package com.twitter.servo.util

import com.google.common.base.Charsets

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

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

import com.twitter.util.{Future, Return, Try, Throw}

import java.nio.{ByteBuffer, CharBuffer}

import java.nio.charset.{Charset, CharsetEncoder, CharsetDecoder}

/\*\*

\* Transformer is a (possibly partial) bidirectional conversion

\* between values of two types. It is particularly useful for

\* serializing values for storage and reading them back out (see

\* com.twitter.servo.cache.Serializer).

\*

\* In some implementations, the conversion may lose data (for example

\* when used for storage in a cache). In general, any data that passes

\* through a conversion should be preserved if the data is converted

\* back. There is code to make it easy to check that your Transformer

\* instance has this property in

\* com.twitter.servo.util.TransformerLawSpec.

\*

\* Transformers should take care not to mutate their inputs when

\* converting in either direction, in order to ensure that concurrent

\* transformations of the same input yield the same result.

\*

\* Transformer forms a category with `andThen` and `identity`.

\*/

trait Transformer[A, B] { self =>

def to(a: A): Try[B]

def from(b: B): Try[A]

@deprecated("Use Future.const(transformer.to(x))", "2.0.1")

def asyncTo(a: A): Future[B] = Future.const(to(a))

@deprecated("Use Future.const(transformer.from(x))", "2.0.1")

def asyncFrom(b: B): Future[A] = Future.const(from(b))

/\*\*

\* Compose this transformer with another. As long as both

\* transformers follow the stated laws, the composed transformer

\* will follow them.

\*/

def andThen[C](t: Transformer[B, C]): Transformer[A, C] =

new Transformer[A, C] {

override def to(a: A) = self.to(a) andThen t.to

override def from(c: C) = t.from(c) andThen self.from

}

/\*\*

\* Reverse the direction of this transformer.

\*

\* Law: t.flip.flip == t

\*/

lazy val flip: Transformer[B, A] =

new Transformer[B, A] {

override lazy val flip = self

override def to(b: B) = self.from(b)

override def from(a: A) = self.to(a)

}

}

object Transformer {

/\*\*

\* Create a new Transformer from the supplied functions, catching

\* exceptions and converting them to failures.

\*/

def apply[A, B](tTo: A => B, tFrom: B => A): Transformer[A, B] =

new Transformer[A, B] {

override def to(a: A): Try[B] = Try { tTo(a) }

override def from(b: B): Try[A] = Try { tFrom(b) }

}

def identity[A]: Transformer[A, A] = pure[A, A](a => a, a => a)

/\*\*

\* Lift a pair of (total) conversion functions to a Transformer. The

\* caller is responsible for ensuring that the resulting transformer

\* follows the laws for Transformers.

\*/

def pure[A, B](pureTo: A => B, pureFrom: B => A): Transformer[A, B] =

new Transformer[A, B] {

override def to(a: A): Try[B] = Return(pureTo(a))

override def from(b: B): Try[A] = Return(pureFrom(b))

}

/\*\*

\* Lift a transformer to a transformer on optional values.

\*

\* None bypasses the underlying conversion (as it must, since there

\* is no value to transform).

\*/

def optional[A, B](underlying: Transformer[A, B]): Transformer[Option[A], Option[B]] =

new Transformer[Option[A], Option[B]] {

override def to(optA: Option[A]) = optA match {

case None => Return.None

case Some(a) => underlying.to(a) map { Some(\_) }

}

override def from(optB: Option[B]) = optB match {

case None => Return.None

case Some(b) => underlying.from(b) map { Some(\_) }

}

}

//////////////////////////////////////////////////

// Transformers for accessing/generating fields of a Map.

//

// These transformers are useful for serializing/deserializing to

// storage that stores Maps, for example Hamsa.

/\*\*

\* Thrown by `requiredField` when the field is not present.

\*/

case class MissingRequiredField[K](k: K) extends RuntimeException

/\*\*

\* Get a value from the map, yielding MissingRequiredField when the

\* value is not present in the map.

\*

\* The inverse transform yields a Map containing only the one value.

\*/

def requiredField[K, V](k: K): Transformer[Map[K, V], V] =

new Transformer[Map[K, V], V] {

override def to(m: Map[K, V]) =

m get k match {

case Some(v) => Return(v)

case None => Throw(MissingRequiredField(k))

}

override def from(v: V) = Return(Map(k -> v))

}

/\*\*

\* Attempt to get a field from a Map, yielding None if the value is

\* not present.

\*

\* The inverse transform will put the value in a Map if it is Some,

\* and omit it if it is None.

\*/

def optionalField[K, V](k: K): Transformer[Map[K, V], Option[V]] =

pure[Map[K, V], Option[V]](\_.get(k), \_.map { k -> \_ }.toMap)

/\*\*

\* Transforms an Option[T] to a T, using a default value for None.

\*

\* Note that the default value will be converted back to None by

\* .from (.from will never return Some(default)).

\*/

def default[T](value: T): Transformer[Option[T], T] =

pure[Option[T], T](\_ getOrElse value, t => if (t == value) None else Some(t))

/\*\*

\* Transforms `Long`s to big-endian byte arrays.

\*/

lazy val LongToBigEndian: Transformer[Long, Array[Byte]] =

new Transformer[Long, Array[Byte]] {

def to(a: Long) = Try(Longs.toByteArray(a))

def from(b: Array[Byte]) = Try(Longs.fromByteArray(b))

}

/\*\*

\* Transforms `Int`s to big-endian byte arrays.

\*/

lazy val IntToBigEndian: Transformer[Int, Array[Byte]] =

new Transformer[Int, Array[Byte]] {

def to(a: Int) = Try(Ints.toByteArray(a))

def from(b: Array[Byte]) = Try(Ints.fromByteArray(b))

}

/\*\*

\* Transforms UTF8-encoded strings to byte arrays.

\*/

lazy val Utf8ToBytes: Transformer[String, Array[Byte]] =

stringToBytes(Charsets.UTF\_8)

/\*\*

\* Transforms strings, encoded in a given character set, to byte arrays.

\*/

private[util] def stringToBytes(charset: Charset): Transformer[String, Array[Byte]] =

new Transformer[String, Array[Byte]] {

private[this] val charsetEncoder = new ThreadLocal[CharsetEncoder]() {

protected override def initialValue() = charset.newEncoder

}

private[this] val charsetDecoder = new ThreadLocal[CharsetDecoder]() {

protected override def initialValue() = charset.newDecoder

}

override def to(str: String): Try[Array[Byte]] = Try {

// We can't just use `String.getBytes("UTF-8")` here because it will

// silently replace UTF-16 surrogate characters, which will cause

// CharsetEncoder to throw exceptions.

val bytes = charsetEncoder.get.encode(CharBuffer.wrap(str))

bytes.array.slice(bytes.position, bytes.limit)

}

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

charsetDecoder.get.decode(ByteBuffer.wrap(bytes)).toString

}

}

/\*\*

\* Transforms a ThriftStruct to a byte-array using Thrift's TBinaryProtocol.

\*/

def thriftStructToBytes[T <: ThriftStruct](c: ThriftStructCodec[T]): Transformer[T, Array[Byte]] =

new Transformer[T, Array[Byte]] {

private[this] val ser = BinaryThriftStructSerializer(c)

def to(a: T) = Try(ser.toBytes(a))

def from(b: Array[Byte]) = Try(ser.fromBytes(b))

}

}

/\*\*

\* transforms an Option[T] to a T, using a default value for None

\*/

@deprecated("Use Transformer.default", "2.0.1")

class OptionToTypeTransformer[T](default: T) extends Transformer[Option[T], T] {

override def to(b: Option[T]): Try[T] = Return(b.getOrElse(default))

override def from(a: T): Try[Option[T]] = a match {

case `default` => Return.None

case \_ => Return(Some(a))

}

}