package com.twitter.tweetypie

package backends

import com.twitter.finagle.context.Deadline

import com.twitter.finagle.service.RetryBudget

import com.twitter.finagle.service.RetryPolicy

import com.twitter.servo.util.FutureArrow

import com.twitter.servo.util.RetryHandler

import com.twitter.tweetypie.core.OverCapacity

import com.twitter.util.Timer

import com.twitter.util.TimeoutException

object Backend {

val log: Logger = Logger(getClass)

/\*\*

\* Common stuff that is needed as part of the configuration of all

\* of the backends.

\*/

case class Context(val timer: Timer, val stats: StatsReceiver)

/\*\*

\* All backend operations are encapsulated in the FutureArrow type. The Builder type

\* represents functions that can decorate the FutureArrow, typically by calling the various

\* combinator methods on FutureArrow.

\*/

type Builder[A, B] = FutureArrow[A, B] => FutureArrow[A, B]

/\*\*

\* A Policy defines some behavior to apply to a FutureArrow that wraps an endpoint.

\*/

trait Policy {

/\*\*

\* Using an endpoint name and Context, returns a Builder that does the actual

\* application of the policy to the FutureArrow.

\*/

def apply[A, B](name: String, ctx: Context): Builder[A, B]

/\*\*

\* Sequentially combines policies, first applying this policy and then applying

\* the next policy. Order matters! For example, to retry on timeouts, the FailureRetryPolicy

\* needs to be applied after the TimeoutPolicy:

\*

\* TimeoutPolicy(100.milliseconds) >>> FailureRetryPolicy(retryPolicy)

\*/

def andThen(next: Policy): Policy = {

val first = this

new Policy {

def apply[A, B](name: String, ctx: Context): Builder[A, B] =

first(name, ctx).andThen(next(name, ctx))

override def toString = s"$first >>> $next"

}

}

/\*\*

\* An alias for `andThen`.

\*/

def >>>(next: Policy): Policy = andThen(next)

}

/\*\*

\* Applies a timeout to the underlying FutureArrow.

\*/

case class TimeoutPolicy(timeout: Duration) extends Policy {

def apply[A, B](name: String, ctx: Context): Builder[A, B] = {

val stats = ctx.stats.scope(name)

val ex = new TimeoutException(name + ": " + timeout)

(\_: FutureArrow[A, B]).raiseWithin(ctx.timer, timeout, ex)

}

}

/\*\*

\* Attaches a RetryHandler with the given RetryPolicy to retry failures.

\*/

case class FailureRetryPolicy(

retryPolicy: RetryPolicy[Try[Nothing]],

retryBudget: RetryBudget = RetryBudget())

extends Policy {

def apply[A, B](name: String, ctx: Context): Builder[A, B] = {

val stats = ctx.stats.scope(name)

(\_: FutureArrow[A, B])

.retry(RetryHandler.failuresOnly(retryPolicy, ctx.timer, stats, retryBudget))

}

}

/\*\*

\* This policy applies standardized endpoint metrics. This should be used with every endpoint.

\*/

case object TrackPolicy extends Policy {

def apply[A, B](name: String, ctx: Context): Builder[A, B] = {

val stats = ctx.stats.scope(name)

(\_: FutureArrow[A, B])

.onFailure(countOverCapacityExceptions(stats))

.trackOutcome(ctx.stats, (\_: A) => name)

.trackLatency(ctx.stats, (\_: A) => name)

}

}

/\*\*

\* The default "policy" for timeouts, retries, exception counting, latency tracking, etc. to

\* apply to each backend operation. This returns a Builder type (an endofunction on FutureArrow),

\* which can be composed with other Builders via simple function composition.

\*/

def defaultPolicy[A, B](

name: String,

requestTimeout: Duration,

retryPolicy: RetryPolicy[Try[B]],

ctx: Context,

retryBudget: RetryBudget = RetryBudget(),

totalTimeout: Duration = Duration.Top,

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

): Builder[A, B] = {

val scopedStats = ctx.stats.scope(name)

val requestTimeoutException = new TimeoutException(

s"$name: hit request timeout of $requestTimeout"

)

val totalTimeoutException = new TimeoutException(s"$name: hit total timeout of $totalTimeout")

base =>

base

.raiseWithin(

ctx.timer,

// We defer to a per-request deadline. When the deadline is missing or wasn't toggled,

// 'requestTimeout' is used instead. This mimics the behavior happening within a standard

// Finagle client stack and its 'TimeoutFilter'.

Deadline.currentToggled.fold(requestTimeout)(\_.remaining),

requestTimeoutException

)

.retry(RetryHandler(retryPolicy, ctx.timer, scopedStats, retryBudget))

.raiseWithin(ctx.timer, totalTimeout, totalTimeoutException)

.onFailure(countOverCapacityExceptions(scopedStats))

.trackOutcome(ctx.stats, (\_: A) => name, exceptionCategorizer)

.trackLatency(ctx.stats, (\_: A) => name)

}

/\*\*

\* An onFailure FutureArrow callback that counts OverCapacity exceptions to a special counter.

\* These will also be counted as failures and by exception class name, but having a special

\* counter for this is easier to use in success rate computations where you want to factor out

\* backpressure responses.

\*/

def countOverCapacityExceptions[A](scopedStats: StatsReceiver): (A, Throwable) => Unit = {

val overCapacityCounter = scopedStats.counter("over\_capacity")

{

case (\_, ex: OverCapacity) => overCapacityCounter.incr()

case \_ => ()

}

}

/\*\*

\* Provides a simple mechanism for applying a Policy to an endpoint FutureArrow from

\* an underlying service interface.

\*/

class PolicyAdvocate[S](backendName: String, ctx: Backend.Context, svc: S) {

/\*\*

\* Tacks on the TrackPolicy to the given base policy, and then applies the policy to

\* a FutureArrow. This is more of a convenience method that every Backend can use to

\* build the fully configured FutureArrow.

\*/

def apply[A, B](

endpointName: String,

policy: Policy,

endpoint: S => FutureArrow[A, B]

): FutureArrow[A, B] = {

log.info(s"appling policy to $backendName.$endpointName: $policy")

policy.andThen(TrackPolicy)(endpointName, ctx)(endpoint(svc))

}

}

}