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

import com.twitter.finagle.stats.Counter

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.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. Supports native functions and asynchronous operations.

\*/

object ResultsObserver {

val Total = "total"

val Found = "found"

val NotFound = "not\_found"

/\*\*

\* Helper function to observe a stitch and result counts

\*

\* @see [[StitchResultsObserver]]

\*/

def stitchResults[T](

size: T => Int,

statsReceiver: StatsReceiver,

scopes: String\*

): StitchResultsObserver[T] = {

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

}

/\*\*

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

\*

\* @see [[StitchResultsObserver]]

\*/

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

statsReceiver: StatsReceiver,

scopes: String\*

): StitchResultsObserver[T] = {

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

}

/\*\*

\* Helper function to observe an arrow and result counts

\*

\* @see [[ArrowResultsObserver]]

\*/

def arrowResults[In, Out](

size: Out => Int,

statsReceiver: StatsReceiver,

scopes: String\*

): ArrowResultsObserver[In, Out] = {

new ArrowResultsObserver[In, Out](size, statsReceiver, scopes)

}

/\*\*

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

\*

\* @see [[ArrowResultsObserver]]

\*/

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

statsReceiver: StatsReceiver,

scopes: String\*

): ArrowResultsObserver[In, Out] = {

new ArrowResultsObserver[In, Out](\_.size, statsReceiver, scopes)

}

/\*\*

\* Helper function to observe an arrow and result counts

\*

\* @see [[TransformingArrowResultsObserver]]

\*/

def transformingArrowResults[In, Out, Transformed](

transformer: Out => Try[Transformed],

size: Transformed => Int,

statsReceiver: StatsReceiver,

scopes: String\*

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

new TransformingArrowResultsObserver[In, Out, Transformed](

transformer,

size,

statsReceiver,

scopes)

}

/\*\*

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

\*

\* @see [[TransformingArrowResultsObserver]]

\*/

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

transformer: Out => Try[Transformed],

statsReceiver: StatsReceiver,

scopes: String\*

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

new TransformingArrowResultsObserver[In, Out, Transformed](

transformer,

\_.size,

statsReceiver,

scopes)

}

/\*\*

\* Helper function to observe a future and result counts

\*

\* @see [[FutureResultsObserver]]

\*/

def futureResults[T](

size: T => Int,

statsReceiver: StatsReceiver,

scopes: String\*

): FutureResultsObserver[T] = {

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

}

/\*\*

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

\*

\* @see [[FutureResultsObserver]]

\*/

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

statsReceiver: StatsReceiver,

scopes: String\*

): FutureResultsObserver[T] = {

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

}

/\*\*

\* Helper function to observe a function and result counts

\*

\* @see [[FunctionResultsObserver]]

\*/

def functionResults[T](

size: T => Int,

statsReceiver: StatsReceiver,

scopes: String\*

): FunctionResultsObserver[T] = {

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

}

/\*\*

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

\*

\* @see [[FunctionResultsObserver]]

\*/

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

statsReceiver: StatsReceiver,

scopes: String\*

): FunctionResultsObserver[T] = {

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

}

/\*\* [[StitchObserver]] that also records result size \*/

class StitchResultsObserver[T](

override val size: T => Int,

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends StitchObserver[T](statsReceiver, scopes)

with ResultsObserver[T] {

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

super

.apply(stitch)

.onSuccess(observeResults)

}

/\*\* [[ArrowObserver]] that also records result size \*/

class ArrowResultsObserver[In, Out](

override val size: Out => Int,

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

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

with ResultsObserver[Out] {

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

super

.apply(arrow)

.onSuccess(observeResults)

}

/\*\*

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

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

\*

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

\*/

class TransformingArrowResultsObserver[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 ResultsObserver[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

}

}

/\*\* [[FutureObserver]] that also records result size \*/

class FutureResultsObserver[T](

override val size: T => Int,

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends FutureObserver[T](statsReceiver, scopes)

with ResultsObserver[T] {

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

super

.apply(future)

.onSuccess(observeResults)

}

/\*\* [[FunctionObserver]] that also records result size \*/

class FunctionResultsObserver[T](

override val size: T => Int,

override val statsReceiver: StatsReceiver,

override val scopes: Seq[String])

extends FunctionObserver[T](statsReceiver, scopes)

with ResultsObserver[T] {

override def apply(f: => T): T = observeResults(super.apply(f))

}

/\*\* [[ResultsObserver]] provides methods for recording stats for the result size \*/

trait ResultsObserver[T] {

protected val statsReceiver: StatsReceiver

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

protected val scopes: Seq[String]

protected val totalCounter: Counter = statsReceiver.counter(scopes :+ Total: \_\*)

protected val foundCounter: Counter = statsReceiver.counter(scopes :+ Found: \_\*)

protected val notFoundCounter: Counter = statsReceiver.counter(scopes :+ NotFound: \_\*)

/\*\* given a [[T]] returns it's size. \*/

protected val size: T => Int

/\*\* Records the size of the `results` using [[size]] and return the original value. \*/

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

val resultsSize = size(results)

observeResultsWithSize(results, resultsSize)

}

/\*\*

\* Records the `resultsSize` and returns the `results`

\*

\* This is useful if the size is already available and is expensive to calculate.

\*/

protected def observeResultsWithSize(results: T, resultsSize: Int): T = {

if (resultsSize > 0) {

totalCounter.incr(resultsSize)

foundCounter.incr()

} else {

notFoundCounter.incr()

}

results

}

}

}