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

import com.twitter.finagle.stats.StatsReceiver

import com.twitter.stitch.Stitch

import com.twitter.timelines.configapi.HasParams

import com.twitter.timelines.configapi.Param

/\*\*

\* transform a or a list of candidate for target T

\*

\* @tparam T target type

\* @tparam C candidate type

\*/

trait Transform[-T, C] {

// you need to implement at least one of the two methods here.

def transformItem(target: T, item: C): Stitch[C] = {

transform(target, Seq(item)).map(\_.head)

}

def transform(target: T, items: Seq[C]): Stitch[Seq[C]]

def mapTarget[T2](mapper: T2 => T): Transform[T2, C] = {

val original = this

new Transform[T2, C] {

override def transformItem(target: T2, item: C): Stitch[C] = {

original.transformItem(mapper(target), item)

}

override def transform(target: T2, items: Seq[C]): Stitch[Seq[C]] = {

original.transform(mapper(target), items)

}

}

}

/\*\*

\* sequential composition. we execute this' transform first, followed by the other's transform

\*/

def andThen[T1 <: T](other: Transform[T1, C]): Transform[T1, C] = {

val original = this

new Transform[T1, C] {

override def transformItem(target: T1, item: C): Stitch[C] =

original.transformItem(target, item).flatMap(other.transformItem(target, \_))

override def transform(target: T1, items: Seq[C]): Stitch[Seq[C]] =

original.transform(target, items).flatMap(other.transform(target, \_))

}

}

def observe(statsReceiver: StatsReceiver): Transform[T, C] = {

val originalTransform = this

new Transform[T, C] {

override def transform(target: T, items: Seq[C]): Stitch[Seq[C]] = {

statsReceiver.counter(Transform.InputCandidatesCount).incr(items.size)

statsReceiver.stat(Transform.InputCandidatesStat).add(items.size)

StatsUtil.profileStitchSeqResults(originalTransform.transform(target, items), statsReceiver)

}

override def transformItem(target: T, item: C): Stitch[C] = {

statsReceiver.counter(Transform.InputCandidatesCount).incr()

StatsUtil.profileStitch(originalTransform.transformItem(target, item), statsReceiver)

}

}

}

}

trait GatedTransform[T <: HasParams, C] extends Transform[T, C] {

def gated(param: Param[Boolean]): Transform[T, C] = {

val original = this

(target: T, items: Seq[C]) => {

if (target.params(param)) {

original.transform(target, items)

} else {

Stitch.value(items)

}

}

}

}

object Transform {

val InputCandidatesCount = "input\_candidates"

val InputCandidatesStat = "input\_candidates\_stat"

}

class IdentityTransform[T, C] extends Transform[T, C] {

override def transform(target: T, items: Seq[C]): Stitch[Seq[C]] = Stitch.value(items)

}