package com.twitter.tweetypie

package config

import com.twitter.conversions.DurationOps.\_

import com.twitter.finagle.Backoff

import com.twitter.finagle.memcached.exp.localMemcachedPort

import com.twitter.finagle.mtls.authentication.ServiceIdentifier

import com.twitter.finagle.ssl.OpportunisticTls

import com.twitter.finagle.thrift.ClientId

import com.twitter.flockdb.client.thriftscala.Priority

import com.twitter.servo.repository.CachedResult

import com.twitter.servo.util.Availability

import com.twitter.tweetypie.backends.\_

import com.twitter.tweetypie.caching.SoftTtl

import com.twitter.tweetypie.handler.DuplicateTweetFinder

import com.twitter.tweetypie.repository.TombstoneTtl

import com.twitter.tweetypie.service.\_

import com.twitter.tweetypie.storage.ManhattanTweetStorageClient

import com.twitter.util.Duration

case class InProcessCacheConfig(ttl: Duration, maximumSize: Int)

class TweetServiceSettings(val flags: TweetServiceFlags) {

/\*\*

\* Convert a Boolean to an Option

\* > optional(true, "my value")

\* res: Some(my value)

\*

\* > optional(false, "my value")

\* res: None

\*/

def optional[T](b: Boolean, a: => T): Option[T] = if (b) Some(a) else None

/\*\* atla, localhost, etc. \*/

val zone: String = flags.zone()

/\*\* dc is less specific than zone, zone=atla, dc=atl \*/

val dc: String = zone.dropRight(1)

/\*\* one of: prod, staging, dev, testbox \*/

val env: Env.Value = flags.env()

/\*\* instanceId of this aurora instance \*/

lazy val instanceId: Int = flags.instanceId()

/\*\* total number of tweetypie aurora instances \*/

val instanceCount: Int = flags.instanceCount()

/\*\* The Name to resolve to find the memcached cluster \*/

val twemcacheDest: String =

// If twemcacheDest is explicitly set, always prefer that to

// localMemcachedPort.

flags.twemcacheDest.get

// Testbox uses this global flag to specify the location of the

// local memcached instance.

.orElse(localMemcachedPort().map("/$/inet/localhost/" + \_))

// If no explicit Name is specified, use the default.

.getOrElse(flags.twemcacheDest())

/\*\* Read/write data through Cache \*/

val withCache: Boolean = flags.withCache()

/\*\*

\* The TFlock queue to use for background indexing operations. For

\* production, this should always be the low priority queue, to

\* allow foreground operations to be processed first.

\*/

val backgroundIndexingPriority: Priority = flags.backgroundIndexingPriority()

/\*\* Set certain decider gates to this overridden value \*/

val deciderOverrides: Map[String, Boolean] =

flags.deciderOverrides()

/\*\* use per host stats? \*/

val clientHostStats: Boolean =

flags.clientHostStats()

val warmupRequestsSettings: Option[WarmupQueriesSettings] =

optional(flags.enableWarmupRequests(), WarmupQueriesSettings())

/\*\* enables request authorization via a allowlist \*/

val allowlistingRequired: Boolean =

flags.allowlist.get.getOrElse(env == Env.prod)

/\*\* read rate limit for unknown clients (when allowlistingRequired is enabled) \*/

val nonAllowListedClientRateLimitPerSec: Double =

flags.grayListRateLimit()

/\*\* enables requests from production clients \*/

val allowProductionClients: Boolean =

env == Env.prod

/\*\* enables replication via DRPC \*/

val enableReplication: Boolean = flags.enableReplication()

/\*\* enables forking of some traffic to configured target \*/

val trafficForkingEnabled: Boolean =

env == Env.prod

val scribeUniquenessIds: Boolean =

env == Env.prod

/\*\* ClientId to send to backend services \*/

val thriftClientId: ClientId =

flags.clientId.get.map(ClientId(\_)).getOrElse {

env match {

case Env.dev | Env.staging => ClientId("tweetypie.staging")

case Env.prod => ClientId("tweetypie.prod")

}

}

/\*\*

\* Instead of using DRPC for calling into the async code path, call back into the

\* current instance. Used for development and test to ensure logic in the current

\* instance is being tested.

\*/

val simulateDeferredrpcCallbacks: Boolean = flags.simulateDeferredrpcCallbacks()

/\*\*

\* ClientId to set in 'asynchronous' requests when simulateDeferredrpcCallbacks is

\* true and Tweetypie ends up just calling itself synchronously.

\*/

val deferredrpcClientId: ClientId = ClientId("deferredrpc.prod")

/\*\*

\* ServiceIdentifier used to enable mTLS

\*/

val serviceIdentifier: ServiceIdentifier = flags.serviceIdentifier()

/\*\*

\* Decider settings

\*/

val deciderBaseFilename: Option[String] = Option(flags.deciderBase())

val deciderOverlayFilename: Option[String] = Option(flags.deciderOverlay())

val vfDeciderOverlayFilename: Option[String] = flags.vfDeciderOverlay.get

/\*\*

\* Used to determine whether we should fail requests for Tweets that are likely too young

\* to return a non-partial response. We return NotFound for Tweets that are deemed too young.

\* Used by [[com.twitter.tweetypie.repository.ManhattanTweetRepository]].

\*/

val shortCircuitLikelyPartialTweetReads: Gate[Duration] = {

// interpret the flag as a duration in milliseconds

val ageCeiling: Duration = flags.shortCircuitLikelyPartialTweetReadsMs().milliseconds

Gate(tweetAge => tweetAge < ageCeiling)

}

// tweet-service internal settings

val tweetKeyCacheVersion = 1

/\*\* how often to flush aggregated count updates for tweet counts \*/

val aggregatedTweetCountsFlushInterval: Duration = 5.seconds

/\*\* maximum number of keys for which aggregated cached count updates may be cached \*/

val maxAggregatedCountsSize = 1000

/\*\* ramp up period for decidering up forked traffic (if enabled) to the full decidered value \*/

val forkingRampUp: Duration = 3.minutes

/\*\* how long to wait after startup for serversets to resolve before giving up and moving on \*/

val waitForServerSetsTimeout: Duration = 120.seconds

/\*\* number of threads to use in thread pool for language identification \*/

val numPenguinThreads = 4

/\*\* maximum number of tweets that clients can request per getTweets RPC call \*/

val maxGetTweetsRequestSize = 200

/\*\* maximum batch size for any batched request (getTweets is exempt, it has its own limiting) \*/

val maxRequestSize = 200

/\*\*

\* maximum size to allow the thrift response buffer to grow before resetting it. this is set to

\* approximately the current value of `srv/thrift/response\_payload\_bytes.p999`, meaning roughly

\* 1 out of 1000 requests will cause the buffer to be reset.

\*/

val maxThriftBufferSize: Int = 200 \* 1024

// \*\*\*\*\*\*\*\*\* timeouts and backoffs \*\*\*\*\*\*\*\*\*\*

/\*\* backoffs for OptimisticLockingCache lockAndSet operations \*/

val lockingCacheBackoffs: Stream[Duration] =

Backoff.exponentialJittered(10.millisecond, 50.milliseconds).take(3).toStream

/\*\* retry once on timeout with no backoff \*/

val defaultTimeoutBackoffs: Stream[Duration] = Stream(0.milliseconds).toStream

/\*\* backoffs when user view is missing \*/

val gizmoduckMissingUserViewBackoffs: Stream[Duration] = Backoff.const(10.millis).take(3).toStream

/\*\* backoffs for retrying failed async-write actions after first retry failure \*/

val asyncWriteRetryBackoffs: Stream[Duration] =

Backoff.exponential(10.milliseconds, 2).take(9).toStream.map(\_ min 1.second)

/\*\* backoffs for retrying failed deferredrpc enqueues \*/

val deferredrpcBackoffs: Stream[Duration] =

Backoff.exponential(10.milliseconds, 2).take(3).toStream

/\*\* backoffs for retrying failed cache updates for replicated events \*/

val replicatedEventCacheBackoffs: Stream[Duration] =

Backoff.exponential(100.milliseconds, 2).take(10).toStream

val escherbirdConfig: Escherbird.Config =

Escherbird.Config(

requestTimeout = 200.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs

)

val expandodoConfig: Expandodo.Config =

Expandodo.Config(

requestTimeout = 300.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs,

serverErrorBackoffs = Backoff.const(0.millis).take(3).toStream

)

val creativesContainerServiceConfig: CreativesContainerService.Config =

CreativesContainerService.Config(

requestTimeout = 300.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs,

serverErrorBackoffs = Backoff.const(0.millis).take(3).toStream

)

val geoScrubEventStoreConfig: GeoScrubEventStore.Config =

GeoScrubEventStore.Config(

read = GeoScrubEventStore.EndpointConfig(

requestTimeout = 200.milliseconds,

maxRetryCount = 1

),

write = GeoScrubEventStore.EndpointConfig(

requestTimeout = 1.second,

maxRetryCount = 1

)

)

val gizmoduckConfig: Gizmoduck.Config =

Gizmoduck.Config(

readTimeout = 300.milliseconds,

writeTimeout = 300.milliseconds,

// We bump the timeout value to 800ms because modifyAndGet is called only in async request path in GeoScrub daemon

// and we do not expect sync/realtime apps calling this thrift method

modifyAndGetTimeout = 800.milliseconds,

modifyAndGetTimeoutBackoffs = Backoff.const(0.millis).take(3).toStream,

defaultTimeoutBackoffs = defaultTimeoutBackoffs,

gizmoduckExceptionBackoffs = Backoff.const(0.millis).take(3).toStream

)

val limiterBackendConfig: LimiterBackend.Config =

LimiterBackend.Config(

requestTimeout = 300.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs

)

val mediaInfoServiceConfig: MediaInfoService.Config =

MediaInfoService.Config(

requestTimeout = 300.milliseconds,

totalTimeout = 500.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs

)

val scarecrowConfig: Scarecrow.Config =

Scarecrow.Config(

readTimeout = 100.milliseconds,

writeTimeout = 400.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs,

scarecrowExceptionBackoffs = Backoff.const(0.millis).take(3).toStream

)

val socialGraphSeviceConfig: SocialGraphService.Config =

SocialGraphService.Config(

socialGraphTimeout = 250.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs

)

val talonConfig: Talon.Config =

Talon.Config(

shortenTimeout = 500.milliseconds,

expandTimeout = 150.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs,

transientErrorBackoffs = Backoff.const(0.millis).take(3).toStream

)

/\*\*

\* page size when retrieving tflock pages for tweet deletion and undeletion

\* tweet erasures have their own page size eraseUserTweetsPageSize

\*/

val tflockPageSize: Int = flags.tflockPageSize()

val tflockReadConfig: TFlock.Config =

TFlock.Config(

requestTimeout = 300.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs,

flockExceptionBackoffs = Backoff.const(0.millis).take(3).toStream,

overCapacityBackoffs = Stream.empty,

defaultPageSize = tflockPageSize

)

val tflockWriteConfig: TFlock.Config =

TFlock.Config(

requestTimeout = 400.milliseconds,

timeoutBackoffs = defaultTimeoutBackoffs,

flockExceptionBackoffs = Backoff.const(0.millis).take(3).toStream,

overCapacityBackoffs = Backoff.exponential(10.millis, 2).take(3).toStream

)

val timelineServiceConfig: TimelineService.Config = {

val tlsExceptionBackoffs = Backoff.const(0.millis).take(3).toStream

TimelineService.Config(

writeRequestPolicy =

Backend.TimeoutPolicy(4.seconds) >>>

TimelineService.FailureBackoffsPolicy(

timeoutBackoffs = defaultTimeoutBackoffs,

tlsExceptionBackoffs = tlsExceptionBackoffs

),

readRequestPolicy =

Backend.TimeoutPolicy(400.milliseconds) >>>

TimelineService.FailureBackoffsPolicy(

timeoutBackoffs = defaultTimeoutBackoffs,

tlsExceptionBackoffs = tlsExceptionBackoffs

)

)

}

val tweetStorageConfig: ManhattanTweetStorageClient.Config = {

val remoteZone = zone match {

case "atla" => "pdxa"

case "pdxa" => "atla"

case "atla" | "localhost" => "atla"

case \_ =>

throw new IllegalArgumentException(s"Cannot configure remote DC for unknown zone '$zone'")

}

ManhattanTweetStorageClient.Config(

applicationId = "tbird\_mh",

localDestination = "/s/manhattan/cylon.native-thrift",

localTimeout = 290.milliseconds,

remoteDestination = s"/srv#/prod/$remoteZone/manhattan/cylon.native-thrift",

remoteTimeout = 1.second,

maxRequestsPerBatch = 25,

serviceIdentifier = serviceIdentifier,

opportunisticTlsLevel = OpportunisticTls.Required

)

}

val userImageServiceConfig: UserImageService.Config =

UserImageService.Config(

processTweetMediaTimeout = 5.seconds,

updateTweetMediaTimeout = 2.seconds,

timeoutBackoffs = defaultTimeoutBackoffs

)

val adsLoggingClientTopicName = env match {

case Env.prod => "ads\_client\_callback\_prod"

case Env.dev | Env.staging => "ads\_client\_callback\_staging"

}

/\*\* Delay between successive cascadedDeleteTweet calls when deleting retweets. Applied via decider. \*/

val retweetDeletionDelay: Duration = 20.milliseconds

/\*\*

\* Delay to sleep before each tweet deletion of an eraseUserTweets request.

\* This is a simple rate limiting mechanism. The long term solution is

\* to move async endpoints like user erasures and retweet deletions out

\* of the the main tweetypie cluster and into an async cluster with first class

\* rate limiting support

\*/

val eraseUserTweetsDelay: Duration = 100.milliseconds

val eraseUserTweetsPageSize = 100

val getStoredTweetsByUserPageSize = 20

val getStoredTweetsByUserMaxPages = 30

// \*\*\*\*\*\*\*\*\* ttls \*\*\*\*\*\*\*\*\*\*

// Unfortunately, this tombstone TTL applies equally to the case

// where the tweet was deleted and the case that the tweet does not

// exist or is unavailable. If we could differentiate between those

// cases, we'd cache deleted for a long time and not

// found/unavailable for a short time. We chose 100

// milliseconds for the minimum TTL because there are known cases in

// which a not found result can be erroneously written to cache on

// tweet creation. This minimum TTL is a trade-off between a

// thundering herd of database requests from clients that just got

// the fanned-out tweet and the window for which these inconsistent

// results will be available.

val tweetTombstoneTtl: CachedResult.CachedNotFound[TweetId] => Duration =

TombstoneTtl.linear(min = 100.milliseconds, max = 1.day, from = 5.minutes, to = 5.hours)

val tweetMemcacheTtl: Duration = 14.days

val urlMemcacheTtl: Duration = 1.hour

val urlMemcacheSoftTtl: Duration = 1.hour

val deviceSourceMemcacheTtl: Duration = 12.hours

val deviceSourceMemcacheSoftTtl: SoftTtl.ByAge[Nothing] =

SoftTtl.ByAge(softTtl = 1.hour, jitter = 1.minute)

val deviceSourceInProcessTtl: Duration = 8.hours

val deviceSourceInProcessSoftTtl: Duration = 30.minutes

val placeMemcacheTtl: Duration = 1.day

val placeMemcacheSoftTtl: SoftTtl.ByAge[Nothing] =

SoftTtl.ByAge(softTtl = 3.hours, jitter = 1.minute)

val cardMemcacheTtl: Duration = 20.minutes

val cardMemcacheSoftTtl: Duration = 30.seconds

val tweetCreateLockingMemcacheTtl: Duration = 10.seconds

val tweetCreateLockingMemcacheLongTtl: Duration = 12.hours

val geoScrubMemcacheTtl: Duration = 30.minutes

val tweetCountsMemcacheTtl: Duration = 24.hours

val tweetCountsMemcacheNonZeroSoftTtl: Duration = 3.hours

val tweetCountsMemcacheZeroSoftTtl: Duration = 7.hours

val cacheClientPendingRequestLimit: Int = flags.memcachePendingRequestLimit()

val deviceSourceInProcessCacheMaxSize = 10000

val inProcessCacheConfigOpt: Option[InProcessCacheConfig] =

if (flags.enableInProcessCache()) {

Some(

InProcessCacheConfig(

ttl = flags.inProcessCacheTtlMs().milliseconds,

maximumSize = flags.inProcessCacheSize()

)

)

} else {

None

}

// Begin returning OverCapacity for tweet repo when cache SR falls below 95%,

// Scale to rejecting 95% of requests when cache SR <= 80%

val tweetCacheAvailabilityFromSuccessRate: Double => Double =

Availability.linearlyScaled(0.95, 0.80, 0.05)

// \*\*\*\*\*\*\* repository chunking size \*\*\*\*\*\*\*\*

val tweetCountsRepoChunkSize = 6

// n times `tweetCountsRepoChunkSize`, so chunking at higher level does not

// generate small batches at lower level.

val tweetCountsCacheChunkSize = 18

val duplicateTweetFinderSettings: DuplicateTweetFinder.Settings =

DuplicateTweetFinder.Settings(numTweetsToCheck = 10, maxDuplicateAge = 12.hours)

val backendWarmupSettings: Warmup.Settings =

Warmup.Settings(

// Try for twenty seconds to warm up the backends before giving

// up.

maxWarmupDuration = 20.seconds,

// Only allow up to 50 outstanding warmup requests of any kind

// to be outstanding at a time.

maxOutstandingRequests = 50,

// These timeouts are just over the p999 latency observed in ATLA

// for requests to these backends.

requestTimeouts = Map(

"expandodo" -> 120.milliseconds,

"geo\_relevance" -> 50.milliseconds,

"gizmoduck" -> 200.milliseconds,

"memcache" -> 50.milliseconds,

"scarecrow" -> 120.milliseconds,

"socialgraphservice" -> 180.milliseconds,

"talon" -> 70.milliseconds,

"tflock" -> 320.milliseconds,

"timelineservice" -> 200.milliseconds,

"tweetstorage" -> 50.milliseconds

),

reliability = Warmup.Reliably(

// Consider a backend warmed up if 99% of requests are succeeding.

reliabilityThreshold = 0.99,

// When performing warmup, use a maximum of 10 concurrent

// requests to each backend.

concurrency = 10,

// Do not allow more than this many attempts to perform the

// warmup action before giving up.

maxAttempts = 1000

)

)

}