package com.twitter.search.common.relevance;

import java.util.List;

import java.util.Map;

import java.util.Set;

import java.util.concurrent.Executors;

import java.util.concurrent.ScheduledExecutorService;

import java.util.concurrent.TimeUnit;

import java.util.concurrent.atomic.AtomicLong;

import java.util.stream.Collectors;

import scala.runtime.BoxedUnit;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

import com.google.common.collect.ImmutableMap;

import com.google.common.collect.Sets;

import com.google.common.util.concurrent.ThreadFactoryBuilder;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.finagle.Service;

import com.twitter.finagle.ThriftMux;

import com.twitter.finagle.builder.ClientBuilder;

import com.twitter.finagle.builder.ClientConfig;

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

import com.twitter.finagle.mtls.client.MtlsClientBuilder;

import com.twitter.finagle.stats.DefaultStatsReceiver;

import com.twitter.finagle.thrift.ThriftClientRequest;

import com.twitter.search.common.metrics.RelevanceStats;

import com.twitter.search.common.metrics.SearchCounter;

import com.twitter.trends.plus.Module;

import com.twitter.trends.plus.TrendsPlusRequest;

import com.twitter.trends.plus.TrendsPlusResponse;

import com.twitter.trends.service.gen.Location;

import com.twitter.trends.trending\_content.thriftjava.TrendingContentService;

import com.twitter.trends.trends\_metadata.thriftjava.TrendsMetadataService;

import com.twitter.util.Duration;

import com.twitter.util.Future;

import com.twitter.util.Try;

/\*\*

\* Manages trends data retrieved from trends thrift API and perform automatic refresh.

\*/

public final class TrendsThriftDataServiceManager {

private static final Logger LOG =

LoggerFactory.getLogger(TrendsThriftDataServiceManager.class.getName());

private static final int DEFAULT\_TIME\_TO\_KILL\_SEC = 60;

@VisibleForTesting

protected static final Map<String, String> DEFAULT\_TRENDS\_PARAMS\_MAP = ImmutableMap.of(

"MAX\_ITEMS\_TO\_RETURN", "10"); // we only take top 10 for each woeid.

@VisibleForTesting

protected static final int MAX\_TRENDS\_PER\_WOEID = 10;

private final Duration requestTimeout;

private final Duration refreshDelayDuration;

private final Duration reloadIntervalDuration;

private final int numRetries;

// a list of trends cache we want to update

private final List<NGramCache> trendsCacheList;

private final SearchCounter getAvailableSuccessCounter =

RelevanceStats.exportLong("trends\_extractor\_get\_available\_success");

private final SearchCounter getAvailableFailureCounter =

RelevanceStats.exportLong("trends\_extractor\_get\_available\_failure");

private final SearchCounter getTrendsSuccessCounter =

RelevanceStats.exportLong("trends\_extractor\_success\_fetch");

private final SearchCounter getTrendsFailureCounter =

RelevanceStats.exportLong("trends\_extractor\_failed\_fetch");

private final SearchCounter updateFailureCounter =

RelevanceStats.exportLong("trends\_extractor\_failed\_update");

private final ServiceIdentifier serviceIdentifier;

private ScheduledExecutorService scheduler;

@VisibleForTesting

protected Service<ThriftClientRequest, byte[]> contentService;

protected TrendingContentService.ServiceToClient contentClient;

protected Service<ThriftClientRequest, byte[]> metadataService;

protected TrendsMetadataService.ServiceToClient metadataClient;

@VisibleForTesting

protected TrendsUpdater trendsUpdater;

/\*\*

\* Returns an instance of TrendsThriftDataServiceManager.

\* @param serviceIdentifier The service that wants to call

\* into Trend's services.

\* @param numRetries The number of retries in the event of

\* request failures.

\* @param requestTimeout The amount of time we wait before we consider a

\* a request as failed.

\* @param initTrendsCacheDelay How long to wait before the initial

\* filling of the Trends cache in milliseconds.

\* @param reloadInterval How often to refresh the cache with updated trends.

\* @param trendsCacheList The cache of trends.

\* @return An instance of TrendsThriftDataServiceManager configured

\* with respect to the params provided.

\*/

public static TrendsThriftDataServiceManager newInstance(

ServiceIdentifier serviceIdentifier,

int numRetries,

Duration requestTimeout,

Duration initTrendsCacheDelay,

Duration reloadInterval,

List<NGramCache> trendsCacheList) {

return new TrendsThriftDataServiceManager(

serviceIdentifier,

numRetries,

requestTimeout,

initTrendsCacheDelay,

reloadInterval,

trendsCacheList);

}

/\*\*

\* Resume auto refresh. Always called in constructor. Can be invoked after a

\* stopAuthRefresh call to resume auto refreshing. Invoking it after shutDown is undefined.

\*/

public synchronized void startAutoRefresh() {

if (scheduler == null) {

scheduler = Executors.newSingleThreadScheduledExecutor(

new ThreadFactoryBuilder().setDaemon(true).setNameFormat(

"trends-data-refresher[%d]").build());

scheduler.scheduleAtFixedRate(

trendsUpdater,

refreshDelayDuration.inSeconds(),

reloadIntervalDuration.inSeconds(),

TimeUnit.SECONDS);

}

}

/\*\*

\* Stop auto refresh. Wait for the current execution thread to finish.

\* This is a blocking call.

\*/

public synchronized void stopAutoRefresh() {

if (scheduler != null) {

scheduler.shutdown(); // Disable new tasks from being submitted

try {

// Wait a while for existing tasks to terminate

if (!scheduler.awaitTermination(DEFAULT\_TIME\_TO\_KILL\_SEC, TimeUnit.SECONDS)) {

scheduler.shutdownNow(); // Cancel currently executing tasks

// Wait a while for tasks to respond to being cancelled

if (!scheduler.awaitTermination(DEFAULT\_TIME\_TO\_KILL\_SEC, TimeUnit.SECONDS)) {

LOG.info("Executor thread pool did not terminate.");

}

}

} catch (InterruptedException ie) {

// (Re-)Cancel if current thread also interrupted

scheduler.shutdownNow();

// Preserve interrupt status

Thread.currentThread().interrupt();

}

scheduler = null;

}

}

/\*\* Shuts down the manager. \*/

public void shutDown() {

stopAutoRefresh();

// clear the cache

for (NGramCache cache : trendsCacheList) {

cache.clear();

}

if (contentService != null) {

contentService.close();

}

if (metadataService != null) {

metadataService.close();

}

}

private TrendsThriftDataServiceManager(

ServiceIdentifier serviceIdentifier,

int numRetries,

Duration requestTimeoutMS,

Duration refreshDelayDuration,

Duration reloadIntervalDuration,

List<NGramCache> trendsCacheList) {

this.numRetries = numRetries;

this.requestTimeout = requestTimeoutMS;

this.refreshDelayDuration = refreshDelayDuration;

this.reloadIntervalDuration = reloadIntervalDuration;

this.serviceIdentifier = serviceIdentifier;

this.trendsCacheList = Preconditions.checkNotNull(trendsCacheList);

trendsUpdater = new TrendsUpdater();

metadataService = buildMetadataService();

metadataClient = buildMetadataClient(metadataService);

contentService = buildContentService();

contentClient = buildContentClient(contentService);

}

@VisibleForTesting

protected Service<ThriftClientRequest, byte[]> buildContentService() {

ClientBuilder<

ThriftClientRequest,

byte[], ClientConfig.Yes,

ClientConfig.Yes,

ClientConfig.Yes

>

builder = ClientBuilder.get()

.stack(ThriftMux.client())

.name("trends\_thrift\_data\_service\_manager\_content")

.dest("")

.retries(numRetries)

.reportTo(DefaultStatsReceiver.get())

.tcpConnectTimeout(requestTimeout)

.requestTimeout(requestTimeout);

ClientBuilder mtlsBuilder =

new MtlsClientBuilder.MtlsClientBuilderSyntax<>(builder).mutualTls(serviceIdentifier);

return ClientBuilder.safeBuild(mtlsBuilder);

}

@VisibleForTesting

protected TrendingContentService.ServiceToClient buildContentClient(

Service<ThriftClientRequest, byte[]> service) {

return new TrendingContentService.ServiceToClient(service);

}

@VisibleForTesting

protected Service<ThriftClientRequest, byte[]> buildMetadataService() {

ClientBuilder<

ThriftClientRequest,

byte[],

ClientConfig.Yes,

ClientConfig.Yes,

ClientConfig.Yes

>

builder = ClientBuilder.get()

.stack(ThriftMux.client())

.name("trends\_thrift\_data\_service\_manager\_metadata")

.dest("")

.retries(numRetries)

.reportTo(DefaultStatsReceiver.get())

.tcpConnectTimeout(requestTimeout)

.requestTimeout(requestTimeout);

ClientBuilder mtlsBuilder =

new MtlsClientBuilder.MtlsClientBuilderSyntax<>(builder).mutualTls(serviceIdentifier);

return ClientBuilder.safeBuild(mtlsBuilder);

}

@VisibleForTesting

protected TrendsMetadataService.ServiceToClient buildMetadataClient(

Service<ThriftClientRequest, byte[]> service) {

return new TrendsMetadataService.ServiceToClient(service);

}

/\*\*

\* Updater that fetches available woeids and corresponding trending terms.

\*/

@VisibleForTesting

protected class TrendsUpdater implements Runnable {

@Override

public void run() {

populateCacheFromTrendsService();

}

private Future<BoxedUnit> populateCacheFromTrendsService() {

long startTime = System.currentTimeMillis();

AtomicLong numTrendsReceived = new AtomicLong(0);

return metadataClient.getAvailable().flatMap(locations -> {

if (locations == null) {

getAvailableFailureCounter.increment();

LOG.warn("Failed to get woeids from trends.");

return Future.value(BoxedUnit.UNIT);

}

getAvailableSuccessCounter.increment();

return populateCacheFromTrendLocations(locations, numTrendsReceived);

}).onFailure(throwable -> {

LOG.info("Update failed", throwable);

updateFailureCounter.increment();

return BoxedUnit.UNIT;

}).ensure(() -> {

logRefreshStatus(startTime, numTrendsReceived);

return BoxedUnit.UNIT;

});

}

private Future<BoxedUnit> populateCacheFromTrendLocations(

List<Location> locations,

AtomicLong numTrendsReceived) {

List<Future<TrendsPlusResponse>> trendsPlusFutures = locations.stream()

.map(location -> makeTrendsPlusRequest(location))

.collect(Collectors.toList());

Future<List<Try<TrendsPlusResponse>>> trendsPlusFuture =

Future.collectToTry(trendsPlusFutures);

return trendsPlusFuture.map(tryResponses -> {

populateCacheFromResponses(tryResponses, numTrendsReceived);

return BoxedUnit.UNIT;

});

}

private Future<TrendsPlusResponse> makeTrendsPlusRequest(Location location) {

TrendsPlusRequest request = new TrendsPlusRequest()

.setWoeid(location.getWoeid())

.setMaxTrends(MAX\_TRENDS\_PER\_WOEID);

long startTime = System.currentTimeMillis();

return contentClient.getTrendsPlus(request)

.onSuccess(response -> {

getTrendsSuccessCounter.increment();

return BoxedUnit.UNIT;

}).onFailure(throwable -> {

getTrendsFailureCounter.increment();

return BoxedUnit.UNIT;

});

}

private void populateCacheFromResponses(

List<Try<TrendsPlusResponse>> tryResponses,

AtomicLong numTrendsReceived) {

Set<String> trendStrings = Sets.newHashSet();

for (Try<TrendsPlusResponse> tryResponse : tryResponses) {

if (tryResponse.isThrow()) {

LOG.warn("Failed to fetch trends:" + tryResponse.toString());

continue;

}

TrendsPlusResponse trendsPlusResponse = tryResponse.get();

numTrendsReceived.addAndGet(trendsPlusResponse.modules.size());

for (Module module : trendsPlusResponse.modules) {

trendStrings.add(module.getTrend().name);

}

}

for (NGramCache cache : trendsCacheList) {

cache.addAll(trendStrings);

}

}

}

private void logRefreshStatus(long startTime, AtomicLong numTrendsReceived) {

LOG.info(String.format("Refresh done in [%dms] :\nfetchSuccess[%d] fetchFailure[%d] "

+ "updateFailure[%d] num trends received [%d]",

System.currentTimeMillis() - startTime,

getTrendsSuccessCounter.get(),

getTrendsFailureCounter.get(),

updateFailureCounter.get(),

numTrendsReceived.get()));

}

}