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

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.product\_mixer.shared\_library.observer.Observer.ArrowObserver

import com.twitter.product\_mixer.shared\_library.observer.Observer.FunctionObserver

import com.twitter.product\_mixer.shared\_library.observer.Observer.FutureObserver

import com.twitter.product\_mixer.shared\_library.observer.Observer.Observer

import com.twitter.product\_mixer.shared\_library.observer.Observer.StitchObserver

import com.twitter.product\_mixer.shared\_library.observer.ResultsObserver.ResultsObserver

import com.twitter.stitch.Arrow

import com.twitter.stitch.Stitch

import com.twitter.util.Future

import com.twitter.util.Try

/\*\*

\* Helper functions to observe requests, successes, failures, cancellations, exceptions, latency,

\* and result counts and time-series stats. Supports native functions and asynchronous operations.

\*

\* Note that since time-series stats are expensive to compute (relative to counters), prefer

\* [[ResultsObserver]] unless a time-series stat is needed.

\*/

object ResultsStatsObserver {

val Size = "size"

/\*\*

\* Helper function to observe a stitch and result counts and time-series stats

\*/

def stitchResultsStats[T](

size: T => Int,

statsReceiver: StatsReceiver,

scopes: String\*

): StitchResultsStatsObserver[T] = {

new StitchResultsStatsObserver[T](size, statsReceiver, scopes)

}

/\*\*

\* Helper function to observe a stitch and traversable (e.g. Seq, Set) result counts and

\* time-series stats

\*/

def stitchResultsStats[T <: TraversableOnce[\_]](

statsReceiver: StatsReceiver,

scopes: String\*

): StitchResultsStatsObserver[T] = {

new StitchResultsStatsObserver[T](\_.size, statsReceiver, scopes)

}

/\*\*

\* Helper function to observe an arrow and result counts and time-series stats

\*/

def arrowResultsStats[T, U](

size: U => Int,

statsReceiver: StatsReceiver,

scopes: String\*

): ArrowResultsStatsObserver[T, U] = {

new ArrowResultsStatsObserver[T, U](size, statsReceiver, scopes)

}

/\*\*

\* Helper function to observe an arrow and traversable (e.g. Seq, Set) result counts and

\* \* time-series stats

\*/

def arrowResultsStats[T, U <: TraversableOnce[\_]](

statsReceiver: StatsReceiver,

scopes: String\*

): ArrowResultsStatsObserver[T, U] = {

new ArrowResultsStatsObserver[T, U](\_.size, statsReceiver, scopes)

}

/\*\*

\* Helper function to observe an arrow and result counts

\*

\* @see [[TransformingArrowResultsStatsObserver]]

\*/

def transformingArrowResultsStats[In, Out, Transformed](

transformer: Out => Try[Transformed],

size: Transformed => Int,

statsReceiver: StatsReceiver,

scopes: String\*

): TransformingArrowResultsStatsObserver[In, Out, Transformed] = {

new TransformingArrowResultsStatsObserver[In, Out, Transformed](

transformer,

size,

statsReceiver,

scopes)

}

/\*\*

\* Helper function to observe an arrow and traversable (e.g. Seq, Set) result counts

\*

\* @see [[TransformingArrowResultsStatsObserver]]

\*/

def transformingArrowResultsStats[In, Out, Transformed <: TraversableOnce[\_]](

transformer: Out => Try[Transformed],

statsReceiver: StatsReceiver,

scopes: String\*

): TransformingArrowResultsStatsObserver[In, Out, Transformed] = {

new TransformingArrowResultsStatsObserver[In, Out, Transformed](

transformer,

\_.size,

statsReceiver,

scopes)

}

/\*\*

\* Helper function to observe a future and result counts and time-series stats

\*/

def futureResultsStats[T](

size: T => Int,

statsReceiver: StatsReceiver,

scopes: String\*

): FutureResultsStatsObserver[T] = {

new FutureResultsStatsObserver[T](size, statsReceiver, scopes)

}

/\*\*

\* Helper function to observe a future and traversable (e.g. Seq, Set) result counts and

\* time-series stats

\*/

def futureResultsStats[T <: TraversableOnce[\_]](

statsReceiver: StatsReceiver,

scopes: String\*

): FutureResultsStatsObserver[T] = {

new FutureResultsStatsObserver[T](\_.size, statsReceiver, scopes)

}

/\*\*

\* Helper function observe a function and result counts and time-series stats

\*/

def functionResultsStats[T](

size: T => Int,

statsReceiver: StatsReceiver,

scopes: String\*

): FunctionResultsStatsObserver[T] = {

new FunctionResultsStatsObserver[T](size, statsReceiver, scopes)

}

/\*\*

\* Helper function observe a function and traversable (e.g. Seq, Set) result counts and

\* time-series stats

\*/

def functionResultsStats[T <: TraversableOnce[\_]](

statsReceiver: StatsReceiver,

scopes: String\*

): FunctionResultsStatsObserver[T] = {

new FunctionResultsStatsObserver[T](\_.size, statsReceiver, scopes)

}

class StitchResultsStatsObserver[T](

override val size: T => Int,

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends StitchObserver[T](statsReceiver, scopes)

with ResultsStatsObserver[T] {

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

super

.apply(stitch)

.onSuccess(observeResults)

}

class ArrowResultsStatsObserver[T, U](

override val size: U => Int,

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends ArrowObserver[T, U](statsReceiver, scopes)

with ResultsStatsObserver[U] {

override def apply(arrow: Arrow[T, U]): Arrow[T, U] =

super

.apply(arrow)

.onSuccess(observeResults)

}

/\*\*

\* [[TransformingArrowResultsStatsObserver]] functions like an [[ArrowObserver]] except

\* that it transforms the result using [[transformer]] before recording stats.

\*

\* The original non-transformed result is then returned.

\*/

class TransformingArrowResultsStatsObserver[In, Out, Transformed](

val transformer: Out => Try[Transformed],

override val size: Transformed => Int,

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends Observer[Transformed]

with ResultsStatsObserver[Transformed] {

/\*\*

\* Returns a new Arrow that records stats on the result after applying [[transformer]] when it's run.

\* The original, non-transformed, 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 {

case (response, stitchRunDuration) =>

observe(response.flatMap(transformer), stitchRunDuration)

.onSuccess(observeResults)

response

}.lowerFromTry

}

}

class FutureResultsStatsObserver[T](

override val size: T => Int,

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends FutureObserver[T](statsReceiver, scopes)

with ResultsStatsObserver[T] {

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

super

.apply(future)

.onSuccess(observeResults)

}

class FunctionResultsStatsObserver[T](

override val size: T => Int,

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends FunctionObserver[T](statsReceiver, scopes)

with ResultsStatsObserver[T] {

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

observeResults(super.apply(f))

}

}

trait ResultsStatsObserver[T] extends ResultsObserver[T] {

private val sizeStat: Stat = statsReceiver.stat(scopes :+ Size: \_\*)

protected override def observeResults(results: T): T = {

val resultsSize = size(results)

sizeStat.add(resultsSize)

observeResultsWithSize(results, resultsSize)

}

}

}