package com.twitter.servo.util

import com.twitter.finagle.Backoff

import com.twitter.finagle.service.{RetryBudget, RetryPolicy}

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

import com.twitter.util.\_

import java.util.concurrent.CancellationException

import scala.util.control.NonFatal

/\*\*

\* A RetryHandler can wrap an arbitrary Future-producing operation with retry logic, where the

\* operation may conditionally be retried multiple times.

\*/

trait RetryHandler[-A] {

/\*\*

\* Executes the given operation and performs any applicable retries.

\*/

def apply[A2 <: A](f: => Future[A2]): Future[A2]

/\*\*

\* Wraps an arbitrary function with this RetryHandler's retrying logic.

\*/

def wrap[A2 <: A, B](f: B => Future[A2]): B => Future[A2] =

b => this(f(b))

}

object RetryHandler {

/\*\*

\* Builds a RetryHandler that retries according to the given RetryPolicy. Retries, if any,

\* will be scheduled on the given Timer to be executed after the appropriate backoff, if any.

\* Retries will be limited according the given `RetryBudget`.

\*/

def apply[A](

policy: RetryPolicy[Try[A]],

timer: Timer,

statsReceiver: StatsReceiver,

budget: RetryBudget = RetryBudget()

): RetryHandler[A] = {

val firstTryCounter = statsReceiver.counter("first\_try")

val retriesCounter = statsReceiver.counter("retries")

val budgetExhausedCounter = statsReceiver.counter("budget\_exhausted")

new RetryHandler[A] {

def apply[A2 <: A](f: => Future[A2]): Future[A2] = {

firstTryCounter.incr()

budget.deposit()

retry[A2](policy, timer, retriesCounter, budgetExhausedCounter, budget)(f)

}

}

}

/\*\*

\* Builds a RetryHandler that will only retry on failures that are handled by the given policy,

\* and does not consider any successful future for retries.

\*/

def failuresOnly[A](

policy: RetryPolicy[Try[Nothing]],

timer: Timer,

statsReceiver: StatsReceiver,

budget: RetryBudget = RetryBudget()

): RetryHandler[A] =

apply(failureOnlyRetryPolicy(policy), timer, statsReceiver, budget)

/\*\*

\* Builds a RetryHandler that will retry any failure according to the given backoff schedule,

\* until either either the operation succeeds or all backoffs are exhausted.

\*/

def failuresOnly[A](

backoffs: Stream[Duration],

timer: Timer,

stats: StatsReceiver,

budget: RetryBudget

): RetryHandler[A] =

failuresOnly(

RetryPolicy.backoff[Try[Nothing]](Backoff.fromStream(backoffs)) { case Throw(\_) => true },

timer,

stats,

budget

)

/\*\*

\* Builds a RetryHandler that will retry any failure according to the given backoff schedule,

\* until either either the operation succeeds or all backoffs are exhausted.

\*/

def failuresOnly[A](

backoffs: Stream[Duration],

timer: Timer,

stats: StatsReceiver

): RetryHandler[A] =

failuresOnly(backoffs, timer, stats, RetryBudget())

/\*\*

\* Converts a RetryPolicy that only handles failures (Throw) to a RetryPolicy that also

\* handles successes (Return), by flagging that successes need not be retried.

\*/

def failureOnlyRetryPolicy[A](policy: RetryPolicy[Try[Nothing]]): RetryPolicy[Try[A]] =

RetryPolicy[Try[A]] {

case Return(\_) => None

case Throw(ex) =>

policy(Throw(ex)) map {

case (backoff, p2) => (backoff, failureOnlyRetryPolicy(p2))

}

}

private[this] def retry[A](

policy: RetryPolicy[Try[A]],

timer: Timer,

retriesCounter: Counter,

budgetExhausedCounter: Counter,

budget: RetryBudget

)(

f: => Future[A]

): Future[A] = {

forceFuture(f).transform { transformed =>

policy(transformed) match {

case Some((backoff, nextPolicy)) =>

if (budget.tryWithdraw()) {

retriesCounter.incr()

schedule(backoff, timer) {

retry(nextPolicy, timer, retriesCounter, budgetExhausedCounter, budget)(f)

}

} else {

budgetExhausedCounter.incr()

Future.const(transformed)

}

case None =>

Future.const(transformed)

}

}

}

// similar to finagle's RetryExceptionsFilter

private[this] def schedule[A](d: Duration, timer: Timer)(f: => Future[A]) = {

if (d.inNanoseconds > 0) {

val promise = new Promise[A]

val task = timer.schedule(Time.now + d) {

if (!promise.isDefined) {

try {

promise.become(f)

} catch {

case NonFatal(cause) =>

// Ignore any exceptions thrown by Promise#become(). This usually means that the promise

// was already defined and cannot be transformed.

}

}

}

promise.setInterruptHandler {

case cause =>

task.cancel()

val cancellation = new CancellationException

cancellation.initCause(cause)

promise.updateIfEmpty(Throw(cancellation))

}

promise

} else forceFuture(f)

}

// (Future { f } flatten), but without the allocation

private[this] def forceFuture[A](f: => Future[A]) = {

try {

f

} catch {

case NonFatal(cause) =>

Future.exception(cause)

}

}

}