package com.twitter.servo.util

import com.twitter.finagle.service.RetryPolicy

import com.twitter.finagle.stats.Stat

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.finagle.tracing.Trace

import com.twitter.finagle.FailedFastException

import com.twitter.finagle.Filter

import com.twitter.finagle.Service

import com.twitter.util.\_

import scala.util.control.NonFatal

/\*\*

\* A collection of FutureArrow factory functions.

\*/

object FutureArrow {

/\*\*

\* Produce a FutureArrow from a function `A => Future[B]`.

\*/

def apply[A, B](f: A => Future[B]): FutureArrow[A, B] =

new FutureArrow[A, B] {

override def apply(a: A): Future[B] =

try f(a)

catch {

case NonFatal(e) => Future.exception(e)

}

}

/\*\*

\* Produce a FutureArrow that supports recursive calls. Recursing from a `Future`

\* continuation is stack-safe, but direct recursion will use the stack, like a

\* normal method invocation.

\*/

def rec[A, B](f: FutureArrow[A, B] => A => Future[B]): FutureArrow[A, B] =

new FutureArrow[A, B] { self =>

private val g: A => Future[B] = f(this)

override def apply(a: A): Future[B] =

try g(a)

catch {

case NonFatal(e) => Future.exception(e)

}

}

/\*\*

\* Produce a FutureArrow from an FunctionArrow.

\*/

def fromFunctionArrow[A, B](f: FunctionArrow[A, B]): FutureArrow[A, B] =

FutureArrow[A, B](a => Future(f(a)))

/\*\*

\* Produce a FutureArrow from a function.

\*/

def fromFunction[A, B](f: A => B): FutureArrow[A, B] = fromFunctionArrow(FunctionArrow(f))

/\*\*

\* Produce a FutureArrow from a function `A => Try[B]`.

\*

\* The Try is evaluated within a Future. Thus, Throw results are translated

\* to `Future.exception`s.

\*/

def fromTry[A, B](f: A => Try[B]): FutureArrow[A, B] =

FutureArrow[A, B](a => Future.const(f(a)))

/\*\*

\* A FutureArrow that simply returns a Future of its argument.

\*/

def identity[A]: FutureArrow[A, A] =

FutureArrow[A, A](a => Future.value(a))

/\*\*

\* A FutureArrow with a constant result, regardless of input.

\*/

def const[A, B](value: Future[B]): FutureArrow[A, B] =

FutureArrow[A, B](\_ => value)

/\*\*

\* Appends two FutureArrows together.

\*

\* This forms a category with 'identity'.

\*/

def append[A, B, C](a: FutureArrow[A, B], b: FutureArrow[B, C]) = a.andThen(b)

/\*\*

\* Produce a FutureArrow that applies an FutureEffect, returning the argument

\* value as-is on success. If the effect returns an Future exception, then the

\* result of the filter will also be that exception.

\*/

def effect[A](effect: FutureEffect[A]): FutureArrow[A, A] =

apply(a => effect(a).map(\_ => a))

/\*\*

\* Produces a FutureArrow that proxies to one of two others, depending on a

\* predicate.

\*/

def choose[A, B](predicate: A => Boolean, ifTrue: FutureArrow[A, B], ifFalse: FutureArrow[A, B]) =

FutureArrow[A, B](a => if (predicate(a)) ifTrue(a) else ifFalse(a))

/\*\*

\* Produces a FutureArrow whose application is guarded by a predicate. `f` is

\* applied if the predicate returns true, otherwise the argument is simply

\* returned.

\*/

def onlyIf[A](predicate: A => Boolean, f: FutureArrow[A, A]) =

choose(predicate, f, identity[A])

/\*\*

\* Produces a FutureArrow that forwards to multiple FutureArrows and collects

\* the results into a `Seq[B]`. Results are gathered via Future.collect, so

\* failure semantics are inherited from that method.

\*/

def collect[A, B](arrows: Seq[FutureArrow[A, B]]): FutureArrow[A, Seq[B]] =

apply(a => Future.collect(arrows.map(arrow => arrow(a))))

private val RetryOnNonFailedFast: PartialFunction[Try[Any], Boolean] = {

case Throw(\_: FailedFastException) => false

case Throw(\_: Exception) => true

}

}

/\*\*

\* A function encapsulating an asynchronous computation.

\*

\* Background on the Arrow abstraction:

\* http://en.wikipedia.org/wiki/Arrow\_(computer\_science)

\*/

trait FutureArrow[-A, +B] extends (A => Future[B]) { self =>

/\*\*

\* Composes two FutureArrows. Produces a new FutureArrow that performs both in

\* series, depending on the success of the first.

\*/

def andThen[C](next: FutureArrow[B, C]): FutureArrow[A, C] =

FutureArrow[A, C](a => self(a).flatMap(next.apply))

/\*\*

\* Combines this FutureArrow with another, producing one that translates a

\* tuple of its constituents' arguments into a tuple of their results.

\*/

def zipjoin[C, D](other: FutureArrow[C, D]): FutureArrow[(A, C), (B, D)] =

FutureArrow[(A, C), (B, D)] {

case (a, c) => self(a) join other(c)

}

/\*\*

\* Converts a FutureArrow on a scalar input and output value into a FutureArrow on a

\* Sequence of input values producing a pairwise sequence of output values. The elements

\* of the input sequence are processed in parallel, so execution order is not guaranteed.

\* Results are gathered via Future.collect, so failure semantics are inherited from that method.

\*/

def liftSeq: FutureArrow[Seq[A], Seq[B]] =

FutureArrow[Seq[A], Seq[B]] { seqA =>

Future.collect(seqA.map(this))

}

/\*\*

\* Converts this FutureArrow to a FutureEffect, where the result value is ignored.

\*/

def asFutureEffect[A2 <: A]: FutureEffect[A2] =

FutureEffect(this.unit)

/\*\*

\* Combines this FutureArrow with another, producing one that applies both

\* in parallel, producing a tuple of their results.

\*/

def inParallel[A2 <: A, C](other: FutureArrow[A2, C]): FutureArrow[A2, (B, C)] = {

val paired = self.zipjoin(other)

FutureArrow[A2, (B, C)](a => paired((a, a)))

}

/\*\*

\* Wrap a FutureArrow with an ExceptionCounter, thus providing

\* observability into the arrow's success and failure.

\*/

def countExceptions(

exceptionCounter: ExceptionCounter

): FutureArrow[A, B] =

FutureArrow[A, B](request => exceptionCounter(self(request)))

/\*\*

\* Returns a chained FutureArrow in which the given function will be called for any

\* input that succeeds.

\*/

def onSuccess[A2 <: A](f: (A2, B) => Unit): FutureArrow[A2, B] =

FutureArrow[A2, B](a => self(a).onSuccess(b => f(a, b)))

/\*\*

\* Returns a chained FutureArrow in which the given function will be called for any

\* input that fails.

\*/

def onFailure[A2 <: A](f: (A2, Throwable) => Unit): FutureArrow[A2, B] =

FutureArrow[A2, B](a => self(a).onFailure(t => f(a, t)))

/\*\*

\* Translate exception returned by a FutureArrow according to a

\* PartialFunction.

\*/

def translateExceptions(

translateException: PartialFunction[Throwable, Throwable]

): FutureArrow[A, B] =

FutureArrow[A, B] { request =>

self(request).rescue {

case t if translateException.isDefinedAt(t) => Future.exception(translateException(t))

case t => Future.exception(t)

}

}

/\*\*

\* Apply a FutureArrow, lifting any non-Future exceptions thrown into

\* `Future.exception`s.

\*/

def liftExceptions: FutureArrow[A, B] =

FutureArrow[A, B] { request =>

// Flattening the Future[Future[Response]] is equivalent, but more concise

// than wrapping the arrow(request) call in a try/catch block that transforms

// the exception to a Future.exception, or at least was more concise before

// I added a four-line comment.

Future(self(request)).flatten

}

/\*\*

\* Wrap a FutureArrow in exception-tracking and -translation. Given a

\* filter and a handler, exceptional results will be observed and translated

\* according to the function passed in this function's second argument list.

\*/

def cleanly(

exceptionCounter: ExceptionCounter

)(

translateException: PartialFunction[Throwable, Throwable] = { case t => t }

): FutureArrow[A, B] = {

liftExceptions

.translateExceptions(translateException)

.countExceptions(exceptionCounter)

}

/\*\*

\* Produces a FutureArrow that tracks its own application latency.

\*/

@deprecated("use trackLatency(StatsReceiver, (A2 => String)", "2.11.1")

def trackLatency[A2 <: A](

extractName: (A2 => String),

statsReceiver: StatsReceiver

): FutureArrow[A2, B] =

trackLatency(statsReceiver, extractName)

/\*\*

\* Produces a FutureArrow that tracks its own application latency.

\*/

def trackLatency[A2 <: A](

statsReceiver: StatsReceiver,

extractName: (A2 => String)

): FutureArrow[A2, B] =

FutureArrow[A2, B] { request =>

Stat.timeFuture(statsReceiver.stat(extractName(request), "latency\_ms")) {

self(request)

}

}

/\*\*

\* Produces a FutureArrow that tracks the outcome (i.e. success vs failure) of

\* requests.

\*/

@deprecated("use trackOutcome(StatsReceiver, (A2 => String)", "2.11.1")

def trackOutcome[A2 <: A](

extractName: (A2 => String),

statsReceiver: StatsReceiver

): FutureArrow[A2, B] =

trackOutcome(statsReceiver, extractName)

def trackOutcome[A2 <: A](

statsReceiver: StatsReceiver,

extractName: (A2 => String)

): FutureArrow[A2, B] =

trackOutcome(statsReceiver, extractName, \_ => None)

/\*\*

\* Produces a FutureArrow that tracks the outcome (i.e. success vs failure) of

\* requests.

\*/

def trackOutcome[A2 <: A](

statsReceiver: StatsReceiver,

extractName: (A2 => String),

exceptionCategorizer: Throwable => Option[String]

): FutureArrow[A2, B] =

FutureArrow[A2, B] { request =>

val scope = statsReceiver.scope(extractName(request))

self(request).respond { r =>

statsReceiver.counter("requests").incr()

scope.counter("requests").incr()

r match {

case Return(\_) =>

statsReceiver.counter("success").incr()

scope.counter("success").incr()

case Throw(t) =>

val category = exceptionCategorizer(t).getOrElse("failures")

statsReceiver.counter(category).incr()

scope.counter(category).incr()

scope.scope(category).counter(ThrowableHelper.sanitizeClassnameChain(t): \_\*).incr()

}

}

}

/\*\*

\* Observe latency and success rate for any FutureArrow[A, B] where A is Observable

\*/

def observed[A2 <: A with Observable](

statsReceiver: StatsReceiver

): FutureArrow[A2, B] =

observed(statsReceiver, exceptionCategorizer = \_ => None)

/\*\*

\* Observe latency and success rate for any FutureArrow[A, B] where A is Observable

\*/

def observed[A2 <: A with Observable](

statsReceiver: StatsReceiver,

exceptionCategorizer: Throwable => Option[String]

): FutureArrow[A2, B] =

self.observed(

statsReceiver.scope("client\_request"),

(a: A2) => a.requestName,

exceptionCategorizer

)

/\*\*

\* Observe latency and success rate for any FutureArrow

\*/

def observed[A2 <: A](

statsReceiver: StatsReceiver,

statsScope: A2 => String,

exceptionCategorizer: Throwable => Option[String] = \_ => None

): FutureArrow[A2, B] =

self

.trackLatency(statsReceiver, statsScope)

.trackOutcome(statsReceiver, statsScope, exceptionCategorizer)

/\*\*

\* Trace the future arrow using local spans as documented here:

\* https://docbird.twitter.biz/finagle/Tracing.html

\*/

def traced[A2 <: A](

traceScope: A2 => String

): FutureArrow[A2, B] = {

FutureArrow[A2, B] { a =>

Trace.traceLocalFuture(traceScope(a))(self(a))

}

}

/\*\*

\* Produces a new FutureArrow where the given function is applied to the input, and the result

\* passed to this FutureArrow.

\*/

def contramap[C](f: C => A): FutureArrow[C, B] =

FutureArrow[C, B](f.andThen(self))

/\*\*

\* Produces a new FutureArrow where the given function is applied to the result of this

\* FutureArrow.

\*/

def map[C](f: B => C): FutureArrow[A, C] =

mapResult(\_.map(f))

/\*\*

\* Produces a new FutureArrow where the given function is applied to the resulting Future of

\* this FutureArrow.

\*/

def mapResult[C](f: Future[B] => Future[C]): FutureArrow[A, C] =

FutureArrow[A, C](a => f(self(a)))

/\*\*

\* Produces a new FutureArrow which translates exceptions into futures

\*/

def rescue[B2 >: B](

rescueException: PartialFunction[Throwable, Future[B2]]

): FutureArrow[A, B2] = {

FutureArrow[A, B2] { a =>

self(a).rescue(rescueException)

}

}

/\*\*

\* Produces a new FutureArrow where the result value is ignored, and Unit is returned.

\*/

def unit: FutureArrow[A, Unit] =

mapResult(\_.unit)

/\*\*

\* Returns a copy of this FutureArrow where the returned Future has its `.masked`

\* method called.

\*/

def masked: FutureArrow[A, B] =

mapResult(\_.masked)

/\*\*

\* Wraps this FutureArrow by passing the underlying operation to the given retry handler

\* for possible retries.

\*/

def retry(handler: RetryHandler[B]): FutureArrow[A, B] =

FutureArrow[A, B](a => handler(self(a)))

def retry[A2 <: A](

policy: RetryPolicy[Try[B]],

timer: Timer,

statsReceiver: StatsReceiver,

extractName: (A2 => String)

): FutureArrow[A2, B] =

FutureArrow[A2, B] { a =>

val scoped = statsReceiver.scope(extractName(a))

RetryHandler(policy, timer, scoped)(self(a))

}

/\*\*

\* Produces a new FutureArrow where the returned Future[B] must complete within the specified

\* timeout, otherwise the Future fails with a com.twitter.util.TimeoutException.

\*

\* The [[timeout]] is passed by name to take advantage of deadlines passed in the request context.

\*

\* ''Note'': On timeout, the underlying future is NOT interrupted.

\*/

def withTimeout(timer: Timer, timeout: => Duration): FutureArrow[A, B] =

mapResult(\_.within(timer, timeout))

/\*\*

\* Produces a new FutureArrow where the returned Future must complete within the specified

\* timeout, otherwise the Future fails with the specified Throwable.

\*

\* The [[timeout]] is passed by name to take advantage of deadlines passed in the request context.

\*

\* ''Note'': On timeout, the underlying future is NOT interrupted.

\*/

def withTimeout(timer: Timer, timeout: => Duration, exc: => Throwable): FutureArrow[A, B] =

mapResult(\_.within(timer, timeout, exc))

/\*\*

\* Produces a new FutureArrow where the returned Future[B] must complete within the specified

\* timeout, otherwise the Future fails with a com.twitter.util.TimeoutException.

\*

\* The [[timeout]] is passed by name to take advantage of deadlines passed in the request context.

\*

\* ''Note'': On timeout, the underlying future is interrupted.

\*/

def raiseWithin(timer: Timer, timeout: => Duration): FutureArrow[A, B] =

mapResult(\_.raiseWithin(timeout)(timer))

/\*\*

\* Produces a new FutureArrow where the returned Future must complete within the specified

\* timeout, otherwise the Future fails with the specified Throwable.

\*

\* [[timeout]] is passed by name to take advantage of deadlines passed in the request context.

\*

\* ''Note'': On timeout, the underlying future is interrupted.

\*/

def raiseWithin(timer: Timer, timeout: => Duration, exc: => Throwable): FutureArrow[A, B] =

mapResult(\_.raiseWithin(timer, timeout, exc))

/\*\*

\* Produces a finagle.Service instance that invokes this arrow.

\*/

def asService: Service[A, B] = Service.mk(this)

/\*\*

\* Produces a new FutureArrow with the given finagle.Filter applied to this instance.

\*/

def withFilter[A2, B2](filter: Filter[A2, B2, A, B]): FutureArrow[A2, B2] =

FutureArrow[A2, B2](filter.andThen(asService))

/\*\*

\* Produces a new FutureArrow with the given timeout which retries on Exceptions or timeouts and

\* records stats about the logical request. This is only appropriate for idempotent operations.

\*/

def observedWithTimeoutAndRetry[A2 <: A](

statsReceiver: StatsReceiver,

extractName: (A2 => String),

timer: Timer,

timeout: Duration,

numTries: Int,

shouldRetry: PartialFunction[Try[B], Boolean] = FutureArrow.RetryOnNonFailedFast

): FutureArrow[A2, B] = {

val retryPolicy = RetryPolicy.tries(numTries, shouldRetry)

withTimeout(timer, timeout)

.retry(retryPolicy, timer, statsReceiver, extractName)

.trackLatency(statsReceiver, extractName)

.trackOutcome(statsReceiver, extractName)

}

/\*\*

\* Produces a new FutureArrow with the given timeout and records stats about the logical request.

\* This does not retry and is appropriate for non-idempotent operations.

\*/

def observedWithTimeout[A2 <: A](

statsReceiver: StatsReceiver,

extractName: (A2 => String),

timer: Timer,

timeout: Duration

): FutureArrow[A2, B] =

withTimeout(timer, timeout)

.trackLatency(statsReceiver, extractName)

.trackOutcome(statsReceiver, extractName)

}