package com.twitter.servo.cache

import com.google.common.primitives.{Ints, Longs}

import com.twitter.finagle.thrift.Protocols

import com.twitter.io.Buf

import com.twitter.scrooge.{ThriftStruct, ThriftStructCodec, ThriftStructSerializer}

import com.twitter.servo.util.Transformer

import com.twitter.util.{Time => UtilTime, Try}

import java.io.{ByteArrayInputStream, ByteArrayOutputStream}

import java.nio.ByteBuffer

import org.apache.thrift.TBase

import org.apache.thrift.protocol.{TCompactProtocol, TProtocolFactory}

import org.apache.thrift.transport.TIOStreamTransport

object Serializers { self =>

val CompactProtocolFactory = new TCompactProtocol.Factory

val EmptyByteArray = Array.empty[Byte]

val Unit = Transformer[Unit, Array[Byte]](\_ => EmptyByteArray, \_ => ())

object Long {

val Simple = Transformer[Long, Array[Byte]](Longs.toByteArray, Longs.fromByteArray)

}

object CachedLong {

val Compact: Serializer[Cached[Long]] =

new CachedSerializer(self.Long.Simple, CompactProtocolFactory)

}

object SeqLong {

val Simple: Serializer[Seq[Long]] = new SeqSerializer(self.Long.Simple, 8)

}

object CachedSeqLong {

val Compact: Serializer[Cached[Seq[Long]]] =

new CachedSerializer(self.SeqLong.Simple, CompactProtocolFactory)

}

object Int {

val Simple = Transformer[Int, Array[Byte]](Ints.toByteArray, Ints.fromByteArray)

}

object CachedInt {

val Compact: Serializer[Cached[Int]] =

new CachedSerializer(self.Int.Simple, CompactProtocolFactory)

}

object SeqInt {

val Simple: Serializer[Seq[Int]] = new SeqSerializer(self.Int.Simple, 4)

}

object CachedSeqInt {

val Compact: Serializer[Cached[Seq[Int]]] =

new CachedSerializer(self.SeqInt.Simple, CompactProtocolFactory)

}

object String {

val Utf8: Serializer[String] = Transformer.Utf8ToBytes

}

object CachedString {

val Compact: Serializer[Cached[String]] =

new CachedSerializer(self.String.Utf8, CompactProtocolFactory)

}

object SeqString {

val Utf8: Serializer[Seq[String]] = new SeqSerializer(self.String.Utf8)

}

object CachedSeqString {

val Compact: Serializer[Cached[Seq[String]]] =

new CachedSerializer(self.SeqString.Utf8, CompactProtocolFactory)

}

/\*\*

\* We take care not to alter the buffer so that this conversion can

\* safely be used multiple times with the same buffer, and that

\* other threads cannot view other states of the buffer.

\*/

private[this] def byteBufferToArray(b: ByteBuffer): Array[Byte] = {

val a = new Array[Byte](b.remaining)

b.duplicate.get(a)

a

}

/\*\*

\* Convert between a ByteBuffer and an Array of bytes. The

\* conversion to Array[Byte] makes a copy of the data, while the

\* reverse conversion just wraps the array.

\*/

val ArrayByteBuffer: Transformer[Array[Byte], ByteBuffer] =

Transformer(ByteBuffer.wrap(\_: Array[Byte]), byteBufferToArray)

val ArrayByteBuf: Transformer[Array[Byte], Buf] =

Transformer(Buf.ByteArray.Shared.apply, Buf.ByteArray.Shared.extract)

/\*\*

\* Isomorphism between Time and Long. The Long represents the number

\* of nanoseconds since the epoch.

\*/

val TimeNanos: Transformer[UtilTime, Long] =

Transformer.pure[UtilTime, Long](\_.inNanoseconds, UtilTime.fromNanoseconds)

/\*\*

\* Transformer from Time to Array[Byte] always succeeds. The inverse

\* transform throws BufferUnderflowException if the buffer is less

\* than eight bytes in length. If it is greater than eight bytes,

\* the later bytes are discarded.

\*/

// This is lazy because if it is not, it may be initialized before

// Long.Simple. In that case, Long.Simple will be null at

// initialization time, and will be captured here. Unfortunately,

// this is dependent on the order of class initialization, which may

// vary between runs of a program.

lazy val Time: Serializer[UtilTime] = TimeNanos andThen Long.Simple

}

/\*\*

\* A Serializer for Thrift structs generated by Scrooge.

\*

\* @param codec used to encode and decode structs for a given protocol

\* @param protocolFactory defines the serialization protocol to be used

\*/

class ThriftSerializer[T <: ThriftStruct](

val codec: ThriftStructCodec[T],

val protocolFactory: TProtocolFactory)

extends Serializer[T]

with ThriftStructSerializer[T] {

override def to(obj: T): Try[Array[Byte]] = Try(toBytes(obj))

override def from(bytes: Array[Byte]): Try[T] = Try(fromBytes(bytes))

}

/\*\*

\* A Serializer for Thrift structs generated by the Apache code generator.

\*

\* @param tFactory a factory for Thrift-defined objects of type T. Objects

\* yielded by the factory are read into and returned during

\* deserialization.

\*

\* @param protocolFactory defines the serialization protocol to be used

\*/

class TBaseSerializer[T <: TBase[\_, \_]](tFactory: () => T, protocolFactory: TProtocolFactory)

extends Serializer[T] {

override def to(obj: T): Try[Array[Byte]] = Try {

val baos = new ByteArrayOutputStream

obj.write(protocolFactory.getProtocol(new TIOStreamTransport(baos)))

baos.toByteArray

}

override def from(bytes: Array[Byte]): Try[T] = Try {

val obj = tFactory()

val stream = new ByteArrayInputStream(bytes)

obj.read(protocolFactory.getProtocol(new TIOStreamTransport(stream)))

obj

}

}

object CachedSerializer {

def binary[T](valueSerializer: Serializer[T]): CachedSerializer[T] =

new CachedSerializer(valueSerializer, Protocols.binaryFactory())

def compact[T](valueSerializer: Serializer[T]): CachedSerializer[T] =

new CachedSerializer(valueSerializer, new TCompactProtocol.Factory)

}

/\*\*

\* A Serializer of Cached object.

\*

\* @param valueSerializer an underlying serializer of the values to be cached.

\* @param protocolFactory defines the serialization protocol to be used

\*/

class CachedSerializer[T](valueSerializer: Serializer[T], protocolFactory: TProtocolFactory)

extends Serializer[Cached[T]] {

private[this] val underlying = new ThriftSerializer(CachedValue, protocolFactory)

override def to(cached: Cached[T]): Try[Array[Byte]] =

underlying.to(cached.toCachedValue(valueSerializer))

private[this] val asCached: CachedValue => Cached[T] =

t => Cached(t, valueSerializer)

override def from(bytes: Array[Byte]): Try[Cached[T]] =

underlying.from(bytes).map(asCached)

}