package com.twitter.servo.util

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.util.Future

import scala.collection.mutable

/\*\*

\* Categorizes an exception according to some criteria.

\* n.b. Implemented in terms of lift rather than apply to avoid extra allocations when

\* used when lifting the effect.

\*/

trait ExceptionCategorizer {

import ExceptionCategorizer.\_

def lift(effect: Effect[Category]): Effect[Throwable]

def apply(t: Throwable): Set[Category] = {

val s = mutable.Set.empty[Category]

lift(Effect(s += \_))(t)

s.toSet

}

/\*\*

\* construct a new categorizer that prepends scope to all categories returned by this categorizer

\*/

def scoped(scope: Seq[String]): ExceptionCategorizer =

if (scope.isEmpty) {

this

} else {

val scopeIt: Category => Category = Memoize(scope ++ \_)

fromLift(effect => lift(effect.contramap(scopeIt)))

}

/\*\*

\* construct a new categorizer that returns the union of the categories returned by this and that

\*/

def ++(that: ExceptionCategorizer): ExceptionCategorizer =

fromLift(effect => this.lift(effect).also(that.lift(effect)))

/\*\*

\* construct a new categorizer that only returns categories for throwables matching pred

\*/

def onlyIf(pred: Throwable => Boolean): ExceptionCategorizer =

fromLift(lift(\_).onlyIf(pred))

}

object ExceptionCategorizer {

type Category = Seq[String]

def const(categories: Set[Category]): ExceptionCategorizer = ExceptionCategorizer(\_ => categories)

def const(c: Category): ExceptionCategorizer = const(Set(c))

def const(s: String): ExceptionCategorizer = const(Seq(s))

def apply(fn: Throwable => Set[Category]): ExceptionCategorizer =

new ExceptionCategorizer {

def lift(effect: Effect[Category]) = Effect[Throwable](t => fn(t).foreach(effect))

override def apply(t: Throwable) = fn(t)

}

def fromLift(fn: Effect[Category] => Effect[Throwable]): ExceptionCategorizer =

new ExceptionCategorizer {

def lift(effect: Effect[Category]) = fn(effect)

}

def singular(fn: Throwable => Category): ExceptionCategorizer =

fromLift(\_.contramap(fn))

def simple(fn: Throwable => String): ExceptionCategorizer =

singular(fn.andThen(Seq(\_)))

def default(

name: Category = Seq("exceptions"),

sanitizeClassnameChain: Throwable => Seq[String] = ThrowableHelper.sanitizeClassnameChain

): ExceptionCategorizer =

ExceptionCategorizer.const(name) ++

ExceptionCategorizer.singular(sanitizeClassnameChain).scoped(name)

}

/\*\*

\* Increments a counter for each category returned by the exception categorizer

\*

\* @param statsReceiver

\* the unscoped statsReceiver on which to hang the counters

\* @param categorizer

\* A function that returns a list of category names that a throwable should be counted under.

\*/

class ExceptionCounter(statsReceiver: StatsReceiver, categorizer: ExceptionCategorizer) {

/\*\*

\* alternative constructor for backwards compatibility

\*

\* @param statsReceiver

\* the unscoped statsReceiver on which to hang the counters

\* @param name

\* the counter name for total exceptions, and scope for individual

\* exception counters. default value is `exceptions`

\* @param sanitizeClassnameChain

\* A function that can be used to cleanup classnames before passing them to the StatsReceiver.

\*/

def this(

statsReceiver: StatsReceiver,

name: String,

sanitizeClassnameChain: Throwable => Seq[String]

) =

this(statsReceiver, ExceptionCategorizer.default(List(name), sanitizeClassnameChain))

/\*\*

\* provided for backwards compatibility

\*/

def this(statsReceiver: StatsReceiver) =

this(statsReceiver, ExceptionCategorizer.default())

/\*\*

\* provided for backwards compatibility

\*/

def this(statsReceiver: StatsReceiver, name: String) =

this(statsReceiver, ExceptionCategorizer.default(List(name)))

/\*\*

\* provided for backwards compatibility

\*/

def this(statsReceiver: StatsReceiver, sanitizeClassnameChain: Throwable => Seq[String]) =

this(

statsReceiver,

ExceptionCategorizer.default(sanitizeClassnameChain = sanitizeClassnameChain)

)

private[this] val counter = categorizer.lift(Effect(statsReceiver.counter(\_: \_\*).incr()))

/\*\*

\* count one or more throwables

\*/

def apply(t: Throwable, throwables: Throwable\*): Unit = {

counter(t)

if (throwables.nonEmpty) apply(throwables)

}

/\*\*

\* count n throwables

\*/

def apply(throwables: Iterable[Throwable]): Unit = {

throwables.foreach(counter)

}

/\*\*

\* wrap around a Future to capture exceptions

\*/

def apply[T](f: => Future[T]): Future[T] = {

f onFailure { case t => apply(t) }

}

}

/\*\*

\* A memoized exception counter factory.

\*

\* @param stats

\* the unscoped statsReceiver on which to hang the counters

\* @param categorizer

\* A function that returns a list of category names that a throwable should be counted under.

\*/

class MemoizedExceptionCounterFactory(stats: StatsReceiver, categorizer: ExceptionCategorizer) {

/\*\*

\* A memoized exception counter factory using the default categorizer.

\*

\* @param stats

\* the unscoped statsReceiver on which to hang the counters

\*/

def this(stats: StatsReceiver) =

this(stats, ExceptionCategorizer.default())

/\*\*

\* A memoized exception counter factory using a categorizer with the given suffix.

\*

\* @param stats

\* the unscoped statsReceiver on which to hang the counters

\* @param suffix

\* All created exception counters will have the

\* specified suffix added. This allows compatibility with

\* Servo's ExceptionCounter's name param (allows creating

\* exception counters that default to the "exceptions" namespace

\* as well as those with an otherwise-specified scope).

\*/

def this(stats: StatsReceiver, suffix: Seq[String]) =

this(stats, ExceptionCategorizer.default(suffix))

private[this] val getCounter =

Memoize { (path: Seq[String]) =>

new ExceptionCounter(stats, categorizer.scoped(path))

}

def apply(path: String\*): ExceptionCounter = getCounter(path)

}