seq).foldLeft(new KeyValueResultBuilder[K, V]) {

case (bldr, (key, tryRes)) =>

tryRes match {

case Return(Some(value)) => bldr.addFound(key, value)

case Return(None) => bldr.addNotFound(key)

case Throw(t) => bldr.addFailed(key, t)

}

} result ()

} handle {

case t =>

KeyValueResult[K, V](failed = keys.map { \_ -> t }.toMap)

}

}

/\*\*

\* Builds a future KeyValueResult using a future map with value Try[Option[V]]. A value of

\* Return[Some[V]] is counted as a found result, a value of Return[None] is counted as a notFound

\* result, and a value of Throw[V] is counted as a failed result.

\*

\* notFound will be filled in from the missing keys. Exceptions will be handled by counting all

\* keys as failed. Values that are in map but not keys will be ignored.

\*/

def fromMapTryOptions[K, V](

keys: Iterable[K]

)(

futureMapTryOptions: Future[Map[K, Try[Option[V]]]]

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

futureMapTryOptions map { mapTryOptions =>

keys.foldLeft(new KeyValueResultBuilder[K, V]) {

case (builder, key) =>

mapTryOptions.get(key) match {

case Some(Return(Some(value))) => builder.addFound(key, value)

case Some(Return(None)) | None => builder.addNotFound(key)

case Some(Throw(failure)) => builder.addFailed(key, failure)

}

} result ()

} handle {

case t =>

KeyValueResult[K, V](failed = keys.map { \_ -> t }.toMap)

}

}

/\*\*

\* Reduces several KeyValueResults down to just 1, by combining as if by ++, but

\* more efficiently with fewer intermediate results.

\*/

def sum[K, V](results: Iterable[KeyValueResult[K, V]]): KeyValueResult[K, V] = {

val bldr = new KeyValueResultBuilder[K, V]

results foreach { result =>

bldr.addFound(result.found)

bldr.addNotFound(result.notFound)

bldr.addFailed(result.failed)

}

val res = bldr.result()

if (res.notFound.isEmpty && res.failed.isEmpty) {

res

} else {

val foundKeySet = res.found.keySet

val notFound = NotFound(res.notFound, foundKeySet)

val failed = NotFound(NotFound(res.failed, foundKeySet), res.notFound)

KeyValueResult(res.found, notFound, failed)

}

}

}

case class KeyValueResult[K, +V](

found: Map[K, V] = Map.empty[K, V]: immutable.Map[K, V],

notFound: Set[K] = Set.empty[K]: immutable.Set[K],

failed: Map[K, Throwable] = Map.empty[K, Throwable]: immutable.Map[K, Throwable])

extends Iterable[(K, Try[Option[V]])] {

/\*\*

\* A cheaper implementation of isEmpty than the default which relies

\* on building an iterator.

\*/

override def isEmpty = found.isEmpty && notFound.isEmpty && failed.isEmpty

/\*\*

\* map over the keyspace to produce a new KeyValueResult

\*/

def mapKeys[K2](f: K => K2): KeyValueResult[K2, V] =

copy(

found = found.map { case (k, v) => f(k) -> v },

notFound = notFound.map(f),

failed = failed.map { case (k, t) => f(k) -> t }

)

/\*\*

\* Maps over found values to produce a new KeyValueResult. If the given function throws an

\* exception for a particular value, that value will be moved to the `failed` bucket with

\* the thrown exception.

\*/

def mapFound[V2](f: V => V2): KeyValueResult[K, V2] = {

val builder = new KeyValueResultBuilder[K, V2]()

found.foreach {

case (k, v) =>

builder.update(k, Try(Some(f(v))))

}

builder.addNotFound(notFound)

builder.addFailed(failed)

builder.result()

}

/\*\*

\* map over the values provided by the iterator, to produce a new KeyValueResult

\*/

def mapValues[V2](f: Try[Option[V]] => Try[Option[V2]]): KeyValueResult[K, V2] = {

val builder = new KeyValueResultBuilder[K, V2]()

found.foreach {

case (k, v) =>

builder.update(k, f(Return(Some(v))))

}

notFound.foreach { k =>

builder.update(k, f(Return.None))

}

failed.foreach {

case (k, t) =>

builder.update(k, f(Throw(t)))

}

builder.result()

}

/\*\*

\* Map over found values to create a new KVR with the existing notFound and failed keys intact.

\*/

def mapFoundValues[V2](f: V => Try[Option[V2]]): KeyValueResult[K, V2] = {

val builder = new KeyValueResultBuilder[K, V2]()

found.foreach {

case (k, v) => builder.update(k, f(v))

}

builder.addNotFound(notFound)

builder.addFailed(failed)

builder.result()

}

/\*\*

\* map over the pairs of results, creating a new KeyValueResult based on the returned

\* tuples from the provided function.

\*/

def mapPairs[K2, V2](f: (K, Try[Option[V]]) => (K2, Try[Option[V2]])): KeyValueResult[K2, V2] = {

val builder = new KeyValueResultBuilder[K2, V2]

def update(k: K, v: Try[Option[V]]): Unit =

f(k, v) match {

case (k2, v2) => builder.update(k2, v2)

}

found.foreach {

case (k, v) =>

update(k, Return(Some(v)))

}

notFound.foreach { k =>

update(k, Return.None)

}

failed.foreach {

case (k, t) =>

update(k, Throw(t))

}

builder.result()

}

/\*\*

\* filter the KeyValueResult, to produce a new KeyValueResult

\*/

override def filter(p: ((K, Try[Option[V]])) => Boolean): KeyValueResult[K, V] = {

val builder = new KeyValueResultBuilder[K, V]

def update(k: K, v: Try[Option[V]]): Unit = {

if (p((k, v)))

builder.update(k, v)

}

found.foreach {

case (k, v) =>

update(k, Return(Some(v)))

}

notFound.foreach { k =>

update(k, Return.None)

}

failed.foreach {

case (k, t) =>

update(k, Throw(t))

}

builder.result()

}

/\*\*

\* filterNot the KeyValueResult, to produce a new KeyValueResult

\*/

override def filterNot(p: ((K, Try[Option[V]])) => Boolean): KeyValueResult[K, V] = {

filter(!p(\_))

}

/\*\*

\* Returns an Iterator that yields all found, notFound, and failed values

\* represented in the combined Try[Option[V]] type.

\*/

def iterator: Iterator[(K, Try[Option[V]])] =

(found.iterator map { case (k, v) => k -> Return(Some(v)) }) ++

(notFound.iterator map { k =>

k -> Return.None

}) ++

(failed.iterator map { case (k, t) => k -> Throw(t) })

/\*\*

\* Returns a copy in which all failed entries are converted to misses. The specific

\* failure information is lost.

\*/

def convertFailedToNotFound =

copy(

notFound = notFound ++ failed.keySet,

failed = Map.empty[K, Throwable]

)

/\*\*

\* Returns a copy in which all not-found entries are converted to failures.

\*/

def convertNotFoundToFailed(f: K => Throwable) =

copy(

notFound = Set.empty[K],

failed = failed ++ (notFound map { k =>

k -> f(k)

})

)

/\*\*

\* Returns a copy in which failures are repaired with the supplied handler

\*/

def repairFailed[V2 >: V](handler: PartialFunction[Throwable, Option[V2]]) =

if (failed.isEmpty) {

this

} else {

val builder = new KeyValueResultBuilder[K, V2]

builder.addFound(found)

builder.addNotFound(notFound)

failed map { case (k, t) => builder.update(k, Throw(t) handle handler) }

builder.result()

}

/\*\*

\* Combines two KeyValueResults. Conflicting founds/notFounds are resolved

\* as founds, and conflicting (found|notFound)/failures are resolved as (found|notFound).

\*/

def ++[K2 >: K, V2 >: V](that: KeyValueResult[K2, V2]): KeyValueResult[K2, V2] = {

if (this.isEmpty) that

else if (that.isEmpty) this.asInstanceOf[KeyValueResult[K2, V2]]

else {

val found = this.found ++ that.found

val notFound = NotFound(this.notFound ++ that.notFound, found.keySet)

val failed = NotFound(NotFound(this.failed ++ that.failed, found.keySet), notFound)

KeyValueResult(found, notFound, failed)

}

}

/\*\*

\* Looks up a result for a key.

\*/

def apply(key: K): Try[Option[V]] = {

found.get(key) match {

case some @ Some(\_) => Return(some)

case None =>

failed.get(key) match {

case Some(t) => Throw(t)

case None => Return.None

}

}

}

/\*\*

\* Looks up a result for a key, returning a provided default if the key is not

\* found or failed.

\*/

def getOrElse[V2 >: V](key: K, default: => V2): V2 =

found.getOrElse(key, default)

/\*\*

\* If any keys fail, will return the first failure. Otherwise,

\* will convert founds/notFounds to a Seq[Option[V]], ordered by

\* the keys provided

\*/

def toFutureSeqOfOptions(keys: Seq[K]): Future[Seq[Option[V]]] = {

failed.values.headOption match {

case Some(t) => Future.exception(t)

case None => Future.value(keys.map(found.get))

}

}

// This is unfortunate, but we end up pulling in Iterable's toString,

// which is not all that readable.

override def toString(): String = {

val sb = new StringBuilder(256)

sb.append("KeyValueResult(")

sb.append("found = ")

sb.append(found)

sb.append(", notFound = ")

sb.append(notFound)

sb.append(", failed = ")

sb.append(failed)

sb.append(')')

sb.toString()

}

}

class KeyValueResultBuilder[K, V] {

private[this] val found = Map.newBuilder[K, V]

private[this] val notFound = Set.newBuilder[K]

private[this] val failed = Map.newBuilder[K, Throwable]

def addFound(k: K, v: V) = { found += (k -> v); this }

def addNotFound(k: K) = { notFound += k; this }

def addFailed(k: K, t: Throwable) = { failed += (k -> t); this }

def addFound(kvs: Iterable[(K, V)]) = { found ++= kvs; this }

def addNotFound(ks: Iterable[K]) = { notFound ++= ks; this }

def addFailed(kts: Iterable[(K, Throwable)]) = { failed ++= kts; this }

def update(k: K, tryV: Try[Option[V]]) = {

tryV match {

case Throw(t) => addFailed(k, t)

case Return(None) => addNotFound(k)

case Return(Some(v)) => addFound(k, v)

}

}

def result() = KeyValueResult(found.result(), notFound.result(), failed.result())

}