package com.twitter.ann.common

import com.twitter.finagle.stats.CategorizingExceptionStatsHandler

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.tracing.DefaultTracer

import com.twitter.finagle.tracing.Trace

import com.twitter.finagle.util.DefaultTimer

import com.twitter.finagle.util.Rng

import com.twitter.inject.logging.MDCKeys

import com.twitter.util.Closable

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Time

import com.twitter.util.Timer

import com.twitter.util.logging.Logging

import java.util.concurrent.atomic.AtomicInteger

import org.slf4j.MDC

/\*\*

\* A Task that will be scheduled to execute periodically on every interval. If a task takes

\* longer than an interval to complete, it will be immediately scheduled to run.

\*/

trait Task extends Closable { self: Logging =>

// Exposed if the implementation of `task` need to report failures

val exnStatsHandler = new CategorizingExceptionStatsHandler(categorizer = \_ => Some("failures"))

protected val statsReceiver: StatsReceiver

private val totalTasks = statsReceiver.counter("total")

private val successfulTasks = statsReceiver.counter("success")

private val taskLatency = statsReceiver.stat("latency\_ms")

private val activeTasks = new AtomicInteger(0)

protected[common] val rng: Rng = Rng.threadLocal

protected[common] val timer: Timer = DefaultTimer

@volatile private var taskLoop: Future[Unit] = null

/\*\* Execute the task wih bookkeeping \*\*/

private def run(): Future[Unit] = {

totalTasks.incr()

activeTasks.getAndIncrement()

val start = Time.now

val runningTask =

// Setup a new trace root for this task. We also want logs to contain

// the same trace information finatra populates for requests.

// See com.twitter.finatra.thrift.filters.TraceIdMDCFilter

Trace.letTracerAndNextId(DefaultTracer) {

val trace = Trace()

MDC.put(MDCKeys.TraceId, trace.id.traceId.toString)

MDC.put(MDCKeys.TraceSampled, trace.id.\_sampled.getOrElse(false).toString)

MDC.put(MDCKeys.TraceSpanId, trace.id.spanId.toString)

info(s"starting task ${getClass.toString}")

task()

.onSuccess({ \_ =>

info(s"completed task ${getClass.toString}")

successfulTasks.incr()

})

.onFailure({ e =>

warn(s"failed task. ", e)

exnStatsHandler.record(statsReceiver, e)

})

}

runningTask.transform { \_ =>

val elapsed = Time.now - start

activeTasks.getAndDecrement()

taskLatency.add(elapsed.inMilliseconds)

Future

.sleep(taskInterval)(timer)

.before(run())

}

}

// Body of a task to run

protected def task(): Future[Unit]

// Task interval

protected def taskInterval: Duration

/\*\*

\* Start the task after random jitter

\*/

final def jitteredStart(): Unit = synchronized {

if (taskLoop != null) {

throw new RuntimeException(s"task already started")

} else {

val jitterNs = rng.nextLong(taskInterval.inNanoseconds)

val jitter = Duration.fromNanoseconds(jitterNs)

taskLoop = Future

.sleep(jitter)(timer)

.before(run())

}

}

/\*\*

\* Start the task without applying any delay

\*/

final def startImmediately(): Unit = synchronized {

if (taskLoop != null) {

throw new RuntimeException(s"task already started")

} else {

taskLoop = run()

}

}

/\*\*

\* Close the task. A closed task cannot be restarted.

\*/

override def close(deadline: Time): Future[Unit] = {

if (taskLoop != null) {

taskLoop.raise(new InterruptedException("task closed"))

}

Future.Done

}

}