package com.twitter.servo.util

import com.twitter.logging.Logger

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

import java.util.concurrent.CancellationException

import scala.collection.mutable.ArrayBuffer

@deprecated("Use `Future.batched`", "2.6.1")

trait BatchExecutorFactory {

def apply[In, Out](f: Seq[In] => Future[Seq[Out]]): BatchExecutor[In, Out]

}

/\*\*

\* A BatchExecutorFactory allows you to specify the criteria in which a batch

\* should be flushed prior to constructing a BatchExecutor. A BatchExecutor asks for a

\* function that takes a Seq[In] and returns a Future[Seq[Out]], in return it gives you

\* a `In => Future[Out]` interface so that you can incrementally submit tasks to be

\* performed when the criteria for batch flushing is met.

\*

\* Examples:

\* val batcherFactory = BatchExecutorFactory(sizeThreshold = 10)

\* def processBatch(reqs: Seq[Request]): Future[Seq[Response]]

\* val batcher = batcherFactory(processBatch)

\*

\* val response: Future[Response] = batcher(new Request)

\*

\* the batcher will wait until 10 requests have been submitted, then delegate

\* to the processBatch method to compute the responses.

\*

\* you can also construct a BatchExecutor that has a time-based threshold or both:

\* val batcherFactory = BatchExecutorFactory(

\* sizeThreshold = 10, timeThreshold = 10.milliseconds, timer = new JavaTimer(true))

\*

\* A batcher's size can be controlled at runtime through a bufSizeFraction function

\* that should return a float between 0.0 and 1.0 that represents the fractional size

\* of the sizeThreshold that should be used for the next batch to be collected.

\*

\*/

@deprecated("Use `Future.batched`", "2.6.1")

object BatchExecutorFactory {

final val DefaultBufSizeFraction = 1.0f

lazy val instant = sized(1)

def sized(sizeThreshold: Int): BatchExecutorFactory = new BatchExecutorFactory {

override def apply[In, Out](f: Seq[In] => Future[Seq[Out]]) = {

new BatchExecutor(sizeThreshold, None, f, DefaultBufSizeFraction)

}

}

def timed(timeThreshold: Duration, timer: Timer): BatchExecutorFactory =

sizedAndTimed(Int.MaxValue, timeThreshold, timer)

def sizedAndTimed(

sizeThreshold: Int,

timeThreshold: Duration,

timer: Timer

): BatchExecutorFactory =

dynamicSizedAndTimed(sizeThreshold, timeThreshold, timer, DefaultBufSizeFraction)

def dynamicSizedAndTimed(

sizeThreshold: Int,

timeThreshold: Duration,

timer: Timer,

bufSizeFraction: => Float

): BatchExecutorFactory = new BatchExecutorFactory {

override def apply[In, Out](f: (Seq[In]) => Future[Seq[Out]]) = {

new BatchExecutor(sizeThreshold, Some(timeThreshold, timer), f, bufSizeFraction)

}

}

}

@deprecated("Use `Future.batched`", "2.6.1")

class BatchExecutor[In, Out] private[util] (

maxSizeThreshold: Int,

timeThreshold: Option[(Duration, Timer)],

f: Seq[In] => Future[Seq[Out]],

bufSizeFraction: => Float) { batcher =>

private[this] class ScheduledFlush(after: Duration, timer: Timer) {

@volatile private[this] var cancelled = false

private[this] val task = timer.schedule(after.fromNow) { flush() }

def cancel(): Unit = {

cancelled = true

task.cancel()

}

private[this] def flush(): Unit = {

val doAfter = batcher.synchronized {

if (!cancelled) {

flushBatch()

} else { () =>

()

}

}

doAfter()

}

}

private[this] val log = Logger.get("BatchExecutor")

// operations on these are synchronized on `this`

private[this] val buf = new ArrayBuffer[(In, Promise[Out])](maxSizeThreshold)

private[this] var scheduled: Option[ScheduledFlush] = None

private[this] var currentBufThreshold = newBufThreshold

private[this] def shouldSchedule = timeThreshold.isDefined && scheduled.isEmpty

private[this] def currentBufFraction = {

val fract = bufSizeFraction

if (fract > 1.0f) {

log.warning(

"value returned for BatchExecutor.bufSizeFraction (%f) was > 1.0f, using 1.0",

fract

)

1.0f

} else if (fract < 0.0f) {

log.warning(

"value returned for BatchExecutor.bufSizeFraction (%f) was negative, using 0.0f",

fract

)

0.0f

} else {

fract

}

}

private[this] def newBufThreshold = {

val size: Int = math.round(currentBufFraction \* maxSizeThreshold)

if (size < 1) {

1

} else if (size >= maxSizeThreshold) {

maxSizeThreshold

} else {

size

}

}

def apply(t: In): Future[Out] = {

enqueue(t)

}

private[this] def enqueue(t: In): Future[Out] = {

val promise = new Promise[Out]

val doAfter = synchronized {

buf.append((t, promise))

if (buf.size >= currentBufThreshold) {

flushBatch()

} else {

scheduleFlushIfNecessary()

() => ()

}

}

doAfter()

promise

}

private[this] def scheduleFlushIfNecessary(): Unit = {

timeThreshold foreach {

case (duration, timer) =>

if (shouldSchedule) {

scheduled = Some(new ScheduledFlush(duration, timer))

}

}

}

private[this] def flushBatch(): () => Unit = {

// this must be executed within a synchronize block

val prevBatch = new ArrayBuffer[(In, Promise[Out])](buf.length)

buf.copyToBuffer(prevBatch)

buf.clear()

scheduled foreach { \_.cancel() }

scheduled = None

currentBufThreshold = newBufThreshold // set the next batch's size

() =>

try {

executeBatch(prevBatch)

} catch {

case e: Throwable =>

log.warning(e, "unhandled exception caught in BatchExecutor: %s", e.toString)

}

}

private[this] def executeBatch(batch: Seq[(In, Promise[Out])]): Unit = {

val uncancelled = batch filter {

case (in, p) =>

p.isInterrupted match {

case Some(\_cause) =>

p.setException(new CancellationException)

false

case None => true

}

}

val ins = uncancelled map { case (in, \_) => in }

// N.B. intentionally not linking cancellation of these promises to the execution of the batch

// because it seems that in most cases you would be canceling mostly uncanceled work for an

// outlier.

val promises = uncancelled map { case (\_, promise) => promise }

f(ins) respond {

case Return(outs) =>

(outs zip promises) foreach {

case (out, p) =>

p() = Return(out)

}

case Throw(e) =>

val t = Throw(e)

promises foreach { \_() = t }

}

}

}