package com.twitter.follow\_recommendations.common.base

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.product\_mixer.core.quality\_factor.QualityFactorObserver

import com.twitter.stitch.Arrow

import com.twitter.stitch.Stitch

import com.twitter.util.Stopwatch

import java.util.concurrent.TimeUnit

import scala.util.control.NonFatal

object StatsUtil {

val LatencyName = "latency\_ms"

val RequestName = "requests"

val SuccessName = "success"

val FailureName = "failures"

val ResultsName = "results"

val ResultsStat = "results\_stat"

val EmptyResultsName = "empty"

val NonEmptyResultsName = "non\_empty"

val ValidCount = "valid"

val InvalidCount = "invalid"

val InvalidHasReasons = "has\_reasons"

val Reasons = "reasons"

val QualityFactorStat = "quality\_factor\_stat"

val QualityFactorCounts = "quality\_factor\_counts"

/\*\*

\* Helper function for timing a stitch, returning the original stitch.

\*/

def profileStitch[T](stitch: Stitch[T], stat: StatsReceiver): Stitch[T] = {

Stitch

.time(stitch)

.map {

case (response, stitchRunDuration) =>

stat.counter(RequestName).incr()

stat.stat(LatencyName).add(stitchRunDuration.inMilliseconds)

response

.onSuccess { \_ => stat.counter(SuccessName).incr() }

.onFailure { e =>

stat.counter(FailureName).incr()

stat.scope(FailureName).counter(getCleanClassName(e)).incr()

}

}

.lowerFromTry

}

/\*\*

\* Helper function for timing an arrow, returning the original arrow.

\*/

def profileArrow[T, U](arrow: Arrow[T, U], stat: StatsReceiver): Arrow[T, U] = {

Arrow

.time(arrow)

.map {

case (response, stitchRunDuration) =>

stat.counter(RequestName).incr()

stat.stat(LatencyName).add(stitchRunDuration.inMilliseconds)

response

.onSuccess { \_ => stat.counter(SuccessName).incr() }

.onFailure { e =>

stat.counter(FailureName).incr()

stat.scope(FailureName).counter(getCleanClassName(e)).incr()

}

}

.lowerFromTry

}

/\*\*

\* Helper function to count and track the distribution of results

\*/

def profileResults[T](results: T, stat: StatsReceiver, size: T => Int): T = {

val numResults = size(results)

stat.counter(ResultsName).incr(numResults)

if (numResults == 0) {

stat.counter(EmptyResultsName).incr()

results

} else {

stat.stat(ResultsStat).add(numResults)

stat.counter(NonEmptyResultsName).incr()

results

}

}

/\*\*

\* Helper function to count and track the distribution of a list of results

\*/

def profileSeqResults[T](results: Seq[T], stat: StatsReceiver): Seq[T] = {

profileResults[Seq[T]](results, stat, \_.size)

}

/\*\*

\* Helper function for timing a stitch and count the number of results, returning the original stitch.

\*/

def profileStitchResults[T](stitch: Stitch[T], stat: StatsReceiver, size: T => Int): Stitch[T] = {

profileStitch(stitch, stat).onSuccess { results => profileResults(results, stat, size) }

}

/\*\*

\* Helper function for timing an arrow and count the number of results, returning the original arrow.

\*/

def profileArrowResults[T, U](

arrow: Arrow[T, U],

stat: StatsReceiver,

size: U => Int

): Arrow[T, U] = {

profileArrow(arrow, stat).onSuccess { results => profileResults(results, stat, size) }

}

/\*\*

\* Helper function for timing a stitch and count a seq of results, returning the original stitch.

\*/

def profileStitchSeqResults[T](stitch: Stitch[Seq[T]], stat: StatsReceiver): Stitch[Seq[T]] = {

profileStitchResults[Seq[T]](stitch, stat, \_.size)

}

/\*\*

\* Helper function for timing a stitch and count optional results, returning the original stitch.

\*/

def profileStitchOptionalResults[T](

stitch: Stitch[Option[T]],

stat: StatsReceiver

): Stitch[Option[T]] = {

profileStitchResults[Option[T]](stitch, stat, \_.size)

}

/\*\*

\* Helper function for timing a stitch and count a map of results, returning the original stitch.

\*/

def profileStitchMapResults[K, V](

stitch: Stitch[Map[K, V]],

stat: StatsReceiver

): Stitch[Map[K, V]] = {

profileStitchResults[Map[K, V]](stitch, stat, \_.size)

}

def getCleanClassName(obj: Object): String =

obj.getClass.getSimpleName.stripSuffix("$")

/\*\*

\* Helper function for timing a stitch and count a list of PredicateResult

\*/

def profilePredicateResults(

predicateResult: Stitch[Seq[PredicateResult]],

statsReceiver: StatsReceiver

): Stitch[Seq[PredicateResult]] = {

profileStitch[Seq[PredicateResult]](

predicateResult,

statsReceiver

).onSuccess {

\_.map {

case PredicateResult.Valid =>

statsReceiver.counter(ValidCount).incr()

case PredicateResult.Invalid(reasons) =>

statsReceiver.counter(InvalidCount).incr()

reasons.map { filterReason =>

statsReceiver.counter(InvalidHasReasons).incr()

statsReceiver.scope(Reasons).counter(filterReason.reason).incr()

}

}

}

}

/\*\*

\* Helper function for timing a stitch and count individual PredicateResult

\*/

def profilePredicateResult(

predicateResult: Stitch[PredicateResult],

statsReceiver: StatsReceiver

): Stitch[PredicateResult] = {

profilePredicateResults(

predicateResult.map(Seq(\_)),

statsReceiver

).map(\_.head)

}

/\*\*

\* Helper function for timing an arrow and count a list of PredicateResult

\*/

def profilePredicateResults[Q](

predicateResult: Arrow[Q, Seq[PredicateResult]],

statsReceiver: StatsReceiver

): Arrow[Q, Seq[PredicateResult]] = {

profileArrow[Q, Seq[PredicateResult]](

predicateResult,

statsReceiver

).onSuccess {

\_.map {

case PredicateResult.Valid =>

statsReceiver.counter(ValidCount).incr()

case PredicateResult.Invalid(reasons) =>

statsReceiver.counter(InvalidCount).incr()

reasons.map { filterReason =>

statsReceiver.counter(InvalidHasReasons).incr()

statsReceiver.scope(Reasons).counter(filterReason.reason).incr()

}

}

}

}

/\*\*

\* Helper function for timing an arrow and count individual PredicateResult

\*/

def profilePredicateResult[Q](

predicateResult: Arrow[Q, PredicateResult],

statsReceiver: StatsReceiver

): Arrow[Q, PredicateResult] = {

profilePredicateResults(

predicateResult.map(Seq(\_)),

statsReceiver

).map(\_.head)

}

/\*\*

\* Helper function for timing a stitch code block

\*/

def profileStitchSeqResults[T](

stats: StatsReceiver

)(

block: => Stitch[Seq[T]]

): Stitch[Seq[T]] = {

stats.counter(RequestName).incr()

profileStitch(stats.stat(LatencyName), TimeUnit.MILLISECONDS) {

block onSuccess { r =>

if (r.isEmpty) stats.counter(EmptyResultsName).incr()

stats.stat(ResultsStat).add(r.size)

} onFailure { e =>

{

stats.counter(FailureName).incr()

stats.scope(FailureName).counter(e.getClass.getName).incr()

}

}

}

}

/\*\*

\* Time a given asynchronous `f` using the given `unit`.

\*/

def profileStitch[A](stat: Stat, unit: TimeUnit)(f: => Stitch[A]): Stitch[A] = {

val start = Stopwatch.timeNanos()

try {

f.respond { \_ => stat.add(unit.convert(Stopwatch.timeNanos() - start, TimeUnit.NANOSECONDS)) }

} catch {

case NonFatal(e) =>

stat.add(unit.convert(Stopwatch.timeNanos() - start, TimeUnit.NANOSECONDS))

Stitch.exception(e)

}

}

def observeStitchQualityFactor[T](

stitch: Stitch[T],

qualityFactorObserverOption: Option[QualityFactorObserver],

statsReceiver: StatsReceiver

): Stitch[T] = {

qualityFactorObserverOption

.map { observer =>

Stitch

.time(stitch)

.map {

case (response, stitchRunDuration) =>

observer(response, stitchRunDuration)

val qfVal = observer.qualityFactor.currentValue.floatValue() \* 10000

statsReceiver.counter(QualityFactorCounts).incr()

statsReceiver

.stat(QualityFactorStat)

.add(qfVal)

response

}

.lowerFromTry

}.getOrElse(stitch)

}

}