package com.twitter.product\_mixer.shared\_library.observer

import com.twitter.finagle.stats.Counter

import com.twitter.finagle.stats.RollupStatsReceiver

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.servo.util.CancelledExceptionExtractor

import com.twitter.stitch.Arrow

import com.twitter.stitch.Stitch

import com.twitter.util.Duration

import com.twitter.util.Future

import com.twitter.util.Throwables

import com.twitter.util.Try

/\*\*

\* Helper functions to observe requests, success, failures, cancellations, exceptions, and latency.

\* Supports native functions and asynchronous operations.

\*/

object Observer {

val Requests = "requests"

val Success = "success"

val Failures = "failures"

val Cancelled = "cancelled"

val Latency = "latency\_ms"

/\*\*

\* Helper function to observe a stitch

\*

\* @see [[StitchObserver]]

\*/

def stitch[T](statsReceiver: StatsReceiver, scopes: String\*): StitchObserver[T] =

new StitchObserver[T](statsReceiver, scopes)

/\*\*

\* Helper function to observe an arrow

\*

\* @see [[ArrowObserver]]

\*/

def arrow[In, Out](statsReceiver: StatsReceiver, scopes: String\*): ArrowObserver[In, Out] =

new ArrowObserver[In, Out](statsReceiver, scopes)

/\*\*

\* Helper function to observe a future

\*

\* @see [[FutureObserver]]

\*/

def future[T](statsReceiver: StatsReceiver, scopes: String\*): FutureObserver[T] =

new FutureObserver[T](statsReceiver, scopes)

/\*\*

\* Helper function to observe a function

\*

\* @see [[FunctionObserver]]

\*/

def function[T](statsReceiver: StatsReceiver, scopes: String\*): FunctionObserver[T] =

new FunctionObserver[T](statsReceiver, scopes)

/\*\*

\* [[StitchObserver]] can record latency stats, success counters, and

\* detailed failure stats for the results of a Stitch computation.

\*/

class StitchObserver[T](

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends Observer[T] {

/\*\*

\* Record stats for the provided Stitch.

\* The result of the computation is passed through.

\*

\* @note the provided Stitch must contain the parts that need to be timed.

\* Using this on just the result of the computation the latency stat

\* will be incorrect.

\*/

def apply(stitch: => Stitch[T]): Stitch[T] =

Stitch.time(stitch).map(observe.tupled).lowerFromTry

}

/\*\*

\* [[ArrowObserver]] can record the latency stats, success counters, and

\* detailed failure stats for the result of an Arrow computation.

\* The result of the computation is passed through.

\*/

class ArrowObserver[In, Out](

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends Observer[Out] {

/\*\*

\* Returns a new Arrow that records stats when it's run.

\* The result of the Arrow is passed through.

\*

\* @note the provided Arrow must contain the parts that need to be timed.

\* Using this on just the result of the computation the latency stat

\* will be incorrect.

\*/

def apply(arrow: Arrow[In, Out]): Arrow[In, Out] =

Arrow.time(arrow).map(observe.tupled).lowerFromTry

}

/\*\*

\* [[FutureObserver]] can record latency stats, success counters, and

\* detailed failure stats for the results of a Future computation.

\*/

class FutureObserver[T](

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends Observer[T] {

/\*\*

\* Record stats for the provided Future.

\* The result of the computation is passed through.

\*

\* @note the provided Future must contain the parts that need to be timed.

\* Using this on just the result of the computation the latency stat

\* will be incorrect.

\*/

def apply(future: => Future[T]): Future[T] =

Stat

.timeFuture(latencyStat)(future)

.onSuccess(observeSuccess)

.onFailure(observeFailure)

}

/\*\*

\* [[FunctionObserver]] can record latency stats, success counters, and

\* detailed failure stats for the results of a computation computation.

\*/

class FunctionObserver[T](

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends Observer[T] {

/\*\*

\* Record stats for the provided computation.

\* The result of the computation is passed through.

\*

\* @note the provided computation must contain the parts that need to be timed.

\* Using this on just the result of the computation the latency stat

\* will be incorrect.

\*/

def apply(f: => T): T = {

Try(Stat.time(latencyStat)(f))

.onSuccess(observeSuccess)

.onFailure(observeFailure)

.apply()

}

}

/\*\* [[Observer]] provides methods for recording latency, success, and failure stats \*/

trait Observer[T] {

protected val statsReceiver: StatsReceiver

/\*\* Scopes that prefix all stats \*/

protected val scopes: Seq[String]

private val rollupStatsReceiver = new RollupStatsReceiver(statsReceiver.scope(scopes: \_\*))

private val requestsCounter: Counter = statsReceiver.counter(scopes :+ Requests: \_\*)

private val successCounter: Counter = statsReceiver.counter(scopes :+ Success: \_\*)

// create the stats so their metrics paths are always present but

// defer to the [[RollupStatsReceiver]] to increment these stats

rollupStatsReceiver.counter(Failures)

rollupStatsReceiver.counter(Cancelled)

/\*\* Serialize a throwable and it's causes into a seq of Strings for scoping metrics \*/

protected def serializeThrowable(throwable: Throwable): Seq[String] =

Throwables.mkString(throwable)

/\*\* Used to record latency in milliseconds \*/

protected val latencyStat: Stat = statsReceiver.stat(scopes :+ Latency: \_\*)

/\*\* Records the latency from a [[Duration]] \*/

protected val observeLatency: Duration => Unit = { latency =>

latencyStat.add(latency.inMilliseconds)

}

/\*\* Records successes \*/

protected val observeSuccess: T => Unit = { \_ =>

requestsCounter.incr()

successCounter.incr()

}

/\*\* Records failures and failure details \*/

protected val observeFailure: Throwable => Unit = {

case CancelledExceptionExtractor(throwable) =>

requestsCounter.incr()

rollupStatsReceiver.counter(Cancelled +: serializeThrowable(throwable): \_\*).incr()

case throwable =>

requestsCounter.incr()

rollupStatsReceiver.counter(Failures +: serializeThrowable(throwable): \_\*).incr()

}

/\*\* Records the latency, successes, and failures \*/

protected val observe: (Try[T], Duration) => Try[T] =

(response: Try[T], runDuration: Duration) => {

observeLatency(runDuration)

response

.onSuccess(observeSuccess)

.onFailure(observeFailure)

}

}

}