package com.twitter.search.earlybird\_root;

import java.util.Collections;

import java.util.List;

import java.util.Map;

import java.util.SortedSet;

import java.util.TreeSet;

import javax.inject.Inject;

import javax.inject.Named;

import javax.inject.Singleton;

import com.google.common.base.Preconditions;

import com.google.common.collect.Lists;

import com.google.common.collect.Maps;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import com.twitter.finagle.Service;

import com.twitter.finagle.SimpleFilter;

import com.twitter.finagle.stats.StatsReceiver;

import com.twitter.search.common.decider.SearchDecider;

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

import com.twitter.search.common.root.PartitionConfig;

import com.twitter.search.common.root.PartitionLoggingSupport;

import com.twitter.search.common.root.RequestSuccessStats;

import com.twitter.search.common.root.RootClientServiceBuilder;

import com.twitter.search.common.root.ScatterGatherService;

import com.twitter.search.common.root.ScatterGatherSupport;

import com.twitter.search.common.root.SearchRootModule;

import com.twitter.search.common.schema.earlybird.EarlybirdCluster;

import com.twitter.search.earlybird.config.TierConfig;

import com.twitter.search.earlybird.config.TierInfo;

import com.twitter.search.earlybird.config.TierInfoSource;

import com.twitter.search.earlybird.config.TierInfoUtil;

import com.twitter.search.earlybird.config.TierInfoWrapper;

import com.twitter.search.earlybird.thrift.EarlybirdRequest;

import com.twitter.search.earlybird.thrift.EarlybirdResponse;

import com.twitter.search.earlybird.thrift.EarlybirdResponseCode;

import com.twitter.search.earlybird.thrift.EarlybirdService.ServiceIface;

import com.twitter.search.earlybird.thrift.ThriftSearchResults;

import com.twitter.search.earlybird\_root.common.EarlybirdRequestContext;

import com.twitter.search.earlybird\_root.filters.EarlybirdTimeRangeFilter;

import com.twitter.search.earlybird\_root.filters.RequestContextToEarlybirdRequestFilter;

import com.twitter.util.Function;

import com.twitter.util.Future;

@Singleton

public class EarlybirdServiceChainBuilder {

private static final Logger LOG = LoggerFactory.getLogger(EarlybirdServiceChainBuilder.class);

private static final String SEARCH\_METHOD\_NAME = "search";

private static final EarlybirdResponse TIER\_SKIPPED\_RESPONSE =

new EarlybirdResponse(EarlybirdResponseCode.TIER\_SKIPPED, 0)

.setSearchResults(new ThriftSearchResults())

.setDebugString("Request to cluster dropped by decider, or sent as dark read.");

private final EarlybirdTierThrottleDeciders tierThrottleDeciders;

private final RequestContextToEarlybirdRequestFilter requestContextToEarlybirdRequestFilter;

private final SearchDecider decider;

private final String normalizedSearchRootName;

private final RootClientServiceBuilder<ServiceIface> clientServiceBuilder;

private final String partitionPath;

private final int numPartitions;

private final SortedSet<TierInfo> tierInfos;

private final PartitionAccessController partitionAccessController;

private final StatsReceiver statsReceiver;

/\*\*

\* Construct a ScatterGatherServiceChain, by loading configurations from earlybird-tiers.yml.

\*/

@Inject

public EarlybirdServiceChainBuilder(

PartitionConfig partitionConfig,

RequestContextToEarlybirdRequestFilter requestContextToEarlybirdRequestFilter,

EarlybirdTierThrottleDeciders tierThrottleDeciders,

@Named(SearchRootModule.NAMED\_NORMALIZED\_SEARCH\_ROOT\_NAME) String normalizedSearchRootName,

SearchDecider decider,

TierInfoSource tierConfig,

RootClientServiceBuilder<ServiceIface> clientServiceBuilder,

PartitionAccessController partitionAccessController,

StatsReceiver statsReceiver) {

this.partitionAccessController = partitionAccessController;

this.tierThrottleDeciders = Preconditions.checkNotNull(tierThrottleDeciders);

this.requestContextToEarlybirdRequestFilter = requestContextToEarlybirdRequestFilter;

this.normalizedSearchRootName = normalizedSearchRootName;

this.decider = decider;

this.statsReceiver = statsReceiver;

List<TierInfo> tierInformation = tierConfig.getTierInformation();

if (tierInformation == null || tierInformation.isEmpty()) {

LOG.error(

"No tier found in config file {} Did you set SEARCH\_ENV correctly?",

tierConfig.getConfigFileType());

throw new RuntimeException("No tier found in tier config file.");

}

// Get the tier info from the tier config yml file

TreeSet<TierInfo> infos = new TreeSet<>(TierInfoUtil.TIER\_COMPARATOR);

infos.addAll(tierInformation);

this.tierInfos = Collections.unmodifiableSortedSet(infos);

this.clientServiceBuilder = clientServiceBuilder;

this.partitionPath = partitionConfig.getPartitionPath();

this.numPartitions = partitionConfig.getNumPartitions();

LOG.info("Found the following tiers from config: {}", tierInfos);

}

/\*\* Builds the chain of services that should be queried on each request. \*/

public List<Service<EarlybirdRequestContext, EarlybirdResponse>> buildServiceChain(

ScatterGatherSupport<EarlybirdRequestContext, EarlybirdResponse> support,

PartitionLoggingSupport<EarlybirdRequestContext> partitionLoggingSupport) {

// Make sure the tier serving ranges do not overlap and do not have gaps.

TierInfoUtil.checkTierServingRanges(tierInfos);

List<Service<EarlybirdRequestContext, EarlybirdResponse>> chain = Lists.newArrayList();

for (TierInfo tierInfo : tierInfos) {

String tierName = tierInfo.getTierName();

if (tierInfo.isEnabled()) {

String rewrittenPartitionPath = partitionPath;

// This rewriting rule must match the rewriting rule inside

// EarlybirdServer#joinServerSet().

if (!TierConfig.DEFAULT\_TIER\_NAME.equals(tierName)) {

rewrittenPartitionPath = partitionPath + "/" + tierName;

}

clientServiceBuilder.initializeWithPathSuffix(

tierInfo.getTierName(),

numPartitions,

rewrittenPartitionPath);

try {

chain.add(createTierService(

support, tierInfo, clientServiceBuilder, partitionLoggingSupport));

} catch (Exception e) {

LOG.error("Failed to build clients for tier: {}", tierInfo.getTierName());

throw new RuntimeException(e);

}

} else {

LOG.info("Skipped disabled tier: {}", tierName);

}

}

return chain;

}

private Service<EarlybirdRequestContext, EarlybirdResponse> createTierService(

ScatterGatherSupport<EarlybirdRequestContext, EarlybirdResponse> support,

final TierInfo tierInfo,

RootClientServiceBuilder<ServiceIface> builder,

PartitionLoggingSupport<EarlybirdRequestContext> partitionLoggingSupport) {

final String tierName = tierInfo.getTierName();

RequestSuccessStats stats = new RequestSuccessStats(tierName);

List<Service<EarlybirdRequest, EarlybirdResponse>> services =

builder.safeBuildServiceList(SEARCH\_METHOD\_NAME);

// Get the client list for this tier, and apply the degradationTrackerFilter to each response.

//

// We currently do this only for the EarlybirdSearchMultiTierAdaptor (the full archive cluster).

// If we want to do this for all clusters (or if we want to apply any other filter to all

// earlybird responses, for other clusters), we should change ScatterGatherService's constructor

// to take in a filter, and apply it there.

ClientBackupFilter backupFilter = new ClientBackupFilter(

"root\_" + EarlybirdCluster.FULL\_ARCHIVE.getNameForStats(),

tierName,

statsReceiver,

decider);

List<Service<EarlybirdRequestContext, EarlybirdResponse>> clients = Lists.newArrayList();

ClientLatencyFilter latencyFilter = new ClientLatencyFilter(tierName);

for (Service<EarlybirdRequest, EarlybirdResponse> client : services) {

clients.add(requestContextToEarlybirdRequestFilter

.andThen(backupFilter)

.andThen(latencyFilter)

.andThen(client));

}

clients = SkipPartitionFilter.wrapServices(tierName, clients, partitionAccessController);

// Build the scatter gather service for this tier.

// Each tier has their own stats.

ScatterGatherService<EarlybirdRequestContext, EarlybirdResponse> scatterGatherService =

new ScatterGatherService<>(

support, clients, stats, partitionLoggingSupport);

SimpleFilter<EarlybirdRequestContext, EarlybirdResponse> tierThrottleFilter =

getTierThrottleFilter(tierInfo, tierName);

EarlybirdTimeRangeFilter timeRangeFilter =

EarlybirdTimeRangeFilter.newTimeRangeFilterWithQueryRewriter(

(requestContext, userOverride) -> new TierInfoWrapper(tierInfo, userOverride),

decider);

return tierThrottleFilter

.andThen(timeRangeFilter)

.andThen(scatterGatherService);

}

private SimpleFilter<EarlybirdRequestContext, EarlybirdResponse> getTierThrottleFilter(

final TierInfo tierInfo,

final String tierName) {

// A filter that throttles request rate.

final String tierThrottleDeciderKey = tierThrottleDeciders.getTierThrottleDeciderKey(

normalizedSearchRootName, tierName);

SimpleFilter<EarlybirdRequestContext, EarlybirdResponse> tierThrottleFilter =

new SimpleFilter<EarlybirdRequestContext, EarlybirdResponse>() {

private final Map<TierInfo.RequestReadType, SearchCounter> readCounts =

getReadCountsMap();

private Map<TierInfo.RequestReadType, SearchCounter> getReadCountsMap() {

Map<TierInfo.RequestReadType, SearchCounter> readCountsMap =

Maps.newEnumMap(TierInfo.RequestReadType.class);

for (TierInfo.RequestReadType readType : TierInfo.RequestReadType.values()) {

readCountsMap.put(readType,

SearchCounter.export("earlybird\_tier\_" + tierName + "\_"

+ readType.name().toLowerCase() + "\_read\_count"));

}

return Collections.unmodifiableMap(readCountsMap);

}

private final SearchCounter tierRequestDroppedByDeciderCount =

SearchCounter.export("earlybird\_tier\_" + tierName

+ "\_request\_dropped\_by\_decider\_count");

@Override

public Future<EarlybirdResponse> apply(

EarlybirdRequestContext requestContext,

Service<EarlybirdRequestContext, EarlybirdResponse> service) {

// a blank response is returned when a request is dropped by decider, or

// a request is sent as a dark read.

final Future<EarlybirdResponse> blankTierResponse = Future.value(TIER\_SKIPPED\_RESPONSE);

if (tierThrottleDeciders.shouldSendRequestToTier(tierThrottleDeciderKey)) {

TierInfoWrapper tierInfoWrapper =

new TierInfoWrapper(tierInfo, requestContext.useOverrideTierConfig());

TierInfo.RequestReadType readType = tierInfoWrapper.getReadType();

readCounts.get(readType).increment();

switch (readType) {

case DARK:

// dark read: call backend but do not wait for results

service.apply(requestContext);

return blankTierResponse;

case GREY:

// grey read: call backend, wait for results, but discard results.

return service.apply(requestContext).flatMap(

new Function<EarlybirdResponse, Future<EarlybirdResponse>>() {

@Override

public Future<EarlybirdResponse> apply(EarlybirdResponse v1) {

// No matter what's returned, always return blankTierResponse.

return blankTierResponse;

}

});

case LIGHT:

// light read: return the future from the backend service.

return service.apply(requestContext);

default:

throw new RuntimeException("Unknown read type: " + readType);

}

} else {

// Request is dropped by throttle decider

tierRequestDroppedByDeciderCount.increment();

return blankTierResponse;

}

}

};

return tierThrottleFilter;

}

}