package com.twitter.visibility.interfaces.conversations

import com.twitter.decider.Decider

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.gizmoduck.thriftscala.Label

import com.twitter.servo.repository.KeyValueResult

import com.twitter.servo.util.Gate

import com.twitter.spam.rtf.thriftscala.SafetyLabel

import com.twitter.spam.rtf.thriftscala.SafetyLabelType

import com.twitter.spam.rtf.thriftscala.SafetyLabelValue

import com.twitter.stitch.Stitch

import com.twitter.util.Future

import com.twitter.util.Return

import com.twitter.util.Stopwatch

import com.twitter.util.Try

import com.twitter.visibility.VisibilityLibrary

import com.twitter.visibility.builder.tweets.TweetIdFeatures

import com.twitter.visibility.builder.FeatureMapBuilder

import com.twitter.visibility.builder.VerdictLogger

import com.twitter.visibility.builder.VisibilityResult

import com.twitter.visibility.builder.tweets.FosnrPefetchedLabelsRelationshipFeatures

import com.twitter.visibility.builder.users.AuthorFeatures

import com.twitter.visibility.common.UserRelationshipSource

import com.twitter.visibility.common.UserSource

import com.twitter.visibility.configapi.configs.VisibilityDeciderGates

import com.twitter.visibility.features.AuthorUserLabels

import com.twitter.visibility.features.ConversationRootAuthorIsVerified

import com.twitter.visibility.features.FeatureMap

import com.twitter.visibility.features.HasInnerCircleOfFriendsRelationship

import com.twitter.visibility.features.TweetConversationId

import com.twitter.visibility.features.TweetParentId

import com.twitter.visibility.logging.thriftscala.VFLibType

import com.twitter.visibility.models.ContentId.TweetId

import com.twitter.visibility.models.SafetyLevel.TimelineConversationsDownranking

import com.twitter.visibility.models.SafetyLevel.TimelineConversationsDownrankingMinimal

import com.twitter.visibility.models.SafetyLevel.toThrift

import com.twitter.visibility.models.ContentId

import com.twitter.visibility.models.SafetyLevel

import com.twitter.visibility.models.TweetSafetyLabel

import com.twitter.visibility.models.UnitOfDiversion

object TimelineConversationsVisibilityLibrary {

type Type =

TimelineConversationsVisibilityRequest => Stitch[TimelineConversationsVisibilityResponse]

def apply(

visibilityLibrary: VisibilityLibrary,

batchSafetyLabelRepository: BatchSafetyLabelRepository,

decider: Decider,

userRelationshipSource: UserRelationshipSource = UserRelationshipSource.empty,

userSource: UserSource = UserSource.empty

): Type = {

val libraryStatsReceiver = visibilityLibrary.statsReceiver

val tweetIdFeatures = new TweetIdFeatures(

statsReceiver = libraryStatsReceiver,

enableStitchProfiling = Gate.False

)

val tweetIdFeaturesMinimal = new TweetIdFeatures(

statsReceiver = libraryStatsReceiver,

enableStitchProfiling = Gate.False

)

val vfLatencyOverallStat = libraryStatsReceiver.stat("vf\_latency\_overall")

val vfLatencyStitchBuildStat = libraryStatsReceiver.stat("vf\_latency\_stitch\_build")

val vfLatencyStitchRunStat = libraryStatsReceiver.stat("vf\_latency\_stitch\_run")

val visibilityDeciderGates = VisibilityDeciderGates(decider)

val verdictLogger =

createVerdictLogger(

visibilityDeciderGates.enableVerdictLoggerTCVL,

decider,

libraryStatsReceiver)

request: TimelineConversationsVisibilityRequest =>

val elapsed = Stopwatch.start()

var runStitchStartMs = 0L

val future = request.prefetchedSafetyLabels match {

case Some(labels) => Future.value(labels)

case \_ =>

batchSafetyLabelRepository((request.conversationId, request.tweetIds))

}

val fosnrPefetchedLabelsRelationshipFeatures =

new FosnrPefetchedLabelsRelationshipFeatures(

userRelationshipSource = userRelationshipSource,

statsReceiver = libraryStatsReceiver)

val authorFeatures = new AuthorFeatures(userSource, libraryStatsReceiver)

Stitch.callFuture(future).flatMap {

kvr: KeyValueResult[Long, scala.collection.Map[SafetyLabelType, SafetyLabel]] =>

val featureMapProvider: (ContentId, SafetyLevel) => FeatureMap = {

case (TweetId(tweetId), safetyLevel) =>

val constantTweetSafetyLabels: Seq[TweetSafetyLabel] =

kvr.found.getOrElse(tweetId, Map.empty).toSeq.map {

case (safetyLabelType, safetyLabel) =>

TweetSafetyLabel.fromThrift(SafetyLabelValue(safetyLabelType, safetyLabel))

}

val replyAuthor = request.tweetAuthors.flatMap {

\_(tweetId) match {

case Return(Some(userId)) => Some(userId)

case \_ => None

}

}

val fosnrPefetchedLabelsRelationshipFeatureConf = replyAuthor match {

case Some(authorId) if visibilityLibrary.isReleaseCandidateEnabled =>

fosnrPefetchedLabelsRelationshipFeatures

.forTweetWithSafetyLabelsAndAuthorId(

safetyLabels = constantTweetSafetyLabels,

authorId = authorId,

viewerId = request.viewerContext.userId)

case \_ => fosnrPefetchedLabelsRelationshipFeatures.forNonFosnr()

}

val authorFeatureConf = replyAuthor match {

case Some(authorId) if visibilityLibrary.isReleaseCandidateEnabled =>

authorFeatures.forAuthorId(authorId)

case \_ => authorFeatures.forNoAuthor()

}

val baseBuilderArguments = (safetyLevel match {

case TimelineConversationsDownranking =>

Seq(tweetIdFeatures.forTweetId(tweetId, constantTweetSafetyLabels))

case TimelineConversationsDownrankingMinimal =>

Seq(tweetIdFeaturesMinimal.forTweetId(tweetId, constantTweetSafetyLabels))

case \_ => Nil

}) :+ fosnrPefetchedLabelsRelationshipFeatureConf :+ authorFeatureConf

val tweetAuthorUserLabels: Option[Seq[Label]] =

request.prefetchedTweetAuthorUserLabels.flatMap {

\_.apply(tweetId) match {

case Return(Some(labelMap)) =>

Some(labelMap.values.toSeq)

case \_ =>

None

}

}

val hasInnerCircleOfFriendsRelationship: Boolean =

request.innerCircleOfFriendsRelationships match {

case Some(keyValueResult) =>

keyValueResult(tweetId) match {

case Return(Some(true)) => true

case \_ => false

}

case None => false

}

val builderArguments: Seq[FeatureMapBuilder => FeatureMapBuilder] =

tweetAuthorUserLabels match {

case Some(labels) =>

baseBuilderArguments :+ { (fmb: FeatureMapBuilder) =>

fmb.withConstantFeature(AuthorUserLabels, labels)

}

case None =>

baseBuilderArguments :+ { (fmb: FeatureMapBuilder) =>

fmb.withConstantFeature(AuthorUserLabels, Seq.empty)

}

case \_ =>

baseBuilderArguments

}

val tweetParentIdOpt: Option[Long] =

request.tweetParentIdMap.flatMap(tweetParentIdMap => tweetParentIdMap(tweetId))

visibilityLibrary.featureMapBuilder(builderArguments :+ { (fmb: FeatureMapBuilder) =>

fmb.withConstantFeature(

HasInnerCircleOfFriendsRelationship,

hasInnerCircleOfFriendsRelationship)

fmb.withConstantFeature(TweetConversationId, request.conversationId)

fmb.withConstantFeature(TweetParentId, tweetParentIdOpt)

fmb.withConstantFeature(

ConversationRootAuthorIsVerified,

request.rootAuthorIsVerified)

})

case \_ =>

visibilityLibrary.featureMapBuilder(Nil)

}

val safetyLevel =

if (request.minimalSectioningOnly) TimelineConversationsDownrankingMinimal

else TimelineConversationsDownranking

val evaluationContextBuilder = visibilityLibrary

.evaluationContextBuilder(request.viewerContext)

.withUnitOfDiversion(UnitOfDiversion.ConversationId(request.conversationId))

visibilityLibrary

.runRuleEngineBatch(

request.tweetIds.map(TweetId),

featureMapProvider,

evaluationContextBuilder,

safetyLevel

)

.map { results: Seq[Try[VisibilityResult]] =>

val (succeededRequests, \_) = results.partition(\_.exists(\_.finished))

val visibilityResultMap = succeededRequests.flatMap {

case Return(result) =>

scribeVisibilityVerdict(

result,

visibilityDeciderGates.enableVerdictScribingTCVL,

verdictLogger,

request.viewerContext.userId,

safetyLevel)

result.contentId match {

case TweetId(id) => Some((id, result))

case \_ => None

}

case \_ => None

}.toMap

val failedTweetIds = request.tweetIds diff visibilityResultMap.keys.toSeq

val response = TimelineConversationsVisibilityResponse(

visibilityResults = visibilityResultMap,

failedTweetIds = failedTweetIds

)

runStitchStartMs = elapsed().inMilliseconds

val buildStitchStatMs = elapsed().inMilliseconds

vfLatencyStitchBuildStat.add(buildStitchStatMs)

response

}

.onSuccess(\_ => {

val overallStatMs = elapsed().inMilliseconds

vfLatencyOverallStat.add(overallStatMs)

val runStitchEndMs = elapsed().inMilliseconds

vfLatencyStitchRunStat.add(runStitchEndMs - runStitchStartMs)

})

}

}

def scribeVisibilityVerdict(

visibilityResult: VisibilityResult,

enableVerdictScribing: Gate[Unit],

verdictLogger: VerdictLogger,

viewerId: Option[Long],

safetyLevel: SafetyLevel

): Unit = if (enableVerdictScribing()) {

verdictLogger.scribeVerdict(

visibilityResult = visibilityResult,

viewerId = viewerId,

safetyLevel = toThrift(safetyLevel),

vfLibType = VFLibType.TimelineConversationsVisibilityLibrary)

}

def createVerdictLogger(

enableVerdictLogger: Gate[Unit],

decider: Decider,

statsReceiver: StatsReceiver

): VerdictLogger = {

if (enableVerdictLogger()) {

VerdictLogger(statsReceiver, decider)

} else {

VerdictLogger.Empty

}

}

}