package com.twitter.follow\_recommendations.common.base

import com.twitter.finagle.stats.NullStatsReceiver

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.follow\_recommendations.common.models.FilterReason

import com.twitter.stitch.Arrow

import com.twitter.stitch.Stitch

trait Predicate[-Q] {

def apply(item: Q): Stitch[PredicateResult]

def arrow: Arrow[Q, PredicateResult] = Arrow.apply(apply)

def map[K](mapper: K => Q): Predicate[K] = Predicate(arrow.contramap(mapper))

/\*\*

\* check the predicate results for a batch of items for convenience.

\*

\* mark it as final to avoid potential abuse usage

\*/

final def batch(items: Seq[Q]): Stitch[Seq[PredicateResult]] = {

this.arrow.traverse(items)

}

/\*\*

\* Syntax sugar for functions which take in 2 inputs as a tuple.

\*/

def apply[Q1, Q2](item1: Q1, item2: Q2)(implicit ev: ((Q1, Q2)) => Q): Stitch[PredicateResult] = {

apply((item1, item2))

}

/\*\*

\* Runs the predicates in sequence. The returned predicate will return true iff both the predicates return true.

\* ie. it is an AND operation

\*

\* We short-circuit the evaluation, ie we don't evaluate the 2nd predicate if the 1st is false

\*

\* @param p predicate to run in sequence

\*

\* @return a new predicate object that represents the logical AND of both predicates

\*/

def andThen[Q1 <: Q](p: Predicate[Q1]): Predicate[Q1] = {

Predicate({ query: Q1 =>

apply(query).flatMap {

case PredicateResult.Valid => p(query)

case PredicateResult.Invalid(reasons) => Stitch.value(PredicateResult.Invalid(reasons))

}

})

}

/\*\*

\* Creates a predicate which runs the current & given predicate in sequence.

\* The returned predicate will return true if either current or given predicate returns true.

\* That is, given predicate will be only run if current predicate returns false.

\*

\* @param p predicate to run in sequence

\*

\* @return new predicate object that represents the logical OR of both predicates.

\* if both are invalid, the reason would be the set of all invalid reasons.

\*/

def or[Q1 <: Q](p: Predicate[Q1]): Predicate[Q1] = {

Predicate({ query: Q1 =>

apply(query).flatMap {

case PredicateResult.Valid => Stitch.value(PredicateResult.Valid)

case PredicateResult.Invalid(reasons) =>

p(query).flatMap {

case PredicateResult.Valid => Stitch.value(PredicateResult.Valid)

case PredicateResult.Invalid(newReasons) =>

Stitch.value(PredicateResult.Invalid(reasons ++ newReasons))

}

}

})

}

/\*

\* Runs the predicate only if the provided predicate is valid, otherwise returns valid.

\* \*/

def gate[Q1 <: Q](gatingPredicate: Predicate[Q1]): Predicate[Q1] = {

Predicate { query: Q1 =>

gatingPredicate(query).flatMap { result =>

if (result == PredicateResult.Valid) {

apply(query)

} else {

Stitch.value(PredicateResult.Valid)

}

}

}

}

def observe(statsReceiver: StatsReceiver): Predicate[Q] = Predicate(

StatsUtil.profilePredicateResult(this.arrow, statsReceiver))

def convertToFailOpenWithResultType(resultType: PredicateResult): Predicate[Q] = {

Predicate { query: Q =>

apply(query).handle {

case \_: Exception =>

resultType

}

}

}

}

class TruePredicate[Q] extends Predicate[Q] {

override def apply(item: Q): Stitch[PredicateResult] = Predicate.AlwaysTrueStitch

}

class FalsePredicate[Q](reason: FilterReason) extends Predicate[Q] {

val InvalidResult = Stitch.value(PredicateResult.Invalid(Set(reason)))

override def apply(item: Q): Stitch[PredicateResult] = InvalidResult

}

object Predicate {

val AlwaysTrueStitch = Stitch.value(PredicateResult.Valid)

val NumBatchesStat = "num\_batches\_stats"

val NumBatchesCount = "num\_batches"

def apply[Q](func: Q => Stitch[PredicateResult]): Predicate[Q] = new Predicate[Q] {

override def apply(item: Q): Stitch[PredicateResult] = func(item)

override val arrow: Arrow[Q, PredicateResult] = Arrow(func)

}

def apply[Q](outerArrow: Arrow[Q, PredicateResult]): Predicate[Q] = new Predicate[Q] {

override def apply(item: Q): Stitch[PredicateResult] = arrow(item)

override val arrow: Arrow[Q, PredicateResult] = outerArrow

}

/\*\*

\* Given some items, this function

\* 1. chunks them up in groups

\* 2. lazily applies a predicate on each group

\* 3. filters based on the predicate

\* 4. takes first numToTake items.

\*

\* If numToTake is satisfied, then any later predicates are not called.

\*

\* @param items items of type Q

\* @param predicate predicate that determines whether an item is acceptable

\* @param batchSize batch size to call the predicate with

\* @param numToTake max number of items to return

\* @param stats stats receiver

\* @tparam Q type of item

\*

\* @return a future of K items

\*/

def batchFilterTake[Q](

items: Seq[Q],

predicate: Predicate[Q],

batchSize: Int,

numToTake: Int,

stats: StatsReceiver

): Stitch[Seq[Q]] = {

def take(

input: Iterator[Stitch[Seq[Q]]],

prev: Seq[Q],

takeSize: Int,

numOfBatch: Int

): Stitch[(Seq[Q], Int)] = {

if (input.hasNext) {

val currFut = input.next()

currFut.flatMap { curr =>

val taken = curr.take(takeSize)

val combined = prev ++ taken

if (taken.size < takeSize)

take(input, combined, takeSize - taken.size, numOfBatch + 1)

else Stitch.value((combined, numOfBatch + 1))

}

} else {

Stitch.value((prev, numOfBatch))

}

}

val batchedItems = items.view.grouped(batchSize)

val batchedFutures = batchedItems.map { batch =>

Stitch.traverse(batch)(predicate.apply).map { conds =>

(batch.zip(conds)).withFilter(\_.\_2.value).map(\_.\_1)

}

}

take(batchedFutures, Nil, numToTake, 0).map {

case (filtered: Seq[Q], numOfBatch: Int) =>

stats.stat(NumBatchesStat).add(numOfBatch)

stats.counter(NumBatchesCount).incr(numOfBatch)

filtered

}

}

/\*\*

\* filter a list of items based on the predicate

\*

\* @param items a list of items

\* @param predicate predicate of the item

\* @tparam Q item type

\* @return the list of items that satisfy the predicate

\*/

def filter[Q](items: Seq[Q], predicate: Predicate[Q]): Stitch[Seq[Q]] = {

predicate.batch(items).map { results =>

items.zip(results).collect {

case (item, PredicateResult.Valid) => item

}

}

}

/\*\*

\* filter a list of items based on the predicate given the target

\*

\* @param target target item

\* @param items a list of items

\* @param predicate predicate of the (target, item) pair

\* @tparam Q item type

\* @return the list of items that satisfy the predicate given the target

\*/

def filter[T, Q](target: T, items: Seq[Q], predicate: Predicate[(T, Q)]): Stitch[Seq[Q]] = {

predicate.batch(items.map(i => (target, i))).map { results =>

items.zip(results).collect {

case (item, PredicateResult.Valid) => item

}

}

}

/\*\*

\* Returns a predicate, where an element is true iff it that element is true for all input predicates.

\* ie. it is an AND operation

\*

\* This is done concurrently.

\*

\* @param predicates list of predicates

\* @tparam Q Type parameter

\*

\* @return new predicate object that is the logical "and" of the input predicates

\*/

def andConcurrently[Q](predicates: Seq[Predicate[Q]]): Predicate[Q] = {

Predicate { query: Q =>

Stitch.traverse(predicates)(p => p(query)).map { predicateResults =>

val allInvalid = predicateResults

.collect {

case PredicateResult.Invalid(reason) =>

reason

}

if (allInvalid.isEmpty) {

PredicateResult.Valid

} else {

val allInvalidReasons = allInvalid.reduce(\_ ++ \_)

PredicateResult.Invalid(allInvalidReasons)

}

}

}

}

}

/\*\*

\* applies the underlying predicate when the param is on.

\*/

abstract class GatedPredicateBase[Q](

underlyingPredicate: Predicate[Q],

stats: StatsReceiver = NullStatsReceiver)

extends Predicate[Q] {

def gate(item: Q): Boolean

val underlyingPredicateTotal = stats.counter("underlying\_total")

val underlyingPredicateValid = stats.counter("underlying\_valid")

val underlyingPredicateInvalid = stats.counter("underlying\_invalid")

val notGatedCounter = stats.counter("not\_gated")

val ValidStitch: Stitch[PredicateResult.Valid.type] = Stitch.value(PredicateResult.Valid)

override def apply(item: Q): Stitch[PredicateResult] = {

if (gate(item)) {

underlyingPredicateTotal.incr()

underlyingPredicate(item)

} else {

notGatedCounter.incr()

ValidStitch

}

}

}