package com.twitter.tweetypie

package backends

import com.twitter.concurrent.AsyncSemaphore

import com.twitter.util.Timer

import com.twitter.util.Promise

import scala.util.control.NoStackTrace

/\*\*

\* Tools for building warmup actions on backend clients. The basic

\* idea is to make requests to backends repeatedly until they succeed.

\*/

object Warmup {

/\*\*

\* Signals that a warmup action was aborted because warmup is

\* complete.

\*/

object WarmupComplete extends Exception with NoStackTrace

/\*\*

\* Configuration for warmup actions.

\*

\* @param maxOutstandingRequests: Limit on total number of outstanding warmup requests.

\* @param maxWarmupDuration: Total amount of time warmup is allowed to take.

\* @param requestTimeouts: Time limit for individual warmup actions.

\* @param reliability: Criteria for how many times each warmup should be run.

\*/

case class Settings(

maxOutstandingRequests: Int,

maxWarmupDuration: Duration,

requestTimeouts: Map[String, Duration],

reliability: Reliably) {

def toRunner(logger: Logger, timer: Timer): Runner =

new WithTimeouts(requestTimeouts, timer)

.within(new Logged(logger))

.within(new LimitedConcurrency(maxOutstandingRequests))

.within(reliability)

def apply[A: Warmup](value: A, logger: Logger, timer: Timer): Future[Unit] =

toRunner(logger, timer)

.run(value)

.raiseWithin(maxWarmupDuration)(timer)

.handle { case \_ => }

}

/\*\*

\* Strategy for running Warmup actions.

\*/

trait Runner { self =>

/\*\*

\* Run one single warmup action.

\*/

def runOne(name: String, action: => Future[Unit]): Future[Unit]

/\*\*

\* Compose these two Runners by calling this Runner's runOne

\* inside of other's runOne.

\*/

final def within(other: Runner): Runner =

new Runner {

override def runOne(name: String, action: => Future[Unit]): Future[Unit] =

other.runOne(name, self.runOne(name, action))

}

/\*\*

\* Execute all of the warmup actions for the given value using

\* this runner.

\*/

final def run[T](t: T)(implicit w: Warmup[T]): Future[Unit] =

Future.join(w.actions.toSeq.map { case (name, f) => runOne(name, f(t).unit) })

}

/\*\*

\* Set a ceiling on the amount of time each kind of warmup action is

\* allowed to take.

\*/

class WithTimeouts(timeouts: Map[String, Duration], timer: Timer) extends Runner {

override def runOne(name: String, action: => Future[Unit]): Future[Unit] =

timeouts.get(name).map(action.raiseWithin(\_)(timer)).getOrElse(action)

}

/\*\*

\* Execute each action until its reliability is estimated to be

\* above the given threshold. The reliability is initially assumed

\* to be zero. The reliability is estimated as an exponential moving

\* average, with the new data point given the appropriate weight so

\* that a single data point will no longer be able to push the

\* average below the threshold.

\*

\* The warmup action is considered successful if it does not throw

\* an exception. No timeouts are applied.

\*

\* The threshold must be in the interval [0, 1).

\*

\* The concurrency level determines how many outstanding requests

\* to maintain until the threshold is reached. This allows warmup

\* to happen more rapidly when individual requests have high

\* latency.

\*

\* maxAttempts limits the total number of tries that we are allowed

\* to try to reach the reliability threshold. This is a safety

\* measure to prevent overloading whatever subsystem we are

\* attempting to warm up.

\*/

case class Reliably(reliabilityThreshold: Double, concurrency: Int, maxAttempts: Int)

extends Runner {

require(reliabilityThreshold < 1)

require(reliabilityThreshold >= 0)

require(concurrency > 0)

require(maxAttempts > 0)

// Find the weight at which one failure will not push us under the

// reliabilityThreshold.

val weight: Double = 1 - math.pow(

1 - reliabilityThreshold,

(1 - reliabilityThreshold) / reliabilityThreshold

)

// Make sure that rounding error did not cause weight to become zero.

require(weight > 0)

require(weight <= 1)

// On each iteration, we discount the current reliability by this

// factor before adding in the new reliability data point.

val decay: Double = 1 - weight

// Make sure that rounding error did not cause decay to be zero.

require(decay < 1)

override def runOne(name: String, action: => Future[Unit]): Future[Unit] = {

def go(attempts: Int, reliability: Double, outstanding: Seq[Future[Unit]]): Future[Unit] =

if (reliability >= reliabilityThreshold || (attempts == 0 && outstanding.isEmpty)) {

// We hit the threshold or ran out of tries. Don't cancel any

// outstanding requests, just wait for them all to complete.

Future.join(outstanding.map(\_.handle { case \_ => }))

} else if (attempts > 0 && outstanding.length < concurrency) {

// We have not yet hit the reliability threshold, and we

// still have available concurrency, so make a new request.

go(attempts - 1, reliability, action +: outstanding)

} else {

val sel = Future.select(outstanding)

// We need this promise wrapper because if the select is

// interrupted, it relays the interrupt to the outstanding

// requests but does not itself return with a

// failure. Wrapping in a promise lets us differentiate

// between an interrupt coming from above and the created

// Future failing for another reason.

val p = new Promise[(Try[Unit], Seq[Future[Unit]])]

p.setInterruptHandler {

case e =>

// Interrupt the outstanding requests.

sel.raise(e)

// Halt the computation with a failure.

p.updateIfEmpty(Throw(e))

}

// When the select finishes, update the promise with the value.

sel.respond(p.updateIfEmpty)

p.flatMap {

case (tryRes, remaining) =>

val delta = if (tryRes.isReturn) weight else 0

go(attempts, reliability \* decay + delta, remaining)

}

}

go(maxAttempts, 0, Seq.empty)

}

}

/\*\*

\* Write a log message recording each invocation of each warmup

\* action. The log message is comma-separated, with the following

\* fields:

\*

\* name:

\* The supplied name.

\*

\* start time:

\* The number of milliseconds since the start of the Unix

\* epoch.

\*

\* duration:

\* How long this warmup action took, in milliseconds.

\*

\* result:

\* "passed" or "failed" depending on whether the Future

\* returned an exception.

\*

\* exception type:

\* If the result "failed", then this will be the name of

\* the exception that casued the failure. If it "passed",

\* it will be the empty string.

\*

\* These messages should be sufficient to get a picture of how

\* warmup proceeded, since they allow the messages to be ordered

\* and sorted by type. You can use this information to tune the

\* warmup parameters.

\*/

class Logged(logger: Logger) extends Runner {

override def runOne(name: String, action: => Future[Unit]): Future[Unit] = {

val start = Time.now

val startStr = start.sinceEpoch.inMilliseconds.toString

action.respond {

case Throw(WarmupComplete) =>

// Don't log anything for computations that we abandoned

// because warmup is complete.

case r =>

val duration = (Time.now - start).inMilliseconds

val result = r match {

case Throw(e) => "failed," + e.toString.takeWhile(\_ != '\n')

case \_ => "passed,"

}

logger.info(s"$name,${startStr}ms,${duration}ms,$result")

}

}

}

/\*\*

\* Ensure that no more than the specified number of invocations of a

\* warmup action are happening at one time.

\*/

class LimitedConcurrency(limit: Int) extends Runner {

private[this] val sem = new AsyncSemaphore(limit)

override def runOne(name: String, action: => Future[Unit]): Future[Unit] =

sem.acquireAndRun(action)

}

/\*\*

\* Create a new Warmup that performs this single action.

\*/

def apply[A](name: String)(f: A => Future[\_]): Warmup[A] = new Warmup(Map(name -> f))

/\*\*

\* Create a Warmup that does nothing. This is useful in concert with

\* warmField.

\*/

def empty[A]: Warmup[A] = new Warmup[A](Map.empty)

}

/\*\*

\* A set of independent warmup actions. Each action should be the

\* minimum work that can be done in order to exercise a code

\* path. Runners can be used to e.g. run the actions repeatedly or

\* with timeouts.

\*/

class Warmup[A](val actions: Map[String, A => Future[\_]]) {

def ++(other: Warmup[A]) = new Warmup[A](actions ++ other.actions)

/\*\*

\* The names of the individual warmup actions that this warmup is

\* composed of.

\*/

def names: Set[String] = actions.keySet

/\*\*

\* Create a new Warmup that does all of the actions of this warmup

\* and additionally does warmup on the value specified by `f`.

\*/

def warmField[B](f: A => B)(implicit w: Warmup[B]): Warmup[A] =

new Warmup[A](actions ++ (w.actions.mapValues(f.andThen)))

}