package com.twitter.search.earlybird;

import java.util.concurrent.ArrayBlockingQueue;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.RejectedExecutionException;

import java.util.concurrent.ThreadFactory;

import java.util.concurrent.ThreadPoolExecutor;

import java.util.concurrent.TimeUnit;

import scala.Function0;

import com.google.common.annotations.VisibleForTesting;

import com.google.common.base.Preconditions;

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

import com.twitter.search.common.concurrent.ThreadPoolExecutorStats;

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

import com.twitter.search.earlybird.common.config.EarlybirdProperty;

import com.twitter.util.ExecutorServiceFuturePool;

import com.twitter.util.Future;

import com.twitter.util.FuturePool;

/\*\*

\* A future pool that delegates all calls to an underlying futurePool, which can be recreated.

\*/

public class EarlybirdFuturePoolManager implements FuturePool {

private volatile ExecutorServiceFuturePool pool = null;

private final String threadName;

private final ThreadPoolExecutorStats threadPoolExecutorStats;

public EarlybirdFuturePoolManager(String threadName) {

this.threadName = threadName;

this.threadPoolExecutorStats = new ThreadPoolExecutorStats(threadName);

}

final synchronized void createUnderlyingFuturePool(int threadCount) {

Preconditions.checkState(pool == null, "Cannot create a new pool before stopping the old one");

ExecutorService executorService =

createExecutorService(threadCount, getMaxQueueSize());

if (executorService instanceof ThreadPoolExecutor) {

threadPoolExecutorStats.setUnderlyingExecutorForStats((ThreadPoolExecutor) executorService);

}

pool = new ExecutorServiceFuturePool(executorService);

}

final synchronized void stopUnderlyingFuturePool(long timeout, TimeUnit timeunit)

throws InterruptedException {

Preconditions.checkNotNull(pool);

pool.executor().shutdown();

pool.executor().awaitTermination(timeout, timeunit);

pool = null;

}

boolean isPoolReady() {

return pool != null;

}

@Override

public final <T> Future<T> apply(Function0<T> f) {

return Preconditions.checkNotNull(pool).apply(f);

}

@VisibleForTesting

protected ExecutorService createExecutorService(int threadCount, int maxQueueSize) {

if (maxQueueSize <= 0) {

return Executors.newFixedThreadPool(threadCount, createThreadFactory(threadName));

}

SearchRateCounter rejectedTaskCounter =

SearchRateCounter.export(threadName + "\_rejected\_task\_count");

return new ThreadPoolExecutor(

threadCount, threadCount, 0, TimeUnit.MILLISECONDS,

new ArrayBlockingQueue<>(maxQueueSize),

createThreadFactory(threadName),

(runnable, executor) -> {

rejectedTaskCounter.increment();

throw new RejectedExecutionException(threadName + " queue is full");

});

}

@VisibleForTesting

protected int getMaxQueueSize() {

return EarlybirdProperty.MAX\_QUEUE\_SIZE.get(0);

}

@VisibleForTesting

static ThreadFactory createThreadFactory(String threadName) {

return new ThreadFactoryBuilder()

.setNameFormat(threadName + "-%d")

.setDaemon(true)

.build();

}

@Override

public int poolSize() {

return Preconditions.checkNotNull(pool).poolSize();

}

@Override

public int numActiveTasks() {

return Preconditions.checkNotNull(pool).numActiveTasks();

}

@Override

public long numCompletedTasks() {

return Preconditions.checkNotNull(pool).numCompletedTasks();

}

}