package com.twitter.servo.util

import com.twitter.finagle.stats.{StatsReceiver, Stat}

import com.twitter.logging.{Logger, NullLogger}

import com.twitter.util.\_

object FutureEffect {

private[this] val \_unit = FutureEffect[Any] { \_ =>

Future.Unit

}

/\*\*

\* A FutureEffect that always succeeds.

\*/

def unit[T]: FutureEffect[T] =

\_unit.asInstanceOf[FutureEffect[T]]

/\*\*

\* A FutureEffect that always fails with the given exception.

\*/

def fail[T](ex: Throwable): FutureEffect[T] =

FutureEffect[T] { \_ =>

Future.exception(ex)

}

/\*\*

\* Lift a function returning a Future to a FutureEffect.

\*/

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

new FutureEffect[T] {

override def apply(x: T) = f(x)

}

/\*\*

\* Performs all of the effects in order. If any effect fails, the

\* whole operation fails, and the subsequent effects are not

\* attempted.

\*/

def sequentially[T](effects: FutureEffect[T]\*): FutureEffect[T] =

effects.foldLeft[FutureEffect[T]](unit[T])(\_ andThen \_)

/\*\*

\* Perform all of the effects concurrently. If any effect fails, the

\* whole operation fails, but any of the effects may or may not have

\* taken place.

\*/

def inParallel[T](effects: FutureEffect[T]\*): FutureEffect[T] =

FutureEffect[T] { t =>

Future.join(effects map { \_(t) })

}

def fromPartial[T](f: PartialFunction[T, Future[Unit]]) =

FutureEffect[T] { x =>

if (f.isDefinedAt(x)) f(x) else Future.Unit

}

/\*\*

\* Combines two FutureEffects into one that dispatches according to a gate. If the gate is

\* true, use `a`, otherwise, use `b`.

\*/

def selected[T](condition: Gate[Unit], a: FutureEffect[T], b: FutureEffect[T]): FutureEffect[T] =

selected(() => condition(), a, b)

/\*\*

\* Combines two FutureEffects into one that dispatches according to a nullary boolean function.

\* If the function returns true, use `a`, otherwise, use `b`.

\*/

def selected[T](f: () => Boolean, a: FutureEffect[T], b: FutureEffect[T]): FutureEffect[T] =

FutureEffect[T] { t =>

if (f()) a(t) else b(t)

}

}

/\*\*

\* A function whose only result is a future effect. This wrapper

\* provides convenient combinators.

\*/

trait FutureEffect[T] extends (T => Future[Unit]) { self =>

/\*\*

\* Simplified version of `apply` when type is `Unit`.

\*/

def apply()(implicit ev: Unit <:< T): Future[Unit] = self(())

/\*\*

\* Combines two Future effects, performing this one first and

\* performing the next one if this one succeeds.

\*/

def andThen(next: FutureEffect[T]): FutureEffect[T] =

FutureEffect[T] { x =>

self(x) flatMap { \_ =>

next(x)

}

}

/\*\*

\* Wraps this FutureEffect with a failure handling function that will be chained to

\* the Future returned by this FutureEffect.

\*/

def rescue(

handler: PartialFunction[Throwable, FutureEffect[T]]

): FutureEffect[T] =

FutureEffect[T] { x =>

self(x) rescue {

case t if handler.isDefinedAt(t) =>

handler(t)(x)

}

}

/\*\*

\* Combines two future effects, performing them both simultaneously.

\* If either effect fails, the result will be failure, but the other

\* effects will have occurred.

\*/

def inParallel(other: FutureEffect[T]) =

FutureEffect[T] { x =>

Future.join(Seq(self(x), other(x)))

}

/\*\*

\* Perform this effect only if the provided gate returns true.

\*/

def enabledBy(enabled: Gate[Unit]): FutureEffect[T] =

enabledBy(() => enabled())

/\*\*

\* Perform this effect only if the provided gate returns true.

\*/

def enabledBy(enabled: () => Boolean): FutureEffect[T] =

onlyIf { \_ =>

enabled()

}

/\*\*

\* Perform this effect only if the provided predicate returns true

\* for the input.

\*/

def onlyIf(predicate: T => Boolean) =

FutureEffect[T] { x =>

if (predicate(x)) self(x) else Future.Unit

}

/\*\*

\* Perform this effect with arg only if the condition is true. Otherwise just return Future Unit

\*/

def when(condition: Boolean)(arg: => T): Future[Unit] =

if (condition) self(arg) else Future.Unit

/\*\*

\* Adapt this effect to take a different input via the provided conversion.

\*

\* (Contravariant map)

\*/

def contramap[U](g: U => T) = FutureEffect[U] { u =>

self(g(u))

}

/\*\*

\* Adapt this effect to take a different input via the provided conversion.

\*

\* (Contravariant map)

\*/

def contramapFuture[U](g: U => Future[T]) = FutureEffect[U] { u =>

g(u) flatMap self

}

/\*\*

\* Adapt this effect to take a different input via the provided conversion.

\* If the output value of the given function is None, the effect is a no-op.

\*/

def contramapOption[U](g: U => Option[T]) =

FutureEffect[U] {

g andThen {

case None => Future.Unit

case Some(t) => self(t)

}

}

/\*\*

\* Adapt this effect to take a different input via the provided conversion.

\* If the output value of the given function is future-None, the effect is a no-op.

\* (Contravariant map)

\*/

def contramapFutureOption[U](g: U => Future[Option[T]]) =

FutureEffect[U] { u =>

g(u) flatMap {

case None => Future.Unit

case Some(x) => self(x)

}

}

/\*\*

\* Adapt this effect to take a sequence of input values.

\*/

def liftSeq: FutureEffect[Seq[T]] =

FutureEffect[Seq[T]] { seqT =>

Future.join(seqT.map(self))

}

/\*\*

\* Allow the effect to fail, but immediately return success. The

\* effect is not guaranteed to have finished when its future is

\* available.

\*/

def ignoreFailures: FutureEffect[T] =

FutureEffect[T] { x =>

Try(self(x)); Future.Unit

}

/\*\*

\* Allow the effect to fail but always return success. Unlike ignoreFailures, the

\* effect is guaranteed to have finished when its future is available.

\*/

def ignoreFailuresUponCompletion: FutureEffect[T] =

FutureEffect[T] { x =>

Try(self(x)) match {

case Return(f) => f.handle { case \_ => () }

case Throw(\_) => Future.Unit

}

}

/\*\*

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

\* input that succeeds.

\*/

def onSuccess(f: T => Unit): FutureEffect[T] =

FutureEffect[T] { x =>

self(x).onSuccess(\_ => f(x))

}

/\*\*

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

\* input that fails.

\*/

def onFailure(f: (T, Throwable) => Unit): FutureEffect[T] =

FutureEffect[T] { x =>

self(x).onFailure(t => f(x, t))

}

/\*\*

\* Translate exception returned by a FutureEffect according to a

\* PartialFunction.

\*/

def translateExceptions(

translateException: PartialFunction[Throwable, Throwable]

): FutureEffect[T] =

FutureEffect[T] { request =>

self(request) rescue {

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

case t => Future.exception(t)

}

}

/\*\*

\* Wraps an effect with retry logic. Will retry against any failure.

\*/

def retry(backoffs: Stream[Duration], timer: Timer, stats: StatsReceiver): FutureEffect[T] =

retry(RetryHandler.failuresOnly(backoffs, timer, stats))

/\*\*

\* Returns a new FutureEffect that executes the effect within the given RetryHandler, which

\* may retry the operation on failures.

\*/

def retry(handler: RetryHandler[Unit]): FutureEffect[T] =

FutureEffect[T](handler.wrap(self))

@deprecated("use trackOutcome", "2.11.1")

def countExceptions(stats: StatsReceiver, getScope: T => String) = {

val exceptionCounterFactory = new MemoizedExceptionCounterFactory(stats)

FutureEffect[T] { t =>

exceptionCounterFactory(getScope(t)) { self(t) }

}

}

/\*\*

\* Produces a FutureEffect that tracks the latency of the underlying operation.

\*/

def trackLatency(stats: StatsReceiver, extractName: T => String): FutureEffect[T] =

FutureEffect[T] { t =>

Stat.timeFuture(stats.stat(extractName(t), "latency\_ms")) { self(t) }

}

def trackOutcome(

stats: StatsReceiver,

extractName: T => String,

logger: Logger = NullLogger

): FutureEffect[T] = trackOutcome(stats, extractName, logger, \_ => None)

/\*\*

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

\* requests, including counting exceptions by classname.

\*/

def trackOutcome(

stats: StatsReceiver,

extractName: T => String,

logger: Logger,

exceptionCategorizer: Throwable => Option[String]

): FutureEffect[T] =

FutureEffect[T] { t =>

val name = extractName(t)

val scope = stats.scope(name)

self(t) respond { r =>

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

r match {

case Return(\_) =>

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

case Throw(t) =>

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

scope.counter(category).incr()

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

logger.warning(t, s"failure in $name")

}

}

}

/\*\*

\* Observe latency and success rate for any FutureEffect

\* @param statsScope a function to produce a parent stats scope from the argument

\* to the FutureEffect

\* @param exceptionCategorizer a function to assign different Throwables with custom stats scopes.

\*/

def observed(

statsReceiver: StatsReceiver,

statsScope: T => String,

logger: Logger = NullLogger,

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

): FutureEffect[T] =

self

.trackLatency(statsReceiver, statsScope)

.trackOutcome(statsReceiver, statsScope, logger, exceptionCategorizer)

/\*\*

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

\* FutureEffect.

\*/

def mapResult(f: Future[Unit] => Future[Unit]): FutureEffect[T] =

FutureEffect[T] { x =>

f(self(x))

}

/\*\*

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

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

\*

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

\*/

def withTimeout(timer: Timer, timeout: Duration): FutureEffect[T] =

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

/\*\*

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

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

\*

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

\*/

def withTimeout(timer: Timer, timeout: Duration, exc: => Throwable): FutureEffect[T] =

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

/\*\*

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

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

\*

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

\*/

def raiseWithin(timer: Timer, timeout: Duration): FutureEffect[T] =

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

/\*\*

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

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

\*

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

\*/

def raiseWithin(timer: Timer, timeout: Duration, exc: => Throwable): FutureEffect[T] =

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

}