package com.twitter.tweetypie.caching

import scala.collection.mutable

import com.twitter.util.Future

import com.twitter.stitch.Stitch

import com.twitter.stitch.Runner

import com.twitter.stitch.FutureRunner

import com.twitter.stitch.Group

/\*\*

\* Workaround for a infelicity in the implementation of [[Stitch.async]].

\*

\* This has the same semantics to [[Stitch.async]], with the exception

\* that interrupts to the main computation will not interrupt the

\* async call.

\*

\* The problem that this implementation solves is that we do not want

\* async calls grouped together with synchronous calls. See the

\* mailing list thread [1] for discussion. This may eventually be

\* fixed in Stitch.

\*/

private[caching] object StitchAsync {

// Contains a deferred Stitch that we want to run asynchronously

private[this] class AsyncCall(deferred: => Stitch[\_]) {

def call(): Stitch[\_] = deferred

}

private object AsyncGroup extends Group[AsyncCall, Unit] {

override def runner(): Runner[AsyncCall, Unit] =

new FutureRunner[AsyncCall, Unit] {

// All of the deferred calls of any type. When they are

// executed in `run`, the normal Stitch batching and deduping

// will occur.

private[this] val calls = new mutable.ArrayBuffer[AsyncCall]

def add(call: AsyncCall): Stitch[Unit] = {

// Just remember the deferred call.

calls.append(call)

// Since we don't wait for the completion of the effect,

// just return a constant value.

Stitch.Unit

}

def run(): Future[\_] = {

// The future returned from this innter invocation of

// Stitch.run is not linked to the returned future, so these

// effects are not linked to the outer Run in which this

// method was invoked.

Stitch.run {

Stitch.traverse(calls) { asyncCall: AsyncCall =>

asyncCall

.call()

.liftToTry // So that an exception will not interrupt the other calls

}

}

Future.Unit

}

}

}

def apply(call: => Stitch[\_]): Stitch[Unit] =

// Group together all of the async calls

Stitch.call(new AsyncCall(call), AsyncGroup)

}